DAY - 4

Detailed Report on How I Created these Components:

This report explains the development process of the Four major React components: **Product Detail Client**, **Comment Box**, **Quantity Control** and **Add to Cart** Functionality. The goal of these components is to create an interactive product detail page where users can view product details, leave comments, and adjust the product quantity.

---

1. ProductDetailClient Component

Purpose:

The `ProductDetailClient` component is responsible for displaying the product details, such as images, description, price, availability, and additional product information. It also allows users to add the product to their cart and change the quantity.

Steps to Create:

1. **Component Setup**:
    - The component is designed as a **client-side** React component using Next.js. This is indicated by the `"use client"` directive at the top of the file.
    - **Props**: The component receives two props:
        - `product`: An object representing the details of the product (e.g., name, description, image, price, availability).
        - `slug`: A string representing the URL slug of the product. This is used to fetch additional information (like product tabs) for the product.
2. **Banner Section**:
    - A banner is placed at the top of the page, with a background image and a title indicating the product details page.
    - The banner has two `Link` elements for navigation: one to the homepage and one to the product details page.

3. **Product Image Gallery**:
    - A set of thumbnail images is displayed on the left side, showing multiple views of the product.
    - The main product image is displayed prominently in the center.

4. **Product Information Section**:
    - Displays the product's availability (in stock or out of stock) with a button showing the status.
    - Displays the product name, description, and price.
    - The price is displayed with two decimal places for better presentation.
    - An image of the product's rating (static for now) is shown below the price.
    - The product's category and tags are listed.

5. **Social Media Share Buttons**:
    - Social media icons for Facebook, Instagram, Twitter, and YouTube are included to allow users to share the product.
    - Each social media platform is linked via `Link` components.

6. **Quantity Control and Add to Cart**:
    - The `QuantityControl` component (which manages product quantity) is used to allow users to adjust the quantity of the product.
    - The `AddToCart` component allows users to add the product with the selected quantity to their cart.
    -
7. **Tabs Section**:
    - A `TabComponent` is included to display more product-related information like specifications or user reviews, with the `slug` passed as a prop to fetch dynamic data.

---

2. CommentBox Component

Purpose:

The `CommentBox` component allows users to post comments about the product. The comments are saved to `localStorage`, ensuring persistence across page reloads. It also shows all previous comments.

Steps to Create:

1. **Component Setup**:
   - The `CommentBox` component is created using React functional components with hooks (`useState` and `useEffect`).
   - **State Management**: It uses state variables to manage:
     - `comment`: The current comment being typed by the user.
     - `comments`: An array of all comments.
     - `name`: The name of the user posting the comment.

2. **Loading Comments from `localStorage`**:
   - On the initial render (`useEffect` with an empty dependency array), it checks `localStorage` for any stored comments.
   - If comments exist, they are parsed and loaded into the `comments` state.

3. **Adding a Comment**:
   - The `addComments` function is triggered when the user clicks the "Add Comment" button.
   - If the comment is not empty, the comment is saved to the `comments` state and also stored in `localStorage`.

4. **Rendering Comments**:
   - The component renders all the saved comments in a list.
   - Each comment is displayed with the user's name (if provided) or "Anonymous" if no name is entered.

5. **UI Elements**:
   - An input field for the user's name.

- A `textarea` for writing the comment.
- A button to submit the comment, which triggers the `addComments` function.
- A list of all comments displayed below the input fields, with some styling for visual appeal.

---

3. QuantityControl Component

Purpose:

The `QuantityControl` component allows the user to increase or decrease the quantity of the product they want to purchase. It ensures that the quantity stays within a set range (1 to 50).

Steps to Create:

1. **Component Setup**:
   - The component is designed using the `useState` hook to manage the quantity of the product.
   - The initial quantity is set to `1` and stored in the `quantity` state.

2. **Increment and Decrement Logic**:
   - The `increment` function increases the quantity by 1, but only if the current quantity is less than the `maxQuantity` (set to 50).
   - The `decrement` function decreases the quantity by 1, but ensures the quantity doesn't go below 1.

3. **UI Elements**:
   - Two buttons are created to decrease and increase the quantity. The buttons use the `onClick` event to call the `decrement` and `increment` functions.
   - The current quantity is displayed between the two buttons, styled with Tailwind CSS classes for spacing, padding, and borders.

4. **UI Feedback**:
    - The buttons and the displayed quantity are styled to be visually appealing and functional. The buttons change the quantity dynamically in real-time.

---

1. `AddToCart` Component

Purpose:

The `AddToCart` component is responsible for allowing users to add products to their cart. Once the product is added, a confirmation message is displayed.

Key Features:

- **Prop**: The `product` prop is passed down to the `AddToCart` component, which contains all the necessary information about the item (name, price, description, image, etc.).
- **State Management**: A local state `addedToCart` is used to toggle a message indicating that the item has been successfully added to the cart.
- **Cart Integration**: The `addToCart` function from the `CartContext` is called to add the product to the global cart state.
- **Timeout for Message Reset**: After displaying the "Item added to cart!" message for 2 seconds, it resets to hide the message.

Code Explanation:

```
const AddToCart: React.FC<AddToCartProps> = ({ product }) => {
  const [addedToCart, setAddedToCart] = useState(false);
  const { addToCart } = useCart(); // Get the function to add
items to the cart

  const handleAddToCart = () => {
    addToCart(product); // Add the product to the cart state
    setAddedToCart(true); // Show confirmation message
    setTimeout(() => setAddedToCart(false), 2000); // Hide
message after 2 seconds
  };

  return (
```

```
    <div>
      <button
        onClick={handleAddToCart}
        className="w-36 text-sm p-3 text-white bg-[#FF9F0D]
hover:bg-red-700"
      >
        Add to Cart
      </button>
      {addedToCart && (
        <p className="text-green-500 mt-4">Item added to
cart!</p>
      )}
    </div>
  );
};
```

- The button triggers the `handleAddToCart` function.
- After adding the item to the cart, it shows a success message for 2 seconds, then resets it.

---

2. `CartContext` and `CartProvider`

Purpose:

The `CartContext` manages the global state of the cart across the application. It provides methods to add items, remove items, and update quantities in the cart. The `CartProvider` wraps the application to make the cart context available globally.

Key Features:

- **State Persistence**: The cart items are saved in `localStorage` to persist even after page reloads.
- **`addToCart` Method**: Adds a new item to the cart or increments the quantity if the item already exists in the cart.
- **`removeFromCart` Method**: Removes an item from the cart.
- **`updateQuantity` Method**: Allows updating the quantity of a cart item.
- **`CartContext.Provider`**: Provides the cart state and methods to the rest of the application.

Code Explanation:

```
export const CartProvider: React.FC<{ children: React.ReactNode
}> = ({ children }) => {
  const [cartItems, setCartItems] = useState<CartItem[]>(() => {
    const savedCart = localStorage.getItem("cartItems");
    return savedCart ? JSON.parse(savedCart) : [];
  });

  const saveToLocalStorage = (items: CartItem[]) => {
    localStorage.setItem("cartItems", JSON.stringify(items));
  };

  const addToCart = (product: FoodItem) => {
    setCartItems((prevItems) => {
      const existingItem = prevItems.find(item => item.name ===
product.name);
      if (existingItem) {
        existingItem.quantity += 1;
        saveToLocalStorage(prevItems);
        return [...prevItems];
      } else {
        const newItem = { ...product, quantity: 1 };
        saveToLocalStorage([...prevItems, newItem]);
        return [...prevItems, newItem];
      }
    });
  };

  const removeFromCart = (product: FoodItem) => {
    setCartItems((prevItems) => {
      const updatedItems = prevItems.filter(item => item.name
!== product.name);
      saveToLocalStorage(updatedItems);
      return updatedItems;
    });
  };

  const updateQuantity = (product: FoodItem, quantity: number)
=> {
    setCartItems((prevItems) => {
      const updatedItems = prevItems.map((item) =>
        item.name === product.name ? { ...item, quantity } :
item
      );
      saveToLocalStorage(updatedItems);
      return updatedItems;
```

```
    });
  };

  return (
    <CartContext.Provider value={{ cartItems, addToCart,
removeFromCart, updateQuantity }}>
      {children}
    </CartContext.Provider>
  );
};
```

- **State Initialization**: The `cartItems` are initialized by checking `localStorage`. If there are saved items, they are loaded; otherwise, an empty array is used.

- **State Update**: The `setCartItems` function updates the state of the cart and also stores the updated cart items in `localStorage`.

---

3. `Cart` Page (Cart Viewing and Management)

Purpose:

The `Cart` page displays all the products that the user has added to the cart. It allows users to:

- View cart items, including their names, prices, descriptions, and quantities.
- Update the quantity of items in the cart.
- Remove items from the cart.
- Proceed to checkout.

Key Features:

- **Displaying Cart Items**: The cart items are fetched using the `useCart` hook and mapped to display each item.
- **Quantity Management**: Buttons for incrementing and decrementing the quantity of each cart item.

- **Total Calculation**: A total price is displayed based on the quantity and price of all items in the cart.

Code Explanation:

```
const Cart = () => {
  const { cartItems, removeFromCart, updateQuantity } =
useCart(); // Get the cart state and methods

  if (cartItems.length === 0) {
    return (
      <div>Your cart is empty. <Link href="/shop">Go back to the
menu</Link></div>
    );
  }

  return (
    <div>
      <h2>Your Cart</h2>
      {cartItems.map((item, index) => (
        <div key={index}>
          <div>{item.name}</div>
          <div>{item.description}</div>
          <div>{item.price}</div>
          <div>
            <button onClick={() => updateQuantity(item,
item.quantity - 1)}>-</button>
            {item.quantity}
            <button onClick={() => updateQuantity(item,
item.quantity + 1)}>+</button>
          </div>
          <button onClick={() =>
removeFromCart(item)}>Remove</button>
        </div>
      ))}
      <div>
        <h3>Total: ${cartItems.reduce((total, item) => total +
item.price * item.quantity, 0)}</h3>
        <Link href="/checkout">
          <button>Proceed to Checkout</button>
        </Link>
      </div>
    </div>
  );
};
```

- **Cart Items**: We map through `cartItems` to display each item and allow for updates to quantity and removal.
- **Total**: The total price is dynamically calculated based on the cart items.

---

4. `CartProvider` in `layout.tsx`

Purpose:

Wrapping the `CartProvider` around the application ensures that all components within the application can access the cart state via the context.

Code:

```
        <CartProvider>
          <Header />
          {children}
          <Footer />
        </CartProvider>
```

- **CartContext** is provided at the root level, ensuring it is accessible throughout the app.

---

5. Using `AddToCart` and `Cart` Components in the Product Page

Implementation:

- **`AddToCart`** is used within the product detail page, allowing users to add products to the cart.
- **`View Cart`** link takes the user to the cart page where they can view and modify cart items.

Code:

```
<AddToCart product={product} />
<Link href="/cart" className="mt-0 border border-[#FF9F0D] px-8
py-[4px] text-center text-yellow-500">
  View Cart
```

```
</Link>
```

This implementation provides a comprehensive cart system that allows users to add, view, and manage products in their cart, with state persistence across page reloads using `localStorage`. By leveraging React context, hooks, and state management, we create a dynamic and responsive user experience.

General Approach and Technologies Used:

1. **React (with hooks)**:
    - The components are built using React functional components with hooks (`useState` and `useEffect`) for state management and side effects.
    - `useEffect` is used to load comments from `localStorage` on the initial render.

2. **Next.js**:
    - The `ProductDetailClient` component uses Next.js' `Link` component for navigation between pages. This enables client-side routing without page reloads.

3. **Tailwind CSS**:
    - Tailwind CSS is used for utility-first styling of the components, ensuring that the design is responsive and visually consistent across different screen sizes.
    - Tailwind classes are applied for layout, spacing, borders, text styling, and hover effects.

4. **State Persistence**:

- The comments are stored in `localStorage` to persist across page reloads. This ensures that the comments remain visible even after refreshing the page.

5. **Client-Side Rendering**:
   - All three components are designed to render client-side, which is indicated by the `"use client"` directive at the top of the file.
   - This approach ensures interactivity and a dynamic user experience.

Conclusion:

The development of these components involved creating a dynamic and interactive product detail page that integrates a cart system, comment functionality, quantity control and Add to Cart Functionality. By utilizing React hooks, Next.js navigation, and Tailwind CSS, the components provide a modern, responsive user interface. The use of `localStorage` for comments allows data persistence, ensuring users can interact with the product page even after reloading. Further improvements could include backend integration for comments and ratings and advanced features like authentication for users to manage their reviews.
