React Authentication and State Management Guide

Amr A Khllaf

July 31, 2025

Contents

1 Important Note

- **Any change to the UI requires updating the state. Whenever the UI needs to reflect new data or user interactions, a corresponding state change is triggered, which causes React to re-render the affected components.**

2 What Is a Token?

- A token is a piece of data representing a user's identity, used to **authenticate and authorize access** to resources in a web application. In React and web development, tokens are commonly used with APIs to ensure secure client-server communication.
- Tokens come in various formats, such as JSON Web Tokens (JWT), OAuth tokens, or session tokens. They typically contain user information and permissions, and are often signed to prevent tampering.
- Tokens are usually stored in the browser's local storage or cookies, allowing the application to maintain the user's session across different pages and interactions.
- **When a user logs in, the server generates a token and sends it to the client. The client then includes this token in the headers of subsequent API requests to authenticate itself.**

2.1 Are Tokens Encrypted?

- Tokens can be encrypted, but this depends on the implementation and token type. For example, JSON Web Tokens (JWT) can be signed and optionally encrypted to ensure integrity and confidentiality.
- **Signing a token ensures it hasn't been altered, while encryption protects its contents from unauthorized access.**
- Many tokens are signed but not encrypted by default. If sensitive information is included, encrypting the token before sending it to the client is recommended.
- Always use HTTPS to transmit tokens securely, store them safely, and implement proper expiration and revocation mechanisms.

2.2 What is JWT?

- **JWT (JSON Web Token)** is a compact, URL-safe way to represent claims between two parties. JWTs are commonly used for authentication and information exchange in web applications.
- A JWT consists of three parts: a header, a payload, and a signature.
  - **Header**: Metadata about the token, such as type (JWT) and signing algorithm (e.g., HMAC SHA256).
  - **Payload**: Claims or statements about an entity (typically the user) and additional data.
  - **Signature**: Created by combining the encoded header and payload with a secret key using the specified algorithm, ensuring the token hasn't been tampered with.
- JWTs are often used in stateless authentication systems, where the server does not store session information. The token itself contains all necessary information to authenticate the user.

- JWTs can be stored in local storage, session storage, or cookies, and are typically sent in the `Authorization` header of HTTP requests.

2.2.1 JWT Tool

- Use JWT.io to decode, verify, and generate JWTs. This tool lets you inspect a JWT's header, payload, and signature, and verify its integrity.
- To generate JWTs in Node.js:

```
npm install jsonwebtoken
```

- **Comparison: Encrypted vs. Unencrypted JWT**

	Unencrypted JWT	Encrypted JWT
Contents	Readable by anyone	Only readable by parties with the decryption key
Integrity	Signed for integrity	Signed and encrypted for integrity and confidentiality
Use Case	Non-sensitive data	Sensitive data requiring confidentiality
Storage	Local storage	Local storage or cookies with encryption
Security	Vulnerable to interception	More secure against interception and tampering

- **JWTs are not encrypted by default**; they are signed for integrity. Encrypt JWTs if they contain sensitive information.
- **Example JWT Structure:**

```
{
  "header": {
    "alg": "HS256",
    "typ": "JWT"
  },
  "payload": {
    "sub": "1234567890",
    "name": "John Doe",
    "iat": 1516239022
  },
  "signature": "HMACSHA256(base64UrlEncode(header) + '.' +
  ↪  base64UrlEncode(payload), secret)"
}
```

- The header specifies the algorithm and type. The payload contains user info. The signature ensures integrity.
- **Role of the Signature in JWT:**
  - **Integrity**: Ensures the token hasn't been altered.
  - **Authentication**: Verifies the token was issued by a trusted source.
  - **Non-repudiation**: Proves the issuer created the token.
- To verify a JWT, recalculate the signature using the same algorithm and secret key. If it matches, the token is valid.

2.3 Benefits of Using JWT

- **Stateless Authentication**: No need for server-side session storage; all info is in the token.
- **Cross-Domain Authentication**: Suitable for single sign-on (SSO) scenarios.
- **Compact and URL-Safe**: Easily transmitted in URLs, headers, or cookies.
- **Self-Contained**: Contains all necessary user and permission info.
- **Interoperability**: Based on open standards, compatible with many languages and platforms.

2.4 How JWT Works in Our Application

- **Fetching Profile Data**: The client includes the JWT in the `Authorization` header. The server validates the token and retrieves user data.
- **Authenticated User Flow**: After login, the server generates a JWT and returns it to the client, which stores it and attaches it to future requests.

---

3 Parent Component & Children as Props

- In React, a parent component can pass data to child components using **props**. Props are read-only and enable communication from parent to child.
- **Example:**

```
import React from "react";
function ParentComponent() {
  const name = "John Doe";
  return (
    <div>
      <ChildComponent name={name} />
    </div>
  );
}
function ChildComponent({ name }) {
  return <h1>Hello, {name}!</h1>;
}
```

- **Props vs. State**:
  - **Props**: Passed from parent to child, immutable in the child.
  - **State**: Local to a component, can be changed with `setState`.

3.1 Inline Style for Components

- Use the `style` attribute with a JavaScript object for inline styles.

```
import React from "react";
function StyledComponent() {
  const style = {
    color: "blue",
```

```
    fontSize: "20px",
    backgroundColor: "lightgray",
    padding: "10px",
    borderRadius: "5px",
  };
  return <div style={style}>This is a styled component!</div>;
}
export default StyledComponent;
```

4 Rendering Multiple Children in a Parent

- Render multiple children:

```
import React from "react";
function ParentComponent() {
  return (
    <div>
      <ChildComponent1 />
      <ChildComponent2 />
    </div>
  );
}
function ChildComponent1() {
  return <h1>This is Child Component 1</h1>;
}
function ChildComponent2() {
  return <h1>This is Child Component 2</h1>;
}
```

- Conditionally render children based on a prop:

```
import React from "react";
function ParentComponent({ showChild1 }) {
  return <div>{showChild1 ? <ChildComponent1 /> : <ChildComponent2
  ↪  />}</div>;
}
```

4.1 Passing Children as Props in React

- Anything between a component's opening and closing tags is passed as the `children` prop.

```
function Parent(props) {
  return <div>{props.children}</div>;
}

// Usage:
<Parent>
  <h1>Hello from Child!</h1>
  <p>This is another child element.</p>
</Parent>;
```

5 Parent and Children Components Example

- **App.jsx**

```jsx
import Parent from "./Components/Parent/Parent";
function App() {
  return (
    <>
      <Parent flag={true} />
      <Parent flag={false} />
    </>
  );
}
export default App;
```

- **Parent.jsx**

```jsx
import Child1 from "../Child1/Child1";
import Child2 from "../Child2/Child2";
function Parent({ flag }) {
  return <>{flag ? <Child1 /> : <Child2 />}</>;
}
export default Parent;
```

- **Child1.jsx**

```jsx
function Child1() {
  return (
    <div
      style={{
        color: "blue",
        backgroundColor: "lightblue",
        padding: "20px",
        textAlign: "center",
      }}
    >
      Child1
    </div>
  );
}
export default Child1;
```

- **Child2.jsx**

```jsx
function Child2() {
  const style = {
    color: "black",
    backgroundColor: "lightcoral",
    padding: "20px",
  };
  return <div style={style}>Child2</div>;
```

```
}
export default Child2;
```

- This demonstrates passing props, conditional rendering, and inline styles.

5.1 Using Component Name in Any Tag

- You can use a component as a self-closing tag or with children. Anything between the tags is passed as `children`.
- **App.jsx**

```
import Parent from "./Components/Parent/Parent";
function App() {
  return (
    <>
      <Parent name="Amr" age={20}>
        <h2>Hello ya Naymeen</h2>
        <h2>Hello ya Naymeen</h2>
        <h2>Hello ya Naymeen</h2>
      </Parent>
    </>
  );
}
export default App;
```

- **Parent.jsx**

```
function Parent({ children, name, age }) {
  return (
    <div
      style={{
        border: "1px solid black",
        padding: "10px",
        marginTop: "10px",
      }}
    >
      <h3>Parent Component</h3>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
      <div
        style={{
          border: "1px solid red",
          padding: "5px",
          marginTop: "5px",
        }}
      >
        <h4>Children Props:</h4>
        {children}
        {children[0]}
        {children[1]}
```

```
      </div>
    </div>
  );
}
export default Parent;
```

- This pattern is useful for reusable, flexible components.

---

6 State Management and Context

6.1 State Management

- **State management enables sharing data across multiple components without manually passing props through each level. This simplifies code and improves maintainability, especially in large apps.**
- Prop drilling occurs when you pass data through many layers, even if some components don't need it, making code harder to maintain.

6.1.1 React Context to Solve Prop Drilling

- **Context** acts as a centralized store for shared data. Wrap components needing access with a `Context.Provider`.

6.2 `createContext` Function

- Use `createContext` to create a context object, typically with a default value.
- The context object provides `Provider` and `Consumer` components.

6.2.1 Example: Creating Context

```
import React, { createContext, useState } from "react";
export const MyContext = createContext();
export const MyProvider = ({ children }) => {
  const [state, setState] = useState("Hello, World!");
  return (
    <MyContext.Provider value={{ state, setState }}>
      {children}
    </MyContext.Provider>
  );
};
```

6.2.2 Using the Provider

```
import { MyProvider } from "./Context/MyContext";
function App() {
  return (
    <MyProvider>
      <ChildComponent />
```

```
    </MyProvider>
  );
}
```

6.2.3 Consuming Context

```
import React, { useContext } from "react";
import { MyContext } from "../Context/MyContext";
function ChildComponent() {
  const { state, setState } = useContext(MyContext);
  return (
    <div>
      <h1>{state}</h1>
      <button onClick={() => setState("Hello from Child!")}>
        Update State
      </button>
    </div>
  );
}
```

---

7 Context Naming Convention

- Name context files with `Context` at the end, e.g., `AuthContext.js`.
- The provider is usually named `<ContextName>Provider`, and the context object as `<ContextName>Context`.

7.0.1 Example

```
import React, { createContext, useState } from "react";
export const AuthContext = createContext();
export const AuthProvider = ({ children }) => {
  const [user, setUser] = useState(null);
  return (
    <AuthContext.Provider value={{ user, setUser }}>
      {children}
    </AuthContext.Provider>
  );
};
```

7.0.2 Using Context in Components

```
import React, { useContext } from "react";
import { AuthContext } from "../Context/AuthContext";
function UserProfile() {
  const { user, setUser } = useContext(AuthContext);
  const handleLogin = () => {
    setUser({ name: "John Doe", email: "john.doe@example.com" });
  };
  return (
```

```
    <div>
      <h2>User Profile</h2>
      {user ? (
        <div>
          <p>Name: {user.name}</p>
          <p>Email: {user.email}</p>
        </div>
      ) : (
        <p>Please log in.</p>
      )}
      <button onClick={handleLogin}>Log In</button>
    </div>
  );
}
```

- Wrap components needing context with the provider in `App.jsx`:

```
import { AuthProvider } from "./Context/AuthContext";
function App() {
  return (
    <AuthProvider>
      <UserProfile />
    </AuthProvider>
  );
}
```

- ES6 shorthand: `{ user, setUser }` is equivalent to `{ user: user, setUser: setUser }`.

---

8 Managing User Token in E-Commerce App

8.1 Where is the Token Used?

- The token is used throughout the app: Navbar (to show login/logout), Profile page (fetch/update user data), Cart page (manage cart), and Login page (authenticate and store token).
- Example login function:

```
function loginUser(values) {
  axios
    .post("https://ecommerce.routemisr.com/api/v1/auth/signin",
    ↪  values)
    .then(function (x) {
      localStorage.setItem("userToken", x.data.token);
    });
}
```

- Retrieve the token with `localStorage.getItem("userToken")`.

---

9 Applying Context for Token Management

9.1 Steps to Implement AuthContext

1. **Create a `Context` folder** in `src`.
2. **Create `AuthContext.jsx`** inside `Context`.
3. **Set up the context:**

```
import React, { createContext, useState, useEffect } from "react";
export const AuthContext = createContext();

export function AuthProvider({ children }) {
  const [token, setToken] = useState(
    localStorage.getItem("userToken") || null
  );

  useEffect(() => {
    if (token) {
      localStorage.setItem("userToken", token);
    } else {
      localStorage.removeItem("userToken");
    }
  }, [token]);

  return (
    <AuthContext.Provider value={{ token, setToken }}>
      {children}
    </AuthContext.Provider>
  );
}
```

4. **Wrap your app with the provider:**

```
import { AuthProvider } from "./Context/AuthContext";
function App() {
  return (
    <AuthProvider>
      <RouterProvider router={router} />
    </AuthProvider>
  );
}
```

5. **Consume the context:**

```
import React, { useContext } from "react";
import { AuthContext } from "../Context/AuthContext";
function SomeComponent() {
  const { token, setToken } = useContext(AuthContext);
}
```

6. **Update the token on login/logout:**
   Call `setToken(newToken)` after login, and `setToken(null)` on logout.

7. **Persist authentication state:**
   Store the token in both React state and local storage for persistence.
8. **Best practices:**
   - Export both `AuthContext` and `AuthProvider`.
   - Remove the token from storage on logout.
   - Never store sensitive data directly in the token or local storage.

---

10 Updating UI Based on Token

- Use `useContext` to access the token and conditionally render UI elements.

```
import { AuthContext } from "../../Context/AuthContext";
function Navbar() {
  const { token } = useContext(AuthContext);
  return (
    <ul>
      {token ? (
        <li>
          <span>Logout</span>
        </li>
      ) : (
        <>
          <li>
            <NavLink to="/register">Register</NavLink>
          </li>
          <li>
            <NavLink to="/login">Login</NavLink>
          </li>
        </>
      )}
    </ul>
  );
}
```

---

11 Handling Token Persistence on Refresh

- Refreshing the page resets in-memory state, so always synchronize your token state with persistent storage (e.g., `localStorage`).
- On app initialization, read the token from storage:

```
const [token, setToken] = useState(localStorage.getItem("userToken")
↪  || null);
```

- Use a `useEffect` to update storage whenever the token changes.
- This ensures the user stays authenticated across reloads.

11.1 How to Detect a Page Refresh

- A refresh causes the component to re-mount. You can detect this in React using the `useEffect` hook with an empty dependency array, which runs once on mount (similar to `componentDidMount`).
- In AuthContext, you can check if the token exists in `localStorage` when the component mounts by using `useEffect`:

AuthContext.jsx File:

```jsx
import { createContext, useEffect, useState } from "react";

export const authContext = createContext();

export default function AuthContext({ children }) {
  // You can add state and functions here to manage authentication
  const [token, setToken] = useState(null);

  // We Will Use `useEffect` to check if the token exists in localStorage
  // and set it in the context when the component mounts.

  useEffect(() => {
    const storedUserToken = localStorage.getItem("userToken");
    // Check if the token exists in localStorage
    // If it does, set it in the context state
    if (storedUserToken != null) {
      logger("Token found in localStorage:", storedUserToken);
      setToken(storedUserToken);
    }
  }, []);

  return (
    <authContext.Provider value={{ token: token, setToken: setToken }}>
      {children}
    </authContext.Provider>
  );
}
```

Note:
When using a `setState` function inside a `useEffect` hook, be careful to avoid unnecessary state updates. Since `useEffect` runs after the initial render, setting state unconditionally can cause an **infinite re-render loop**. To prevent this, only call `setState` if the new value is different from the current state.

- We can Do it with another way making a shortcut to the `localStorage`:

```jsx
import { createContext, useEffect, useState } from "react";

export const authContext = createContext();
```

```
export default function AuthContext({ children }) {
  // You can add state and functions here to manage authentication
  const [token, setToken] = useState(null);

  // We Will Use `useEffect` to check if the token exists in localStorage
  // and set it in the context when the component mounts.

  useEffect(() => {
    // Check if the token exists in localStorage ( userToken or Null)
    const storedUserToken = localStorage.getItem("userToken");
  }, []);

  return (
    <authContext.Provider value={{ token: token, setToken: setToken }}>
      {children}
    </authContext.Provider>
  );
}
```

- This way, you can easily access the token from `localStorage` and set it in your context state when the component mounts. This ensures that your application can maintain the user's authentication state even after a page refresh.
- This approach allows you to manage the authentication state effectively, ensuring that the user remains logged in even after a page refresh, as long as the token is stored in `localStorage`.

---

12 User Logout

- To implement user logout, You need To Follow these steps:

1. **Create a logout function** that clears the token from both the context and `localStorage`.
2. **Update the UI** to reflect the logged-out state, such as showing a login button instead of user information.
3. **Redirect the user** to the login page or home page after logout, using useNavigate State and this return statement.

- return of the Navigate function is a promise, so you can use `then` to handle the redirection after the logout process is complete.
- And Navigate is a hook that allows you to programmatically navigate to different routes in your application.
- **Example Logout Function:**
- Navbar.jsx File:

```
<ul className="flex items-center gap-2.5 ">
  {token ? (
    <li>
      <span
```

```
          className="cursor-pointer hover:text-red-600 transition
          ↪  duration-400"
          onClick={handleLogout}
          // handleLogout is a function that clears the token and redirects
          ↪  the user to the login page.
        >
          Logout
        </span>
      </li>
    ) : (
      <>
        <li className="hover:text-green-600 transition duration-400">
          <NavLink to="/register">Register</NavLink>
        </li>
        <li className="hover:text-green-600 transition duration-400">
          <NavLink to="/login">Login</NavLink>
        </li>
      </>
    )}
</ul>
```

- Function to handle logout in Navbar.jsx:

```
const { token, setToken } = useContext(authContext);
const navigate = useNavigate();
function handleLogout() {
  // Clear the token from localStorage and context
  localStorage.removeItem("userToken");
  setToken(null); // To Update UI
  navigate("/login"); // Redirect to login page
}
```

---

13 Discussing How To Prevent User from Accessing Protected Routes

- To prevent users from accessing Protected routes without authentication, you can create a **ProtectedRoute** component that checks if the user is authenticated (i.e., if a valid token exists). If not, redirect them to the login page.
- This component can be used to wrap any route that requires authentication.

```
import React, { useContext } from "react";
import { Navigate } from "react-router-dom";
import { authContext } from "../Context/AuthContext";

const ProtectedRoute = ({ children }) => {
  const { token } = useContext(authContext);
  return token ? children : <Navigate to="/login" />;
};
export default ProtectedRoute;
```

```
- This component can be used to wrap any route that requires
↪  authentication.
```

13.0.1 Problem

- By default, if a user tries to access a protected route in our application, they can navigate to it directly—even if they are not authenticated. This can expose sensitive pages to unauthorized users.

13.0.2 Solution

- To prevent unauthorized access, we can implement a **ProtectedRoute** component. This component checks if the user is authenticated before allowing access to the protected route. If the user is not authenticated, they are redirected to the login page or another appropriate location.

---

14 Protected Route

- The **ProtectedRoute** component can be used to wrap any route that requires authentication.
- This component checks if the user is authenticated by verifying the presence of a valid token in the context. If the token exists, it renders the child components; otherwise, it redirects the user to the login page.
- **Example Usage:**

```jsx
import ProtectedRoute from "./Components/ProtectedRoute/ProtectedRoute";
import Cart from "./Components/Cart/Cart";
import { BrowserRouter as Router, Route, Routes } from "react-router-dom";

function App() {
  return (
    <Router>
      <Routes>
        <Route
          path="/cart"
          element={
            <ProtectedRoute>
              <Cart />
            </ProtectedRoute>
          }
        />
      </Routes>
    </Router>
  );
}
```

14.1 Navigate Component

- The `Navigate` component from `react-router-dom` is used to programmatically redirect users to a different route. It can be used in the `ProtectedRoute` component to redirect unauthenticated users to the login page.
- **Example:**

```
import { Navigate } from "react-router-dom";
const ProtectedRoute = ({ children }) => {
  const { token } = useContext(authContext);
  return token ? children : <Navigate to="/login" />;
};
```

14.2 Steps to Implement Protected Routes

1. **Create a `ProtectedRoute` component** that checks if the user is authenticated.
2. **Wrap protected routes** with the `ProtectedRoute` component in your main routing file (e.g., `App.jsx`).
3. **Check authentication** by verifying if a token exists in `localStorage`.
4. **Redirect unauthenticated users** to the login page using the `Navigate` component.
5. **Import and use `ProtectedRoute`** wherever authentication is required.

14.2.1 ProtectedRoute.jsx

```
import { Navigate } from "react-router-dom";

function ProtectedRoute({ children }) {
  const token = localStorage.getItem("userToken");
  if (!token) {
    return <Navigate to="/login" replace />;
  }
  return <>{children}</>;
}

export default ProtectedRoute;
```

- This component checks for a token in `localStorage`. If not found, it redirects to `/login`; otherwise, it renders the child components.

14.2.2 App.jsx

```
import React from "react";
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import Layout from "./Components/Layout/Layout";
import Register from "./Components/Register/Register";
import Login from "./Components/Login/Login";
import Notfound from "./Components/NotFound/Notfound";
import AuthContext from "./Context/AuthContext";
import Cart from "./Components/Cart/Cart";
import Brands from "./Components/Brands/Brands";
```

```
import Categories from "./Components/Categories/Categories";
import ProtectedRoute from "./Components/ProtectedRoute/ProtectedRoute";
import Products from "./Components/Products/Products";

function App() {
  const router = createBrowserRouter([
    {
      path: "",
      element: <Layout />,
      children: [
        { path: "register", element: <Register /> },
        { path: "login", element: <Login /> },
        {
          path: "cart",
          element: (
            <ProtectedRoute>
              <Cart />
            </ProtectedRoute>
          ),
        },
        {
          path: "brands",
          element: (
            <ProtectedRoute>
              <Brands />
            </ProtectedRoute>
          ),
        },
        {
          path: "categories",
          element: (
            <ProtectedRoute>
              <Categories />
            </ProtectedRoute>
          ),
        },
        {
          path: "products",
          element: (
            <ProtectedRoute>
              <Products />
            </ProtectedRoute>
          ),
        },
        { path: "*", element: <Notfound /> },
      ],
    },
  ]);
  return (
    <AuthContext>
```

```
      <RouterProvider router={router} />
    </AuthContext>
  );
}

export default App;
```

- Use `ProtectedRoute` to secure routes and ensure only authenticated users can access them.

---

15 Fetching Products and Displaying Them

- To fetch products from an API and display them in your React application, you can use the `useEffect` hook to make an API call when the component mounts. Store the fetched products in state using the `useState` hook.
- Review the Styling and Structure of the Products Component:
- Products.jsx File:

```
import axios from "axios";
import { useEffect, useState } from "react";
import { ThreeDots } from "react-loader-spinner";
import { motion } from "framer-motion";

function Products() {
  // This component will render the products
  // You can fetch products from an API or use static data
  const [allProducts, setAllProducts] = useState(null);
  async function getAllProducts() {
    const { data } = await axios.get(
      "https://ecommerce.routemisr.com/api/v1/products"
    );

    setAllProducts(data.data);
  }

  // Call the function to fetch products in componentMount or useEffect
  // If using functional components, you can use useEffect

  useEffect(() => {
    getAllProducts();
  }, []);

  return (
    <>
      <div className="container mx-auto">
        <div className="grid grid-cols-1 md:grid-cols-3 lg:grid-cols-6
        ↪  gap-6 p-4 ">
          {/* Products will be rendered here */}
```

```
        {allProducts ? (
          allProducts.map((product) => (
            <div
              key={product._id}
              className="product bg-white rounded-lg shadow-md p-4 flex
              ↪  flex-col items-center hover:shadow-lg transition-shadow
              ↪  duration-300"
            >
              <img
                src={product.imageCover}
                alt={product.title}
                className="w-full  object-cover rounded-md mb-4
                ↪  hover:scale-105 transition-transform duration-300"
              />
              <h6 className="text-sm text-gray-500 mb-1">
                {product.category.name}
              </h6>
              <h2 className="text-lg font-semibold mb-2">
                {product.title.split(" ").slice(0, 2).join(" ")}
              </h2>
              <div className="flex flex-col w-full">
                <div className="flex items-center justify-between">
                  <span
                    className={`${
                      product.priceAfterDiscount
                        ? "line-through text-red-500 text-sm"
                        : "text-green-600 "
                    }`}
                  >
                    {product.price} <span className="text-sm">EGP</span>
                  </span>
                  {/* Show rate only if no discount */}
                  {!product.priceAfterDiscount && (
                    <p className="flex items-center text-yellow-500
                    ↪  ml-2">
                      <i className="fa-solid fa-star mr-1"></i>
                      {product.ratingsAverage}
                    </p>
                  )}
                </div>
                {product.priceAfterDiscount && (
                  <div className="flex items-center justify-between
                  ↪  mt-1">
                    <div className="text-green-600 flex justify-between
                    ↪  w-full">
                      {/* Price After Discount */}
                      <div className="flex items-center">
                        {product.priceAfterDiscount}
                        <span className="text-sm ml-1">EGP</span>
                      </div>
```

```
                        {/* Rating  */}
                        <div className="flex items-center text-yellow-500">
                          <i className="fa-solid fa-star mr-1"></i>
                          {product.ratingsAverage}
                        </div>
                      </div>
                    </div>
                  )}
                </div>
              </div>
            ))
          ) : (
            <motion.div
              initial={{ opacity: 0 }}
              animate={{ opacity: 1 }}
              exit={{ opacity: 0 }}
              className="fixed inset-0 bg-white/50 backdrop-blur-sm z-50
              ↪  flex justify-center items-center"
            >
              <div className="flex flex-col items-center">
                <ThreeDots
                  visible={true}
                  height="80"
                  width="80"
                  color="#4fa94d"
                  radius="9"
                  ariaLabel="three-dots-loading"
                  wrapperStyle={{}}
                  wrapperClass=""
                />
                <p className="mt-4 text-lg text-gray-700 font-medium
                ↪  animate-pulse">
                  Loading products...
                </p>
              </div>
            </motion.div>
          )}
        </div>
      </div>
    </>
  );
}

export default Products;
```

15.1 Review Styling and Structure and Explanation

- The `Products` component fetches product data from an API and displays it in a grid layout. It uses the `useState` hook to manage the product data and the `useEffect` hook

to fetch the data when the component mounts.

- The component checks if `allProducts` is null (indicating data is still being fetched). If it is, a loading spinner is displayed using the `ThreeDots` component from `react-loader-spinner`.
- Once the data is fetched, it maps over the `allProducts` array and renders each product in a card format. Each product card includes an image, title, category, price, and rating.
- The product image is displayed with a hover effect that scales it slightly. The title is truncated to show only the first two words for better readability.
- The price is displayed with a strike through if there is a discount, and the discounted price is shown alongside the rating.
- The component uses Tailwind CSS classes for styling, ensuring a responsive design that adapts to different screen sizes.
- The overall structure is clean and easy to read, making use of modern React features and best practices.

---

16 React Slick For Home Slider

- **React Slick** is a popular carousel component for React applications, allowing you to create responsive sliders with various customization options.
- To use React Slick, you need to install it along with its CSS dependencies.
- You Can reed the DOCS from the site: React Slick Site

```
npm install react-slick slick-carousel
```

- After installation, you can import the necessary components and styles in your React component.
- Here's an example of how to implement a simple slider using React Slick:

```
import React from "react";
import Slider from "react-slick";
import "slick-carousel/slick/slick.css";
import "slick-carousel/slick/slick-theme.css";
import "./HomeSlider.css"; // Custom styles for the slider
function HomeSlider() {
  const settings = {
    dots: true,
    infinite: true,
    speed: 500,
    slidesToShow: 1,
    slidesToScroll: 1,
  };

  return (
    <div className="slider-container">
      <Slider {...settings}>
        <div className="slide">
          <img src="/images/slide1.jpg" alt="Slide 1" />
```

```
        </div>
        <div className="slide">
          <img src="/images/slide2.jpg" alt="Slide 2" />
        </div>
        <div className="slide">
          <img src="/images/slide3.jpg" alt="Slide 3" />
        </div>
      </Slider>
    </div>
  );
}
```

- In this example, we define the slider settings such as `dots`, `infinite`, `speed`, `slidesToShow`, and `slidesToScroll`. The `Slider` component wraps individual slide elements, which can contain images or any other content.
- The `slick-carousel` CSS files are imported to apply default styles to the slider. You can also create a custom CSS file (e.g., `HomeSlider.css`) to style the slider further.
- Make sure to replace the image paths with your actual image URLs or paths.
- You can customize the slider further by adjusting the settings object or adding additional props to the `Slider` component.

---

17 What is Tanstack and Intro to React Query?

- **Tanstack** is a collection of libraries for building modern web applications, including React Query, which simplifies data fetching and state management in React applications.
- **Vercel Company:** is the creator of Tanstack, and it provides a suite of tools for building efficient and scalable applications. Tanstack Site
- **React Query** is a powerful library for managing server state **(Asynchronous State Management)** in React applications. It provides tools for fetching, caching, synchronizing, and updating server data without the need for complex state management solutions.
- **Key Features of React Query:**
  - **Handling async State**: Manages asynchronous data fetching and state updates seamlessly.
  - **Query (Automatic) Caching**: `Automatically caches fetched data to improve performance and reduce unnecessary network requests`
  - **Data Fetching**: Simplifies fetching data from APIs with built-in caching and background updates.
  - **Background Updates**: Automatically refetches data in the background to keep it fresh.
  - **Query Invalidation**: Allows you to invalidate and refetch queries when data changes.
  - **DevTools**: Provides a set of tools for debugging and inspecting queries.
- We will use Tanstack Query (React Query) in our e-commerce application to manage product data, user authentication, and other server interactions efficiently. React Query Documentation

---

17.1 What is State Management?

- **State management** is the process of handling and organizing the data (state) that determines how your application behaves and what it displays. In React, effective state management ensures that your UI stays in sync with the underlying data as users interact with the app.
- It involves **storing, updating, and sharing data between components** so that changes in state are accurately reflected across the user interface.
- State management becomes especially important in complex applications where multiple components need to access or update **shared data**. It helps maintain a consistent user experience and ensures that UI updates are predictable and reliable.
- In React, you can manage state using `local component state`, `the Context API`, or `external libraries such as Redux, MobX, or Zustand`. The choice depends on your app's complexity and data flow needs.
- For server-side data, **React Query** streamlines state management by providing hooks to fetch, cache, and synchronize data with the server, making it much easier to handle asynchronous state in React applications.

---

18 Initializing React Query in Our Application

- To Install React Query in your application, you need to install the `@tanstack/react-query` package. This package provides the necessary hooks and components for managing server state in React applications.

```
npm install @tanstack/react-query
```

18.1 Setting Up React Query Provider

- After installing React Query, you need to set up the `QueryClient` and `QueryClientProvider` in your application. This is typically done in your main entry file (e.g., `index.js` or `App.js`).
- Here's how to do it:
- App.jsx File:

```
import React, { useEffect, useState } from "react";
import { createBrowserRouter, RouterProvider } from "react-router-dom";
import Layout from "./Components/Layout/Layout";
import Register from "./Components/Register/Register";
import Login from "./Components/Login/Login";
import Notfound from "./Components/NotFound/Notfound";
import AuthContext from "./Context/AuthContext";
import Cart from "./Components/Cart/Cart";
import Brands from "./Components/Brands/Brands";
import Categories from "./Components/Categories/Categories";
import ProtectedRoute from "./Components/ProtectedRoute/ProtectedRoute";
import Products from "./Components/Products/Products";
```

```
import { QueryClientProvider } from
↪  "./../node_modules/@tanstack/react-query/src/QueryClientProvider";
import { QueryClient } from
↪  "./../node_modules/@tanstack/query-core/src/queryClient";

function App() {
  const router = createBrowserRouter([
    {
      path: "",
      element: <Layout />,
      children: [
        {
          index: true,
          element: (
            <ProtectedRoute>
              <Products />
            </ProtectedRoute>
          ),
        },
        {
          path: "products",
          element: (
            <ProtectedRoute>
              <Products />
            </ProtectedRoute>
          ),
        },
        {
          path: "register",
          element: <Register />,
        },
        {
          path: "login",
          element: <Login />,
        },

        {
          path: "cart",
          element: (
            <ProtectedRoute>
              <Cart />
            </ProtectedRoute>
          ),
        },
        {
          path: "brands",
          element: (
            <ProtectedRoute>
              <Brands />
            </ProtectedRoute>
```

```
          ),
        },
        {
          path: "categories",
          element: (
            <ProtectedRoute>
              <Categories />
            </ProtectedRoute>
          ),
        },

        {
          path: "*",
          element: <Notfound />,
        },
      ],
    }, // Routes
  ]);
  const reactQueryClientConfig = new QueryClient();
  return (
    <>
      <AuthContext>
        <QueryClientProvider client={reactQueryClientConfig}>
          <RouterProvider router={router} />
        </QueryClientProvider>
      </AuthContext>
    </>
  );
}

export default App;
```

- The QueryClient is the core of React Query, managing all queries and their states. The QueryClientProvider wraps your application, providing access to the QueryClient instance throughout your component tree.
- It has the following properties:
  - `client`: The QueryClient instance that manages queries and their states **(configuration options can be passed here)**.
  - `children`: The components that will have access to the QueryClient.
  - `contextSharing`: (optional) If set to true, allows sharing the QueryClient instance across multiple providers.
- The `QueryClientContext` is used to provide the QueryClient instance to your application. You can use the `useQuery` and `useMutation` hooks from React Query to fetch and mutate data in your components.
- The `QueryClient` instance is created using the `new QueryClient()` constructor, which initializes the client with default settings. You can customize these settings as needed.
- The `QueryClientContext` is imported from the React Query library, and it is used to wrap your application, allowing you to access the QueryClient instance in any component

that needs to fetch or mutate data.

---

19 Applying useQuery and Handling States

- To fetch data using React Query, you can use the `useQuery` hook. This hook takes a query key and a function that returns a promise (usually an API call) to fetch the data.
- The `useQuery` hook returns an object containing the query state, including the data, loading status, and error information.
- The `useQuery` is an Object that contains the following properties:
- `queryKey`: A **unique key** for the query, used to identify it in the cache.
- `queryFn`: A function that returns a promise, which is used to fetch the data.
- `data`: The fetched data from the query.
- `error`: Any error that occurred during the query.
- `isLoading`: A boolean indicating if the query is currently loading.
- `isError`: A boolean indicating if the query has encountered an error.
- `isSuccess`: A boolean indicating if the query was successful.
- `refetch`: A function to manually refetch the data.
- We use query keys to uniquely identify queries. This allows React Query to cache and manage the data efficiently. The query key can be a string or an array of strings.
- Here's an example of how to use `useQuery` to fetch products in the `Products` component:

```
import React from "react";
import { useQuery } from "@tanstack/react-query";
import axios from "axios";
function Products() {
  const { data, error, isLoading, isError } = useQuery({
    queryKey: ["products"],
    queryFn: async () => {
      const response = await axios.get(
        "https://ecommerce.routemisr.com/api/v1/products"
      );
      return response.data.data;
    },
  });

  if (isLoading) {
    return <div>Loading...</div>;
  }

  if (isError) {
    return <div>Error: {error.message}</div>;
  }
```

```
  return (
    <div className="products">
      {data.map((product) => (
        <div key={product._id} className="product">
          <h2>{product.title}</h2>
          <p>{product.price} EGP</p>
        </div>
      ))}
    </div>
  );
}
```

- In this example, the `useQuery` hook is used to fetch products from the API. The `queryKey` is set to `["products"]`, which uniquely identifies this query.
- The `queryFn` is an asynchronous function that makes the API call using Axios and returns the fetched data.

Note on `queryKey`:

- The `queryKey` property in React Query uniquely identifies each query in the cache. It can be either a string or an array.
- Using a **string** (e.g., `queryKey: "allProducts"`) is suitable for simple queries.
- Using an **array** (e.g., `queryKey: ["products", categoryId]`) is recommended for more complex queries, such as when you need to differentiate queries by parameters (like filtering by category or ID).

Examples:

```
// Simple query with a string key
queryKey: "allProducts";

// Equivalent array form (not recommended for a single string)
queryKey: ["allProducts"];

// Query with multiple parameters (recommended)
queryKey: ["products", categoryId];

// Example: Fetching product with ID "123"
queryKey: ["products", "123"];
// This identifies the query for product with ID 123
```

- Using arrays for `queryKey` helps React Query manage and cache queries more effectively, especially when queries depend on dynamic parameters.
- Products.jsx File

```
function Products() {
  // This component will render the products
  // You can fetch products from an API or use static data
  // const [allProducts, setAllProducts] = useState(null);
  // async function getAllProducts() {
  //   const { data } = await axios.get(
  //     "https://ecommerce.routemisr.com/api/v1/products"
```

```
//   );

//   setAllProducts(data.data);
// }

// // Call the function to fetch products in componentMount or
↪  useEffect
// // If using functional components, you can use useEffect

// useEffect(() => {
//   getAllProducts();
//   // Initialize AOS for animations
//   AOS.init({
//     duration: 1000, // Animation duration in milliseconds
//     // once: false, // Whether animation should happen only once
//     mirror: true, // Whether elements should animate out while
↪  scrolling past them
//     easing: "ease-in-out", // Easing function for the animations
//     delay: 100, // Delay before the animation starts
//   });
// }, []);

// -----------------Applying Use Query -------------------

// Function to fetch all products
function getAllProducts() {
  return axios.get("https://ecommerce.routemisr.com/api/v1/products");
}

const { data, isLoading, error, isFetching } = useQuery({
  queryKey: ["productDetails"],
  queryFn: getAllProducts,
});
console.log("Data:", data);
console.log("Is Loading:", isLoading);
console.log("Error:", error);
console.log("Is Fetching:", isFetching);

// Initialize AOS when component mounts
useEffect(() => {
  AOS.init({
    duration: 1000,
    mirror: true,
    easing: "ease-in-out",
    delay: 100,
  });
}, []);

// Refresh AOS when new data (products) is available
useEffect(() => {
```

```
    if (data) {
      AOS.refresh();
    }
  }, [data]);

  if (isLoading) {
    return <LazyLoading message="Loading Products..." fullPage={true} />;
  }
  if (error) {
    return <div>Error fetching products: {error.message}</div>;
  }

  const allProducts = data?.data?.data || []; // This is mean allProducts
  ↪  What i want to display
}
```

---

20 Additional useQuery Configuration Options

- The **Business logic** of React Query is highly customizable, allowing you to control how data is fetched, cached, and updated. You can configure various options to optimize the behavior of your queries based on your application's needs.
- React Query's `useQuery` hook offers several options to fine-tune data fetching and caching behavior:

20.1 Common useQuery Configuration Options

1. **`refetchOnMount`**

- Controls whether the query refetches when the component mounts.
- `true` (default): Refetches every time the component mounts.
- `false`: Does not refetch on mount (useful for static data).
- Example: `refetchOnMount: false`

2. **`refetchInterval`**

- Sets an automatic refetch interval (in milliseconds).
- Example: `refetchInterval: 10000` (refetches every 10 seconds)

3. **`staleTime`**

- Duration (in milliseconds) that data is considered "fresh" before becoming "stale" and eligible for refetching.
- Example: `staleTime: 60000` (data is fresh for 1 minute)

4. **`retry`**

- Number of retry attempts if the query fails.
- Default: `3` in development, `0` in production.
- Set to `false` to disable retries.
- Example: `retry: 2`

5. **`retryDelay`**

- Delay (in milliseconds) between retry attempts.
- Example: `retryDelay: 2000` (2 seconds between retries)

6. **`cacheTime`**

- How long unused/inactive query data remains in the cache before being garbage collected (in milliseconds).
- Default: 5 minutes (`300000` ms).
- Example: `cacheTime: 300000`

7. **`placeholderData`**

- Temporary data displayed while the query is loading (useful for skeleton UIs or optimistic updates).

8. **`keepPreviousData`**

- If `true`, keeps showing the previous data while fetching new data (helpful for pagination or filtering).

9. **`windowFocusRefetching`**

- If `true` (default), refetches queries when the browser window regains focus.
- Example: `windowFocusRefetching: false` (disables refetching on window focus)
- **Note:** If `windowFocusRefetching` is `false`, queries will not refetch on window focus, even if the data is stale.

Example usage:

```
const { data, isLoading } = useQuery({
  queryKey: ["products"],
  queryFn: fetchProducts,
  refetchOnMount: false,
  staleTime: 60000,
  refetchInterval: 10000,
  retry: 2,
  retryDelay: 1500,
  cacheTime: 300000,
  keepPreviousData: true,
  placeholderData: keepPreviousData ? previousData : [],
});
// Explain The above code
// This code fetches products using React Query with specific
 ↪  configurations:
// - `queryKey`: Unique identifier for the query.
// - `queryFn`: Function to fetch data.
// - `refetchOnMount`: Do not refetch on mount.
// - `staleTime`: 1 minute before data becomes stale.
// - `refetchInterval`: Refetch every 10 seconds.
// - `retry`: 2 retry attempts on failure.
// - `retryDelay`: 1.5 seconds delay between retries.
// - `cacheTime`: 5 minutes before unused data is garbage collected.
```

```
// - `keepPreviousData`: Keeps previous data while fetching new data.
// - `placeholderData`: Displays previous data while loading new data.
```

---

21 Controlling Query Execution with `enabled` Property and `refetch` Function

React Query allows you to control when a query should run using the `enabled` option in the `useQuery` hook. By setting `enabled` to a boolean value, you can conditionally enable or disable the query based on your application's logic. This is especially useful when you want to delay fetching data until certain conditions are met (e.g., waiting for user input or another piece of state).

Additionally, React Query provides a `refetch` function, which you can call manually to re-fetch data whenever needed, regardless of the `enabled` state.

Example:

```
const { data, isLoading, refetch } = useQuery({
  queryKey: ["userData", userId],
  queryFn: fetchUserData,
  enabled: !!userId, // Only fetch if userId exists
});

// Manually trigger a refetch
<button onClick={() => refetch()}>Refresh Data</button>;
```

- Use `enabled` to control automatic fetching.
- Use `refetch` to manually trigger data fetching on demand.

---

22 The Advantage of QueryKey in Categories Page & Slider and Custom Hooks

- Using a `queryKey` in React Query is crucial for managing and caching data effectively. It allows you to uniquely identify each query, enabling React Query to cache results and avoid unnecessary network requests. This is particularly useful when dealing with dynamic data, such as categories or products, where the data may change based on user interactions or filters.
- By using a structured `queryKey`, you can easily differentiate between queries that fetch different sets of data, such as products by category or brand. This helps in optimizing performance and ensuring that your application only fetches the data it needs when it needs it.
- For example, when fetching products by category, you can use a `queryKey` like `["products", categoryId]`. This allows React Query to cache the results for each category separately, improving performance and reducing redundant API calls.

- **We Have Applied the same function of the useQuery hook in our Categories Page and Categories Slider so when i click on a category, it will fetch the products related to that category using the `queryKey` to differentiate between different categories.**

```
function getAllCategories() {
  // Fetch or retrieve category data here
  return axios.get("https://ecommerce.routemisr.com/api/v1/categories");
}
const { data, isLoading, error } = useQuery({
  queryKey: ["categoryDetails"], // Unique key for the query
  queryFn: getAllCategories, // Function to fetch data
});

// Initialize AOS when component mounts
useEffect(() => {
  Aos.init({
    duration: 1000,
    mirror: true,
    easing: "ease-in-out",
    delay: 100,
  });
}, []);

// Refresh AOS when new data (products) is available
useEffect(() => {
  if (data) {
    Aos.refresh();
  }
}, [data]);

if (isLoading) {
  return <LazyLoading message="Loading Categories..." fullPage={true} />;
}
if (error) {
  return <div>Error fetching categories: {error.message}</div>;
}
```

- Note: We have a little problem that i have repeated the same code for fetching categories in both the CategorySlider and Products components. To avoid code duplication, we can create a custom hook that encapsulates the category fetching logic and use it in both components.

22.1 Creating a Custom Hook for Fetching Categories

- To avoid duplicating category-fetching logic across components, you can create a custom React hook. This hook encapsulates the data fetching using React Query and can be reused wherever you need category data.

Example: `useCategories.js`

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

function fetchCategories() {
  return axios.get("https://ecommerce.routemisr.com/api/v1/categories");
}

export function useCategories() {
  return useQuery({
    queryKey: ["categories"],
    queryFn: fetchCategories,
  });
}
```

Usage in a Component:

```
import { useCategories } from "../hooks/useCategories";

function CategoryList() {
  const { data, isLoading, error } = useCategories();

  if (isLoading) return <div>Loading categories...</div>;
  if (error) return <div>Error: {error.message}</div>;

  return (
    <ul>
      {data.data.data.map((category) => (
        <li key={category._id}>{category.name}</li>
      ))}
    </ul>
  );
}
```

Note:
Custom hooks must be called at the top level of a React function component or another custom hook—not inside regular functions or conditionals.

22.2 Steps to Create a Custom Hook

1. **Create a `CustomHooks` folder** inside your `src` directory.
2. **Name your custom hook starting with `use`**, such as `useAllCategories`. This follows React's naming convention for hooks.
3. **Custom hooks do not return JSX**. Instead, they return shared logic or data (usually as an object) that can be reused across components.
4. **A custom hook must use at least one React hook** (e.g., `useQuery`, `useState`, `useEffect`) internally.

Note:
Custom hooks help you extract and reuse logic between components, keeping your code DRY and maintainable.

22.2.1 Example: `useAllCategories.jsx` Custom Hook

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";

export default function useAllCategories() {
  // Function to fetch all categories
  function getAllCategories() {
    return axios.get("https://ecommerce.routemisr.com/api/v1/categories");
  }
  // Use React Query's useQuery hook
  return useQuery({
    queryKey: ["categoryDetails"], // Unique key for caching
    queryFn: getAllCategories,
  });
}
```

22.2.2 Using the Custom Hook in Components

Instead of duplicating fetching logic in each component, simply use your custom hook:

```
import useAllCategories from "../CustomHooks/useAllCategories";

const { data, isLoading, error } = useAllCategories();
```

You can now use this in any component, such as `CategorySlider.jsx` or `Categories.jsx`, to access category data, loading, and error states.

22.2.3 Creating and Using a Custom AOS Hook in React

To avoid repeating AOS initialization and refresh logic across components, you can create a reusable custom hook.

`useAOS.jsx` Custom Hook

```
import { useEffect } from "react";
import AOS from "aos";

export default function useAOS(options = {}, dependencies = []) {
  // Default AOS settings
  const defaultOptions = {
    duration: 1000,
    mirror: true,
    easing: "ease-in-out",
    delay: 100,
  };

  // Initialize AOS on mount with merged options
  useEffect(() => {
    AOS.init({ ...defaultOptions, ...options });
  }, []);

  // Refresh AOS when dependencies change
```

```
  useEffect(() => {
    AOS.refresh();
  }, dependencies);
}
```

How to Use the Custom Hook Call `useAOS` at the **`top of your functional component`** to initialize and refresh AOS as needed:

```
import useAOS from "../CustomHooks/useAOS";

function MyComponent({ data }) {
  useAOS(
    {
      duration: 1500,
      easing: "ease-out-back",
      once: true,
    },
    [data] // dependencies to trigger refresh
  );

  // ...component code
}
```

Benefits:

- Keeps your code DRY by centralizing AOS logic.
- Easily customize AOS options and refresh behavior per component.
- Ensures consistent animation initialization and updates across your app.

---

23 Product Details Page: Navigation, Dynamic Routing, and Data Fetching

23.1 Overview

The Product Details page displays comprehensive information about a specific product. Access to this page is protected—only authenticated users can view it. When a user clicks a product in the products list, they are navigated to the details page, which fetches and displays data for the selected product.

23.2 1. Navigating to Product Details with URL Parameters

- **Best Practice:** Pass the product ID as a URL parameter using React Router's `<Link>` component. This avoids unnecessary use of local storage and keeps navigation clean and RESTful.

Example:

```
<Link
  to={`/productDetails/${product._id}`}
```

```
  className="product bg-white rounded-lg shadow-md p-4 flex flex-col
   ↪  items-center hover:shadow-lg transition-shadow duration-300"
>
  {/* Product image and details */}
</Link>
```

23.3 2. Defining the Protected Route

- In your router configuration, define a dynamic route for product details and wrap it with your `ProtectedRoute` component:

```
{
  path: "productDetails/:id",
  element: (
    <ProtectedRoute>
      <ProductDetails />
    </ProtectedRoute>
  ),
}
```

- The `:id` segment makes the route dynamic, allowing you to access the product ID in the details page.

23.4 3. Accessing URL Parameters with `useParams`

- Use the `useParams` hook from React Router to extract the product ID from the URL inside the `ProductDetails` component.

Example:

```
import { useParams } from "react-router-dom";

function ProductDetails() {
  const { id } = useParams(); // id is the product ID from the URL
  // ...fetch and display product details
}
```

23.5 4. Fetching Product Details with React Query

- Use React Query's `useQuery` hook to fetch product data based on the dynamic ID.
- Use a dynamic `queryKey` (e.g., `["productDetails", id]`) for efficient caching and refetching.

Example:

```
import { useQuery } from "@tanstack/react-query";
import axios from "axios";
import { useParams } from "react-router-dom";
import { useState, useEffect } from "react";
```

```
import LazyLoading from "../LazyLoading/LazyLoading";

function ProductDetails() {
  const { id } = useParams();
  const [imageCover, setImageCover] = useState(null);

  function getProductDetails() {
    return
     ↪  axios.get(`https://ecommerce.routemisr.com/api/v1/products/${id}`);
  }

  const { data, isLoading, error } = useQuery({
    queryKey: ["productDetails", id],
    queryFn: getProductDetails,
  });

  useEffect(() => {
    if (data?.data?.data && !imageCover) {
      setImageCover(data.data.data.imageCover);
    }
  }, [data, imageCover]);

  if (isLoading) {
    return <LazyLoading message="Loading Product Details..."
     ↪  fullPage={true} />;
  }
  if (error) {
    return (
      <div className="flex items-center justify-center min-h-screen
       ↪  bg-gradient-to-br from-red-50 to-red-100">
        <div className="p-6 bg-white/80 backdrop-blur-lg rounded-xl
         ↪  shadow-lg border border-red-200">
          <p className="text-red-600 text-lg font-semibold">
            Error fetching product details: {error.message}
          </p>
        </div>
      </div>
    );
  }

  const product = data.data.data;

  return (
    <div className="container mx-auto px-4 py-12 min-h-screen
     ↪  bg-gradient-to-br from-gray-50 to-gray-100">
      <div className="grid grid-cols-1 lg:grid-cols-2 gap-8 bg-white/80
       ↪  backdrop-blur-lg rounded-2xl shadow-xl p-8 md:p-12 transition-all
       ↪  duration-300 hover:shadow-2xl border border-gray-100">
        {/* Image Section */}
        <div className="flex flex-col items-center space-y-8">
```

```
          <div className="relative group w-96">
            <img
              src={imageCover || product.imageCover}
              alt={product.title}
              className="w-96 h-[28rem] object-cover rounded-2xl border
              ↪  border-gray-200 shadow-lg group-hover:scale-105
              ↪  transition-transform duration-500 ease-out"
            />
            <div className="absolute inset-0 bg-gradient-to-t from-black/20
            ↪  to-transparent rounded-2xl opacity-0
            ↪  group-hover:opacity-100 transition-opacity duration-300" />
          </div>
          {product.images && product.images.length > 0 && (
            <div className="flex flex-wrap gap-4 justify-center w-96">
              {product.images.map((img, idx) => (
                <img
                  key={idx}
                  src={img}
                  alt={`Product thumbnail ${idx + 1}`}
                  onClick={() => setImageCover(img)}
                  className="w-20 h-20 object-cover rounded-lg border
                  ↪  border-gray-200 shadow-sm hover:scale-110
                  ↪  hover:border-emerald-400 transition-all duration-300
                  ↪  cursor-pointer"
                />
              ))}
            </div>
          )}
        </div>

        {/* Details Section */}
        <div className="flex flex-col h-full">
          <div className="flex-grow space-y-4">
            <h1 className="text-3xl font-bold text-gray-900/90
            ↪  tracking-tight">
              {product.title}
            </h1>
            <h5 className="text-sm font-medium text-gray-800 uppercase">
              {product.category.name}
            </h5>
            <p className="text-gray-600 leading-relaxed text-base">
              {product.description}
            </p>
          </div>
          <div className="border-t border-gray-200 my-4" />
          <div className="space-y-4 pt-2">
            <div className="flex items-center justify-between">
              <span className="text-lg font-semibold text-gray-600/70">
                EGP {product.price.toLocaleString()}
              </span>
```

```
              <div className="flex items-center space-x-2">
                <span className="text-yellow-400 text-lg">
                  <i className="fa-solid fa-star"></i>
                </span>
                <span className="text-gray-700/60 font-semibold">
                  {product.ratingsAverage}
                </span>
              </div>
            </div>
            <button className="w-full bg-gradient-to-r from-emerald-500
            ↪  to-teal-500 hover:from-emerald-600 hover:to-teal-600
            ↪  text-white py-2 px-6 rounded-xl font-semibold text-lg
            ↪  shadow-md hover:shadow-xl transition-all duration-300
            ↪  active:scale-95 focus:outline-none focus:ring-4
            ↪  focus:ring-emerald-200 cursor-pointer">
              + Add to Cart
            </button>
          </div>
        </div>
      </div>
    </div>
  );
}

export default ProductDetails;
```

23.6 5. Key Points and Best Practices

- **Use URL parameters** for navigation between pages with dynamic data (e.g., product IDs).
- **Access parameters** in components with `useParams`.
- **Fetch data** using React Query with a dynamic `queryKey` for optimal caching.
- **Protect sensitive routes** with a `ProtectedRoute` wrapper.
- **Display loading and error states** for better user experience.

By following these steps, you ensure a robust, maintainable, and user-friendly product details workflow in your React application.

---

24 Discussing Why We Need Context For Cart and Initializing Cart Context

- In our e-commerce application, we need to manage the shopping cart state across different components. This includes adding items to the cart, removing items, and displaying the cart contents.

24 Discussing Why We Need Context For Cart and Initializing Cart Context

- Using React's Context API allows us to create a global state for the cart that can be accessed and modified from any component in the application.
    - **This is Called State Management Concept**.
- The Context API provides a way to share values (like the cart state) between components without having to pass props down through every level of the component tree. This simplifies the code and makes it easier to manage the cart state.
- By creating a Cart Context, we can encapsulate all cart-related logic (adding, removing, updating items) in one place, making it easier to maintain and extend the functionality in the future.