Amazon Clone Session 6

In our previous session we stopped on the login page with the input and password fields to capture the data using the useState, useEffect and useReducer. We will continue with that right after we fix up the styles for our landing page.

For our landing page I would like to change the border colour and add some interactivity onto our page as well as add another item to the first row so that we have 4 cards displaying at the top.

Let's first change the styling on the Products.css:

```
/* Products.css*/
.products_container{
    display: flex;
    flex-direction: column;
    align-items: center;
    margin: 50px 0 0 0;
    z-index: 1;
}

.products_row{
    display: flex;
    margin: 0 0 20px 0;
}

.product{
    margin: 10px;
}
```

Then we can modify our Product.css file to change up the button styles and the border colour:

```
/* Product.css */
.product {
    background-color: #fff;
    border: 1px solid #ddd;
    z-index: 100;
    padding: 40px;
}
```

```
.product_rating {
    display: flex;
    justify-content: center;
}

.product_button {
    background-color: #febd69;
    padding: 15px;
    cursor: pointer;
    border: 1px solid #febd69;
    transition: background-color 0.3s ease, border-color 0.3s ease;
}

.product_button:hover {
    background-color: #fcb251;
    border: 1px solid #777777;
}

.product img {
    max-width: 200px;
}
```

I also want to add an additional product to my row to look more like the Amazon page with 4 cards. The rest of the cards below can fill the width to show some flexibility of the cards.

Next to navigate to our login page from our landing page or our home page, we need to update the link in our header:

```
      <Link to="/login" className="header_optionLink">
        <div className="header_option">
          <span className="header_optionOne">Hello Guest</span>
          <span className="header_optionTwo">Sign-in</span>
        </div>
      </Link>
```

Now our link should work correctly and navigate us to our login page. The next thing we need to do is just update the styles on the link so that we can see it clearly without the purple colour and the underline.

Now that we are back on the login page, we can update our styles on our buttons quick, I would like to add a bit of a transition on the button so that it does not look like it just flashes to a different colour.

```
transition: background-color 0.3s ease, border-color 0.3s ease;
```

Now our styles are all up to date. Just for a summary on useState and useReducer before we continue:

When to Use `useState`

- **Simple State**: Ideal for managing simple state variables such as strings, numbers, booleans, or simple objects.
- **Independent State**: Best for state variables that are independent and don't require complex updates.

When to Use `useReducer`

- **Complex State**: Suitable for managing complex state objects or when state updates involve multiple actions.
- **Related State**: Useful when state variables are related and require combined logic for updates.
- **Predictable State Transitions**: Helps encapsulate complex state transitions and rules.

Using `useState` and `useReducer` Together

- **Local Simple State**: Use `useState` for simple, local state variables.
- **Global/Complex State**: Use `useReducer` for managing complex, interconnected state logic.

Understanding Global State and Context in React

What is Global State?

In React, state refers to the data that controls the behavior and rendering of components. Normally, state is managed within individual components. However, when multiple components need to access or modify the same state, managing state locally within each component can become complex and inefficient. This is where the concept of ***global state*** comes in.

Global state is a way to manage state that is accessible by multiple components across the application. Instead of passing down state through props from parent to child components (which can lead to "prop drilling" and messy code), global state provides a centralized place for state management.

Example: Logging In

Imagine an application where users need to log in. The logged-in state (whether the user is authenticated or not) and user information need to be accessed by various components:
- The header component needs to display the user's name if they are logged in.
- A profile component needs to show the user's details.
- A dashboard component needs to load data specific to the logged-in user.

Managing the logged-in state locally within each of these components would be cumbersome and error-prone. Instead, we can use global state to manage this information in a single place.

How Context Can Assist

Context is a feature in React that allows you to share state across the entire application (or a portion of it) without having to pass props down manually at every level.

1. Centralized State Management: Context provides a way to create a global state that can be accessed by any component in the tree. For example, you can create an `AuthContext` to manage the authentication state.

2. Avoid Prop Drilling: By using context, you avoid the need to pass down props through many layers of components. Instead, any component that needs access to the global state can subscribe to the context.

3. Simplifies Code: With context, your code becomes cleaner and easier to manage because the state logic is centralized. Components that need access to the global state can simply consume the context, making the application easier to understand and maintain.

Logging In with Context

Using logging in example, you can create an `AuthContext` that holds the authentication state and provides functions to log in and log out. This context can then be used by any component that needs to check if a user is logged in, display user information, or handle authentication-related logic.

In summary:
- ***Global state*** helps manage state that needs to be shared across multiple components.
- ***Context*** provides a way to create and manage this global state, making your application more modular and easier to maintain.

By introducing context to manage global state, you learn how to handle complex state management scenarios in a scalable and efficient way, paving the way for more advanced state management solutions like Redux in the future.

Let's have a look at how we can apply it in our own application.

Firstly we will be using the useState to set whether a user is logged in or not.

```
const [isLoggedIn, setIsLoggedIn] = useState(false);
```

Then we can manage this with a useEffect:

```
useEffect(() => {
  const userInfo = localStorage.getItem('isLoggedIn')

  if(userInfo === '1'){
    setIsLoggedIn(true)
  }
}, [])
```

Make sure both the useState and useEffect are imported:

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

Now that we have that in place we need to create 2 functions. One to set the logged in status when a user logs in as well as a function to set it to false when a user logs out.

```
const loginHandler = (email, password) => {
  localStorage.setItem('isLoggedIn', '1');
  setIsLoggedIn(true);
}

const logoutHandler = () => {
  localStorage.removeItem('isLoggedIn');
  setIsLoggedIn(false);
}
```

Now if we just stop here and discuss this for a bit. When we look at our login page, our button is present on that component and then we also have the header component where the user can log out once they have signed in. So we need to work with both our header as well as the login page.

So we need to now pass this logged in status to our header.

```
<Header isAuthenticated={isLoggedIn}/>
```

And we should also pass our logoutHandler function as props;

```
<Header isAuthenticated={isLoggedIn} onLogout={logoutHandler}/>
```

Now we can go ahead and pass our loginHandler as props

```
<Route path="/login" element={<Login onLogin={loginHandler}/>}></Route>
```

In our ***Header.js*** we need to accept these new props:

```
const Header = ({isAuthenticated, onLogout}) => {
```

We can now use these props in our header component. Where the Sign in link is in the header, we need to add a conditional statement to render the component based on the logged in status of the user. If the user is logged in then it shows us the Sign out option

and the other way around. While we are working on this we can also insert our onLogout props that we have passed. This you can add in the onClick event in the link.

```
      {isAuthenticated ? (
        <Link to="/login" className="header_optionLink"
onLogout={logoutHandler}
>
          <div className="header_option">
            <span className="header_optionOne">Hello User</span>
            <span className="header_optionTwo">Sign Out</span>
          </div>
        </Link>
      ) : (
        <Link to="/login" className="header_optionLink">
          <div className="header_option">
            <span className="header_optionOne">Hello Guest</span>
            <span className="header_optionTwo">Sign In</span>
          </div>
        </Link>
      )}
```

Now this should navigate us through to our home page when the user logs out. Now if the user goes back to the login page we need to ensure that we are triggering the login handler in our ***App.js.***

This is also passed down as props to our login route:

```
<Route path="/login" element={<Login onLogin={loginHandler}/>}></Route>
```

So we need to accept this onLogin prop on our login page as well:

```
const Login = ({onLogin}) => {
```

Then we should add the onLogin prop to our signIn function where it requires the 2 parameters, email and password:

```
const signIn = (e) => {
  e.preventDefault();
  console.log("Entered Email: ", state.emailValue)
  console.log("Entered Password: ", state.passwordValue)
  onLogin(state.emailValue, state.passwordValue)
};
```

Now logging in and logging out should be possible and the key and value should be stored and removed when we click on these buttons / links.

So the method we used in this case is what we refer to as props drilling. Passing the props deeper and deeper into the application.This method is fine for a small application as what we are building but for larger applications this can be really difficult to manage.

So this is where context becomes a very valuable tool for us to use where instead of drilling down each component with our props we can just fetch it from the context instead.

Let's replace this with using the context method instead.

In your source folder (src) create a new folder called context, and inside the context folder create a new file called ***authContext.js***

Inside of this new ***authContext.js*** file add we need to first import ***createContext*** from react:

```
import { createContext } from "react";
```

We then need to use the built in function createContext and then pass the object with the initial state, which is isLoggedIn and the initial state is false.

```
createContext({isLoggedIn: false})
```

Since we will be using it outside of this file we will need to assign it to a variable.

```
const AuthContext = createContext({isLoggedIn: false})
```

Then we need to export it to ensure that it can be used.

```
export default AuthContext
```

Now in your ***App.js*** we need to add the AuthContext.Provider and wrap our entire App,js router in it and pass the values in the AuthContext.Provider.

```
  return (
    <AuthContext.Provider value={{isLoggedIn: isLoggedIn}}>
      <Router>

        <Header isAuthenticated={isLoggedIn} onLogout={logoutHandler}/>

        <main>

          <Routes>

            {/* Prevents root dir access */}
            <Route path="/" element={<Navigate to="/home"/>}/>

            {/* This is the parent route */}
            <Route path="/home" element={<Home />}></Route>

            {/* Routes for products */}
            <Route path="/products" element={<Products />}></Route>
            <Route path="/products/:id" element={<ProductDetails
/>}></Route>

            <Route path="/login" element={<Login
onLogin={loginHandler}/>}></Route>

            {/* Catch all route for non-existing routes */}
            <Route path="*" element={<NotFound />} />
          </Routes>
        </main>
      </Router>
    </AuthContext.Provider>
  );
```

Make sure you have the authContext imported at the top of your file:

```
import AuthContext from "./context/authContext";
```

Next we need to make adjustments to our header component and wrap that in the AuthContext.Provider as well:

```
import AuthContext from "../../context/authContext";

const Header = ({ isAuthenticated, onLogout }) => {
  return (
    <AuthContext.Consumer>
      {(ctx) => {
        return (
          <header className="header">
          </header>
        );
      }}
    </AuthContext.Consumer>
  );
};

export default Header;
```

We also need to update the initial conditions that we put in place to display Sign in or Sign Out:

```
            {ctx.isLoggedIn ? (
              <Link to="/" className="header_optionLink"
onClick={onLogout}>
                <div className="header_option">
                  <span className="header_optionOne">Hello User</span>
                  <span className="header_optionTwo">Sign Out</span>
                </div>
              </Link>
            ) : (
              <Link to="/login" className="header_optionLink">
                <div className="header_option">
                  <span className="header_optionOne">Hello Guest</span>
                  <span className="header_optionTwo">Sign In</span>
                </div>
              </Link>
            )}
```

Next you can remove the isAuthenticated prop from your Header component in the App.js:

```
<Header onLogout={logoutHandler}/>
```

You can also remove the isAuthenticated prop from your header component to look like this.

```
const Header = ({ onLogout }) => {
```

Thats it now we have started using context to access variables or states globally instead of using the prop drilling method.

Next we can look at the useContext Hook. Navigate to your Header.js component again. Create a new constant which we will use to replace our ctx for our conditional rendering for the sign out and sign in.

```
const Header = ({ onLogout }) => {
  const ctx = useContext(AuthContext)
```

And remove the AuthContext.Consumer as we no longer need it as we are fetching it from the AuthContext.

```
const Header = ({ onLogout }) => {
const ctx = useContext(AuthContext)

  return (
    <AuthContext.Consumer>
      {(ctx) => {
        return (
          <header className="header">
          </header>
        );
      }}
    </AuthContext.Consumer>
  );
};

export default Header;
```

Let's do the same for our onLogout. All we need to do is modify our App.js Header component. Remove the logoutHandler from the header component, and then add it to your AuthContext.Provider.

```
<AuthContext.Provider value={{isLoggedIn: isLoggedIn, onLogout: logoutHandler}}>
```

```
<Header onLogout={logoutHandler}/>
```

Now to round it off we can add the onLogout function to our AuthContext as well. This will ensure that it comes up in our suggestions.

```
const AuthContext = createContext({isLoggedIn: false, onLogout: () => {}})
```

Now we can go ahead and sort out the rest of the application and clean up our code and really make use of the AuthContext provider add

Add your Login to your AuthContext variable:

```
const AuthContext = createContext({
  isLoggedIn: false,
  onLogout: () => {},
  onLogin: (email, password) => {},
});
```

Then add a new export variable at the bottom.

```
export const AuthContextProvider = (props) => {

    return <AuthContext.Provider>{props.children}</AuthContext.Provider>
}
```

Then cut the isLogin variable and the functions from your App.js and paste them in your new export variable.

```
export const AuthContextProvider = (props) => {
    const [isLoggedIn, setIsLoggedIn] = useState(false);

    useEffect(() => {
    const userInfo = localStorage.getItem('isLoggedIn');

    if(userInfo === '1'){
      setIsLoggedIn(true);
    }
  }, []);

  const loginHandler = (email, password) => {
    localStorage.setItem('isLoggedIn', '1');
    setIsLoggedIn(true);
  }

  const logoutHandler = () => {
    localStorage.removeItem('isLoggedIn');
    setIsLoggedIn(false);
  }
```

Remember to also add your useState and useEffect:

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

Update the return with the AuthContext.Provider to include the values for your login and logout handlers and states.

```
return <AuthContext.Provider value={{isLoggedIn: isLoggedIn, onLogout:
logoutHandler, onLogin:
loginHandler}}>{props.children}</AuthContext.Provider>
```

Your App.js should now look like this:

```
import {
  BrowserRouter as Router,
  Routes,
```

```
  Route,
  Navigate,
} from "react-router-dom";
import "./App.css";
import Home from "./components/Home";
import Products from "./components/Products";
import Header from "./components/layouts/Header";
import ProductDetails from "./components/Product";
import NotFound from "./components/NotFound";
import Login from "./components/Login";

const App = () => {
  return (
    <>
      <Router>
        <Header />
        <main>
          <Routes>
            {/* Prevents root dir access */}
            <Route path="/" element={<Navigate to="/home" />} />

            {/* This is the parent route */}
            <Route path="/home" element={<Home />}></Route>

            {/* Routes for products */}
            <Route path="/products" element={<Products />}></Route>
            <Route path="/products/:id" element={<ProductDetails
/>}></Route>

            <Route
              path="/login"
              element={<Login />}
            ></Route>

            {/* Catch all route for non-existing routes */}
            <Route path="*" element={<NotFound />} />
          </Routes>
        </main>
      </Router>
    </>
```

```
  );
};

export default App;
```

Now we need to import and wrap our application in our index.js with the AuthContextProvider:

```
import React from "react";
import ReactDOM from "react-dom/client";
import "./index.css";
import App from "./App";
import { AuthContextProvider } from "./context/authContext";

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(
  <AuthContextProvider>
    <App />
  </AuthContextProvider>
);
```

The last cleanup is the login form. We need to import AuthContext and then create a new variable in our login component:

```
import AuthContext from "../context/authContext";
```

```
const Login = () => {
  const { onLogin } = useContext(AuthContext);
```

Remember to remove the login props and add useContext import from react.

Now you should see that the authentication process works well and our code looks cleaner using the useContext method instead of trying to manage the states locally by prop drilling.