Custom Hooks

Our app is doing fine with authentication but there's a little bit of technical debt written into that algorithm; it could benefit from some abstraction. If we had a centralized function that could remember authentication and authorization status across components and across rerenders we could re-use logic and cut back on the low-level coding.

Let's make a custom hook - useAuth() to encapsulate all the doings of authentication.

Writing the hook itself

1. Create a new file called useAuth.ts

2. It should export a function called useAuth. The useAuth function should return a simple object. For now make it an empty object:

```
export type AuthHookType = {}
export function useAuth(): AuthHookType {
  return {};
}
```

Are you authenticated?

3. Add a const to the function called isAuthenticated. Initialize it to true.

4. Add isAuthenticated to the returned object:

```
return { isAuthenticated };
```

5. Import this hook into NavBar.tsx:

```
import { useAuth } from './hooks/useAuth.ts';  // or wherever you actually put it
```

6. Inside NavBar() at the top, execute useAuth():

```
const { isAuthenticated } = useAuth();
```

7. This variable is, of course, hardcoded to true. Console.log() it to make certain.

8. Now, use it. Change this:

```
{user ? null : <Link to="/login">Log in</Link>}
{user && <Link to="#" onClick={() => setUser(undefined)}>Welcome {user.first} (Log
out)</Link>}
```

9. ... to this:

```
{isAuthenticated ? null : <Link to="/login">Log in</Link>}
```

```
{isAuthenticated && <Link to="#" onClick={() => setUser(undefined)}>Welcome
{user?.first} (Log out)</Link>}
```

10. Run and test. Change the hardcoded iaAuthenticated from true to false and re-test.

Got the idea of how it'll work? Cool. Let's make that hardcode smarter.

Adding to the hook

11. Edit your useAuth hook. Add a function to login. Here's an example to get you going, There are some additional hints and instructions below.

```
const login = (username: string, password: string) => {
  loginToServer(username, password)
    .then(user => (setUser(user), user))
    .then(user => toast.success(`Welcome ${user?.first}`))
    .then(() => setIsAuthenticated(true))
    .catch(err => toast.error(`Can't log in. ${err.message}`));
}
```

Obviously this doesn't work just yet; we have to import some things (loginToServer and toast) and define some things (like setUser and setAuthentication). See if you can guess how we're going to provide these values before you read the answers below.

12. You're going to need to import the login function and toast.

```
import { login as loginToServer } from '../data/authentication';
import toast from "react-hot-toast";
```

13. Remember how we hardcoded authenticated? Change it so that authenticated is a state variable. In other words, create a state object for authenticated and setAuthenticated. Hint: import useState() and call useState() just like you would in a functional component.

14. Do it again for user/setUser.

15. Make sure user and login are returned from the useAuth hook function as part of its object.

We are going to test these soon but we need to write one more function, logout.

16. Write a logout function similar to the login function. See if you can write it without copy/pasting. It should receive no parameters, set user to undefined and isAuthenticated to false. Optionally pop up a toast notification when the user is logged out.

Note: Do not return setUser or setAuthenticated from the hook. Do you see why? These functions are 100% handled inside of useAuth and therefore should be encapsulated. If you expose them, some developer will be tempted to use them and circumvent our internal business rules.

Using the login function

Now let's refactor the login process to use our new hook.

17. Edit Login.tsx. Remove the import to the old login function and import your new useAuth hook.

18. Call your hook like this:

```
const { login } = useAuth();
```

19. Since we're now setting the user inside the hook, you can remove all references to setUser from Login, including the input prop. (Hint: this means you'll need to change how <Login /> is called in App.tsx).

20. If we did this right, handleLogin is now invoking login from our hook. But there's a problem; the old login returned a promise. Ours doesn't need to do that. So refactor handleLogin. Maybe something like this:

```
function handleLogin(): void {
  login(username, password);
}
```

And there's the major benefit of custom hooks; you simplify your component logic as you centralize their implementation!

21. Run and test using a known good username/password. Did you get the toast notification? If so, you're logged in.

Or are you? Did you notice the major issue? The NavBar should be greeting us by name but it's not because the user in NavBar is not the same instance as the user in Login. Same with isAuthenticated in fact.

Putting the hook's data into context

We can fix this with React's context. We must have the value provided by the context be the one that everyone uses.

22. Create a new component called AuthProvider in AuthProvider.tsx

```
import { createContext, ReactElement, ReactNode } from "react"
import { AuthHookType, useAuth } from "../hooks/useAuth"
const initialValue: AuthHookType = {
  isAuthenticated: false,
  user: undefined,
  login: (u, p) => undefined,
  logout: () => undefined,
}
export const AuthContext = createContext<AuthHookType>(initialValue);
export const AuthProvider = ({ children }: { children: ReactNode }): ReactElement => {
  const auth = useAuth();
  return (
    <AuthContext.Provider value={auth}>
      {children}
    </AuthContext.Provider>
  )
}
```

This will become our provider that provides the same instance of the auth hook to everyone. And that means that everything needs to be wrapped in this new component.

23. Edit App.tsx. Wrap the entire App in this provider. Change this ...

```
<UserContext.Provider value={user}>
```

to this ...

```
<AuthProvider>
```

24. Edit Login.tsx. Change this...

```
const { login } = useAuth();
```

to this ...

```
const { login } = useContext(AuthContext);
```

25. Do the same change for NavBar.tsx. Change this ...

```
const { user, isAuthenticated } = useAuth();
```

to this ...

```
const { user, isAuthenticated } = useContext(AuthContext);
```

26. Run and test. They should be in sync.

Now they're both sharing the same provided instances of user, isAuthenticated, and login. But what about logging out?

Making logout work

27. If you hadn't already, please click the log out option in the nav menu. It does nothing.

28. Edit NavBar.tsx. Without peeking at the answer below, see if you can see why it doesn't work. What would you do to fix it?

It's using the local setUser(undefined) instead of logging out in the Context. It should use our logout() function from the useAuth hook in context.

29. Change setUser(undefined) to use our logout. (Hint: include logout in the destructured context object) and invoke it on the click event handler.

30. Run and test. You should be able to login and logout.

Making Cart work

31. Log in with any user who has a credit card on file. (Hint: Test User does. They have a username of "me" and a password of "pass").

32. Navigate to Check out/Cart. Look at credit card field. Is it populated?

Cart.tsx isn't using our context so it doesn't see the logged in user.

33. Make Cart work with our context. Now that we've done it a few times you should be able to figure it out with fewer instructions.

34. Run and test. When you can log in as a user with a credit card, and see that card, you know you've got it right.

Making Orders work

35. Make sure you're logged in still. From the top nav bar, navigate to Past orders.

No matter how many users you try, you'll see no orders because OrderSummary.tsx isn't sharing useAuth with the other components.

36. As you did above, make OrderSummary.tsx share the user object from context.

37. Run and test. When a logged-in user can see their orders, you did it right.

You are officially finished with this entire lab. Congratulations! But if you finished early, here are some bonus exercises.

Bonus! Cleaning up

38. Register.tsx is not sharing context with the rest of the components. Make sure it is.

39. Note that we no longer need the UserContext anywhere in our project. If it is still around in any components, you may delete it.

40. No longer need user or setUser. Look through any places where you've eliminated the need for these but you haven't cleaned them up yet. Go ahead and remove that dead code, especially in props.