Isolating Implementation Details

In this lesson, we'll centralize the implementation for the user state and provider in one place.

WE'LL COVER THE FOLLOWING

The solution highlighted in the previous lesson works but not without some caveats.

A better solution will be to centralize the logic for the user state and `Provider` in one place. This is a pretty common practice. Let me show you what I mean.

UserContext.js

Instead of having the `Root` component manage the state for `loggedInUser`, we will create a new file called `UserContext.js`.

This file will have the related logic for updating `loggedInUser`. It will also expose a context `Provider` and `Consumer` to make sure `loggedInUser` and any updater functions are accessible from anywhere in the component tree.

This sort of modularity becomes important when you have many different context objects. For example, you could have a `ThemeContext` and `LanguageContext` object in the same app.

Extracting these into separate files and components proves more manageable and effective over time.

Consider the following:

```
// UserContext.js
import React, { createContext, Component } from 'react'
import { USER } from '../api'
```

```
const { Provider, Consumer } = createContext()
class UserProvider extends Component {
  state = {

    loggedInUser: null
  }
  handleLogin = evt => {
    evt.preventDefault()
    this.setState({
      loggedInUser: USER
    })
  }

  render () {
    const { loggedInUser } = this.state
    return (
      <Provider
        value={{
          user: loggedInUser,
          handleLogin: this.handleLogin
        }}
      >
        {this.props.children}
      </Provider>
    )
  }
}
export { UserProvider as default, Consumer as UserConsumer }
```

This represents the content of the new `context/UserContext.js` file. The logic previously handled by the `Root` component has been moved here.

Note how it handles every logic regarding the `loggedInUser` state value and passes the needed values to `children` via a `Provider`.

```
...
<Provider
     value={{
       user: loggedInUser,
       handleLogin: this.handleLogin
      }}
     >
      {this.props.children}
</Provider>
...
```

In this case, the `value` prop is an object with the `user` value and function, `handleLogin`, to update it.

Also, note that the `Provider` and `Consumer` are both exported. This makes it easy to consume the values from any components in the application.

```
export { UserProvider as default, Consumer as UserConsumer }
```

With this decoupled setup, you can use the `loggedInUser` state value anywhere in your component tree, and have it updated from anywhere in your component tree as well.

Here's an example of using this in the `Greeting` component:

```
import React from 'react'
import { UserConsumer } from '../context/UserContext'
const Greeting = () => {
  return (
    <UserConsumer>
      {({ user }) => <p>Welcome, {user.name}! </p>}
    </UserConsumer>
  )
}
export default Greeting
```

How easy.

Removing the Prop References

Now, I've taken the effort to delete every reference to `loggedInUser` where the prop had to be passed down needlessly. Thanks, Context!

For example:

```
// before
const User = ({ loggedInUser, profilePic }) => {
  return (
    <div>
      <img src={profilePic} alt='user' />
      <Greeting loggedInUser={loggedInUser} />
    </div>
  )
}
// after: Greeting consumes UserContext
const User = ({profilePic }) => {
  return (
    <div>
      <img src={profilePic} alt='user' />
      <Greeting />
    </div>
  )
}
export default User
```

Be sure to look in the accompanying code folder for the final implementation.

```
export const USER = {
  name: 'June',
  totalAmount: 2500701
}
```

In the next lesson, we'll discuss the withdrawal method of the mini-bank application by updating the Context values.