Pokedex Hooks: Phase 3

At this point, you've converted a class based project with the `useState` and `useEffect` hooks and converted a Redux-based project with the `useSelector` and `useDispatch` hooks. Now you'll work on of a project which stores it's state in the `<App>` component and threads the state down through multiple components by using `props`. You'll be adding a `Context` in an `AppWithContext` component and will use `useContext` to access this context in all the components that need access.

Your instructor will supply you with the phase 3 starter project.

The backend

This application talks to the very same pokedex backend app you've been using for the first two phases. You can leave the backend running if you already have it up and going.

A note on your phase 1 and 2 project

Remember to shutdown your Redux based Pokedex app before you start this one or you may get an error about port 3000 being already in use.

Exploring this application

If you look around this application you'll see it is mostly the same as the other pokedex apps we've been using. But it does have a few differences.

The first major difference is that there's no Redux here. This app is simply storing it's state in the `<App>` component and prop threading it down to every component under it that needs it.

Also you may notice that the fetch calls have been moved into modules in the `fetches` folder. This is a good way to keep your `useEffect` hooks in the components small.

If you look at the `<App>` component you'll see it uses several `useState()` hooks, including one for the `token` and one for the `pokemon` list. It then passes those state variables and state functions down to lots of difference components, including `<LoginPanel>`, `<PokemonBrowser>`, etc.

What we want to do is put a context at the top level of our component tree, and get all of these components, including `<App>` to use the context instead.

Using the `useContext` hook to manage application state

Previously, you created a Redux cycle to pass Pokedex information through your components. Now you'll manage your application's global information by using React Context instead! Remember that you still generate context with the `createContext` function, just as you would for class components. You also still use `<Context.Provider>` components to set the `value` of your context object.

Think of how you would make use of the `useContext` hook instead of Redux's `connect()` function to pass slices of state as well as functions to update the global state.

Providing context

Begin by creating a `PokemonContext` with the `createContext` function from React. If you don't remember how to do this, check the Context documentation.

Then you'll need to set the context value with a `<Context.Provider>` component by making a wrapper Provider component for the `App` component. Create an `AppWithContext` component as the wrapper component for `App`.

Make the following `useState` hooks and then add them to the `Context.Provider` value attribute:

- pokemon - This defaults to an empty array and will hold our pokemon
- setPokemon - This is the function to update our pokemon
- selectedPokemon - This holds the currently selected pokemon
- setSelectedPokemon - This is to update the currently selected pokemon
- token - This default to an empty string and will hold our login token
- setToken - This is the function to update out token

  Hint: It might be cleaner to build a small POJO to hold these instead of trying to cram all of these into the `value` attribute.

Remember to import the `App` component and render it inside of the `<Context.Provider>`!

Rendering your wrapper component

After you have set up the wrapper component, make sure to replace the `App` that is rendered in your `index.js` file with your new `AppWithContext` wrapper component. Now it's time to change your application from being a state-based application to a context-based application.

Consuming context with the useContext hook

Your application's consuming components should access the `PokemonContext` through using the `useContext` hook. Feel free to reference the [Hooks API Reference] to revisit the documentation on the `useContext` hook. As a reminder, the `useContext` hook replaces the `static contextType` property of class components:

useContext

```
// Receive access to context with React Hooks in function components:
const context = useContext(PokemonContext);
```

```
// Access context with the `useContext` hook:
context
```

Now it's time to set up how your application components *consume* the `PokemonContext`!

Note on debugging your context

Feel free to console log the `context` in any component you are accessing the `PokemonContext`. You can also use the `debugger` command to pause execution right after you do `useContext` to view what's currently stored in the context.

This way upon the mounting of a component, you have a general sense of the `context` object the component is receiving. Since you'll be using Hooks and Context to manage your user authentication, you may need to clear your `localStorage` items to reset your application to allow for future testing and debugging. As a reminder, you can go to the `Application` tab of your developer tools to find a `Storage` section with your `Local Storage` items. There you can right click to delete all items stored in `localStorage`.

Refactor the components

You'll want to refactor the following components to use the context:

- `App`
- `PokemonBrowser`
- `PokemonDetail`
- `PokemonForm`
- `LoginPanel`
- `LogoutButton`

As with the other refactoring you've done today, do one component at a time.

Once you add the code to pull the information out of the context, make sure you remove any props that conflict, and also change the parent component to not pass down those props anymore!

> Tip, when you use `useContext` it might be helpful to destructure only the bit you need from the context, so for instance, if you only needed the token, you might do this : `const { token } = useContext(PokemonContext);` Doing it this way might mean you have to change less code in the components, since the context variables will be the same as the ones the app previously destructured from `props`

App

Take a moment to compare the code that currently lives in your `AppWithContext` with the code that lives in you `App` component. Notice how there is a lot of duplicated logic. This is because you `App` was the main component managing your application's state-based information. Now that you have moved all the logic to your `AppWithContext` component, you can refactor your `App` component to simply use the `useContext` hook to pass the `token` value to the `<LoginPanel>` it renders. You can also remove all other props passed through the routes. Your refactored `App` component should look something like this:

```
// App.js
const App = () =>  {
  const { token, setToken } = useContext(PokemonContext);

  useEffect(() => {
    (async() => {
      const localToken = window.localStorage.getItem("token");
      if (localToken) {
        setToken(localToken);
      }
    })();
  },[setToken]);

  const needLogin = !token;

  return (
    <BrowserRouter>
      <Switch>
        <Route
          path="/login"
          render={(props) => (
            <LoginPanel {...props}/>
          )}
        />
        <PrivateRoute
          path="/"
          exact={true}
          needLogin={needLogin}>
          <PokemonBrowser
          />
        </PrivateRoute>
        <PrivateRoute
          path="/pokemon/:pokemonId"
          exact={true}
          needLogin={needLogin}>
          <PokemonBrowser
          />
        </PrivateRoute>
      </Switch>
    </BrowserRouter>
  );
}

export default App;
```

You'll notice that the `App` component doesn't even need the `pokemon` and `setPokemon` anymore so we aren't destructuring them from the context anymore! This is because inside of `<PokemonBrowser>` it'll use the context to get the `pokemon` and `setPokemon`. No more prop threading!

LoginPanel

Let's begin by refactoring your `LoginPanel` component! The `LoginPanel` should access the context's `setToken` function `token` value.

```
const {token, setToken} = useContext(PokemonContext);
console.log(authToken);
```

The rest of the components

Now go ahead and refactor the rest of the component using what you've learned. The process should be very similar. Follow these steps.

1. Determine what props the component is using.
2. If those props are stored in the context, remove them from `props` and add a `useContext` call instead.
3. Visit the parent component to remove any props attributes being passed down.

After you have finished refactoring your context based hooks application, compare it to the earlier redux solution. Using Redux instead of Context results in a lot of boilerplate code in your application. React 16 revamped the Context API and deemed the `useContext` hook as a basic hook to improve React's built-in state management. For simple applications `Context` can be a good option without having to rely on Redux, but you do lose the nice Redux debugging tools to inspect the state of your application. In your career you may encounter apps using either approach or even applications that use a combination of both.