Pokedex Hooks: Phase 2

As you might remember from the Redux-based Pokedex project, implementing Redux results in a lot of boilerplate code. Using Redux hooks can help clean up and get rid of a lot of boilerplate code. In this phase you will refactor the Redux-based project to use React hooks and implement Redux hooks!

Using Redux hooks to manage application state

In this phase, you'll be refactoring all your component files to use Redux hooks instead of the `mapStateToProps`, `mapDispatchToProps`, and Redux `connect` functions. Just like in phase 1, you might hit bugs and break your application while refactoring your application's components. Make sure to test that your refactored code is working before moving on to refactor the next component. As a general overview, you'll be refactoring the code for the following components:

1. `LogoutButton`
2. `LoginPanel`
3. `PokemonDetail`
4. `PokemonForm`
5. `PokemonBrowser`

Refactoring to use `useDispatch` and `useSelector`

You'll need to do some refactoring so that your component doesn't receive any props. Instead of receiving slices of state and dispatchable action functions as props from the `connect` wrapper, you will use the `useSelector` hook to access a slice a state from within the component and the `useDispatch` hook to dispatch actions from within the component.

Take a look at the `LogoutButton` component as an example.

Based on the `mapStateToProps` and `mapDispatchToProps` functions in the `LogoutButton.js` file, you can tell that the component is accessing Redux by receiving `loggedOut` and `logout` props.

```
// We'll convert this into a useSelector hook
const mapStateToProps = state => {
  return {
    loggedOut: !state.authentication.token,
  };
};

// We'll convert this into a useDispatch hook
const mapDispatchToProps = dispatch => {
  return {
    logout: () => dispatch(logout()),
  };
};

export default connect(mapStateToProps, mapDispatchToProps)(LogoutButton);
```

Take a moment to import the `useSelector` and `useDispatch` from the Redux library into the file.

```
import { useDispatch, useSelector } from 'react-redux';
```

Now you'll use Redux hooks within the `LoginButton` component so that you can remove the `mapStateToProps`, `mapDispatchToProps`, and `connect` functions!

Refactor `mapStateToProps` to `useSelector`

Instead of receiving a `loggedOut` prop, you'll use the `useSelector` hook to access the state's `authentication.token`.

Add the `useSelector` to the top of your component.

```
const LogoutButton = () => {
  const loggedOut = useSelector(state => !state.authentication.token);

  // CODE SHORTENED FOR BREVITY
};
```

> Feel free to visit the Redux Hooks documentation to view [`useSelector` examples](#).

Then inside the component you can convert the `props.loggedOut` to simply use the new variable `loggedOut`

```
const LogoutButton = () => {
  const loggedOut = useSelector(state => !state.authentication.token);
  if (loggedOut) {
    return <Redirect to="/login" />;
  }

  return (
    <div id="logout-button-holder">
      <button onClick={handleClick}>Logout</button>
    </div>
  );
};
```

Refactor `mapDispatchToProps` to `useDispatch`

Notice how the `logout` thunk action creator function has already been imported into your `LogoutButton.js` file. You'll use `useDispatch` hook to return a reference to the `dispatch` function from the Redux store:

```
const dispatch = useDispatch();
```

Then you can use the `dispatch` function to dispatch the `logout` function directly inside the `handleClick` function instead of doing it in `mapDispatchToProps`.

So much simpler!

```
const LogoutButton = () => {
  const loggedOut = useSelector(state => !state.authentication.token);
  const dispatch = useDispatch();
  const handleClick = () => dispatch(logout());

  // CODE SHORTENED FOR BREVITY
};
```

> Feel free to visit the Redux Hooks documentation to view `useDispatch` examples.

Removing `mapStateToProps`, `mapDispatchToProps`, and `connect`

Now that we aren't referencing `props.logout` and `props.loggedOut`, you can remove all the old Redux boilerplate code.

Lastly, you'll want to remove the `mapStateToProps` and `mapDispatchToProps` functions from the file, and replace the `connect` function to an export statement that exports the `LoginButton` component by default:

```
export default LoginButton;
```

Now that you've gone over how to refactor your `LogoutButton` component, follow the same pattern to implement Redux hooks into your `LoginPanel`, `PokemonDetail`, `PokemonForm`, and `PokemonBrowser` components.

Router hooks: `useParams`

Notice the references to the React Router `match` prop accessed in your `PokemonBrowser` and `PokemonDetail` components? Instead of having your component take in a `match` prop to access the route parameters, you'll implement the `useParams` prop and use object destructuring to access the `pokemonId` parameter in the `PokemonBrowser` component and the `id` parameter in the `PokemonDetail` component. Feel free to visit React Router documentation to view examples of using the `useParams` hook.

Once you have finished refactoring, take a moment to commit your changes:

```
git add .
git commit -m "Refactor app to implement redux and react router hooks"
```

As you can see, refactoring to Functional components with Hooks makes your code much smaller and easier to understand! Going forward with your coding career you'll probably want to always use hooks instead of class based components, but it's good that you know how to read class based components and you'll be able to refactor class based component code to functional hooks components whereever you find them!

Next Phase... `Context`

Now that you have practiced refactoring your application to implement Redux hooks, it's time to work on a Context-based project utilizing React's `useContext` hook! However we can't very well do this in our Redux project since it uses Redux to store all of the state instead of `Context`! So we'll be starting with an alternative Pokedex project which uses `Context` instead.