Pokedex Hooks: Phase 1

In today's project you will refactor class-based components that make use of lifecycle methods to function components that make use of React Hooks! The project we will use is the Pokedex Redux project you just completed. However you will be starting with the official solution instead of your code base.

In this project, you'll build your application with React and Redux hooks! You will implement the:

- `useState` hook to manage a component's state
- `useEffect` hook to manage a component's side effect operations
- `useDispatch` hook to dispatch actions from with a component file
- `useSelector` hook access slices of state from the Redux store
- `useContext` hook to manage your application with Context instead of Redux
- `useParams` hook to grab the parameters from React Router

Phase 1: Refactor to use React Hooks

In this phase you'll refactor the class based components into functional components using these two React Hooks:

- `useState`
- `useEffect`

First you'll need the backend for the Pokedex application. If you don't already have it running from your previous work, take a moment to clone it from https://github.com/appacademy-starters/pokedex-backend and get it set up and running. Remember you may need to get the database created and all the migrations and seeds running for the backend, if you haven't already done so.

The API for the backend is also documented in the repository's README.

Once you have that up and running, you'll begin working out of the official solution from the Pokemon Redux project.

Throughout today, you'll work on refactoring each class component in the application to be a function component that makes use of React Hooks!

Explore the reference application

As you might remember, your current application comprises of the following components:

- `App`: Does the browser routing and top-level fetches of data to draw the data
- `LoginPanel`: Shows the login panel
- `PokemonBrowser`: The browser that draws the list on the left after logging in and has a route to the `PokemonDetail` when the route matches "/pokemon/:id"
- `PokemonDetail`: Makes a fetch to the API on mount and update to load the details of the selected Pokemon
- `PokemonForm`: Renders a form in place of the `PokemonDetail` component to add a Pokemon, makes a fetch call to get the available types of the pokemon and then makes a fetch call to post the form data to the backend.

Refactor components

As you're refactoring your application's components, you'll most likely hit bugs and break your application. While you're refactoring each component, make sure to test that your refactored code is working before moving on to refactor the next component. As a general guideline, you should refactor each component from the lowest, most nested component up to the top-most parent:

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

Once a component is refactored, your app should behave exactly as it always did. Remember functional components with hooks is just an alternative to class based components with state.

You'll update how each component sets its default state by using the `useState` hook. You'll also refactor the lifecycle methods of each component into side effect operations managed by the `useEffect` hook. At the end of this exercise, you should have a good understanding of how to use the basic `useState` and `useEffect` hooks to write function components with side effect operations.

As a reminder use this as a guide for what you need to convert.

Class based	Functional hooks based
any refernce to `this.state`	`useState()`
`componentDidMount`, `componentWillUnmount`	`useEffect()`

If you can't remember how to use the new hooks, try looking at the official documentation:

- Using the State Hook
- Using the Effect Hook
- Hooks API Reference

A note on Redux

As this app uses Redux, you'll see each component uses `connect`, `mapStateToProps`, and `mapDispatchToProps`. While we will be refactoring our components to use the Redux hooks in the next phase, for now you are *only* converting to use the React hooks `useState` and `useEffect`. This means you can leave the `connect` wrapper in place. The connect functions works equally well with class based or functional components. You will just need to swap `this.props` for `props` accordingly.

Finish up

Once you have finished refactoring, take a moment to commit your changes to the main branch of your project:

```
git add .
git commit -m "Refactor app to implement React hooks"
```

In the next section we'll be refactoring the Redux part of this application, by utilizing the Redux specific hooks `useSelector` and `useDispatch`.