- ⏱️ 3 hours

Star Trek Project

In this project, you will make enhancements to an online trading card application. This application allows a user to select cards to build a deck for game play or trading in the future.

The original developers set up the navigation (using `react-router-dom`) and created the two primary screen layouts (using function components). Now, they need your help with capturing, storing, and displaying the user's selection in their deck.

To begin

- Install the packages (`npm install`)
- Run the application and explore what's there (`npm start`)
  - Notice that nothing happens when you click `Buy` buttons on the home page
  - See that the sample friend's deck has cards, but the user's deck does not
  - These are the gaps you will close!
- Commit this version before you make changes to a *GitHub* repo in your account
- Look through the code for the components, including UI and react-router setup

In particular, you should be looking for functionality gaps, such as the `const` definitions at the start of several of the components. Write these down as you'll need the information later to provide those values through the context you create.

Tip: Test early and often! Use your debugging skills to ensure each change is working before making additional changes. An important skill for software developers is knowing when to ask for assistance!

Phase 1 - Set up the framework

You'll be using *React Context* to track the purchases. In this initial phase, you will set up the context by creating a new function component called `AppContextProvider` with all the required pieces (importing React, writing the function, and exporting the function).

As you may recall, one of the Object-Oriented Programming (OOP) concepts is *encapsulation*, and a good principle to follow is grouping together code that changes for the same reason / shares a single responsibility. So that's the approach you're going to walk through now. Specifically, this means putting the context and provider together with the variables shared through the context (a.k.a., `value`), as well as the functions that update these variables.

Define the ***context*** and its ***provider***

Create a folder in `src` called `context`. Inside of the `context` folder, create a file called `AppContext.js`. In this file, define the `AppContextProvider` function component.

The props of the function will include one variable named `children`. This variable is built into *React*. It holds a collection of JSX content passed to the component between its opening and closing tags (which you'll do in the App in just a moment). In case you forgot, the easiest way to define a function with named props is by deconstructing the props object into its individual attributes. In other words, `const AppContextProvider = ({children}) => {}`.

Above the function (and outside of it), declare a new constant (e.g., `AppContext`) that creates the context. Export this constant. (Later, you'll connect hooks to it in the UI components.)

Inside the function, return the provider for the context you created and place the children between the opening and closing JSX tags. For example:

```
return (
  <AppContext.Provider>
    {children}
  </AppContext.Provider>
);
```

Include one variable in the **`value`** as a starting point

Finally (for the moment), set the `value` prop on the component to an object. In this object, you can start with only the `cards` array. The original developers left you *mock data* to use for this purpose, so you can `import` the appropriate constant from the class in `mockdata`.

In case you're stuck, here are a few hints...

```
import { initialCards } from "../mockdata/CardData";

// component function and other stuff

return (
  <AppContext.Provider value={{
    cards: initialCards
  }}>
    {children}
  </AppContext.Provider>
);
```

Wrap the application in the ***provider***

Open *App.js*. Import the new provider you just made. Place it in the application such that all components that need to use the context are wrapped inside of it. In other words, those components are children (or grandchildren, etc.) of the context provider.
Tip: This is a good time to commit an update to `git` and push up to GitHub.

Phase 2 - Adding values to the *context provider*

Look back at the notes you took while you were reviewing the application. Which variables did you find that were incomplete in the components?

Hopefully, your list matches this one:

- cards
- decks
- inventory
- buyCard

If your notes do not include details on the variable type and structure of each of these, please review the components again to gather this information. Take your time! Being thorough now will save you time and effort and reduce your frustration later.

Now, think through each variable and determine which ones change together all or most of the time. When working with global context, it is important to keep the number of state variables to a minimum. Updating 3 variables can cause 3 rerenders of ALL children, whether or not they use the context.

Hint: The previous developers informed you that when a card is bought, it will be put into the user's deck, and removed from the inventory (meaning the count gets reduced).

Connecting ***state*** variable(s) to the ***context provider***

Back in `AppContextProvider`, you can create a state variable (using the `useState` hook) to hold the decks and the inventory (e.g. `applicationState`). The initial values you should get from the mock data (which could someday be replaced with fetch calls to a backend API, but that is beyond the scope of this project).
Since the components want to access the `decks` and `inventory` variables separately, you'll want to use decomposition to incorporate them into the `value` of the context provider.

```
<AppContext.Provider value={{
  ...applicationState,
  cards: initialCards
}}>
```

Including a function variable in the ***context provider***

The fourth variable that needs to go in the `value` of the *context* is the `buyCard` function. In order to reduce the number of rerenders, you'll want to utilize the useCallback hook when defining this function (e.g., `buyCardForPlayer`).
In your analysis, you may have noticed the function takes 1 argument--the `cardId`--so be sure to include it.
For the dependencies of `useCallback`, you'll want to include the state variable you made (i.e., `applicationState`), since the function will need to take values out of it in order to update them.
For now, it's okay to do the equivalent of "nothing" inside the function. If you would like to clean up any React warnings (like the one stating that the function associated with the `useState` hook is defined but not used), then you can call the state function passing in the state variable (e.g., `updateApplicationState(applicationState)`). You may also want to include a `console.log` statement so you can easily tell when this is being called. (It is NOT being called yet, but it will be very soon.)
The last step here is to add the function to the `value` of the *context provider*.

```
    <AppContext.Provider value={{
      ...applicationState,
      cards: initialCards,
      buyCard: buyCardForPlayer
    }}>
```

There's nothing new to test, but it's always a good idea to save and make sure there are no errors or warnings in the browser console.

Phase 3 - Buy cards

Now, you can start using your context! Begin in the `Store` component by importing the context you created and the `useContext` hook. Then, replace the `buyCard` constant with a deconstructed object that utilizes the `useContext` hook.

```
const { buyCard } = useContext(AppContext);
```

When you test in the browser, you should be able to see the output from the `console.log` you added to the function in `AppContextProvider`.

Updating inventory count

Finally, it's time to get into the heart of the algorithm which modifies the data (`value`) in the *context*.

FYI, much of the time, *React* development goes this way. You begin by setting up JSX for the user interface (which other developers did for you on this project!), then add hooks for state, context, and so on. At some point you will find you need to write "classic" JavaScript to implement data updates, business logic, or other feature functionality. This mix of different kinds of work is what attracts many front-end developers to *React*. You are blessed to be learning both backend and frontend, so you'll have many choices for your career path!
Before you get too deep into the code for updating the state, you may want to set up some debugging to make it easier to verify that your code is working as expected.
A `useEffect` hook is particularly useful in this situation.
Since you have to pick some place to start, it might as well be the inventory. Place any and all `useEffect` calls outside of any functions (probably between the function definition you added in phase 2 (i.e., `buyCard`) and the `return` of the context provider).

```
  // Useful for debugging inventory updates
  useEffect(() => {
    console.log("INVENTORY UPDATED", applicationState.inventory)
  }, [applicationState.inventory]);
```

Now, inside the body of your `buyCard` function, add code for the following features

- Don't allow the inventory to go below zero
- Make a copy of the current inventory object
  - This is an important approach because updating the current value could cause unanticipated re-renders!
- Decrement the inventory value for the card that was clicked
- Modify the call to the state objects update function to provide a new object that has the new inventory (while leaving the decks unchanged).

Here's a bit of a hint on that last one...

```
updateApplicationState({
  decks: applicationState.decks,
  inventory: newInventory,
});
```

When you run and test this in the browser, be sure to click the same card enough times to drive the value to zero so you can verify the first bullet above is covered.

Show updated inventory in the store

Jump back to the `Store` component to replace both `inventory` and `cards` constants with the ones from the context. Don't worry if you find this quite simple; that just means you're probably doing it right!

Test in the browser and verify the inventory numbers change correctly, and do NOT go below zero. Also, check for warnings (like unused imports), and address those as appropriate (deleting the unused imports).

Tip: Once you are seeing both the inventory updating properly, you know it's a good time to commit an update to `git` and push up to GitHub.

Phase 4 - Display player's deck

Now, you can work on the player's deck. Remember, that's the first one in the `decks` array.
If you'd like, you can connect the UI components to the context as the first step. That means edits in both `Navbar` and `Deck`.

You shouldn't see any visible changes since the default values are the same and nothing is updating the decks after initialization. Nonetheless, verifying there are no code errors is valuable.

Updating player's deck

The exact implementation is left up to you. You're an experienced developer who will be able to figure this out. Here's one approach to help get you started if you get stuck.

- Add a `useEffect` for logging updates to the decks (similar to the inventory one above).
- Copy the current decks into a new array (to avoid inadvertent rerenders).
- Set a variable to the player's deck inside the new array for easy reference. (Remember that the player's deck will always be first.)
- Find the card object in the cards list.
    - Tip: You can make a constant for this at the beginning of your function component (set to `initialCards`, for now). This will allow you to set it as a dependency of the `useCallback`, as well as clean up the first `value` you set. While it doesn't affect this particular project, the right thing to do is imagine you will soon be switching the mock data to real data from a backend service.
- Merge the new card into a copy of the list of cards in the player's deck and store the result as the (new) card list in the player's deck
- Modify the object sent to the state update function to include the updated decks array.

At any point along the way, you can and should test and use your debugging skills to solve any challenges. As you've probably heard by now, this is a key skill and common practice in development.

Tip: You'll definitely want to commit an update to `git` and push up to GitHub once all the pieces are working.

Congratulations! You finished the core features of this project!

Bonus A

There is at least one cool enhancement you can make to your implementation. Specifically, name your own hook!

The short explanation is that you'll move the use of `useContext` into the `AppContext` file with your context provider. Here are the detailed steps.
Edit `AppContext`.

- Remove `export` from the constant for the context `const AppContext = React.createContext({});`
- Immediately following, make a new, exported constant with the custom name for your hook. Generally, hooks start with the `use` prefix. `export const useAppContext = () => useContext(AppContext);`
- Go to the components and replace useContext with your custom hook, e.g., `const { decks } = useAppContext();`

Important Note

You'll need to declare the custom hook as a function so that it runs inside each component which uses it. If you forget this, you'll get an error that starts something like this:

```
Error: Invalid hook call. Hooks can only be called inside of the
body of a
function component.
```

Bonus B

You've probably noticed that when the browser refreshes, all the cards in the player's deck return to the store. If you'd like them to be persisted, then you can use localStorage to save and restore.

Remember to use the mock data for the initial state when the local storage is not present (yet). You can reset the local storage in *Chrome dev tools* on the "Application" tab under "Local Storage" on the left.

Hint: All of this can be done within AppContextProvider.

Bonus C

If you implemented `localStorage`, you'll soon be wanting a way to return cards to the store (otherwise known as deleting them from the player's deck). Go ahead and implement this now.

Hints:

- *A new function in the context may prove useful.*
- *You may find it helpful to have a button the player can click to return a card. You can use the `Buy` button from the `Store` as a model. Remember that the button should appear only when displaying cards from the player's deck!*

Did you find this lesson helpful?