FSW-130 Learn-It Notes

Wk1

* Context is not a state management system by itself.
* Context provides a way to pass data through the component tree without having to pass props down manually at every level.
* In a typical React application, data is passed top-down (parent to child) via props, but this can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application.
* Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.
* By using Context, we can avoid passing props through intermediate elements.
* Context is primarily used when some data needs to be accessible by many components at different nesting levels. Apply it sparingly because it makes component resuse more difficult.
* If you only want to avoid passing some props through many levels, component composition is often a simpler solution than context.
* Component composition is sufficient for many cases when you need to decouple a child from its immediate parents. You can take it even further with render props if the child needs to communicate with the parent before rendering.
* Sometimes the same data needs to be accessible by many components in the tree, and at different nesting levels. Context lets you “broadcast” such data, and changes to it, to all components below. Common examples where using Context might be simpler than the alternatives include managing the current locale, theme, or a data cache.
* The method createContext creates a Context object. When React renders a component that subscribes to this Context object it will read the current Context value from the closest matching Provider above it in the tree.
  + const MyContext = React.createContext(defaultValue);
* React provides a method called createContext which will generate a composite component made up of two smaller components: Provider and Consumer. Provider is responsible for holding data, while Consumer is responsible for accessing it.
  + //index.js

Export const {Consumer , Provider} = React.createContext()

* Handling authentication, themes, and API data can all be maintained by independent Contexts. To use them simultaneusly simply stack the providers like this:
  + <AuthProvider>

<ThemeProvider>

<App/>

</ThemeProvider>

</AuthProvider>

* The defaultValue argument is only used when a component does not have a matching Provider above it in the tree. This can be helpful for testing components in isolation without wrapping them. Note: passing undefined as a Provider value does not cause consuming components to use defaultValue.
  + const MyContext = React.createContext(defaultValue);
* Every Context object comes with a Provider React component that allows consuming components to subscribe to Context changes.
  + <MyContext.Provider value={/\*some value \*/}>
* Context provides a way to share values between components without having to explicitly pass a prop through every level of the tree.
* Accepts a value prop to be passed to consuming components that are descendants of this Provider. One provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.
  + <MyContext.Provider value={/\*some value \*/}>
* All consumers that are descendants of a Provider will re-render whenever the Provider’s value prop changes.
* The propagation from Provider to its descendant consumers is not subject to the shouldComponentUpdate method, so the consumer is updated even when an ancestor component bails out of the update.

Wk2

* Context provides a way to pass data through the component tree without having to pass props down manually at every level.
* Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.
* The contextType property on a class can be assigned a Context object created by React.createContext(). This lets you consume the nearest current value of that Context type using this.context. You can reference this in any of the lifecycle methods including the render function.
* React Context is a very accessible and powerful API for managing state across multiple components. Separate chunks of state can be maintained in individual components and then exposed through the Provider and Consumer.
* A React component that subscribes to Context changes. This lets you subscribe to a Context within a function component.
* Requires a function as a child. The function receives the current Context value and returns a React node.
* The value argument passed to the function will be equal to the value prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the value argument will be equal to the defaultValue that was passed to createContext().
* Context object accepts a displayName string property. React DevTools uses this string to determine what to display for the context. For example, the following component will appear as MyDisplayName in the DevTools:
  + Cont MyContext = React.createContext;

MyContext.diplayNmae = ‘MyDisplayName’;

<MyContext.Provider> // “MyDisplayName” in DevTools

<MyContext.Consumer> // “MyDisplayName” in DevTools

* Because context uses reference identity to determine when to re-render, there are some gotchas that could trigger unintentional renders in consumers when a provider’s parent re-renders. For example, the code below will re-render all consumers every time the Provider re-renders because a new object is always created for value:
* Redux, the state management tool we will be using, is actually built upon/with the Context API.
* The Context API React provides an internal solution for passing state to where it is needed and avoids the possibility of prop drilling.
* Using a mix of local state and React Context can help you manage state well in any React application.
* 99% of the time that you're going to be creating and using context in your application, you want your context consumers (those using useContext) to be rendered within a provider which can provide a useful value.
* There are situations where default values are useful, but most of the time they're not necessary or useful. The React docs suggest that providing a default value "can be helpful in testing components in isolation without wrapping them." While it's true that it allows you to do this, it is instead better than wrapping your components with the necessary context.
* You shouldn't be reaching for context to solve every state sharing problem that crosses your desk. -----True-----
* Context does not have to be global to the whole app, but can be applied to one part of your tree.
* You can (and probably should) have multiple logically separated contexts in your app.

Wk3