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

* By now, you've likely been using React and state, setState, and props to pass data around in your application. For many simple apps, this works great. However, once you start increasing the complexity of your application or turn it into a SPA (single page app), you'll find yourself passing data down several nested component layers and increasing the ancestry-level of your state just to be able to access data everywhere in your application.
* Redux is a(n) state management tool.
* Redux takes the data component of React and turns the idea of compartmentalization on its head - instead of only having components in charge of state (whether that state is important to just that component or the app as a whole), Redux creates a global "store" (essentially state) that is accessible to the whole application.
* Facebook pioneered the concept of Flux as a way to handle data changes in an application. Redux took the idea of Flux and added a bit to it, giving it some extra power.
* Flux is a pattern for handling data in your application.
* Redux takes the data component of React and turns the idea of compartmentalization on its head - instead of only having components in charge of state, Redux creates a global "store" (essentially state) that is accessible to the whole application.
* The Redux DevTools make it easy to trace when, where, why, and how your application's state changed.
* Facebook decided to try a different kind of architecture, where the data flows in one direction — only one direction — and when you need to insert new data, the flow starts all over again at the beginning. They called their architecture Flux.
* Redux's architecture lets you log changes, use "time-travel debugging", and even send complete error reports to a server.
* Centralizing your application's state and logic within Redux's Predictable State Container enables powerful capabilities like undo/redo, state persistence, and much more.
* Redux helps you write applications that behave consistently, run in different environments (client, server, and native), and are easy to test.
* Redux works with any UI layer, and has a large ecosystem of addons to fit your needs.
* Redux itself is small and unopinionated. But there is also a separate addon package called Redux Toolkit, which includes some opinionated defaults that help you use Redux more effectively. It's the official recommended approach for writing Redux logic.
* Redux Toolkit includes utilities that help simplify many common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "slices" of state at once.
* In Redux the whole state of your app is stored in an object tree inside a single store.
* In Redux the only way to change the state tree is to emit a(n) action, an object describing what happened.
* In Redux to specify how the actions transform the state tree, you write pure reducers.

Wk4

* In Redux, instead of mutating the state directly, you specify the mutations you want to happen with plain objects called actions. Then you write a special function called a reducer to decide how every action transforms the entire application's state.
* Actions are just plain objects. They can be logged, serialized, stored, and later replayed for debugging or testing purposes.
* The main idea of Redux is that you describe how your state is updated over time in response to action objects, and 90% of the code you write is just plain JavaScript, with no use of Redux itself, its APIs, or any magic.
* A single state tree makes it easier to debug or inspect an application; it also enables you to persist your app's state in development, for a faster development cycle.
* A fundamental idea of Redux is that the state of your whole application is stored in an object tree within a single store, a single source of truth.
* In a typical Redux app, there is just a single store with a single root reducing function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.
* Some functionality which has been traditionally difficult to implement - Undo/Redo, for example - can suddenly become trivial to implement, if all of your state is stored in a single tree.
* State is read-only. The only way to change the state is to emit a(n) action, an object describing what happened.
* Actions have at least one property, type, which represents an instruction to follow so that the store gets updated correctly.
* In Redux changes are made with pure functions.
* To specify how the state tree is transformed by actions, you write pure reducers.
* Types are conventionally written in all caps and underscores — colloquially known as "Screaming Snake Case".
* Reducers are just pure functions that take the previous state and an action, and return the next state.
* Remember to return new state objects, instead of mutating the previous state.
* You typically start with a(n) single reducer, and as your app grows, split it off into smaller reducers that manage specific parts of the state tree.
* Because reducers are just functions, you can control the order in which they are called, pass additional data, or even make reusable reducers for common tasks such as pagination.
* Actions can also optionally contain any sort of data included with them. Just add another property to the action object. The property name is up to you, but can be helpful if you standardize the name throughout your app, such as data or payload.
* Actions are created by "Action Creators", which are just functions that return action objects. You normally won't hard code actions manually, but instead will create functions that return the action objects. Hence, the examples below are just for illustration and are not syntactically correct in terms of action creators.

Wk5

* Actions are simple objects that describe the changes that need to be made in the data store.
* Action creators are functions that, when called, return action objects.
* If there is any data you need to include in your action, you should pass that data in as parameters to the action creator.
* The dispatcher sends actions to the reducer.
* Where the action is essentially a description of what should change and the reducer is the thing that makes the change (coming up next), the dispatcher is the vehicle that sends an action to a reducer.
* In Redux, the dispatcher is built in to the store and can be called with the store's dispatch() method.
* The reducer's job is to take the old info from the store and update it with the new data from the action that the dispatcher gave it.
* In Redux, the reducer is conceptually equivalent to React's setState method.
* Reducers are just functions, and they need to be "pure functions", meaning they won't alter the previous state directly, but instead will return a new state that will be used to overwrite the old state.
* When you have a very simple application, you can get away with using a single reducer. Once your app scales to handle any appreciable amount of data, however, you'll want to use Redux's combineReducers function to write reducers that each handle small pieces of the store's data.
* The reducer function takes 2 parameters: the current version of state and the action object that the dispatcher gave it.
* The reducer should look at the action object and determine what to do based on the action's type property. Typically, this decision process is handled with a JavaScript switch statement.
* The store is the "single source of truth" in regards to the data of your application - all data that needs to be used across the entire application is kept in the store.
* You use Redux's createStore function to create a new store. When doing so, you need to pass the reducer to the store so it knows how update itself whenever certain actions are dispatched.
* There's a utility method on the store called subscribe() that lets us run some code every time something in the store changes. For our own benefit, lets use this in combination with another method on the store, getState(), which will return the current state object.
* When you create the store, you pass the reducer function (like a tool) of what should happen when any given action is dispatched.
* The store has the dispatcher built into it (store.dispatch()), which should be passed a(n) action (an object with a "type" property at least). With that action, the reducer will run the switch statement in order to figure out what it should do with any given action type.
* React Redux is designed to work with React's component model. You define how to extract the values your component needs from Redux, and your component receives them as props.
* React Redux creates wrapper components that manage the store interaction logic for you, so you don't have to write it yourself.
* React Redux automatically implements complex performance optimizations, so that your own component only re-renders when the data it needs has actually changed.

Wk6

* React Redux is the official React binding for Redux. It lets your React components read data from a Redux store, and dispatch actions to the store to update data.
* In React Redux where we want to dispatch and action you
* use connect in the export statement.
* Connect creates a container component. It needs to know 1 —which state we want to use and connect to props (first param) and 2 — which actions to map to the dispatcher (second param). It then returns a function that we need to call with the component we want to be the presentational component for the container connect creates.
* React Redux provides Provider, which makes the Redux store available to the rest of your app.
* Developing React apps becomes a lot more pleasant when you utilize the React Devtools. In many ways, it looks like your DOM tree does in the elements tab, and you will explore it in the same way.
* If you use Redux, you'll want to use the Redux developer tools as well. You can get to the store with the React Developer Tools, but it's not easy to find, and you'll want to be able to explore what's going on step by step in your app as easily.
* Once the Redux developer tools has been installed you will also need to put this in your code where you create your store:
  + window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_ && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_\_()
* React Redux provides a connect function for you to connect your component to the store. Normally, you’ll call connect in this way:
* Redux itself is a standalone library that can be used with any UI layer or framework, including React, Angular, Vue, Ember, and vanilla JS. Although Redux and React are commonly used together, they are independent of each other.
* If you are using Redux with any kind of UI framework, you will normally use a "UI binding" library to tie Redux together with your UI framework, rather than directly interacting with the store from your UI code.
* React Redux is the official Redux UI binding library for React. If you are using Redux and React together, you should also use React Redux to bind these two libraries.
* When integrating Redux with a UI: Using Redux with any UI layer requires the same consistent set of steps:
* he process of subscribing to the store, checking for updated data, and triggering a re-render can be made more generic and reusable. A UI binding library like React Redux handles the store interaction logic, so you don't have to write that code yourself.
* Redux was inspired by several important qualities of Flux. Like Flux, Redux prescribes that you concentrate your model update logic in a certain layer of your application (“stores” in Flux, “reducers” in Redux). Instead of letting the application code directly mutate the data, both tell you to describe every mutation as a plain object called an “action”.
* Unlike Flux, Redux does not have the concept of a Dispatcher. This is because it relies on pure functions instead of event emitters, and pure functions are easy to compose and don't need an additional entity managing them.
* Another important difference from Flux is that Redux assumes you never mutate your data. You can use plain objects and arrays for your state just fine, but mutating them inside the reducers is strongly discouraged. You should always return a new object, which is easy with the object spread operator proposal, or with a library like Immutable.
* While Redux can be used with any UI layer, it was originally designed and intended for use with React. There are UI binding layers for many other frameworks, but React Redux is maintained directly by the Redux team.
* As the offical Redux binding for React, React Redux is kept up-to-date with any API changes from either library, to ensure that your React components behave as expected. Its intended usage adopts the design principles of React - writing declarative components.
* React components are a lot like functions. While it's possible to write all your code in a single function, it's usually better to split that logic into smaller functions that each handle a specific task, making them easier to understand.
* While you can write large React components that handle many different tasks, it's usually better to split up components based on responsibilities. In particular, it is common to have "container" components that are responsible for collecting and managing some kind of data, and "presentational" components that simply display UI based on whatever data they've received as props.
* The React Redux connect function generates "container" wrapper components that handle the process of interacting with the store for you. That way, your own components can focus on other tasks, whether it be collecting other data, or just displaying a piece of the UI. In addition, connect abstracts away the question of which store is being used, making your own components more reusable.
* As a general architectural principle, we want to keep our own components "unaware" of Redux. They should simply receive data and functions as props, just like any other React component. This ultimately makes it easier to test and reuse your own components.