04/03/24

Episode - 11 Data is the new oil.

★ Higher order components in React.

- → A normal JS func. which takes a component & returns a compo
- → It takes an existing component as an input and its add some extra features into it & returns it.

→ We will use higher order function to add a "promoted sticker" on some of the restaurant cards.

→ Inside higher order components we do not change anything of the input component, we just add some more extra features or func.

→ Syntax

```
export const HigherOrder = (PrevComp) {
    return () => {
        return (
            <div>
                <PrevComp />;
            </div>
        )
    }
};
```

→ Here you will receive the props.

→ extra added feature logic

returning a component or JS function

↓

fun component is nothing but a normal JS func. which returns some piece of JSX

```
const NewComp = HigherOrder(PrevComp);
```

Here we will get a new component

```
< NewComp data = {restaurants} />;
```

Recieving the prop.

```
const HigherOrder = (PrevComp) {
    return (Props) =>
    { return (
        <div>
          !
        < PrevComp {...Props} />
        </div>
    } );
};
```

Here we will receive our previous component as a parameter

Here we will receive our prop inside ~~this~~ New Comp.

New Comp → our New Comp will be equal to this part

Spread operation with this we are sending received Props to PrevComp with any unintentional changes into it.

→ All the react app have 2 layers ① UI & ② Data layer

→ UI layer is powered by data layer → JSX

→ Data layer consists of all our Props, local variables, state what ever data you have.

→ If you know how to handle your data layer then your app will be very fast.

05/03/24

☆ Control & un-control component

→ Lifting the State up.

→ Let's say that we have a parent component & inside parent component we have some children.

→ Now all the childrens have their own state, all of them are controlled by themselves. So they are uncontrolled components we can'not controll all the components together as they have their own different individual state variables.

→ Now let's say I want to control all the children's State together. With one click I want to change the State of all the children.

→ To do so, I need to firstly remove their individual State variables. and then I need to declare a State variable inside the parent component to change the State of all the components together.

→ As we have lifted or replaced the state variable from child component to one level up (to the nearest parent component) This is known as lifting the State up.

→ Now the parent component which can control all their child component's state together, then their child components are is known as controlled components

→ If I want to control the State from the parent component, then I need to send that State as a prop to child comp.

→ In react we can not send a local State variable as a prop to any child component.

→ To do so we can send it by wraping inside a function

```
const parent = () ⇒
{
    const [localprop, setLocalprop] = useState(null);

    return (
        <div>
            <child1 setLocalprop={() ⇒ setLocalprop()}/>
            <child2 setLocalprop={() ⇒ setLocalprop()} />
        </div>
    );
};
```

Here we need to write the updated value which we want to give into localprop

This is just a normal variable name. We can name it anything.

→ From childcomponent you can receive this prop as a normal prop. and when you want to update the value of this value then just call that function inside child compo.

→ When you will call it, the state variable will get updated as state variable will get updated ~~all~~ the parent component & child component will get re-rendered and our UI will change.

→ Read the doc of lifting the state up on official react website

★ Props drilling

→ In react data flow in unidirectional. you can send data parent to child component only, not from child component to parent.

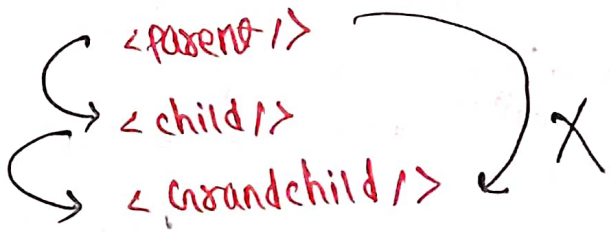
→
```
const parent = () ⇒
    return ( <div>
                <child/>
            </div>
    );
```

```
const child = () => {
    return (
        <Grandchild />
    )
};
```

→ Now Suppose you want to send any data from parent component to Grandchild component, you ~~cant~~ can not do it directly.

→ You need to send data from parent to child & then from child to grand child.

→ But here child doesnot have to do anything with that data. &



→ So this concept in react is known as props drilling.

→ Suppose you have a very big project then sending data like this to intermediate components can cause a problem if their is a very heavy long component nesting into it.

→ We Should avoid ~~com~~ Props drilling. for this react gives us a superpower known as React context

→ To Solve the problem of props drilling we can use other ways also like using any State managemet tool, for example redux or mobx

→ React contex is one of the ways to Solve Props drilling

☆ React context.

→ React context are like a global store from where any component can directly access that ~~tabel~~ data.

→ React context is nothing but an object is we can access that object directly | globally in any component of our app.

→ How to create react context

1.) → import { createContext } from react

2.) Make a new file :-

keep it capital

```
const ~~create~~UsexContext = createContex ({
    name: "Nihal",
});

export default UsexContext;
```

→ It takes an object as an input.

→ default value into our usexcontext

→ How to use inside functional component

① import { usecontext } from react

```
const comp = () {
    const {name} = usecontext(Usex Context);
    return (
        <div>
          {name}
        </div>
    )
};
```

HOOK

→ our context file.

→ How to use inside class based components.

```
const classcomp extends React.component {

    render
    ~~return~~ () {

        return (
            <usercontext.consumer>
                { (data) => console.log(data); }
            </usercontext.consumer>
        );
    };
};
```

As we are using it as a component that's why its 1st letter should be capital.

→ as a component, it takes a function inside

Here we will get our data

→ How to change it.

From here we are removing the previous content of context and put this data into it

```
const AppRouter = () => {
    <usercontext.Provider value={{name: "khushi"}}>
        <Home/>
        <Body/>
    </usercontext.Provider>
};
```

we are passing a new object into it ~~& removing~~ which will replace the previous one

It will provide the contex to all the components mentioned into it. with the data as passed into value

→ All the components which are mentioned inbetween .Provider will only ~~th~~ consider the update content of context not the previous default value of our Usercontext.

→ In value we can also pass a State variable.

```
const appRouter = () => {
  const [userName, setUserName] = useState();
  useEffect( () => {
      const data = {
        name: "xyz"
      };
      setUserName(data);
  }, [ ]);
```

After initial rendering we are updating our state variable.

we are also include setUserName into our userName context

```
return (
    <UserContext.Provider value={{name: userName, setUserName}}
```

→ Now as we have included setUserName into our user context we can use it to change the value of our UserContext

→ Every time the value gets changed the whole appRouter will get re-render and all the components inside UserContext.Provider will get the new updated content/value

```
const {name, setUserName} = useContext(UserContext);
```

If I will do setUserName("Rohit");

(means all the components mentioned inside .Provider)

Then every where the UserContext's object will change and inside name we will get "Rohit"

```
< usercontext.Provider value = { {name : "Nihal"} } >
    <Header />
    < Body />
    <usercontext.Provider value = { {name : "xyz"} } >
      < Footer />
    < /usercontext.Provider>
< /usercontext>.
```

inside footer component we will get name as xyz.

and in all other components we will get name as Nihal.

```
<UserContext.Provider>
    <div className="app">
      <Header />
      <Outlet />
    </div>
  </UserContext.Provider>
```

If I don't give nay value to UserContext.provider react throws the below error

Warning: The `value` prop is required for the `<Context.Provider>`. Did you misspell it or forget to pass it?
    at Applayout (http://localhost:1234/index.7271efb6.js:2985:71)
    at RenderedRoute (http://localhost:1234/index.7271efb6.js:29192:11)
    at RenderErrorBoundary (http://localhost:1234/index.7271efb6.js:29143:9)
    at DataRoutes (http://localhost:1234/index.7271efb6.js:27966:11)
    at Router (http://localhost:1234/index.7271efb6.js:29721:21)
    at RouterProvider (http://localhost:1234/index.7271efb6.js:27743:11)

SPREAD OPERATOR (…)

The spread operator in JavaScript is represented by three dots (`...`). It is a syntax used for several purposes, primarily related to working with arrays and objects. Here are its main uses:

1. Copying Arrays:

```
const originalArray = [1, 2, 3];
const copiedArray = [...originalArray];
```

The spread operator can be used to create a shallow copy of an array. This is useful to avoid modifying the original array unintentionally.

2. Merging Arrays:

```
const array1 = [1, 2, 3];
const array2 = [4, 5, 6];
const mergedArray = [...array1, ...array2];
```

It allows you to combine multiple arrays into a single one.

3. Passing Function Arguments:

```
const numbers = [1, 2, 3, 4, 5];
const sum = (a, b, c, d, e) => a + b + c + d + e;
const result = sum(...numbers);
```

When calling a function, the spread operator can be used to pass each element of an array as individual arguments to the function.

4. Copying Objects:

```
const originalObject = { name: 'John', age: 30 };
const copiedObject = { ...originalObject };
```

Similarly to arrays, the spread operator creates a shallow copy of an object.

5. Merging Objects:

```
const object1 = { name: 'John' };

const object2 = { age: 30 };

const mergedObject = { ...object1, ...object2 };
```

It allows you to merge the properties of multiple objects into a new one.

The spread operator is a concise and powerful feature that simplifies common tasks in JavaScript, especially when working with data structures like arrays and objects. It enhances code readability and reduces the need for more complex alternatives.

Dot (.) notation and bracket [“ ”] notation

The syntax `items.card.name` and `items["card"]["name"]` are two different ways of accessing the value of the property named "name" nested within the "card" property of the object referred to by the variable `items` in JavaScript.

In JavaScript, both dot notation (`items.card.name`) and bracket notation (`items["card"]["name"]`) can be used to access object properties. They are functionally equivalent, and you can use either depending on your preference or the specific requirements of your code.

The notation `items.card.name` is known as dot notation, where each dot represents a level of nesting within the object. On the other hand, `items["card"]["name"]` is known as bracket notation, where square brackets are used to access properties.

Both notations are commonly used, and the choice between them often depends on the specific situation or personal coding style preferences.