REACT →

THE IMPORTANCE / SIGNIFICANCE!

→ Try to create a TODO application using DOM manipulation.

* The problem with this approach is it very hard to add and remove elements and no central state

↳ what if there is a server where these todos are put?

↳ what if you update a TODO from mobile app & you will get back the new array of TODOs on the frontend how will you update the DOM then?

↳ writing addTodo fn is correct then what about updateTodo / removeTodo.

TRY

```
const element = document.createElement('div')
```

↳ just creates element

```
element.innerHTML = 'some content'
document.getElementById('todo').appendChild(element)
```

this doesn't add element to screen (document binds) rather only creates it

↓

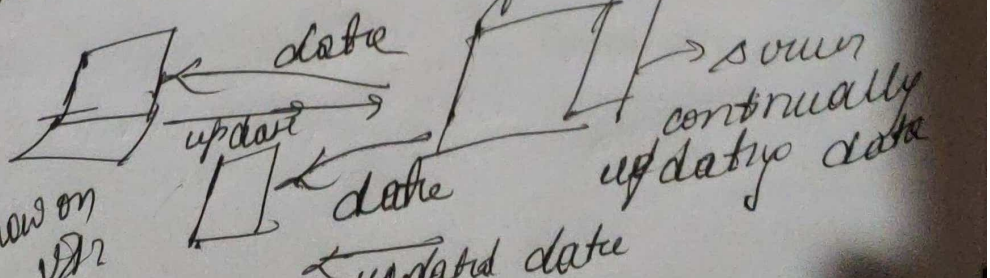
It binds element to screen

adds to DOM.

* If you hover in console screen gets highlighted else not

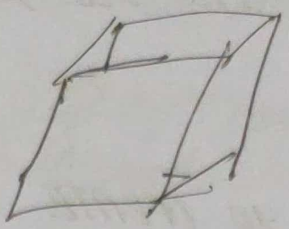
* On a deeper note: consider if there is a backend that spontaneously sends us the data

How to manage large scale data to show on UI



Consider Linkedin

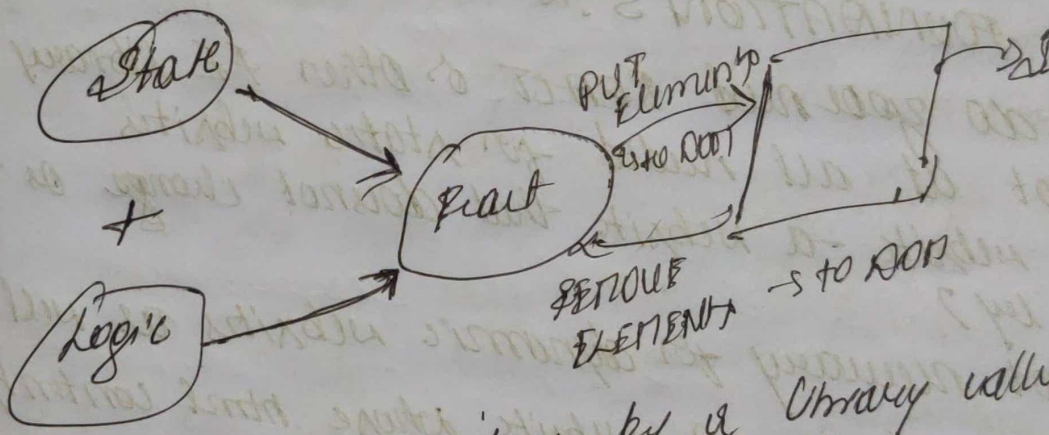
```
[ {
    name: "...",
    profile photo: "...",
    jobs: [ {
        } ,
        {
        }
    ],
    posts: [ {
        },
        {
        },
    ]
}
```



Data in JSON

The whole data about can be considered as a state.

*Now it would be very helpful if we have some thing that takes state and based on some logic which takes care of how to put elements to the dom & how to remove elements from the dom



*This something is done by a library called react

=>

*In the TODO App earlier DOM manipulation, we have chaining the innerHTML and appending things again

=> NOT OPTIMAL BECAUSE!

*What if backend returns same state again?

* We are re-uploading same thing

->BETTER WAY

*Whenever a new state comes, calculate the diff b/w old & new & accordingly update the required element. (not whole)

* Remembering the old state in variable for VIRTUAL DOM

EASIEST WAY TO CREATE Dynamic Frontend website

1. Update state variable → updateState(newState)
2. Delegate the task of figuring out diff to a diffy function
3. Tell diffy function, how to add, update & remove elements in DOM

→ By REACT & libs are by FE developers

We have to consider them ⇒

REACT → Basically for interfaces → Platform-agnostic

REACT-DOM → for DOM manipulations → DOM is only for web Applications

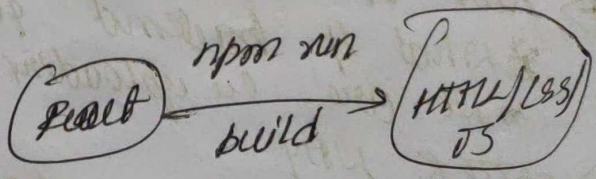
REACT FOUNDATIONS :~

⇒ Why do you need REACT: or other FE library

* Not at all needed for static websites (static website - a website that doesn't change as time goes by)
* Not necessary for dynamic websites as well (dynamic website - a website whose html content change over time)
* for dynamic websites, these libraries make it easier to do DOM manipulation.

→ REACT is just an easier way to write normal HTML/CSS/JS.

→ It's a new syntax that under the hood gets converted to HTML/CSS/JS



* People realized it's harder to do DOM manipulation the conventional way.
* Some libraries made it slightly easy (JQuery) but for very big app it was difficult.
* Eventually, VueJS/React created a new syntax to do frontends.
* under the hood, react compiles convert code to HTML/CSS/JS

-> To create a react app, we must know
- State
- Components
- Re-rendering.

-> All frontend framework is divided into two parts

(State) (Components)

=> STATE / COMPONENTS / RERENDERING.

(State) -> An object that represents current State of the app
-> It represents dynamic things in your app (things that change).
Eg: value of counter

COUNTER
Eg: {
counter: 1
}

LINKEDIN TOPBAR
{ topbar: {
home: 0,
mynetwork: "99+",
jobs: 0,
messaging: 0,
notifications: 10
} }

* Need to focus on updating State & REACT renders it.

component → how a DOM element should render given the state
→ Bp a re-usable snippit that changes given the state

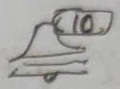
RE-RENDERING:

STATE

COMPONENT

{
  topbar: {
    name: 0,
    myNotifs: 999+,
    jobs: 0,
    messaging: 0,
    notifications: 11
  }
}

→ A state change triggers a re-rendering.
→ A re-render represents the virtual DOM being manipulated when state changes.



=> you usually have to define all your components once; & then all you have to do is update the state of your app, React takes care of re-rendering your app.

Check 1_counter.html

↳ Translated the REACT
1) call buttonComponentRender() for first time to get the first time rendering
2) there it calls buttonComponent(state.count)
3) the buttonComponent creates a button with given count & suggests onClick to buttonOnPress() function
4) the buttonOnPress() increments the state.count & calls re-render
5) this time in re-rendering previous state.count is valued & calls the buttonComponent

→ buttonOnPress()
→ buttonComponentRender()
→ buttonComponent()
→ state: { count: 0 }

⇒ In react we use JSX

JSX ⇒ Javascript and XML

NOTE: All the returned thing all are not and not HTML

* to call a funcn in JSX / component we just mention the function name ✓ not call it ✗

```
let stat: {count:0}
function App()
{
    function clickHandler()
    {
        console.log("clicked");
        stat.count++;
    }
    return (
        <div>
            <button onClick={clickHandler}>BUTTON
            {stat.count}</button>
        </div>
    );
}
```

↳ will only increment but not rendered

OBSERVE

⇒ REACT says if you want to define a state variable, you have to define it in a certain way so that I'm watching it.

⇒ If not watching. It doesn't know that it must update the DOM

⇒ Hence to define state we use useState hook

```
const [counter
```

* How the above return ('someHTML') is creating DOM elements? Ans:

* Earlier we used React.createElement('button', {onClick: clickHandler}, 'counter $$--.$');

* Now the new version/way gets TRANSPILED to above format

* Whenever a parent re-rendered the child also gets re-rendered
* If props passed to child are state variables then the above happens.

⇒ consider todos = [
{
Name: "",
Description: "",
status: ""
},
{ — — — }
]

* If a state has an array of obj/elp. like todo, make a component to just display one todo

REACT DEEPER DIVE: ⇒ React returns, re-rendering, key, Wrapper components, useEffect, useMemo, useCallback, useRef

⇒ React Component return: 3-react-component-return

* A component can only return a single top level xml

WHY?? The process of figuring out what DOM updates need to happen as application grows

1. Makes easy to do reconciliation
2.

```
function App()
{
  return (
    <Header title="some" />
    <Header title="Brother" />
  );
}
```

→ Not Possible

Possible ways

```
return (
  <div>
    <Header --- />
    <Header --- />
  </div>
);
```

→ Wrap <Header> inside div or input

Don't wrap Header on input

```
return (
  <>
    <Header --- />
    <Header --- />
```

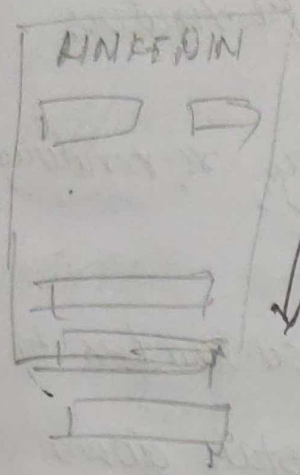
→ Fragments

⇒ Re-rendering. 4-re-render

* Any time a final DOM manipulation happens / any time React actually updates the DOM, it's what's considered as a re-render

→ Why do we need React??

To create Dynamic Websites:

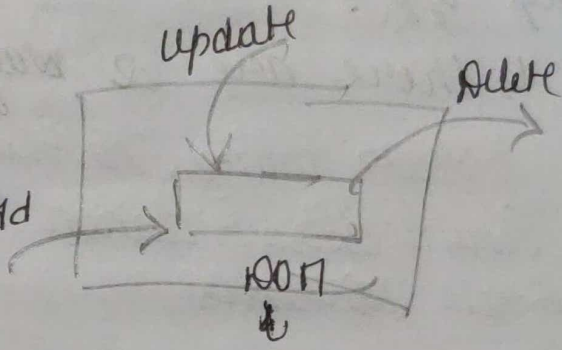


→ Whenever we scroll on LinkedIn, more and more content gets added / more posts get added to bottom, & this is what a dynamic website

→ React gives us the easy way to create these kind of websites

⇒ Whenever such dynamic changes happen like

→ When add something to DOM
→ Update something to DOM
→ Delete something from DOM



✓ What is re-render

- Actual example is the counter app
- When we click on button the value changes in UI from 0 to 1, 1 to 2, 2 to 3 ... & this change is called re-render

* While making DYNAMIC WEBSITES, the rule of them is to MINIMIZE the number of RE-RENDERS.

⇒ A RERENDER MEANS that:

1. React did some WORK to calculate what all should update in this component
2. The component actually got called (confirm by log statement)
3. The inspector shows you a bounding box around the component ↳ from React dev tools extension

RE-RENDER HAPPENS when :-

1. A state variable that is being used inside a component changes. → * even if variable present in a component lifecycle & used by child component from props

2. Parent component re-renders triggers all children re-rendering

* You want to minimize the number of re-renders to make a highly optimal react app.
* The more the components that are getting re-rendered, the slow the app is

=> How can you minimize the number of re-renders in this app?

Is there are 2 ways → 1. Push the state down

Make changes so that Parent doesn't contain state variables ← make a HeaderWithButton component that contains state variable, not using any state variable in App component

→ 2. Use React.memo (Component definition)

Here parent re-renders & the only child where props are changed is re-rendered ← -> memo lets you skip re-rendering a component when its props are unchanged

-> Warning when parent has values as root element & not wrapped in <> -> a fragment

=> Keys in React:
* When we create todos or array & render it using map function
* If we don't use key prop for the component, we get a "Each child in a list should have a unique key prop".

→ you need to give each array item a key - a string that uniquely identifies it among other array items

→ The key tells React which array item each component corresponds to, so that it can match them up later. → This becomes important if your array items can move (due to sorting), get inserted, or get deleted.
→ A well chosen key helps React INFER what exactly has happened, & make correct update to DOM tree.

=> WRAPPER COMPONENTS: 5 - Wrapper component

* Let's say we want to build a site that contains multiple cards & they look same.

* We can create a wrapper card component that takes the inner React component as an input.

```
function App()
{
  return (<>
    <Wrapper components
    , <h1> Hello </h1>
    </Wrapper component>
  );
}
```

```
function Wrapper component({children})
{
  return (
    <div>
      {children}
    </div>
  )
}
```

→ Access by the children property.

HOOKS :-

→ useState
→ useEffect
→ useMemo
→ useCallback
→ useRef
→ useReducer
→ useContext
→ useLayoutEffect

→ Know the working of useState.

*These functions starts with use are called hooks.

*Hooks in React are functions that allow you to "hook into" React state and lifecycle features from functional components

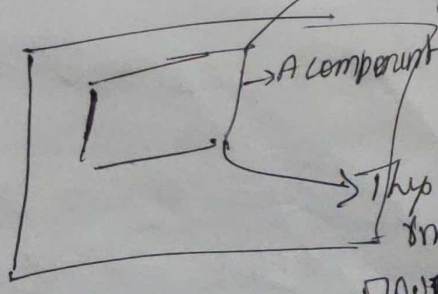
⇒ What are Lifecycle features ?? ⇒ Lifecycle Events
*Earlier React used to be different
o Firing mount if there is we used class components
*These class based components gave us some methods i.e. lifecycle features like
→ componentDidMount()
→ componentWillUnMount()
→ onRender()

} Now in functional based components, we access them through hooks

⇒ useEffect() HOOK :

*If we want to perform some action on a component at Mount, use the useEffect() hook

WHAT IS MOUNT




```
function App()
{
  useEffect(function()
  {
    alert("hi");
  }, []);
  return <div></div>;
}
```

```
useEffect(() => {
  fetch('---').then(
    async (res) => {
      const json = await res.json();
      setTodos(json.todos);
    }
  );
}, []);
```

→ Logic to run

Dependency array

* If we want to fetch something at an interval (polling)

```
useEffect(() => {
  setInterval(function() {
    fetch('---').then(
      async (res) => {
        const json = await res.json();
        setTodos(json.todos);
      }
    );
  }, 10000);
}, []);
```

→ runs on component / rerender one

→ 10s timer interval

→ re renders multiple time

dependency array

⇒ Check useAsyncEffect

⇒ SIDE EFFECTS

* The side effect encompasses any operations that reach outside the functional scope of a react component,
* These operations can affect our other components, interact with the browser, or perform asynchronous data fetching.

→ setTimeout
→ fetch
→ setInterval
→ document.getElementById()

=> HOOKS: (more about definition & use)

* Hooks are a feature introduced in React 16.8 that allow you to use state and other React features without writing a class.

* They enable functional components to have access to stateful logic and lifecycle features, which previously only possible in class components.

* They has led to more concise and readable way of writing components in React.

=> useState: Lets you describe the state of your app. whenever state update, it triggers a re-render which finally results in a DOM update

=> useEffect: It allows you to perform side effects in functional components
- serves the same purpose as 'componentDidMount', 'componentDidUpdate' and 'componentWillUnmount' in React class components, but unified into a single API

useEffect(() => { logic }, [ ])
↳ Dependency Array
↳ When it should happen.

that should happen

The specifies a set of conditions that at what time does the logic runs that passed into useEffect hook

* If dependency array is empty, then useEffect is run only first time the App's component is rendered.

→ DEPENDENCY ARRAY:

It says when should the callback function of useEffect run.

* It takes state variables as input

* The callback cannot be async function. If we want that then define an async function outside & call that within useEffect

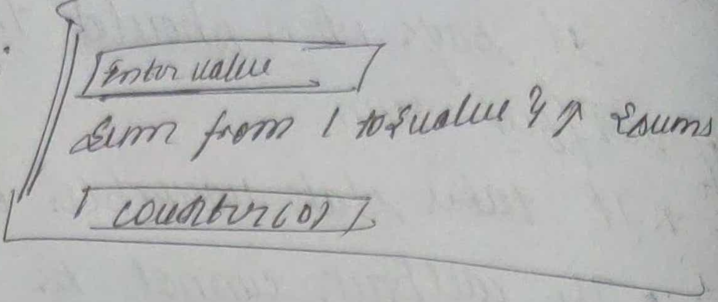
```
function App() {
    async function fetchTodos()
    { const res = await axios.get('---');
      setTodos(res.data);
    }
    useEffect(() => {
      fetchTodos();
    }, [permaStateVariable]);
    return <>
      ---
    </>
}
```

'That if the state variable changes & a call go to some 2-nos state variable again changed & the code in it not executed the second one it executed faster??

& these not the best way to do

=> useMemo:

Problem statement:



- useMemo

Here we use 2 state variables

1. for range / value -> this calculate sum till range -> causes re-render
2. counter -> incrementing counter -> this also cause re-render

* When counter state change causes re-render the for loop is run to calculate sum
* this is not optimized

-> you can do

```
useEffect(() =>
{
    let cnt=0;
    for ___

    setSum(cnt);
},
[range]);
```

} has un-necessary re-render & although it's a decent solution

=> useMemo

```
let sum = useMemo(() => {
    let cnt=0;
    for(let i =0; i<=range; i++)
        cnt +=i;
    return cnt;
}, [range]);
```

runs only when range changes

* here sum is updated before render happens.

=> useCallback: It is used to memoize functions, which can help in optimizing the performance of your application, especially in cases involving child components that rely on reference equality to prevent un-necessary re-renders

=> What is reference equality?

consider ~~var a=10~~ var a = 10;
var b = 10;

a == b -> True => 1.

{ func sum(a, b) { return a+b; }

func sum2(a, b) { return a+b; }

sum == sum2 -> False by reference equality

The reference of both funcs are different

func App()
{
const [count, setCount] = useState(0); ✗ 2
const printLog = () => { -> React is not smart enough to know this same func in child rely on reference equality.
console.log('printing in child')
} ✗]
return <div>
<Child inputFunc={printLog} />
<button onClick={() => { setCount(count+1); }}
> click {count} </button> ② -> ∴ The child also re-renders ✗
</div> ① -> Even though memo is used. Since when the parent re-renders func printLog is not reference equal.
}
func Child = memo(({inputFunc}) => { console.log('Child renders');
return <div> --- </div>
});

: to overcome that

```
const printLog = () => {
    console.log("printing log");
};
```

dependency

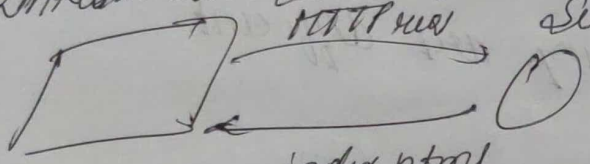
=> ROUTING, PROP DRILLING, CONTEXT API

=> ROUTING:

1. Single Page Applications: React -> a single page Applications

Before REACT:-

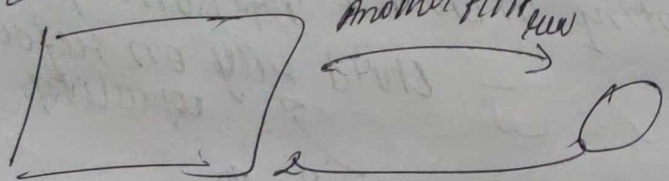
① LinkedIn.com (Can browse) HTTP req Server



index.html
index.js

② for LinkedIn.com/messages,

LinkedIn.com/messages another HTTP req

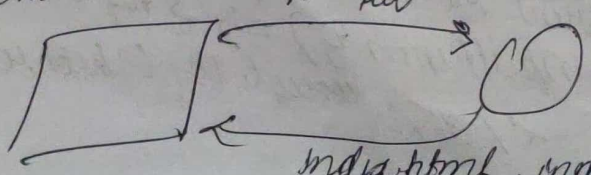


index.html
index.js

} we can see a hard reload for messages, until react comes in

=> REACT: SINGLE PAGE APPLICATION

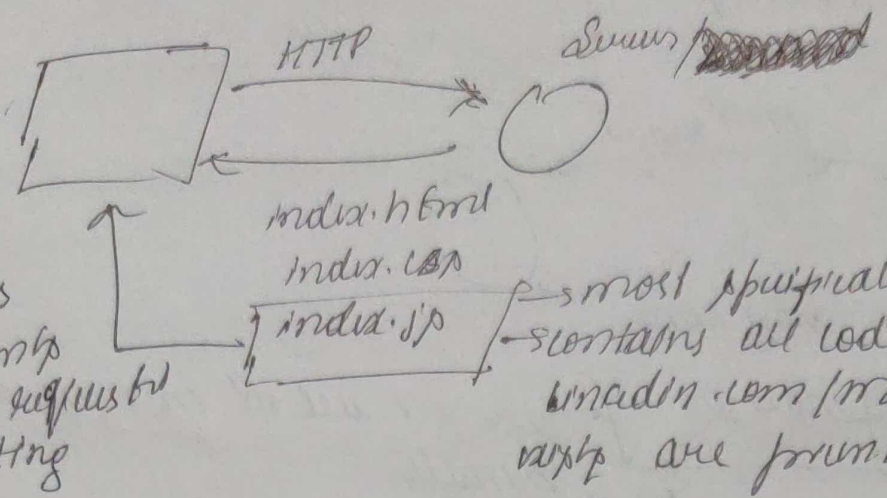
LinkedIn.com HTTP req



index.html, index.js

} Every thing comes in single http request for messages, there is no hard reload happens. client side routing happens.

2. Client Side Bundle



most specifically JS is called bundle contains all code for linkedin.com, linkedin.com/messages, all page that are present in the single JS file

→ How to do routing in react?

* react-router-dom & react-router for dom.

```
<BrowserRouter>
  <Routes>
    <Route path='/dashboard' element={<Dashboard />} />
    <Route path='/' element={<Landing />} />.
  </Routes>
</BrowserRouter>
```

→ for switching b/w 2 page, utilize useNavigate hook from "react-router-dom"

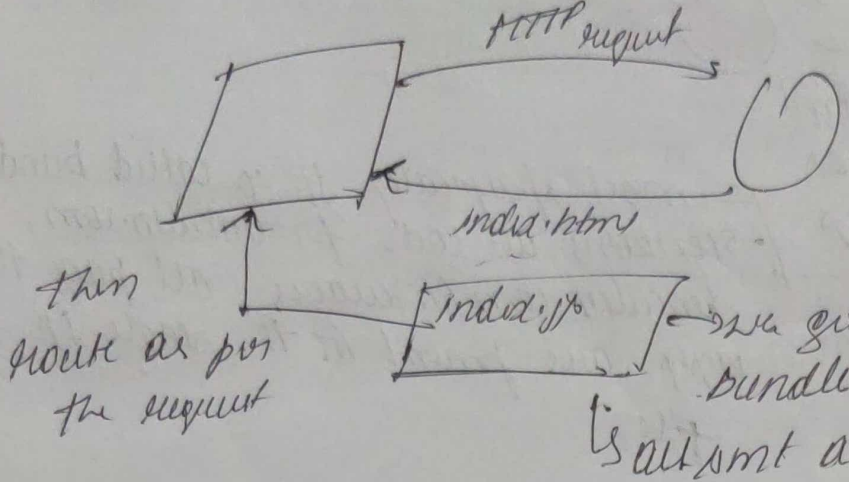
```
const navigate = useNavigate();
function handleClick() {
  navigate('/dashboard'); }
return (
  <div>
    <button onClick={handleClick}>Dashboard</button>
  </div>
);
```

This component must be within <BrowserRouter> useNavigate() or navigate('path') Does only work within <BrowserRouter> only

=> LAZY LOADING:

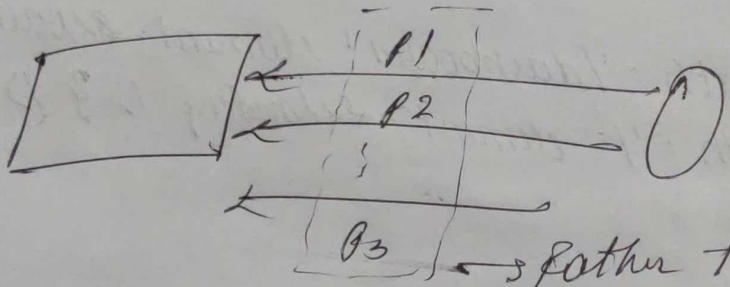
DAY 78 - X

What we did till now ?



Here is Lazy loading:

consider there are 200 pages & 200 diff routes




```
const Dashboard = React.Lazy ( () => import("./components/dashboard") );
```

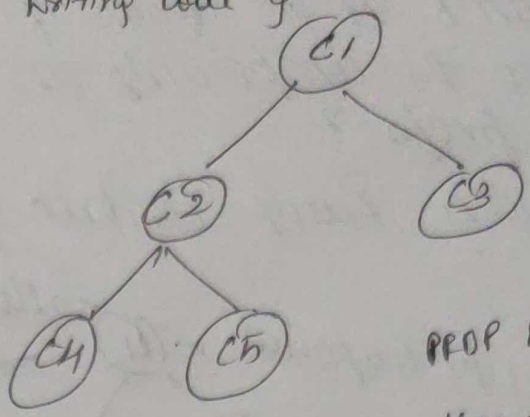
Then we it as element in routes.

* Should use Suspense API

```
<Route, path="/dashboard", element = <Suspense fallback="loading..."> <Dashboard /> </Suspense>
```

* Since Dashboard is dynamically imported, hence takes some time.
* We should tell some how to react that if it delayed then print fallback element by Suspense API - X.

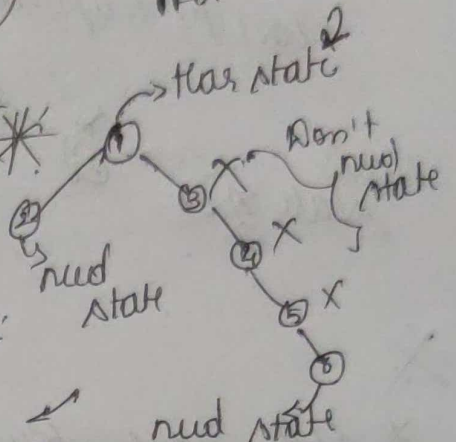
→ PROP DRILLING. { doesn't mean that parent re-renders it just means the syntactic uneasiness when writing code }



*How can we manage state (& application prop drilling)

1. Keep everything in top level component ✓

2. Keep everything as low as possible
eat the LCA of children that need state) *



even though C3, C4, C5 don't need state they pass it down - ✗

```
fy: function App()
{
  const [count, setCount] = useState(0);
  return
    (<div>
      <Count count={count} setCount={setCount} />
    </div>
    );
}
function Count ({count, setCount})
{
  return <div>
    {count}
    <Button count={count} setCount={setCount} />
  </div>
}
function Button ({count, setCount})
{
  return <div>
    <button onClick={() => setCount(count+1)}>Increase</button>
    <button onClick={() => setCount(count-1)}>Decrease</button>
  </div>;
}
```

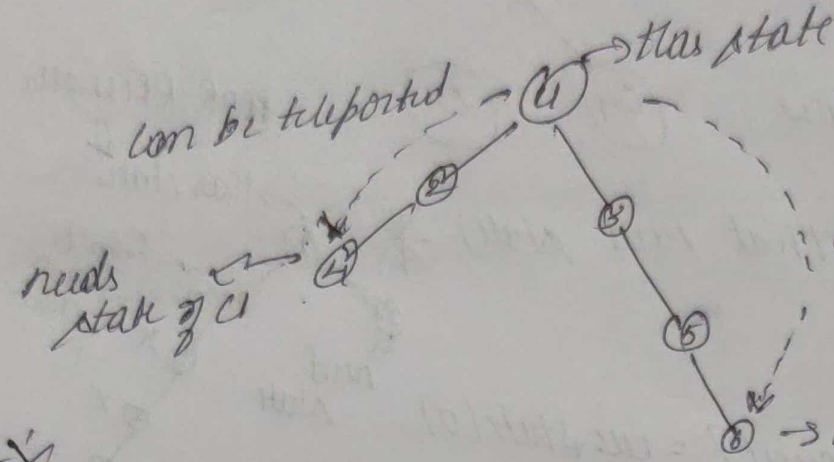
Here we are drilling down the props!!

Here count component only need count & not setCount. But the child within count needs it. It's a ugly way

→ CONTEXT API

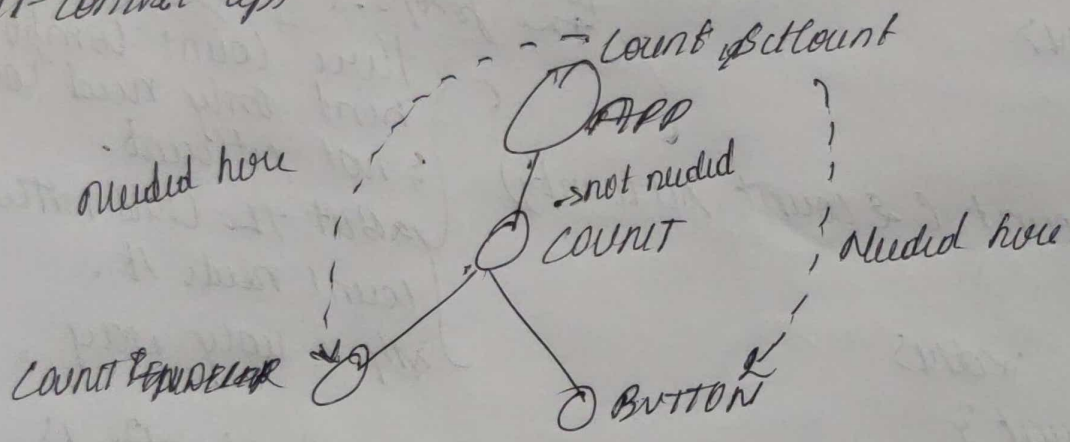
*Wouldn't it be great if there were a way to "teleport" data to the components in the tree that need it without passing props?

* With React context feature, there is!



* Lets you keep all state logic outside of your core React component *

II- context-api



STEPS:

1. Create a context that allows us to teleport the state from one component to another

context.jsx

import from 'react';

export const CountContext = createContext({ })

2. Wrap the one that wants to use the teleported value inside a provider

<CountContext.Provider value = {{count, setCount}}>

<Count />

</CountContext.Provider>

3. use the exported value in child component

```
function CounterDisplay()
{
    const { count } = useContext(CountContext);
    return <div>{count}</div>
}
function Button()
{
    const { count, setCount } = useContext(CountContext);
    return
        ==
        ==
    ;
}
```

* Context makes the code cleaner.
* state management tools → cleaner + optimized
* Look about redux / useReducer.