Week 5

→ 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

Why React?
People realised it's harder to do DOM manipulation the conventional way
There were libraries that came into the picture that made it slightly easy, but still for a very big app it's very hard (JQuery)
Eventually, VueJS/React created a new syntax to do frontends Under the hood, the react compiler convert your code to HTML/CSS/JS

→ Some react jargon
To create a react app, you usually need to worry about two things

1) State

State/Components/Re-rendering
State
An object that represents the current state of the app
It represents the dynamic things in your app (things that change)
For example, the value of the counter

⇒ anything that changes need to be in state.

2) Components

How a DOM element should render, given a state
It is a re-usable, dynamic, HTML snippet that changes given the state

3) Re-rendering

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

⇒ virtual DOM

⇒ You usually have to define all your components once And then all you have to do is update the state of your app, React takes care of re-rendering your app

⇒ while working with JWT use async await otherwise wrong working

⇒

```
function onButtonPress() {
    state.count = state.count + 1;
    buttonComponentReRender();
}

function buttonComponentReRender() {
    document.getElementById("buttonParent").innerHTML = "";
    const component = buttonComponent(state.count);
    console.log(component);
    document.getElementById("buttonParent").appendChild(component);
}

function buttonComponent(count) {
    const button = document.createElement("button");
    button.innerHTML = `Counter ${count}`;
    button.setAttribute("onclick", "onButtonPress()");
    return button;
}
```

normal JS code

c > App.jsx > ...

```jsx
import React from 'react'
function App() {
  const [count, setCount] = React.useState(0)

  return (
    <div>
      <Button count={count} setCount={setCount}></Button>
    </div>
  )
}
//component
function Button(props) {
  function onButtonClick() {
    props.setCount(props.count + 1);
  }
  return <button onClick={onButtonClick}>Counter {props.count}</button>
}

export default App
```

→ react library

define initial stage

Array destructuring

→ way to define state

var

function

⇒ const [a, b] = arr ⇒ a = 1, b = 6

⇒ JSX ⇒ file where we can write both JS and XML

⇒ <APP /> → self closing

⇒

```jsx
import {useState} from "react";

function App() {
  const [count, setCount] = useState(0); // [1, 2]

  function onClickHandler() {
    setCount(count + 1);
  }

  return (
    <div>
      <button onClick={onClickHandler}>Counter {count}</button>
    </div>
  )
}

export default App
```

npm run build → html, JS, CSS (normal)

⇒ Anytime a parent re-render, it's child re-renders as well.

{{ }} → for Inline CSS.

⇒ npm create vite@latest

⇒ React Deeper dive:

React returns, re-rendering, Key, Wrapper Components, useEffect, useMemo, useCallback, useRef

⇒ React component return Single top level xml.

why? → it make easy to do reconcilation

Process of figuring what should update and when

React.Fragment → <> </>

Re-rendering: for Dynamic websites try to minimize re-renders. Static parts should not rerender.

A re-render means that
1. React did some work to calculate what all should update in this component
2. The component actually got called (you can put a log to confirm this)
3. The inspector shows you a bounding box around the component

It happens when
1. A state variable that is being used inside a component changes
2. A parent component re-render triggers all children re-rendering

React.memo → it will prevent re-rendering.

↑ It is not useMemo

Keys in react: Key are used for uniquely identifying each data so dom re-updating can be easy.

Wrapper components : it is component which will contain other components. We can acces it by children variable.

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

Side effects :

In React, line concept of side effects encompasses any operations that reach outside the functional scope of a React component. These operations can affect other components, interact with the browser, or perform asynchronous data fetching

Hooks :

Hooks are a feature introduced in React 76.8 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 were previously only possible in class components. This has led to a more concise and readable way of writing components in React.

useState() ⇒ State of upp update and it triggers a re-render which finally results in a Dom update

useEffect() ⇒ Dependency Array.

The useffect hook is a feature in React, a popular JavaScript library for building user interfaces. It allows you to perform side effects in function components. Side effects are operations that can affect other components or can't be done during rendering, such as data fetching, subscriptions, or manually changing the DOM in React components.
The useffect hook serves the same purpose as *componentDidMount, *componentDidUpdate, and 'componentWillUnmount* in React class components, but unified into a single API.

mount -> ???
bind, this

useMemo ⇒ It means remembering some output given an input and not computing it again. → Memoization.

It is same as useEffect but by doing this we can save some re-rendering. If we use useEffect then there will be 2 render in Ideal case.

useCallback :

*useCallback is a hook in React, a popular JavaScript library for building user interfaces. 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 unnecessary renders.

referencial difference.

Customc hooks : Just like useState, useEffect, you can write your own hooks
Only condition is - It should start with use(naming convention)

Reconcilication :

useRef : it is used to get reference of Dom application.

Routing : React is Single Page Application rather than client side routing/rendering

react-router-dom : In App.jsx

```
<BrouserRouter>
  <Routes>
    <Route path="/" element={<E1/>} />
```

change page with button :- window.location.href = "/";
not right way

Right Way :- useNavigate() = hook

we can use this inside BrowserRouter only and for that we create component which will handle logic.

lazy loading :-

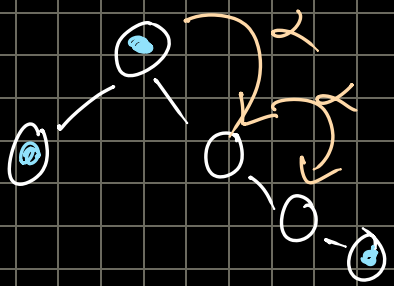
```
const DB = React.lazy(() => import("./db.jsx"));

Route path="/db" element={ <DB /> } />

<suspense fallback={"loading..."}> </suspense>
```

Suspense API :- for Asynchronous component fetching, asynchronous data fatching

Prop Drilling :- try to push down state as much as possible



Prop drilling

Prop drilling doesn't mean that parent re-renders children It just means the syntactic uneasiness when writing code

Passing props is a great way to explicitly pipe data through your UI tree to the components that use it.
But passing props can become verbose and inconvenient when you need to pass some prop deeply through the tree, or if many components need the same prop. The nearest common ancestor could be far removed from the components that need data, and lifting state up that high can lead to a situation called "prop drilling".

Context API :-

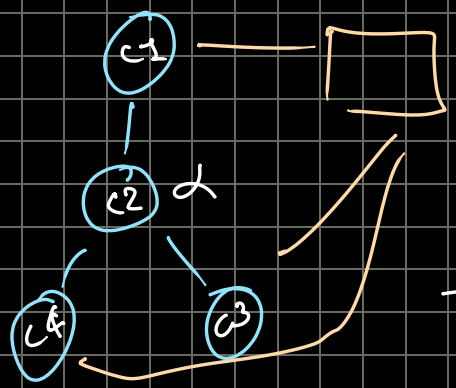
If you use the context api, you're pushing your state management outside the code react components

1) create file and export that var with createContext(0);

2) Anyone who want to use that context wrap them inside .Provider and give value like value={county}

3) useContext(variable)

downside of using Context API :



expected that if we use context API c2 will not re-render. but actualy it will re-render.

→ Don't fix re-rendering, only fixes prop drilling.

State Management :

A cleaner way to store the state of your app
Until now, the cleanest thing you can do is use the Context API.
It lets you teleport state
But there are better solutions that get rid of the problems that Context Api has (unnecessary re-renders)

Recoil : state management library for React

other popular ones -

1- Zustand

2- Redux

⇒ Store state in Store/atoms/...

⇒ define atom

Has a concept of an atom to store the state
An atom can be defined outside the component
Can be teleported to any component

→ npm i recoil

- RecoilRoot → wrap everything inside this
- atom
- useRecoilState
- useRecoilValue
- useSetRecoilState
- selector

=> const countAtom = atom({
key: "countAtom", ← for uniquely identifying
default: 0 ← default value
});

[cout, setCount] = useState(0);

useSetRecoilState ↑ useRecoilState

useRecoilVale => Just give value

const count = useRecoilValue(countAtom);

const [count, setCount] = useRecoilState(cA);

Selectors:-

```
export const evenSelector =
selector({
name: "evenSelector",
get: ({get}) => {
const count = get (countAtom) ;
return count & 2;
})
```

← Key
← function getting var and saying it depends on this variable.

A selector represents a piece of derived state. You can think of derived state as the output of passing state to a pure function that derives a new value from the said state.

if we are not passing value to other component we use useState("");

=> Recoil :- it is same as useState but more optimized. -> RecoilRoot will wrap hook.
-> we mostly use useRecoilValue.

Async data queries : Get data and put in Recoil State

selectors can be used as one way to incorporate async data.

Atom family : Dynamic Atom.

How to define :

```
export
const todosAtomFamily = atomFamily({
key:
' todosAtomFamily',
default: id =>{
return TODOS. find (x = x.id === id)
7);
```

↑ import {TODO}

How to find ? → = useRecoilValue(todoAtomFamily(1))

if we give TODOS in default then if todo change all of the components rerender.

SelectorFamily : Selector family is same as Atom family

default family for atomfamily must be selectorfamily.

```
export const todosAtomFamily = atomFamily({
key: 'todosAtomFamily', default: selectorFamily& key: "todoSelectorFamily",
get: (id) =>async ({get}) => {
const res = await axios.get(https://sum-server.100xdevs.com/todo?id=${id});
return res.data. todo;
});
```

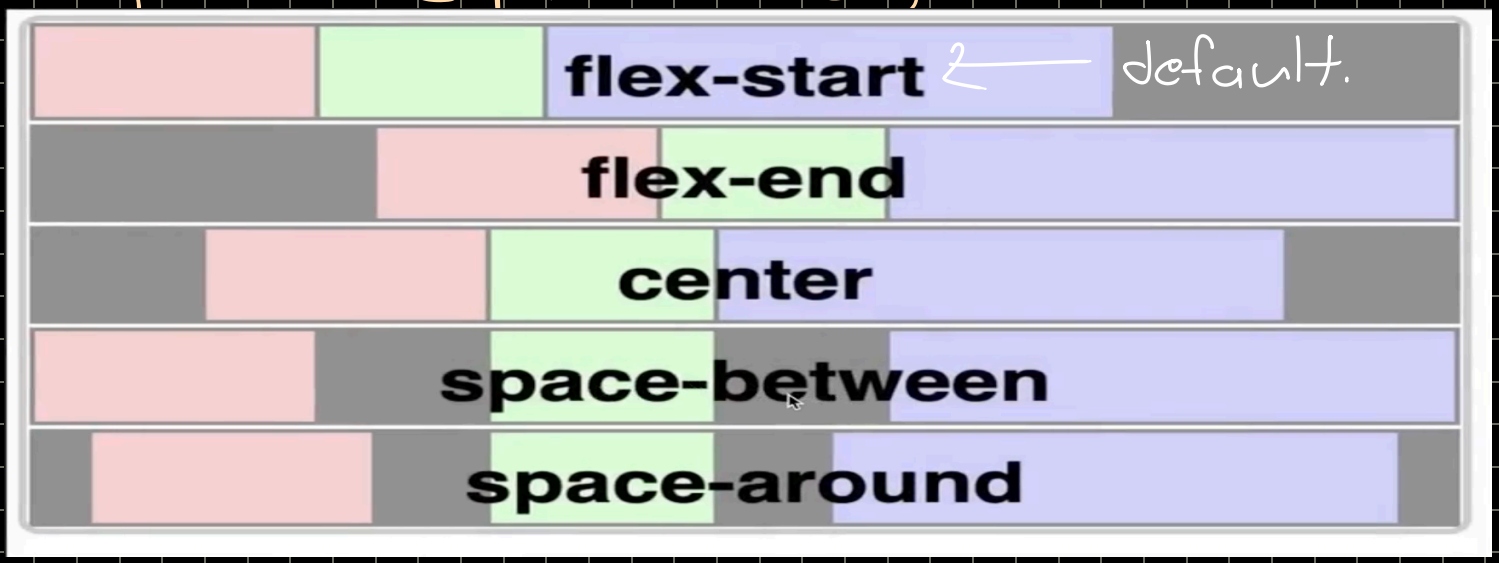
As Atomfamily create multiple Atom component if we use selector then that selector will be same for all atoms.

useRecoilStateLoadable, useRecoilValueLoadable

// Suspense, ErrorBoundary.

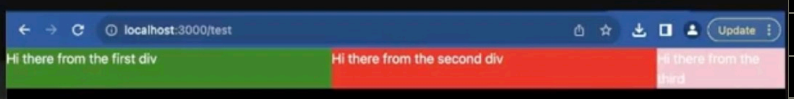
Tailwind CSS :-
fronted basic :- flex
Grid
Responsiveness.
Background color, text color, hover.

2) flex :- flex will put everything in same line & JustifyContent will Justify components (space-between)



className :- flex , grid grid-cols-12
total 12 space
col-span-5 ← this will take 5 span space

```
my-app > src > pages > TS test.tsx > ...
1
2   export default function Test() {
3       return <div className="grid grid-cols-12">
4               <div className="col-span-5" style={{background: "green"}}>
5                   Hi there from the first div
6               </div>
7               <div className="col-span-5" style={{background: "red"}}>
8                   Hi there from the second div
9               </div>
10              <div className="col-span-2" style={{background: "pink"}}>
11                  Hi there from the third
12              </div>
13      </div>
14  }
```



Responsiveness 의

Breakpoint prefix	Minimum width	CSS
`sm`	640px	`@media (min-width: 640px) { ... }`
`md`	768px	`@media (min-width: 768px) { ... }`
`lg`	1024px	`@media (min-width: 1024px) { ... }`
`xl`	1280px	`@media (min-width: 1280px) { ... }`
`2xl`	1536px	`@media (min-width: 1536px) { ... }`

Working mobile-first

By default, Tailwind uses a mobile-first breakpoint system, similar to what you might be used to in other frameworks like Bootstrap.

What this means is that unprefixed utilities (like `uppercase`) take effect on all screen sizes, while prefixed utilities (like `md:uppercase`) only take effect at the specified breakpoint *and above*.

Targeting mobile screens

Where this approach surprises people most often is that to style something for mobile, you need to use the unprefixed version of a utility, not the `sm:` prefixed version. Don't think of `sm:` as meaning "on small screens", think of it as "at the small *breakpoint*".

Don't use `sm:` to target mobile devices

```
<!-- This will only center text on screens 640px and wider, not on small screens -->
<div class="sm:text-center"></div>
```

Impo

Use unprefixed utilities to target mobile, and override them at larger breakpoints

```
<!-- This will center text on mobile, and left align it on screens 640px and wider --
<div class="text-center sm:text-left"></div>
```

Background font / Colors:

style={{background: "green", color: 'red'}}

className="bg-green-500 text-red-500"

⇒ 8.2: (Almost same as 8.1)

Grid-cols-5 : 5 element in 1 row.

col-span-2 : take 2 space insted of 1

w-[40%] : Same as col Span

Week-9: Custom Hooks.

Hooks are a feature introduced in React 16.8 that allow you to use state and other React features without writing a class. They are functions that let you "hook into" React state and lifecycle features from function components.

Class based components

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  incrementCount = () => {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>{this.state.count}</p>
        <button onClick={this.incrementCount}>Increment</button>
      </div>
    );
  }
}
```

Functional components

```
import React, { useState } from 'react';

function MyComponent() {
  const [count, setCount] = useState(0);

  const incrementCount = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>{count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}
```

avoid Classbased Components.

Lifecycle event: when component mount or unmount ;t is lifecycle event.

What are custom hooks
Hooks that you create yourself, so other people can use them are called custom hooks.
A custom hook is effectively a function, but with the following properties -
Uses another hook internally (useState, useEffect, another custom hook)
Starts with use

A few good examples of this can be
Data fetching hooks
Browser functionality related hooks - useOnlineStatus , useWindowSize, useMousePosition
Performance/Timer based - useInterval, useDebounce

when making hook makefunction where name of that function's name will start from use , Ex: useTodos().

⇒ window.navigator.onLine ⇒ will return if user is online or offline.

, eventlistner will add event listener.

Debounce :- wait for user to type and then send req to backend.