Single page loader

When we use to switch from one page to another in earlier time(html,css,js) there is a refresh loader in the top.

SPL is like when you are not switching the whole page just some content is changing and the top bar remains the same. You are contents which has to be changed from the backened. (eg allen, linkedln)

Removes the white screen while switching the page. (no fresh load)

Routing:

```
import './App.css'
import {BrowserRouter, Routes, Route} from "react-router-dom";
function App() {
  return <div>
    <a href="/">Allen</a>
    <a href="/neet/online-coaching-class-11"> | Class 11</a>
    <a href="/neet/online-coaching-class-12"> | Class 12</a>
    <BrowserRouter>
        <Routes>
            <Route path = "/neet/online-coaching-class-11" element =
            {<Class11Program />} />
            <Route path = "/neet/online-coaching-class-12" element =
            {<Class12Program />} />
            <Route path = "/" element = {<Landing />} />
        </Routes>
    </BrowserRouter>
  </div>
}
function Landing(){
    return <div>
        Welcome to allen
    </div>
}
function Class11Program(){
    return <div>
        NEET programs for Class 11th
    </div>
}
function Class12Program(){
    return <div>
        NEET programs for Class 12th
    </div>
}
```

```
function App() {
    const router = [{
        path: "/neet/online-coaching-class-11",
        element: <Class11Program />
    }]
  return <div>
```

This is the way of writing the routes in some codebases and later iterating the array in the route section

The problem in this is the page is loading again and again but we just want to get the changed part in the page to do that use **links and useNavigates.**

```
<BrowserRouter>
    <Link to="/">ALLen</Link>
    <Link to="/neet/online-coaching-class-11"> | Class 11</Link>
    <Link to="/neet/online-coaching-class-12"> | Class 12</Link>
    <Routes>
        <Route path = "/neet/online-coaching-class-11" element =
        {<Class11Program />} />
        <Route path = "/neet/online-coaching-class-12" element =
        {<Class12Program />} />
        <Route path = "/" element = {<Landing />} />
    </Routes>
</BrowserRouter>
```

Just change the a href to link and should be inside browser Router.
Now it's a SPA(Single page application)

useNavigate - This is used when we want to add any custom logic to the function (Like when we visit class 12 page load the content of class 11 page also)

(Any function with a name of use is a hook)

```
function Class12Program(){
    const navigate = useNavigate();

    function redirectUser(){
        navigate("/");
    }
    return <div>
        NEET programs for Class 12th
        <button onClick = {redirectUser}>Go back to Landing page</button>
    </div>
}
```

Default page for random routes on a website

```
            <Route path = "*" element = {<ErrorPage/>} />
        </Routes>
    </BrowserRouter>
  </div>
}
function ErrorPage(){
    return <div>
        Sorry Page not found
    </div>
}
```

Layout

```
    return <div>
      <BrowserRouter>
          <Routes>
              <Route path = "/" element = {<Layout />}>
                  <Route path = "/neet/online-coaching-class-11" element =
                  {<Class11Program />} />
                  <Route path = "/neet/online-coaching-class-12" element =
                  {<Class12Program />} />
                  <Route path = "/" element = {<Landing />} />
                  <Route path = "*" element = {<ErrorPage/>} />
              </Route>
          </Routes>
          Footer | Contact us
      </BrowserRouter>
    </div>
}
function Layout(){
    return <div style = {{height: "100vh",background: "green"}}>
        <Header />
        <div style = {{height: "90vh", background: "red"}}>
            <Outlet />
        </div>
        footer
    </div>
}
function Header(){
    return <div>
        <Link to="/">Allen</Link>
        <Link to="/neet/online-coaching-class-11"> | Class 11</Link>
        <Link to="/neet/online-coaching-class-12"> | Class 12</Link>
    </div>
}
```

This is a way of doing layout by dividing the code in header and footer code and just help to make the code look prettier.

Something compli compli

```
<Route path = "/neet" element = {<Layout />}>
    <Route path = "/neet/online-coaching-class-11" element =
    {<Class11Program />} />
    <Route path = "/neet/online-coaching-class-12" element =
    {<Class12Program />} />
    <Route path = "/" element = {<Landing />} />
    <Route path = "*" element = {<ErrorPage/>} />
</Route>
```

Use of the first line is to ensure that url is starting with /neet only other routes will not be taken and will throw error.

UseRef

1. **Problem:**We have a signup page and we want to signup by clicking the button but if the username blank is empty then we want the cursor to reach that box for that use useref.

```
import './App.css'

function App() {

  function focusOnInput() {
    document.getElementById("name").focus()
  }

  return <div>
    Sign up
    <input id="name" type={"text"}></input>
    <input type={"text"}></input>
    <button onClick={focusOnInput}>submit</
    button>
  </div>
}

export default App
```

```
3  function App() {
4      const inputRef = useRef();
5      function focusOnInput(){
6          inputRef.current.focus();
7      }
8      return <div>
9          Sign up
10         <input ref = {inputRef} type= {"text"}></input>
11         <input type= {"text"}></input>
12         <button onClick = {focusOnInput}>submit</button>
13     </div>
14 }
15 export default App
16
```

This can be done but we should never directly use the dom element so use useref

2. Never define the variable directly define it using the useref and usestate so that it can guard the variable.

```
function App() {
    const [currentCount, setCurrentCount] = useState(0);
    Let timer = 0;
    function startClock(){
        timer = setInterval(function(){
                setCurrentCount(c=>c+1);
        },1000);
    }
    function stopClock(){
        console.log(timer);
        clearInterval(timer);
    }
    return <div>
        {currentCount}
        <br />
        <button onClick={startClock}>Start</button>
        <button onClick={stopClock}>Stop</button>
    </div>
}
```

Define the state variable now its guarded

Never use the variable like this the variable gets overridden after every rerun .

```
function App() {
    const [currentCount, setCurrentCount] = useState(1);
    const [timer, setTimer] = useState(0);
    function startClock(){
        let value = setInterval(function(){
                setCurrentCount(c=>c+1);
        },1000);
        setTimer(value);
    }
    function stopClock(){
        console.log(timer);
        clearInterval(timer);
    }
    return <div>
        {currentCount}
        <br />
        <button onClick={startClock}>Start</button>
        <button onClick={stopClock}>Stop</button>
    </div>
}
```

This approach is good but the problem is that the initially the timer is re rendered again so it should not happen to remove that one rerender we are using the useref.

Basically a value which you don't want to render in the screen, just you want to store is something you should store in referencing that is when you use the reference. (save the one re render (when you click on start there is a re render))

Client side rendering is what when the code in in your system only and then executed in that only

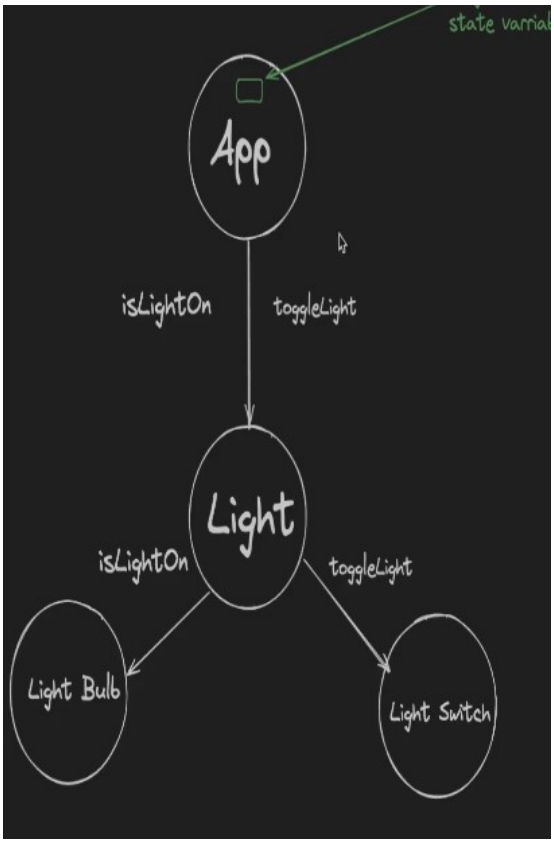
This hook provides a way to create a reference to a value or a DOM element that persists across renders but does not trigger a re render when the value changes.

Rolling up the state and unoptimal re-renders

State management(managing the state variable)

Props we send from parent to child

It basically means that we are creating a state variable which is accessed in two child (eg shown below that)




```jsx
function App() {
    return <div>
        <LightBulb />
    </div>
}
function LightBulb(){
    return <div>
        <BulbState />
        <ToggleBulbState />
    </div>
}
function BulbState(){
    const [bulbOn, setBulbOn] = useState(true);
    return <div>
        {bulbOn ? "Bulb on" : "Bulb Off"}
    </div>
}
function ToggleBulbState(){
    return <div>
        <button>Toggle the Bulb </button>
    </div>
}
```

Currently in this code the bulbOn state variable is used but the setBulbOn is required in the toggleBulbState but it is not able to achieve that bcoz it cannot access it
So access karne kae liye it should be in parent **LIghtBulb function**
Pass it as a prop to the below function

(Refering the left side code)
This is like defining the variable in the top function and then using that in another child functions and this is called as rolling up state (like dhere dhere joh state variable banaya usko upar hii pahucha diya)

Ugly thing is directly function app mai hii define kardiya sara lol (Learning about State management in this code)

```jsx
function LightBulb(){
    const [bulbOn, setBulbOn] = useState(false);
    return <div>
        <BulbState bulbOn = {bulbOn} />
        <ToggleBulbState bulbOn = {bulbOn} setBulbOn = {setBulbOn}/>
    </div>
}
function BulbState({bulbOn}){
    return <div>
        {bulbOn ? "Bulb On" : "Bulb Off"}
    </div>
}
function ToggleBulbState({bulbOn,setBulbOn}){
    function toggle(){
        setBulbOn(!bulbOn);
    }
    return <div>
        <button onClick = {toggle}>Toggle the Bulb </button>
    </div>
}
export default App
```

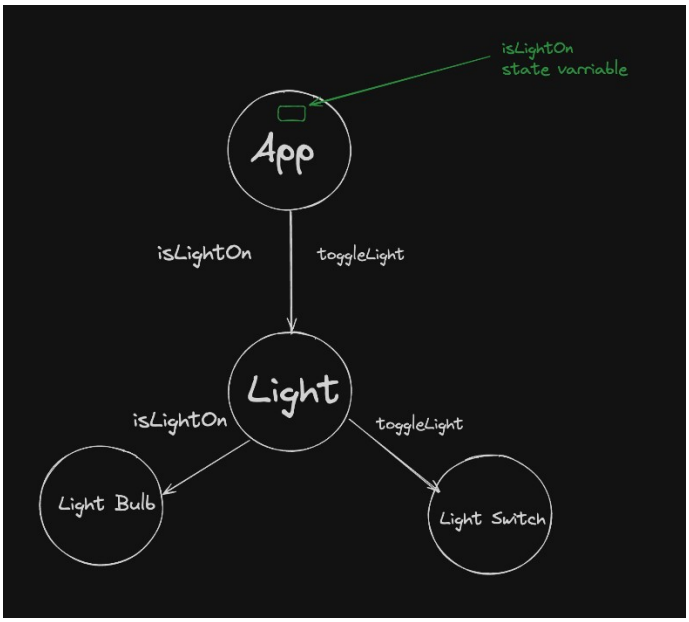
Prop – the argument we pass to a function from parent

(eg bulbOn is a prop to the Bulb State component,

bublOn, setBulbOn are props to the ToggleBulbState component)

Prop drilling

This means when we want to send the data from the parent to not to the immediate child but to child kae child ko.



Prop drilling occurs when you need to pass data from a higher-level component down to a lower-level component that is several layers deep in the component tree. This often leads to the following issues:

- **Complexity:** You may have to pass props through many intermediate components that don't use the props themselves, just to get them to the component that needs them.
- **Maintenance:** It can make the code harder to maintain, as changes in the props structure require updates in multiple components.

(From the side image) the variable is in App and we want to use that in light bulb and light switch

```
function App() {
    const [bulbOn, setBulbOn] = useState(false);
    return <div>
        <Light bulbOn = {bulbOn} setBulbOn = {setBulbOn}/>
    </div>
}
function Light({bulbOn,setBulbOn}){
    return <div>
        <LightBulb bulbOn = {bulbOn} />
        <LightSwitch bulbOn = {bulbOn} setBulbOn = {setBulbOn}/>
    </div>
}
function LightBulb({bulbOn}){
    return <div>
        {bulbOn ? "Bulb On" : "Bulb Off"}
    </div>
}
function LightSwitch({bulbOn,setBulbOn}){
    function toggle(){
        setBulbOn(!bulbOn);
    }
    return <div>
        <button onClick = {toggle}>Toggle the Bulb </button>
    </div>
}
```

In this you can see that variable is loading down through the functions

Socho agar koi aisa tarika ho ki tuum main function mai define karo and child mai access karlo easy

TO fix the above thing CONTEXT API fixes the props drilling

The Context API is a powerful feature in React that enables you to manage state across your application more effectively, especially when dealing with deeply nested components.

The Context API provides a way to share values (state, functions, etc.) between components without having to pass props down manually at every level.

Jargon

- **Context**: This is created using React.createContext(). It serves as a container for the data you want to share.
- **Provider**: This component wraps part of your application and provides the context value to all its descendants. Any component that is a child of this Provider can access the context.
- **Consumer**: This component subscribes to context changes. It allows you to access the context value (using useContext hook)

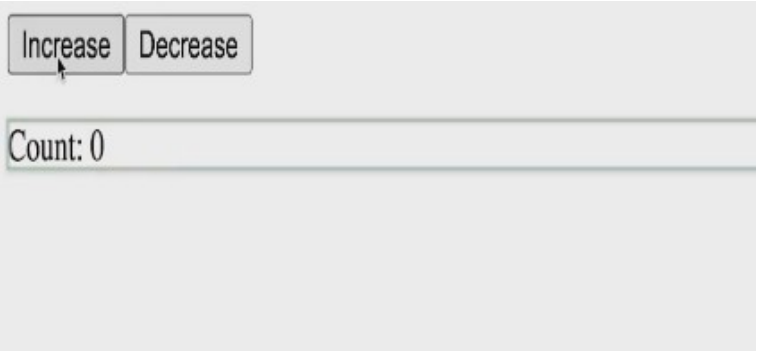
Step 1: Define the context
Step 2: Define who needs the value using the provider
Step 3: Use the Context
Code without the props drilling

Some wrapper

```jsx
import { useState,createContext, useContext } from 'react'
import './App.css'
const BulbContext = createContext();
function App() {
    const [bulbOn, setBulbOn] = useState(false);
    return <div>
        <BulbContext.Provider value={{
            bulbOn: bulbOn,
            setBulbOn: setBulbOn
        }}>
            <Light />
        </BulbContext.Provider>
    </div>
}
function Light(){
    return <div>
        <LightBulb />
        <LightSwitch />
    </div>
}
function LightBulb(){
    const { bulbOn } = useContext(BulbContext);
    return <div>
        {bulbOn ? "Bulb On" : "Bulb Off"}
    </div>
}
function LightSwitch(){
    const { bulbOn, setBulbOn } = useContext(BulbContext);
    function toggle(){
        setBulbOn(!bulbOn);
    }
    return <div>
        <button onClick = {toggle}>Toggle the Bulb </button>
    </div>
}
```

Performance Testing in state component and Context API

When we check the re rendering the State management libraries(redux , recoil) takes less re renders from context api



(Eg when we have three components in context api all are re rendering but in state management libraries only the count function is re rendering)

Custom Hooks

useState => Store a state variable, re-render a component when the variable changes

useEffect => Run a effect whenever a dependency changes

Custom Hooks should always use a another hooks inside the function and the name should always start with use (eg useValue())

Motive is to make the code look cleaner and make the main function easy to read

Same as function its used to encapsulate some amount of code inside this

Custom hooks in React are a powerful feature that allows you to encapsulate and reuse stateful logic across different components. They are essentially JavaScript functions that can use React hooks internally. By creating custom hooks, you can abstract away complex logic, making your components cleaner and more manageable.

YOU SHOULD NOT CREATE A FUNCTION WITH A HOOK IT SHOULD BE A CUSTOM HOOK.

```
function useCounter(){
    const [count,setCount] = useState(0);
    function increaseCount(){
        setCount(count+1);
    }
    return {
        count: count,
        increaseCount: increaseCount
    }
}
function App() {
    return <div>
        <Counter />
        <Counter />
        <Counter />
        <Counter />
    </div>
}
function Counter(){
    const {count, increaseCount} = useCounter();
    return <div>
        <button onClick = {increaseCount}>increase {count}</button>
    </div>
}
```

In this we made a hook which is used in counter and can be used multiple times

Useeffect when nothing inside the dependency array it just works on mount but if you add something like url inside the array, then whenever the url changes it will work.

UseFetch simple implementation

```jsx
import { useState,useEffect } from 'react';
import './App.css'
import { useFetch } from './hooks/useFetch';

function App() {
    const [currentPost, setCurrentPost] = useState(1);
    const {finalData, Loading} = useFetch("https://jsonplaceholder.typicode
    posts/"+ currentPost);
    if (Loading){
        return <div>
            Loading...
        </div>
    }
    return <div>
        <button onClick={() => setCurrentPost(1)}>1</button>
        <button onClick={() => setCurrentPost(2)}>2</button>
        <button onClick={() => setCurrentPost(3)}>3</button>
        {JSON.stringify(finalData)}
    </div>
}

export default App
```

```jsx
import { useState,useEffect } from 'react';
export function useFetch(url){
    const [finalData, setFinalData] = useState({});
    const [Loading, setLoading] = useState(true);
    async function getDetails(){
        setLoading(true);
        const response = await fetch(url);
        const json = await response.json();
        setFinalData(json);
        setLoading(false);
    }
    useEffect(() =>{
        getDetails();
    },[url])
    return {
        finalData,
        Loading
    }
}
```

In the right side code the below usefetch code is useful
Adding a loading option is also a good idea for ui

```jsx
async function getDetails(){
    setLoading(true);
    const response = await fetch(url);
    const json = await response.json();
    setFinalData(json);
    setLoading(false);
}
useEffect(() =>{
    getDetails();
},[url])
useEffect(() =>{
    setInterval(getDetails,10 *1000);
},[])
```

Usefetch with re-fetching that every 10 seconds it will check whether there is a change in the data or not.

UsePrev (Used to track the previous value of the state variable/ right behind the state variable uski phele value kya thi) **SOME HIGH LVL SHIT**

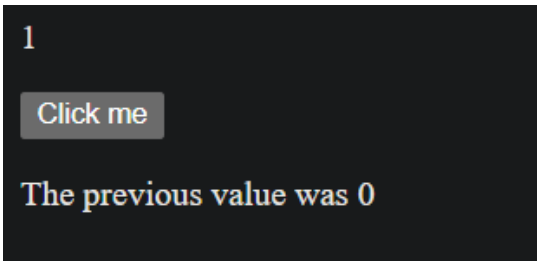
Ref variable is like whenever you use this , this is used to track the value and kitni bhi baar value update karlo it will not re render the component.

```jsx
import { useState,useEffect } from 'react';
import './App.css'
import { usePrev } from './hooks/usePrev';

function App() {
   const [state, setState] = useState(0);
   const prev = usePrev(state);
   return (
   <>
        <p>{state}</p>
        <button onClick ={()=>
            setState((curr) => curr+1)
        } >
            Click me
        </button>
        <p>The previous value was {prev}</p>
   </>
   )
}

export default App
```

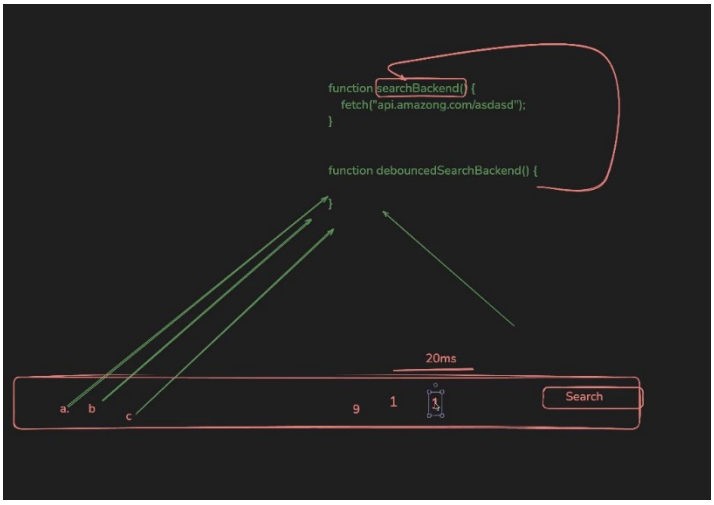
```jsx
import { useEffect, useRef } from "react";
export const usePrev = (value) => {
    const ref = useRef();
    console.log("re render happended with new value"+value);
    useEffect(() => {
        console.log("updated the ref to be "+value);
        ref.current = value;
    }, [value]);
    console.log("returned " + ref.current);
    return ref.current;
}
```

What's the console looks like for the side one?

```
updated the ref to be 0
re render happended with new value1
returned 0
updated the ref to be 1
```



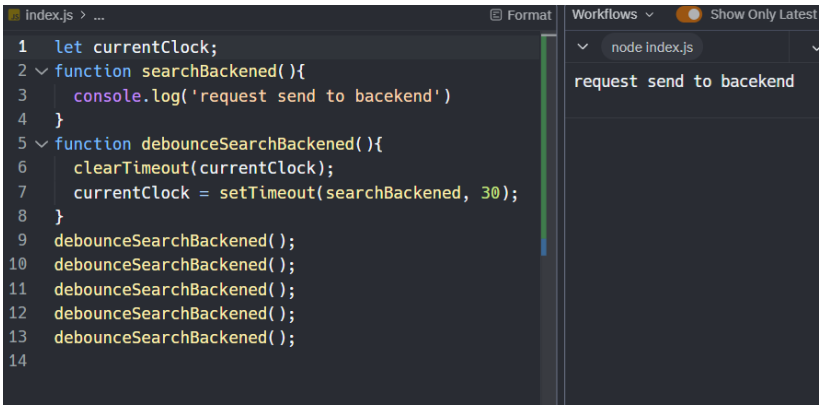
In this first the ref variable stores the value then the ref.current returns the prev value and then the useeffect gets the new value.



Debounce

This is like when we type something in amazon search and we type a lot then for some time there will be no results. If some words are

types then search results are shown.If you are typing too fast then after 20-50 ms when you stop then it will send a backend request to check for the result of the searches.



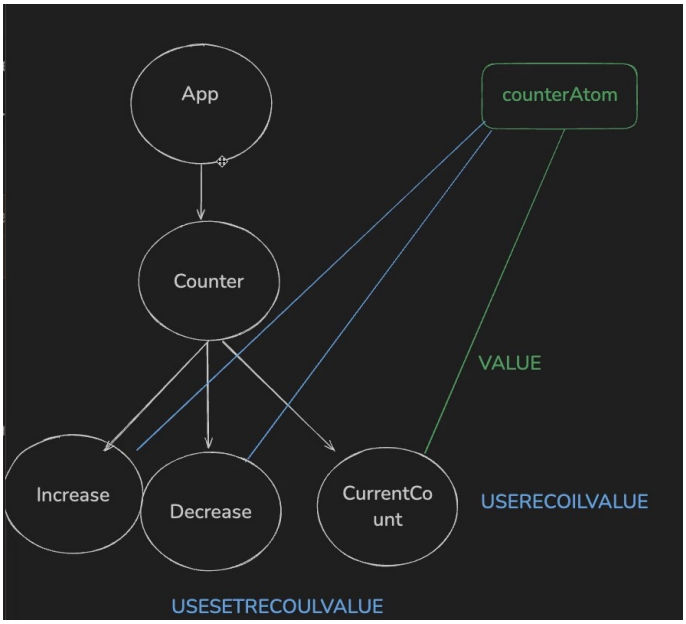
In this the code was only shown once because the clock was reset whenever a new request was called.

useDebounce

Again using ref which does not re render but stores the value

Recoil (Used to reduce the no of re- renders (Contex api, state variables both re renders a lot))

Recoil optimize the re-renders(Increases the performance)



In the given code the CurrentCount function access the value so that only will be re rendered not the other ones.

```
import { atom } from "recoil";

export const counterAtom = atom({
    default:0,
    key:"counter"
})
```

```
import { useState } from 'react'
import './App.css'
import { RecoilRoot,atom, useSetRecoilState, useRecoilValue  } from 'recoil';
import { counterAtom } from './store/atoms/counter';
function App() {
  return <div>
    <RecoilRoot>
        <Counter />
    </RecoilRoot>
  </div>
}
function Counter(){
    return <div>
        <CurrentCount />
        <Increase  />
        <Decrease  />
    </div>
}
function CurrentCount(){
    const count = useRecoilValue(counterAtom);
    return <div>
        {count}
    </div>
}
function Increase(){
    const setCount = useSetRecoilState(counterAtom);
    function increase(){
        setCount(c => c+1);
    }
    return <div>
        <button onClick ={increase}>Increase</button>
    </div>
}
```

The variable we want to must be inside a atom.
Step1: Create a atom(inside store/atoms/counter.js)
Step2:Using the functions like useRecoilValue(for them who want to display the value), useSetRecoilState(for them who want to change the value)

```
const setCount = useSetRecoilState(counterAtom);
const count = useRecoilValue(counterAtom);

const [count, setCount] = useRecoilState(counterAtom);
```

MEMO(Used to increase the performance of the useState)

When we use useState that time if any variable changes in the parent then all the childs will also re render but the issue is they are not even using that state variable. (TO FIX THAT WE HAVE TO USE MEMO)

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

  useEffect(() => {
    setInterval(() => {
      setCount(c => c + 1)
    }, 3000)
  }, []);

  return <div>
    <CurrentCount />
    <Increase />
    <Decrease />
  </div>
}

function CurrentCount() {
  return <div>
```

This currently just update this counter function every 3 seconds but the childrens increase, decrease are also re rendering.

IT WILL NOT RE RENDER UNTIL A PROP CHANGES OR A STATE CHANGES IF WE USE MEMO

```
function CurrentCount(){
    return <div>
        1
    </div>
}
```

```
const CurrentCount = memo(function (){
    return <div>
        1
    </div>
})
```

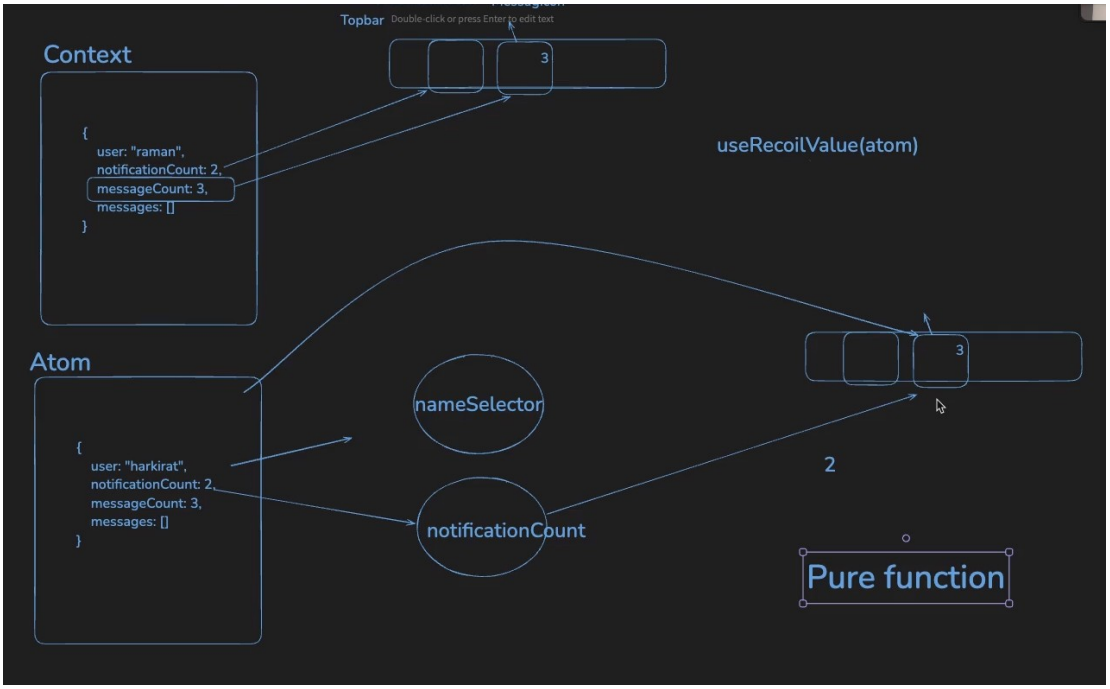
Now everything will not re render ez just that function where prop is changing.

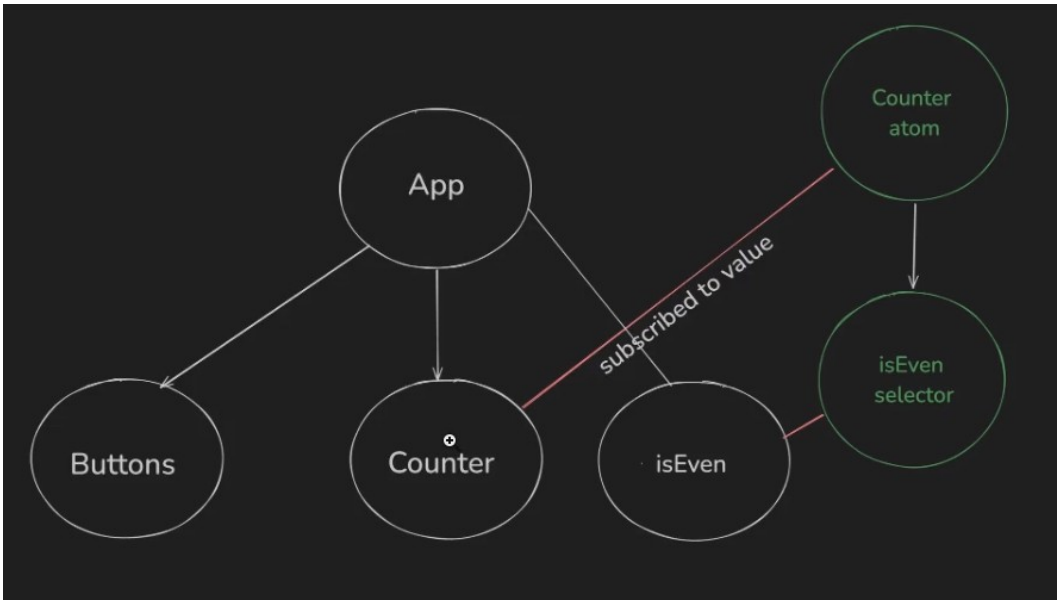
The prop here is a state variable

Selector using RECOIL

PURE FUNCTION – FOR A GIVEN INPUT ALWAYS GIVES A SAME OUTPUT

Problem: Suppose the context contains everythings and we have a change in the message icon from 10 -11 then we should change that message icon only but as they are in one context box everything re render again to fix this we need **Selector**.





CODE FOR THE ABOVE DIAGRAM

```jsx
import { atom,selector } from "recoil";

export const counterAtom = atom({
    default:0,
    key:"counter"
})
export const evenSelector = selector({
    key: "isEvenSelector",
    get: function({get}){
        const currentCount = get(counterAtom);
        if(currentCount % 2 ==0){
            return true;
        }
        else{
            return false;
        }
        return isEven;
    }
})
```

```jsx
import './App.css'
import { RecoilRoot, useRecoilValue,useSetRecoilState } from 'recoil'
import { counterAtom, evenSelector } from './store/atoms/counter'
function App() {
    return <div>
        <RecoilRoot>
            <Buttons />
            <Counter />
            <IsEven />
        </RecoilRoot>
    </div>
}
function Buttons(){
    const setCount = useSetRecoilState(counterAtom);

    function increase(){
        setCount(c => c+2)
    }
    function decrease(){
        setCount(c => c-1)
    }
    return <div>
        <button onClick= {increase}>Increase</button>
        <button onClick= {decrease}>Decrease</button>
    </div>
}
```

```jsx
function Buttons(){
    const setCount = useSetRecoilState(counterAtom);

    function increase(){
        setCount(c => c+2)
    }
    function decrease(){
        setCount(c => c-1)
    }
    return <div>
        <button onClick= {increase}>Increase</button>
        <button onClick= {decrease}>Decrease</button>
    </div>
}
function Counter(){
    const count = useRecoilValue(counterAtom);
    return <div>
        {count}
    </div>
}
function IsEven(){
    const even = useRecoilValue(evenSelector)
    return <div>
        {even ? "Even" : "Odd"}
    </div>
}
export default App
```

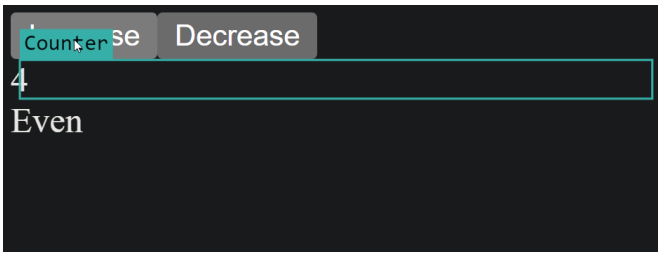
SO WHAT WE ARE DOING IS SAVING THE NO. OF RE RENDERS:

So we have a counter which will be rendering everytime, but we also have to show that its even or odd. When the no. is divisible its even and when we are again adding 2 to increase again It will be even only so to save that re render we have a selector which re renders the isEven function only when the no. changes from even to odd or vice versa which helps in

reducing the re render. Otherwise if we don’t have that then the isEven funciton would have re render at everytime.

ISEVEN SELECTORS – It’s a derived state of counter atom, its like subset of the atom(isEven is subscribed to iseven selectors)

COUNTER – Subscribed to Coutner Atom



In this just the counter changes

In this both changes bcoz selector changes