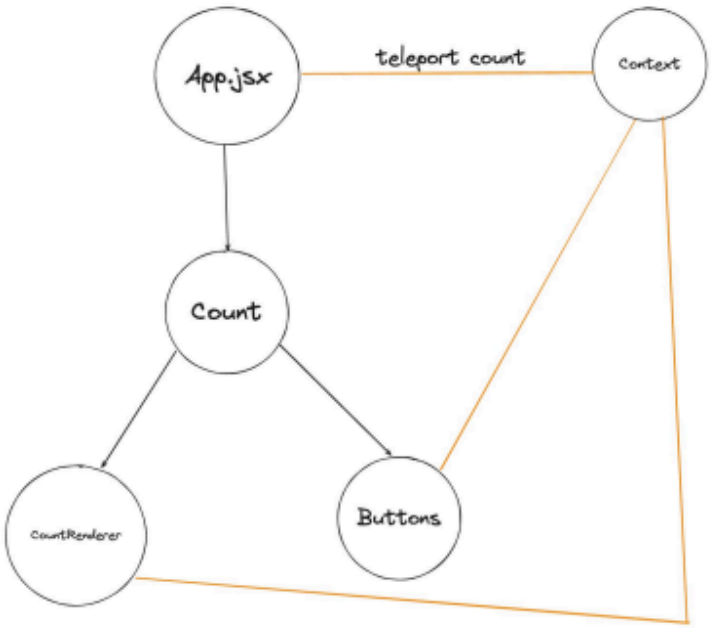
Context, State management, recoil

(https://github.com/100xdevs-cohort-2/week-7)

State management is very hard in React if the App scales.

React Introduced Context API to solve the problem of prop drilling.



We can define state outside React Component tree.

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

  // wrap anyone that wants to use the teleported value inside a provider
  return (
    <div>
      <CountContext.Provider value={count}>
        <Count setCount={setCount} />
      </CountContext.Provider>
    </div>
  )
}

function CountRenderer() {
  const count = useContext(CountContext);

  return <div>
    {count}
  </div>
}
```

App.jsx

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

export const CountContext = createContext(0);
```

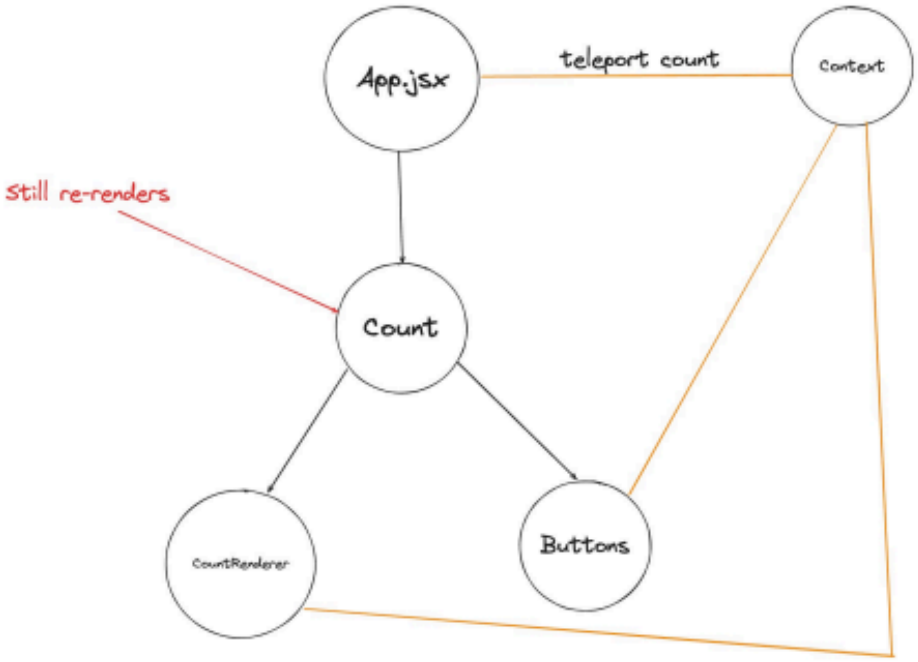
context.js

```jsx
import { useContext, useState } from "react"
import { CountContext } from "./Context";

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

  return (
    <div>
      {/* we need to wrap all the component inside provider */}
```

```
      <CountContext.Provider value={count}>
        <Count  setCount={setCount}/>
      </CountContext.Provider>
    </div>
  )
}

function Count({setCount}){
  return <div>
    <CountRenderer />
    {/* Directly teleported */}
    <Buttons  setCount={setCount}/>
  </div>
}

function CountRenderer(){
  const count =useContext(CountContext);
  // acess to count without being passed as props
  return <div>
    <b>
      {/* we may assume that c2 is not re-rendered */}
      {/* whoever did useContext(count COntexf a) */}
      {count}
    </b>
  </div>
}

function Buttons({setCount}){
  const count = useContext(CountContext);
  // const {count,setCount} = useContext(CountContext);
  return <div>
    <button onClick={()=>{
      setCount(count+1)
    }}>Increase</button>

    <button onClick={()=>{
      setCount(count-1)
    }}>Decrease</button>
  </div>
}
```

```
export default App
```

Problem with context? Doesn’t fix re-rendering only fixes prop drilling



State management Libraries help us to both decrease re-renders and prop drilling

State Management

What is state management

A cleaner way to store the state of your app

Until now, the cleanest thing you can doo is to 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)

Any react Project can be divided into two major parts

- State
- Components

With state management library we can separate two parts
With the use of recoil
We will now define all our component in the separate folder and the State in separate folders called Store.

State management using Recoil

(Why Recoil??)

https://youtu.be/_ISAA_Jt9kI?si=HvmxZsvkruH7RUUq

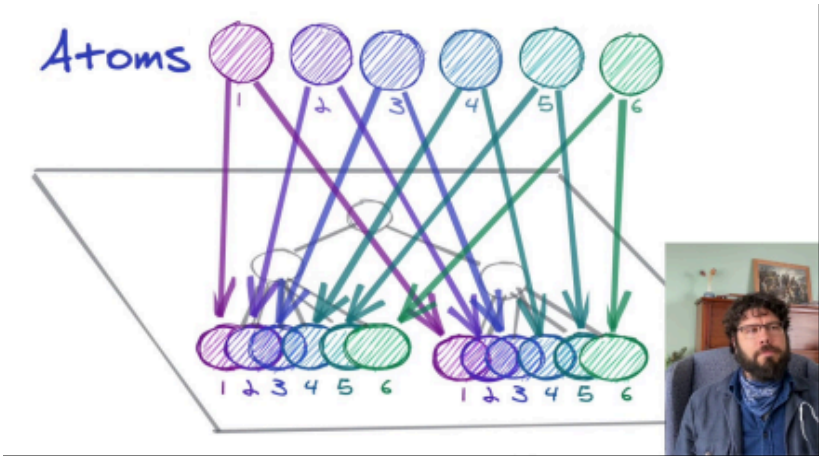
Recoil

A state management library for React.
Other popular ones are :

- Zustand
- Redux

These all lets us divide our project into two parts Component and States

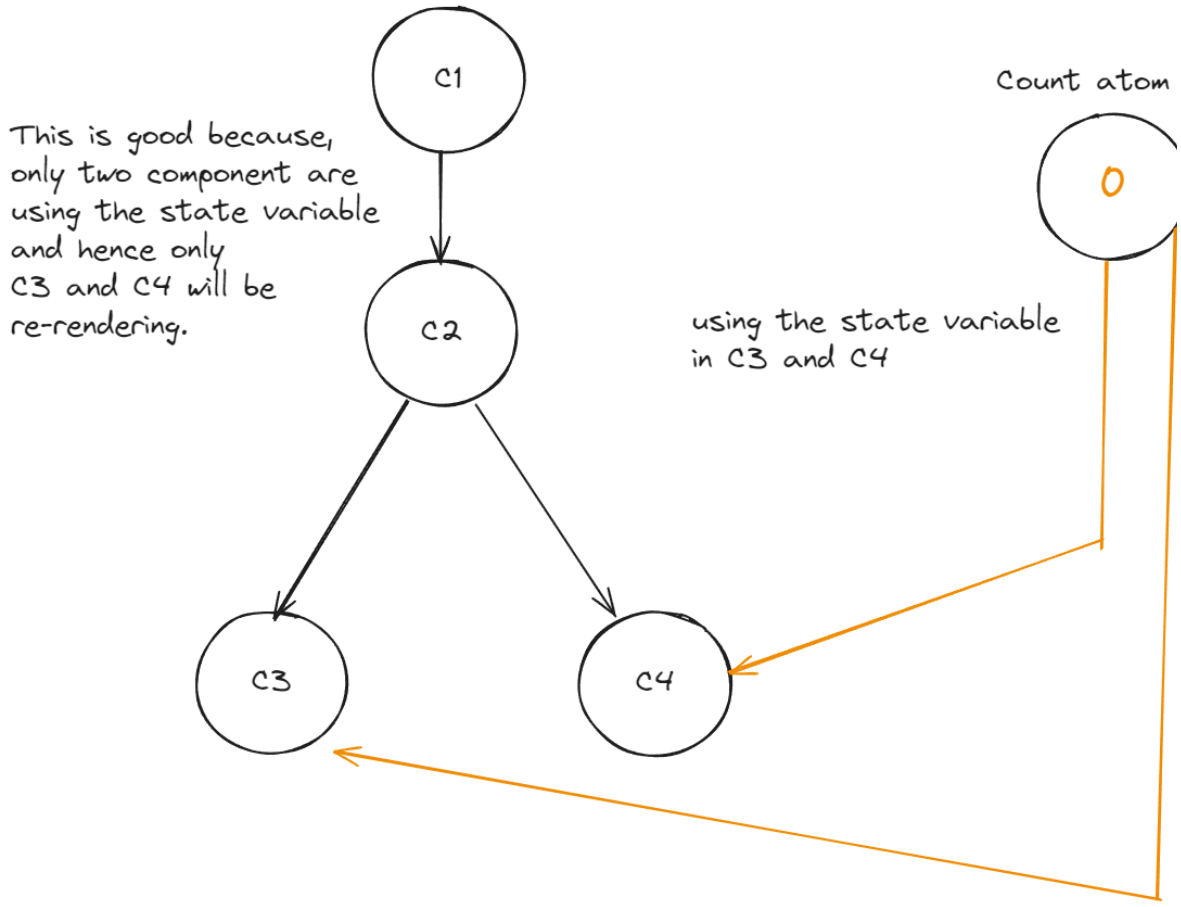
- Recoil has a concept of an **atom** to store the state
- An atom can be defined outside the component
- Can be teleported to any component



Creating Atoms is what mostly we do in Recoil.

Atom —----------------------> useState
We will update useState by setCount
In atom there is also way to get and update the Atom(state variable)

Biggest Benefit of the atom is that now we can define our component tree however we want. And out Atom will always be define outside.



Project using this will have store folder where there is separate State management logic.

npm install recoil

Things to Learn:

- RecoilRoot
- atom
- useRecoilState
- useRecoilValue
- useSetRecoilState

- selector

src -> store -> atom -> count.jsx

```
import {atom} from "recoil"

export const countAtom = atom({
    key:"countAtom",
    default:0
});

// export const todoAtom = atom({
//     key:"todoAtom",
// //  make sure the key is unique, we should not have same keys for two
Atoms
//     default:0
// })
```

Now our main project doesn't need any state variable and now we can get rid of useState() and useContext() . We have defined all the state in folder called count.jsx

Now we have to use it

Go to **App.jsx** and get rid of all the context logic and state variables.

```
function App() {
  return (
    <div>
      {/* we need to wrap all the component inside provider */}
        <Count/>
    </div>
  )
}

function Count(){
  return <div>
    <CountRenderer />
```

```
    <Buttons />
  </div>
}

function CountRenderer(){
  return <div>
    <b>
    </b>
  </div>
}

function Buttons(){
  return <div>
    <button onClick={()=>{
    }}>Increase</button>

    <button onClick={()=>{
    }}>Decrease</button>
  </div>
}

export default App
```

Now lets replace it with recoil logic:

Now we need count(**just the value**) in CountRenderer and setCount in Buttons component.

Wherever we want to use Recoil logic we have to wrap it into RecoilRoot

[count,setCount] = useState(0);

[_ , _] useRecoilState

useRecoil value

useSetRecoilValue

```jsx
import { RecoilRoot,useRecoilState,useRecoilValue } from "recoil"
import { countAtom } from "./store/atom/count"

function App() {
  return (
    <div>
      <RecoilRoot>
      {/* we need to wrap all the component inside provider */}
        <Count/>
        </RecoilRoot>
    </div>
  )
}

function Count(){
  return <div>
    <CountRenderer />
    <Buttons />
  </div>
}

function CountRenderer(){
  const count = useRecoilValue(countAtom)
  // it will give performance benefit if we only want value
  // different from
  // const [count,setCount] = useRecoilValue(countAtom)

  return <div>
    <b>
      {count}
    </b>
  </div>
}

function Buttons(){
  const [count, setCount] = useRecoilState(countAtom)
  return <div>
    <button onClick={()=>{
      setCount(count+1)
    }}>Increase</button>
```

```
    <button onClick={()=>{
      setCount(count-1)
    }}>Decrease</button>
  </div>
}

export default App
```

All state values come directly inside the component.
The things that actually create our website/ global variable we will use, useState().

For forms react-hook-forms

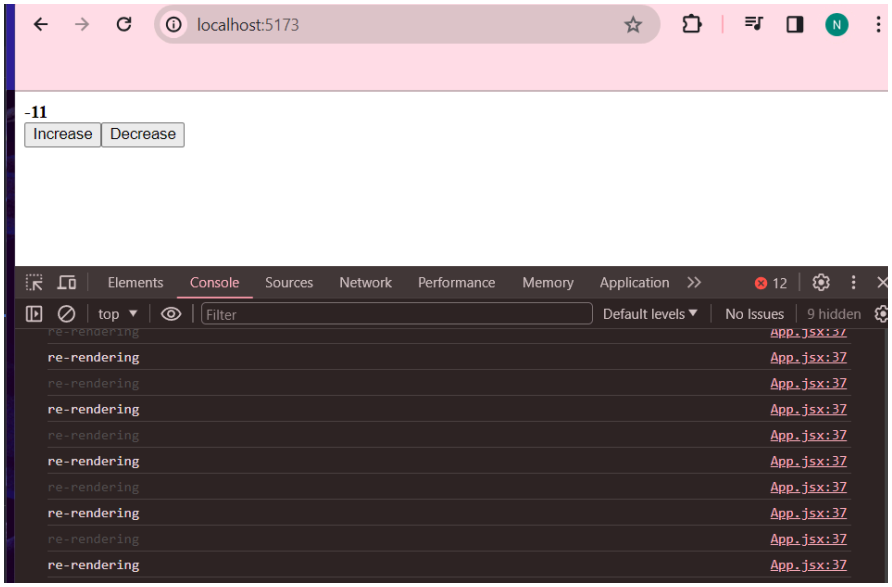
To check if Recoil has make our app performant , do console.log in different components.

```
function Count(){
  console.log("re-renders")
  return <div>
    <CountRenderer />
    <Buttons />
  </div>
}
```

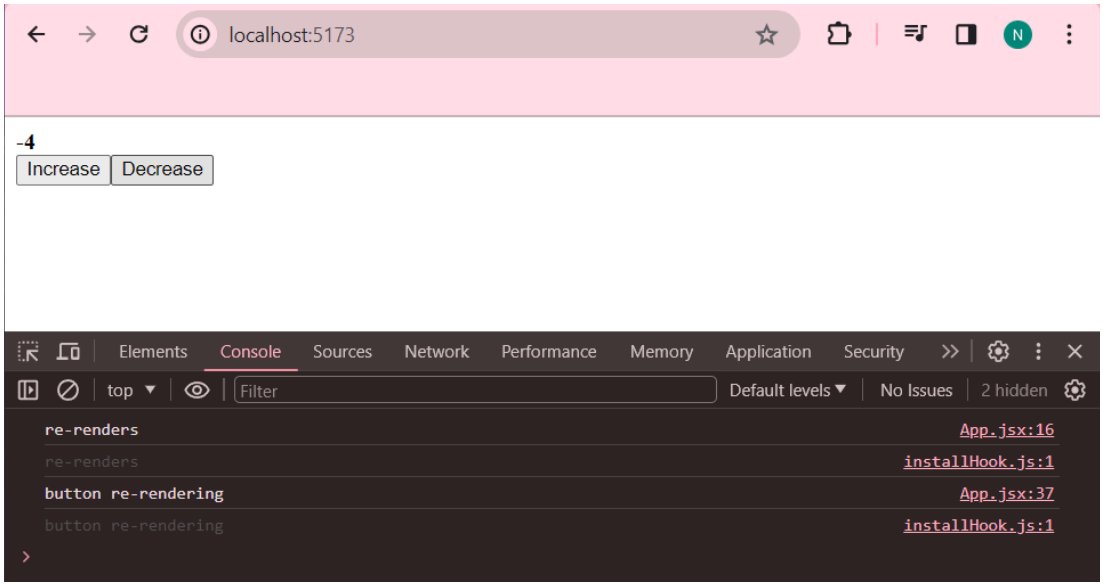
Here is little problem that button component is re-rendering although it is not changing.

```
function Buttons(){
  console.log("re-rendering");
  const [count, setCount] = useRecoilState(countAtom)
  return <div>
    <button onClick={()=>{
      setCount(count+1)
```

```
    }}>Increase</button>

    <button onClick={()=>{
      setCount(count-1)
    }}>Decrease</button>
  </div>
}
```




```
function Buttons(){
  console.log("button re-rendering");
  const setCount = useSetRecoilState(countAtom)
  return <div>
    <button onClick={()=>{
      setCount(count=>count+1)
    }}>Increase</button>

    <button onClick={()=>{
      setCount(count=>count -1)
    }}>Decrease</button>
  </div>
}
```



Now button component has stopped re-rendering.

Most Expensive part of react is re-rendering

selector

Let's say I ask you to render it is even if the current count is even.

```
function CountRenderer(){
  console.log("CountReRenders")
  const count = useRecoilValue(countAtom)
  return <div>
    <b>
      {count}
    </b>
    <EvenCountRenderer/>
  </div>
}
```

```
function EvenCountRenderer(){
  const count = useRecoilValue(countAtom);
  return <div>
    {(count % 2==0) ? "It is even":null}
  </div>
}
```

It is not optimized.

```
function EvenCountRenderer(){
  const count = useRecoilValue(countAtom);
  const isEven = useMemo(()=>{
    return count % 2 == 0
  },[count])
  // only re-renders when count changes , if we want to do same in Recoil
we use the selector
  return <div>
    {isEven ? "It is even":null}
  </div>
}
```

Lets see how we will do samething in Recoil using **selector .**

src -> store -> atoms -> count.jsx

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

export const countAtom = atom({
    key:"countAtom",
    default:0
});

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

Selector just depend on countAtom , it can depend on many things

We have to give a function inside which we have access to get

get: (props) => {

const count = props.get(cointAtom);

```
    return count % 2;
}
```

Q/Na:

1. When to use useState() and when to use atom , let take example of the Github website , the repository name etc should be stored in the atom and the things which we search by writing should be stored in useState.
2. If we use memo then it will be hard to connect two or more condition which are in different component.Hence it becomes had using useMemo() since it depend on ten different components .