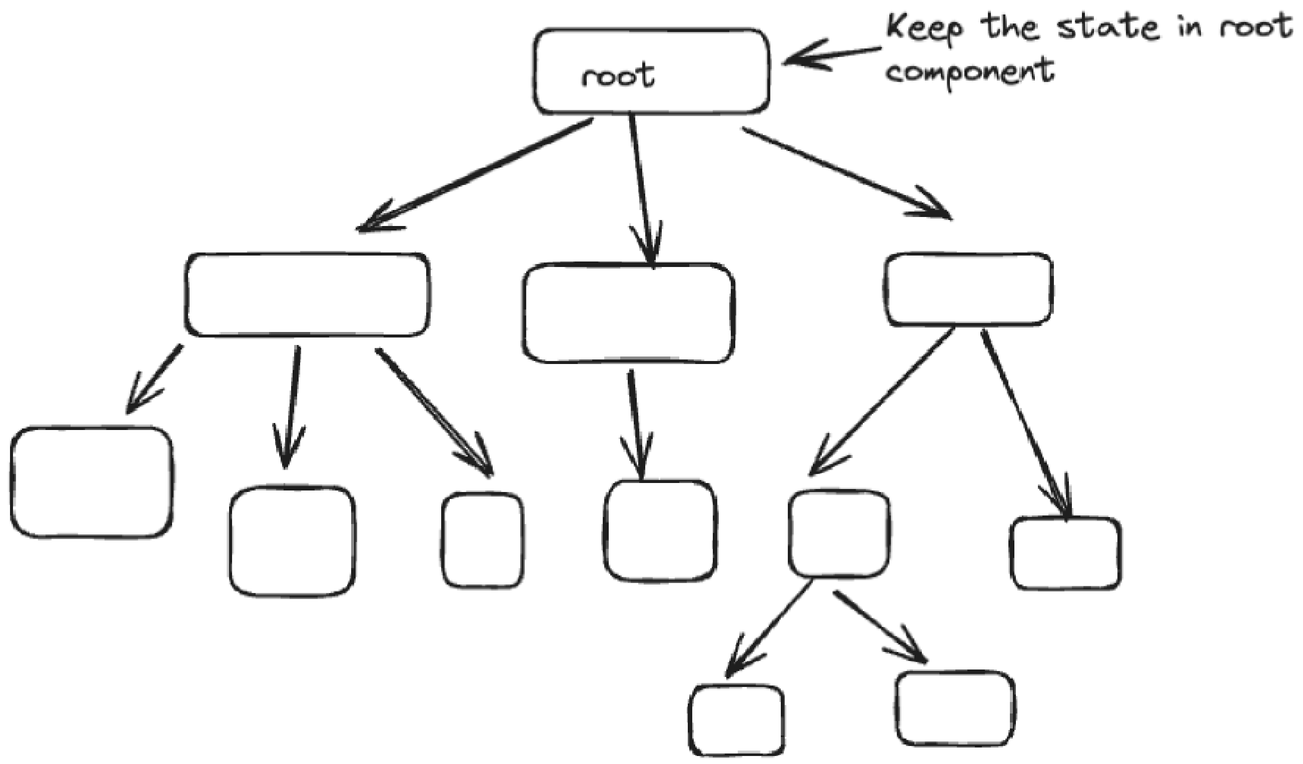
If we have a very complex component hierarch, then if we want to update a state in a component and see it's effects in another component, then it will be pretty challenging task. We might have to use a lot of callbacks and then only we will be pass things around. But having too many callbacks passing the states here and there, will make the components very tightly coupled.

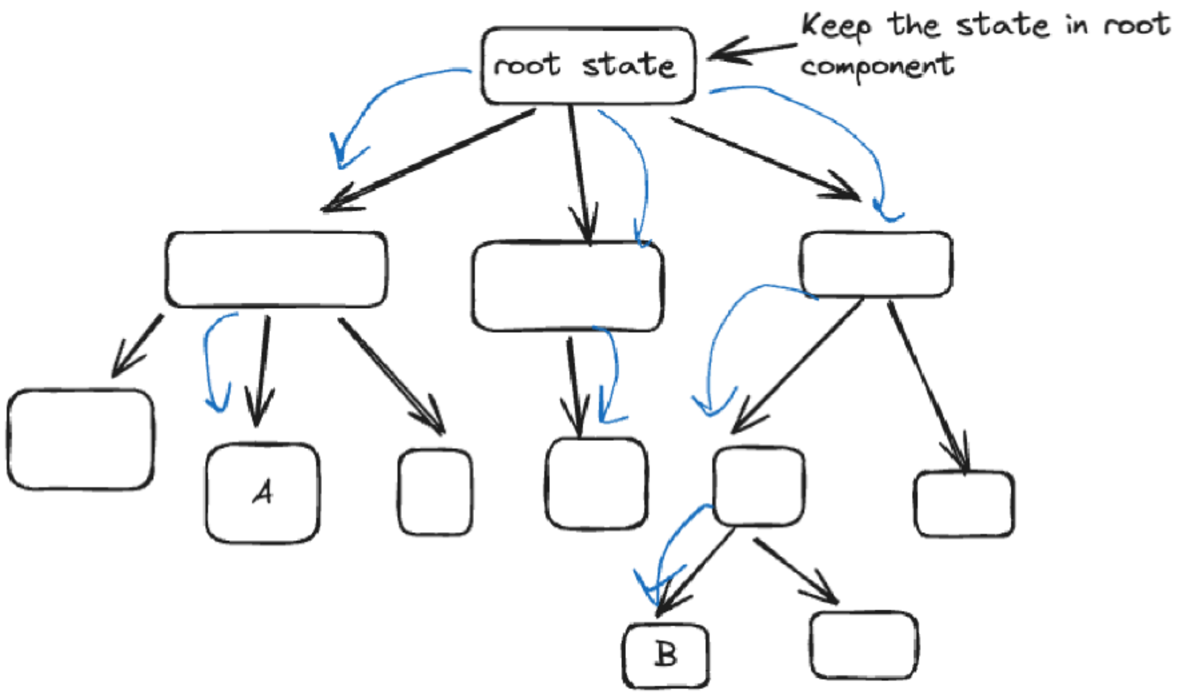
That means we need to have a better mechanism.

That's where lifting the state up in the component hierarchy comes into the picture. Meaning of lifting the state up is that, we try to bring up the state at the very top level of the component hierarchy.



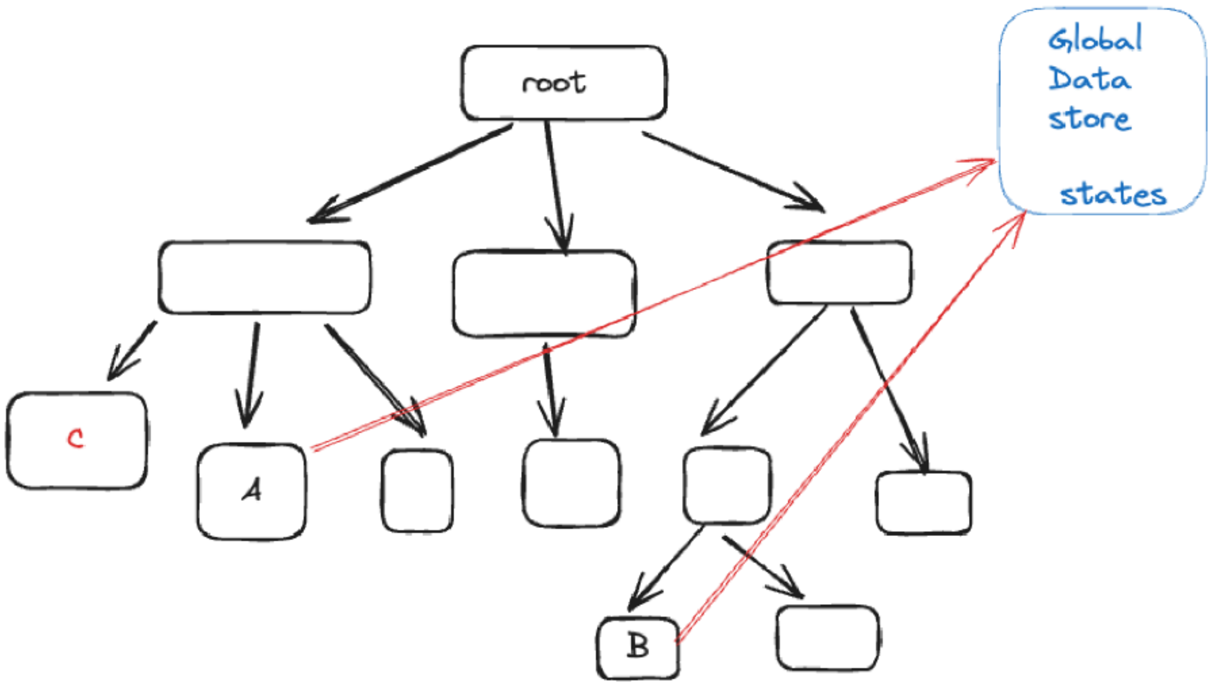
With this approach we can actually have the states mentioned in the root component and then root component will pass down these states and their updater function in the component hierarchy.

But there is one more problem with this approach.



From the very top of the hierarchy we need to pass our states as a prop very low level in the hierarchy, now if the application is super complex, then this props passing can be extremely hard to maintain. And this problem is also referred as `Prop drilling`.

Maintain the state as a global store:



If we want a couple of components to share a state, then we can maintain a global data store, which will be having some states, which can we access by any component and can be updated by any other component. For example, may be A component updates the state and B component consumes the state, so when A triggers an update, B will re-render itself.

How to achieve this;

- React context API
- Redux
- Zustand
- MobX
- and more.....

React context API :

React from some of the latest versions, started supporting an in house solution for better state management. This is called as context API. Context api helps us to describe a state storage and then help us to access it from the component hierarchy without prop drilling.

1. Create a context object

```
import { createContext } from "react";
export const CurrencyContext = createContext();   //it creates context object
this will hepls us to maintain global state
```

2. Once we have created the context object, then we can use it to store some states and access them.
3. To make the context accessible across multiple components, we can wrap those components inside `Context.Provider` component. It is a component, that will be automatically given to use by the context object.

```
function App() {
  const [currency, setCurrency] = useState('usd')
  return(
   <>
   <CurrencyContext.Provider value={ { currency, setCurrency} }>  {/* When we
put home inside currencyprovide Home is accessible anywhere in the Home
component */}
    <Home />
   </CurrencyContext.Provider>
   </>
  )
}

export default App;
```

4. Here `CurrencyContext.Provider` is a component that is automatically created by the context object. We will wrap out `Home` component inside it, due to which in the complete hierarchy of the home component we can access the CurrencyContext.
5. Now in the provider, we can add a value prop, which takes an object and inside this object we can store any state or functions, that we want to make accessible inside our `Home` component. The line `value={ { currency, setCurrency} }` passed currency and setCurrency in an object which is assigned to the value prop.
6. To access this context object anywhere in the `Home` component we can use the `useContext` Hook. This takes in the context object as a parameter and return the value object to us, that we can de-structure, however we want.

```
function Navbar() {
 const { setCurrency } =  useContext(CurrencyContext);
 return(
       <>
       .....
       </>
 );
 }
```

```
function CoinTable() {
  const {currency} = useContext(CurrencyContext);
  return(
    <>
      {currency}
      .....
    </>
  )
```

Build :

Build is a process that internally uses some tools that helps us to compile the whole project which has multiple separate modules if we compile individual modules it will take lot of time to run the project `Build tools` simplify the dependency graphs whatever modules compile first they automatically compile step by step and bring our project in a runable state .

Redux :(my-redux-project)

Redux is an open-source JavaScript library that manages application state, and is often used with React to build user interfaces. It provides a predictable and maintainable way to write applications that can run in different environments and are easy to test. It provides a way to store and manage the state of your application in a single object called the store.

Reducer function:

Reducer function is a function that updates an application's state in response to actions. Reducers are pure functions that are the only source of state change in a Redux application. They are an important part of Redux state management.
Reducer function takes two parameters

1. `object to be updated` (incoming object)
2. `action` (by using this object we can decide how to update the incoming object)

- action object has `two main properties` :
  `type` : "name of action"
  `payload` : any extra object that you need to action to fill full

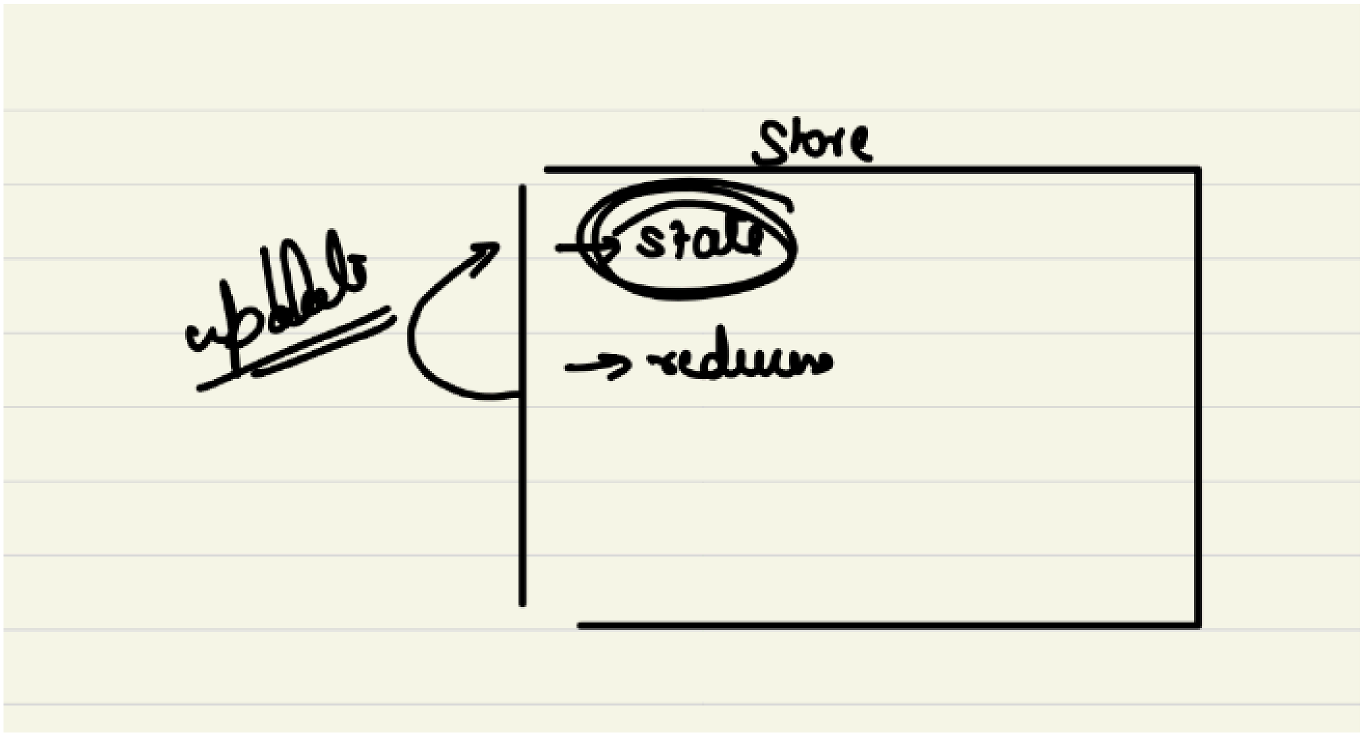
inside reducer function to write the algorithm to update the object. Output of the reducer function is updated object.

```
function numberReducer(incomingObject, action) {

    if(action.type == 'ADD') {
        const numberTOAdd = action.payload;
        return {
             ...incomingObject,
            number: incomingObject + numberTOAdd
        }

    }else if(action.type == 'SUBTRACT') {
        const numberTOSubtract = action.payload;
        return {
             ...incomingObject,
            number: incomingObject + numberTOSubtract
        }

    }else if(action.type == 'INCREMENT') {
        // we don't need payload for increment // action: {type: 'INCREMENT' ,
payload: null}

        return {
            ...incomingObject,
            number: incomingObject.number + 1
        }

    }else if(action.type == 'DECREMENT') {

        // we don't need decrement

        return {
            ...incomingObject,
            number: incomingObject.number - 1
        }

    }else {
        return incomingObject;

    }
}
const init = { number: 1 };
console.log(numberReducer(init,{type:'ADD',payload:10}));
```

`action.type` : defines which algo to use
`action.payload` : defines any input required for the algo to work
in payload we can pass any value number null etc;

So, Redux is a state management tool and it is responsible for storing and updating the states for updating the states it uses reducer function.

Redux have couple of functions ;
`createStore` : createStore function creates a common global store to keep the stores. it takes two arguments reducer and initialState
It takes reducer function as a argument.
`initialState`



to install redux:

```
npm i redux
```

It is core redux library which is independent of react or any other framework.

- Reducers can update an object
- The object we want yo update (state) is stored in the store object.
- To trigger our reducer so that we can update the state stored in store we use `dispatch method`. what this method will do is it will trigger the reducer object and update the state object.

```
import { createStore } from "redux";
// Reducer function

function numberReducer(incomingObject, action) {

    if(action.type == 'ADD') {

        const numberTOAdd = action.payload;

        return {

            ...incomingObject,

            number: incomingObject.number + numberTOAdd
        }

    }else if(action.type == 'SUBTRACT') {

        const numberTOSubtract = action.payload;

        return {

             ...incomingObject,

            number: incomingObject.number - numberTOSubtract

        }


    }else if(action.type == 'INCREMENT') {

        // we don't need payload for increment // action: {type: 'INCREMENT' ,
payload: null}

        return {

             ...incomingObject,

            number: incomingObject.number + 1
        }
    }else if(action.type == 'DECREMENT') {

        // we don't need decrement

        return {

             ...incomingObject,

            number: incomingObject.number - 1

        }

    }else {

        return incomingObject;

    }

}
```

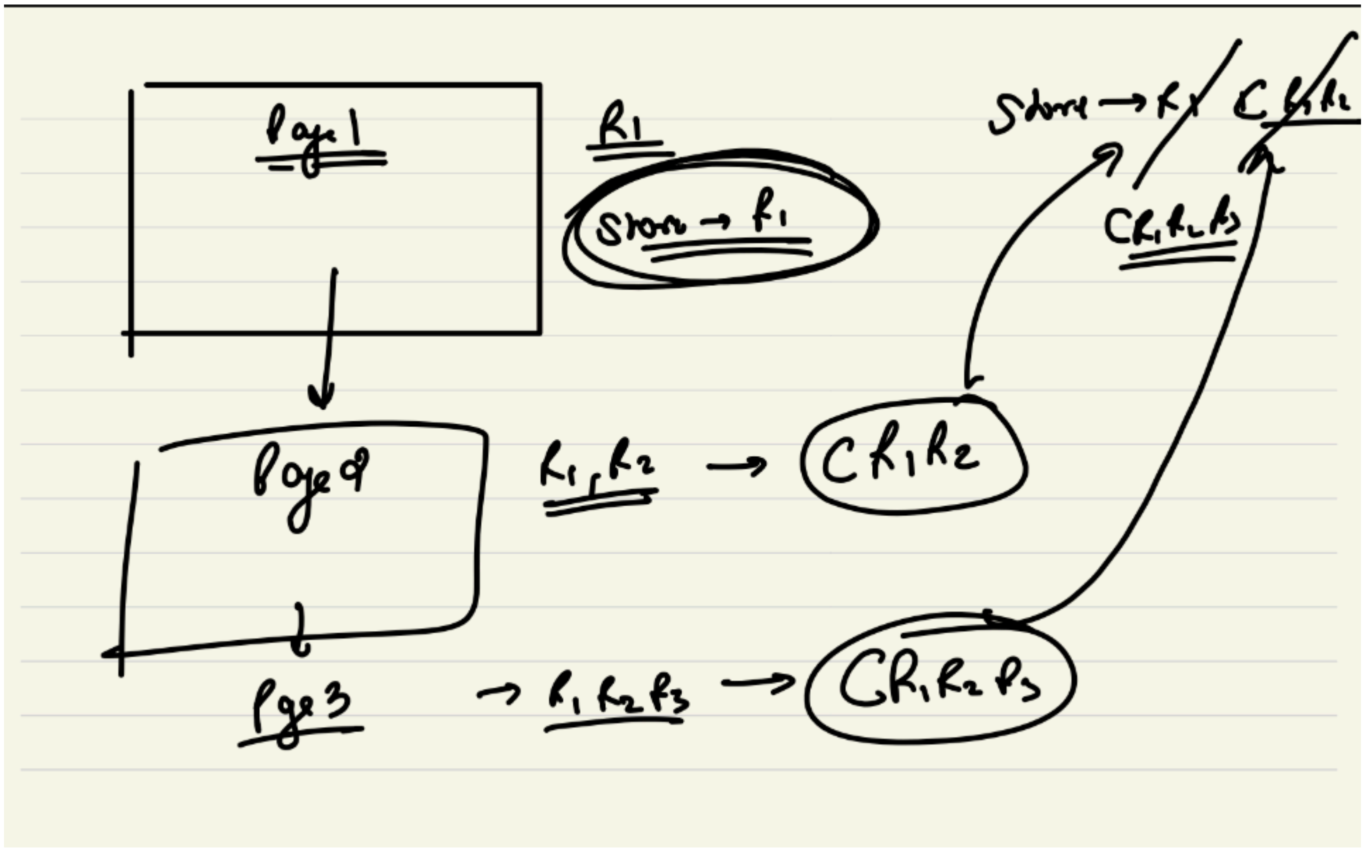
```
const init = {number:1};

console.log(numberReducer(init,{type:'ADD',payload:10}));

// 10 + 1 = 11



 const initialState = {

    number: 0

 }

const store = createStore(numberReducer, initialState);

console.log(store);

console.log(store.getState());

store.dispatch({type: 'ADD', payload:17}); //dispatch takes an action

console.log(store.getState()); //this action has added 17 to the initial
number (0)
```

combine Reducers :

The combine Reducers function **allows you to combine multiple reducer functions into a single function that can be passed to the Redux store**.

Replace Reducer :

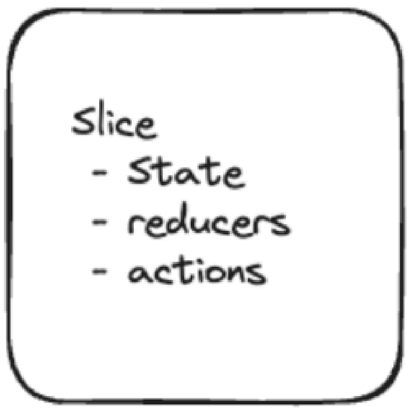
The Redux store exposes a replaceReducer function, which **replaces the current active root reducer function with a new root reducer function**.

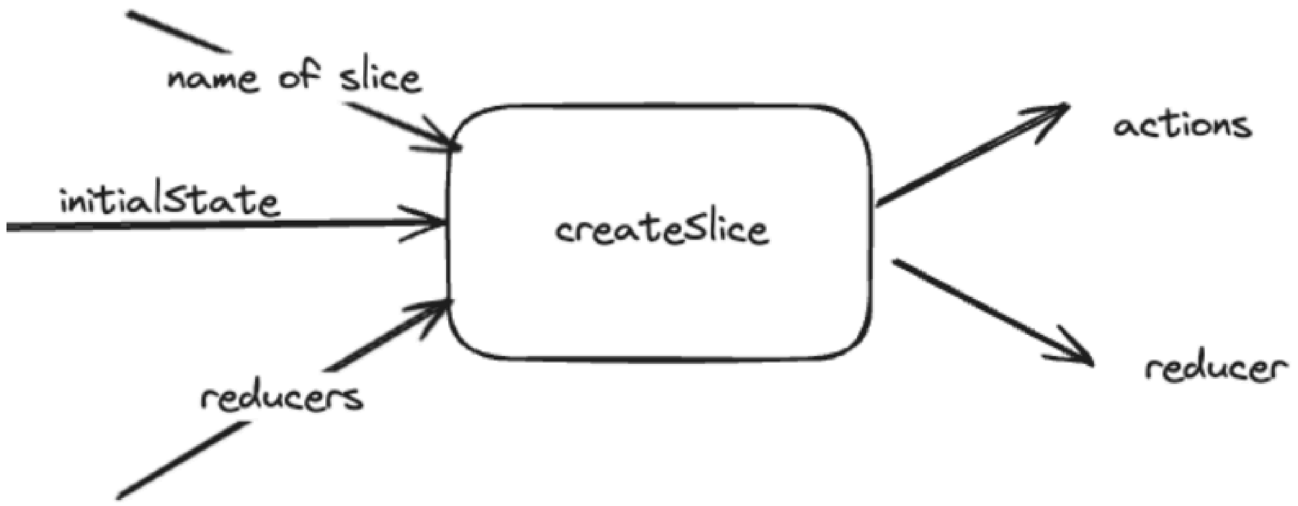


we have three reducer function R1 , R2 and R3 first we need to lazyly load R1 and then we have to lazyly load R2 , R3 at a time but createStore function only take 1 reducer.
To pass multiple Reducers apart form createStore function there is another method called as `combineReducers`.
combine Reducers combine your reducers see the above image
in page2 we have to lazyly load R2 and R3 so we are combining both R2 and R3 using combine-Reducers and then call `replaceReducer` to place with it R2 and R3.

Redux-toolkit :(files D:ProjectsReact/redux_and_testing)

For any particular state management in redux we need 3 parts `state, reducer, action.` what redux- toolkit says that if you want to define all of these you just define a `slice`. inside slice you define reducer and state and slice will automatically create actions for you and all of these things going to reside at one place.


Slice
- State
- reducers
- actions


name of slice
initialState
reducers
createSlice
actions
reducer