2024-09-15

Today's Content

- What and why of state management library
- What is redux
- Important concepts of redux
  - store
  - provider
  - action
  - dispatch
  - reducer
- useselector and usereducer hooks
- Integrating redux with react application
- Implementing redux with some examples

React Component

Let us create a component with the name Counter. In this component, we have two buttons one for incrementing and one for decrementing counter. The count value is displayed on the page. Write the code given inside that component.

```
import React, {useState} from 'react';
function Counter(){
    const {count, setCount} = useState(0);

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

    const handleDecrement = () => {
        setCount(count - 1);
    }
    return{
        <>
            <button onClick = {handleIncrement}> + </button>
            <h3>{count}</h3>
            <button onClick = {handleDecrement}> - </button>
```

```
        </>
    }
}

export default Counter
```

This component code has the following sections:

- **State management:** useState is providing the initial count value and `count + 1` and `count - 1` is used for incrementing and decrementing the count value.
- **Event handlers:** We attach and help in state management.
- UI
- Business logic

State Management Library

State management basically used for state management. **State management:** We usually do the following two things for state management:

- Set the state
- Update of the state

We have learnt that every react component has state management.

Let us assume we have an application, and let us assume it is a real-time application like LinkedIn, facebook, scaler, etc. It:

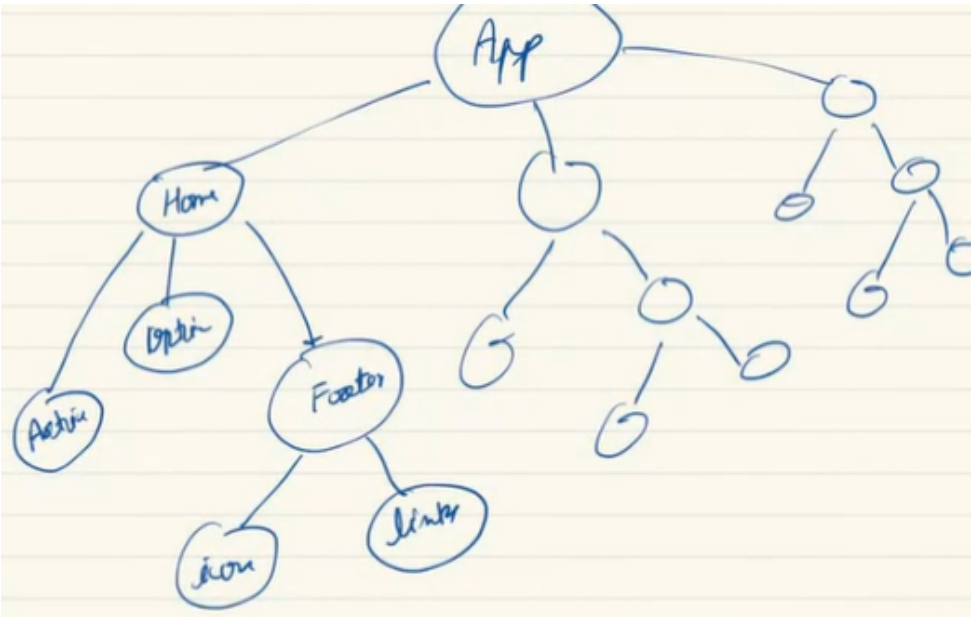
- Has 1000 of components. And that components are arranged in a certain manner, in any application where we want to generate business will have number of pages and number of sections and all these sections are nothing but component.
- There is also a problem called prop drilling.

Prop Drilling

Let us assume we have a app application, that has different components,

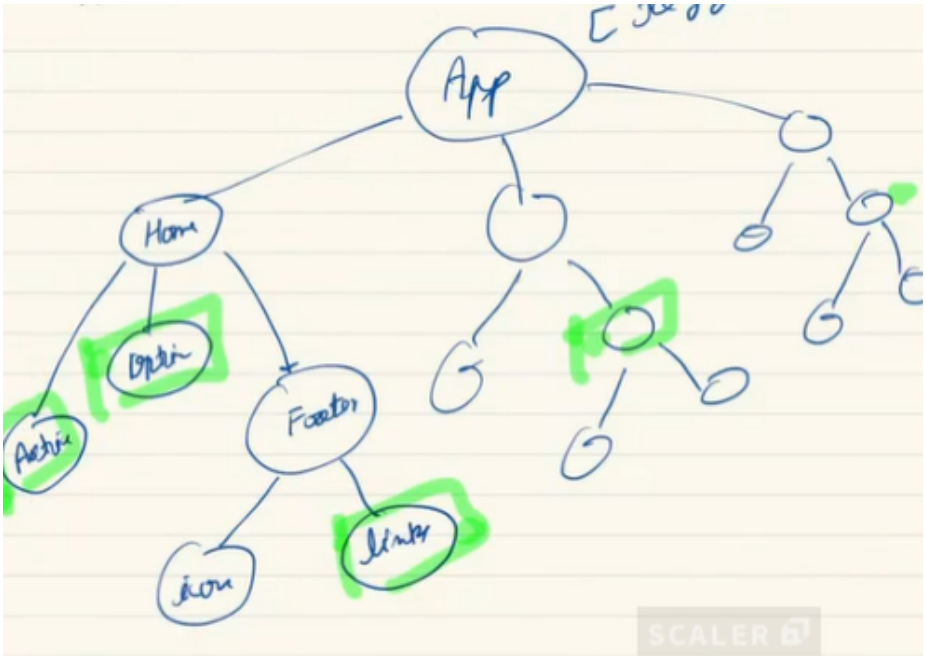
- **Home:** Home has options, articles and a footer and the footer has some icons and social links.

- And it has some other components.

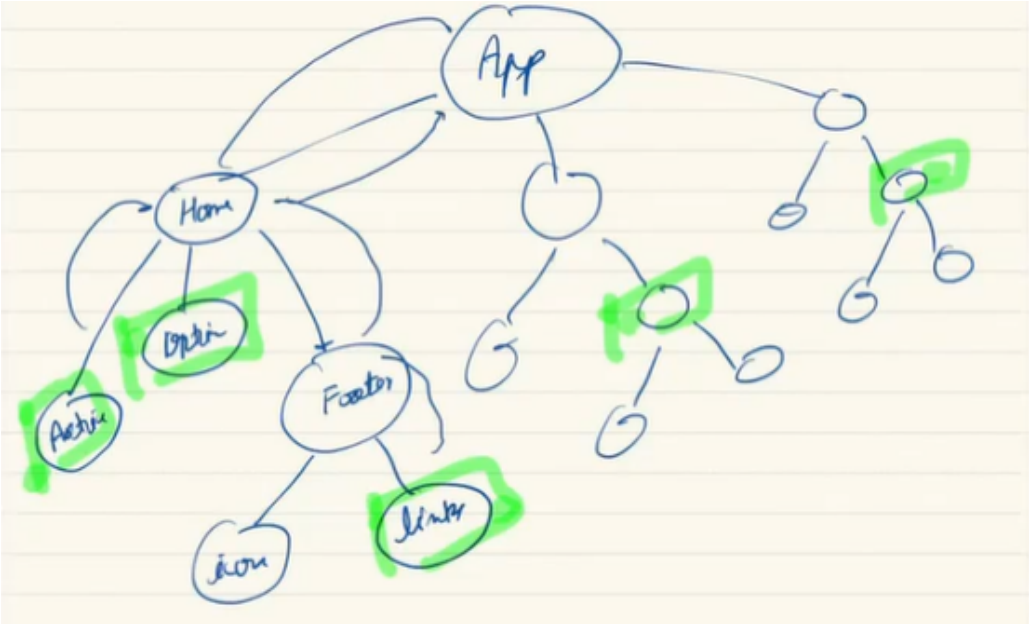


And we have a feature of the toggle theme, which toggles your theme between dark and light.

When the theme switches from one to another, only the article, options and footer components need to be changed.



If we want to change these components, we have to move the state if the state is shared.



- If an app has a `[toggleTheme]` state, then it will be passed to every component via the system so that they get to know about the dark/light theme.

Here we are simply passing props to the whole chain, this does not require, as if links need a theme, then it will be passed from the app to the footer then it will be passed to links, the above two passings are needless.

This is a problem that arises in big application that has a common state and that common state is required to be passed to multiple components.

The passing of props to all the elements is known as prop drilling, even when a few components do not require that prop.

Issue with 1000 of components

Require individual state management. Let us assume we are in a remote village and it has 2 homes and all the homes are on the river side and every house produces its electricity individually by the river. This will work for the village having fewer houses, but it will not work for the big cities. In this way every component has its own state management, then it becomes complicated for big applications.

Solution to prop drilling and multiple components of big application

The solution of passing state to different components and individual state management of every component is provided by the state management library.

State management library

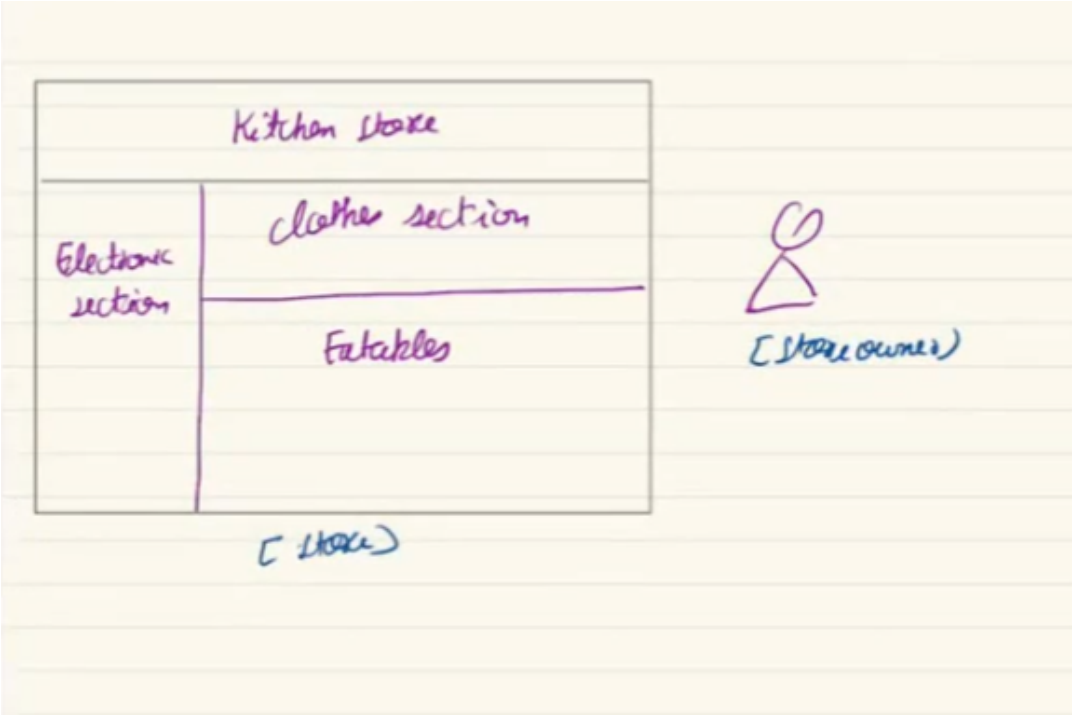
So according to the issues we have seen, the main responsibilities of state management library should be:

- There should be **central state management**, all the states that are required to be changed should be changed in one place.
- It should **avoid prop drilling**.

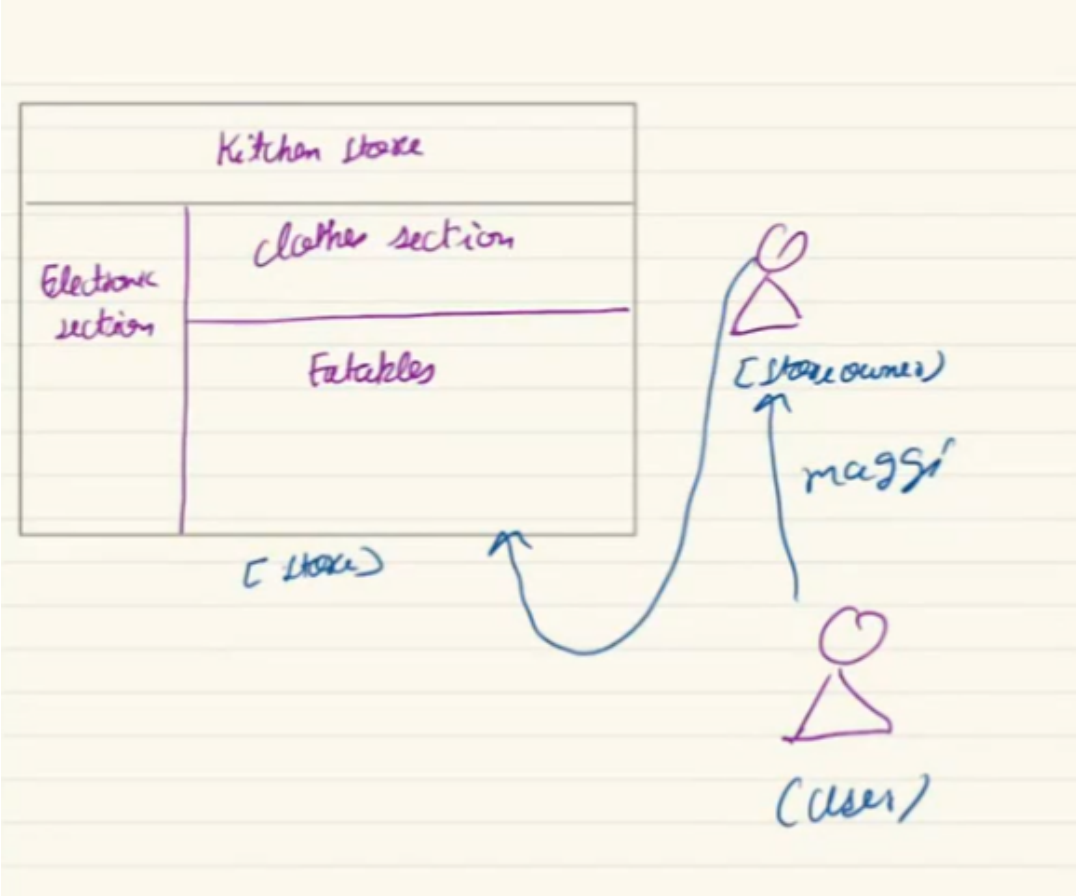
Redux

- Redux is a third-party javascript library for state management.
- It can be integrated with every front-end framework of JavaScript.
- We just need to install it.
- It gives a feature known as store where all the states are stored.
- It also provides a centralised state management feature with the help of a feature known as slice. Let us assume a simple grocery store, let us assume this shop has different sections:
- **Kitchen section:** For storing all the kitchen-related items.
- Electronics section
- Clothes section
- Eatables

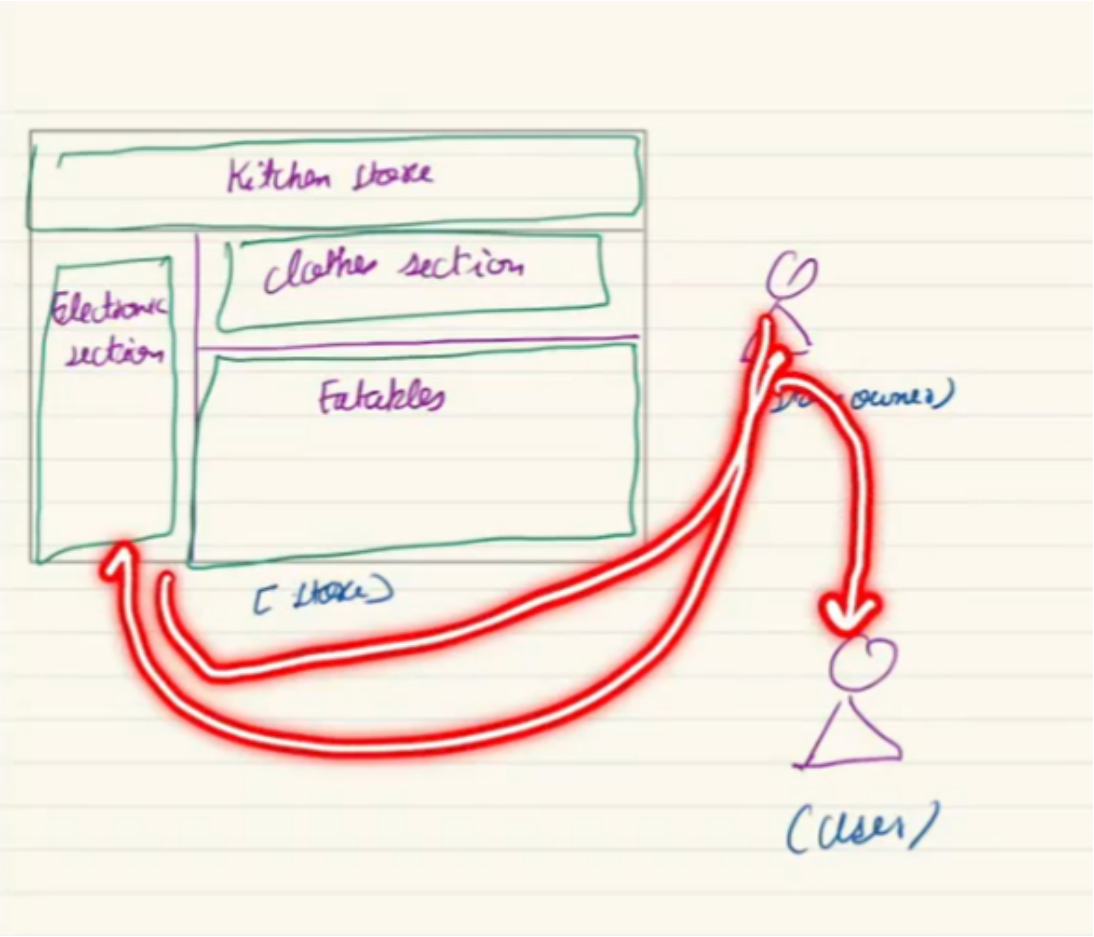
And let us assume there is a store owner



Let us assume a user arrives and asks for Maggie, then the owner will go to eatables and give you Maggie.



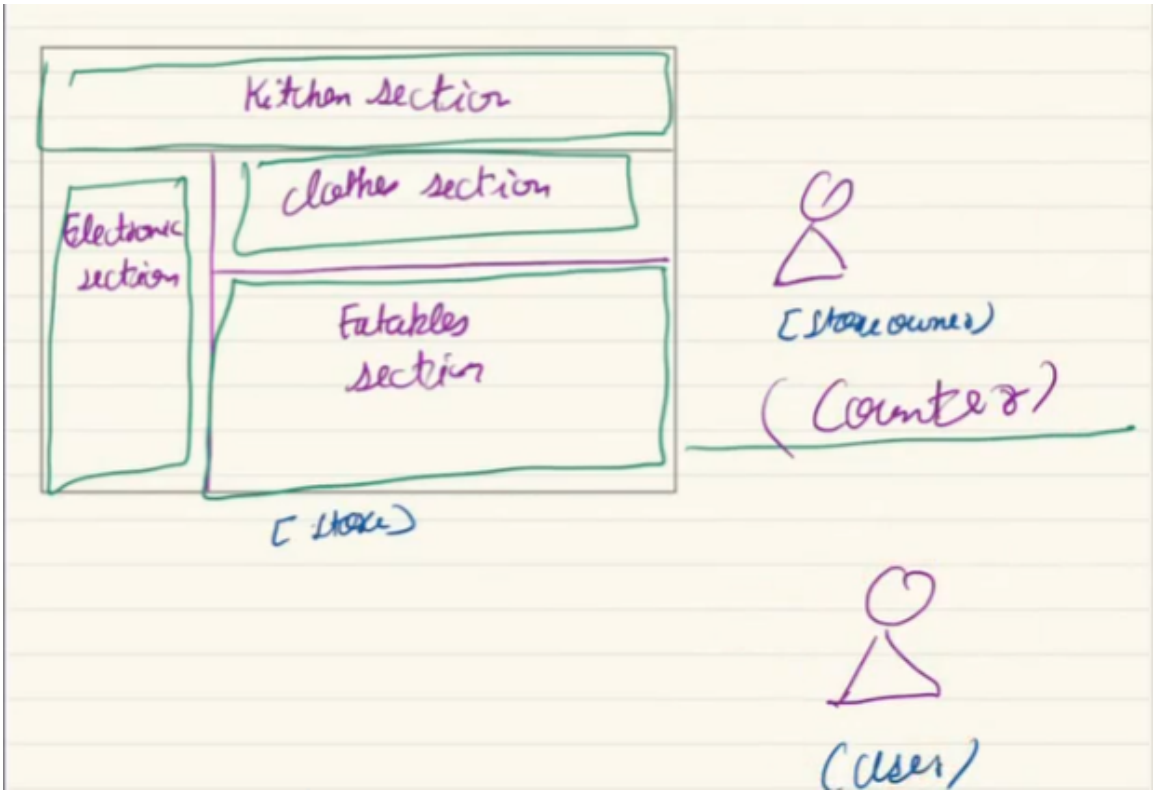
If a user asks for headphones, the owner will directly go to the electronics bring them and give them to the user.



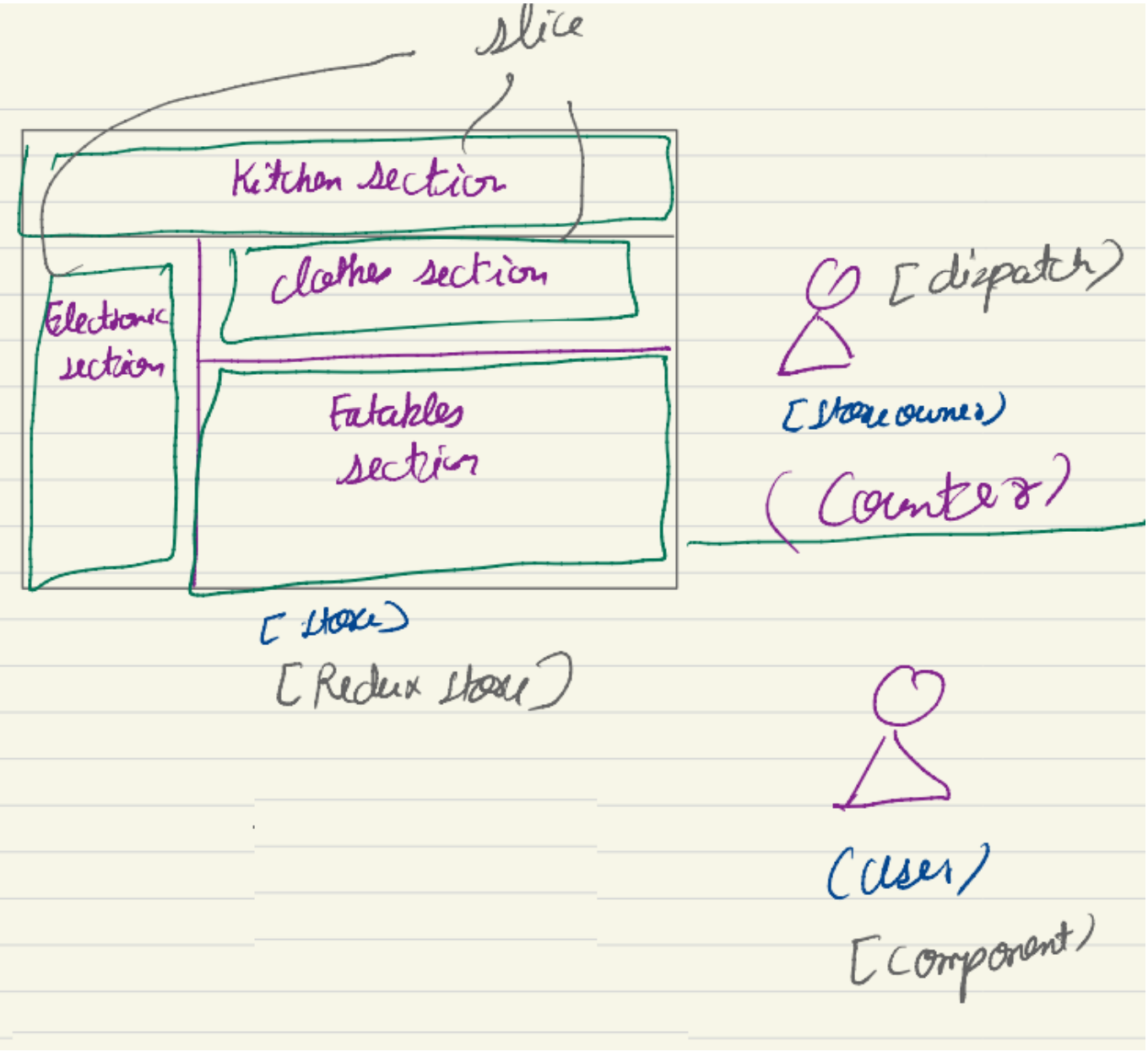
The job of the store owner is to go to a particular section, bring material and give it to the user.

Redux store is similar to grocery store and these sections are similar to slice. In redux, all the slices are combined to make the store.

Now assume the user is a component, the user will never directly go to the section to fetch the items, it is the responsibility of the owner to go to the section bring the material and come back to the counter to give the item to the user. We have a counter between the store owner and the user.



And **the store owner will work as dispatch.**



Installing redux

Open the terminal inside your application folder, and write the below commands:

```
npm i @reduxjs/toolkit
npm i react-redux
```

Now in `package.json`, a dependency for `@reduxjs/toolkit` and `react-redux` will be added.

State management has two things:

- Set the state(providing initial value)

- Updating the state.

With any state management library, we do not need to think about updating the state, all will be delegated to the state management library. And we do not need to use `useState` with `redux`.

Providing State to The Component

For this we are creating two components, create two folders inside the components folder:

- `normalComponents`: They have individual state management.
- `reduxComponents`: They delecates state management to the `redux`.

As we are taking the `Counter` component we have created as an example, so:

- Add the `counter` component in `normalComponents` with the name `Counter.jsx`.
- Add the `counter` component in `reduxComponents` with the name `CounterRedux.jsx`.

Inside `src`, create a folder named `redux`.

Inside the `CounterRedux.jsx`, we do not need to use `useState`, and we do not need to increment and decrement count value.

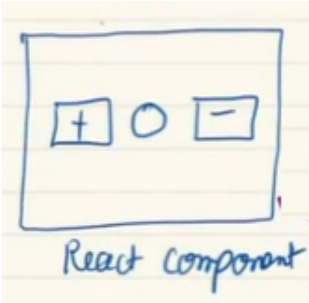
```
import React from 'react';
function Counter(){
    const handleIncrement = () => {
        console.log("increment will happen");
    }

    const handleDecrement = () => {
        console.log("decrement will happen");
    }
    return{
        <>
            <button onClick = {handleIncrement}> + </button>
            <h3>{count}</h3>
            <button onClick = {handleDecrement}> - </button>
        </>
    }
}

export default Counter
```

1. The first step is to get the state for my counter component using redux.

Let us assume we have a react component.

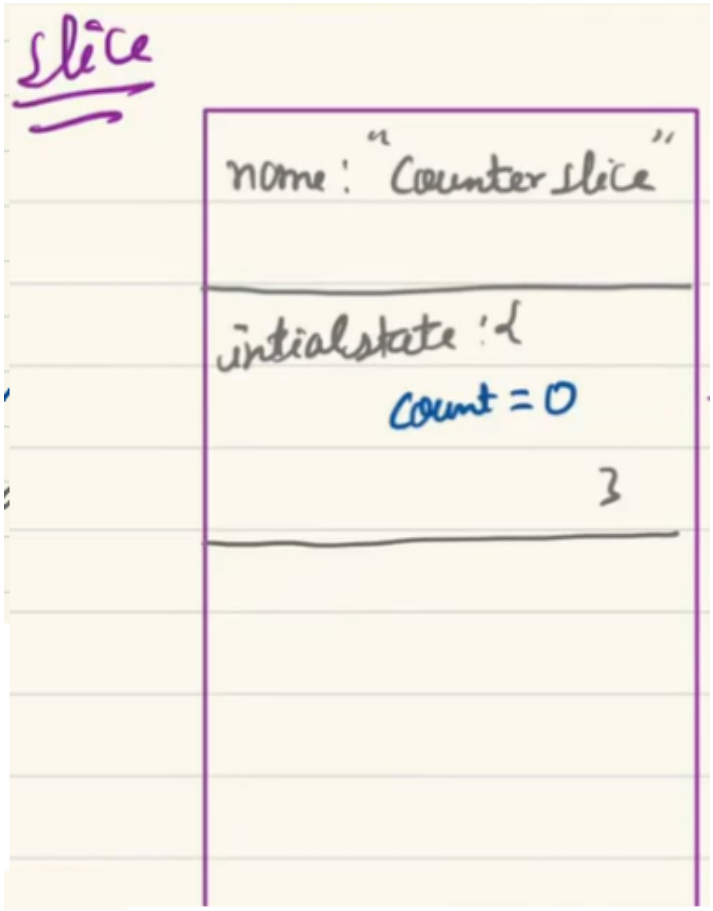


We have a slice which contains the information of the state, it has:

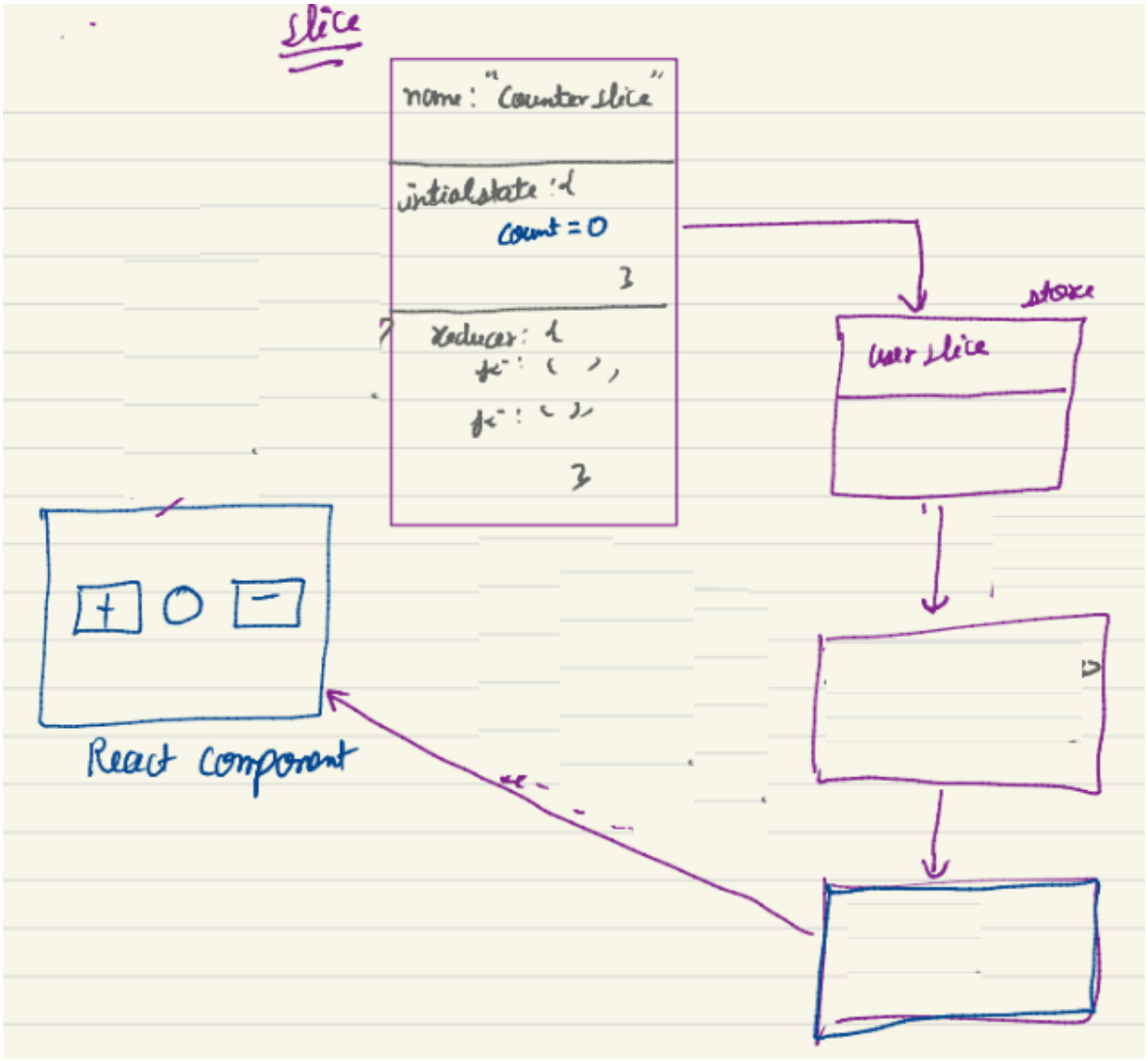
- name: CounterSlice
- initialState
  - count = 0

name is to identify different slices, multiple components have different slices, so name is used to identify the slice. **initialState** is an object that will contain the state variable, it defines the initial state for react component.

We give the above two properties to the slice.



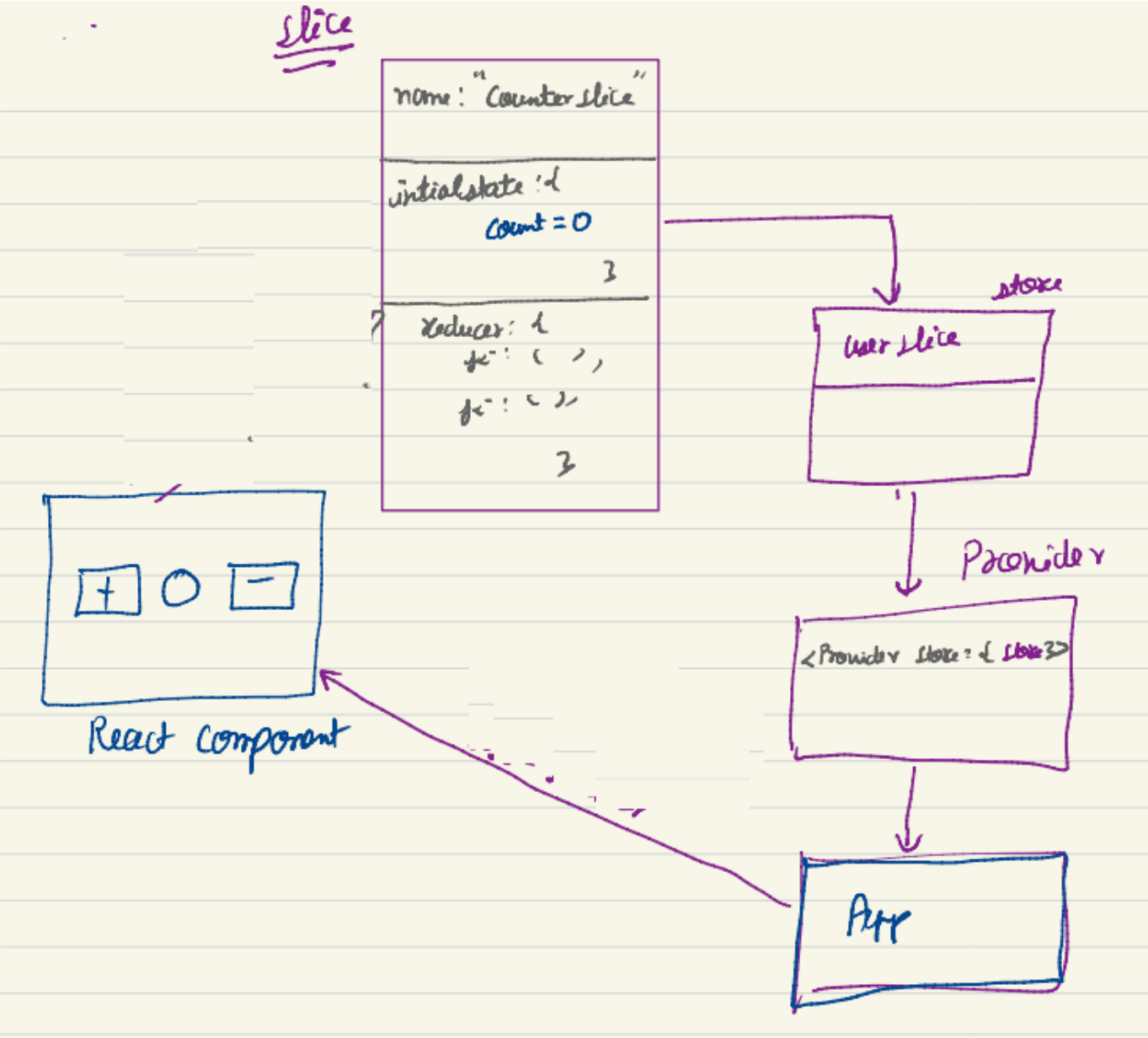
Now we have a store, which has userSlice, and the slice has an initial state.



We have a **provider** and we have an app component, and this app is a part of react.

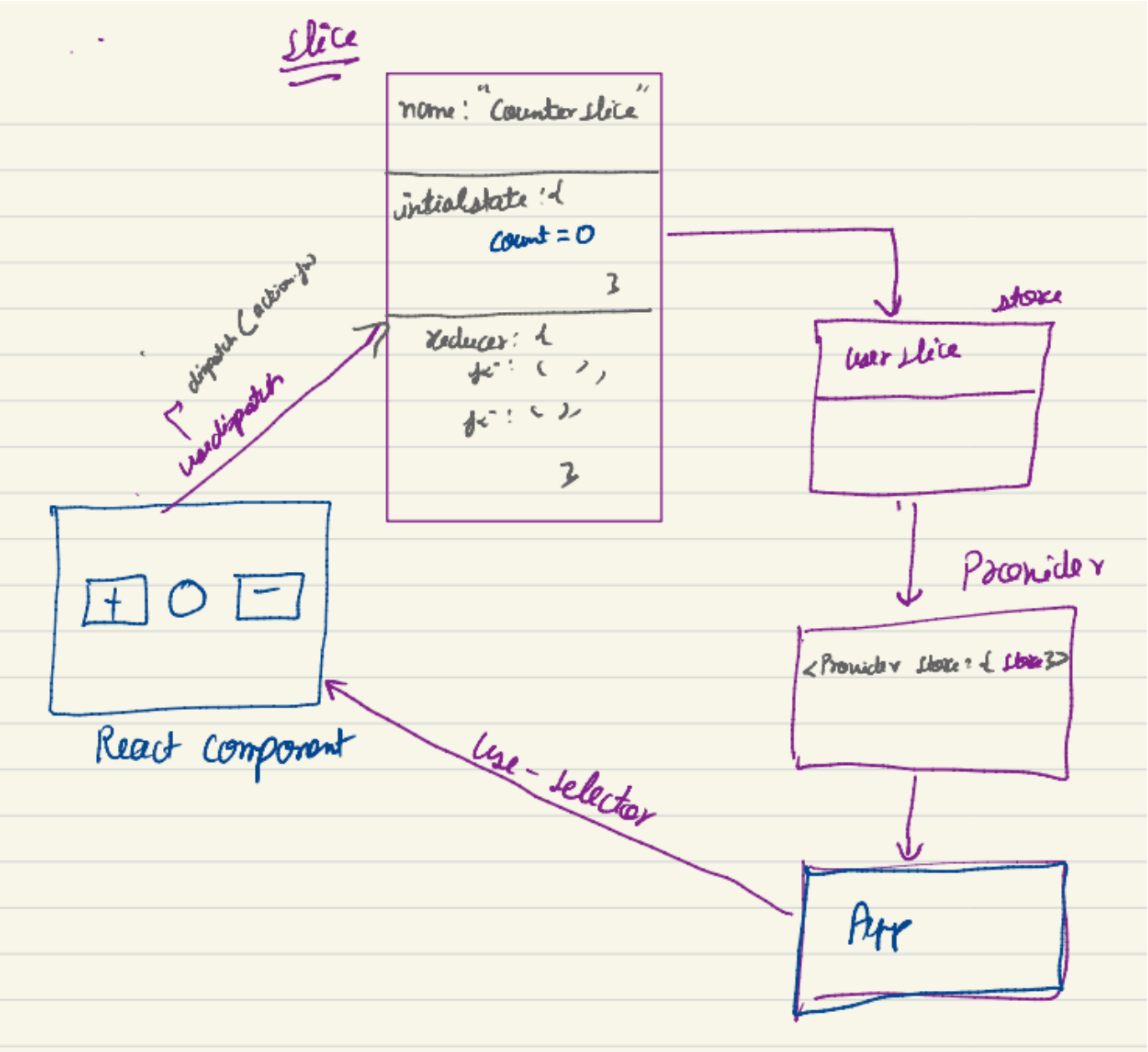
Provider is a component provided by the redux that wraps around the whole application.

It has an inbuilt prop known as a store in which we pass the name of the store we have created.



Firstly we are putting a slice into the store and then adding a store to the provider and now the next step is that the provider will wrap around the whole application.

When the react component wants to use, it will use `useSelector` method.



Now we will implement all this.

1. Create a file named `counterSlice.jsx` inside the `redux` folder.
2. Inside that file first import `createSlice`.

```
import { createSlice } from "@reduxjs/toolkit";
```

3. Now we will `createSlice` in `counterSlice` and it has name and initialState.

```
import { createSlice } from "@reduxjs/toolkit";


const counterSlice = createSlice({
    name:"counterSlice",
```

```
    initialState:{
        count:5
    }
});

export default counterSlice;
```

4. The next step is to create a store, so create `store.js` inside the `redux` folder.

There is only a single store and multiple slices.

5. Inside `store.js` write the below code, store will take the reducer of the slice, instead of directly taking the slice.

```
import { configureStore } from "@reduxjs/toolkit";

import counterSlice from './counterSlice';

const store = configureStore({
    reducer: {
        counterState: counterSlice.reducer
    }
});

export default store;
```

6. Inside `app.jsx`, import `Provider` and `store`,

```
import { Provider } from react-redux;
import store from "./redux/store.js"
```

7. Now just create a provider component and pass the store to it.

```
import React from 'react';
import { Provider } from react-redux;
import store from "./redux/store.js"


function App(){
    return {
        <Provider store = {store}>
            <Counter></Counter>
```

```
        </Provider>
    }
}

export default App;
```

8. Instead of doing it here, do it in `main.jsx`, to wrap the whole application using, inside `main.jsx` add the following imports and code

App.jsx

```
import React from 'react';
import CounterRedux from './components/reduxComponenets/CounterRedux';


function App(){
    return {
        <CounterRedux></CounterRedux>
    }
}

export default App;
```

main.jsx

```
import React from 'react';
import ReactDOM from 'react-dom/client'
import App from './App.jsx'
imprt './index.css'
import { Provider } from react-redux;
import store from "./redux/store.js"

ReactDOM.creatRoot(document.getElementById('root')).render
(
    <React.StrictMode>
        <Provider store = {store}>
            <App/>
        </Provider>
    </React.StrictMode>,
)
```

9. Now at the level of react component, we need to get the state, we use `useSelector` for this.
10. Open the file `CounterRedux.js`, and modify it:

```
import React from 'react';
import {useSelector} from "react-redux";
function Counter(){
    const handleIncrement = () => {
        console.log("increment will happen");
    }

    const handleDecrement = () => {
        console.log("decrement will happen");
    }
    return{
        <>
            <button onClick = {handleIncrement}> + </button>
            <h3>{count}</h3>
            <button onClick={handleDecrement}>-</button>
        </>
    }
}

export default Counter
```

11. We will pass a callback to the `useSelector`.

```
import React from 'react';
import {useSelector} from "react-redux";
function Counter(){
    const count = useSelector((store) => {return store.counterState.count});
    const handleIncrement = () => {
        console.log("increment will happen");
    }

    const handleDecrement = () => {
        console.log("decrement will happen");
    }
    return{
        <>
            <button onClick = {handleIncrement}> + </button>
            <h3>{count}</h3>
```

```
            <button onClick = {handleDecrement}> - </button>
        </>
    }
}

export default Counter;
```

Redux- Update the state

- All the things should be in slice, slice contains reducers object and here we will describe all the possible state changes.
- Here we have centralized state management, as we have an initial state in the slice and we are writing all the possible methods for state change inside the slice.

counterSlice.jsx

```
import { createSlice } from "@reduxjs/toolkit";


const counterSlice = createSlice({
    name:"counterSlice",
    initialState:{
        count:5
    },

    // all the update logic
    reducers: {

        //In that state we will get the initial state initially and that is
updated
        increment : (state) => {
            state.count += 1;
        }
        decrement : (state) => {
            state.count -= 1;
        }
    }
});

export default counterSlice;
```

Now inside `CounterRedux.jsx`, use this method for incrementing and decrementing, we will reducers of slice by the `actions`, `const actions = counterSlice.actions;`

```
import React from 'react';
import {useSelector} from "react-redux";
import counterSlice from "../../redux/counterSlice";

const actions = counterSlice.actions

function Counter(){
    const count = useSelector((store) => {return store.counterState.count});
    const handleIncrement = () => {
        console.log("increment will happen");
    }

    const handleDecrement = () => {
        console.log("decrement will happen");
    }
    return{
        <>
            <button onClick = {handleIncrement}> + </button>
            <h3>{count}</h3>
            <button onClick = {handleDecrement}> - </button>
        </>
    }
}

export default Counter;
```

For using increment and decrement methods, we will use the `useDispatch()` hook.

```
import React from 'react';
import {useDispatch, useSelector} from "react-redux";
import counterSlice from "../../redux/counterSlice";

const actions = counterSlice.actions

function Counter(){

    // get initial state
    const count = useSelector((store) => {return store.counterState.count});
```

```
    // is used to call any method from the reducer
    const dispatch = useDispatch();

    const handleIncrement = () => {
        console.log("increment will happen");

        // here it is used
        dispatch(actions.decrement());
    }

    const handleDecrement = () => {
        console.log("decrement will happen");
        dispatch(actions.increment());
    }
    return{
        <>
            <button onClick = {handleIncrement}> + </button>
            <h3>{count}</h3>
            <button onClick = {handleDecrement}> - </button>
        </>
    }
}

export default Counter;
```