

REDUX

Redux is a popular state management library for JavaScript applications, particularly those built with React. It helps manage the state of an application in a predictable and centralized manner, making it easier to understand, debug, and test.

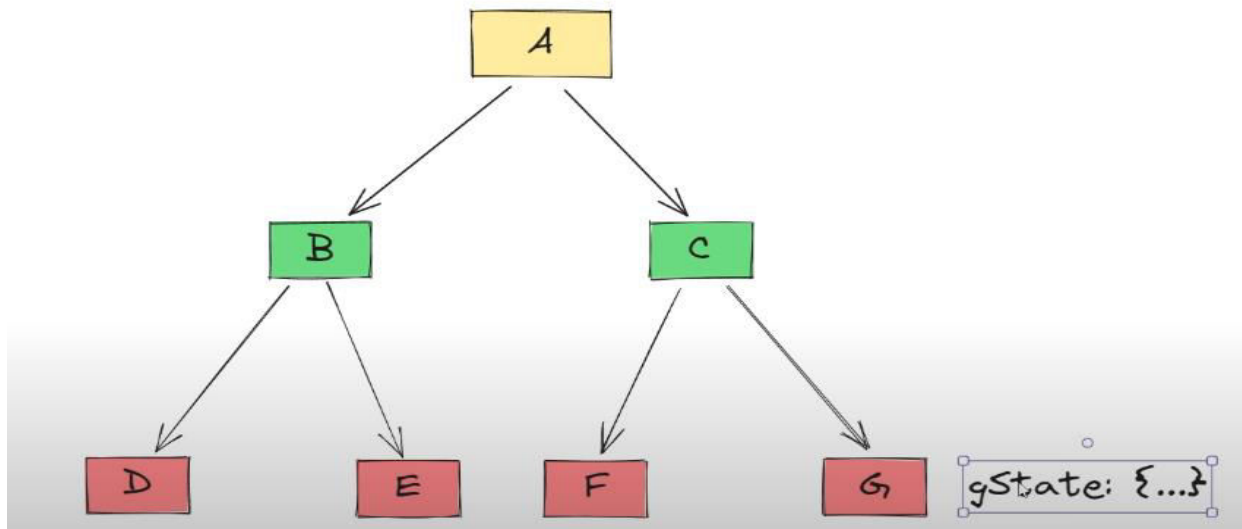
Key Concepts of Redux

1. **Store:** The single source of truth where the entire state of the application is stored. There is only one store in a Redux application.
2. **Actions:** Plain JavaScript objects that represent an intention to change the state. Actions typically contain a `type` property that indicates the type of action being performed, and they may also contain a `payload` with additional data.
3. **Reducers:** Pure functions that take the current state and an action as arguments and return a new state. Reducers specify how the state changes in response to an action.
4. **Dispatch:** A function used to send actions to the store. When an action is dispatched, the store's reducers process it and update the state accordingly.
5. **Selectors:** Functions that extract specific pieces of data from the store, making it easier to access and use parts of the state in components.

Why Redux is Used

1. **Predictable State Management:** Redux ensures that state changes are predictable by enforcing a strict unidirectional data flow. The state is only updated through pure functions (reducers) in response to actions, making it easier to track and manage changes. **Jab bhi state change hoga hamari application ko turant pata chal jaega.**
2. **Centralized State:** By storing the entire application's state in a single store, Redux makes it easier to manage and debug state. This centralization is particularly useful in large applications where state management can become complex.
3. **Debugging and Developer Tools:** Redux offers powerful developer tools, such as the Redux DevTools Extension, which allow developers to inspect every state change, action, and time travel through state history. This enhances debugging and makes it easier to understand how the state evolves over time.
4. **Scalability:** Redux's architecture scales well with larger applications. As the application grows, the centralized state and predictable state updates help maintain performance and manageability.
5. **Middleware:** Redux supports middleware, allowing developers to extend its capabilities. Middleware can be used for tasks such as logging, crash reporting, making asynchronous API calls, and more. Popular middleware includes `redux-thunk` for handling async logic and `redux-saga` for more complex side effects.
6. **Consistency Across Components:** Redux ensures that all components have consistent access to the state. This avoids issues where different parts of the application might have conflicting state or need to pass data through multiple layers of components.

Now lets see a case:-



A, B, C and more represent the components.

B and C are the children of A and similarly, D and E are of B, F and G are of C.

Lets say there is a state g which is inside the G component, but F needs the state g.

ab G se direct F to bhej nhi skte, to abhi to ek hi tarika h, F se C denge fir C se F, ye chota sa transfer krdia humne, aur app thik se chala.

Par ab maanlo ki F se E bhej na h g state ko to firse whi sb krna padega, F se pehle C me bhej ke store karaenge fir C se A me jaega, fir A se B me jaega, tb jake B se E me jaega.

ab g state ki jarurat E, F and G me hi thi pr

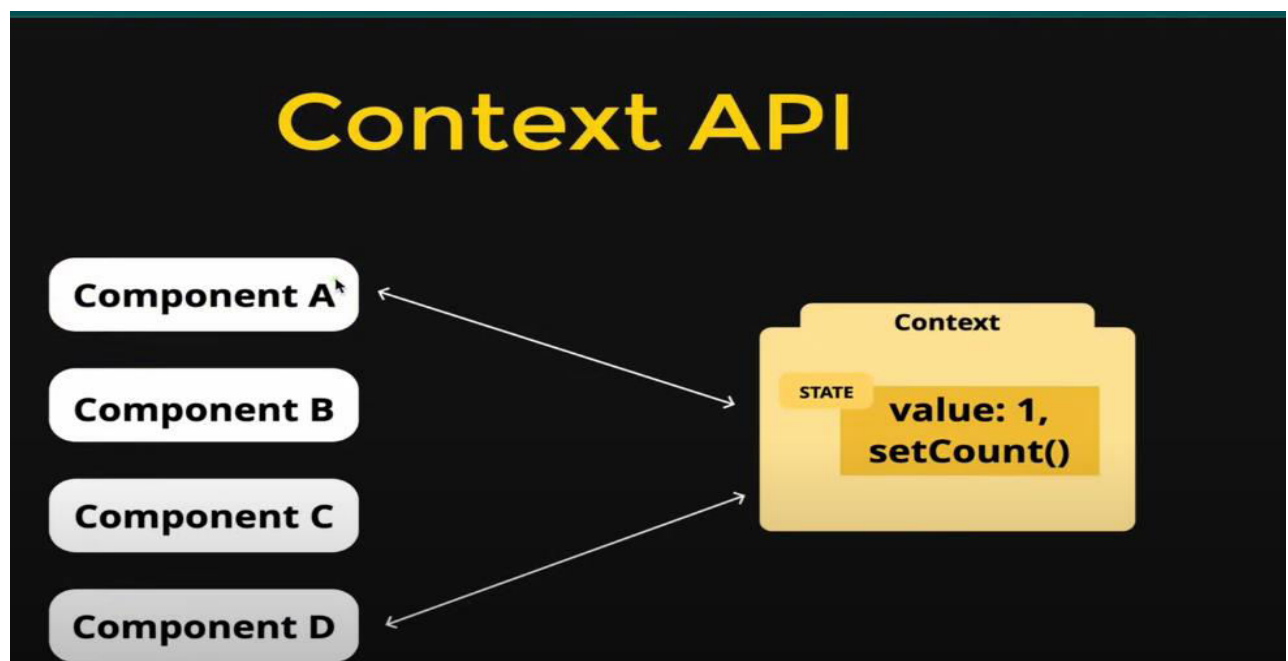
humne g state ko bhej A, B and C ke paas, aur in teeno ko g state ko apne paas rakhna pada.

Aur is phenomenon ko kehte h prop drilling.

To prop drilling ko avoid krne ke liye hum context API ka use krte h.

CONTEXT API:-

Jo pehle hum state ko component G se manga ke E me rakhe, ab kya karenge ki ek context type ka cheez h jo hamara state store krega, ab kisi bhi component ko wo state chahiye to direct use maang sakta h.



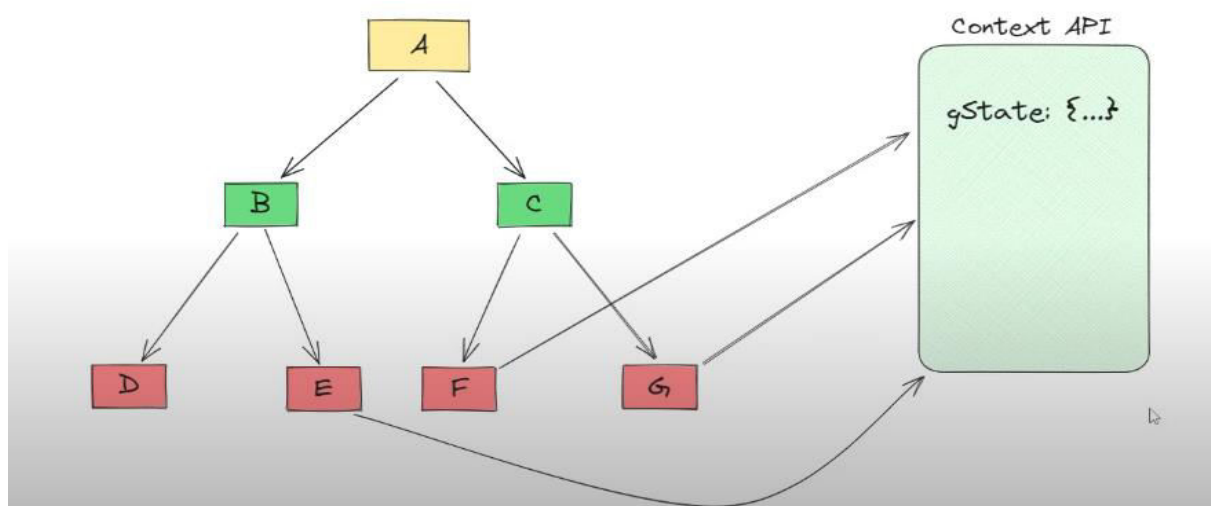
Agr component A ne state ko change kia to component B khud refresh/re-render hojaega

becoz component B bhi updated data ke sath rehna chahega. Jo bhi components context se attach h, to jb bhi state change hoga, to wo sabhi components khud ko re-render krlenge wo bhi updated states ke sath.

Aur isi wjh se state management bht easy hojata h.

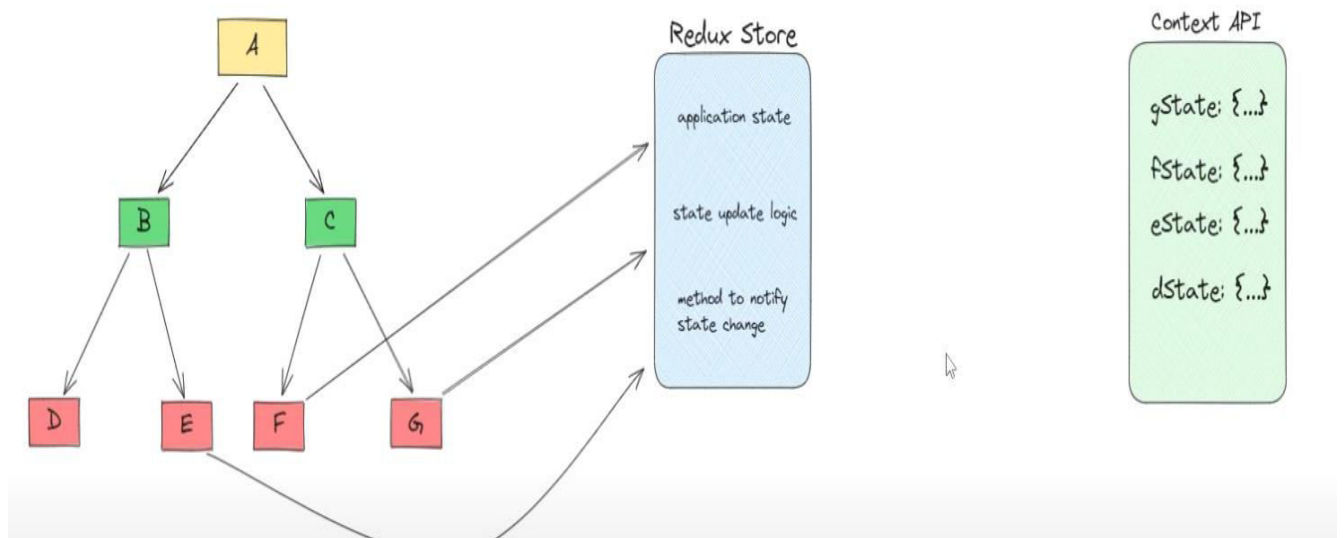
Agr hame apne components ko context ka access dena h to hame apne components ko context provider ke andar rap krna hota h.

To redux jo h ek tarah se context API use krta h.

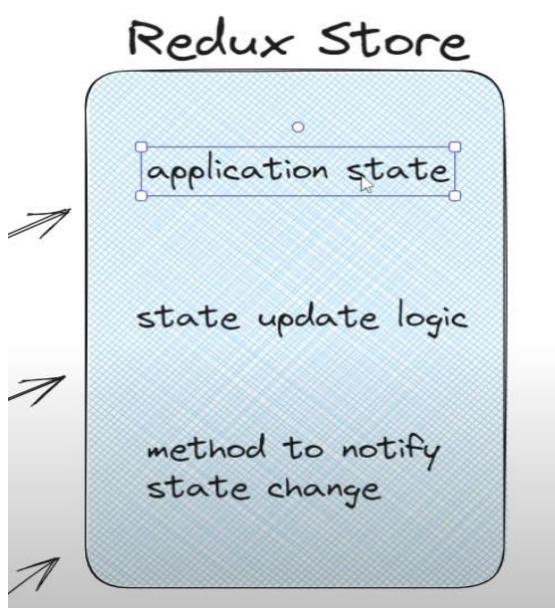


This is how redux uses context API to solve the problem of prop drilling.

Pr redux me component directly context API se connect nhi hota, components connect REDUX STORE se hota aur redux store behind the scene context API use krta h.



Now, the redux store has 3 main fundamental things:-

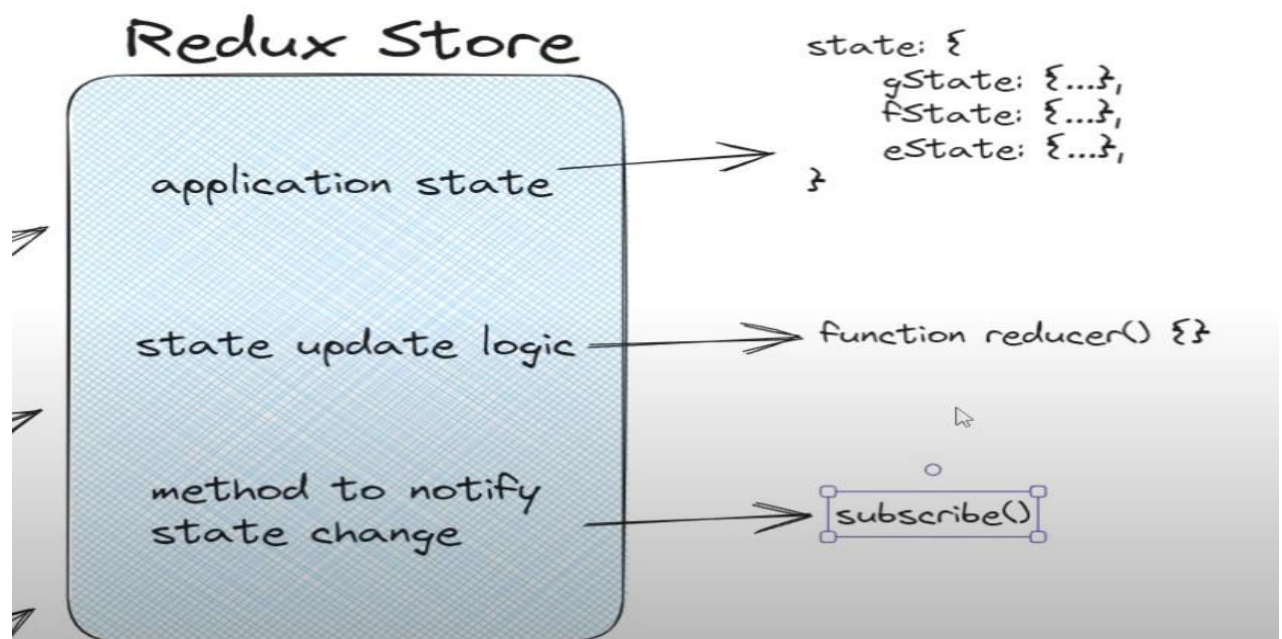


Application state me sabhi components ke state ka information rahega.

Application state me ek object hoga, jiske andar sabhi components ka state stored hoga.

State update logic me state ko update krne ka logic hota, aur state ko update krne ka kaam reducer function ka rehta h. ye reducer function shayad dikhega nhi pr rahega jarur.

Method to notify state change, ye achieve hota h subscribe method ke through.



To update the state, redux uses non-mutating state.

Non-mutating state:-

```

    ✓ let state = {
      count: 0,
      name: 'Anurag Singh',
      age: 26,
    }

    let prevState = state
    💡
    ✓ function increment() {
      //*** Mutating State ***//
      // state.count = state.count + 1

      //*** Not Mutating State ***//
      state = { ...state, count: state.count + 1 }
    }

```

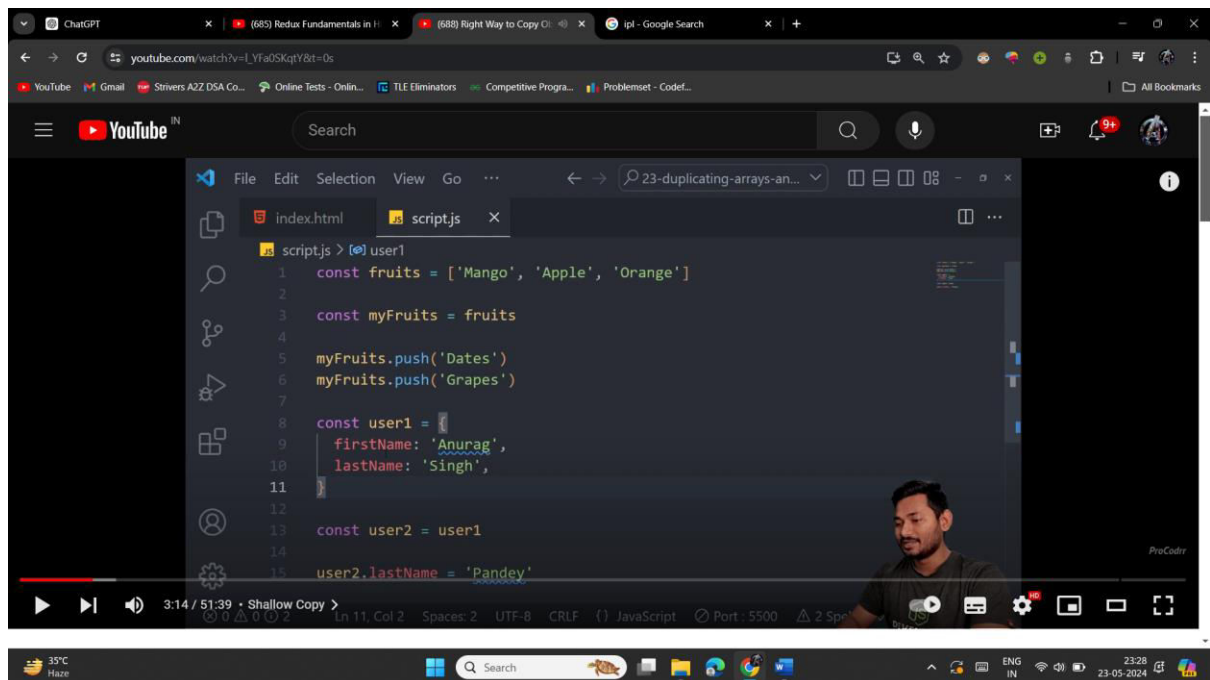
{...state lagake copy kia phir comma laga ke jo values change krna tha wo kia}

Ab increment function ke upper let a=state likdo, ab tum agr non mutating state use kroge, to a=count:0 dikhaega aur state=count:3, dikhaega leking agr tum mutating state use kroge to a=state=count:3 dikhaega, dono me same aaega.

Aur iss cheez ko hone ka reason h deep copy aur shallow copy ka concept.

Deep copy and shallow copy:-

Lets see this case:-



The screenshot shows a YouTube video player with a code editor overlay. The code editor displays the following JavaScript code:

```
1 const fruits = ['Mango', 'Apple', 'Orange']
2
3 const myFruits = fruits
4
5 myFruits.push('Dates')
6 myFruits.push('Grapes')
7
8 const user1 = {
9   firstName: 'Anurag',
10  lastName: 'Singh',
11 }
12
13 const user2 = user1
14
15 user2.lastName = 'Pandey'
```

The video player interface includes a search bar, a video title "23-duplicating-arrays-an...", and a progress bar showing 3:14 / 51:39. The video is titled "Shallow Copy".

If we print fruits and my fruits, then both will be printed as with both the new elements dates and grapes.

Similarly same thing will happen in the objects too, when user2 changes his surname, then automatically user1 surname will be changed.

In JS, when you do this, let a=5 , let b=a;

Here 5 is stored in a single address where a and b point to 5. That's why when we update (same types of updates as shown in the above example) something in b also reflects in a.

So if we do not want to interfere the values of 1st by updating or changing the values of 2nd then we do this:-

```
const myFruits = fruits

myFruits.push('Dates')
myFruits.push('Grapes')

const user1 = {
  firstName: 'Anurag',
  lastName: 'Singh',
}

const user2 = {}

Object.assign(user2, user1)
```

Object.assign(p1,p2), will assign the values of p2(old variable) to p1(new variable). Now

both user1 and user2 with same values would be stored in different addresses, so by changing the values of user2 does not make any changes to user1 values.

But object.assign() is the old way, so we will use the new way, the new way is the use of spread operator (...), these dots are know as spread operator.

```
const user1 = {
  firstName: 'Anurag',
  lastName: 'Singh',
}

// const user2 = {
//   firstName: 'Anurag',
//   lastName: 'Singh',
// }

// Object.assign(user2, user1)
const user2 = {...user1}
```

By doing const user2={...user1} is same as object.assign(user2,user1).

By doing ...user1, it copies the properties of user1 in user2.

The above two methods are known as shallow copy.

So that's the shallow copy.

Now back to redux,
redux uses the non-mutating method to copy the values. This is somewhat the same as a shallow copy.

Now we will see how exactly redux used non-mutating method:-

```
let reduxState = {  
  count: 0,  
  name: 'Anurag Singh',  
  age: 26,  
}  
  
function reducer(state) {  
  return { ...state, count: state.count + 1 }  
}  
  
// What Redux will Do //  
reduxState = reducer(reduxState)
```

Redux changes/updates the state by using the reducer function only. Here reducer function updates the state and returns the new state

without hampering the original state. In last `reduxState` is assigned with the updated state.

Par jruri nhi hi hr baar +1 hi kare hum -1 bhi krsktte the, to kab kya krna h wo batata h action, reducer function me ek aur parameter jata h, joki ek object rehta h aur uska naam, action hota h.

Lets take an example,

Maanlo ek bank(same as redux store) h, wo bank bht logon(component) ka account(state) khole hue h, pr agr kisi aadmi ko apne account se paisa nikalna h to wo direct to jaega nhi account locker ke pass aur jake paisa nikal lega, wo bank me ghuste hi jaega cashier(Reducer) ke pass aur bulega mera kaam karo(Action) to cashier puchegi kya karun? to wo bolega ki(Action ko elaborate krega) paisa withdraw karo , ab paisa withdraw krna to Action ka `{(type)}` hua, ye thik h pr cashier puchegi ki kitna paisa chahie, ab yhn pe dusri cheez jo wo bataega ki kitna

paisa nikalna h, to iss dusri cheez ko bolte h `{(payload)}`. To yhn pe type btaega ki state ke sath kya krna h, aur payload btaega ki state me kitna value dena ya lena h. waise payload hr baar kuch ho hi ye jaruri nhi h, ab maanlo bank me wo sirf paisa check krne gya to payload deke kya karoge.

NOTE:- Action object has two properties:- 1) Type 2) Payload. The type property is mandatory and the Payload property is optional.



Demo code:-

```
let reduxState = {
  post: 0,
  name: 'Anurag Singh',
  age: 26,
}

function reducer(state, action) {
  if (action.type === 'post/increment') {
    return { ...state, post: state.post + 1 }
  } else if (action.type === 'post/decrement') {
    return { ...state, post: state.post - 1 }
  }
}

// What Redux will Do //

reduxState = reducer(reduxState, { type: 'post/increment' })
console.log(reduxState)
reduxState = reducer(reduxState, { type: 'post/increment' })
reduxState = reducer(reduxState, { type: 'post/decrement' })
```

This is how the reducer function works.

Now we will practically see how redux works in react.

First, in your frontend install some libraries:-

Redux, react-redux, react-thunk.

Create a folder in src named as state and inside the state folder create two folders named as actions and reducers.

Now inside the action folder create an action.js file.

```
export const depositMoney = (amount) => {  
  return (dispatch) => {  
    dispatch({  
      type: 'deposit',  
      payload: amount  
    })  
  }  
}  
  
export const withdrawMoney = (amount) => {  
  return (dispatch) => {  
    dispatch({  
      type: 'withdraw',  
      payload: amount  
    })  
  }  
}
```

This is the syntax of declaring action, which takes the amount as a prop. It returns a function, the syntax of returning a function is similar to calling a function in the react component. It returns a function with the object inside it containing the information about the action.

Now inside the reducer folder create a file named as amountreducer.js, this reduces is only for amount there may be many reducers required but now we are using only one.

```
const reducer = (state=0, action)=>{  
  if(action.type==='deposit'){  
    return state + action.payload  
  }  
  else if(action.type==='withdraw'){  
    return state - action.payload  
  }  
  else{  
    return state;  
  }  
}  
  
export default reducer;
```

This is the syntax of the reducer function that takes two parameters, 1) initial state and 2) action.

Then performs the required function according to the action given.

Now generally we have more than one reducer. So we have to combine them all and send them to the store.

Now create file named as combine.js inside the reducer folder.

```
import { combineReducers } from "redux";
import amountReducer from "../amountReducer";

const reducers = combineReducers({
  amount: amountReducer
});

export default reducers
```

Here we have used combineReducers from redux which combines every reducer imported from different pages, and converts all the reducers into a single one.

Filhal abhi ek hi reducer h combined reducer ke andar.

Now create a file named store.js inside the state folder.

Abe hum apne pure app ko wo store dena chahte h, to hame redux ke provider ki zaroorat hogi, provider store ko pure app se connect krta h kuch is tarah:-

```
index.js
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
import { Provider } from 'react-redux';
import { store } from './state/store';

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

Jo store humne banaya tha wo store ab app se connect hogya h provider ki madad se.

Abhi tk sirf humne store ka access dia h pure app ko.

Now we will see how we access state, state is accessed by the use of the useSelector hook from react-redux.

Now say there is a page termed as navbar.js where you have to use the state.

```
import React from 'react'
import { useSelector } from 'react-redux';

const Navbar = () => {
  const amount = useSelector(state => state.amount)
  return (
    <div>
```

Here the amount variable is fetching the value of the amount that is stored in the store.

Now there is another component named a shop where, there are two buttons + and -, clicking on which increases or decreases the value of the amount. so we will see how without sending any information to the navbar.js page, the value of the amount is going to be changed.

We know if we want to update/change something on state then we have to use reducer.

But first create a file named as index.js in state folder, from there we going to send js file of action.

```
Navbar.js M App.js Shop.js 1, M index.js ...action-creators U index.js ...reducers U
src > state > index.js
1 export * as actionCreators from "../action-creators/index"
```

Here we exported, index.js (this file is different from state folder index.js) from the action-creators folder.

Now import this file in shop.js file where we have to use reducers.

```
import React from 'react'
import { useDispatch } from 'react-redux';
// import { bindActionCreators } from 'redux';
import { actionCreators } from '../state/index'

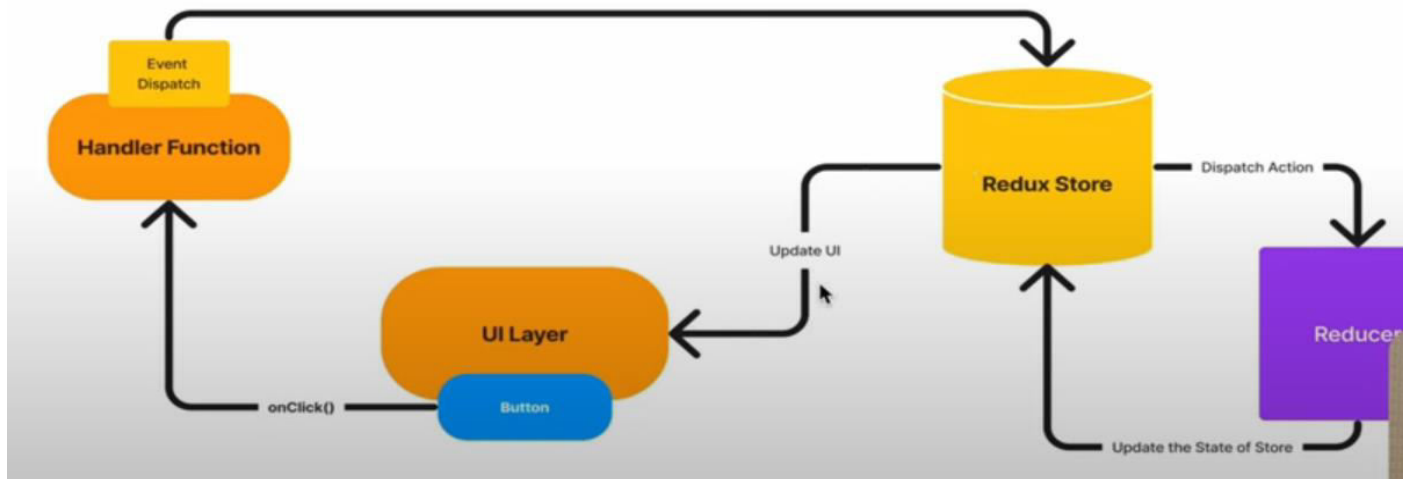
const Shop = () => {
  const dispatch = useDispatch();

  return (
    <div>
      <h2>Deposit/Withdraw Money</h2>
      <button className="btn btn-primary mx-2" onClick={() => { dispatch(actionCreators.withdrawMoney(100)) }}></button>
      Update Balance
      <button className="btn btn-primary mx-2" onClick={() => { dispatch(actionCreators.depositMoney(100)) }}></button>
    </div>
  )
}

export default Shop
```

Here we have used dispatch functions which will dispatch, the information (type and payload of action) to the reducer.

Redux Architecture




COMPLETE PATH WAY OF HOW REDUX IS USED IN REACT

1. Install Redux and React-Redux

First, ensure you have the necessary libraries installed:

sh


 Copy code

```
npm install redux react-redux
```

2. Create Action Types


Define your action types as constants to avoid typos and to make it easier to manage.

javascript

 Copy code

```
// src/redux/actionTypes.js
export const INCREMENT = 'INCREMENT';
export const DECREMENT = 'DECREMENT';
```

3. Create Actions

Actions are payloads of information that send data from your application to your Redux store. They are the only source of information for the store. 

javascript

```
// src/redux/actions.js
import { INCREMENT, DECREMENT } from './actionTypes';

export const increment = () => ({
  type: INCREMENT,
});

export const decrement = () => ({
  type: DECREMENT,
});
```

4. Create Reducer

Reducers specify how the application's state changes in response to actions sent to the store.

javascript

 Copy

```
// src/redux/reducer.js
import { INCREMENT, DECREMENT } from './actionTypes';

const initialState = {
  count: 0,
};

const counterReducer = (state = initialState, action) => {
  switch (action.type) {
    case INCREMENT:
      return {
        ...state,
        count: state.count + 1,
      };
    case DECREMENT:
      return {
```

```
        ...state,
        count: state.count - 1,
    };
    default:
        return state;
    }
};


export default counterReducer;
```



5. Create Store

The store holds the whole state tree of your application. The only way to change the state inside it is to dispatch an action on it.

javascript

 Copy code

```
// src/redux/store.js
import { createStore } from 'redux';
import counterReducer from './reducer';

const store = createStore(counterReducer);

export default store;
```

6. Integrate Redux with React

Use the `Provider` component from `react-redux` to pas

javascript

```
// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from '../redux/store';
import App from './App';

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



7. Create React Components

Use the `useSelector` hook to access state and the `useDispatch` hook to dispatch

javascript

```
// src/components/Counter.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../redux/actions';

const Counter = () => {
  const count = useSelector((state) => state.count);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
};
```



8. Use the Counter Component

Finally, include the `counter` component in your app.

javascript

```
// src/App.js
import React from 'react';
import Counter from '../components/Counter';

const App = () => {
  return (
    <div>
      <h1>Redux Counter Example</h1>
      <Counter />
    </div>
  );
};

export default App;
```



There were two questions:-

*Q1 how does selector fetch data?
// ans:- note:- you have rapped
store on the app, so without*

importing store, any component can directly access store.

// agr tum ye soch rhe ho ki ye selector kaise directly state ke count ko fetch krle rha h,
//kyuki humne to reducer se isko connect hi nhi kia, to bata dun ki ye reducer se nhi store se le rha h, aur store me tumne reducers dia hua h. aur jo store h, wo sabhi reducers ko collect krke ek state tree banata h, aur jab bhi tum state.{kuch bhi} karoge to wo jo (state) h wo koi partucular component ka state nhi h, wo collective jitne bhi reducers h store me, aur hr reducer ka apna apna kuch state hoga , to wo sabhi state combine hoke ek mega STATE banate, aur whi state baaki component use krte h, to jb humne state.count kia, to idhar sirf ek hi reducer tha jiska state ka

variable count that , wo store se directly access hogya.

Jab counter.js file me state.count kia to jo state tha wo mega STATE h jo ki sabhi states ko combine karke bana h , aur ab count usi mega STATE ka ek variable bn chuka h.

Q2 how does dispatch sends actions to desired reducers?

// In Redux, the dispatch function doesn't directly send actions to specific reducers. Instead, it sends actions to the store, and then the store handles distributing those actions to all relevant reducers.jb dispatch action ko dispatch krta h to wo store me jata h , ab store me to bht sare reducer jo skte h, to store us action ko sabhi reducer ko dedets hm agr wo sare stores us actions ke sath agr kuch krskte h

to krdenge nhi to aisa hi chor denge.

Detailed answer for Q1:-


Using Selectors in Components: When you use `useSelector` in a React component, it accesses the state provided by the store. The `state` parameter in `useSelector` corresponds to the entire state tree managed by the Redux store.

1. Here, `state.count` works because `state` refers to the entire state object managed by the store, and `count` is a property of that state object.

Example with Combined Reducers

If you have a more complex state with multiple slices managed by different reducers, you can use `combineReducers` to create a root reducer. Each slice will have its own part of the state tree.


javascript

 Copy code

```
// src/redux/rootReducer.js
import { combineReducers } from 'redux';
import counterReducer from './counterReducer';
import anotherReducer from './anotherReducer';

const rootReducer = combineReducers({
  counter: counterReducer,
  another: anotherReducer,
});

export default rootReducer;
```

Then, create the store using the `rootReducer` 

Then, create the store using the `rootReducer`:

javascript

```
// src/redux/store.js
import { createStore } from 'redux';
import rootReducer from './rootReducer';

const store = createStore(rootReducer);

export default store;
```

In this case, the state structure will be:

javascript

```
{
  counter: {
    count: 0,
  },
  another: {
    // other state properties
  },
}
```



You would use selectors like this:


javascript

```
// src/components/Counter.js
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from '../redux/actions';

const Counter = () => {
  const count = useSelector((state) => state.counter.count);
  const dispatch = useDispatch();

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
    </div>
  );
};

export default Counter;
```



Here, `state.counter.count` works because `state` now includes `counter` and another as top-level keys, each managed by their respective reducers.

Summary

Selectors know which part of the state to fetch data from based on the structure defined by your reducers and the state tree created by combining those reducers. The `useSelector` hook provides access to the entire state, and the selector function you pass to it determines which slice of the state you are interested in.