What is redux?

1.It is separate library which we import in our project and use it to manage state in global

2.it is not mandatory in react and redux is not only library

State management library

* Redux
* Zustand
* Etc

When we use redux it will be easy to debug the application with redux dev tools

There are 2 libraries

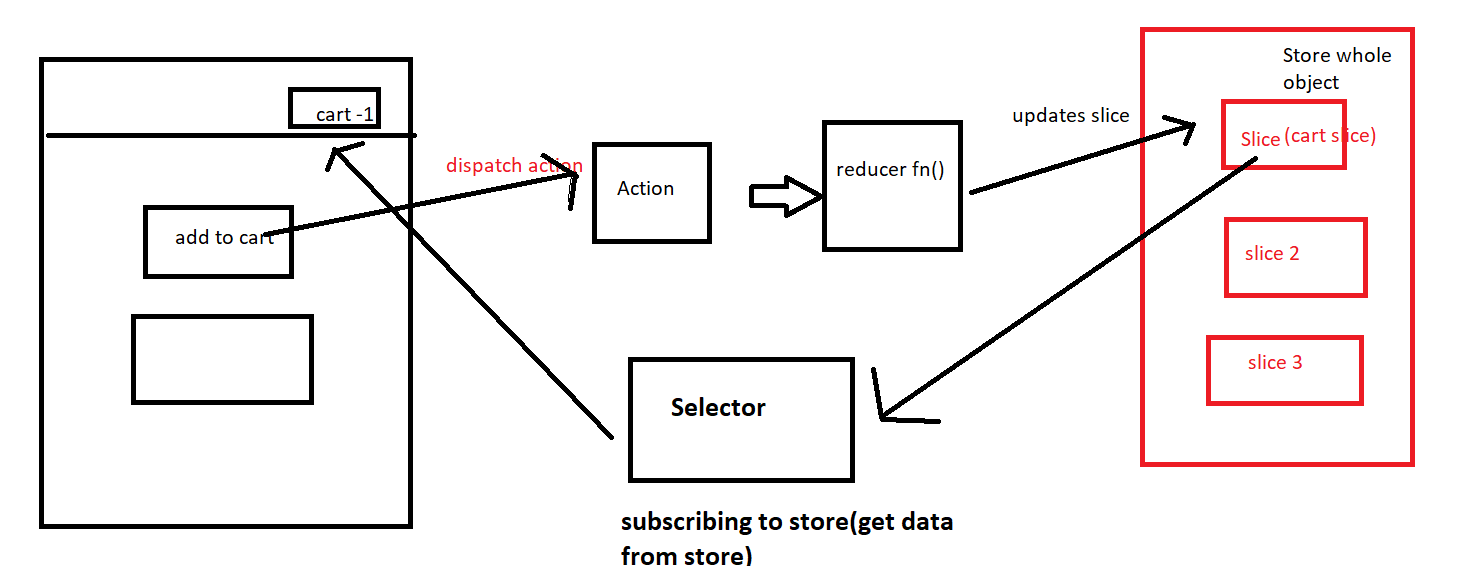
* React-redux
* Redux toolkit 🡪 new way of building redux and store(RTK)

Redux toolkit

* It is standard way of writing redux logic
* Major problem to solve old vanilla redux
  + To solve complex learning curve
  + To prevent many library to import and make redux work
  + New redux toolkit is very small and sweet

Redux store:

* It is very big object with data in it which can be accessed by any component in application
* We have part of store which have slices
* What is slices and why do we need?
  + To keep data separate we create slice to partition the logic of application
  + Create logical separation to keep data in store



When user click the add button 🡪it dispatch(action) 🡪calls reducer function🡪 which updates the store(slice of our store)

Our cart component is subscribed to store using selector

**onClick ={handleClick} onClick ={handleClick()} onClick ={()=>{handleClick()}} what is difference b/t above 3 in react with simple exmaple**

In React, the way you define the onClick event handler can have a subtle but important difference. Let's break down the three approaches you've mentioned and understand the difference.

**1. onClick={handleClick}**

* **What happens**: This is the most common approach. Here, handleClick is passed directly as the event handler function. React will automatically call handleClick when the button is clicked.
* **How it works**: React will call the function with the event as an argument.

**Example:**

import React from 'react';

const MyComponent = () => {

const handleClick = () => {

alert('Button clicked!');

};

return (

<button onClick={handleClick}>Click Me</button>

);

};

export default MyComponent;

* **Explanation**: When the button is clicked, React automatically calls handleClick().

**2. onClick={handleClick()}**

* **What happens**: This syntax **immediately invokes** handleClick when the component renders (not when the button is clicked). This is a **bad practice** in React because it causes handleClick to be executed immediately when the component renders, instead of when the button is clicked.
* **How it works**: Since you are calling the function **immediately** (by adding () after it), the result of handleClick() will be passed to onClick, which is typically **not what you want**.

**Example:**

js

CopyEdit

import React from 'react';

const MyComponent = () => {

const handleClick = () => {

alert('Button clicked!');

};

return (

<button onClick={handleClick()}>Click Me</button>

);

};

export default MyComponent;

* **Explanation**: This will immediately call handleClick when the component is rendered, and the alert will pop up as soon as the component mounts, rather than waiting for the button to be clicked.

**3. onClick={() => { handleClick() }}**

* **What happens**: This is a **function expression** passed as the event handler. It doesn't call handleClick immediately but rather defines a new anonymous function that, when executed (i.e., when the button is clicked), will call handleClick().
* **How it works**: This is the most flexible approach. You can add logic, pass parameters, or perform other operations inside this arrow function.

**Example:**

js

CopyEdit

import React from 'react';

const MyComponent = () => {

const handleClick = () => {

alert('Button clicked!');

};

return (

<button onClick={() => { handleClick() }}>Click Me</button>

);

};

export default MyComponent;

* **Explanation**: When the button is clicked, the arrow function is called first, and then it calls handleClick(). This ensures that handleClick is executed only when the button is clicked.

**Summary of Differences:**

1. **onClick={handleClick}**: Directly passes the function as the event handler. **Correct way**.
2. **onClick={handleClick()}**: Immediately invokes the function during render, which is usually incorrect and causes the function to run immediately. **Not recommended**.
3. **onClick={() => { handleClick() }}**: Defines an inline anonymous function that gets called when the event happens. **Flexible but can lead to performance issues** if used excessively, especially with heavy functions.

**Performance Consideration:**

* **onClick={handleClick}** is more **performance-efficient** than the inline function because it avoids creating a new function on every render.
* **onClick={() => { handleClick() }}** creates a new anonymous function on each render, which can lead to unnecessary re-renders in some cases, especially in performance-sensitive applications.

**Create store in redux:**

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

// create a store

const appStore = configureStore({});

export default appStore;

to provide the store to react we use provider from react-redux 🡪 it takes store as props from appStore

we can use provider to few components alone

**Create slice :**

Other options in RTK to create reducer and actions

* createReducer()
* createAction

is it possible to call async call like api calls in redux? 🡪Yes with **createAsyncThunk** function

**CreateAsyncThunk:**

* it takes 2 parameter as argument
  + **action type** eg:user/fetchData or any name that suites the api call
  + **async callback function(payload creator) 🡪 it has fetch async logic 🡪 always returns a promise**
  + **createAsyncThunk doesn’t create reducer function as it doesn’t know when the async operation will be completed**

**payload creator(async function 2nd parameter to createasyncthunk accepts 2 parameters**

* arg 🡪 it is argument passed to action when it is dispatched 🡪dispatch(fetchuser(**123**))
* thunkAPI 🡪object contains all options to be passed to redux thunk function
  + dispatch 🡪redux store dispatch method
  + getState->redux store gteState method
  + extra 🡪additional parameters
  + signal
  + rejectWithValue 🡪 unity function to throw error from async operation to handle error
* What createAsynThunk returns?
  + It returns action creator🡪pending,fulfilled,rejected
  + When dispatch action called
    - Calls callback function in payloadCreator(fetchuser) and wait for promise to settle
* when the promise settles:
  + if the promise resolved successfully, dispatch the fulfilled action with the promise value as action.payload
  + if the promise resolved with a rejectWithValue(value) return value, dispatch the rejected action with the value passed into action.payload and 'Rejected' as action.error.message
  + if the promise failed and was not handled with rejectWithValue, dispatch the rejected action with a serialized version of the error value as action.error
* Promise lifecyle action
  + Each createAsyncThunk return action creator(pending,fulfilled,rejected) to make it use in reducer logic and respond to action when dispatched
  + To handle these actions in reducer(reference from createSlice) uses builder notation

const reducer2 = createSlice({  
 name: 'users',  
 initialState,  
 reducers: {},  
 extraReducers: (builder) => {  
 builder.addCase(fetchUserById.fulfilled, (state, action) => {})  
 },  
})

Additionally, a settled matcher is attached, for matching against both fulfilled and rejected actions. Conceptually this is similar to a finally block.

Make sure you use addMatcher instead of addCase, since settled is a matcher rather than an action creator.

const reducer1 = createReducer(initialState, (builder) => {  
 builder.addMatcher(fetchUserById.settled, (state, action) => {})  
})

handle thunk return result

* The thunk from createAsyncThunk always returns promise which is either full filled or rejected
* To handle return we use following
* **The promise returned by the dispatched thunk has an unwrap property which can be called to extract the payload of a fulfilled action or to throw either the error or, if available, payload created by rejectWithValue from a rejected action:**

**// in the component**

**const onClick = () => {**

**dispatch(fetchUserById(userId))**

**.unwrap()**

**.then((originalPromiseResult) => {**

**// handle result here**

**})**

**.catch((rejectedValueOrSerializedError) => {**

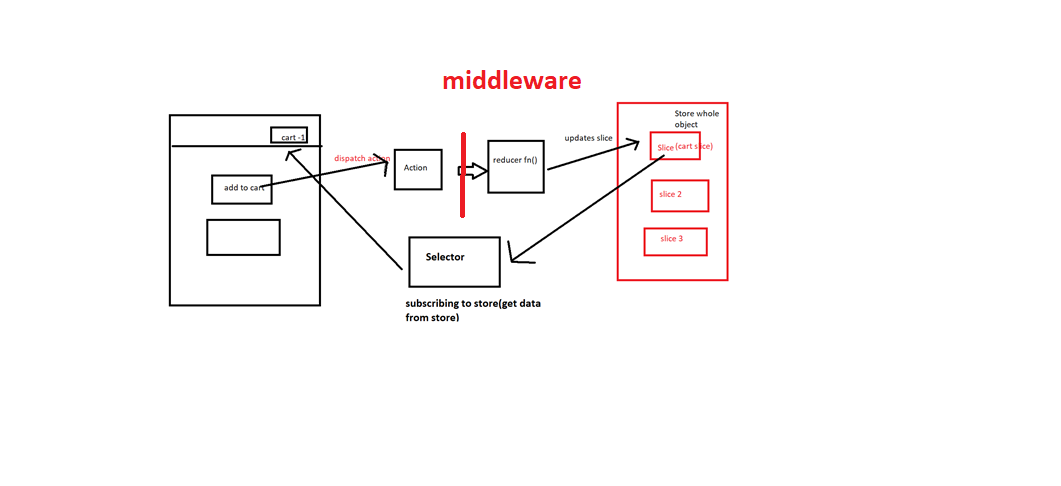
**// handle error here**

**})**

**}**

Middlware in redux

* Middleware act as pipeline b/t the action before it reaches reducer to perform some async operations like delay api call,log the data etc 🡪 **Middleware** in Redux is a way to extend Redux's capabilities. It's a function that sits between **dispatching an action** and the **moment it reaches the reducer**.



 Perform side effects (like API calls)

 Log actions

 Delay or cancel actions

 Modify actions

CreateAsynThunk 🡪by default it manages the middleware actions through **pending**, **fulfilled**, and **rejected**

### How Middleware Works with createAsyncThunk:

When you create an async thunk with createAsyncThunk, Redux Toolkit takes care of dispatching actions related to the async process (like **pending**, **fulfilled**, and **rejected**) and manages the flow through middleware. You don't need to manually dispatch these actions or deal with async logic directly.

Here's what happens behind the scenes:

1. **Thunk Function:**
   * The async function you pass to createAsyncThunk is treated as the "payload creator."
   * This function is dispatched automatically as a thunk function via the redux-thunk middleware (which is already included in Redux Toolkit).
2. **Action Types (pending, fulfilled, rejected):**
   * **Pending:** This action is dispatched when the async operation starts. It’s usually used to show a loading indicator.
   * **Fulfilled:** This action is dispatched when the async operation completes successfully. The result (like fetched data) is passed with this action.
   * **Rejected:** This action is dispatched when the async operation fails (e.g., if there’s an error in the API call).

**React tool kit query(RTK)**

* it is additional addons for react tool kit to perform caching and data fetching in app

Important notes in redux

1.we only need to subscribe to only state that is needed not whole state which is performance issue

  // we subscribe to the store through selector and get the data required

  // important note --> when subscribing to store subscribte to exact data needed from cart..

  //if we dont subscribe to correct data in state it will be big performance issue

  const cartItems = useSelector((state) => state.cart.items);

  // performance issue code -- bothe above code and below is same but we dont want to subscribe to whole store

  //  when there is change to some other data in store

  //  const data = useSelector((state)=>state);

  //  const items = data.cart.items;

2.in app store we have one big reducer for whole app but in slice there are many reducers function so we are use reducers in slice

const cartSlice = createSlice({

  name: "cart",

  initialState: {

    items: [],

  },

  // reducer is object that holds the action:and reducer function that hold logic to update the store

  reducers: {

    // addItem is action and it has reducer function

    // reducer ->2 parater ->old store data,action to it

    // action.payload is paremter we pass to the addItem action during dispatch

    addItem: (state, action) => {

      state.items.push(action.payload);

    },

    // payload holds index of item

    removeItem: (state, action) => {

      state.items.splice(action.payload, 1);

    },

    clearCart: (state) => {

      // this will make items array empty

      state.items.length = 0;

    },

  },

});

Important in reducer