**REDUX:**

We want to separate the view from the data(state). It is use for state management on client side.

Redux toolkit is build upon redux, it is not redux. Redux toolkit give us the guidelines. Redux is independent of pattern, we can do any pattern.

Redux create predictable state container. The state you update at one place, it will be updated everywhere where ever it is use. Redux creates store. And all states remain in store from where, we can access them and update them

Main installation:

yarn add redux react-redux

createStore() requires reducer, so we need to create reducer to pass in it

similar to context api, which provide us with provider so we wrap all of our components under provider so they all can access all the data in context. Redux also provides provider which we use to wrap index.js components and it requires store, like context provider requires value. So this provider give access to store data to all App and child components

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

import { Provider } from 'react-redux';

import { store } from './store/store';

ReactDOM.render(

  <React.StrictMode>

    <Provider store={store}>

    <App />

    </Provider>

  </React.StrictMode>,

  document.getElementById('root')

);

Reducer is a pure function

useDispatch():

Reducer works on messages instead of lots of useState. So to update state or data, we need to pass the messages. Dispatch function which uses useDispatch hook is used to pass the messages

Dispatch function takes object which takes message. And this message will be pass to reducer to take the action based on the type of message and update state.

We pass default initial state(which is a object) to state, so that we can update its state as per require action.

SPREAD OPERATOR:

Reducer function takes all the values from the existing state, and change or update them. For this, we use spread operator OR you can say it copy everything from which we applied spread operator. And after copying all the things, we can add new properties or update the previous properties

Var user = {name:”Muneeb”, email:”abc@gmail.com”}

Updating user;

Var newUser = {…user, name:”Zia Khan”, age:24}

So what we does is, we copy all the properties from the user, then update name, and add age property as well and store it in newUser.

Payload:

And to send data to reducer like to send some value to reducer so that it update the previous value based on the new value, we use payload property as well

dispatch ( {type:"INCREMENT\_BY\_AMOUNT", payload: Number(value)} )

So this time, it will take some value, and update counter based on that value, so we accept that value in payload. We can also pass object to payload like payload: {name:”Muneeb”, age:”20”}

And we access the information from payload using action.payload

        case "INCREMENT\_BY\_AMOUNT":{

            return {

                ...state, counter: state.counter+action.payload

            }

Complete reducer uptill now.

// Var user = {name:”Muneeb”, email:”abc@gmail.com”}

// Updating user;

// Var newUser = {…user, name:”Zia Khan”, age:24}

// So what we does is, we copy all the properties from the user, then update name, and add age property as well and store it in newUser.

const initialState={

    counter: 0

}

export const counterReducer = (state={initialState}, action)=>{

    switch(action.type){

        case "INCREMENT":{

            return {...state, counter: state.counter+1}

        }

        case "DECREMENT":{

            return {...state, counter: state.counter-1}

        }

        case "INCREMENT\_BY\_AMOUNT":{

            return {

                ...state, counter: state.counter+action.payload

            }

        }

    }

}

Now to update state everywhere in our app, we pass this reducer in createStore hook in store.js file

And remember, we have call that store in Provider tag in index.js, so all components of app will have access to store data and updation will be performed in all over the app.

useSelector():

Now we have updated the store, but we are not using counter from store till now. To access or use values from the store, we have a hook called useSelector() that takes function and in that function, we can access values from the state

As in this case, we have counter in the state, so to use counter from state in our app, we can access it like;

    //const [counter, setCounter] = useState(0);

    const [value, setValue]= useState(0);

    const dispatch = useDispatch();

    const counter = useSelector( (state)=>{

        return state.counter

    } );

Now we don’t need useState for the counter as we are using dispatch which pass message to reducer to update the state.

import { useState } from "react";

import { useDispatch, useSelector } from 'react-redux';

function Counter() {

    //const [counter, setCounter] = useState(0);

    const [value, setValue]= useState(0);

    const dispatch = useDispatch();

    const counter = useSelector( (state)=>{

        return state.counter

    } );

  return (

    <div>

        <div>Counter:{counter}</div>

        <div>

            <button onClick= {()=>{

                //setCounter(counter+1)

                dispatch ( {type:"INCREMENT"} )

            }}>Increament</button>

            <br/>

            <button onClick= {()=>{

                //setCounter(counter-1)

                dispatch ( {type:"DECREMENT"} )

            }}>decreament</button>

            <br/>

        </div>

        <br/>

        <div>

            <input type='text' onChange={(e)=>{

                setValue(e.target.value)

            }}/>

            <br/>

            <button onClick= {()=>{

                //setCounter(counter+Number(value))

                dispatch ( {type:"INCREMENT\_BY\_AMOUNT", payload: Number(value)} )

                }}>Increament by value</button>

        </div>

    </div>

  );

}

export default Counter;

Conclusion:

1. First set provider in index.js

    <Provider store={store}>

    <App />

    </Provider>

Their will be a single store, that work as a central part consisting of all the states using reducer.

1. Create directory store, and file store.js in it. We use createStore hook and pass reducer to it. It contains all the data to update, actions to performed on the state based on the messages coming from the dispatch.
   1. Rule#1: reducer function should be a pure function. Means it does not do any kind of side effects.
   2. Rule#2: default should be their which consist of state. So initially it does not give us the null.
2. When we are going to update the store, we use useDispatch function naming it as dispatch, and when we use value or state, we use useSelector();
3. We call dispatch and pass the object with message which we want to send to reducer, so that reducer take action based on that message. If we also want to send data with dispatch, we use payload and we can use it reducer with action.payload.
4. useSelector is used to update UI by getting data from the store.

Redux Toolit:

Intallation:

First create react app, in that run these commands.

yarn add react-redux

# NPM

npm install @reduxjs/toolkit

OR

# Yarn

yarn add @reduxjs/toolkit

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

export const counterSlice = createSlice( {

    name: "counter",

    initialState: {

        count:0

    },

    reducers: {

        increment: (state) =>{

            state.count = state.count+1

            //state.count += 1

        },

        decrement: (state) =>{

            state.count = state.count - 1

        },

        incrementByAmount: (state, action) => {

            state.count = state.count + action.payload

        }

}} )

export const {increment, decrement, incrementByAmount} = counterSlice.actions

export default counterSlice.reducer

Updated counter.js,

As count is a child of name counter, so we use it as state.counter.count in counter.js

import { useState } from "react";

import { useDispatch, useSelector } from 'react-redux';

import { decrement, increment, incrementByAmount } from "../store/action";

function Counter() {

    //const [counter, setCounter] = useState(0);

    const [value, setValue]= useState(0);

    const dispatch = useDispatch();

    const counter = useSelector( (state)=>{

        return state.counter.count

    } );

  return (

    <div>

        <div>Counter:{counter}</div>

        <div>

            <button onClick= {()=>{

                //setCounter(counter+1)

                //dispatch ( {type:"INCREMENT"} )

                dispatch(increment());

            }}>Increament</button>

            <br/>

            <button onClick= {()=>{

                //setCounter(counter-1)

                //dispatch ( {type:"DECREMENT"} )

                dispatch(decrement());

            }}>decreament</button>

            <br/>

        </div>

        <br/>

        <div>

            <input type='text' onChange={(e)=>{

                setValue(e.target.value)

            }}/>

            <br/>

            <button onClick= {()=>{

                //setCounter(counter+Number(value))

                //dispatch ( {type:"INCREMENT\_BY\_AMOUNT", payload: Number(value)} )

                dispatch(incrementByAmount(Number(value)))

                }}>Increament by value</button>

        </div>

    </div>

  );

}

export default Counter;

Updated store:

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

import counterReducer from './counterSlice'

export const store = configureStore({

    reducer: {

        //counter: counterSlice.reducer,

        //OR

        counter: counterReducer

    }

})

Where name of the reducer was counter, and we can use it as counterSlice.Reducer or give it any name and import it from counterSlice.js and we are exporting by default counterSlice.Reducer from counterSlice.js

THUNK:

As redux is used for state management. But we know sometimes we are getting data from some api or database which we use in our local data and before redux, we use useEffect to handle the async task. In Redux to add async data or api data or server or database data to maintain the state management, we need some redux implementation that can handle those async task. For that, thunk is used.

It uses createAsyncThunk for that. We know when are sending request to server for data, it takes time, so we need loader for that, then after that either we get the data or get the error. These all things can be handle by redux toolkit usinf createAsyncThunk().

createAsyncThunk() :

It takes two parameters or properties.

1 is the string that is the name or tag of the async thunk, 2 is the async function in which we use business logic and contacting to server by sending request and getting data from the server.

const fetchUserById = createAsyncThunk(

'users/fetchByIdStatus',

async (userId, thunkAPI) => {

const response = await userAPI.fetchById(userId)

return response.data

}

)

When we are using createAsyncThunk or handling async task, we use extraReducers in the createSlice(). As in case of server, first loading(pending), then either we get the data (Successful or fulfilled) or either we get some error (rejected), so all these things will be done in extraReducers object. So inshort, we have 3 properties as discussed above in redux toolkit for extraReducers section

* pending: 'users/requestStatus/pending'
* fulfilled: 'users/requestStatus/fulfilled'
* rejected: 'users/requestStatus/rejected'

// Then, handle actions in your reducers:

const usersSlice = createSlice({

name: 'users',

initialState: { entities: [], loading: 'idle' },

reducers: {

// standard reducer logic, with auto-generated action types per reducer

},

extraReducers: {

// Add reducers for additional action types here, and handle loading state as needed

[fetchUserById.fulfilled]: (state, action) => {

// Add user to the state array

state.entities.push(action.payload)

}

}

}

UPDATED REDUCER AND ACTIONS OR CREATESLICE():

counterSlice.js

import { createAsyncThunk, createSlice } from "@reduxjs/toolkit";

export const counterUpdate = createAsyncThunk(

    "counter/counterUpdate",

    async(value, thunkAPI) => {

        const response = await fetch("http://localhost:3000/api/updatecounter");

        const data = await response.json();

        return data;

    }

);

export const counterSlice = createSlice( {

    name: "counter",

    initialState: {

        count:0,

        isLoading: false,

        error: null

    },

    reducers: {

        increment: (state) =>{

            state.count = state.count+1

            //state.count += 1

        },

        decrement: (state) =>{

            state.count = state.count - 1

        },

        incrementByAmount: (state, action) => {

            state.count = state.count + action.payload

        }

    },

    extraReducers: {

        [counterUpdate.fulfilled]: (state, action) => {

            state.count = state.count + action.payload

            state.isLoading = false;

        },

        [counterUpdate.pending]: (state) => {

            state.isLoading = true;

        },

        [counterUpdate.rejected]: (state) => {

            state.isLoading = false

            state.error = "Error"

        }

    }

} )

export const {increment, decrement, incrementByAmount} = counterSlice.actions

export default counterSlice.reducer

createAsyncThunk:

export const counterUpdate = createAsyncThunk(

    "counter/counterUpdate",

    async(value, thunkAPI) => {

        const response = await fetch("http://localhost:3000/api/updatecounter");

        const data = await response.json();

        return data;

    }

);

Fetching api and then json in data and then return json data

Now to handle async thunk, we use extra reducer

   extraReducers: {

        [counterUpdate.fulfilled]: (state, action) => {

            state.count = state.count + action.payload

            state.isLoading = false;

        },

        [counterUpdate.pending]: (state) => {

            state.isLoading = true;

        },

        [counterUpdate.rejected]: (state) => {

            state.isLoading = false

            state.error = "Error"

        }

    }

} )

AS we know server takes time as discussed pending then either fulfiled or rejected, so for that, we create other two properties in intialState, isLoading and error. When pending, means page is loading so we set isLoading to true otherwise false everywhere. When rejected, we pass some value to error to present when we get some error and rest is same, and we use same method to update the state as can be seen in counterUpdate.fulfilled which takes arrow function along with state and actions and we update state.count

Button for server:

        <button onClick= {()=>{

                dispatch(counterUpdate(15))

                }}>Increament from Server</button>

        </div>

    </div>

Create mirage server(in server.js):

import { createServer } from "miragejs"

export default function () {

    createServer({

        routes() {

            this.namespace="/api"

            this.get("/updatecounter", () => {

                return 34;

            })//(api, data)//for getting or fetching data

          },

        }

      )

    }

Calling mirage server in index.js

import {mirageServer} from './server';

mirageServer();

ReactDOM.render(

Add loading timing in mirage server:

export const counterUpdate = createAsyncThunk(

    "counter/counterUpdate",

    async(value, thunkAPI) => {

        const response = await fetch("http://localhost:3000/api/updatecounter",

        // {

        //     method: "post",

        //     body: {name: "hello"}//way to post data to server and can be post using this.post in mirage

        // }

        );

        const data = await response.json();

        return data;

    }

);