Redux is a pattern and library for managing and updating application state using events called actions.

It takes the state variables in a react app outside of the components and into a centralized area (store) for all components to access. This eliminates the hassle of moving state variables up to a parent component if multiple child components need to access them.

Redux is mainly used for larger codebases

There are 3 parts of managing state:

The state – source of truth for the app.

The view – usually html code which displays the state somehow

The actions – events which immutably change the state somehow

Data flows from the state, to the view to the actions and back to the state

You must change state immutably since react compares references of objects and arrays to check if any changes have been made. Mutably changing objects and arrays does NOT change the reference so no change is observed.

Actions:

An action is a plain JS object which has a ‘type’ field and can include an additional field containing more information which is normally termed the ‘payload’ field.

The type field should be a string that gives the action a descriptive name.

const addTodoAction = {

type: 'todos/todoAdded',

payload: 'Buy milk'

}

Action creators:

A function which creates and returns an action object to cut down on repeated code.

const addTodo = text => {

return {

type: 'todos/todoAdded',

payload: text

}

}

Reducers:

(state, action) => newState

A reducer is a function which takes the current state and an action object and determines how to change state depending on action.type. Reducers are like event listeners which handles events based on the event type (actions). You should include an initial state in your reducer function.

Reducers must always:

Calculate the new state based on the state and action arguments only.

Immutably change the existing state by creating a copy and changing the copy

Not include any asynchronous logic, calculate random values or other “side effects”

const initialState = { value: 0 }

function counterReducer(state = initialState, action) {

// Check to see if the reducer cares about this action

if (action.type === 'counter/increment') {

// If so, make a copy of `state`

return {

...state,

// and update the copy with the new value

value: state.value + 1

}

}

// otherwise return the existing state unchanged

return state

} //you can use any kind of logic (if/else, switch, loops)

Redux toolkit uses immer which allows you to change state mutably since it creates a copy behind the scenes.

When creating a store, you should include all your reducers in the reducer key.

If you have multiple reducers, you can group them into a root reducer using RTK’s combineReducers()

const rootReducer = combineReducers({

users: usersReducer,

posts: postsReducer,

comments: commentsReducer

})

const store = configureStore({

reducer: rootReducer

})

RTK simplifies the dispatch process. Instead of dispatching an action object, you can dispatch a method call which automatically generates an action object based on the name key and reducer name of your reducer slice.

createSlice takes an object with the following keys:

name: <name of state variable>

initialState: <initial value of state variable>

reducers: <list of reducer functions>

example

const counterSlice = createSlice({

name: 'counter',

initialState: 0,

reducers: {

increment: state => state + 1,

decrement: state => state - 1

}

})

createSlice returns a “slice” object that contains the generated reducer functions in a field called reducer, and the generated action creators in a field called actions (you can access action type strings from the action key as well)

const { actions, reducer } = counterSlice

const { increment, decrement } = actions