Reducers are a pure function in Redux. Pure functions are predictable. Reducers are the only way to change states in Redux. It is the only place where you can write logic and calculations. Reducer function will accept the previous state of app and action being dispatched, calculate the next state and returns the new object.

The following few things should never be performed inside the reducer −

* Mutation of functions arguments
* API calls & routing logic
* Calling non-pure function e.g. Math.random()

The following is the syntax of a reducer −

(state,action) => newState

Let us continue the example of showing the list of product items on a web page, discussed in the action creators module. Let us see below how to write its reducer.

const initialState = {

isLoading: false,

items: []

};

const reducer = (state = initialState, action) => {

switch (action.type) {

case 'ITEMS\_REQUEST':

return Object.assign({}, state, {

isLoading: action.payload.isLoading

})

case ‘ITEMS\_REQUEST\_SUCCESS':

return Object.assign({}, state, {

items: state.items.concat(action.items),

isLoading: action.isLoading

})

default:

return state;

}

}

export default reducer;

Firstly, if you do not set state to ‘initialState’, Redux calls reducer with the undefined state. In this code example, concat() function of JavaScript is used in ‘ITEMS\_REQUEST\_SUCCESS', which does not change the existing array; instead returns a new array.

In this way, you can avoid mutation of the state. Never write directly to the state. In 'ITEMS\_REQUEST', we have to set the state value from the action received.

It is already discussed that we can write our logic in reducer and can split it on the logical data basis. Let us see how we can split reducers and combine them together as root reducer when dealing with a large application.

Suppose, we want to design a web page where a user can access product order status and see wishlist information. We can separate the logic in different reducers files, and make them work independently. Let us assume that GET\_ORDER\_STATUS action is dispatched to get the status of order corresponding to some order id and user id.

/reducer/orderStatusReducer.js

import { GET\_ORDER\_STATUS } from ‘../constants/appConstant’;

export default function (state = {} , action) {

switch(action.type) {

case GET\_ORDER\_STATUS:

return { ...state, orderStatusData: action.payload.orderStatus };

default:

return state;

}

}

Similarly, assume GET\_WISHLIST\_ITEMS action is dispatched to get the user’s wishlist information respective of a user.

/reducer/getWishlistDataReducer.js

import { GET\_WISHLIST\_ITEMS } from ‘../constants/appConstant’;

export default function (state = {}, action) {

switch(action.type) {

case GET\_WISHLIST\_ITEMS:

return { ...state, wishlistData: action.payload.wishlistData };

default:

return state;

}

}

Now, we can combine both reducers by using Redux combineReducers utility. The combineReducers generate a function which returns an object whose values are different reducer functions. You can import all the reducers in index reducer file and combine them together as an object with their respective names.

/reducer/index.js

import { combineReducers } from ‘redux’;

import OrderStatusReducer from ‘./orderStatusReducer’;

import GetWishlistDataReducer from ‘./getWishlistDataReducer’;

const rootReducer = combineReducers ({

orderStatusReducer: OrderStatusReducer,

getWishlistDataReducer: GetWishlistDataReducer

});

export default rootReducer;

Now, you can pass this rootReducer to the createStore method as follows −

const store = createStore(rootReducer);