Полезно тем кто уже знает редакс, посмотреть на примере редакс приложения ртк как аналогию

Полезно тем кто только знакомится с ртк как подходом, будет разобран явный пример различного функционала

Отельные части сегментов показывают отдельные подходы и примеры разработки на ртк

Команда разработчиков ртк пыталась решить 3 основные проблемы:

Конфигурировать сторе проще, чтобы редакс заставить работать нужно кучу доп пакетов, редакс требует кучу бойлерплейт кода

И еще не стоит забывать, используя ртк мы получаем больше шаблонных правил от проекта к проекту а значит становится проще понимать код

Посидев над этими проблемами, и опросив пользователей – они сделали ртк, по своей сути это все таки остается оберткой над редакс и все фундаментальные концепции были впринципе сохранены

Наша с вами задача посмотреть на их решения ну и попробовать сделать какие-то выводы по этому поводу

На основе созданного приложения на старых принципах редакса, без усложнений, мы разработаем аналог на ртк, попутно рассматривая апи которое они нам предоставляют

Здесь на примере проекта мы с вами постараемся разобрать скорее частные случаи использования, чем создать полноценное приложение и поэтому этот проект не логичен и местами может показаться странным )

Введение в созданные проект, но сегментами

Рассказать что вот собрал уже второй проект подключил те же компоненты

Главное что Provider закоментировал

И так давайте приступим, первым делом соберем наш стор

* Создаем файл – store.js

В ртк стор создается при помощи новой функции configureStore [kənˈfɪgə] – это обертка над createStore c набором готовых решений.

1. Первым сделаем хоть один редьюсер, сейчас накидаем для верхней панели его –

Создаем файл topReducer.js в нем const topReducer = (state = {}, action) => {return state};

export default topReducer;

import topReducer from ‘./topReducer’

Const reducers = {topReducer} – при передаче объекта конфиг сам соберет наши редьюсеры по принципу combineReducer функции

Const store = configureStore({ reducer: reducers})

1. Middleware – здесь нам рекомендуют взять getDefaultMiddleWare, хотя можно и по другому

Const middleWares = getDefaultMiddleWare();

Внутри дефолта находится 2 сегмента – это для разработки и для продакшена.

Для разработки они поставляют дополнительное промежуточное ПО, такое как глубокая проверка стейта на мутацию,при обнаружении мутации он укажет в ошибках это и проверка сериализуемости, глубоко проверяет ваше дерево состояний и ваши действия на наличие несереализуемых значений, тоже оповестит в ошибках

И добавляет редакс thunk по умолчанию

Для сегмента продакшена - он оставляет только redux-thunk

Так же остаются подходы для настройки мидлваре, мы возьмем дефолт и закинем еще свой туда

НАПИШем свой быстренько как простой пример и вошьем его к дефолту заодно как просто пример, без фанатизма ЗАААЧЕМ ?? Позволяет подключаться когда мы отправляем действия

const customMiddleWare = store => next => action => {

    console.log(store.getState())

    return next(action);

}

const defaultMiddleWare = (getDefaultMiddleware) => getDefaultMiddleware().concat(customMiddleWare);

const store = configureStore({

    reducer: reducers,

    middleware: defaultMiddleWare

});

1. Разбираем дальше наше создание стора – подключаем ДевТулз, кстати они в девтулз добавили трассировку действий и различные настройки для composeWithDevtools()

Если По дефолту не указывать он будет уже находится там с ТРАССИРОВКОЙ ДЕЙСТВИЙ так и оставим???

1. preloadedState – начальное состояние для криэйтСтора
2. enchancers – функции которые создают новый стор из одного или нескольких создателей стора

кстати applyMiddleware функция это на самом деле енчансер и redux devtools(позволяет перемещаться во времени) ЗАЧЕМ??? – это может изменить то как редукторы обрабатывают данные или как работает диспатч запуск каждый раз? Есть пример в отдельной доки ПОКАЗЫАТЬ ??

Можно разобрать 2 мидлавар – это мутацию и сериализацю путем прокидывания ошибок

Так, мы закончили с созданием стора и обсудили его новую функциональность, теперь давайте начнем собирать верхнюю панель со счетчиком и заодно познакомимся со следующим функционалом ртк

Вернемся в топРедььюсер, - раньше мы писали редьюсер через свитчи на каждое условие действия – (показать пример в топ редьюсере другого проекта),много шаблонного кода и можно забыть регистр в типах или управление начальным состоянием

МОЖНО ЛИ createReducer вмонтировать в createSlice ?

Теперь нам предоставляют createReducer – он предоставляет две различные формы определения редьюсеров регистра для обработки действий: нотация builder callback и map object (предпочтительней callback builder – он хорошо работает с ts)

Начнем с BuilderCallback reducer и напишем на нем инкремент и декремент

Import {createReducer} from ‘@redux

Const initialState = { counter: 0, action: ‘’, disabled: false }

Const topReducer = createReducer ( initialState , и (builder) => {

Builder

.addCase(increment, (state, action) => ({…state, counter: state.counter + 1, action: 'Увеличили'})

} )

builderCallback – Написать пример в комментах - ???Builder.addCase(actionCreator or Type, reducer)

Принимает функцию обратного вызова , которая получает объект builder в качестве аргумента. Этот конструктор предоставляет функции addCase, addMatcher и addDefaultCase, которые предоставляются для определения действий, которые будет обрабатывать текущей редьюсер

Сразу же напишем упрощенный action (сразу рассказать про возможности или потом????)

Export Const increment = createAction(‘topReducer/increment’)

Чтобы не распылятся сразу же пойдемте в компоненту и все подключим а заодно проверим наш новый редьюсер с экшеном. *(Открыть PanelTop.jsx*) для начала подберем хуки для диспатча и селекторов (надо ли рассказывать про селекторы????) import {useDispatch, useSelector} from 'react-redux'; В первом проекте мы использовали connect, но в нынешних реалих я не вижу смысл его использовать, значит напишешь селектор выборки

const getTopReducer = (state) => state.topReducer; пишем фукнцию получения топредьюсера

const {counter} = useSelector(getTopReducer); (в компоненте) получаем топредьюсер через юзСелектор

const dispatch = useDispatch(); подключаем dispatch

пробрасываем в компоненту в первый div{counter} и onClick={() => dispatch(increment())} первой кнопке

*(показываем функционал)* Вот в принципе и все, когда мы допишем весь функционал мы посмотрим на кол-во кода в редаксе и в ртк

Кстати давайте обратим внимание на подход коннект и использование хуков, лично для меня намного удобнее использование хуков, особенно за счет того что я явно вижу сразу что участок кода относится к редаксу, а какой я тяну из пропсов, а не все подряд из пропсов, но впринципе это наверное дело привычки всетаки, но кода меньше

Еще одна удобная особенность использования useSelector так это --- ??? shalowEqual

Теперь когда инкремент работает давайте вернемся в код и посмотрим что еще за методы у нас остались в createReducer, ну а потом допишем остальное

Вообще наш addCase добавляет один конкретный обработчик действия

А еще один метод addMatcher через него проходят все ваши экшены, то есть через него вы можете делать и запускать реакции редьюсера на общие дейсвтия или группы действий к примеру

Ну к примеру накидаем что то снова в консоль .addMatcher((action) => action, (state, action) => console.log('action type -', action.type)) – помоему удобненько, кстати их можно много наплодить

Ну и последний метод addDefaultCase ((state, action) => console.log(state, action)) - он запустится только при условии что экшен до этого не обработался даже экшенматчером

И добавим на вторую кнопку у нас простой диспатч экшена мимо onClick={() => dispatch({type: 'SSS'})}

– случай по умолчанию, дернется когда не было никаких совпадений в экшенах, кстати не разу не смог включить его при наличии функции .addMatcher

Дальше второй экшен редьюсер с мутацией иммера А куда еще запихнуть второй подход? Мб в слайсе использовать?

Продолжим создание приложения

Создаем новый экшен export const decrement = createAction('topReducer/decrement')

.addCase(decrement, (state, action) => {state.counter = state.counter – 1; state.action = 'Уменьшили'})

Сейчас наверное фронты такие смотрят и думают о боже мой, что ты творишь человек, остановись - все хорошо коллеги все схваченно, сейчас все расскажу

В PanelTop import decrement и переделаем диспатч на decrement - нажимаем кнопку и вуаля все работает и никто не ругается!

Все дело в том что мы знаем что редьюсер должен быть чистой функцией, но иногда эта такое обильное кол-во кода в итоге, ртк нам по дефолту подарил иммер внутри себя и теперь он сам автоматически создает новое состояние – Кто не знает что такое иммер – это крошечный пакет, который создает за вас следующее состояние на основе ваших мутация. Но надо не забывать что у него есть и свои особенности, к примеру нам нельзя в редьюсере одновременно мутировать и возвращать новое состояние

Еще одна интересная новая особенность у нас появилась для отладки через консоль. Не всегда для отладки нам хорошо помогает console.log особенно в случае с иммером, который покажет вам конечное действие без промежуточных работ самого иммера(ведь у него свой порядок обработки), так вот нам предоставили функцию current

 .addCase(decrement, (state, action) => {

            console.log('before',current(state));

            state.counter = state.counter – 1;

state.action = ‘Уменьшили’

            console.log('after', current(state))

        })

Наверное надо расшириться под тех кто вообще не знает редакс? СВЯЗАТЬ ПОТОМ ССЫЛКИ С РАЗДЕЛАМИ

АСИНК ДЕЙСТВИЕ И createAsyncThunk

Теперь напишем последнее действие связанное с асинхронным кодом, но пока мы не будем использовать реальные запросы - ДЕМОНСТРАЦИЯ ДЕЙСТВИЯ НА ОСНОВЕ ПРЕДЫДУЩЕГО – В вебе показать – в коде

Первым делом нам следует разобрать функцию ртк для асинхронных действий createAsyncThunk – это функция которая принимает строку экшен тайпа редакс и callback функцию, которая должна возвращать promise. Он генерирует экшен тайп жизненного цикла промис на основе префикса типа действия (пендинг, фалфилд и реджект) который вы передаете и возвращает создателя действия thunk, который вывполнит промис коллбэк и отправит экшены жизненного цикла на основе возращенного промиса.

Он не генерирует никаких редьюсеров, он как бы генерит экшены, а редьюсеры мы должны написать сами, которые нам нужны

Не беспокойтесь, сейчас мы это сделаем в коде и будет все понятно

createAsyncThunk принимает 3 аргумента - стрингу экшен тайпа, payloadCreator callback, и опционально объект options

В ТОПРЕДЬЮСЕР - export const asyncIncrementAction = createAsyncThunk(

    'topReducer/asyncIncrementAction',

)

Из строки экшен тайпа будет автоматически сгенерированы 3 дополнительных экшен тайпа жизненного цикла промиса

В КОДЕ выглядит это будет так - //"topReducer/asyncIncrementAction/pending" ну и так же 2 других состояния, мы еще к пониманию этого вернемся когда будем писать редьюсеры на экшены

Следующий аргумент payloadCreator – коллбэк функция которая должна возвращать промис, содержащий результат некоторой асинхронной логики, кстати может и синхронно возвращать значения (??? если есть ошибка, она должна либо возвращать отклоненный промис(reject) либо простое значение, такое как описательное сообщение об ошибке, либо иным образом успешный промис с аргументом RejectWithValue, возвращаемым функцией thunkApi.rejectWithValue – да согласен что такое вообще происходит,

Функция пейлоадКреатор может содержать любую логику необходимую для вычисления результата Это может включать стандартный запрос на извлечение данных АДЖАКС или несколько таких вызовов с конечным результатом и так далее ??

??А может сначало бить код а потом рассказывать??

Давайте напишем что нам нужно

export const asyncIncrementAction = createAsyncThunk(

    'topReducer/asyncIncrementAction',

    async (\_,thunkApi) => {

        await new Promise(resolve => setTimeout(resolve, 3000))

    }

)

Принимает 2 аргумента - первый принимает значения которые будут переданы экшен креатора танк при его отправке, используется для данных требующихся при запросе, если не одно значение то используется объект

Второй аргумент ThunkApi, содержит все параметры, которые обычно передаются функции redux thunk и дополнительные параметры: типо диспатч, гетСтейт и еще, здесь вам сразу скажу, что там много разных параметров и конечно же все мы рассмотреть не сможем, поэтому загляните в документацию и почитайте

Обработаем его в редьюсере, попутно закоментируем не нужные нам уже консоли, чтобы не мешались и заодно еще и мидлвар

.addCase(asyncIncrementAction.fulfilled, (state, action) => {state.counter += 2; state.action = 'Увеличили асинхронно'})

Теперь закинем на кнопку диспатч и посмотрим на это все еще раз в целом

onClick={()=>dispatch(asyncIncrementAction()) На третью кнопку

Теперь нажмем и посмотрим в консоль где мы отлавливаем все экшен тайпы

ПРОВЕРИТЬ ОТЛАВЛИВАЕТСЯ ЛИ В ДРУГОМ РЕДЬЮСЕРЕ матчинг

Смотрите наглядный итог происходящего – наша функция криэйтАсинтанк сформировала нам от промиса 3 экшена если так много вообще сказать для обработки всех состояний как я говорил до этого и мы обрабатываем на данный момент состояние успеха в редьюсере (ПОКАЗАТЬ)

Если мы посмотрим на консоль, то мы можем обработать и состоние ошибки, давайте закинем в нашу асинхронную функцию реджект

return Promise.reject(new Error('У меня ошибка'))

и напишем addMatcher на ошибку .addMatcher((action) => action.type === 'topReducer/asyncIncrementAction/rejected',

(state, action) => {state.action = action.error.message;})

И подключим в нашу компоненту action из стейта <div>{action}</div> в конце кода

Теперь запустим код, мы забрали ошибку, вы кстати можете увидеть что если написать универсальную функцию на на отлов всех реджект состояний текущего редьюсера и будет делать какие то манипуляции, так же порядок выполнения Матчеров не нарушится, они все равно все выполнятся

ЭКШЕН КРИЭЙТОР - В конце сравнить с редакс, а для тех кто не знает поправить изначально полный рассказ и вспомнить порядок редакса НЕ РАССМОТРЕЛИ ДОП ФУНКЦИОНАЛ ЭКШЕН КРЕЕЙТОРА

Давайте еще установим дисейбл на время асинхронной эмуляции и здесь мы немного расширим наши знания по экшен криэйтору. Мы хотим чтобы экшен криэтор принимал от нас тру или фэлс, чтобы мы сами регулировали когда включить и выключить дисейбл на кнопках

Функция экшен криэйтЭкшен принимает тип действия – строку, впринципе можно и не строку и он ее приведет к строке сам, но не рекомендуется, это что касается создания, вернет он функцию которая будет возвращать наш объект экшена, эту функцию мы можем вызвать с аргументом, которая положит данные в раздел экшенПейлоад нашего возвращаемого экшен объекта, которую мы обработаем уже в редьюсере

Теперь на примере создаем export const disabled = createAction('topReducer/disabled');

Теперь вызовем ее внутри асинхронной функции как раз воспользовавшись thunkApi

async (\_,thunkApi) => {

        thunkApi.dispatch(disabled(true));

        await new Promise(resolve => setTimeout(resolve, 3000));

        thunkApi.dispatch(disabled(false));

        return Promise.reject(new Error('У меня ошибка'))

    }

Добавим обработку в редьюсер

.addCase(disabled, (state, action) => {state.disabled = action.payload})

И подключим в компоненте наблюдение за дисейбл – достаем дисеблед в юзеСелекторе и пробрасываем в асинхронную кнопку  disabled={disabled}

Проверяем – все работает. Кстати вы наверное заметили что с учетом 3х обработчиков на разные состояние запросов вы и там можете делать какие то действия индивидуальные на каждом положении. К примеру

.addCase(asyncIncrementAction.pending, (state, action) => {state.disabled = true})

И закоментируем         thunkApi.dispatch(disabled(true));

Запускаем, вуаля, все работает – все таки по правильному включить выключать именно лоадеры и дисеблы на асинхронные действия лучше обрабатывать в экшенах асинхронных, ну если вы что то не придумали очень интересное )

Мы впринципе сделали все что хотели в верхней панели, давайте посмотрим на предыдущий проект и оценим, наши два кода, - длина кода - разнообразие функционала и упрощение в целом - (зайти в компоненту)

ОТДЕЛИТЬ ДЕЙСТВИЯ ОТ СЛОВ

Мы с вами рассмотрели createReducer c колббэкБилдером, как мы уже обсуждали раньше есть еще подход с МЭП ОБЖЕКТ нотацией, сейчас мы быстренько рассмотрим этот подход, но в нем есть приличные минусы, например он менее читаем и в дальнейшем мы будем рассматривать метод createSlice, который может сразу формировать экшены изнутри и не только, так вот там нельзя будет использовать экшенМатчеры и экшенДефолты. Но я еще напомню про это. Давайте начнем

КРИЭЙТ РЕДЬЮСЕР МЭП ОБЖЕКТ НОТАЦИЯ

Его суть заключается в том, что функция криэйтРедьюсер принимает объект, в котором ключи являются строковыми типами действий, а значения функция редьюсера для обработки этих типов действий.

В документации написано что этот метод не работает с тайпскриптом и плохую интеграцию с ИДЕ, и они рекомендуют использовать билдер коллбэк нотацию

(ИЗ АСИНК ТАНК КОМЕНТИМ РЕТУРН РЕДЖЕКТ)

const topReducer = createReducer(

    initialState,

    {

    [increment]: (state, action) => ({...state, counter: state.counter + 1, action: 'Увеличили'}),

    [asyncIncrementAction.fulfilled]: (state, action) => {state.counter += 2;state.action = 'Увеличили асинхронно'},

    },

    [

        {

            matcher: (action) => (action.type === 'topReducer/asyncIncrementAction/rejected'),

            reducer(state, action){console.log('action type -', action.type)}

        }

    ],

    (state) => {console.log(current(state))}

)

(ПРОВЕРЯЕМ ЧТО МЫ НАПИСАЛИ)

Кстати здесь явно видно как подключается наш дефолт обработчик в редьюсере. Вот впринципе весь подход

На этом разработка в топПанели закончилась и мы переходим в мидлПанел

МИДЛ ПАНЕЛ (ПОКАЗЫВАЕМ ФУНКЦИОНАЛ)

Я хотел там разгуляться и поделать побольше, но потом понял что по времени митапа я абсолютно не укладываюсь во многом и мы здесь будем хитрить, а потом кто захочет посмотреть, код будет доступен в гитхабе

Все таки наша цель познакомится с функционалом

Давайте посмотрим, я перенес уже компоненты и запросы для начала нашей работы, чтобы не отвлекаться