**Redux Essentials, Part 2: Redux Toolkit App Structure**

What You'll Learn

* The structure of a typical React + Redux Toolkit app
* How to view state changes in the Redux DevTools Extension

## The Counter Example App

Боже, ну зачем? Почему нельзя сделать сначала какую-то схему или алгоритм, а потом показать, как он работает на примере. Горит. Я буду по-своему писать.

Шаг 1.

Чтобы пользоваться rtk, нужно установить rtk в свой проект (шаг 0 – создать проект)

npx create-react-app redux-essentials-example --template redux

Шаг 2.

Redux и RTK – одно и то же, просто Redux – это еще полено, а RTK – уже Буратино. В основе своей – дерево, просто в разной степени готовности, так скажем. RTK все еще опирается на концепции Redux. Главная идея redux – отдельно лежащий стор. Создадим стор.

import { configureStore } from '@reduxjs/toolkit' – это базовая функция для создания стора

export default configureStore({

reducer: { }

})

В эту функцию кладется объект с полями. В т.ч. есть поле reducer. Помнишь, мы говорили, что даже на начальном этапе, когда юзер еще ничего не сделал, уже надо что-то показать? Вот редьюсер даже за отрисовку начального состояния отвечает. Ну и за апдейт. Но щас он пустой объект.

Вщ в документации сразу такую штуку нарисовали

import { configureStore } from '@reduxjs/toolkit'

import counterReducer from '../features/counter/counterSlice'

export default configureStore({

reducer: {

counter: counterReducer

}

})

Т.е configureStore принимает объект, где сто проц есть поле reducer, значение которого тоже объект со своими полями. Бред? Ну типа. Дело в том, что у нас таких управленцев – reducerов мб хоть 100 штук. Если мы ток платья продаем одно дело. Одного чувака мб и хватит. А если мы Амазон? Мы продаем всееее. Один паренек reducer не справится с таким объемом. Будет путаница. Куча мелких редьюсеров у нас есть, ок. Но должен же быть босс! И фот этот-то reducer

export default configureStore({

reducer: { - вот этот вот

counter: counterReducer

}

})

Он главный управленец. А таких вот

export default configureStore({

reducer: {

counter: counterReducer – как этот

}

})

Мб дофига и больше. Прикол редакса – стимпанк. Все мелкие редьюсеры собираются в одного большого редьюсера. Ну прямо трансформеры или лего. В простом редаксе ты это делаешь с помощью функции combineReducers. Вручную крч. А тут все под капотом. Тебе просто надо перечислить в reducer (биг босс, в которого собираются мелкие сошки) всех редьюсеров, что ниже рангом.

#### Redux Slices – или шаг 3 – разрежь все на смысловые куски

Все, щас буду тупить. Крч. Целиком колбасу в рот запихивать сложно? Сложно. Проще ее нарезать и съесть.

Редакс тулкит также думает. И про приложения тоже. Надо разрезать приложение на смысловые куски. Ну например в интернет магазе сто проц есть список покупателей (без учетки ты как купишь что-то?), сто проц есть список товаров ну и еще что-то есть.

Вот два тебе смысловых куска – товары и все, что с ними связано, и юзеры и все, что связано уже с ними. Т.е. юзеров не продаем, у товаров пароль и данные карты не спрашиваем. Соответсвенно, делаем так: юзеров регистрируем, логиним, спрашиваем инфу о карте, индивидуальные скидки рассчитываем, статистику покупочек ведем и тп. Товары продаем, добавляем в избранное, следим, скок осталось, делаем статистику продажей, цену отслеживаем и тп. Вот. Это куски смысловые. Разумно варенье с паштетом не мешать? Разумно. Так и в коде. Пусть все, что связано с товарами – лежит в своем куске-файле. А все, что связано, с юзерами – в своем куске-файле. Да, кусок = файл.

Как всегда пример из идиотской документации. Это, если что, слайсы соцсети.

import { configureStore } from '@reduxjs/toolkit'

import usersReducer from '../features/users/usersSlice'

import postsReducer from '../features/posts/postsSlice'

import commentsReducer from '../features/comments/commentsSlice'

export default configureStore({

reducer: {

users: usersReducer,

posts: postsReducer,

comments: commentsReducer

}

})

Что нам тут важно? Импорты слайсов.

import usersReducer from '../features/users/usersSlice'

import postsReducer from '../features/posts/postsSlice'

import commentsReducer from '../features/comments/commentsSlice'

В слайсах лежит все, что угодно. Любая инфа, связанная с темой. Но в этой мешанине варится млекий управленец – редьюсер. Это его болото. Он знает, что нужно делать с этой мешаниной. Т.е. в нашем собственном примере дб редьюсер, отвечающий за товары. Все, он шарит за них. Какой-то вопрос по товару? Это к нему. И дб редьюсер, шарящий в юзерах. Все вопросы по юзерам – к нему.

И вот главный редьюсер просто собирается из таких вот мелких.

Какой алгоритм? Забегая вперед, как я это понимаю: в каждом мелком редьюсере (редьюсер какого-то отдельного смыслового куска приложения) есть свои подчиненные. Тоже редьюсеры. Из которых еще можно достать actions. Вщ бреееед. Боже, что за матрёшка! Ладно, забудь пока. Они -микро, нано пыль. Вщ крошки-малышки. Принеси-подай, называется. Например, есть главный редьюсер, основной, босс, папочка, rootReducer иначе говоря. У него есть руководители отделений в подчинении. Отделение в данном случае – это смысловой кусок. У этих руководителей есть просто рабочие. Планктончики-редьюсеры. Но у них же тоже свои задачи. У каждого своя. Так вот. У одного такого планктона– задача инкрементировать счетчик. У другого наоборот декрементировать. Руководитель отделения – reducer в куске под названием counterSlice. Этот руководитель отделения-куска в свою очередь ниже биг босса, главного редьюсера. В этом же биг боссе есть другие руководители отделений. Например, есть кусок, который отвечает за смену темы.

Биг Босс, директор – редьюсер всего приложения

Редьюсер из куска counterSlice (руководитель отделения)

планктончики (increment, decrement)

Редьюсер из куска ThemeSlice (руководитель отделения)

планктончики (changeTheme, да мб и один)

Про поток еще поговорим, но ладно, пара слов. Мы импортнем мелких редьюсеров в те компоненты, где они нужны. А там уже rtk сам это все распределит, как ему надо. Ну пока я так поняла. Наверное он думает так: ага в компонентче счетчик нужно заинкрементить. Видит слово increment, ищет, откуда оно, ну и там по накатанной? Хз крч, потом.

### Creating Slice Reducers and Actions

Как вщ выглядит кусок?

import { createSlice } from '@reduxjs/toolkit'

export const counterSlice = createSlice({

name: 'counter',

initialState: {

value: 0

},

reducers: {

increment: state => {

// Redux Toolkit allows us to write "mutating" logic in reducers. It

// doesn't actually mutate the state because it uses the immer library,

// which detects changes to a "draft state" and produces a brand new

// immutable state based off those changes

state.value += 1

},

decrement: state => {

state.value -= 1

},

incrementByAmount: (state, action) => {

state.value += action.payload

}

}

})

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

export default counterSlice.reducer

1. Итак. Чтобы сделать (отрезать хах) кусок, надо вызвать функцию createSlice. Она принимает объект. Что лежит в объекте? Всякая инфа про этот кусок.
   1. Например, имя куска. Это поле name. Зачем оно надо? Слайс умный, он сам тебе создаст action creators, сам с их помощью сделает actions, еще и имена (type) им даст – а type у action, как ты, надеюсь, помнишь, состоит из двух частей «где/что». Так вот это где берется из поля name. Например кусок counterSlice. Имя куска counter. Откуда берется что? От имени планктончика.

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

он насоздавал тебе actions. У инкремента например type: ‘counter/increment’. Где – counter (имя куска), что – increment.

* 1. Еще внутри куска лежит начальное состояние куска. По нему идет первичная отрисовка так скажем. Но стейт потом меняется, в зависимости от действий юзера.  
       
     Каждый кусок интересует своя часть общего стора. Представь холодос. Он стоит в квартире, где живут: бодибилдер, анорексичка и веган. Каждый покупает себе еду и кладет ее в общий холодос. Бодибилдер, анорекчика и веган – это смысловые куски – слайсы. Общий холодос – это глобальный стор. Стейт внтури кажжого слайса – это набор продуктов. У куска вегана набор продуктов(стейт) – это морковка. Его морковка наполняет холодос. Теперь читай дальше.

Представь, что кусок бодибилдера в холодильнике ищет протеин и яйца. Ему не сдалась морковка, ему нужны мышцы, а не диета. А кусок анорексички в холодильнике ищет бутылочку водички. И ему вщ плевать на все остальное в холодильнике. Зачем начальное состояние? Холодос кто заполняет? Люди, живущие в одном доме, где этот холодос стоит. Анорексичка, бодибилдер, веган и пр. живут в одном доме. Каждый сам покупает себе продукты и кладет в общий холодос. В итоге их наборы продуктов (состояний) заполняют холодос. Анорексичка положила бутылку с водой. Веган – 3 килограмма морковки. Бодибилдер – яйца и протеин. Итого в холодильнике (большом сторе) лежит: бутылка воды, 3кг морковки, яйца и протеин. Можно ли есть чужое? Чтоб не перепутать(как я это вижу), они подписали, где чье. Это – набор бодибилдера(эти продукты-состояние принадлежат такому-то чуваку-куску). Это – набор анорексички и тд.

reducers: {

increment: state => {

// Redux Toolkit allows us to write "mutating" logic in reducers. It

// doesn't actually mutate the state because it uses the immer library,

// which detects changes to a "draft state" and produces a brand new

// immutable state based off those changes

state.value += 1

},

decrement: state => {

state.value -= 1

},

incrementByAmount: (state, action) => {

state.value += action.payload

}

}

})

Фак, они же не редьюсер называются, а редьюсеры! Блин. А поч они потом экшнс вообще называют? Просто конченное объяснение в документации. Горит.

А! Они потом еще и такую фигню делаюбт export default counterSlice.reducer

Я гений. Я объяснила себе сама то, что они не удосужились объяснить.

Я крч буду объяснять по-своему. Их объяснение – это полное разочарование.

* 1. Поле reducers в куске. Это те планктончики-редьюсеры. Они – функции, которые, как типичные редьюсеры принимают state текущий и прилетевший action. Откуда стейт? Из initial state. Ну и потом его меняют. Крч он тоже внутри куска. Откуда action? Ты его видишь? Я тоже нет. А он есть! Где? А кусок его сам создал. Он очень самостоятельный. Он создал actionCreators, которые лепят actions. Их там даже мб настраивать, но я пока хз. Дойдем, наверное. Как он создает actions? Ну как я догадываюсь так: берем имя куска, берем имя планктончика-редьюсера. Тяп, ляп – получился type. Payload, я так полагаю, по той же схеме – увидел, что этому редьюсеру он нужен, записал в action, что дб payload. Ну или всем автоматически пишет пустые…Нам не говорят, как это работае. Говорят, мол, в отличие от редакса простого в ртк все автоматом, а не вручную. Так вручную это одним способом реализовано, а тут совсем другим! Мы же там сами actionCreators создавали. Сами поля прописывали в action. А тут мы даже не вызываем ничего мол сделай-ка за нас эту работу. Она делается вообще в тени. Даже хз, какой там механизм. Откуда что берется.

reducers: {  
 increment: state => {  
 state.value += 1  
 },  
 decrement: state => {  
 state.value -= 1  
 },  
 incrementByAmount: (state, action) => {  
 state.value += action.payload  
 }  
 }  
}

Внутри плнактончиков-редьюсеров можно мутировать стейт. С чего бы это? С того, что в кусках в принципе можно мутировать стейт. Ведь в кусках вшит immer. Да, rtk тянет и его. Причем ток в одном месте – в кусках. Странная фигня. Просто, чтобы опростоволоситься, если забудешь, что ток в кусках. Ну вот так.

* 1. Экспорты из куска

export const { increment, decrement, incrementByAmount } = counterSlice.actions ------это экспортирует экшены, которые создались автоматически. Эти экшены потом нужно будет передавать через диспатч в компонентах, где нужен этот кусок(я бы добавила логики ну или стора).  
  
export default counterSlice.reducer – все планктончики-редьюсеры по принципу трансформеров или лего собираются в одного редьюсера – руководителя отделения(куска).

Биг Босс, директор. Главный решала

Отделение 1 – кусок 1

Руководитель отделения 1 – редьюсер

Плакнтончики отделения 1 – редьюсеры

Отделение 2 – кусок 2

Руководитель отделения 2 – редьюсер

Плакнтончики отделения 2 – редьюсеры

Отделение 3 – кусок 3

Руководитель отделения 3 – редьюсер

Плакнтончики отделения 3 – редьюсеры

Отделение 3 – кусок 3

Руководитель отделения 3 – редьюсер

Плакнтончики отделения 3 - редьюсеры

### Writing Async Logic with Thunks

Вот он алгортим, ура! Actions are dispatched, the store runs the reducers and calculates the new state, and the dispatch function finishes

Весь наш код до этого синхронный. Любые сайдэффекты в редьюсерах запрещены, в тч асинхрон. Но что делать, если надо?

Делай thunk. Thunk – это по сути кусок кода с задержкой. Вщ неважно, поч задержка и сколько.

Thunks are written using two functions:

* An inside thunk function, which gets dispatch and getState as arguments
* The outside creator function, which creates and returns the thunk function

Я не то чтобы круто понимаю логику этой матрешки. Но по сути голубя схватили, че-то с письмом или с самим голубем поделали и позволили голубю с пиьмом лететь дальше. Голубь – dispatch Письмо – action.

// The function below is called a thunk and allows us to perform async logic.

// It can be dispatched like a regular action: `dispatch(incrementAsync(10))`.

// This will call the thunk with the `dispatch` function as the first argument.

// Async code can then be executed and other actions can be dispatched

export const incrementAsync = amount => dispatch => {

setTimeout(() => {

dispatch(incrementByAmount(amount))

}, 1000)

}

Юзать это точно также можно

store.dispatch(incrementAsync(5))

Это middleware(thunks) потому что это прослойка между отправкой экшена в редьюсер. Диспатч перехватили. И еще имеют доступ к функции getState. Это уже не про голубя. У него бы этого точно не было. Это уже скорее связь со стором. Т.е. нити ведут в обе стороны – можно изменить action, потому что в руках у нас dispatch, и можно получить инфу о сторе через getState(это же его метод).

По сути нужен пакет thunk, но в ртк он вшит

// the outside "thunk creator" function

const fetchUserById = userId => {

// the inside "thunk function"

return async (dispatch, getState) => {

try {

// make an async call in the thunk

const user = await userAPI.fetchById(userId)

// dispatch an action when we get the response back

dispatch(userLoaded(user))

} catch (err) {

// If something went wrong, handle it here

}

}

}

Потом доделаю

### The React Counter Component

Здравствуй, бред перепрыгивания.

По сути, концептуально, о чем была эта часть?

В реакт редакс есть плбшки

Сначала надо наверно установить react-redux? Ни слова об этом нет, мда.

Крч, плюшки – useSelector – это вытащить инфу из стора, какую надо. useDispatch – это чтобы было просто dispatch, когда action передаешь. Крч, чтобы проще голубю в клюв воткнуть письмо.

Без useDispacth store.dispatch({}) – ну ты пон

А с ним так

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

const dispatch = useDispatch()

onClick={() => dispatch(increment())}

useSelector еще покажу

const count = useSelector(selectCount)

(селект если что отсюда

import {  
 decrement,  
 increment,  
 incrementByAmount,  
 incrementAsync,  
 selectCount  
} from './counterSlice'

)

Вот тут этот count, т.е. инфу, которую из стора вытащили и в переменную запихнули, и применили

<span className={styles.value}>{count}</span>

Необязательно кст ток уже готовые из куска вытаскивать экшены. Можно и инлайном все прописать

We don't have to only use selectors that have already been exported, either. For example, we could write a selector function as an inline argument to useSelector:

const countPlusTwo = useSelector(state => state.counter.value + 2)

### Component State and Forms

Состояние, нужное ток одному компоненту, в стор глобальный пихать не надо. Инпуты например не надо. Еще м задать вопрос, нужен ли мне дебаггинг с путешествием во времени. И еще два вопрроса, которые намекают на то, что стор нужен, но по сути реакт бы справился с этим и без редакса. Реакт не просто UIчик делает

 Do you want to cache the data (ie, use what's in state if it's already there instead of re-requesting it)?

 Do you want to keep this data consistent while hot-reloading UI components (which may lose their internal state when swapped)?

### Providing the Store – по сути это шаг 5

У тебя есть стор и вщ все круто. Но ничего не работает. Потому что надо реакту показать, вон, мол, стор наш, дай его всем, кому надо.

import React from 'react'

import ReactDOM from 'react-dom'

import './index.css'

import App from './App'

import store from './app/store'

import { Provider } from 'react-redux'

import \* as serviceWorker from './serviceWorker'

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

Now, any React components that call useSelector or useDispatch will be talking to the Redux store we gave to the <Provider>.

Summary

* **We can create a Redux store using the Redux Toolkit configureStore API**
  + configureStore accepts a reducer function as a named argument
  + configureStore automatically sets up the store with good default settings
* **Redux logic is typically organized into files called "slices"**
  + A "slice" contains the reducer logic and actions related to a specific feature / section of the Redux state
  + Redux Toolkit's createSlice API generates action creators and action types for each individual reducer function you provide
* **Redux reducers must follow specific rules**
  + Should only calculate a new state value based on the state and action arguments
  + Must make *immutable updates* by copying the existing state
  + Cannot contain any asynchronous logic or other "side effects"
  + Redux Toolkit's createSlice API uses Immer to allow "mutating" immutable updates
* **Async logic is typically written in special functions called "thunks"**
  + Thunks receive dispatch and getState as arguments
  + Redux Toolkit enables the redux-thunk middleware by default
* **React-Redux allows React components to interact with a Redux store**
  + Wrapping the app with <Provider store={store}> enables all components to use the store
  + Global state should go in the Redux store, local state should stay in React components