Redux

Библиотека Redux создана для продвинутого контроля данными. Сам по себе Redux универсален и, использовать его можно в проекте без фреймворке, но Redux очень хорошо прижился в связке с React.

**Создание store**

**Store** – это сущность, которая хранит в себе все данные приложения.

import { createStore } from 'redux'

const store = createStore()

Для store необходимо наличие **функции**, которая изменяет содержимое. В эту функции передается само состояние:

function playlist(state = []) {

return state

}

const store = createStore(playlist)

Теперь есть пустое по-умолчанию состояние. Теперь его следует изменить и для есть функция dispatch.

store.dispatch({ type: 'ADD\_TRACK', payload: 'One Step Closer' })

В **dispatch** передается объект **событие** с обязательным свойством **type**, обозначающий тип операции, а следующее свойства, содержащее новое значение, как правило, называют **payload**.

Но вызвав dispatch, состояние store не изменится, т.к. **его нужно обработать**.

function playlist(state = [], action) {

switch (action.type) {

case 'ADD\_TRACK': return [

...state,

action.payload

]

}

return state

}

В новой версии playlist добавили параметр action, принимающий событие. По свойству type события **определяется, что нужно сделать**.

Подмечу, что в Redux все **иммутабельно**, т.е. нельзя изменить все состояние за раз, нужно сначала сделать его копию и уже изменить копию. Иммутабельность позволяют ускорить производительность. Поначалу это кажется непривычным, но с использование оператора spread и деструктуризации станет легче привыкнуть.

**Подписки**

Состояние изменилось, но как его получить? Для этого есть метод **getState**:

store.dispatch({ type: 'ADD\_TRACK', payload: 'One Step Closer' })

console.log(store.getState())

В идеале еще необходимо **следить за состоянием**, поэтому реализованы **подписки**:

store.subscribe(() => {

console.log('subscribe', store.getState())

})

Теперь при любом изменении состояния можно будет про это узнать.

**Отрисовка состояния**

Все примеры выше не использовали React, что нужно исправить. Сначала нужно установить библиотеку **react-redux**, которая позволяет соединить эти две библиотеки.

Теперь в index.js можно **создать store и отрендерить корневой элемент**:

import { createStore } from 'redux'

function reducer(state = [], action) {

switch (action.type) {

case 'ADD\_TRACK': return [

...state,

action.payload

]

}

return state

}

const store = createStore(reducer)

ReactDOM.render(

<App />,

document.getElementById('root')

)

Теперь нужно подключить store к App (и всем его дочерним компонентам) через HOC компонент **Provider**, **передав store**

import { Provider } from 'react-redux'

ReactDOM.render(

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

)

Окей. Теперь в App можно будет получить store, но только через вызов **connect**

import { connect } from 'react-redux'

function App({ store, addTrack }) {

console.log(store, addTrack)

return <div/>

}

export default connect(

state => ({

store: state

}),

dispatch => ({

addTrack(payload) {

dispatch({ type: 'ADD\_TRACK', payload })

}

})

)(App)

Функция connect принимает **две функции**. Первая функцию, в свою очередь, содержит **состояние**, а вторая метод **dispatch**. Обе должны вернуть объекты, которые **добавятся в пропсы** компонента. Вызов connect возвращает HOC, которому нужно передать сам компонент.

Теперь на основе всего можно реализовать полное приложение:

function App({ store, addTrack }) {

const [value, setValue] = useState(' ')

function btnHandler() {

addTrack(value)

setValue(' ')

}

function inputHandler(e) {

setValue(e.target.value)

}

return <div>

<input

value={value}

onChange={inputHandler}

/>

<button

onClick={btnHandler}

>

Add

</button>

<ul>

{store.map((el, index) =>

<li key={index}>{el}</li>

)}

</ul>

</div>

}

**Комбинирование reducers**

В приложении должен быть только один reducer, который добавляет треки, но что если мы нужно добавить жанры (genre)? Во-первых, изначальный state будет объект с двумя полями tracks и genre, соотвественно изменится и изменение состояния.

const initialState = {

tracks: [],

genre: []

}

function reducer(state = initialState, action) {

switch (action.type) {

case 'ADD\_TRACK': return {

...state,

tracks: [

...state.tracks,

action.payload

]

}

}

return state

}

Во-вторых, получение в App треков тоже изменится. Теперь нужно реализовать изменение для жанров.

function reducer(state = initialState, action) {

switch (action.type) {

case 'ADD\_TRACK': return {

...state,

tracks: [

...state.tracks,

action.payload

]

}

case 'ADD\_GENRE': return {

...state,

genre: [

...state.genre,

action.payload

]

}

}

return state

}

Но если писать все в одном reducer, то код станет громоздким и непонятным, поэтому придумали combineReducers, позволяющий разделить reducer на отдельные части.

Все reducer, как правило, создаются в отдельной папке reduce, где же они совмещаются в один reducer.

import { combineReducers } from 'redux'

import genre from './genre'

import tracks from './tracks'

export default combineReducers({

genre,

tracks

})

Файлы genre.js и tracks.js содержат reducer.

genre.js

const initState = [

'rock',

'hard rock',

'alt rock',

'new metal'

]

export default function genre(state = initState, action) {

switch (action.type) {

case 'ADD\_GENRE':

return state.concat(action.payload)

default:

return state

}

}

tracks.js

const initState = [

'Highway to Hell',

'Back in black',

'Rollin',

'Creep'

]

export default function tracks(state = initState, action) {

switch (action.type) {

case 'ADD\_TRACK':

return state.concat(action.payload)

default:

return state

}

}

Описывать reducer стало легче, потому что теперь можно изменить только отведенную область, а остальное обновит сам Redux.

**redux-thunk**

В redux, по-умолчанию, все события синхронные и когда нужно работать с асинхронным кодом, нужно подключить дополнительную прослойку (middleware) для поддержки асинхронного кода.

npm i redux-thunk

Теперь ее необходимо подключим к store.

import thunk from 'redux-thunk'

const store = createStore(reducer, thunk)

Подключив middleware можно работать с асинхронным кодом, если вызывать такой код в подобных конструкциях:

const getTracks = () => dispatch =>

setTimeout(

() => {

dispatch({ type: 'FETCH\_TRACK\_SUCCESS', payload: ['Hello'] })

},

2000

)

Здесь создается отдельная функция, которая возвращает новую функцию с входящим в нее методом dispatch. В возвращаемую функцию можно писать любой асинхронный код, в котором через dispatch можно отправлять события.

Таких функций в проекте может быть много, и все они создаются в отдельной папке actions и использоваться они могут так:

import { getTracks } from './actions/tracks'

function App({ tracks, getTracks }) {

return <div>

<button onClick={getTracks}>

Get Tracks

</button>

<ul>

{tracks.map((el, index) =>

<li key={index}>{el}</li>

)}

</ul>

</div>

}

export default connect(

state => ({

tracks: state.tracks

}),

dispatch => ({

getTracks() {

dispatch(getTracks())

}

})

)(App)

То есть нужно получить функцию с асинхронным кодом внутри и вызвать ее в методе dispatch.