# Блок 3. Redux

## Задание 1. Начинаем применять Redux

Продолжаем работать с нашим React-проектом.

1. К настоящему моменту мы разработали отличное приложение, выполняющее типичные задачи UI. Но мы попробуем переключиться от чистого React к связке React + Redux, чтобы продемонстрировать достоинства этой технологии.
2. Первое, что вам потребуется сделать:

**npm install redux react-redux**

1. Создайте основные папки проекта: Actions, Reducer, Store, Constants, Components, Containers.

mkdir Actions && mkdir Reducer && mkdir Store && mkdir Constants && mkdir Components && mkdir Containers

1. Переместите **grid.js**, **user-details.js** и **summaries.js** из папки **js** в **Components**. Также переместите **app.js** в **Containers**
2. Обновите импорты в **app.js** для UserDetail, GridComponent и Summaries чтобы они импортировались из папки Components
3. Создайте пустой файл **index.js** в папках Reducer, Actions, Store, Constants
4. Окей, можно приступать. Прежде всего, обновите конфигурацию в вашем **webpack.config.js**:

**const** PATHS = {  
 app: "./containers/app",  
 build: path.join(\_\_dirname, 'build')  
};

1. Обновите **app.js**:

**import** { Provider } **from** 'react-redux'

**import** configureStore **from** '../store/index'

**const** store = configureStore();

render(  
 <Provider store={store}>  
 <HashRouter>…</HashRouter>  
 </Provider>,  
 document.getElementById('app')  
);

1. Давайте разберемся, что мы сейчас проделали:

Мы удалили **js/app.js** и добавили **Containers/app.js**. Чтобы webpack мог работать, мы добавили фиксированную точку входа.

Redux приложение должно быть упаковано в Redux Provider, и это то, что мы сделали – поместили компоненты в элемент <Provider>, чтобы redux мог работать с ним как с потомком.

Мы применили store (не забывайте, что у нас единственный эземпляр store) к Provider.

1. Теперь давайте разберемся, что делать со store:

Откройте **Store/index.js** и поместите туда такой код:

**import** { createStore, applyMiddleware } **from** 'redux'  
**import** { rootReducer } **from** '../reducer'  
  
**export default function** configureStore(initialState) {  
 **return** createStore(rootReducer, initialState);  
}

Здесь мы создаем store, передавая на вход корневой (и единственный) редьюсер, который будет с этим Store работать. Затем configureStore() вызывается из app.js, чтобы получить готовый сконфигурированный Store. initialState мы пока что не передаем, но потом оно может быть использовано как начальное значение для Store.

1. Ок, теперь перейдем к редьюсерам – месту, где происходит вся магия:

Прежде всего нам нужно создать отдельные редьюсеры для таблицы и детальной информации. Для этого поместите этот код в **Reducer/index.js**:

**export function** grid(**state = gridRecords**, action){  
 **switch** (action.type) {  
 **default**:  
 **return** state  
 }  
}  
  
**export function** details(**state = detailsRecords**, action){  
 **switch** (action.type) {  
 **default**:  
 **return** state  
 }  
}

Заметили ли вы запись state = gridRecords и state = detailsRecords? В том случае, если параметры не передаются, будут использованы значения по умолчанию – это и есть наши стартовые данные. Почему нам это необходимо? Нам нужно устанавливать начальные данные для первого вызова редьюсера. Когда вы просто стартуете ваше redux приложение, вам необходимо, чтобы в Store уже хранились начальные значения. Исходно Store содержит значение undefined, и мы устанавливаем дефолтные значения непосредственно в редьюсеры. Давайте определим начальные значения следующим образом:

**let** gridRecords = [  
 {firstName: "John", lastName: "Doe", active: **false**, id: 1},  
 {firstName: "Mary", lastName: "Moe", active: **false**, id: 2},  
 {firstName: "Peter", lastName: "Noname", active: **true**, id: 3}  
],  
 detailsRecords = [{  
 id:1,  
 name:"John Doe",  
 about:"Nice guy",  
 hobby:"Likes drinking wine",  
 skills:["html", "javascript", "redux"]  
 },{  
 id:2,  
 name:"Mary Moe",  
 about:"Cute girl",  
 hobby:"Likes playing xbox whole days long",  
 skills:["Fortran", "Lua", "R#"]  
 }];

1. Окей, у нас есть редьюсер на каждом экране. Теперь давайте объединим их в один большой редьюсер – для этого мы воспользуемся очень удобной функцией combineReducers():

**import** { combineReducers } **from** 'redux'

**export const** rootReducer = combineReducers({  
 details,  
 grid  
});

Вам должно быть интересно, почему мы должны делать именно так? Ответ прост: так как у нас только один Store, самый простой способ модифицировать его данные – использовать один редьюсер. Но только попробуйте представить размер и структуру этого редьюсера для большого приложения! Вы не сможете поддерживать и понимать такой огромный редьюсер. Поэтому мы создаем много маленьких редьюсеров, а потом комбинируем их. При этом каждый редьюсер отвечает за свою часть дерева Store. В данном случае в Store лежит 2 объекта: details и grid. За изменения grid отвечает одноименный редьюсер grid(), а за изменения details – одноименный редьюсер details(). Таким образом, у нас к разным узлам дерева состояния (Store) привязаны разные функции, меняющие данные (редьюсеры), что делает всю структуру легко управляемой и поддерживаемой. Сколько логических частей в вашем приложении – столько и редьюсеров вы должны создать.

Вы можете разделять редьюсеры столько раз, сколько это необходимо. Например, если структура объекта grid достаточно сложна, и вам нужно хранить как содержание таблицы в объекте data, так и сводные данные totals, вы можете сделать 2 редьюсера data() и totals(), а в функции grid() скомбинировать их: combineRecucers({data, totals}). Таким образом, обработка даже больших и сложных деревьев состояний становится вполне посильной задачей.

Редьюсеры grid() и details() могут выглядеть, например, так:

**export function** grid(state = gridRecords, action){  
 **switch** (action.type) {  
 **case** "FILTER":  
//I also do something on filter action  
 **return** state;  
 **default**:  
 **return** state  
 }  
}  
  
**export function** details(state = detailsRecords, action){  
 **switch** (action.type) {  
 **case** "FILTER":  
//I also do something on filter action  
 **return** state;  
 **default**:  
 **return** state  
 }  
}

1. Мы вернемся к обсуждению работы с редьюсерами позже, а пока давайте двигаться дальше. Давайте подытожим, что мы узнали к настоящему моменту:
   1. Мы создали редьюсер, который объединяет 2 других редьюсера, а также предоставляет дефолтное состояние для State
   2. Мы создали Store с этим редьюсером
   3. Мы поместили существующее приложение в контейнер Redux, который называется <Provider>

1. Теперь мы можем передать данные в компоненты. Откройте **grid.js** и сделайте следующее:
   1. Удалите начальные данные dataSource – теперь их инициализацией занимается Redux
   2. Удалите export для компонента GridComponent. Теперь вместо GridComponent мы будем экспортировать компонент, привязанный к инфраструктуре Redux - для этого мы воспользуемся функцией connect()
   3. Добавьте такой код:

**import** { connect } **from** 'react-redux'

GridComponent.propTypes = {  
 records: PropTypes.array.isRequired  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 records: state.grid  
 }  
}  
  
**export default** connect(  
 mapStateToProps  
)(GridComponent)

1. Что мы здесь делаем? Мы предоставляем наш GridComponent с данными из нашего Store. Так как данные теперь приходят из Redux и передаются в props, нам нужно заменить state на props повсеместно в компоненте GridComponent:

{**this**.props.records.map((record, index)=>{

1. Также нам надо удалить этот код – мы больше не работаем с данными в компоненте, компонент теперь только отображает данные, полученные от Redux:

**this**.setState({  
 records:dataSource  
})

1. Теперь, если вы обновите страницу, вы увидите тот же компонент, но теперь он работает в инфраструктуре React + Redux
2. Но пока мы прошли только половину пути. Мы можем читать данные из Store, но как нам обновлять Store? Давайте научимся это делать. Прежде всего, нам понадобится изменить метод **toggleActive().** Вместо изменения состояния здесь мы отправляем (dispatch) сообщение (action) в Redux, чтобы это сообщение было использовано для преобразования дерева состояний. Мы передаем детали (payload): что именно произошло – **TOGGLE\_ACTIVE**, то есть переключение активности юзера, и где это произошло – номер чекбокса, по которому щелкнул пользователь (index):

toggleActive(index){  
 **let** {dispatch} = **this**.props;  
 dispatch({  
 type:"TOGGLE\_ACTIVE",  
 value:index  
 });  
}

1. Теперь обновим редьюсер, чтобы он умел реагировать на данный Action:

**export function** grid(state = gridRecords, action){  
 **switch** (action.type) {  
 **case** "TOGGLE\_ACTIVE":  
 **let** newState = [...state];  
 newState[action.value].active = !newState[action.value].active;  
 **return** newState;  
 **case** "FILTER":  
 //Filter will be implemented later  
 **default**:  
 **return** state  
 }  
}

Обратите внимание, что для корректной работы Redux абсолютно необходимо выполнять только иммутабельные преобразования – что мы и делаем, копируя содержимое state в массив newState.

1. Вот и все. Теперь можно перегрузить страницу и увидеть обновленные данные.

#### Задача 6: Сделайте фильтрацию по текстовому полю.

1. Используйте dispatch, action и reducer, чтобы отфильтровать записи в grid таким образом, чтобы поле firstname у отфильтрованных записей совпадало со значением, введенным в поле ввода.

## Задание 2. Продолжаем изучать Redux

1. Теперь давайте разберемся, что делать с деталями пользователя. Во-первых, удалим все захардкоженные значения из компонентов. Затем замаппим данные из хранилища Store в свойства props и передадим их в компонент:

**import** { connect } **from** 'react-redux'  
  
**import** UserDetail **from** './user-detail';

**import** PropTypes **from 'prop-types'**;

**class** UserDetails **extends** React.Component {  
 render(){  
 **return** (  
 <div>  
 {**this**.props.details.map((detail, i)=>{  
 **return** <UserDetail key={i} detail={detail}/>  
 })}  
 </div>  
 )  
 }  
}  
  
UserDetails.propTypes = {  
 details: PropTypes.array.isRequired  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 details: state.details  
 }  
}  
  
**export default** connect(  
 mapStateToProps  
)(UserDetails)

Здесь мы используем, помимо уже знакомой вам функции connect(), функцию mapStateToProps. Эта фунция извлекает из общего хранилища данные, которые нужно отрисовать в данном компоненте, и помещает их в props компонента. Например, в приведенном выше примере берется массив state.details из Store и помещается в props.details, чтобы впоследстии его можно было превратить в HTML в методе render().

Также мы создаем user-details.js и помещаем шаблон компонента внутрь:

**export class** UserDetail **extends** React.Component {  
 render(){  
 **let** {detail} = **this**.props;  
 **return** (  
 <div className="container">  
 <div className="row">  
  
 <div className="col-md-offset-2 col-md-8 col-lg-offset-3 col-lg-6">  
 <div className="well profile">  
 <div className="col-sm-12">  
 <div className="col-xs-12 col-sm-8">  
 <h2>{detail.name}</h2>  
 <p><strong>About: </strong> {detail.about} </p>  
 <p><strong>Hobbies: </strong> {detail.hobbies} </p>  
 <p><strong>Skills: </strong>  
 {detail.skills.map((skill, i)=>{  
 **return** <span key={i} className="tags">{skill}</span>  
 })}  
 </p>  
 </div>  
 <div className="col-xs-12 col-sm-4 text-center">  
 <figure>  
 <img src="http://www.bitrebels.com/wp-content/uploads/2011/02/Original-Facebook-Geek-Profile-Avatar-2.jpg" alt="" className="img-circle img-responsive"/>  
 </figure>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
 </div>  
 )  
 }  
}

Вот и все, должно заработать.

#### Задача 7: Если пользователь щелкает по таблице – отображать только необходимые детали пользователя.

1. Свойство, которое приходит в user-details, используется для того, чтобы отфильтровать записи
2. Если в исходных данных есть 3 записи с id 1,2 и 3, а свойство id, пришедшее из параметров роутера, равно 2, то необходимо отобразить только запись с id 2
3. Окей, неплохо. Но допустим вам понадобится переименовать FILTER\_DETAILS в SEARCH\_DETAILS. Что вам понадобится сделать – это изменить редьюсер и компонент. Это не очень удобно. Мы заменим действия (action) на создатели действий (action creator). Но сначала давайте определим их в одном месте. Откройте Constants/index.js и поместите туда такой код:

//DETAILS PAGE ACTIONS  
**export const** FILTER\_DETAILS = 'FILTER\_DETAILS';  
  
//GRID ACTIONS  
**export const** TOGGLE\_ACTIVE = 'TOGGLE\_ACTIVE';  
**export const** FILTER = 'FILTER';

1. Теперь у вас есть константы, и мы можем написать Action Creators. Перейдите в Actions/index.js и поместите туда такой код:

**import** \* **as** types **from** '../Constants'  
  
**export function** filterDetails(value) {  
 **return** {  
 type: types.FILTER\_DETAILS,  
 value  
 }  
}  
  
**export function** filterGrid(value) {  
 **return** {  
 type: types.FILTER,  
 value  
 }  
}  
  
**export function** toggleActive(value) {  
 **return** {  
 type: types.TOGGLE\_ACTIVE,  
 value  
 }  
}

1. После этого вам надо обновить редьюсер:

**import** {TOGGLE\_ACTIVE, FILTER, FILTER\_DETAILS} **from** '../Constants'

**case** TOGGLE\_ACTIVE:

**case** FILTER:

**case** FILTER\_DETAILS:

1. Мы только что определили все действия как константы в одном месте, а также Action Creators (создателей действий), которые зависят от значения и действия. Осталось обновить вызов dispatch(), чтобы он теперь не создавал Action напрямую, а вызывал Action Creator, делегируя создание Action ему:

Grid.js

**import** {filterGrid, toggleActive} **from** '../Actions'

toggleActive(index){  
 **let** {dispatch} = **this**.props;  
 dispatch(toggleActive(index));  
}  
handleFilterChange(e){  
 **let** {dispatch} = **this**.props;  
 dispatch(filterGrid(e.target.value));  
}

user-details.js

**import** {filterDetails} **from** '../Actions'

componentDidMount(){  
 **let** {dispatch} = **this**.props;  
 dispatch(filterDetails(**this**.props.params.id));  
}

## Задание 3. Работаем с сервером

1. Прекрасно, код чистый и поток данных предсказуем. Теперь давайте посмотрим, как можно загружать данные в таблицу с сервера, а не из захардкоженной переменной. Вы можете открыть папку server в материалах тренинга, установить необходимые серверу зависимости командой

**npm install**

И запустить сервер:

**node server.js**

1. Теперь у нас есть сервер, который возвращает на каждый запрос сгенерированный JSON с gridRecords и detailsRecords. Давайте разберемся, как мы можем работать с сервером, используя Redux.
2. Теперь нам надо запускать создатель действия (action creator). Но как это делать? Давайте начнем с того, что создадим действия (Actions), которые нам понадобятся для работы с сервером. Давайте поместим в константы следующие значения:

**export const** START\_LOADING = 'START\_LOADING';  
**export const** STOP\_LOADING = 'STOP\_LOADING';  
**export const** ADD\_DATA= 'ADD\_DATA';

1. Теперь мы можем определить создателей действий (action creators):

**export function** startLoading() {  
 **return** {  
 type: types.START\_LOADING  
 }  
}  
  
**export function** stopLoading() {  
 **return** {  
 type: types.STOP\_LOADING  
 }  
}  
  
**export function** addData(value) {  
 **return** {  
 type: types.ADD\_DATA,  
 value  
 }  
}

1. Мы собираемся демонстрировать процесс загрузки в grid. Для этого давайте создадим объект gridState, в котором будем хранить начальные данные таблицы, в том числе флаг loading:

**let** gridState = {  
 records:[],  
 filtered: [],  
 loading:**false** }

Теперь редьюсер должен выглядеть так:

**export function** grid(state = gridState, action){

1. Еще добавим пару редьюсеров и констант для работы с ними:

**import** {TOGGLE\_ACTIVE, FILTER, FILTER\_DETAILS, START\_LOADING, STOP\_LOADING, ADD\_DATA} **from** '../Constants'

**case** START\_LOADING:  
 **return** {...state, loading: **true**};  
**case** STOP\_LOADING:  
 **return** {...state, loading: **false**};  
**case** ADD\_DATA:  
 **return** {...state,   
 records:[...action.value]  
 });

1. Так как мы изменили переменную gridState, нам также понадобится изменить параметры вызова mapStateToProps:

GridComponent.propTypes = {  
 records: PropTypes.array.isRequired,  
 filtered: PropTypes.array.isRequired,  
 loading: PropTypes.bool.isRequired  
};  
  
**function** mapStateToProps(state) {  
 **return** {  
 records: state.grid.records,  
 filtered: state.grid.filtered,  
 loading: state.grid.loading  
 }  
}

1. Теперь мы добрались до отправки событий. Вот как вы можете это сделать:

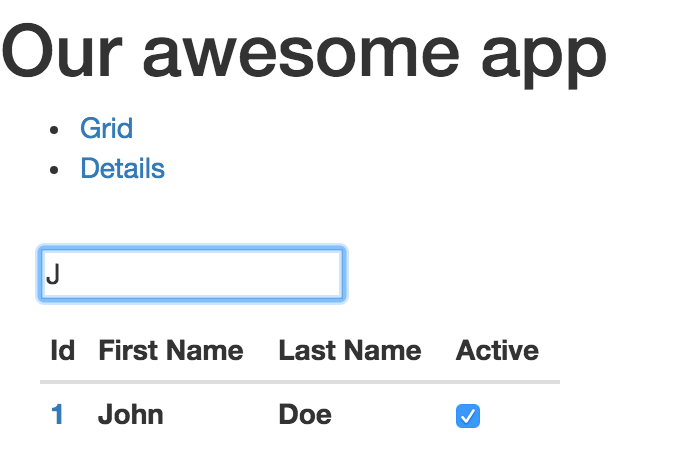
componentDidMount(){  
 **this**.refs.filterInput && **this**.refs.filterInput.focus();  
 **this**.loadData();  
}  
loadData(){  
 **let** {dispatch} = **this**.props;  
 dispatch(startLoading());  
 fetch('http://localhost:4730')  
 .then(**function**(response) {  
 **return** response.json();  
 }).then(**function**(json) {  
 dispatch(addData(json.gridRecords))  
 }).then(**function**(){  
 dispatch(stopLoading());  
 })  
}

1. Если вы сейчас обновите страницу, вы должны увидеть данные в таблицы, которые загружаются с сервера.

#### Задача 8: Необходимо исправить работу TOGGLE\_ACTIVE и FILTER редьюсеров

Пока мы делали изменения и интегрировали Action Creators, сервер и редьюсеры, мы совсем забыли о фильтре данных и переключателе Toggle Active в нашей таблице. И теперь они сломались. Вам надо разобраться, в чем проблема, и починить их.

1. Чекбокс-переключатель активности и фильтр должны заработать снова, после добавления Action Creators и редьюсеров



## Задание 4. Используем Redux Thunk

1. Посмотрите на метод loadData(): мы увидим 3 диспатча, но только одно действие – LOAD\_DATA. Слишком много логики для метода в компоненте – а ведь компонент должен отвечать только за отрисовку данных, а не за работу с сервером. Куда же нам перенести код, отвечающий за работу с сервером? Мы можем перенести его в Action Creator! Но тут есть небольшая проблема: Action Creator понятия не имеет о диспатчере – как он может его вызывать. Решение есть – мы будем использовать одно из дополнений Redux (Redux middleware) – а именно Redux Thunk.
2. Redux Thunk позволяет вам создавать Action Creator, который возвращает функцию вместо действия. В нашем случае мы будем возвращать функцию из loadDataInGrid(), которая позволит выполнять асинхронную обработку данных
3. Чтобы использовать любое из дополнений Redux (thunk, saga, logs) нам нужно установить их, поскольку thunk или saga или logs – отдельные модули, не входящие в Redux. Мы будем использовать thunk:

**npm install --save redux-thunk**

1. Перейдите в Store/index.js и замените ваш код на следующий:

**import** { createStore,applyMiddleware } **from** 'redux'  
**import** thunk **from** 'redux-thunk';  
**import** {rootReducer} **from** '../Reducer'  
  
**export default function** configureStore(initialState) {  
 **const** createStoreWithMiddleware = applyMiddleware(  
 thunk  
 )(createStore);  
  
 **const** store = createStoreWithMiddleware(rootReducer);  
 **return** store;  
}

1. Что мы тут делаем – это просим Redux создать Store, который помимо основной функциональности будет работать с thunk. Thunk будет перехватывать обращения к функции dispatch, и делать возможным использование dispatch непосредственно внутри Action Creator. Для этого функция Action Creator должна возвращать функцию, параметром которой как раз и будет dispatch – и его можно будет использовать внутри этой функции. Таким образом, мы сможем перенести работу с сервером в Action Creator loadDataInGrid():

**export function** loadDataInGrid(){  
 **return** (dispatch)=>{  
 dispatch(startLoading());  
 fetch('http://localhost:4730')  
 .then(**function**(response) {  
 **return** response.json();  
 }).then(**function**(json) {  
 dispatch(addData(json.gridRecords))  
 }).then(**function**(){  
 dispatch(stopLoading());  
 })  
 }  
}

1. Теперь мы можем удалить ссылки на startLoading, stopLoading и addData из grid.js и добавить только один Action Creator – loadDataInGrid:

**import** {filterGrid, toggleActive, loadDataInGrid} **from** '../Actions'

1. Метод loadData() также будет выглядеть гораздо аккуратнее:

loadData(){  
 **let** {dispatch} = **this**.props;  
 dispatch(loadDataInGrid());  
}

1. Теперь, если вы перезагрузите страницу, все должно работать как надо.

#### Задача 9: Необходимо отображать сообщение о загрузке данных в компоненте GridComponent

1. Когда происходит процесс загрузки, и loading устанавливается в true, нам необходимо отображать сообщение об этом в компоненте GridComponent:

<div style={{width:300, height: 300, padding: 20}}>Loading...</div>

1. Когда загрузка завершается, GridComponent должен отображать загруженные с сервера данные

#### Задача 10: Использование redux thunk для загрузки и фильтрациии данных в детальной информации о пользователях (user details)

1. Когда user details открывается через меню, необходимо показывать данные, пришедшие с сервера
2. Когда user details открываются через таблицу, нужно брать последние данные, загруженные с сервера, и отображать только одну запись с id, взятой из параметров