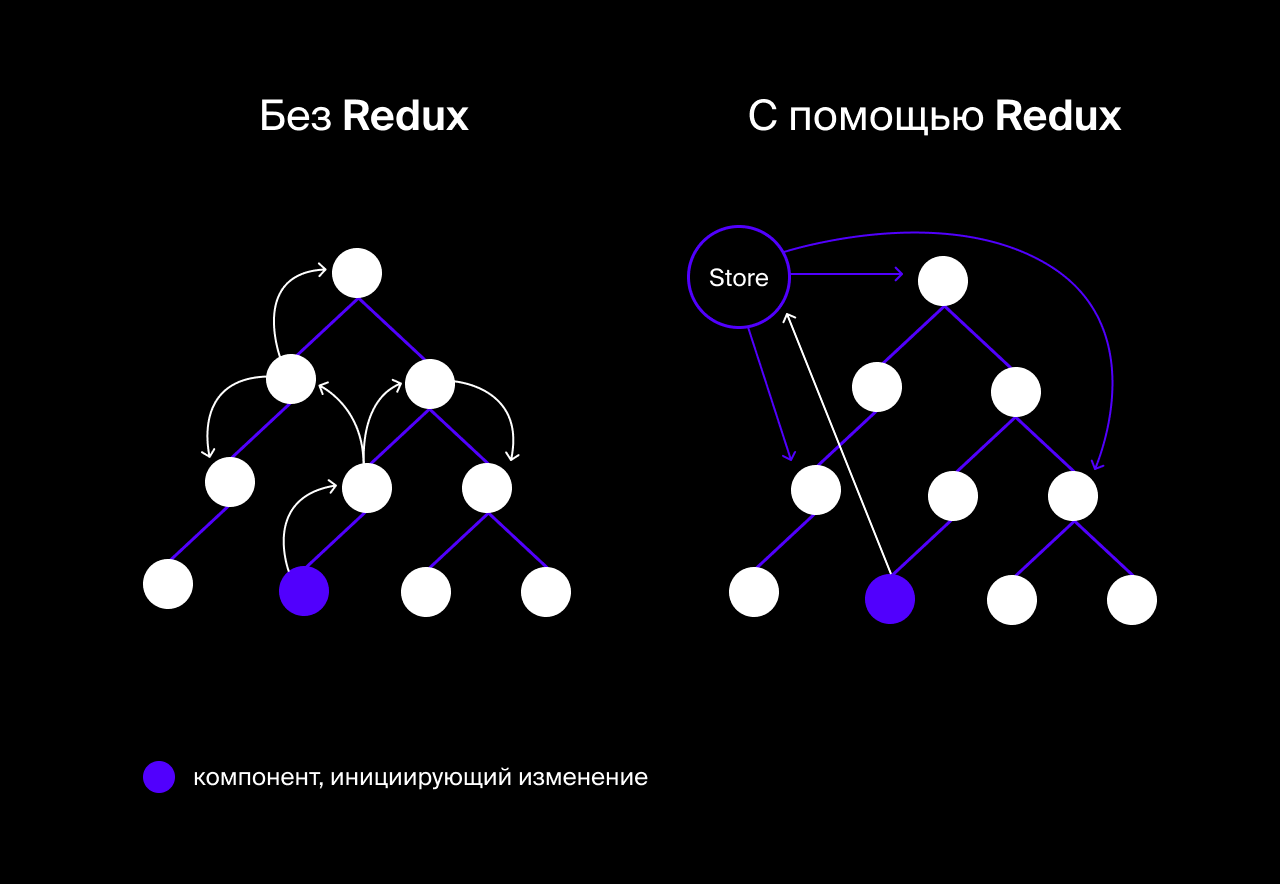
**Введение в Redux**

Redux — небольшая библиотека, которая позволяет организовать управление данными в JavaScript-приложениях. Её в 2015 году создал российский разработчик Дэн Абрамов, когда ему было 23. С тех пор Дэн работает в основной команде React в Facebook и считается одной из самых заметных фигур в современной веб-разработке.

С помощью Redux можно создать что-то похожее на базу данных прямо в JavaScript-приложении. Такая база представляет собой обычный JavaScript-объект и называется хранилищем (англ. store). Во многом этот способ организации данных похож на React Context, однако Redux — более сложная и функциональная концепция.

**Основная идея Redux**

Суть Redux — в создании единого хранилища данных и удобном доступе к этим данным на любом уровне приложения. При этом у Redux нет проблем с глубоким пробрасыванием пропсов, а каждый компонент может получить обновления из единого хранилища. В хранилище могут содержаться все данные приложения и все переменные состояния, которые влияют на интерфейс. Например, в нём можно хранить любые данные, которые пришли с сервера, идентификатор текущей страницы или состояние диалоговых окон (открыто/закрыто).



*На иллюстрации слева мы пробрасываем пропсы и наши компоненты разрастаются. Справа — используем единое хранилище*

**Преимущества**

При таком подходе у приложения появляется единый «источник правды» для его полного состояния в каждый момент времени. Каждому моменту времени соответствует определённое состояние, и с помощью Redux и инструментов отладки можно «путешествовать во времени». Это бывает полезно при исправлении ошибок. Если ошибка возникает, можно легко восстановить полное состояние хранилища и воспроизвести её. Если информация «размазана» по внутренним состояниям отдельных компонентов приложения, это сделать сложнее. По этой же причине в Redux-приложениях легче писать юнит-тесты.

**Недостатки и альтернативы**

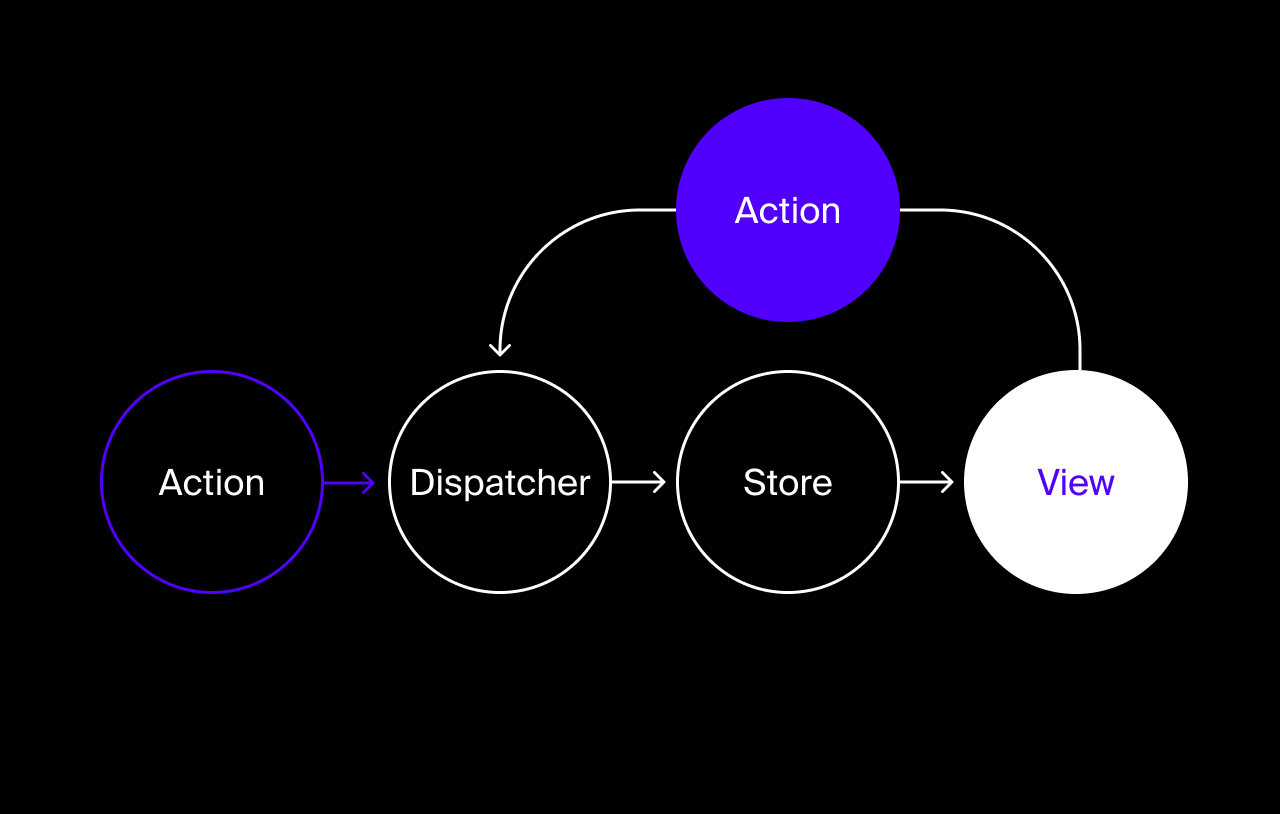
Некоторые разработчики не любят Redux из-за его «многословности». Для использования библиотеки обычно приходится писать много однотипного кода, на сленге — бойлерплейта. В проекте быстро разрастается число экшенов и их типов, а также самих редьюсеров и других вспомогательных сущностей. Что такое экшен и редьюсер — разберём в следующих уроках.

Но из-за того, что принципы Redux полезны для крупных проектов с большим количеством изменяющихся данных, периодически появляются альтернативные, упрощённые реализации. Например, Redux Toolkit призван сократить количество однотипного кода. О нём мы тоже расскажем в этой теме.

# Основные принципы и понятия Redux

Вы уже знаете, что суть Redux — в создании единого «источника истины» для приложения и удобного доступа к данным в хранилище на любом уровне приложения. В этом уроке поговорим о принципах, на которых построен Redux, и о том, как их применять.

## Принцип работы Redux



Вот так работает Redux: экшен отправляется при помощи dispatch ([*только не путайте с dispatcher*](https://redux.js.org/api/store#dispatchaction)) в стор, а редьюсер создаёт новый объект стора, чтобы разработчик получил к нему доступ и изменил представление (View)

При работе с Redux мы будем часто пользоваться тремя терминами: экшен, редьюсер и хранилище. Подробнее о них мы поговорим через несколько уроков, а пока скажем кратко.

**Экшен** (англ. action, «действие») — способ внести изменения в хранилище (store). Хранилище никогда не изменяется напрямую. Для этого и нужен экшен. Он представляет собой JavaScript-объект, который всегда должен иметь ключ type. Экшен также может содержать дополнительные данные.

Экшен отправляется (англ. dispatch) в хранилище с помощью метода store.dispatch. Для удобства работы с store.dispatch разработчики ввели новое понятие — генераторы экшенов (action creators). Они представляют собой функции, которые возвращают экшены. Часто путают термины action и action creator, но разницу легко запомнить:

Скопировать кодJSX

*// Это action*

{

type: 'FEEDBACK\_FORM\_SET\_VALUE',

text: 'Рейв был очень уютным и культурным. Спасибо организаторам! Приду ещё!'

}

*// А вот это генератор экшена*

function setFeedBackFormValue(text){

return {

type: 'FEEDBACK\_FORM\_SET\_VALUE',

text

}

}

**Редьюсеры.** Редьюсер (от англ. reduce, «свёртывать») — функция-обработчик, которая создаёт новый объект хранилища в зависимости от типа экшена и данных в нём. Редьюсер должен описать сам разработчик.

Экшены передаются на обработку редьюсеру внутри метода store.dispatch. Каждый раз редьюсер получает на вход новый экшен вместе с актуальным значением хранилища, а потом создаёт новый объект хранилища на основе предыдущего и возвращает его. Корневой редьюсер обычно состоит из нескольких дочерних редьюсеров, каждый из которых отвечает за обработку своего «участка» общего хранилища.

После окончания работы редьюсера новое значение, которое он вернул, заменяет текущее значение хранилища, и вызываются функции-обработчики. Предварительно эти функции нужно установить с помощью метода store.subscribe. Таким образом, разработчик получает доступ к новому значению хранилища и может произвести изменения в DOM.

**Иммутабельное хранилище.** Важнейшая концепция Redux — неизменяемость, или иммутабельность (англ. immutability), объекта хранилища. Это означает, что при любом изменении данных в хранилище вместо внесения изменений в существующий объект на его основе создаётся новая, изменённая копия. Чаще всего для этого используется оператор ..., который называется spread, и методы массивов типа map. Сравните два примера:

Скопировать кодJSX

const store = { title: 'Hello', name: 'Gregory' };

// Прямое изменение исходного объекта:

store.name = 'James';

// Создание нового объекта на основе исходного:

const newStore = { ...store, name: 'James' };

При таком подходе можно легко определить, изменились ли данные: достаточно сравнить newStore === store. Также можно вести историю изменений и, если нужно, «откатиться» к предыдущей версии. Это бывает полезно при отладке приложения.

Теперь, когда вы знаете основы работы Redux, установим его в проект.

# Подключение библиотеки Redux

В предыдущем уроке вы познакомились с основными принципами библиотеки Redux. Пора её подключить.

## Redux и React

Redux — универсальная библиотека и применяется не только в React. Например, многие разработчики используют её вместе с фреймворком Angular.

Но чтобы подключить Redux именно к React, потребуется установить в проект «соединительную» библиотеку react-redux. Вот как это сделать:

Скопировать кодBASH

*# Используя npm:*

npm install redux react-redux

После установки библиотеки нужно её настроить. Сделаем это в следующих уроках.

Библиотека react-redux предоставляет разработчикам специальный компонент Provider (почти такой же, как в React Context), HOC connect и набор хуков, которые позволяют удобно работать с Redux в функциональных компонентах. С помощью этого инструментария можно подключить хранилище к отдельным компонентам и использовать его данные: превращать их в пропсы и внутреннее состояние.

# Инициализация хранилища. Первый reducer и action

В предыдущем уроке мы научились подключать Redux. Теперь углубимся в терминологию Redux. Это поможет подготовиться к работе с Redux в реальных приложениях.

## Экшены

Экшен — объект с обязательным ключом type. Удобнее всего воспринимать экшены как события: нажата кнопка, загрузились данные, выполняется запрос, произошло изменение формы. В экшенах может содержаться дополнительная информация о таких событиях, которую нужно передавать в хранилище. Экшен — единственный источник информации для хранилища.

Отправить экшен можно с помощью метода store.dispatch(). Также для React существует несколько дополнительных способов отправки, но о них мы расскажем чуть позже.

В обязательном поле type указывают название экшена, события, которое произошло в приложении. Запись в ключе type ещё называют «тип экшена». Вот так выглядят типы экшенов:

Скопировать кодTSX

const TAB\_SELECTED = 'TAB\_SELECTED';

const USER\_LOGIN\_FAILED = 'USER\_LOGIN\_FAILED';

const CHANGES\_UNDO = 'CHANGES\_UNDO';

const CHANGES\_REDO = 'CHANGES\_REDO';

const TOGGLE\_THEME = 'TOGGLE\_THEME'

Тип экшена — всегда строка. Их принято записывать заглавными буквами — так эти переменные легко отличать от других переменных в коде приложения. Тип экшена используется и в редьюсере, который обрабатывает этот экшен. Посмотрите на экшен, который добавляет новую запись в список дел:

Скопировать кодTSX

{

type: ADD\_TODO,

text: 'Заменить шланг подачи холодной воды в бойлере',

expiresAt: '04.05.2021'

}

В этом примере у экшена есть дополнительная информация: содержание задачи и срок выполнения. Информация, которая описана в экшене, может принимать любую форму и сложность, главное требование — поле type. Экшены необязательно содержат дополнительную информацию, к примеру:

Скопировать кодTSX

{

type: SHOW\_SIDEBAR

}

Иногда для удобства использования и структурирования кода применяют специальные функции — генераторы экшенов (action creators). Так выглядит генератор экшена, который добавляет записи в список дел:

Скопировать кодTSX

*// Генератор экшена*

const addToDo = (text, expiresAt) => {

type: ADD\_TODO,

text: text,

expiresAt: expiresAt

}

Такую функцию можно использовать в коде приложения. Правда, с приходом хуков разработчики реже пользуются генераторами экшенов. О симбиозе хуков и Redux мы поговорим в следующем уроке, а пока — погрузимся в редьюсеры.

## Редьюсеры

Если экшены — события, то редьюсеры — обработчики этих событий. Экшены только описывают, что произошло, но не поясняют, как на это реагировать состоянию приложения. Редьюсеры же определяют, как изменится состояние приложения в ответ на экшены, отправленные в хранилище.

Редьюсер ведёт себя как чистая функция: принимает предыдущее состояние и возвращает новое. Каждый редьюсер принимает два аргумента: состояние и экшен. При описании редьюсеров можно задать состояние по умолчанию. Чтобы это сделать, нужно воспользоваться присваиванием в первом аргументе.

Взгляните, как может выглядеть редьюсер для списка дел:

Скопировать кодTSX

import { guid } from '../utils'

import { ADD\_TODO, TOGGLE\_TODO } from './constants'

*// Исходное состояние*

const initialState = [

{

id: guid(),

completed: false,

expiresAt: '08.04.20201',

text: 'Купить авокадо 4 шт.'

}

]

*// Редьюсер*

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

switch (action.type) {

*// Добавление новой задачи в список дел*

case ADD\_TODO:

return [

...state,

{

id: guid(),

text: action.text,

expiresAt: action.expiresAt,

completed: false

}

]

*// Изменение статуса задачи в списке дел*

case TOGGLE\_TODO:

return state.map(todo =>

todo.id === action.id ? { ...todo, completed: !todo.completed } : todo

)

*// Реакция на прочие типы экшенов*

default:

return state

}

}

Для описания редьюсеров удобнее всего использовать конструкции switch-case.

Обратите внимание на блок default — в нём мы просто возвращаем актуальное состояние. Это необходимо, потому что на каждый экшен Redux запускает все редьюсеры, и важно не потерять те части состояния, которые не были затронуты этим экшеном.

Также, чтобы при добавлении новой задачи не потерять текущий список дел, используется оператор .... А при изменении статуса этот оператор помогает обновить только один ключ в одном из элементов массива списка дел. Если дерево состояния вложенное, важно внимательно следить за всеми уровнями состояния в редьюсере и сохранять незатронутые части.

### Работа со множеством редьюсеров

В больших приложениях требуется разделять код хранилища на части. Мы уже говорили, что один из недостатков Redux — его «многословность». Код хранилища будет очень быстро разрастаться. Поэтому, согласитесь, switch-case из 20 тысяч строк — не лучшее решение. При этом для инициализации хранилища нужен один корневой редьюсер, который собирает все части состояния воедино.

Посмотрим, как всё это организовать. Вы уже знаете, что каждый редьюсер — чистая функция, а состояние — объект. Тогда напишем корневой редьюсер для списка дел и представим, что в приложении есть и другие части состояния:

Скопировать кодTSX

*// Редьюсер списка дел*

const todoList = (state, action) => { ... }

*// Редьюсер пользователя приложения*

const user = (state, action) => { ... }

*// Редьюсер коллективной работы над списком дел*

const collaboration = (state, action) => { ... }

*// Корневой редьюсер*

const rootReducer = (state, action) => {

todoList: todoList(state.todoList, action),

user: user(state.user, action),

collaboration: collaboration(state.collaboration, action)

}

Как вы могли убедиться, никакой магии в создании корневого редьюсера нет, это просто объект и редьюсеры, назначаемые ключами этого объекта. Хоть корневой редьюсер и выглядит не очень большим, дублирование кода может смущать. Если не нас, то разработчиков Redux точно. Они смутились и написали вспомогательную функцию combineReducers().

Попробуем переписать корневой редьюсер с применением этой функции:

Скопировать кодTSX

import { combineReducers } from 'redux';

*// Редьюсер списка дел*

const todoList = (state, action) => { ... }

*// Редьюсер пользователя приложения*

const user = (state, action) => { ... }

*// Редьюсер коллективной работы над списком дел*

const collaboration = (state, action) => { ... }

*// Корневой редьюсер*

const rootReducer = combineReducers({

todoList,

user,

collaboration

})

Теперь код выглядит чище, а работать с ним проще. Функция combineReducers() применяется не только для корневого редьюсера, но и для вложенных редьюсеров с иерархичным состоянием.

## Инициализация хранилища

Хранилище — объект, в котором хранится актуальное состояние приложения. В Redux-приложениях используется одно хранилище на всё приложение. Получить доступ к состоянию можно несколькими способами, например с помощью метода store.getState(). Данные из хранилища доступны только для чтения, а чтобы изменить состояние, нужно отправить экшен.

Чтобы инициализировать хранилище, воспользуйтесь функцией createStore(). Она принимает три аргумента:

1. Корневой редьюсер. Это обязательный аргумент.
2. Начальное состояние. Оно необязательно и может быть указанно частично.
3. Дополнения. Они расширяют функциональность хранилища. Это может быть, к примеру, логирование или сохранение в local storage. Мы ещё разберём этот аргумент подробнее в следующих уроках.

Самое простое хранилище с одним аргументом можно инициализировать так:

Скопировать кодTSX

import { createStore } from 'redux';

import { rootReducer } from './reducers';

const store = createStore(rootReducer);

## Доступ к состоянию хранилища. Компонент Provider

Этот компонент нужен для доступа из приложения к актуальному состоянию хранилища. «Под капотом» Provider использует React.Context. Но если контекст можно применять на любом уровне иерархии, то компонент Provider рекомендуется использовать только на верхнем уровне приложения. Так в хранилище можно обратиться из любой точки приложения, как бы глубоко в иерархии ни находился компонент. В качестве пропсов компонент Provider получает объект store, созданный с помощью известной вам функции createStore():

Скопировать кодJSX

import { createStore } from 'redux';

import { Provider } from 'react-redux';

import App from './components/app/app';

*// Корневой редьюсер, который обрабатывает экшены*

import { rootReducer } from './services/reducers';

*// Инициализируем хранилище с помощью корневого редьюсера*

const store = createStore(rootReducer);

ReactDOM.render(

*// Оборачиваем приложение компонентом Provider из пакета react-redux*

<Provider store={store}>

<App />

</Provider>,

document.getElementById('root')

);

Вот и всё! Теперь доступ к хранилищу Redux есть на любом уровне иерархии компонентов.

Вы научились создавать экшены, описывать редьюсеры, инициализировать хранилище Redux и получать доступ к его состоянию. Уже в следующем уроке начнём использовать этот инструментарий в React-приложении.

import {

DELETE\_ITEM,

CANCEL\_PROMO,

DECREASE\_ITEM,

INCREASE\_ITEM,

TAB\_SWITCH,

} from '../actions/cart';

import { recommendedItems, items } from '../initialData';

const initialState = {

items,

recommendedItems,

promoCode: '',

promoDiscount: null,

currentTab: 'items'

};

export const cartReducer = (state = initialState, action) => {

switch (action.type) {

case TAB\_SWITCH: {

return {

...state,

currentTab: state.currentTab === 'items' ? 'postponed' : 'items'

};

}

case INCREASE\_ITEM: {

return {

...state,

items: [...state.items].map(item =>

item.id === action.id ? { ...item, qty: ++item.qty } : item

)

};

}

case DECREASE\_ITEM: {

return {

...state,

items: [...state.items].map(item =>

item.id === action.id ? { ...item, qty: --item.qty } : item

)

};

}

case DELETE\_ITEM: {

return { ...state, items: [...state.items].filter(item => item.id !== action.id) };

}

case CANCEL\_PROMO: {

return {

...state,

promoCode: '',

promoDiscount: null

};

}

default: {

return state;

}

}

};

# Структура приложения Redux

В предыдущем спринте мы уже показывали, [как лучше структурировать код приложения](https://praktikum.yandex.ru/trainer/react/lesson/b0353706-8198-471d-9329-e5fcd2e136b2/). Сейчас предлагаем применить те знания уже с учётом использования Redux. Когда мы добавляем в приложение библиотеку Redux, его объём сильно возрастает. Появляются вопросы: где хранить весь этот код и как лучше его структурировать. Разберём две модели хранения кода в React-Redux приложениях:

* разделение на представление и хранилище,
* разделение по функциональности.

## Разделение на представление и хранилище

В этой модели мы полностью разделяем места хранения состояния и представления. В одной из директорий приложения, например components, собираем основные части интерфейса. А в другой директории — services — собираем бизнес-логику приложения: экшены и редьюсеры. При этом структуру экшенов и редьюсеров также разделяем по файлам. Разберём простой пример:

Скопировать код

└── src/

├── components/

│ ├── user-profile/

│ │ ├── user-profile.js

│ │ ├── user-profile.test.js

│ │ └── user-profile.module.css

│ ├── cart/

│ │ ├── cart.js

│ │ ├── cart.test.js

│ │ └── cart.module.css

│ └── favorites/

│ ├── favorites.js

│ ├── favorites.test.js

│ └── favorites.module.css

│

└── services/

├── actions/

│ ├── user-profile.js

│ ├── cart.js

│ └── favorites.js

└── reducers/

├── cart.js

├── user-profile.js

├── favorites.js

└── index.js

В этом примере хранилище структурно соответствует представлению. Например, компонент user-profile отображает информацию о пользователе: аватар, почту, пароль, адреса доставки. А экшены и редьюсеры user-profile.js описывают получение и изменение этой логики: обновление аватара, смену пароля, почты или удаление адреса доставки.

Такое разделение кода очень эффективно. Мы концентрируем всю Redux-логику в одном месте, и это облегчает взаимодействие с ней. К примеру, скоро вы узнаете про новую фичу Redux Toolkit. Если воспользуетесь этой библиотекой — код станет проще редактировать, ведь всё хранилище будет собрано в одном месте. Организация кода хранилища и его поддержка — главный плюс такого разделения.

Если в приложении появляется множество компонентов — плоская структура хранилища позволяет быстро находить экшены и редьюсеры, ответственные за работу представления.

## Разделение кода по функциональности

Мы уже говорили об этом подходе в предыдущем спринте. В нём мы группируем представление и хранилище в директории компонента. Возьмём предыдущий пример и разделим код по функциональности:

Скопировать код

└── src/

└── components/

├── user-profile/

│ ├── user-profile.js

│ ├── user-profile.test.js

│ ├── user-profile.module.css

│ └── services

│ ├── actions

│ │ └── user-profile.js

│ └── reducers

│ └── user-profile.js

├── cart/

│ ├── cart.js

│ ├── cart.test.js

│ ├── cart.module.css

│ └── services

│ ├── actions

│ │ └── cart.js

│ └── reducers

│ └── cart.js

└── favorites/

├── favorites.js

├── favorites.test.js

├── favorites.module.css

└── services

├── actions

│ └── favorites.js

└── reducers

└── favorites.js

Такая модель создаёт из каждой функциональности приложения отдельную самодостаточную и повторно используемую сущность. Мы объединяем все составляющие компонента в одной директории.

Когда разработчики объединяют код в одном месте, им интуитивно проще добраться до любой его части: доступ к хранилищу и представлению корзины можно получить в одной и той же директории. Но и эта модель несовершенна: компоненты могут расти и усложняться и может быть неясно, в какой момент стоит разделить один компонент на несколько новых. Например, со временем в корзине может появиться информация об оплате и нужно решить, оставлять эту логику в компоненте корзины или выносить в отдельный. Ведь функциональность оплаты может понадобиться и в других компонентах. Мы не подскажем, как понять, что пора декомпозировать компоненты, ведь это приходит с опытом, но разберём, как лучше это сделать.

## Рост представления в глубину. Контейнеры

Когда приходит время делить большой компонент на части, можно добавить файл-контейнер в директорию компонента. Для примера разделим user-profile.js на четыре файла: addresses.js, address.js, profile.js, payment-info.js. Включим в user-profile.js вложенную директорию components и файл-контейнер, который собирает все вложенные компоненты:

Скопировать код

└── src/

└── components/

├── user-profile/

│ ├── user-profile-container.js

│ ├── user-profile-container.module.css

│ ├── components

│ │ ├── addresses.js

│ │ ├── addresses.module.css

│ │ ├── address.js

│ │ ├── address.module.css

│ │ ├── profile.js

│ │ ├── profile.module.css

│ │ ├── payment-info.js

│ │ └── payment-info.module.css

│ └── services

│ ├── actions

│ │ └── user-profile.js

│ └── reducers

│ └── user-profile.js

├── cart/

│ ├── cart.js

│ ├── cart.test.js

│ ├── cart.module.css

│ └── services

│ ├── actions

│ │ └── cart.js

│ └── reducers

│ └── cart.js

└── favorites/

├── favorites.js

├── favorites.test.js

├── favorites.module.css

└── services

├── actions

│ └── favorites.js

└── reducers

└── favorites.js

Такой подход применим для обеих моделей взаимодействия представления и хранилища. Название файла user-profile-container условно: он может называться и index.js, и user-profile.js, но важно придерживаться единого именования.

В этом уроке вы познакомились с двумя способами организации приложения. В рамках работы над проектом вы можете придерживаться любого из подходов. Мы рекомендуем использовать модель с разделением на представление и хранилище: с вашим кодом взаимодействуют и наставник, и ревьюер. А в код с такой структурой будет легче вникнуть.

Пора двигаться к следующему уроку.

# Взаимодействие React c Redux

В предыдущих уроках мы рассказали, как инициализировать хранилище, писать редьюсеры и экшены, а также структурировать Redux-логику. Теперь пора применить всё это в React-приложении.

Вы уже знаете, как работать с React.Context. Механизм применения контекста и Redux очень схож: в коде есть верхнеуровневая «обёртка» и дополнительные функции. Такие функции позволяют получить доступ к данным на любом уровне иерархии компонентов без глубокого проброса пропсов (props drilling). Но обо всём по порядку. Для начала разберёмся с обёрткой.

## Доступ к хранилищу. Функция connect()

Функция connect() — не что иное, как HOC. Её применяют для связывания React-компонента с Redux-хранилищем. Эту функцию используют в классовых компонентах, но её можно применять и в функциональных.

У функции connect() есть два основных аргумента:

1. Первый аргумент — функция mapStateToProps. Её задача — уточнять ключи хранилища, которые передаются в пропсы компонента. Звучит сложно, но на деле всё выглядит совсем безобидно:

Скопировать кодJSX

*// Аргумент функции - объект хранилища Redux.*

const mapStateToProps = store => {

return {

step: store.step,

theme: store.appearance.theme,

totalPrice: store.cart.items.reduce((acc, item)=> acc + item.price, 0),

user: store.account.userData

};

};

Иногда нужно вычислять пропсы, передаваемые в компонент, из хранилища Redux на основании собственных пропсов компонента. Для этого можно воспользоваться вторым аргументом функции mapStateToProps — ownProps:

Скопировать кодJSX

*// Используем два аргумента: хранилище Redux и собственные пропсы компонента*

const mapStateToProps = (store, ownProps) => {

return {

step: store.step,

*// Вычисляемое значение пропса на основании собственных пропсов*

theme: ownProps.theme ? ownProps.theme : store.appearance.theme,

totalPrice: store.cart.items.reduce((acc, item)=> acc + item.price, 0),

user: store.account.userData

};

};

Важно уточнить: собственные пропсы компонента будут по-прежнему доступны в самом компоненте, просто функция mapStateToProps сможет прочитать их из второго аргумента.

1. Второй аргумент — функция mapDispatchToProps. Она передаёт в пропсы компонента колбеки, которые отправляют экшены в Redux. Первый аргумент функции mapDispatchToProps — уже знакомый вам метод dispatch(). Как и в функции mapStateToProps, собственные пропсы компонента доступны через второй аргумент ownProps:

Скопировать кодJSX

*// Фунция dispatch доступна в качестве первого аргумента,*

*// собственные пропсы оборачиваемого компонента передаются вторым аргументом*

const mapDispatchToProps = (dispatch, ownProps) => {

return {

*// В пропсах компонента доступны три функции*

toggleTodo: () => dispatch(toggleTodo(ownProps.todoId)),

addTodo: (text) => dispatch(addToDo(text)),

changeLayout: () => dispatch({ type: 'CHANGE\_LAYOUT' })

};

};

В пакете redux есть вспомогательная функция — bindActionCreators(actionCreators, dispatch). Представим, что у нас есть такие экшены:

Скопировать кодJSX

*// ./services/actions.js*

export function addTodo(text) {

return {

type: 'ADD\_TODO',

text

}

}

export function removeTodo(id) {

return {

type: 'REMOVE\_TODO',

id

}

}

Вот так выглядит функция mapDispatchToProps без bindActionCreators:

Скопировать кодJSX

import { addTodo, removeToDo } from './services/actions';

const mapDispatchToProps = dispatch => {

return {

addTodo: (text) => dispatch(addToDo(text)),

removeToDo: (id) => dispatch(removeTodo(id))

};

};

А так — с использованием bindActionCreators:

Скопировать кодJSX

import \* as TodoActionCreators from './services/actions'

const mapDispatchToProps = dispatch =>

bindActionCreators(TodoActionCreators, dispatch);

По сути bindActionCreators — синтаксический сахар, который позволяет писать чуть меньше кода.

Но существует ещё более ёмкий способ передать функции отправки экшенов в пропсы компонента:

Скопировать кодJSX

import { addTodo, removeToDo } from './services/actions';

const mapDispatchToProps = {

addTodo,

removeToDo

}

Любой из вариантов применим, но если вам нужен доступ к собственным пропсам компонента в mapDispatchToProps, то использование dispatch — наиболее подходящее решение.

Функции mapStateToProps и mapDispatchToProps — основные аргументы connect(). Но у функции connect() есть ещё два аргумента, работа которых связана с основными. Третий и четвёртый аргументы используются редко, поэтому мы не будем подробнее на них останавливаться. Вы можете самостоятельно ознакомиться с ними [в официальной документации](https://react-redux.js.org/api/connect).

А пока — закрепим использование функции connect() на примере:

Скопировать кодJSX

import { connect } from 'react-redux';

function App(props) {

*// ...*

}

const mapDispatchToProps = (dispatch, ownProps) => {

return {

toggleTodo: () => dispatch(toggleTodo(ownProps.todoId)),

addTodo: (text) => dispatch(addToDo(text)),

changeLayout: () => dispatch({ type: 'CHANGE\_LAYOUT' })

};

};

const mapStateToProps = (store, ownProps) => {

return {

todos: store.todos,

theme: ownProps.theme ? ownProps.theme : store.appearance.theme,

user: store.account.userData

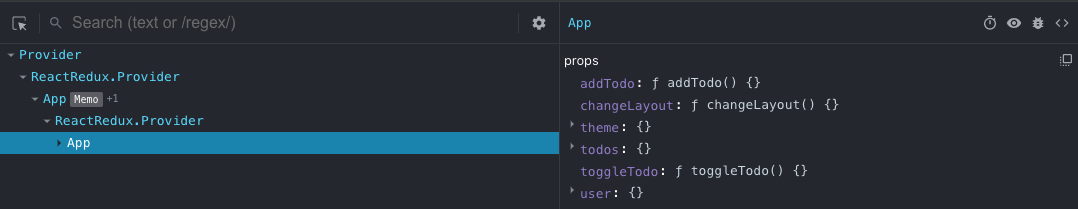
};

};

*// Собираем всё вместе, передав два аргумента в connect*

export default connect(mapStateToProps, mapDispatchToProps)(App);

Компонент, обёрнутый HOC, получает дополнительные пропсы: состояние и функции отправки экшенов. Так в консоли React Developer Tools можно увидеть, что у компонента App теперь есть дополнительные пропсы:



Необязательно передавать оба аргумента в функцию connect(), но если оставить её совсем без аргументов, изменение данных в хранилище не будет вызывать повторный рендеринг, а в пропсах появится дополнительная функция — dispatch(). Эта функция доступна и в случае, если передан только первый аргумент — mapStateToProps. Вы сможете использовать dispatch() из пропсов напрямую в компоненте:

Скопировать кодJSX

class App extends React.Component {

constructor(props) {

super(props);

this.state = { step: 'delivery' }

}

componentDidMount() {

this.props.dispatch({type: 'GET\_DELIVERY\_OPTIONS' });

}

render() {

*// ...*

}

}

export default connect()(App);

Если вам показалось, что connect() — универсальный комбайн, который умеет всё, то вам не показалось. Но освоение такой функции требует сноровки. Правда, с появлением хуков в функциональных компонентах работа с Redux стала удобнее. Рассмотрим, как это повлияло на получение данных из хранилища и отправку экшенов.

## Чтение данных из хранилища. Хук useSelector()

Хук useSelector() — перерождение mapStateToProps из функции connect(). Разберём, как с помощью этого хука «вытащить» кусок состояния в компонент:

Скопировать кодJSX

import { useSelector } from 'react-redux';

function App(props) {

*// Используем хук для получения данных из хранилища*

*// По умолчанию из замыкания есть доступ к собственным пропсам компонента*

const { todos, theme, user } = useSelector(store => ({

todos: store.todos,

theme: props.theme ? props.theme : store.appearance.theme,

user: store.account.userData

}))

*// ...*

}

*// Больше не нужен connect*

export default App;

В этом примере мы использовали хук useSelector для чтения данных из хранилища. Такой компонент можно без труда перемещать вверх и вниз в иерархии компонентов и не волноваться за дополнительные зависимости между компонентами и определённую структуру. Такой компонент — самодостаточная «деталь», которую можно использовать в любом месте приложения.

Первый аргумент хука useSelector — функция вида (store) => store. Не возвращайте всё хранилище целиком, это может сильно ухудшить производительность, ведь любое изменение данных в хранилище будет вызывать дополнительный рендеринг. Старайтесь вычленять из хранилища только те значения, которые важны для конкретного компонента. То же самое справедливо и для функции mapStateToProps.

Второй аргумент хука — функция-компаратор, иначе — функция, которая проверяет равенство двух объектов. Это опциональный аргумент. Если ваше приложение зависит от фактического изменения (или неизменности) конкретных ключей из хранилища, вы можете воспользоваться этим аргументом:

Скопировать кодJSX

import { useSelector, shallowEqual } from 'react-redux';

function App(props) {

*// Используем хук для получения данных из хранилища*

*// По умолчнию из замыкания есть доступ к собственным пропсам компонента*

const { todos, theme, user } = useSelector(store => ({

todos: store.todos,

theme: props.theme ? props.theme : store.appearance.theme,

user: store.account.userData

*// Передаём функцию-компаратор вторым аргументом*

}), shallowEqual)

*// ...*

}

*// Больше не нужен connect*

export default App;

В этом случае мы использовали функцию shallowEqual, которая идёт в пакете react-redux. С её помощью мы реализовали неглубокое сравнивание двух объектов. Если потребуется, вы можете написать собственную реализацию функции-компаратора. Для этого используйте такую сигнатуру функции: (objA, objB) => boolean.

Теперь вы знаете, как с помощью хука useSelector читать данные из хранилища в функциональных компонентах. Осталось разобраться, как отправлять экшены.

### Отправка экшенов. Хук useDispatch()

При использовании connect() без второго аргумента в пропсы компонента попадает метод dispatch(), который позволяет отправлять экшены. Но ещё это можно сделать с помощью хука useDispatch(). Рассмотрим на примере:

Скопировать кодJSX

import { useDispatch } from 'react-redux';

import { useState } from 'react';

import { addTodo, removeToDo } from './services/actions';

function App(props) {

*// Функция dispatch теперь доступна из хука внутри компонента*

const dispatch = useDispatch();

const [text, setText] = useState('');

const onSubmit = () => {

*// Отправляем экшен, используя переменную из хука React.useState*

dispatch(addTodo(text))

}

const onDelete = (id) => {

*// Отправляем экшен*

dispatch(removeToDo(id))

}

const

*// ...*

}

export default App;

Хук useDispatch() можно использовать и так:

Скопировать кодJSX

import { useDispatch } from 'react-redux';

import { useEffect } from 'react';

const App = (props) => {

const dispatch = useDispatch();

useEffect(() => {

*// Отправляем экшен при монтировании компонента*

dispatch({type: 'GET\_DELIVERY\_OPTIONS' });

}, [])

*// ...*

}

export default App;

Только что вы могли в очередной раз убедиться в силе хуков. Ни в одном из примеров с хуками мы не использовали функцию connect(). Хотя это и не обязательное требование, но кажется, что connect() при наличии хуков — слишком многословное решение.

### Заключение

В этом уроке мы рассмотрели функцию connect(), которая связывает React-компоненты с Redux-хранилищем, и два её основных аргумента. Кроме этого, мы познакомились с функцией dispatch(), которая позволяет отправлять экшены. Использование всех этих функций уместно в современных React-приложениях, но с хуками useSelector() и useDispatch() в этом нет нужды. Они применяются в функциональных компонентах и отвечают за чтение данных из хранилища и отправку экшенов.

Это объёмный и непростой урок. Здорово, если он поддался вам с первого раза. Теперь нужно закрепить пройденный материал — с этим помогут задания. Увидимся в следующем уроке!

# Отладка: Redux Devtools

Вы научились работать с Redux в React и вынесли всю бизнес-логику приложения в хранилище. Это удобное структурирование кода, благодаря которому работать с приложением становится проще.

Разработчик всегда проверяет написанный код ещё и в браузере — только так в первое время можно отловить большинство неочевидных багов и неправильное поведение, которые не удалось увидеть в самом коде. Все мы — люди и пишем приложения для людей, поэтому без тестирования функциональности приложения в браузере далеко не уехать. Написание кода приложения без проверки его в браузере больше похоже на соревнование [Code in the Dark](http://codeinthedark.com/), а не на реальную продуктовую разработку.

В предыдущем спринте вы узнали [про расширение React Devtools](https://praktikum.yandex.ru/trainer/react/lesson/4f4c20c3-7fa1-414b-8771-7f8dbb01c803/task/f4e714df-13ed-4d74-b238-91d3b67d378c/) и научились пользоваться им для взаимодействия с React-кодом в браузере. Но сейчас часть логики (причём самой важной) приложения мы вынесли в Redux-хранилище. React Devtools не подходит для работы с Redux-хранилищем. В этом уроке мы расскажем про специальное расширение Redux Devtools и попробуем с ним поработать.

Redux DevTools — мощное расширение для отладки изменений в хранилище. Самый простой способ установки расширения — скачать его с маркета вашего браузера:

* [ссылка на расширение для Chrome](https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd),
* [ссылка на расширение для Firefox](https://addons.mozilla.org/en-US/firefox/addon/reduxdevtools/).

Существуют и другие способы установки расширения, о них вы можете почитать [на этом сайте](https://extension.remotedev.io/).

## Настраиваем расширение Redux DevTools

После установки расширения его иконка появится в панели расширений вашего браузера. Несмотря на то, что вы используете Redux, иконка останется серой и ничего не произойдёт. Дело в том, что хранилище Redux может быть устроено по-разному, поэтому расширению нужно дать доступ к вашему коду и «познакомить» их. Redux по умолчанию предоставляет такую возможность в виде расширителей хранилища.

Подробнее про расширители хранилища мы расскажем в следующем уроке, а пока поверьте нам, что ничего очень страшного расширитель из себя не представляет — он всего лишь позволяет «пропустить» через себя весь Redux-код. Так и происходит знакомство Redux DevTools и Redux хранилища в приложении.

Для знакомства нашего кода с расширением добавьте в корневой файл приложения следующий код:

Скопировать кодJSX

import { compose, createStore } from 'redux';

const composeEnhancers =

typeof window === 'object' && window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_

? window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_({})

: compose;

Мы проверяем, что расширение Redux DevTools есть, а глобальный window вообще существует — это важно, например при SSR. Если всё хорошо, мы используем расширение. В противном случае — вызовем функцию compose. По своей задаче обе функции примерно равны, но window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_({}) позволяет использовать расширение. Функция compose просто последовательно применяет расширители хранилища, если они есть. Опять же, про расширители мы поговорим в следующем уроке.

После вызова расширения нужно применить его к хранилищу. Для этого используется второй аргумент createStore:

Скопировать кодJSX

import { compose, createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

const enhancer = composeEnhancers(applyMiddleware(thunk));

const store = createStore(rootReducer, enhancer);

В коде много нового, в том числе и пакет redux-thunk. Пока просто добавьте этот код, и Redux DevTools на локальном сервере станет активным — окрасится в зелёный цвет. Что именно делает этот код, мы расскажем в следующем уроке.

Можно было бы запустить расширение и без выражения applyMiddleware(thunk) в коде выше, но мы указали его сразу: теперь, когда пройдёте следующий урок, код не придётся переписывать.

В целом это весь код, который необходимо написать, чтобы расширение заработало. Двигаемся дальше.

## Когда использовать расширение

Выше мы говорили, что Redux DevTools помогает тестировать написанную бизнес-логику прямо в приложении. Это удобно и необходимо во время режима разработки. Но цикл жизни приложения состоит не только из режима разработки: после приложение попадает к конечному пользователю — это называется режимом production. И когда это происходит, приложение требуется дорабатывать, ведь разработчику и тестировщикам удаётся найти не все баги интерфейса и функциональности. Разберём особенности использования Redux DevTools в продуктовой версии приложения:

1. Расширение помогает легко отслеживать воспроизведённые пользовательские баги в работе приложения. Это очень удобно. К тому же, не все баги в продуктовой версии возможно воспроизвести в режиме разработки.
2. Расширение предоставляет доступ к данным приложения: уникальные \_id и другие данные вашего приложения сможет посмотреть любой пользователь, который знает Redux и умеет работать с Redux DevTools. Это не очень хорошо, потому что бизнес-логика приложения на то и бизнес, чтобы оставаться скрытой от глаз пользователей. Мы отметим лишь то, что в последних версиях расширения появилась возможность создавать «чёрные списки» экшенов, которые не будут показываться в расширении. Опции вызова Redux DevTools выходят за рамки текущего курса. Подробнее ознакомиться с ними вы сможете [в документации](https://github.com/zalmoxisus/redux-devtools-extension/blob/master/docs/API/Arguments.md).
3. Расширение «накапливает» в себе экшены по мере работы с приложением. Это может приводить к проблемам с производительностью у пользователей с установленным расширением. Разработчики расширения также предоставляют набор опций для контроля количества архивируемых экшенов и частоты их сохранения.

Таким образом, использование расширения в продуктовой версии приложения может быть очень полезным и безболезненным инструментом при тонкой настройке. Применять его в продуктовой версии или нет — решают разработчики. Вы можете попутешествовать по сети и посмотреть, какие проекты используют React и предоставляют доступ к Redux DevTools. Например, на проекте Яндекса [auto.ru](http://auto.ru/) вы увидите заветный зелёный кружок у расширения.

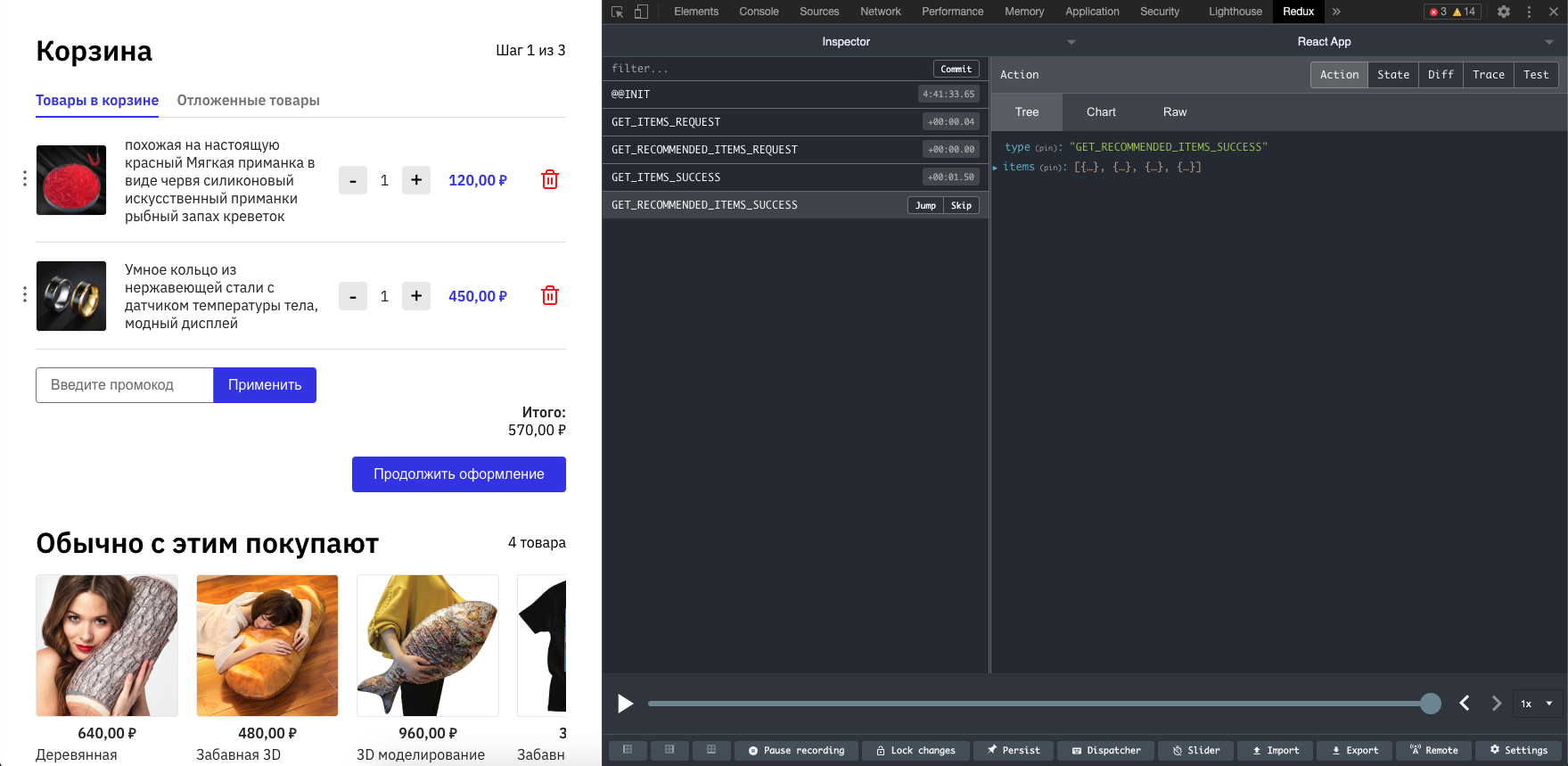
## Функциональность Redux DevTools

Мы уже многое сказали об этом расширении, но вы ещё не видели его в действии — пора это исправить. Разберём функциональность расширения на корзине, которую вы дорабатываете в заданиях тренажёра. Вот что умеет Redux DevTools:

* хранить все отправленные экшены,
* предоставлять доступ к деталям каждого отправленного экшена,
* давать доступ к описанию самого экшена — его типу и отправленным данным,
* давать доступ к описанию изменения хранилища в разных визуализациях,
* давать доступ к сравнению состояния после обработки экшена,
* позволять «версионировать» экшены — откатывать приложение к моменту обработки конкретного экшена.

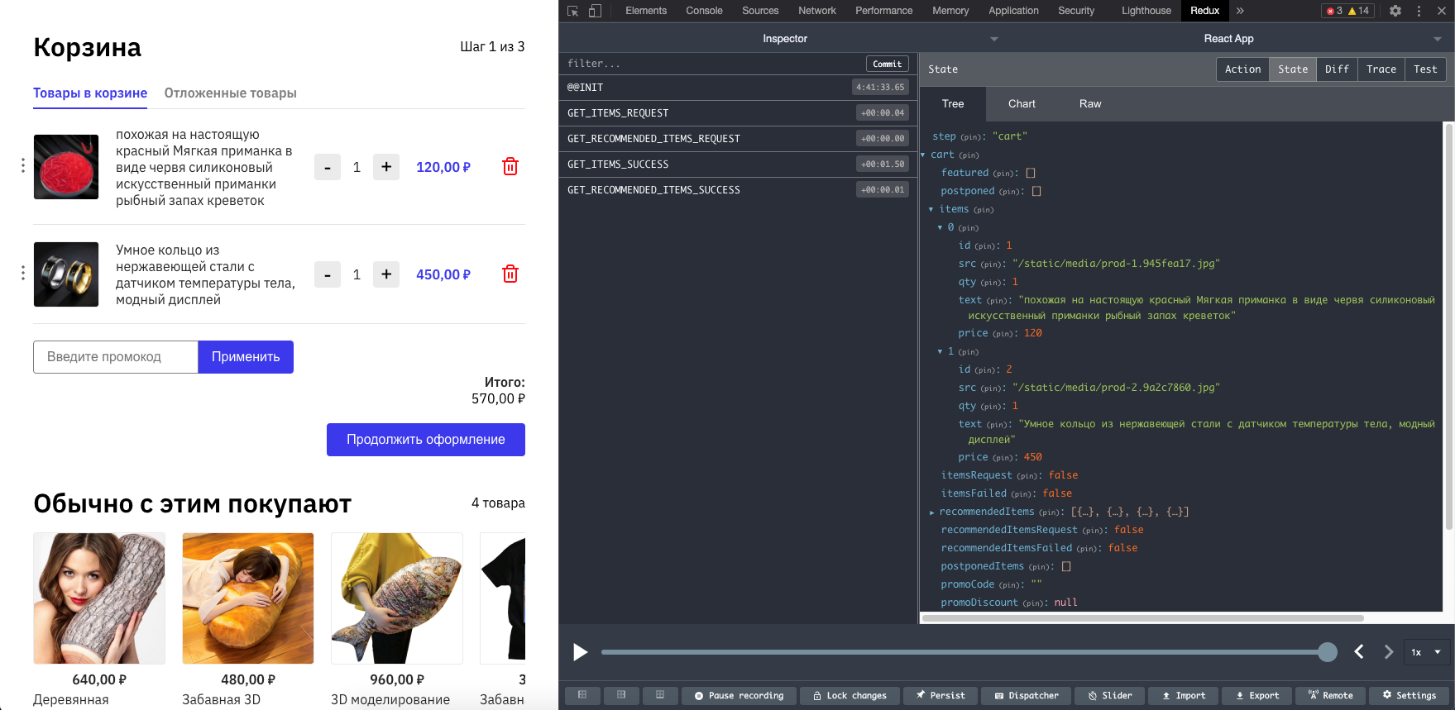
Само расширение во вкладке Inspector отображает в левой панели все произошедшие экшены, а в правой панели — текущий переключатель.

В переключателе Action отображается информация о выбранном в левой панели экшене и отправленных в него данных.

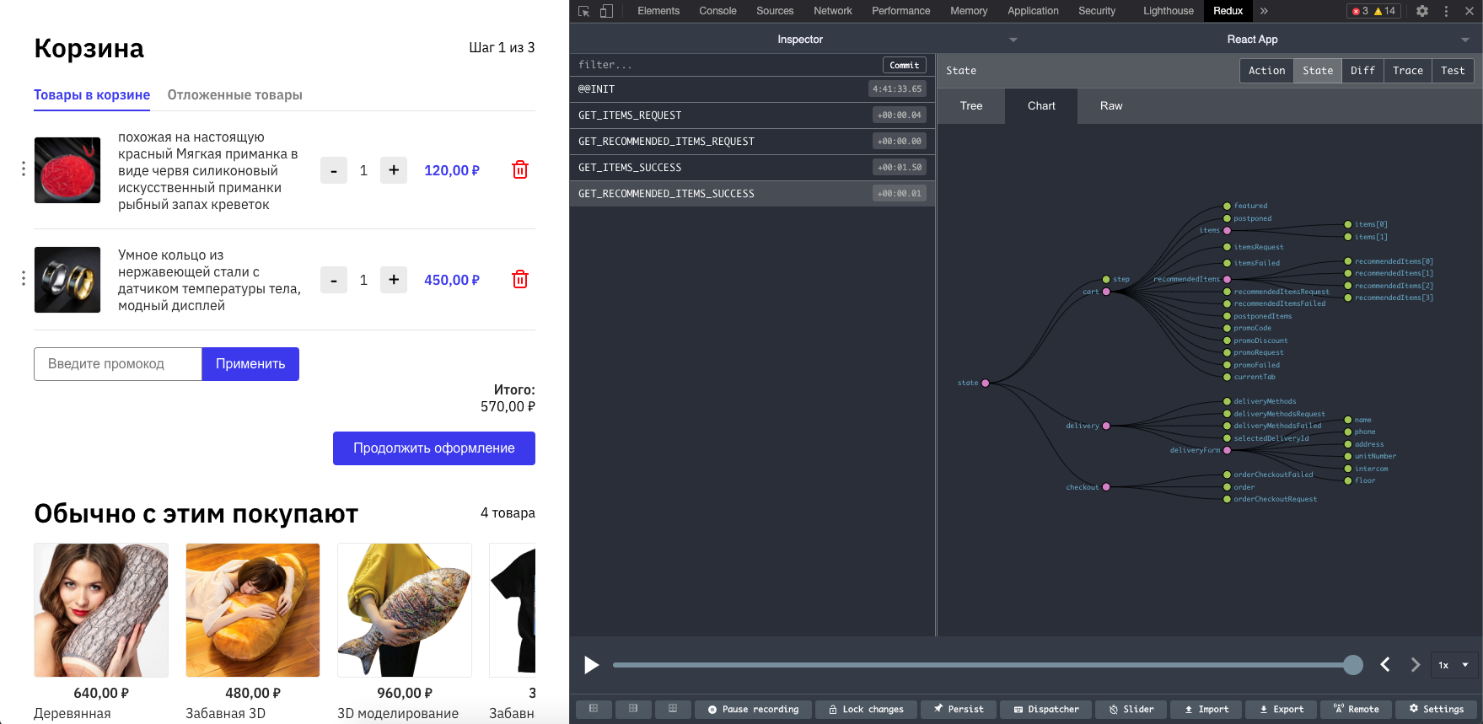


Отображение содержимого вкладки Action

Переключатель State отображает информацию обо всём хранилище, вкладка Tree отображает состояние в виде объектов (что не всегда удобно), а вкладка Chart — в виде древа, в котором подсвечены все обновлённые элементы хранилища.



Содержимое вкладки Tree в State

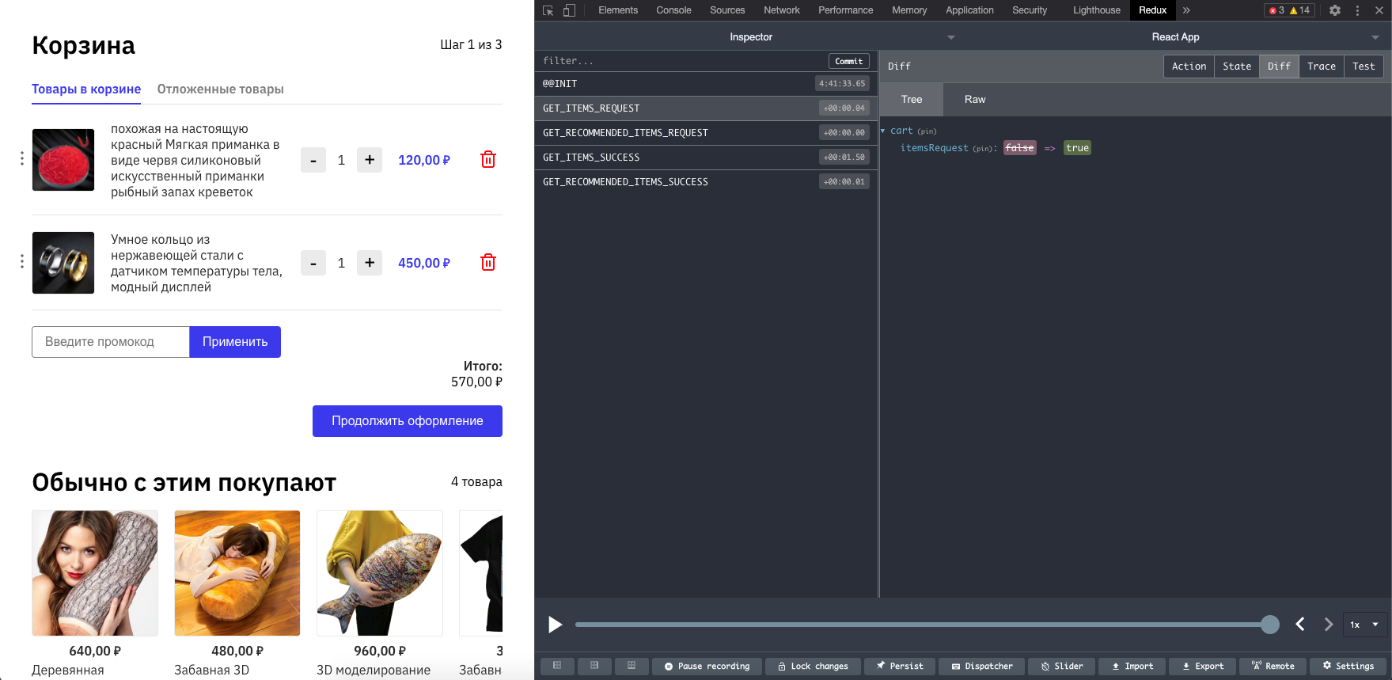


Содержимое вкладки Chart в State

По мере изменения хранилища древо также будет расширяться и добавлять новые элементы в хранилище, если они имеются. Например, после получения товаров в корзине они отобразятся в древе на конкретном экшене GET\_ITEMS\_SUCCESS:

Переключаясь между экшенами, древо хранилища также модифицируется.

Переключатель Diff показывает изменения, обработанные после отправки конкретного экшена.



Отображаемое содержимое вкладки Diff.

Попробуем побегать по всем вкладкам Redux DevTools:

Согласитесь — это удобнее console.log в редьюсерах.

При этом по ходу взаимодействия с приложением история экшенов пополняется моментально, по мере их возникновения и обработки редьюсером. Попробуем ввести какие-нибудь данные в одной из форм — и сразу получим изменения хранилища в Redux DevTools:

Каждое нажатие порождает новый экшен.

Функциональность расширения поражает. Но и это ещё не всё: вы можете не дожидаться обработки какого-либо экшена или кликать по интерфейсу. Вы можете воспроизвести любой тип экшена в приложении самостоятельно, открыв панель ввода (центральная иконка на нижней панели). Отправим экшен NEXT\_STEP для перехода к следующему этапу оформления заказа:

## Путешествие во времени с помощью Redux Devtools

С помощью панели таймлайна в самом низу вы можете перемещаться между обработкой конкретных экшенов. Более того, представление приложения также откатится к состоянию на момент обработки выбранного экшена.

«Хоббит, или туда и обратно», но только в истории приложения.

Обратите внимание, что в приложении может происходить отправка нескольких экшенов. Например, когда мы делаем запросы к серверу. Поэтому, откатываясь в истории, вы будете порождать новые экшены:

Переходя к моменту отправки запроса, например за списком товаров — мы отправляем экшен по их добавлению в хранилище.

Это основы работы с Redux DevTools, которые будут вам полезны, в том числе и при выполнении проектной работы. Это удобное и легко настраиваемое расширение, которое позволяет отдельно работать с бизнес-логикой приложения. В следующем уроке мы подробнее рассмотрим расширители хранилища и работу с API в связке с Redux.

# Сайд-эффекты в Redux. Библиотека redux-thunk

В предыдущих уроках вы познакомились с библиотекой Redux и поняли, как использовать её в связке с React. Порой в приложениях нужно работать с побочными эффектами: логированием, взаимодействием с API, роутингом и другими. В этом уроке мы расскажем, как работать с побочными эффектами в Redux.

## Концепция middleware в Redux

Библиотека Redux обладает потрясающей возможностью: она позволяет использовать сторонний программный код (middleware, усилители). Такой код может влиять на отправку экшенов, а также управлять тем, когда и как эти экшены достигнут редьюсеров. Применение усилителей позволяет расширять возможности библиотеки, не меняя её исходный код.

Для начала разберёмся с тем, как выглядит усилитель:

Скопировать кодJSX

*// ES6*

const sayHiMiddleWare = store => next => action => {

console.log('Привет!');

return next(action);

};

*// ES5*

function sayHiMiddleWare(store) {

return function (next) {

return function (action) {

console.log('Привет!');

return next(action);

};

};

};

Такой усилитель будет выводить в консоль "Привет!" на каждый экшен.

Суть усилителей — взять входные данные, добавить что-то или произвести какие-то побочные эффекты, а затем передать эти данные дальше.

Чтобы разобраться с этим, представьте конвейер на производстве одного популярного газированного напитка. На конвейере работают два робота: один разливает напиток в жестяные банки, а второй их запечатывает. Руководство предприятия принимает авантюристское решение сделать напитки с ванильным вкусом. На конвейер добавляют усилитель (middleware) — робота, который располагается между первым и вторым. Он добавляет щепотку ванилина в разлитый напиток и передаёт банки дальше — третьему роботу. Благодаря усилителю с конвейера выходит напиток, который разделил людей на два лагеря (но это уже совсем другая история). Схожим образом работают и усилители в Redux.

### Подключение middleware в Redux

Чтобы подключить усилитель в Redux, нужно разобраться с такими понятиями:

1. Генератор хранилища — функция, которая создаёт Redux-хранилище. Стоит различать генератор хранилища createStore(), экспортируемый из пакета redux, и генераторы, которые возвращаются из расширителей хранилища.
2. Расширитель хранилища (store enhancer) — функция высшего порядка, которая возвращает новый, расширенный генератор хранилища.

Для применения расширителя хранилища нам потребуется детальнее разобраться с аргументами функции createStore(). Как вы помните [из урока про инициализацию хранилища](https://praktikum.yandex.ru/trainer/react/lesson/c03bd8f1-6a0e-4028-b607-12b61f3a4616/task/7ff5536a-6b04-4387-b416-f512903fec64/), в качестве первого аргумента эта функция принимает корневой редьюсер, но у неё есть второй и третий аргументы. Если использовать только два, вторым аргументом можно передать расширитель хранилища:

Скопировать кодJSX

const store = createStore(rootReducer, enchancer)

Если же передавать три аргумента, вторым нужно передать предварительное состояние хранилища, а третьим — расширители:

Скопировать кодJSX

const store = createStore(rootReducer, { theme: 'dark' }, enchancer)

Рассмотрим первый вариант с двумя аргументами. На практике крайне редко встречаются задачи, для решения которых нужно писать собственные расширители хранилища. К счастью, разработчики Redux позаботились об этом и написали функцию applyMiddleware() — она и есть расширитель хранилища. С её помощью мы сможем подключать усилители к Redux. Напишем усилитель для логирования экшенов и применим его с помощью applyMiddleware():

Скопировать кодJSX

import { createStore, applyMiddleware } from 'redux';

import { rootReducer } from './services/reducers';

*// Наш усилитель*

const actionLogger = store => next => action => {

*// Выводим в консоль время события и его содержание*

console.log(`${new Date().getTime()} | Action: ${JSON.stringify(action)}` );

*// Передаём событие «по конвейеру» дальше*

return next(action);

};

*// Расширитель хранилища принимает в качестве аргумента усилитель*

const enhancer = applyMiddleware(actionLogger);

*// Инициализируем хранилище, использовав расширитель*

const store = createStore(rootReducer, enhancer);

Теперь каждый экшен будет выведен в консоль вместе со временем, соответствующим моменту вызова этого экшена.

Усилителей может быть несколько, функция applyMiddleware() принимает любое количество аргументов и применяет их последовательно:

Скопировать кодJSX

import { createStore, applyMiddleware } from 'redux';

import { rootReducer } from './services/reducers';

*// Усилитель 1*

const actionLogger = store => next => action => {

console.log(`${new Date().getTime()} | Action: ${JSON.stringify(action)}` );

return next(action);

};

*// Усилитель 2*

const errorLogger = store => next => action => {

if (action.type === 'SOMETHING\_FAILED') {

console.error(`Произошла ошибка: ${JSON.stringify(action)}`)

}

return next(action);

};

*// Расширитель хранилища принимает несколько усилителей одновременно*

const enhancer = applyMiddleware(actionLogger, errorLogger);

*// Инициализируем хранилище, использовав расширитель*

const store = createStore(rootReducer, enhancer);

Теперь каждый экшен будет залогирован, а если тип экшена равен SOMETHING\_FAILED, будет выведено сообщение об ошибке. Порядок выполнения усилителей таков:

1. Где-то в приложении вызывается метод store.dispatch() и передаёт экшен в actionLogger.
2. Когда actionLogger вызывает next(action), экшен переходит errorLogger.
3. Когда errorLogger вызывает next(action), экшен переходит в хранилище и выполняются редьюсеры.

Усилители часто применяют для работы с асинхронными экшенами, такими как запрос данных с сервера или подписка на сокеты.

С основами разобрались, теперь рассмотрим один из самых популярных усилителей для Redux — библиотеку redux-thunk.

## Усилитель Redux: библиотека redux-thunk

Как вы знаете, экшены в Redux — просто объекты вида:

Скопировать кодJSX

const someAction = {

type: 'DO\_SOME\_WORK',

// ...дополнительные данные

}

Но в реальном мире простого объекта бывает недостаточно, ведь если говорить про взаимодействие с API, то экшен — функция, которая возвращает Promise. У такой функции должна быть возможность отправлять экшены на каждом этапе жизни промиса. Тут-то и приходит на помощь одна из самых популярных библиотек — redux-thunk. Благодаря возможности добавлять усилители в Redux, redux-thunk позволяет использовать функции (и асинхронные тоже) в качестве экшенов.

Подключим усилитель к хранилищу:

Скопировать кодJSX

import { createStore, applyMiddleware } from 'redux';

import thunk from 'redux-thunk';

import rootReducer from './services/reducers';

const store = createStore(rootReducer, applyMiddleware(thunk));

Вот и всё! Теперь мы можем передавать в store.dispatch() не только объекты, но и функции, которые внутри замыкания имеют доступ к методам store.dispatch() и store.getState(). Звучит сложно, но мы позволим себе этот беспечный поступок и покажем вам исходный код библиотеки redux-thunk, чтобы вы могли убедиться, насколько сильна и одновременно проста концепция усилителей. Вот он, исходный код библиотеки с более чем 16 тысячами звёзд на гитхабе:

Скопировать кодJSX

function createThunkMiddleware(extraArgument) {

return ({ dispatch, getState }) => (next) => (action) => {

if (typeof action === 'function') {

return action(dispatch, getState, extraArgument);

}

return next(action);

};

}

const thunk = createThunkMiddleware();

thunk.withExtraArgument = createThunkMiddleware;

export default thunk;

Нет, мы не опечатались и не удалили случайно 20 страниц кода. Это действительно всё, что нужно для использования функции в качестве аргументов store.dispatch().

Напишем асинхронный экшен и вызовем его. Будем запрашивать данные с сервера и отправлять экшены по мере выполнения запроса:

Скопировать кодJSX

import {

GET\_FEED,

GET\_FEED\_FAILED,

GET\_FEED\_SUCCESS

} from './constants';

*// Наш первый thunk*

export function getFeed() {

*// Воспользуемся первым аргументом из усилителя redux-thunk - dispatch*

return function(dispatch) {

*// Проставим флаг в хранилище о том, что мы начали выполнять запрос*

*// Это нужно, чтоб отрисовать в интерфейсе лоудер или заблокировать*

*// ввод на время выполнения запроса*

dispatch({

type: GET\_FEED

})

*// Запрашиваем данные у сервера*

fetch('/feed').then( res => {

if (res && res.success) {

*// В случае успешного получения данных вызываем экшен*

*// для записи полученных данных в хранилище*

dispatch({

type: GET\_FEED\_SUCCESS,

feed: res.data

})

} else {

*// Если произошла ошибка, отправляем соотвтествующий экшен*

dispatch({

type: GET\_FEED\_FAILED

})

}

}).catch( err => {

*// Если сервер не вернул данных, также отправляем экшен об ошибке*

dispatch({

type: GET\_FEED\_FAILED

})

})

}

}

С применением redux-thunk мы можем передать функцию getFeed() в store.dispatch(). Но прежде чем мы напишем компонент, который вызовет такой экшен, посмотрим на редьюсер:

Скопировать кодJSX

import {

GET\_FEED,

GET\_FEED\_FAILED,

GET\_FEED\_SUCCESS

} from './constants';

const initialState = {

feedRequest: false,

feedFailed: false,

feed: []

}

export const feedReducer = (state = initialState, action) => {

switch (action.type) {

case GET\_FEED: {

return {

...state,

*// Запрос начал выполняться*

feedRequest: true,

*// Сбрасываем статус наличия ошибок от предыдущего запроса*

*// на случай, если он был и завершился с ошибкой*

feedFailed: false,

};

}

case GET\_FEED\_SUCCESS: {

return {

...state,

*// Запрос выполнился успешно, помещаем полученные данные в хранилище*

feed: action.feed,

*// Запрос закончил своё выполнение*

feedRequest: false

};

}

case GET\_FEED\_FAILED: {

return {

...state,

*// Запрос выполнился с ошибкой,*

*// выставляем соответсвующие значения в хранилище*

feedFailed: true,

*// Запрос закончил своё выполнение*

feedRequest: false

};

}

default: {

return state

}

}

}

Чтобы пользовательский интерфейс корректно реагировал на асинхронную операцию, нам потребуется минимум три экшена:

1. Запрос начал выполняться: если значение равно true, показываем в интерфейсе лоудер, блокируем кнопки и так далее.
2. Запрос выполнился успешно: помещаем данные в хранилище для дальнейшей отрисовки их в компонентах, убираем статус выполнения запроса, прячем лоудеры и так далее.
3. Запрос выполнился с ошибкой: показываем в интерфейсе уведомление об ошибке и, к примеру, красим поля ввода в красный цвет. На этом этапе также убираем статус выполнения запроса.

Последнее, что нам осталось сделать, — написать компонент, в котором будет вызвана экшен-функция:

Скопировать кодJSX

import React, { useEffect } from 'react';

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

*// Наш thunk для запроса данных с сервера*

import { getFeed } from '../sevices/actions';

const NewsFeed = () => {

*// Вытаскиваем селектором нужные данные из хранилища*

const { feed, feedRequest, feedFailed } = useSelector(state => state.news);

*// Получаем метод dispatch*

const dispatch = useDispatch();

useEffect(()=> {

*// Отправляем экшен-функцию*

dispatch(getFeed())

}, [])

*// Используем условный рендеринг для разных состояний хранилища*

if (feedFailed) {

return <p>Произошла ошибка при получении данных</p>

} else if (feedRequest) {

return <p>Загрузка...</p>

} else {

return <>{feed.map(post => <NewsPost key={post.id} {...post} />)}</>;

}

}

Благодаря усилителю redux-thunk работа с асинхронными функциями стала удобной и предсказуемой. Вам ещё предстоит воспользоваться redux-thunk в проектной работе и задачах. Закрепите навыки работы с этим Redux-усилителем в тренажёре. Увидимся в следующем уроке!

# Работа с формами в Redux

Вы уже знаете, как работать с асинхронными функциями в Redux. Такие функции используются для взаимодействия с сервером. Но есть ещё кое-что, без чего большинство веб-приложений не имеют смысла — работа с формами. Мы уже разбирали [эту тему в предыдущем спринте](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/6673a3cf-f58b-4824-a5f0-32616f61ff80/lessons/a41af8fb-30a0-4fd3-960f-610aa2a2582b/), а в этом уроке поговорим о работе с формами на Redux.

## Зачем хранить данные форм в Redux

При работе с формами разработчики встают перед выбором: использовать внутреннее состояние компонента или глобальное хранилище Redux. И кажется, что проще выбрать первый вариант, ведь на Redux нужно написать много однотипного кода, что может быть не очень увлекательно. В большинстве случаев действительно удобнее использовать внутреннее состояние компонента для этой задачи. Но у такого подхода есть ряд проблем:

* Значение хранится до размонтирования компонента. После его удаления из DOM удаляются и введённые данные.
* Если форма «размазана» на несколько экранов, например форма покупки авиабилетов, то нужно «поднимать» состояние выше по иерархии компонентов.
* Это влечёт глубокий проброс пропсов (props drilling) и увеличение связанности и неразрывности компонентов.
* По мере усложнения приложения возникает потребность использовать значения, введённые в конкретной форме в разных частях приложения. Например, вы делаете «убийцу» 1С на React.

Поэтому разработчики часто прибегают ко второму варианту — хранению всего состояния в Redux, в том числе и форм. Такой подход хоть и многословнее, но лишён перечисленных недостатков.

## Как работать с формами в Redux

В npm есть несколько готовых решений, которые помогают автоматизировать эту рутинную работу, но мы не будем советовать вам конкретную библиотеку. Вместо этого покажем, что работа с формами в Redux не сложнее, чем работа с формами при использовании хука useState.

Для примера напишем форму регистрации участников на выставку экзотических рептилий. Нам понадобится один ключ в хранилище, редьюсер, экшен и генератор экшена.

Начнём с самого простого — напишем тип экшена, который будет управлять всей формой:

Скопировать кодJSX

*// constants.js*

export const PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE = 'PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE';

Затем создадим генератор экшена. Он будет принимать два аргумента — ключ формы и значение этого ключа:

Скопировать кодJSX

*// actions.js*

import { PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE } from './constants';

export const setParticipantFormValue = (field, value) => ({

type: PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE,

field,

value

})

Теперь опишем глобальное состояние формы и добавим редьюсер:

Скопировать кодJSX

*// reducer.js*

import { PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE } from './constants';

const initialState = {

form: {

name: '',

surname: '',

*// Числовое значение, количество животных*

numberOfPets: null,

*// Булево значение, дополнительная розетка для подключения света / подогрева*

extraSocket: false,

*// Булево значение, наличие собственного стеллажа для террариумов*

ownRack: false,

}

}

export const participantRegistrationReducer = (state = initialState, action) => {

switch(action.type) {

case PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE: {

return {

...state,

[action.field]: action.value

}

}

default: {

return state;

}

}

}

Это всё. Целая форма с разнообразными полями хранится глобально в Redux. Масштабировать такое приложение несложно, ведь не будет пробросов пропсов, а данные из этой формы, если они вдруг потребуются, доступны всему приложению.

Остались финальные штрихи — напишем саму форму:

Скопировать кодJSX

import React from 'react';

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

import { setParticipantFormValue } from './services/register/actions';

export const Registration = () => {

const {

name,

surname,

numberOfPets,

extraSocket,

ownRack

} = useSelector(state => state.participantRegistration.form);

const dispatch = useDispatch();

const onFormChange = (e) => {

if (e.target.name === 'extraSocket' || e.target.name === 'ownRack') {

dispatch(setParticipantFormValue(e.target.name, e.target.checked))

} else {

dispatch(setParticipantFormValue(e.target.name, e.target.value))

}

}

return (

<form>

<label htmlFor="name">Имя</label>

<input type="text" onChange={onFormChange} value={name} name="name" id="name" />

<label htmlFor="surname">Фамилия</label>

<input type="text" onChange={onFormChange} value={surname} name="surname" id="surname" />

<label htmlFor="numberOfPets">Количество питомцев</label>

<input type="number" min="1" onChange={onFormChange} value={numberOfPets} name="numberOfPets" id="numberOfPets"/>

<label htmlFor="extraSocket">Требуется дополнительная розетка</label>

<input type="checkbox" onChange={onFormChange} value={extraSocket} name="extraSocket" id="extraSocket"/>

<label htmlFor="ownRack">Собственный стеллаж</label>

<input type="checkbox" onChange={onFormChange} value={ownRack} name="ownRack" id="ownRack"/>

<button type="submit">Зарегистрироваться</button>

</form>

)

}

Форма готова! Но вы могли заметить, что данные формы никуда не отправляются. Исправим это: воспользуемся возможностями усилителя redux-thunk и напишем ещё один экшен. Так всё приложение будет знать, что в данный момент выполняется запрос на регистрацию. Это может быть полезно, если вы хотите показать лоудер или заблокировать страницу до конца выполнения запроса.

Добавим типы экшенов для отправки формы на сервер:

Скопировать кодJSX

*// constants.js*

export const PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE = 'PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE';

export const PARTICIPANT\_REGISTER\_FORM\_SUBMIT = 'PARTICIPANT\_REGISTER\_FORM\_SUBMIT';

export const PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_SUCCESS = 'PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_SUCCESS';

export const PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_FAILED = 'PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_FAILED';

Доработаем редьюсер:

Скопировать кодJSX

*// reducer.js*

import {

PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE,

PARTICIPANT\_REGISTER\_FORM\_SUBMIT,

PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_SUCCESS,

PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_FAILED

} from './constants';

const initialState = {

form: {

name: '',

surname: '',

numberOfPets: null,

extraSocket: false,

ownRack: false,

}

*// Добавили новые поля в хранилище*

registrationRequest: false,

registrationFailed: false,

}

export const participantRegistrationReducer = (state = initialState, action) => {

switch(action.type) {

case PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE: {

return {

form: {

...state.form,

[action.field]: action.value

}

}

}

case PARTICIPANT\_REGISTER\_FORM\_SUBMIT: {

return {

...state,

registrationRequest: true,

registrationFailed: false

}

}

case PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_SUCCESS: {

return {

...state,

form: {

*// При успешной регистрацией сбрасываем форму до исходного состояния*

...initialState.form

}

registrationRequest: false

}

}

case PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_FAILED: {

return {

...state,

registrationRequest: false,

registrationFailed: true

}

}

default: {

return state;

}

}

}

И экшены:

Скопировать кодJSX

*// actions.js*

import {

PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE,

PARTICIPANT\_REGISTER\_FORM\_SUBMIT,

PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_SUCCESS,

PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_FAILED

} from './constants';

export const setParticipantFormValue = (field, value) => ({

type: PARTICIPANT\_REGISTER\_FORM\_SET\_VALUE,

field,

value

});

export const register = () => (dispatch, getState) => {

dispatch({

type: PARTICIPANT\_REGISTER\_FORM\_SUBMIT

});

fetch('/api/register', {

method: 'POST',

*// В тело запроса помещаем значения из хранилища*

body: JSON.stringify(...getState().participantRegistration.form)

}).then(res=>{

return res.json();

}).then(data => {

dispatch({

type: PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_SUCCESS,

data

});

}).catch(err => {

dispatch({

type: PARTICIPANT\_REGISTER\_FORM\_SUBMIT\_FAILED,

});

})

}

Усилитель redux-thunk передаёт в аргументы экшена методы хранилища (store): dispatch и getState. Таким образом, у нас есть доступ к актуальному состоянию хранилища прямо внутри thunk-экшена.

Доделаем форму:

Скопировать кодJSX

import React from 'react';

import { useSelector, useReducer } from 'react-redux';

import { setParticipantFormValue, register } from './services/register/actions';

export const Registration = () => {

const {

name,

surname,

numberOfPets,

extraSocket,

ownRack

} = useSelector(state => state.participantRegistration.form);

*// Используем статус выполнения запроса для блокировки кнопки регистрации*

const { registrationRequest } = useSelector(state => state.participantRegistration);

const dispatch = useDispatch();

const onFormChange = (e) => {

if (e.target.name === 'extraSocket' || e.target.name === 'ownRack') {

dispatch(setParticipantFormValue(e.target.name, e.target.checked))

} else {

dispatch(setParticipantFormValue(e.target.name, e.target.value))

}

}

const onFormSubmit = (e) => {

*// Предотвращаем дефолтное поведение формы при её отправке*

e.preventDefault();

*// Вызываем наш thunk-экшен*

dispatch(register())

}

return (

<form onSubmit={onFormSubmit}>

<label htmlFor="name">Имя</label>

<input type="text" onChange={onFormChange} value={name} name="name" id="name" />

<label htmlFor="surname">Фамилия</label>

<input type="text" onChange={onFormChange} value={surname} name="surname" id="surname" />

<label htmlFor="numberOfPets">Количество питомцев</label>

<input type="number" min="1" onChange={onFormChange} value={numberOfPets} name="numberOfPets" id="numberOfPets"/>

<label htmlFor="extraSocket">Требуется дополнительная розетка</label>

<input type="checkbox" onChange={onFormChange} value={extraSocket} name="extraSocket" id="extraSocket"/>

<label htmlFor="ownRack">Собственный стеллаж</label>

<input type="checkbox" onChange={onFormChange} value={ownRack} name="ownRack" id="ownRack"/>

<button type="submit" disabled={registrationRequest}>Зарегистрироваться</button>

</form>

)

}

Теперь форма полностью готова и может приносить пользу. Кроме этого, её состояние хранится в Redux. Для пользователя это значит, что форма сохранит введённые им значения даже если он отвлечётся на другие страницы сайта. Конечно, при условии, что пользователь не закроет сайт.

## Заключение

Теперь вы знаете, как работать с формами в Redux. Не всем приложениям требуется хранить значения форм в Redux. К тому же это требует дополнительного кода, написание которого отнимает время. Но уже в следующем уроке мы расскажем, как значительно сократить количество кода и сэкономить время при работе с Redux.

# Избавляемся от лишнего кода: Redux Toolkit

Из предыдущего урока вы узнали, как Redux работает с формами. В этом уроке вы познакомитесь с инструментом, который позволяет упростить работу с этой библиотекой.

Несмотря на то, что Redux — сильный инструмент для разработки проектов разной сложности, у него есть ряд недостатков:

* обладает сложной конфигурацией — необходимо создавать редьюсеры, усилители, подключать средства отладки;
* требует множество вспомогательных сущностей для работы — типов и генераторов экшенов, редьюсеров, селекторов, сайд-эффектов;
* нуждается в большом объёме кода для простых операций.

Многокомпонентность Redux повышает порог входа для начинающих пользователей библиотеки. Приходится писать много однотипного кода для простых операций. Например, чтобы просто изменить одно поле в глобальном стейте, нужно написать экшен, редьюсер, а затем ещё и селектор. При этом сам тип экшена будет одним и тем же. Кроме того, все типы экшенов принято располагать в отдельных компонентах, что тоже увеличивает количество разных файлов с похожими названиями.

При создании редьюсеров также часто появляется большая вложенность. Вы могли обратить на это внимание [в уроке об инициализации хранилища](https://praktikum.yandex.ru/trainer/react/lesson/c03bd8f1-6a0e-4028-b607-12b61f3a4616/task/7ff5536a-6b04-4387-b416-f512903fec64/):

Скопировать кодJSX

*// Редьюсер*

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

switch (action.type) {

*// Добавление новой задачи в список дел*

case ADD\_TODO:

return [

...state,

{

id: guid(),

text: action.text,

expiresAt: action.expiresAt,

completed: false

}

]

*// Изменение статуса задачи в списке дел*

case TOGGLE\_TODO:

return state.map(todo =>

todo.id === action.id ? { ...todo, completed: !todo.completed } : todo

)

*// Реакция на прочие типы экшенов*

default:

return state

}

}

В этом примере все TODO находятся на одном уровне, но часто бывает так, что они располагаются на уровень глубже в ключе.

Именно по этим причинам сообщество Redux разработало [библиотеку Redux Toolkit](https://redux-toolkit.js.org/).

## Установка Redux Toolkit в проект

Библиотека Redux Toolkit уже по умолчанию содержит Redux, поэтому её нужно установить только в проект. Для этого можно воспользоваться одной из команд:

Скопировать кодBASH

*# npm*

npm install @reduxjs/toolkit

или

Скопировать кодBASH

*# yarn*

yarn add @reduxjs/toolkit

Всё, Redux Toolkit установлен. Теперь подробнее рассмотрим эту библиотеку.

## Особенности работы Redux Toolkit

Redux Toolkit не только упрощает и автоматизирует написание компонентов Redux (экшенов и их генераторов, редьюсеров, селекторов), но и сокращает количество ошибок при их создании. Для этой цели разработчики создали несколько функций. Мы разберём основные: configureStore(), createSlice(), createAction(), createReducer().

### Инициализация хранилища. configureStore()

Эта функция очень похожа [на знакомую вам createStore() из Redux](https://practicum.yandex.ru/trainer/web-plus/lesson/772fd43b-9cb5-46b1-b572-78a79d5da695/task/ec156e55-314f-419a-bf8c-7195aeb3100e/), кроме чуть более упрощённого синтаксиса и встроенной возможности включать или выключать devTools, что используют для отладки в браузере.

Так выглядит инициализация глобального хранилища в Redux:

Скопировать кодJSX

import { createStore } from 'redux';

import { rootReducer } from './reducers';

const store = createStore(rootReducer);

Вроде бы просто, но это пока. Когда потребуется подключать мидлвары, средства отладки или другие сущности, код приобретёт совсем другой вид.

Например, здесь мы добавили начальный стейт, средства отладки и мидлвары:

Скопировать кодJSX

import { createStore } from 'redux';

import { rootReducer } from './reducers';

import { composeWithDevTools } from 'redux-devtools-extension';

import { applyMiddleware } from 'redux';

import { midleware } from './midlewares'

const initialState = {todos: []}

const store = createStore(

rootReducer,

initialState,

compositeWithDevTools(applyMiddleWare(...midleware))

);

И это ещё не всё. Средства отладки устанавливаются отдельно при помощи applyMiddleware.

А так можно сделать то же самое, но с помощью configureStore():

Скопировать кодJSX

import { configureStore } from '@reduxjs/toolkit'

import logger from 'redux-logger'

import todosReducer from './todos/todosReducer'

import { customEnhancer } from './enhancers'

const preloadedState = {

todos: [],

}

const store = configureStore({

reducer,

middleware: (getDefaultMiddleware) => getDefaultMiddleware().concat(logger),

devTools: process.env.NODE\_ENV !== 'production',

preloadedState,

enhancers: [customEnhancer],

})

Это очень похоже на Redux, но немного удобнее:

* middleware, enhancers и средства отладки разделены;
* для подключения средств отладки не нужно устанавливать библиотеку, это работает из коробки Redux Toolkit;
* Redux Thunk возможно использовать также из коробки. [По ссылке можно ознакомиться с примером такого использования](https://redux-toolkit.js.org/api/getDefaultMiddleware).

Обратите внимание: если вам требуется использовать другую библиотеку для сайд- эффектов, просто подключите её в качестве еще одного middleware.

### Создание экшенов, генераторов экшенов и редьюсеров. createSlice()

Пожалуй, это самая важная функция, ради которой стоит использовать Redux Toolkit. Функция createSlice() создаёт экшены, генераторы экшенов и редьюсеры. Посмотрим, как она работает — создадим простые экшены без данных (payload):

Скопировать кодJSX

import { createSlice } from '@reduxjs/toolkit

const counterSlice = createSlice({

name: 'counter',

initialState: 0,

reducers: {

increment: (state) => state + 1,

},

})

В коде мы использовали:

* name — ключ, который станет префиксом всех экшенов. Например: type: 'counter/increment');
* reducers — логика редьюсеров;
* initialState — начальное значение стейта этого редьюсера.

Теперь мы можем получить из counterSlice такие элементы:

* counterSlice.actions — возвращает объект с экшенами. В нашем примере — counterSlice.actions.increment();
* counterSlice.reducer — возращает редьюсер.

Теперь немного усложним пример — создадим экшен с данными (payload):

Скопировать кодJSX

import { createSlice } from '@reduxjs/toolkit'

const counterSlice = createSlice({

name: 'counter',

initialState: 0,

reducers: {

incrementByValue: (state, action) => state + action.payload,

},

})

createSlice позволяет изменять данные глобального стейта без необходимости заботиться об иммутабильности. То есть визуально это может выглядеть как мутация, но «под капотом» библиотека создаёт копию стейта и мутации не происходит.

Функция createSlice создаёт редьюсеры и экшены, которые ничем не отличаются от обычных, созданных без Redux Toolkit. Чтобы в этом убедиться, перепишем код [из предыдущих уроков](https://practicum.yandex.ru/trainer/web-plus/lesson/4325ba8c-5145-4bac-a6a7-7219f6596967/task/594a57c7-51aa-4577-8a09-7abb561aaa1d/) с помощью Redux Toolkit:

Скопировать кодJSX

import { useDispatch } from 'react-redux';

import { useEffect } from 'react';

import { counterSlice } from './slices'

const App = (props) => {

const dispatch = useDispatch();

const { actions } = counterSlice();

useEffect(() => {

*// Отправляем экшен при монтировании компонента*

dispatch(actions.incrementByValue(1));

}, [])

*// ...*

}

export default App;

Кроме функции createSlice библиотека Redux Toolkit включает множество дополнительных функций, которые представляют собой синтаксический сахар.

### Создание экшена одновременно с его типом. createAction()

В Redux сначала объявляется константа с типом экшена, а затем создаётся генератор экшена с использованием этой константы:

Скопировать кодJSX

const action = {

type: 'ACTION\_NAME'

}

А в Redux Toolkit с применением createAction() создание экшена и его типа выглядит так:

Скопировать кодJSX

import { createAction } from '@reduxjs/toolkit'

const increment = createAction('ACTION\_NAME')

В случае, если экшен содержит данные(payload), это можно записать так:

Скопировать кодJSX

import { createAction } from '@reduxjs/toolkit'

const addTodo = createAction('todos/add', function prepare(text) {

return {

payload: {

text,

id: 1,

createdAt: '08.04.20201',

},

}

})

console.log(addToDo)

*// {*

*// type: 'todos/add',*

*// payload: {*

*// text,*

*// id: 1,*

*// createdAt: '08.04.20201'*

*// }*

*// }*

В итоге addToDo будет выглядеть таким образом:

Скопировать кодJSX

import { useDispatch } from 'react-redux';

import { useEffect } from 'react';

import { addTodo } from './actions'

const App = (props) => {

const dispatch = useDispatch();

const { actions } = counterSlice();

useEffect(() => {

*// Отправляем экшен при монтировании компонента*

dispatch(addTodo);

}, [])

*// ...*

}

export default App;

### Создание редьюсеров. createReducer()

Эта функция выглядит чуть сложнее, но только внешне. На самом деле, это просто генератор редьюсера в удобной (по мнению авторов библиотеки) форме.

Так можно создать редьюсер без применения Redux Toolkit:

Скопировать кодJSX

...

import { counterReducer } from './reducers'

...

*// Редьюсер списка дел*

const todoList = (state, action) => { ... }

*// Редьюсер пользователя приложения*

const user = (state, action) => { ... }

*// Редьюсер коллективной работы над списком дел*

const collabaration = (state, action) => { ... }

*// Корневой редьюсер*

const rootReducer = (state, action) => {

todoList: todoList(state.todoList, action),

user: user(state.user, action),

collabaration: collabaration(state.collabaration, action),

counterReducer

}

А так можно сделать то же самое, но с использованием функции createReducer:

Скопировать кодJSX

import { createAction, createReducer } from '@reduxjs/toolkit'

const increment = createAction('counter/increment')

const decrement = createAction('counter/decrement')

const incrementByAmount = createAction('counter/incrementByAmount')

const initialState = { value: 0 }

const counterReducer = createReducer(initialState, (builder) => {

builder

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

state.value++

})

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

state.value--

})

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

state.value += action.payload

})

})

На выходе counterReducer будет выглядеть так же, как обычный редьюсер со switch и нескольким case внутри.

В этом уроке мы рассказали о четырёх основных функциях Redux Toolkit. Их достаточно, чтобы начать пользоваться библиотекой. Но это неполный список встроенных функций, с остальными вы можете самостоятельно ознакомиться [на официальном сайте Redux Toolkit](https://redux-toolkit.js.org/).