

REDUX, REDUX THINK



КОНСТАНТИН ПОЛЯНСКИЙ



КОНСТАНТИН ПОЛЯНСКИЙ

Software Engineer

kv.polyanskiy@gmail.com

[@kvpolyanskiy](https://www.instagram.com/kvpolyanskiy)

ПЛАН ЗАНЯТИЯ

1. [Redux](#)
2. [React Redux](#)
3. [Хуки `useSelector`, `useDispatch`](#)
4. [`connect`](#)
5. [Асинхронные действия](#)
6. [Middleware](#)
7. [Redux Thunk](#)



REDUX

UNIDIRECTIONAL FLOW

React использует подход с однонаправленным потоком данных - мы обновляем `state`, а это приводит к перерендерингу, но не наоборот.

Мы уже посмотрели, на то, как:

- использовать локальное состояние
- передавать дочерним компонентам локальное состояние через `props`
- как использовать Context API



REDUX

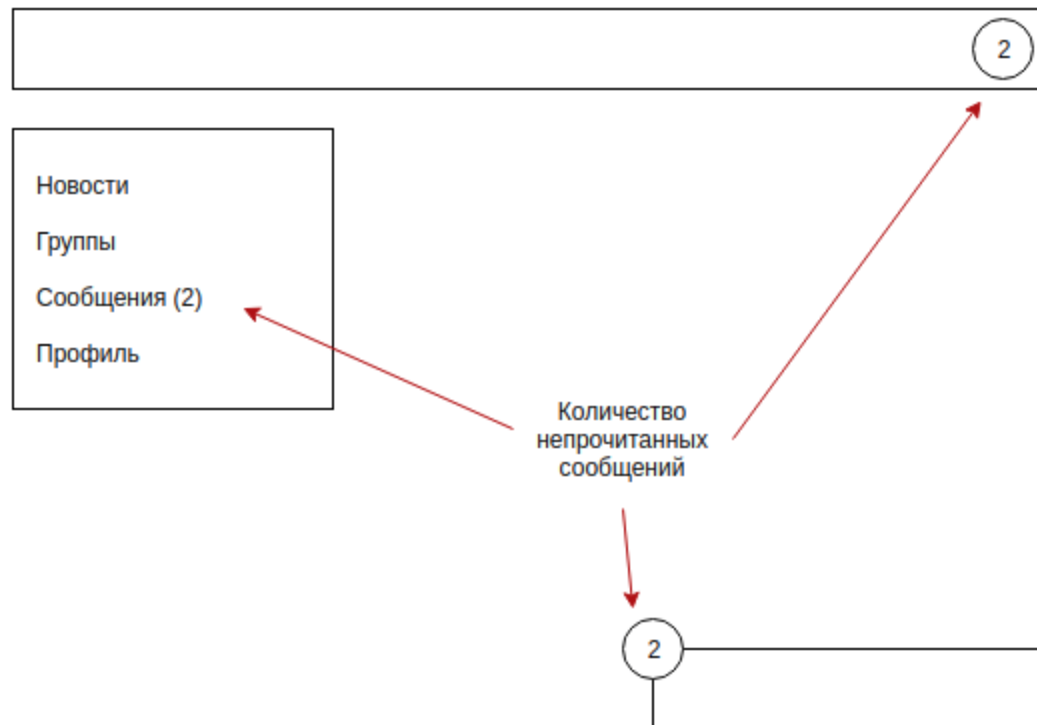
Redux - A predictable state container for JavaScript apps.

Перевод: Redux - предсказуемый контейнер состояния для JavaScript приложений (по факту - библиотека, позволяющая организовать управление состоянием).

REDUX

Q: Зачем он нужен?

А: Представьте, что у нас достаточно большое дерево компонентов и при этом некоторые компоненты достаточно сильно связаны по данным, например (схема интерфейса Fb, Vk):

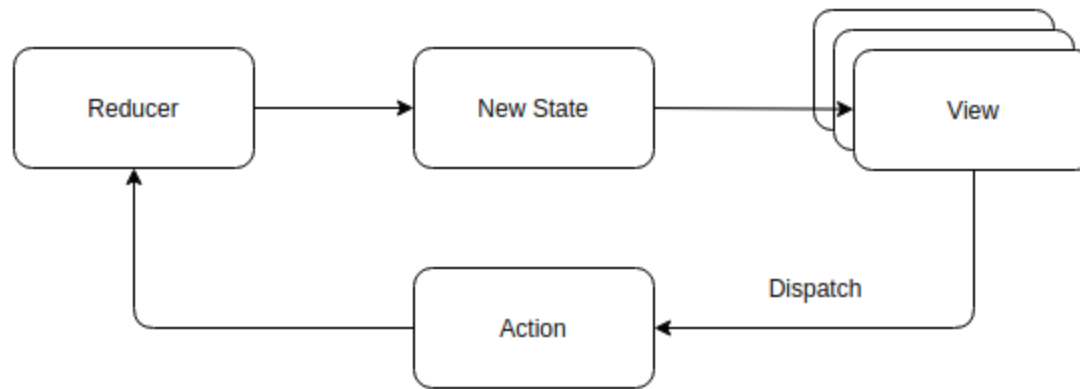


REDUX

Основная идея всё та же, данные перемещаются только в одном направлении, но "действующих лиц" становится чуть больше:

- View
- Action
- Reducer
- Store

REDUX



VIEW

View - визуальная часть (дерево компонентов в React).

Отображает данные, хранящиеся в **Store** (важно: только отображает, но не изменяет). **View** "подписаны" на изменения состояния в **Store**, поэтому обновляются согласно обновлениям данных в **Store**.

Взаимодействие пользователя со **View** приводит к тому, что генерируется **Action**.

ACTION

`Action` - обычный JS-объект (data object), который содержит все необходимые данные для обновления состояния, которое хранится в `Store`.

К нему (в рамках Redux) предъявляется только одно требование: должно быть поле `type`, описывающее тип Action'a.

Вы можете воспринимать `Action`'ы, как события, которые приводят к изменению состояния, хранящегося в `Store`.

STORE

Store - объект, который хранит данные приложения. Его задача, реагировать на Action'ы и, при необходимости, изменять своё состояние (данные внутри **Store**).

Важно, **Store** реагирует только на **Action** 'ы, передавая во **View** изменения своего состояния.

Поскольку **View** занимаются отображением данных из **Store**, эти изменения приводят к перерендерингу **View** (если он необходим).

STORE

У Store есть два ключевых метода:

1. `dispatch` - возможность отправить `Action` в `Store`
2. `subscribe` - возможность подписаться на обновления состояния в `Store`

Сам код `Store` занимает всего около 300 строк (большая часть из которых - комментарии и проверки).

Мы настоятельно вам рекомендуем ознакомиться с этим кодом по ссылке:

<https://github.com/reduxjs/redux/blob/master/src/createStore.js>

DISPATCH

`dispatch` позволяет отправить `Action` в `Store`:

```
1  if (isDispatching) {  
2    throw new Error('Reducers may not dispatch actions.')  
3  }  
4  
5  try {  
6    isDispatching = true  
7    currentState = currentReducer(currentState, action)  
8  } finally {  
9    isDispatching = false  
10 }  
11  
12 const listeners = (currentListeners = nextListeners)  
13 for (let i = 0; i < listeners.length; i++) {  
14   const listener = listeners[i]  
15   listener()  
16 }
```

REDUCER

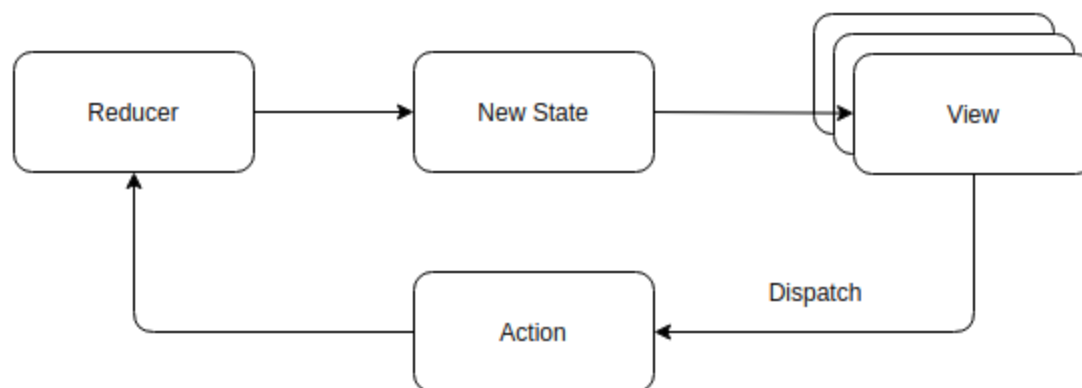
Reducer - Pure Function (чистая функция), которая отвечает за то, чтобы на базе текущего состояния и **Action** 'а сгенерировать новое состояние.

Вспомните примеры с CRUD, которые мы проходили, например, прочтение сообщения (мы никогда не меняем состояние, всегда генерируем новое):

```
// кто-то наверху передаёт id прочитанного сообщения
prevState => (
  {messages: prevState.messages.map(o => o.id !== id ? o : {...o, read: true})}
);
```

REDUX

Таким, образом:



REDUX

Основные идеи:

1. **Store** всего один на всё приложение
2. Полное состояние всего приложения представляется в виде единственного JS-объекта
3. **State** - ReadOnly-объект, который может быть изменён (по факту - сгенерирован новый) с помощью **Action**
4. К **Action** предъявляется требование:
 - это минимально необходимый объём информации, необходимый для генерации нового состояния
 - в нём должно быть поле **type**, описывающее **Action**
5. Вычисление нового состояния на базе **Action** производится с помощью **Reducer** 'ов - чистых функций (Pure Functions).

БЕЗ REDUX

Чуть позже мы рассмотрим живой пример на Redux, но давайте подумаем, как донести до всех этих трёх виджетов актуальную информацию в чистом React (без Redux)?

1. Пробрасывать local state через props сверху вниз (например, от компонента App) - неудобно
2. Использовать Context API - удобно, решает проблему

Q: Но если Context API (который мы с вами уже рассматривали) решает проблему, зачем нам Redux?

A: Это очень правильный вопрос.

CONTEXT API VS REDUX

На сегодняшний день Context API с успехом заменяет Redux во многих проектах. Вы даже можете встретить статьи из серии "Redux is dead", "Context API vs Redux" и т.д..

Кроме того, в небольших проектах и проектах среднего размера Redux (и централизованное управление состоянием) может быть не нужно.

Но, большие проекты используют Redux.

И самое главное - вокруг Redux выстроена целая экосистема, которая позволяет вам необходимости переизобретать велосипеды (Context API этой экосистемой ещё только обзаводится).



REACT REDUX



ЗАДАЧА

Пример с сообщениями достаточно неплох, но давайте разбираться с Redux по порядку.

Начнём с обычного CRUD: сначала это будет хранение данных локально, затем - взаимодействие по HTTP.

Представим, что у нас есть панель управления услугами небольшого сайта по ремонту iPhone, где мы можем редактировать список услуг (для упрощения, мы рассмотрим только просмотр списка и добавление).

REACT REDUX

Redux никак не завязан на React и может использоваться в приложениях, в которых React не используется.

Поэтому для удобной связки React и Redux есть библиотека `react-redux`.

Установим все необходимые зависимости:

```
npx create-react-app frontend  
cd frontend  
npm install nanoid prop-types redux react-redux  
npm start
```

СТРУКТУРА КАТАЛОГОВ

Создадим следующую структуру каталогов

- `components` - наши компоненты
- `actions` - всё, что нужно для создания и работы с `Action`'ами
- `reducers` - `Reducer`'ы
- `store` - код для создания `Store`'ы

SERVICELIST

```
import React from 'react'
import PropTypes from 'prop-types'

function ServiceList() {
  const items = ...;

  handleRemove = id => { }

  return (
    <ul>
      {items.map(o => <li key={o.id}>
        {o.name} {o.price}
        <button onClick={() => handleRemove(o.id)}>x</button>
      </li>)}
    </ul>
  )
}
```


SERVICEADD

```
import React from 'react'
import PropTypes from 'prop-types'

function ServiceAdd() {
  const item = ...;
  const handleChange = event => { const {name, value} = event.target; }
  const handleSubmit = event => { event.preventDefault(); }

  return ( <form onSubmit={handleSubmit}>
    <input name='name' onChange={handleChange} value={item.name} />
    <input name='price' onChange={handleChange} value={item.price} />
    <button type='submit'>Save</button>
  </form>))

export default ServiceAdd;
```

SERVICEADD

```
import React, {Fragment} from 'react';
import ServiceAdd from './components/ServiceAdd';
import ServiceList from './components/ServiceList';

function App() {
  return (
    <Fragment>
      <ServiceAdd />
      <ServiceList />
    </Fragment>
  );
}

export default App
```

REDUCERS: SERVICELIST.JS

```
import nanoid from 'nanoid';
import {ADD_SERVICE, REMOVE_SERVICE} from '../actions/actionTypes'

const initialState = [
  {id: nanoid(), name: 'Замена стекла', price: 21000},
  {id: nanoid(), name: 'Замена дисплея', price: 25000},
];

export default function serviceListReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_SERVICE:
      const {name, price} = action.payload;
      return [...state, {id: nanoid(), name, price: Number(price)}];
    case REMOVE_SERVICE:
      const {id} = action.payload;
      return state.filter(service => service.id !== id);
    default:
      return state;
  }
}
```

REDUCERS: SERVICEADD.JS

```
import {CHANGE_SERVICE_FIELD} from '../actions/actionTypes'

const initialState = {
  name: '',
  price: '',
};

export default function serviceAddReducer(state = initialState, action) {
  switch (action.type) {
    case CHANGE_SERVICE_FIELD:
      const {name, value} = action.payload;
      return {...state, [name]: value};
    default:
      return state;
  }
}
```

ACTION

Как мы уже говорили, `Action` - это обычный JS-объект, к которому предъявляется только одно требование: в нём должно быть поле `type`.

В качестве `type` чаще всего используют строки, и чтобы не приходилось их каждый раз печатать (есть риск ошибиться), вынесем их в константы:

```
export const ADD_SERVICE      = 'ADD_SERVICE';  
export const REMOVE_SERVICE   = 'REMOVE_SERVICE';  
export const CHANGE_SERVICE_FIELD = 'CHANGE_SERVICE_FIELD';
```

ACTION

Существуют разные схемы "именования" `Action` 'ов:

- кто-то предлагает сначала писать объект, к которому относиться `Action`, потом глагол: `SERVICE_ADD`
- кто-то наоборот: `ADD_SERVICE`

Ключевое, конечно же, быть последовательным, и в рамках одного проекта придерживаться единой схемы.

STORE

Поскольку `Store` у нас может быть только один, создадим его в файле `store/index.js`

(это позволит нам писать `import store from './store'`):

```
import { createStore, combineReducers } from "redux";
import serviceListReducer from '../reducers/serviceList';
import serviceAddReducer from '../reducers/serviceAdd';

const reducer = combineReducers({
  serviceList: serviceListReducer,
  serviceAdd: serviceAddReducer,
});

const store = createStore(reducer);

export default store;
```

STORE

Пояснения:

- `createStore` - функция, которая создаёт `Store`, первый аргумент - корневой `Reducer`
- `combineReducers` - функция, которая комбинирует `Reducer` 'ы, создавая корневой `Reducer`

Q: Зачем это нужно?

A: В Redux любой `Action` проходит через все `Reducer` 'ы.

`combineReducer` позволяет нам выделить каждому `Reducer` 'у кусочек `state`.

Когда наш `state` разрастётся, это будет очень удобно.

STORE

Важное замечание: структура `state` целиком зависит от вашего приложения. Например, если вы хотите помимо сервисов ещё учитывать заказы, а также хранить аутентификацию, то структура может быть и такой:

```
{
  services: {
    list: [...],
    newItem: {...},
  },
  purchases: {
    list: [...],
  },
  auth: {
    token: ...,
    profile: {...},
  },
}
```

PROVIDER

`Provider` - это готовый компонент из `react-redux`, который предоставляет нам возможность использовать в дочерних компонентах `Store` (файл `index.js`):

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import store from './store';
import {Provider} from 'react-redux';

ReactDOM.render(
  <Provider store={store}><App /></Provider>,
  document.getElementById('root')
);

serviceWorker.unregister();
```

Осталось только научиться отправлять данные в `Store` (`dispatch` 'ить и выбирать их оттуда).

PROVIDER

В React DevTools можно увидеть, во что это превращается:

The screenshot shows the React DevTools interface with the 'Elements' tab selected. The component tree on the left shows a hierarchy starting with `<Provider>`, which contains a `<Context.Provider>` component. This component is further expanded to show an `<App>` component, which contains `<ServiceAdd>` and `<ServiceList>` components. The right panel shows the props for the selected `<Context.Provider>` component. The props include `children`, `value`, and `subscription`. The `value` prop is a Redux store object with methods like `dispatch`, `getState`, `replaceReducer`, `subscribe`, and `unsubscribe`. The `subscription` prop is a `Subscription` object with methods like `__proto__`, `handleChangeWrapper`, `listeners`, `onStateChange`, `store`, and `unsubscribe`.

```
Elements Profiler
Search (text or /regex/)

▼ <Provider>
  ▼ <Context.Provider> == $r
    ▼ <App>
      ► <ServiceAdd>...</ServiceAdd>
      ► <ServiceList>...</ServiceList>
    </App>
  </Context.Provider>
</Provider>

Props
► children: {...}
▼ value: {...}
  ▼ store: {...}
    ► dispatch: dispatch()
    ► getState: getState()
    ► replaceReducer: replaceReducer()
    ► subscribe: subscribe()
  ▼ subscription: Subscription{...}
    ► __proto__: {...}
    ► handleChangeWrapper: bound handleChangeWrapper()
    ► listeners: {...}
    ► onStateChange: bound notifySubscribers()
    ► store: {...}
    ► unsubscribe: unsubscribe()
```



useSelector
useDispatch

useSelector useDispatch

Эти два хука позволяют нам выбрать нужный кусочек `state` и получить функцию для `dispatch`'а `Action`'ов:

```
// useSelector  
const items = useSelector(state => state.items);
```

```
// useDispatch:  
const dispatch = useDispatch();  
dispatch({type: SERVICE_REMOVE, payload: id});
```

Глядя на последний пример кода несложно заметить, что если один и тот же `Action` мы будем посылать из разных компонентов, то это будет повторение кода.

ACTION CREATORS

Чтобы не повторять процедуру создания `Action` в разных частях кода, мы вынесем их в отдельный файл: `actions/actionCreators.js`:

```
import { ADD_SERVICE, REMOVE_SERVICE, CHANGE_SERVICE_FIELD } from './actionTypes';

export function addService(name, price) {
  return {type: ADD_SERVICE, payload: {name, price}};
}

export function removeService(id) {
  return {type: REMOVE_SERVICE, payload: {id}};
}

export function changeServiceField(name, value) {
  return {type: CHANGE_SERVICE_FIELD, payload: {name, value}}
}
```

В нашем случае это может показаться немного избыточным, но наша задача показать, как строятся комплексные проекты.

SERVICELIST

```
import React from 'react'
import {useSelector, useDispatch} from 'react-redux'
import { removeService } from '../actions/actionCreators';

function ServiceList() {
  const items = useSelector(state => state.serviceList);
  const dispatch = useDispatch();

  const handleRemove = id => {
    dispatch(removeService(id));
  }

  return (
    <ul>
      {items.map(o => <li key={o.id}>
        {o.name} {o.price}
        <button onClick={() => handleRemove(o.id)}>x</button>
      </li>)}
    </ul>
  )
}
```

SERVICEADD

```
import { useSelector, useDispatch } from 'react-redux';
import { changeServiceField, addService } from '../actions/actionCreators';

function ServiceAdd(props) {
  const item = useSelector(state => state.serviceAdd);
  const dispatch = useDispatch();

  const handleChange = evt => {
    const {name, value} = evt.target;
    dispatch(changeServiceField(name, value));
  }

  const handleSubmit = evt => {
    evt.preventDefault();
    dispatch(addService(item.name, item.price));
  }

  return (<form onSubmit={handleSubmit}>
    <input name='name' onChange={handleChange} value={item.name} />
    <input name='price' onChange={handleChange} value={item.price} />
    <button type='submit'>Save</button>
  </form>)
}
```


SERVICEADD

Здесь важно отметить несколько моментов:

1. Любое изменение поля ввода приводит к изменению `state`, а значит вызове всех `Reducer` 'ов. Иногда, чтобы избежать этого, в таких формах оставляют внутреннее состояние.
2. Мы специально передаём в `addService` `name` и `price`, как вы думаете, почему? Ведь они итак хранятся в `Store`.
3. После добавления поля не очищаются. Какие у вас есть идеи по этому поводу?

combineReducers

`combineReducers` подходит для ограниченного числа сценариев, при которых каждый `Reducer` отвечает за свой кусочек `state`.

Поэтому мы не можем из `serviceListReducer` добраться до того, что хранится в `state.serviceAdd`.

Мы можем написать альтернативу, которая будет передавать нужные данные в `Reducer`, либо воспользоваться Redux Thunk, о котором мы будем говорить чуть позже.



connect

connect

react-redux помимо хуков предлагает нам HOC `connect`, который пробросит в `props` нашего компонента `dispatch` и нужные кусочки `state`:

```
class ServiceList extends Component { ... }

const mapStateToProps = (state, ownProps) => {
  const {serviceList} = state;
}
```

```
import React, { Component } from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import { changeServiceField, addService } from '../actions/actionCreators';

class ServiceAddClassBased extends Component {
  handleChange = evt => {
    const { name, value } = evt.target;
    this.props.onChange(name, value);
  }

  handleSubmit = evt => {
    evt.preventDefault();
    this.props.onSave(this.props.item.name, this.props.item.price);
  }

  render() {
    const { item } = this.props;

    return (
      <form onSubmit={this.handleSubmit}>
        <input name='name' onChange={this.handleChange} value={item.name} />
        <input name='price' onChange={this.handleChange} value={item.price} />
        <button type='submit'>Save</button>
      </form>
    )
  }
}
```

```
ServiceAddClassBased.propTypes = {
  item: PropTypes.shape({
    name: PropTypes.string,
    price: PropTypes.string,
  }).isRequired,
  onSave: PropTypes.func.isRequired,
  onChange: PropTypes.func.isRequired,
}

const mapStateToProps = (state, ownProps) => {
  const { serviceAdd } = state;
  return { item: serviceAdd };
}

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    onChange: (name, value) => dispatch(changeServiceField(name, value)),
    onSave: (name, value) => dispatch(addService(name, value)),
  }
};

export default connect(mapStateToProps, mapDispatchToProps)(ServiceAddClassBased);
```

connect

`connect` можно использовать и с функциональными компонентами:

```
function ServiceAdd(props) {
  const { item } = props;
  const handleChange = evt => {
    ...
    props.onChange(name, value);
  }
  const handleSubmit = evt => {
    ...
    props.onSave(item.name, item.price);
  }
  return (... аналогично предыдущему слайду ...)
}
ServiceAddClassBased.propTypes = {
  ... аналогично предыдущему слайду ...
}
const mapStateToProps = (state, ownProps) => {
  ... аналогично предыдущему слайду ...
}
const mapDispatchToProps = (dispatch, ownProps) => {
  ... аналогично предыдущему слайду ...
};
export default connect(mapStateToProps, mapDispatchToProps)(ServiceAdd);
```

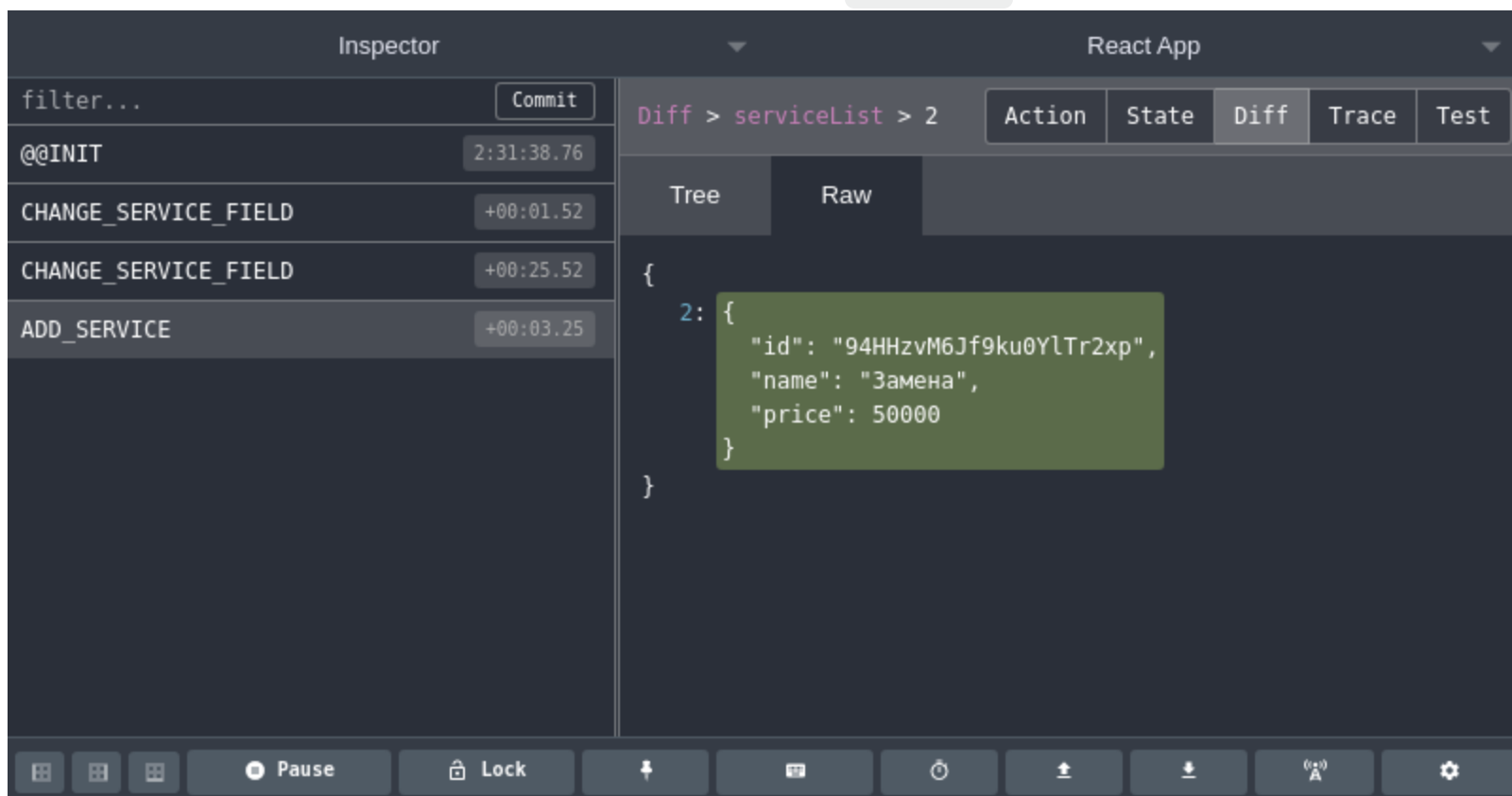


HOOKS VS connect

Какой из вариантов использовать - решать вам, но на данный момент `connect` встречается достаточно часто, и необходимо разбираться в его использовании.

REDUX DEVTOOLS

Удобно отслеживать взаимодействие со `Store` помогает Redux DevTools:



REDUX DEVTOOLS

```
const store = createStore(  
  reducer,  
  window.__REDUX_DEVTOOLS_EXTENSION__ && window.__REDUX_DEVTOOLS_EXTENSION__(),  
);
```



АСИНХРОННЫЕ ОПЕРАЦИИ



HTTP

Это всё хорошо, но данные мы обычно получаем по HTTP. И сохранение/удаление должны не только проходить по HTTP, но после них также должна следовать перезагрузка всего списка.

А, кроме того, нужно ещё отображать индикатор загрузки и отображать ошибки пользователю.



API

```
npm init  
npm install forever koa koa-router koa2-cors koa-body
```

API

.foreverignore:

```
node_modules
```

scripts в package.json:

```
"scripts": {  
  "prestart": "npm install",  
  "start": "forever server.js",  
  "watch": "forever -w server.js"  
},
```

API

```
const http = require('http');
const Koa = require('koa');
const Router = require('koa-router');
const cors = require('koa2-cors');
const koaBody = require('koa-body');

const app = new Koa();
app.use(cors());
app.use(koaBody({json: true}));

let nextId = 1;
const services = [
  { id: nextId++, name: 'Замена стекла', price: 21000, },
  { id: nextId++, name: 'Замена дисплея', price: 25000, },
  { id: nextId++, name: 'Замена аккумулятора', price: 4000, },
  { id: nextId++, name: 'Замена микрофона', price: 2500, },
];
```

```

const router = new Router();
router.get('/api/services', async (ctx, next) => {
  ctx.response.body = services;
});
router.post('/api/services', async (ctx, next) => {
  const id = nextId++;
  services.push({...ctx.request.body, id});
  ctx.response.status = 204;
});
router.delete('/api/services/:id', async (ctx, next) => {
  const id = Number(ctx.params.id);
  const index = services.findIndex(o => o.id === id);
  if (index === -1) {
    ctx.response.status = 404;
    return;
  }
  services.splice(index, 1);
  ctx.response.status = 204;
});

app.use(router.routes());
app.use(router.allowedMethods());

const port = process.env.PORT || 7070;
const server = http.createServer(app.callback());
server.listen(port);

```


STATE

Давайте сначала поговорим о `state`.

Если отображение загрузки и ошибок должно быть не общим (делать один большой лоадер на всё приложение не очень хорошая идея), то нужно будет сделать флаги `loading`, `error` и для списка, и для формы:

```
{
  serviceList: {
    items: [], loading: false, error: null,
  },
  serviceAdd: {
    item: { name: '', price: '', }, loading: false, error: null,
  }
}
```

В соответствии с этим придётся поправить `initialState` у каждого `Reducer` 'а, маппинги в компонентах.

КОМПОНЕНТЫ

```
// ServiceAdd:
const {item, loading, error} = useSelector(state => state.serviceAdd);
// ServiceList
const {items, loading, error} = useSelector(state => state.serviceList);
// ServiceAddClassBased
ServiceAddClassBased.propTypes = {
  item: PropTypes.shape({
    name: PropTypes.string,
    price: PropTypes.string,
  }).isRequired,
  loading: PropTypes.bool.isRequired,
  error: PropTypes.object,
  onSave: PropTypes.func.isRequired,
  onChange: PropTypes.func.isRequired,
}

const mapStateToProps = (state, ownProps) => {
  const { serviceAdd: {item, loading, error} } = state;
  return { item, loading, error };
}
```

ACTION TYPES

Именовать `Action`'ы при загрузке можно по следующей схеме:

```
export const FETCH_SERVICES_REQUEST = 'FETCH_SERVICES_REQUEST';
export const FETCH_SERVICES_FAILURE = 'FETCH_SERVICES_FAILURE';
export const FETCH_SERVICES_SUCCESS = 'FETCH_SERVICES_SUCCESS';
export const ADD_SERVICE_REQUEST    = 'ADD_SERVICE_REQUEST';
export const ADD_SERVICE_FAILURE    = 'ADD_SERVICE_FAILURE';
export const ADD_SERVICE_SUCCESS    = 'ADD_SERVICE_SUCCESS';
// специально оставили синхронным
export const REMOVE_SERVICE         = 'REMOVE_SERVICE';
export const CHANGE_SERVICE_FIELD   = 'CHANGE_SERVICE_FIELD';
```

ACTION CREATORS

Тогда Action Creator'ы:

```
export const fetchServicesRequest = () => ({
  {type: FETCH_SERVICES_REQUEST};
});

export const fetchServicesFailure = error => ({
  {type: FETCH_SERVICES_FAILURE, payload: {error}};
});

export const fetchServicesSuccess = items => ({
  {type: FETCH_SERVICES_SUCCESS, payload: {items}};
});

// далее - по аналогии (кроме Remove и Change)
export const changeServiceField = (name, value) => ({
  type: CHANGE_SERVICE_FIELD, payload: { name, value, },
});

export const removeService = id => ({
  type: REMOVE_SERVICE, payload: {id}
});
```

SERVICELISTREDUCER

```
const initialState = {
  items: [], loading: false, error: null,
};

export default function serviceListReducer(state = initialState, action) {
  switch (action.type) {
    case FETCH_SERVICES_REQUEST:
      return {...state, loading: true, error: null};
    case FETCH_SERVICES_FAILURE:
      const {error} = action.payload;
      return {...state, loading: false, error};
    case FETCH_SERVICES_SUCCESS:
      const {items} = action.payload;
      return {...state, items, loading: false, error: null};
    // на сервере ничего не удаляем
    case REMOVE_SERVICE:
      const {id} = action.payload;
      return {...state, items: state.items.filter(o => o.id !== id)};
    default:
      return state;
  }
}
```

SERVICEADDREDUCER

```
const initialState = {
  item: { name: '', price: '', },
  loading: false,
  error: null,
};

export default function serviceAddReducer(state = initialState, action) {
  switch (action.type) {
    case ADD_SERVICE_REQUEST:
      return { ...state, loading: true, error: null };
    case ADD_SERVICE_FAILURE:
      const {error} = action.payload;
      return { ...state, loading: false, error };
    case ADD_SERVICE_SUCCESS:
      return { ...initialState };
    case CHANGE_SERVICE_FIELD:
      const { name, value } = action.payload;
      const { item } = state;
      return { ...state, item: {...item, [name]: value } };
    default:
      return state;
  }
}
```

ГДЕ ДЕЛАТЬ FETCH?

Остался самый главный вопрос - где мы должны делать `fetch` и `async/await`?

В `Reducer` 'е нельзя, т.к. это чистая функция.

Можно, конечно, написать эту логику в каждом компоненте и из компонентов `dispatch` 'ить `Action` 'ы.

Это вполне рабочий механизм, но если мы закладываемся на то, что эту логику будет использовать ещё кто-то, почему бы не вынести её в отдельную функцию?

Но тогда этой функции нужен будет `dispatch`.

ГДЕ ДЕЛАТЬ FETCH?

```
// actionCreators.js
export const fetchServices = async dispatch => {
  dispatch(fetchServicesRequest());
  try {
    const response = await fetch(`${process.env.REACT_APP_API_URL}`)
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    const data = await response.json();
    console.log(data);
    dispatch(fetchServicesSuccess(data));
  } catch (e) {
    dispatch(fetchServicesFailure(e.message));
  }
}
```


ГДЕ ДЕЛАТЬ FETCH?

```
// actionCreators.js
export const addService = async (dispatch, name, price) => {
  dispatch(addServiceRequest());
  try {
    const response = await fetch(`${process.env.REACT_APP_API_URL}`, {
      method: 'POST',
      headers: {'Content-Type': 'application/json'},
      body: JSON.stringify({name, price}),
    })
    if (!response.ok) {
      throw new Error(response.statusText);
    }
    dispatch(addServiceSuccess());
  } catch (e) {
    dispatch(addServiceFailure(e.message));
  }
  fetchServices(dispatch);
}
```

FUNCTIONAL COMPONENT

```
function ServiceList(props) {  
  const {items, loading, error} = useSelector(state => state.serviceList);  
  const dispatch = useDispatch();  
  
  useEffect(() => {  
    fetchServices(dispatch)  
  }, [dispatch])  
  
  const handleRemove = id => {  
    dispatch(removeService(id));  
  }  
  
  return (...)  
}
```

Работает-то, оно, конечно, работает, но хотелось бы, чтобы вызовы выглядели одинаково.

Можно, конечно, переделать все вызовы из формата

`dispatch(actionCreator())` в `actionCreator(dispatch)`, но это не лучшая идея.

CLASS BASED COMPONENT

```
class ServiceListClassBased extends Component {
  componentDidMount = () => {
    this.props.fetchServices();
  }
  handleRemove = id => {
    this.props.removeService(id);
  }
  render() {
    const {items, loading, error} = this.props;
    return (...)
  }
}

const mapStateToProps = (state, ownProps) => {
  const {serviceList: {items, loading, error}} = state;
  return {items, loading, error};
};

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    fetchServices: () => fetchServices(dispatch),
    removeService: id => dispatch(removeService(id))
  }
};

export default connect(mapStateToProps, mapDispatchToProps)(ServiceListClassBased);
```



MIDDLEWARE

MIDDLEWARE

Так же, как и во многих других фреймворках/библиотеках, Redux поддерживает концепцию Middleware - промежуточного ПО, позволяющего вклиниться в определённый момент обработки.

В случае Redux - это момент между отправкой `Action` 'а и его попаданием в `Reducer`.

MIDDLEWARE

Вспомним нашу проблему - мы можем отправлять только обычные `Action`'ы, а хотелось бы иметь возможность отправлять функции, как `Action`'ы.

Благодаря концепции Middleware мы можем написать функцию, которая бы принимала на вход наши функции и выполняла бы их.



MIDDLEWARE

Эта идея настолько не нова, что уже есть готовые решения, чем мы и воспользуемся.



REDUX THUNK

REDUX THUNK

Redux Thunk - Middleware для Redux, который расширяет возможности `Store`, позволяя диспатчить функции (в том числе с асинхронными запросами).

```
npm install redux-thunk
```

REDUX THUNK

```
// store/index.js
import { createStore, combineReducers, applyMiddleware } from "redux";
import serviceListReducer from '../reducers/serviceList';
import serviceAddReducer from '../reducers/serviceAdd';
import thunk from "redux-thunk";

const reducer = combineReducers({
  serviceList: serviceListReducer,
  serviceAdd: serviceAddReducer,
});

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

export default store;
```

REDUX THUNK

Исходный код Redux Thunk:

```
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;
```

Т.е. с обычными `Action` 'ами делать ничего не нужно.

<https://github.com/reduxjs/redux-thunk/blob/master/src/index.js>

REDUX THUNK

Переделаем наши функции, чтобы они подходили для этого Middleware:

```
// общая схема:  
const func => (наши аргументы) => (dispatch, getState) => {... наш код...}  
// после чего можем делать:  
disptach(func(args));
```

Redux Thunk будет выполнять код нашей функции, в котором может быть (а может и не быть - в зависимости от условий), `dispatch` `Action` 'ов.

Аргумент `getState` позволяет получить доступ к текущему состоянию (мы говорили о этом, когда рассматривали пример с добавлением).

REDUX THUNK

```
function ServiceList(props) {  
  ...  
  
  useEffect(() => {  
    // было:  
    // fetchServices(dispatch)  
    // стало:  
    dispatch(fetchServices())  
  }, [dispatch])  
  
  ...  
}
```

REDUX THUNK

```
class ServiceListClassBased extends Component {
  ...
}

ServiceListClassBased.propTypes = {
  ...
}

const mapStateToProps = (state, ownProps) => {
  ...
};

const mapDispatchToProps = (dispatch, ownProps) => {
  return {
    // было:
    // fetchServices: () => fetchServices(dispatch),
    // стало:
    fetchServices: () => dispatch(fetchServices()),
    removeService: id => dispatch(removeService(id)),
  }
};
```

REDUX DEVTOOLS

```
import { createStore, combineReducers, applyMiddleware, compose } from 'redux';
import serviceListReducer from '../reducers/serviceList';
import serviceAddReducer from '../reducers/serviceAdd';
import thunk from 'redux-thunk';

const reducer = combineReducers({
  serviceList: serviceListReducer,
  serviceAdd: serviceAddReducer,
});

const composeEnhancers = window.__REDUX_DEVTOOLS_EXTENSION_COMPOSE__ || compose;

const store = createStore(reducer, composeEnhancers(applyMiddleware(thunk)));

export default store;
```



ИТОГИ

Сегодня мы рассмотрели достаточно сложную тему: Redux, а также затронули вопросы использования побочных эффектов.

Как мы уже сказали, Redux нужен не всегда, но в больших проектах используется очень часто. Поэтому вам нужно научиться им пользоваться.

Итоговые исходники к материалам сегодняшней лекции будут размещены в репозитории с кодом к лекциям.



Задавайте вопросы и напишите отзыв о лекции!

КОНСТАНТИН ПОЛЯНСКИЙ

kv.polyanskiy@gmail.com

[@kvpolyanskiy](https://www.instagram.com/kvpolyanskiy)