

М. ПАЦИАНСКИЙ



ОСНОВЫ REDUX

2016



Содержание

Вступление	1.1
Подготовка	1.2
Создание package.json	1.2.1
Первые шаги	1.2.2
ES2015, React HMR	1.2.3
ESLint	1.2.4
Установка зависимостей на автомате	1.2.5
React dev tools	1.2.6
Создание	1.3
Основы Redux (теория)	1.3.1
Точка входа	1.3.2
Настройка Store	1.3.3
Создание Reducer	1.3.4
Присоединение данных (connect)	1.3.5
Комбинирование редьюсеров	1.3.6
Контейнеры и компоненты	1.3.7
Создание actions	1.3.8
Константы	1.3.9
Наводим порядок	1.3.10
Middleware (усилители). Логгер	1.3.11
Асинхронные actions	1.3.12
Взаимодействуем с VK	1.3.13
Заключение	1.4

React Redux [RU tutorial]

Всем доброго времени суток. В данном учебном курсе вы найдете 2 раздела:

1. Подготовка
2. Теория redux и создание веб-приложения по шагам

Если вам интересно, именно выполнение [асинхронных действий](#) и [работа с VK API](#), вы можете прочитать соответствующие главы.

Курс предполагает, что читатель уже знаком с React.js. Если вы не знакомы, рекомендую для начала ознакомиться с курсом [React.js для начинающих](#).

Результатом изучения должно стать умение у читателя разрабатывать приложения с помощью Redux, опираясь строго на однонаправленный поток данных и другие правила данной технологии. Курс охватывает все типовые вопросы (как показать preloader, как отобразить ошибку в результате асинхронного запроса и т.д.), но не затрагивает вопросы серьезной оптимизации сборки webpack'a и написания тестов.

Консультации и платные услуги

Общение по скайпу - 1900 руб/час, минимум - 30 минут.

Создание сервисов с использованием react/redux, поиск проблем с производительностью, помощь в собеседовании разработчиков - цена по запросу.

Пишите на maxfarseer@gmail.com с темой "Консультация React/Redux"

Полезные ссылки

[React.js](#) (EN) - офф.сайт, содержит примеры для изучения

[React.js для начинающих](#) (RU) - курс на русском для тех, кто хочет постичь азы react.js

[Redux](#) (EN) - офф.сайт, который полностью охватывает вопрос изучения redux и содержит отличные примеры.

[Redux](#) (RU) - перевод офф.документации в котором переведено практически все.

[Webpack screencast](#) (RU) - скринкаст от Ильи Кантора по webpack

[Twitter](#) аккаунт создателя redux, и его [бесплатный видео курс](#) (EN)

Мой [Twitter](#) - можете задавать вопросы.

Подготовка

Данная глава является обучающей для людей, которые не в курсе, или хотят освежить и пополнить свою базу знаний, по следующим пунктам:

- создание списка зависимостей проекта
- настройка Webpack
 - создание dev-сервера (Express.js)
 - ES2015 / ES7 (Babel 6)
 - Babel 6 + React
 - React hot reload
 - ESLint

Результатом подготовки, будет [следующий код](#).

Если вам понятен код данного раздела, предлагаю сразу переходить к главе "Создание".

Для всех остальных, я предлагаю за несколько простых шагов настроить удобное рабочее окружение.

Создание package.json

В директории с проектом необходимо выполнить:

```
npm init
```

и ответить на вопросы.

NPM создаст для нашего приложения файл-описание. Он пригодится нам для указания зависимостей и прочих вкусокостей.

Первым делом мы добавим в package.json команду для старта сервера (который создадим на следующем шаге). Для этого необходимо добавить:

```
"scripts": {  
  "start": "node server.js"  
}
```

Итоговый файл выглядит примерно так:

package.json

```
{  
  "name": "redux-ru-tutorial",  
  "version": "1.0.0",  
  "description": "Redux RU tutorial",  
  "main": "index.js",  
  "scripts": {  
    "start": "node server.js"  
  },  
  "author": "Maxim Patsianskiy",  
  "license": "MIT"  
}
```

Теперь если написать `npm start` будет выполнена команда `node server.js`, которая нам понадобится позже для запуска сервера.

Так же, по мере роста вашего приложения, вы будете устанавливать новые пакеты с флагом `--save` или `--save-dev`. Таким образом, когда вы, либо кто-то другой будет разворачивать ваш проект, достаточно будет написать `npm install`, чтобы установить все зависимости.

Первые шаги

Для сборки нашего кода будем использовать [Webpack](#).

```
npm i webpack webpack-dev-middleware webpack-hot-middleware --save-dev
```

Флаг `--save-dev` - добавит пакет `webpack` (и парочку вспомогательных) в список зависимостей нашего проекта.

Теперь необходимо создать конфигурационный файл

webpack.config.js

```
var path = require('path')
var webpack = require('webpack')

module.exports = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    'webpack-hot-middleware/client',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.optimize.OccurrenceOrderPlugin(),
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoErrorsPlugin()
  ]
}
```

Даже для тех, кто не знаком с `webpack`'ом, этот конфиг покажется вполне понятным. В `entry` - указывается откуда `webpack`'у начинать сборку, а в `output` - куда сгенерировать. В `devtool` указываем, что нам нужен `source-map` для отладки кода с парой настроек ([cheap](#), [module](#), [eval](#)).

Интерес пытливого читателя может вызвать строка `'webpack-hot-middleware/client'` ([NPM](#)), которая нужна нам для поддержки *hot-reload*, вместе с одним из плагинов - `webpack.HotModuleReplacementPlugin`

Server

Нам понадобится сервер для разработки, для этого мы можем использовать `webpack-dev-server`, либо быстро развернем свой. В данном руководстве рассматривается второй вариант.

Для начала установим `express`

```
npm i express --save-dev
```

и создадим наш web-сервер на его основе.

`server.js`

```
var webpack = require('webpack')
var webpackDevMiddleware = require('webpack-dev-middleware')
var webpackHotMiddleware = require('webpack-hot-middleware')
var config = require('./webpack.config')

var app = new (require('express'))()
var port = 3000

var compiler = webpack(config)
app.use(webpackDevMiddleware(compiler, { noInfo: true, publicPath: config.output.publicPath }))
app.use(webpackHotMiddleware(compiler))

app.get("/", function(req, res) {
  res.sendFile(__dirname + '/index.html')
})

app.listen(port, function(error) {
  if (error) {
    console.error(error)
  } else {
    console.info("==> Listening on port %s. Open up http://localhost:%s/ in your browser.", port, port)
  }
})
```

Обратите внимание, на строку

```
app.use(webpackHotMiddleware(compiler))
```

На этом шаге добавляется немного магии к нашему серверу, а именно: сервер теперь принимает уведомления, когда главный js скрипт собран и вызывает обновления модулей нашего приложения, в остальном сервер всего навсего отдает нам `index.html`, в котором мы подключаем файл, сгенерированный `webpack`'ом.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Redux [RU]Tutorial</title>
  </head>
  <body>
    <div id="root">
    </div>
    <script src="/static/bundle.js"></script>
  </body>
</html>
```

Добавим точку входа для webpack'a. Мы указали ее в нашем `webpack.config.js` в настройке `entry` - `index.js`

src/index.js

```
document.getElementById('root').innerHTML = 'Привет, я готов.'
```

Проверим?

```
npm start
```

<http://localhost:3000/>

Отлично, кое-что уже завелось. Но если сейчас обновить код в файле `index.js` страница в браузере останется прежней. Хотя при этом в консоли мы увидим, что webpack что-то пересобрал.

Hot Reload

Дело в том, что "модуль" `index.js` не умеет сообщать webpack'у каким образом он хотел бы обновиться. Сейчас, для простоты примера, достаточно добавить строку `module.hot.accept()`, которая сообщает webpack'у следующую информацию: "Я (`index.js`) умею hot-reload сам, для этого просто возьми и обнови меня в сгенерированном файле (`/static/bundle.js`)."

src/index.js

```
document.getElementById('root').innerHTML = 'Привет, я готов!'
module.hot.accept()
```

Перезапустите сервер, обновите страницу браузера.

А теперь поменяйте текст, в `index.js` - он так же обновится на экране браузера. Браузер **не перезагрузит** страницу, как в случае с *live-reload*, а сразу отобразит только нужный кусочек. Это гораздо удобнее!

Конечно, постоянно указывать `module.hot.accept()` не удобно, и не разумно.

Следующий шаг поможет нам избавиться от "ручной" настройки hot-reload для react.js кода.

ES2015, React HMR

Для использования возможностей ES6(2015) и ES7 будем использовать babel. С выходом 6й версии, он стал очень модульным, поэтому не пугайтесь большому количеству зависимостей.

Babel 6

Все начинается с

```
npm install babel-core babel-loader --save-dev
```

Далее нужно поставить пресеты (*предустановки*), которые нам нужны.

```
# Для поддержки ES6/ES2015
npm install babel-preset-es2015 --save-dev

# Для поддержки JSX
npm install babel-preset-react --save-dev

# Для поддержки ES7
npm install babel-preset-stage-0 --save-dev
```

Нам однозначно нужен полифил, чтобы все фичи работали в браузере

```
npm install babel-polyfill --save
```

И немного улучшим время сборки, добавив следующие пакеты

```
npm install babel-runtime --save
npm install babel-plugin-transform-runtime --save-dev
```

Для написания этого небольшого текста про babel 6, я использовал статью [Using ES6 and ES7 in the Browser, with Babel 6 and Webpack](#)

В данный момент, у нас достаточно "пакетов", чтобы писать "современный" код и использовать JSX. Давайте в этом убедимся.

Во-первых, подправим конфиг для webpack'a:

webpack.config.js

```
var path = require('path')
var webpack = require('webpack')

module.exports = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    'webpack-hot-middleware/client',
    'babel-polyfill',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.optimize.OccurrenceOrderPlugin(),
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoErrorsPlugin()
  ],
  module: { //Обновлено
    loaders: [ //добавили babel-loader
      {
        loaders: ['babel-loader'],
        include: [
          path.resolve(__dirname, "src"),
        ],
        test: /\.js$/,
        plugins: ['transform-runtime'],
      }
    ]
  }
}
```

Добавилась запись в секции loaders. Теперь все js файлы в *src* директории будут обрабатываться *babel-loader*’ом, которому мы в свою очередь тоже должны указать настройки. Для этого, нужно создать файл *.babelrc* со следующим содержимым:

```
{
  "presets": ["es2015", "stage-0", "react"] //поддержка ES2015, ES7 и JSX
}
```

Если вы знакомы с *gulp*, то можно провести некую аналогию, между плагинами *gulp* и лодерами (loaders) *webpack*’а. Если мы хотим делать какие-то преобразования с кодом внутри файла, будь то *css*, *js* или картинки - мы используем соответствующий *loader*. Причем создавать дополнительные файлы настроек, как в случае с *babel*, обычно не нужно.

Ок, создадим React компонент, не забыв при этом скачать нужные пакеты:

```
npm i react react-dom --save
```

src/index.js

```
import 'babel-polyfill'
import React from 'react'
import { render } from 'react-dom'
import App from './containers/App'

render(
  <App />,
  document.getElementById('root')
)
```

src/containers/App.js

```
import React, { Component, PropTypes } from 'react'

export default class App extends Component {
  render() {
    return <div>Привет из App</div>
  }
}
```

Перезапускаем сборку (`npm start`).

Весь код на текущий момент выложен в [специальную ветку на Github](#). Можете свериться, если что-то не работает.

React + Hot Reload

Возможно, вам встретится аббревиатура HMR ([hot module replacement](#)), что в принципе более правильно отражает суть, поэтому под hot-reload я подразумеваю именно HMR ;)

Как вы помните из прошлой главы - мы добавили `module.hot.accept()`, для того, чтобы webpack обновлял код из файла *index.js* в сборке без перезагрузки страницы в браузере. Если сейчас попробовать изменить что-то в *App.js* - то в результате ничего не случится, ровно по тем же причинам, что и в предыдущем случае. Что ж, это поправимо и благодаря добрым людям, нам не нужно самим вписывать *assert* функцию. Итак, встречайте (и устанавливайте):

```
npm install react-hot-loader --save-dev
```

Достаточно добавить еще один loader в конфиг и мы получим hot-reload для React компонентов.

```
var path = require('path')
var webpack = require('webpack')

module.exports = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    'webpack-hot-middleware/client',
    'babel-polyfill',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.optimize.OccurrenceOrderPlugin(),
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoErrorsPlugin()
  ],
  module: {
    loaders: [
      {
        loaders: ['react-hot', 'babel-loader'], //добавили loader 'react-hot'
        include: [
          path.resolve(__dirname, "src"),
        ],
        test: /\.js$/,
        plugins: ['transform-runtime'],
      }
    ]
  }
}
```

Перезапускаем сборку и проверяем. Теперь HMR работает и для React компонентов, если же нет - сверьтесь [с исходным кодом данного раздела](#).

Для того, чтобы начать писать код redux-приложения, я основательно рекомендую настроить ESLint, чтобы быстро решать синтаксические ошибки и повысить производительность. Этим мы займемся на следующем шаге.

P.S. В официальном репозитории [React-hot-reloader](#)'а говорится о том, что готовится к выходу [React Transform](#), который станет логическим продолжением текущих решений. (31.01.2015)

P.P.S. [Как говорит](#) создатель библиотеки `react-hot-reload`, нам больше не нужно использовать `webpack.NoErrorsPlugin`, который ранее выполнял следующее: если в сборке были ошибки, он не обновлял файл сборки. Поэтому просто удалите соответствующую строку из секции `plugins` внутри `webpack.config.js`

ESLint

Если вы не знакомы с "линтерами", то вы, наверняка, знакомы с типичным поиском ошибки в стиле `myVariable is undefined` и подобными.

Настроив линтер, вы сможете видеть в консоли много полезной информации: от забытой точки-с-запятой (кстати, не актуально для ES2015), до уведомлений о неиспользуемых переменных. Очень удобно для рефакторинга кода.

Современный ESLint пошел еще дальше. С добавлением собственных правил, вы можете поддерживать единый стиль программирования внутри компании!

Но, довольно теории.

Поставим нужные пакеты:

```
npm i babel-eslint eslint eslint-plugin-react --save-dev
```

Теперь, хотя я и говорил, что файлы `.xxxrc` обычно не нужны, для ESLint все же нужно сделать такой. В нем мы опишем правила для синтаксической проверки (*lint*) кода.

.eslintrc

```
{
  "extends": "eslint:recommended",
  "parser": "babel-eslint",
  "env": {
    "browser": true,
    "node": true
  },
  "plugins": [
    "react"
  ],
  "rules": {
    "no-debugger": 0,
    "no-console": 0,
    "new-cap": 0,
    "strict": 0,
    "no-underscore-dangle": 0,
    "no-use-before-define": 0,
    "eol-last": 0,
    "quotes": [2, "single"],
    "jsx-quotes": [1, "prefer-single"],
    "react/jsx-no-undef": 1,
    "react/jsx-uses-react": 1,
    "react/jsx-uses-vars": 1
  }
}
```

Самое интересное, конечно же, секция `rules`, где:

- 0 - правило выключено
- 1 - правило выдаст предупреждение
- 2 - правило выдаст ошибку

Некоторые правила принимают массив аргументов, например `quotes`. В нашем случае, именно это правило можно прочитать так: Показывай ошибку, если встретишь двойную кавычку.

Список всех правил [eslint-plugin-react](#).

Чтобы ESLint работал в автоматическом режиме, мы будем все так же использовать `webpack`.

Наряду с секцией `loaders`, в `webpack` есть секция... `preloaders` (да-да, `postloaders` тоже есть). Я думаю из названия секций уже все понятно: код обрабатывается "до" и "после" `loaders`. Для ESLint нам подходит `preloaders`.

Итак, поставим нужный ладер:

```
npm i eslint-loader --save-dev
```

Поправим конфиг:

webpack.config.js

```
...
module: {
  preLoaders: [ //добавили ESLint в preloaders
    {
      test: /\.js$/,
      loaders: ['eslint'],
      include: [
        path.resolve(__dirname, "src"),
      ],
    }
  ],
  loaders: [ //все остальное осталось не тронутым
    {
      loaders: ['react-hot', 'babel-loader'],
      include: [
        path.resolve(__dirname, "src"),
      ],
      test: /\.js$/,
      plugins: ['transform-runtime'],
    }
  ]
}
...
```

Здесь и в будущем, я буду использовать `...` - если даю фрагмент(ы) файла, а не весь код целиком. Весь код раздела всегда есть на Github, ссылка указана в конце статьи.

Теперь достаточно перезапустить сборку. Должно получиться следующее:

```
[361] ./~/react/lib/getNodeForCharacterOffset.js 1.66 kB {0} [built]
[362] ./~/react/lib/onlyChild.js 1.21 kB {0} [built]
[363] ./~/react/lib/quoteAttributeValueForBrowser.js 746 bytes {0} [built]
[364] ./~/react/lib/renderSubtreeIntoContainer.js 463 bytes {0} [built]
[365] ./~/strip-ansi/index.js 161 bytes {0} [built]
[366] (webpack)-hot-middleware/client-overlay.js 1.01 kB {0} [built]
[367] (webpack)-hot-middleware/client.js 3.39 kB {0} [built]
[368] (webpack)-hot-middleware/process-update.js 3.27 kB {0} [built]

ERROR in ./src/containers/App.js
/Users/mac/development/local/mylikes/src/containers/App.js
  1:28 error "PropTypes" is defined but never used no-unused-vars

× 1 problem (1 error, 0 warnings)
```

Линтер показывает нам, что в файле `src/containers/App.js` есть неиспользуемая переменная `PropTypes`, хотя она определена. Это действительно так, поэтому давайте поправим код:

`src/Containers/App.js`

```
import React, { Component } from 'react'

export default class App extends Component {
  render() {
    return <div>Привет из App</div>
  }
}
```

Сохранив файл, мы увидим в консоли следующее:

```
webpack building...
webpack built 1008fe82235cd53ecbb7 in 397ms
```

Великолепно! Ошибок нет. На всякий случай добавлю: сборку webpack не нужно было перезапускать. Обычно, сборку нужно перезапускать лишь после изменений в webpack.config.js В остальных случаях, так как у нас настроен "режим наблюдения" - webpack сам перезапустится и сгенерирует новый файл сборки.

Итого: на данный момент мы можем писать ES2015/ES7 код, использовать JSX и не переживать за глупые ошибки, а своевременно править их, используя подсказки ESLint. Webpack автоматически пересобирает наш файл сборки (*/static/bundle.js*), при этом мы используем всю мощь Hot Module Replacement, и если изменим что-либо в js коде react-компонентов - изменения прилетят сразу же в браузер без перезагрузки страницы. Поздравляю, мы готовы с комфортом написать Redux приложение.

[Исходный код данного раздела](#)

Для написания этого раздела, я использовал следующие материалы:

- [Linting in Webpack](#) (ENG, текст)

Установка зависимостей на автомате

Внутри этого раздела я ставил все зависимости с помощью `npm install <ИМЯ_ПАКЕТА>`, но есть более удобный способ - использовать плагин для webpack - [npm-install-webpack-plugin](#)

```
npm install npm-install-webpack-plugin --save-dev
```

Плагин будет анализировать наши файлы на предмет зависимостей и устанавливать новые пакеты, если обнаружится неизвестная зависимость. Главное, не забывайте удалять лишние пакеты, если будете экспериментировать и пробовать разные.

.npmrc

```
save=true  
save-exact=true
```

webpack.config.js

```
...  
//добавьте новую зависимость в начале конфига  
var NpmInstallPlugin = require('npm-install-webpack-plugin');  
...  
//добавьте плагин в секцию плагинов  
plugins: [  
  new webpack.optimize.OccurenceOrderPlugin(),  
  new webpack.HotModuleReplacementPlugin(),  
  new NpmInstallPlugin() // <--  
],  
...  
...
```

Далее в руководстве, я все равно буду писать `npm install`, так как это визуально дает хорошее представление о том, какие зависимости нам нужны. Если вы поставили и настроили `npm-install-webpack-plugin`, то можете не беспокоиться и пропускать эти строки.

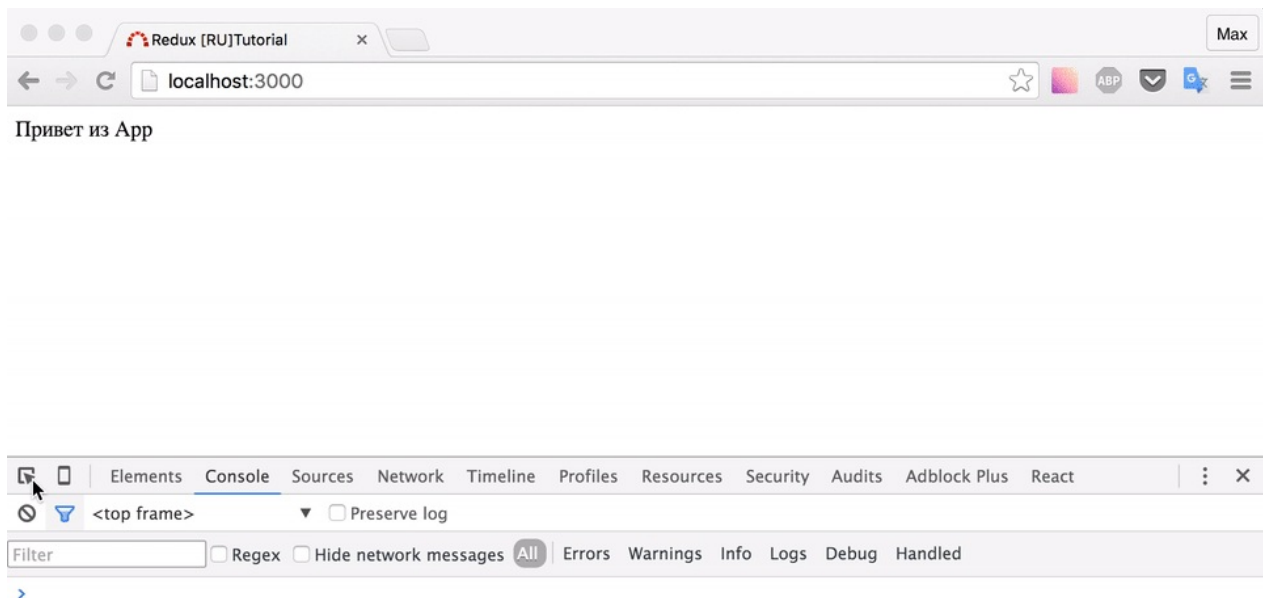
React dev tools

Если у вас не установлено дополнение для консоли хрома (ровно как и сам Google Chrome), рекомендую вам это сделать, так как иногда мы будем обращаться к этим инструментам.

[React dev tools](#) в магазине расширений.

Если вы используете возможность консоли хрома - **\$0**, то вероятно вам понравится и возможность **\$r**.

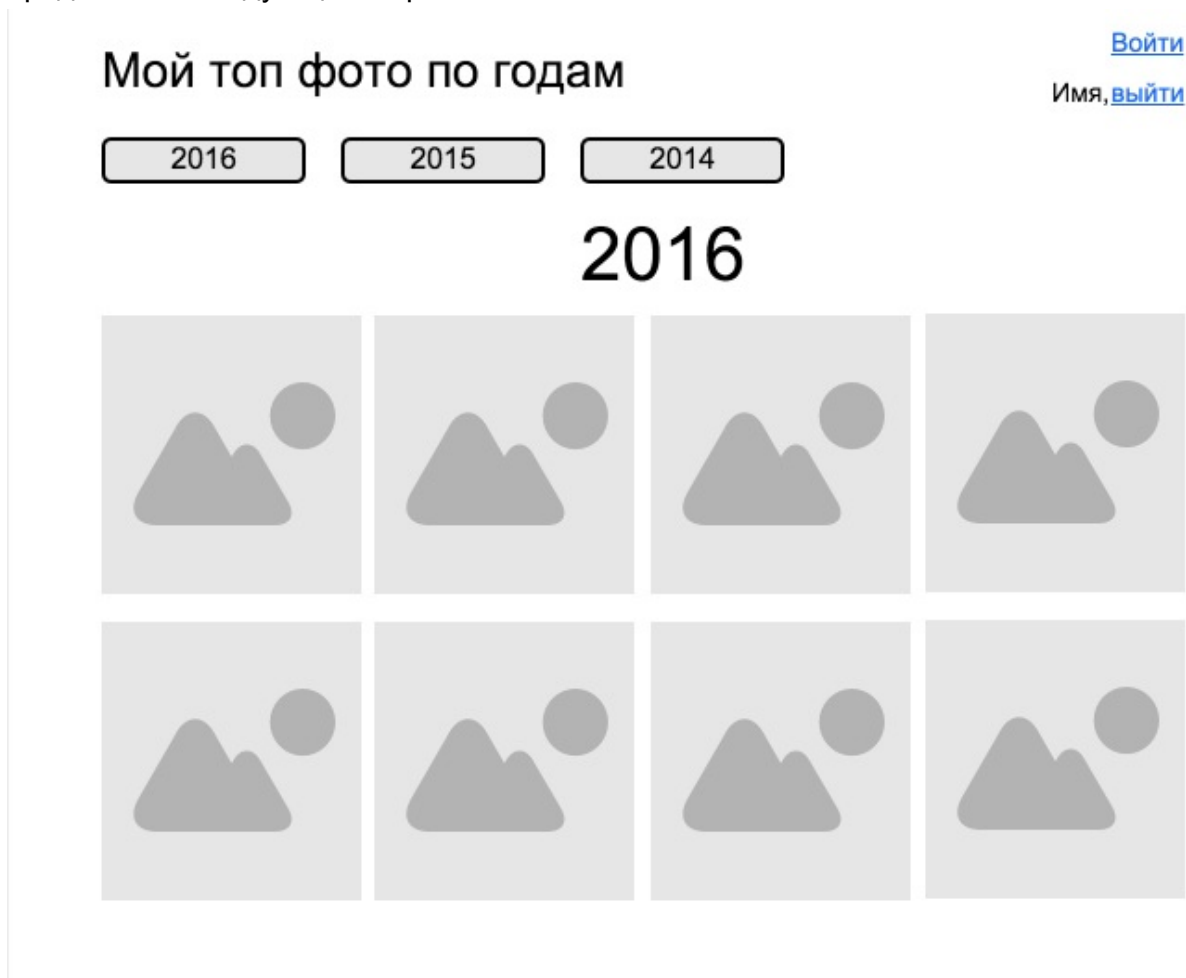
Как работает **\$r**? (ниже *gif*-анимация)



Для тех, кто читает в формате книги: необходимо кликнуть на компонент в консоли хрома, на вкладке *React*, далее переместиться на вкладку *Console* и написать **\$r**. В таком случае, в консоли компонент будет представлен в своей нативной js-реализации.

Создание

Я предлагаю по шагам создать одностраничное приложение, с минимумом функций, которое после логина и подтверждения прав доступа к фото, будет выдавать топ ваших "залайканных" фото в порядке убывания. Схематично, приложение можно представить следующим образом:



Прежде чем описывать структуру, давайте в общих чертах взглянем на Redux.

Redux-приложение это:

- состояние (state) приложения в одном месте
- однонаправленный поток данных

Redux вдохновлен Flux методологией и языком программирования Elm

Под капотом, Redux использует слабо документированную фичу реакта - **context**, которая, к слову, до сих пор является *unstable*, и может быть изменена/удалена. К счастью, этого не происходит и вряд ли произойдет.

Файлы и папки:

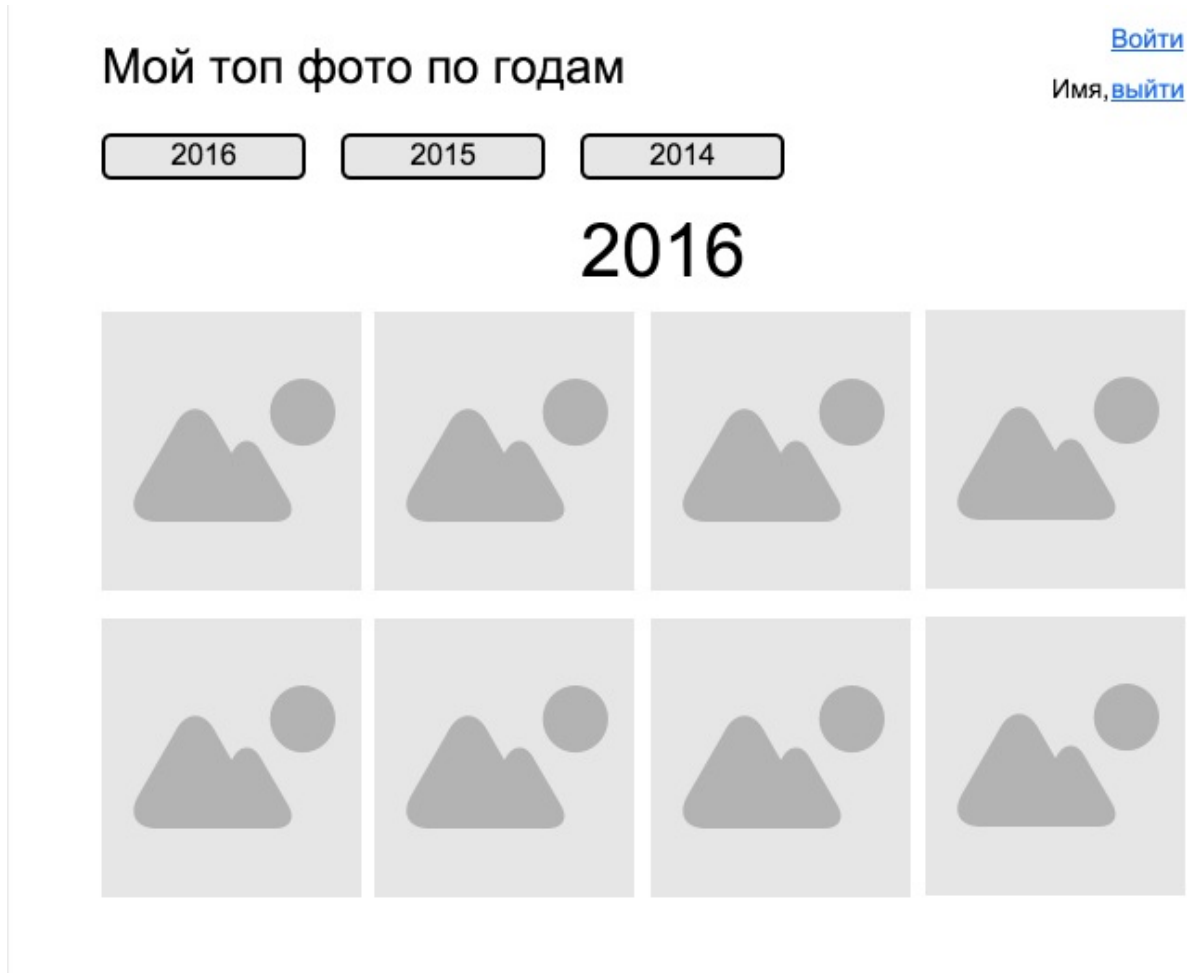
Изначально наше приложение в файловом менеджере должно выглядеть так:

```
+-- src
|   +-- actions
|   +-- components
|   +-- constants
|   +-- containers
|   +-- reducers
|   +-- index.js
+-- index.html
+-- package.json
+-- server.js
+-- webpack.config.js
```


Основы Redux (теория)

Курс рассчитан на создание приложения по шагам, а это значит максимум практики и минимум теории. Тот самый минимум, перед вами.

Давайте еще раз взглянем на схему нашего приложения:



В шапке слева заголовок и три кнопки выбора года. Ниже - фото соответствующего года, отсортированное по количеству лайков.

В шапке справа - ссылка войти/выйти.

Представим, как должны выглядеть данные для такой страницы:

```
app: {
  page: {
    year: 2016,
    photos: [photo, photo, photo...]
  },
  user: {
    name: 'Имя',
    ...
  }
}
```

Поздравляю вас, мы только что описали как должно выглядеть состояние (**state**) нашего приложения.

За содержание всего состояния нашего приложения, отвечает объект **Store**. Как уже не раз упоминалось - это обычный объект. Важно, что в отличии от Flux, в Redux только **один** объект Store.

Не хочется оставлять вас надолго без практики, поэтому процесс создания store и немного подробностей про него я аккуратно вплету в следующие главы, а пока достаточно того, что: *store*, "объединяет" редьюсер (*reducer*) и действия (*actions*), а так же имеет несколько чрезвычайно полезных методов, например:

- `getState()` - позволяет получить состояние приложения;
- `dispatch(actions)` - позволяет обновлять состояния, путем вызова действия;
- `subscribe(listener)` - регистрирует слушателей

Actions

Actions описывают действия.

Actions - это простой объект. Обязательное поле - **type**. Так же, если следовать [соглашению](#), все данные, которые передаются вместе с действием, кладут внутрь свойства `payload`. Таким образом, для нашего приложения, мы можем составить, например такую пару actions:

```
{
  type: 'ЗАГРУЗИ_ФОТО',
  payload: 2016 //год
}
```

```
{
  type: 'ФОТО_ЗАГРУЖЕНЫ_УСПЕШНО',
  payload: [массив фото]
}
```

Чтобы вызвать actions, мы должны написать функцию, которая в рамках Flux/Redux называется - *ActionsCreator* (создатель действия), но перед этим стоит принять во внимание, что обычно тип действия, описывают как константу. Например, константы вашего проекта:

```
const GET_PHOTO_REQUEST = 'GET_PHOTO_REQUEST'
const GET_PHOTO_SUCCESS = 'GET_PHOTO_SUCCESS'
```

Возникает вопрос, зачем? В маленьких проектах - незачем. В больших - это удобно. Пока, *просто запомните*.

Вернемся, к ActionsCreator, один из наших "создателей действий", выглядел бы так:

```
function getPhotos(year) {
  return {
    type: GET_PHOTOS,
    payload: year
  }
}
```

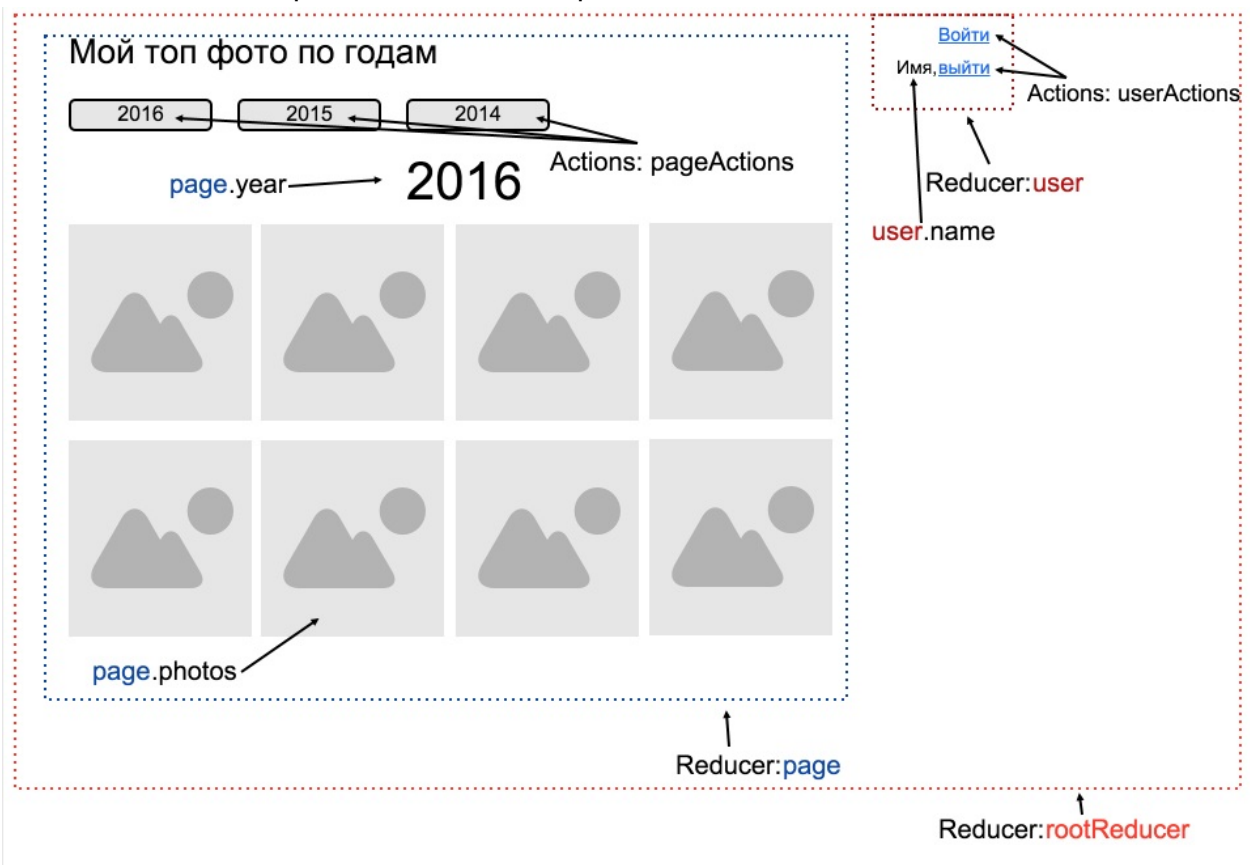
Итого: actions сообщает нашему приложению - "Эй, что-то произошло! И я знаю, что именно!"

Reducer

"Actions описывает факт, что что-то произошло, но не указывает, как состояние приложения должно измениться в ответ, это работа для Reducer'a" - (офф. документация)

Наше приложение не нуждается в нескольких редьюсерах, но крайне необходимо познакомить читателя с **reducer composition**, так как это фундаментальный шаблон построения redux приложений: **мы разбиваем наше глобальное состояние на кусочки, за каждый кусочек отвечает свой reducer**. Кусочки объединяются в Корневом Редьюсере (rootReducer).

Схематично, наше приложение можно представить так:



Так как у нас есть reducer'ы `page` и `user`, можно представить следующий диалог:

```
pageActions: Пришло 123 фото
Reducer (page): Ок, нужно положить эти 123 фото в page.photos
```

А на js выглядело бы так:

```
function page(state = initialState, action) {
  switch (action.type) {
    case GET_PHOTO_SUCCESS:
      return Object.assign({}, state, {
        photos: action.payload
      })
    default:
      return state
  }
}
```

Обратите внимание, мы не **мутировали** наш state, мы создали новый state. Это важно. Крайне важно. В редьюсере, мы всегда должны возвращать новый объект, а не измененный предыдущий.

На практике, я буду использовать [object spread syntax](#), поэтому предыдущую функцию с `Object.assign` можно переписать следующим образом:

```
function page(state = initialState, action) {
  switch (action.type) {
    case GET_PHOTO_SUCCESS:
      return {...state, photos: action.payload} //Object spread syntax
    default:
      return state
  }
}
```

Объект, который мы возвращаем в редьюсере, далее с помощью функции `connect`, превратится в свойства для компонентов. Таким образом, если продолжить пример с фото, то можно написать такой псевдо-код:

```
<Page photos={reducerPage.photos} />
```

Благодаря этому, внутри компонента `<Page />`, мы сможем получить фото, как `this.props.photo`

Я постарался очень кратко дать самую важную теорию.

Если что-то осталось не понятным, не переживайте, на практике мы все закрепим и тогда все встанет на свои места.

Точка входа

Подтянем Redux и react-redux в наш проект:

```
npm i redux react-redux --save
```

Как уже было описано в разделе "подготовка", точка входа в наше приложение - `src/index.js`

Обновим его содержание:

src/index.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import App from './containers/App'

const store = createStore( () => {}, {} ) //WAT ;)

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

Итак, первый компонент из мира Redux - `<Provider />` ([\[EN\] документация](#)).

Благодаря этому компоненту, мы сможем получать необходимые данные из store нашего приложения, если воспользуемся вспомогательной функцией `connect`, речь о которой пойдет далее. Сейчас нам и получать нечего, так как store у нас - пустой объект.

Давайте подробнее посмотрим на строку:

```
const store = createStore( () => {}, {} )
```

Во-первых, если вам трудно читать ES2015 код, то переводите его в привычный ES5, с помощью [babel-playground](#).

Во-вторых, давайте взглянем на документацию метода `createStore`: принимает один обязательный аргумент (функцию `reducer`) и парочку не обязательных (начальное состояние и "усилители").

Теперь переведем то, что мы написали, когда присваивали `store`: Возьми пустую анонимную функцию в качестве редьюсера и пустой объект в качестве начального состояния. Если коротко: возьми ничего и "ничего" не делай.

Предлагаю вынести создание `store` в отдельный файл, для того, чтобы добавить возможность HMR и для более удобной работы с `reducer`'ом и усилителями (`enhancers`).

`src/index.js`

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import App from './containers/App'
import configureStore from './store/configureStore'

const store = configureStore()

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

Усилители - это `middleware` функции. Если читатель знаком с [express.js](#), то он знаком с усилителями в `redux`. Для остальных: типичный усилитель - логгер (`logger`), который просто пишет в консоль все что происходит с наблюдаемым объектом.

Если посмотреть в консоль, `webpack` все так же усердно работает и сообщает об ошибке: нет файла `configureStore...` Пора создать, а текущий код можно взять [здесь](#).

Настройка Store

Для начала, повторим то, что нам уже известно про store, и быть может добавим чуть-чуть нового. Итак:

Store хранит состояние приложения. Единственный путь изменить store - это отправить действие (**dispatch action**).

Store - это не класс. Это обычный объект с несколькими методами, а именно:

- `getState()`
- `dispatch(action)`
- `subscribe(listener)`
- `replaceReducer(nextReducer)`

Создадим функцию конфигурации store

store/configureStore.js

```
import { createStore, applyMiddleware } from 'redux'
import rootReducer from '../reducers'

export default function configureStore(initialState) {
  const store = createStore(rootReducer, initialState)
  return store
}
```

Ничего необычного, строго по [документации](#): передаем *rootReducer* в только что импортированную функцию *createStore*.

В Redux версии 2.x.x мы должны явно указать, что reducers поддерживают hot reload. Сделать это достаточно просто. Если взглянуть в начало кода, видно, что мы импортируем так называемый корневой редьюсер (*rootReducer*), который по сути и отражает все состояние нашего приложения. Теперь посмотрим еще выше по tutorialu - ага, у store есть подходящая функция - `replaceReducer`

Теперь взяв за основу [отличный видео скринкаст про Webpack](#), мы знаем, что hot reload ожидает от нас функции accept. Нуаля, пора вносить правки.

store/configureStore.js


```
import { createStore } from 'redux'
import rootReducer from '../reducers'

export default function configureStore(initialState) {
  const store = createStore(rootReducer, initialState)

  if (module.hot) {
    module.hot.accept('../reducers', () => {
      const nextRootReducer = require('../reducers')
      store.replaceReducer(nextRootReducer)
    })
  }

  return store
}
```

К сожалению, **наш код** до сих пор не работает. Webpack ругается на отсутствующий reducer. Давайте исправим это, и я обещаю, наконец-то можно будет посмотреть на результат в браузере.

Создание Reducer

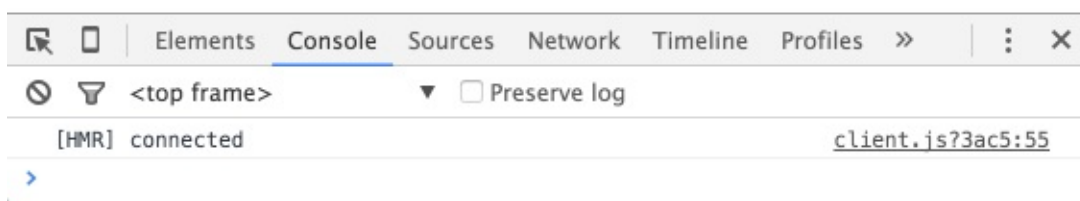
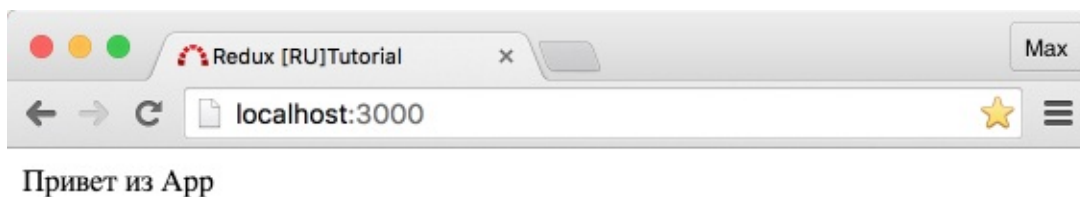
По-моему, в этом коде нечего даже комментировать!

src/reducers/index.js

```
const initialState = {
  user: 'Unknown User'
};

export default function userstate(state = initialState) {
  return state;
}
```

Пора открыть браузер и посмотреть в консоль разработчика. Если там есть предупреждения/ошибки - значит, где-то HMR не сработал. Достаточно обновить браузер. Все-таки с последнего раза мы создали достаточно файлов, чтобы реакт/вебпак растерял долю своей магии и попросил бы нас обновить страницу.



Все ок, но ничего интересного. В следующей главе добавим вывод имени юзера.

Связывание данных из store с компонентами приложения

В разделе "Точка входа" шла речь о некой функции *connect*, которая поможет нам получить в качестве props для компонента `<App />` данные из store. Добавим ее:

src/containers/App.js

```
import React, { Component } from 'react'
import { connect } from 'react-redux'

class App extends Component {
  render() {
    return <div>Привет из App, { this.props.user }!</div>
  }
}

function mapStateToProps (state) {
  return {
    user: state.user
  }
}

export default connect(mapStateToProps)(App)
```

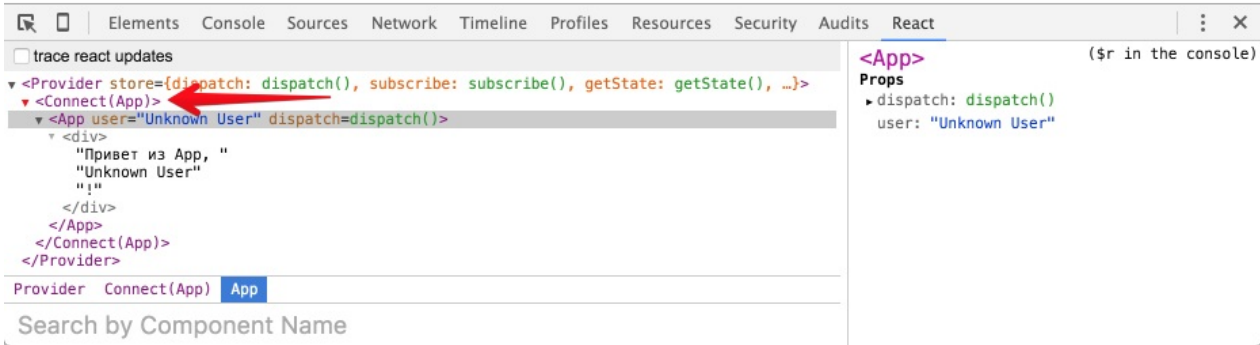
Назначение функции *connect* вытекает из названия: **подключи** React компонент к Redux store.

Результат работы функции *connect* - новый присоединенный компонент, который оборачивает переданный компонент.

У нас был компонент `<App />`, а на выходе получился `<Connected(App)>`. В этом не трудно убедиться, если взглянуть в react dev tools.



Привет из App, Unknown User!



Взгляните на правую часть скриншота, и вы увидите, что в свойствах (*props*) нашего компонента `<App />` теперь есть метод `redux` store - **dispatch**, и объект (в нашем случае, пока что строка) `user`. Это так же результат работы функции `connect`.

Давайте еще поиграемся с простым примером. Для начала изменим набор данных:

`src/reducers/index.js`

```
const initialState = {
  name: 'Василий',
  surname: 'Реактов',
  age: 27
}

export default function userstate(state = initialState) {
  return state
}
```

затем подкрутим компонент:

`src/containers/App.js`

```
import React, { Component } from 'react'
import { connect } from 'react-redux'

class App extends Component {
  render() {
    const { name, surname, age } = this.props.user
    return <div>
      <p>Привет из App, {name} {surname}!</p>
      <p>Тебе уже {age} ?</p>
    </div>
  }
}

function mapStateToProps (state) {
  return {
    user: state
  }
}

export default connect(mapStateToProps)(App)
```

Все работает ровно так, как мы указали: в объект user "подключилось" все состояние нашего приложения state, которое сейчас очень простое и описано в *src/reducer/index.js*.

Прежде чем мы перейдем к созданию actions и взаимодействию пользователя со страницей, давайте поговорим о комбинировании редьюсеров (*combineReducers*) и создадим реальную структуру нашего будущего приложения.

[Исходный код](#) на текущий момент.

P.S. если вы переживаете, что HMR в данный момент не работает, обратите внимание на данный [комментарий](#) (это не обязательно, а только для тех, кто столкнулся с проблемой).

Полезные ссылки:

- [connect](#) (офф.документация)

Комбинирование редьюсеров

Зачем? Когда наше приложение разрастается, хочется еще больше модульности, чтобы каждый кусочек кода отвечал за конкретную часть. Так же и с редьюсерами, мы можем разбить наш главный редьюсер на несколько более мелких, и с помощью `combineReducers` из пакета `redux` собрать их воедино. Причем, абсолютно никакой магии, `combineReducers` просто возвращает "составной" редьюсер.

Для нашего приложения, можно выделить следующие reducer'ы (согласно [схеме](#)):

- user
- page

Создадим их:

src/reducers/user.js

```
const initialState = {
  name: 'АНОНИМ'
}

export default function user(state = initialState) {
  return state
}
```

src/reducers/page.js

```
const initialState = {
  year: 2016,
  photos: []
}

export default function page(state = initialState) {
  return state
}
```

Обновим точку входа для редьюсеров:

src/reducers/index.js

```
import { combineReducers } from 'redux'
import page from './page'
import user from './user'

export default combineReducers({
  page,
  user
})
```

Обратите внимание, что структура объекта, которым можно описать все состояние нашего приложения - не изменилась. Все осталось также.

```
{
  user: {
    name: 'Аноним'
  }
  page: {
    year: 2016,
    photos: []
  }
}
```

Тем не менее, в браузере у нас нерабочее приложение. В чем же проблема?

Ответ кроется в работе функции *connect* и в функции *mapStateToProps* из нашего файла *App.js*. Сейчас у нас там написано следующее:

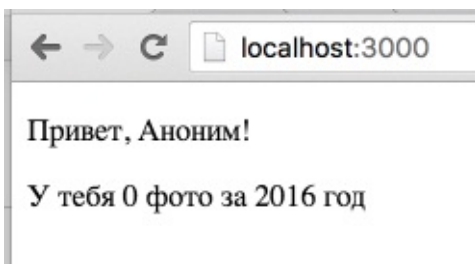
```
function mapStateToProps (state) {
  return {
    user: state
  }
}
```

Что можно перевести так: возьми полностью "стейт" приложения и присоедини его в переменную *user*, дабы она была доступна из компонента *App.js* как `this.props.user`

И действительно, если взглянуть в dev tools, все так и есть:



Здесь, я предложу простую задачку на понимание происходящего. Измените компонент `App.js` и функцию `mapStateToProps` так, чтобы получить следующую картину:



Ответ:

`src/containers/App.js`


```
import React, { Component } from 'react'
import { connect } from 'react-redux'

class App extends Component {
  render() {
    const { name } = this.props.user // (1)
    const { year, photos } = this.props.page // (2)
    return <div>
      <p>Привет, {name}!</p>
      <p>У тебя {photos.length} фото за {year} год</p>
    </div>
  }
}

function mapStateToProps (state) {
  return {
    user: state.user, // (1)
    page: state.page // (2)
  }
}

export default connect(mapStateToProps)(App)
```

Сносками (1) и (2) я пометил связь.

Супер, сейчас у нас в *user* - попадет все из нашего приложения, что будет связано с пользователем, а в *page* - попадет все что связано с отображением соответствующего блока (год и массив фото).

Второй раз за главу, возникает вопрос - зачем? Ответ прежний: модульность, меньший объем кода в каждом файле и лучшая читаемость. А **главное**, мы так добьемся меньшего количества ненужных обновлений, представьте: пользователь кликнул на кнопку "2015", и обновился блок *page*, при этом блок *user* остался **бы** не тронутым вообще, если **бы** он являлся отдельным компонентом.

Нам ничего не мешает исправить это. Продолжим в следующей главе.

[Исходный код](#) на текущий момент.

Полезные ссылки:

- [combineReducers](#) (офф. документация)

Контейнеры и компоненты

Прежде чем мы разобьем App.js на компоненты User.js и Page.js хотелось бы отметить про разделение на "компоненты" и "контейнеры", иначе называемые: "глупые" и "умные" компоненты, "Presentational" и "Container" и быть может как-то еще.

Позволю себе в очередной раз прибегнуть к [офф. документации](#) и перевести таблицу различий, которая отлично и кратко отражает суть.

	Компонент (глупый)	Контейнер (умный)
Цель	Как это выглядит (разметка, стили)	Как это работает (получение данных, обновление состояния)
Осведомлен о Redux	Нет	Да
Для считывания данных	Читает данные из props	Подписан на Redux state (состояние)
Для изменения данных	Вызывает callback из props	Отправляет (<i>dispatch</i>) Redux действие (actions)
Пишутся	Вручную	Обычно, генерируются Redux

Магия таблиц обычно проявляется не сразу. Если переписать наше приложение, а потом взглянуть сюда еще раз - многое станет гораздо яснее. Предлагаю так и поступить. Поехали!

Создадим компоненты.

src/components/User.js

```
import React, { PropTypes, Component } from 'react'

export default class User extends Component {
  render() {
    const { name } = this.props
    return <div>
      <p>Привет, {name}!</p>
    </div>
  }
}

User.propTypes = {
  name: PropTypes.string.isRequired
}
```

src/components/Page.js

```
import React, { PropTypes, Component } from 'react'

export default class Page extends Component {
  render() {
    const { year, photos } = this.props
    return <div>
      <p>У тебя {photos.length} фото за {year} год</p>
    </div>
  }
}

Page.propTypes = {
  year: PropTypes.number.isRequired,
  photos: PropTypes.array.isRequired
}
```

Наш файл App.js уже практически и есть container, он даже лежит в соответствующей папке. Изменим-с...

src/containers/App.js

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import User from '../components/User'
import Page from '../components/Page'

class App extends Component {
  render() {
    const { user, page } = this.props
    return <div>
      <User name={user.name} />
      <Page photos={page.photos} year={page.year} />
    </div>
  }
}

function mapStateToProps (state) {
  return {
    user: state.user,
    page: state.page
  }
}

export default connect(mapStateToProps)(App)
```

Теперь можно обновлять компоненты *Page* и *User* независимо друг от друга. Чем мы и займемся в следующей главе, изучая *actions*.

Создание actions

Наконец-то мы подходим к вопросу взаимодействия с пользователем приложения. Практически любое действие пользователя в интерфейсе = **отправка действия** (*dispatch actions*)

По клику на кнопку года, наше приложение:

- устанавливает заголовок
- загружает фото этого года

Сейчас предлагаю рассмотреть установку заголовка. Загрузка фото требует выполнения асинхронного запроса, а чтобы добраться до этого, мы должны рассмотреть несколько интересных вещей. К тому же, установка заголовка отлично показывает на простом примере, как *вращаются* данные внутри redux-приложения, а именно:

1. Приложение получило изначальное состояние (*initial state*)
2. Пользователь нажав кнопку, отправил действие (*dispatch action*)
3. Соответствующий редьюсер обновил часть приложения, в согласии с тем, что узнал от действия.
4. Приложение изменилось и теперь отражает новое состояние.
5. ... (все повторяется по кругу, с пункта 2)

Это и есть **однонаправленный** поток данных.

Создадим page action:

src/actions/PageActions.js

```
export function setYear(year) {  
  
  return {  
    type: 'SET_YEAR',  
    payload: year  
  }  
  
}
```

Напоминаю, что поля `type` и `payload` - всего лишь "негласное" соглашение. Немного об этом, можно почитать на английском [тут](#).

Поправим редьюсер page:

src/reducers/page.js

```
const initialState = {
  year: 2016,
  photos: []
}

export default function page(state = initialState, action) {

  switch (action.type) {
    case 'SET_YEAR':
      return { ...state, year: action.payload }

    default:
      return state;
  }
}
```

Обратите внимание, в аргументах у функции page указан второй аргумент - *action*. Это стандартные аргументы redux reducer'a. Благодаря этому, мы можем легко обрабатывать различные действия по их типу, попадая в нужную секцию case оператора switch.

Так же обратите внимание, что мы не **изменили** объект state, а вернули **новый** с полем *year* равным *action.payload* (а значит годом, выбранным пользователем).

Добавляем вызов actions из компонентов

У нас есть *action*, и есть *reducer* готовый изменить *state* приложения (да, я нарочно пишу иногда эти слова по-английски). Но наш компонент не знает как обратиться к необходимому действию.

Согласно таблице из прошлого раздела: для изменения данных, наш *компонент* Page.js, должен вызывать *callback* из *this.props*, а наш контейнер* App.js - отправлять действие (*dispatch action*).

* я говорю, контейнер, хотя правильнее называть контейнером `<Connect(App) />`, но так как он генерируется функцией *connect* на основе App.js, считаю это допустимым.

Из документации функции *connect*, нам так же становится ясно, что с помощью этой функции мы можем не только подписаться на обновления данных, но и "прокинуть" наши *actions* в контейнер.

`connect`, первым аргументом принимает "маппинг" (соответствие) state к props, а вторым маппинг dispatch к props. Как бы дико это не звучало, на практике это значит, что нам достаточно передать второй аргумент.

Исправим App.js

src/containers/App.js

```
import React, { Component } from 'react'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'
import User from '../components/User'
import Page from '../components/Page'
import * as pageActions from '../actions/PageActions'

class App extends Component {
  render() {
    const { user, page } = this.props
    const { setYear } = this.props.pageActions

    return <div>
      <User name={user.name} />
      <Page photos={page.photos} year={page.year} setYear={setYear} />
    </div>
  }
}

function mapStateToProps(state) {
  return {
    user: state.user,
    page: state.page
  }
}

function mapDispatchToProps(dispatch) {
  return {
    pageActions: bindActionCreators(pageActions, dispatch)
  }
}

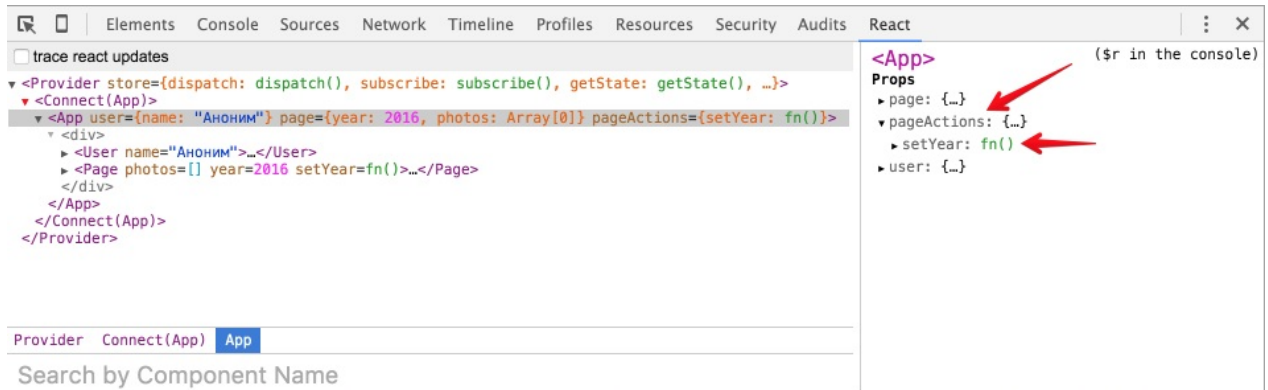
export default connect(mapStateToProps, mapDispatchToProps)(App)
```

Начнем с разбора `mapDispatchToProps`. Внутри функции мы использовали вспомогательную функцию из redux - `bindActionCreators` ([офф. документация](#), которая позволила вызывать `setYear`, если выразиться просто с некоторыми допущениями как:

```
store.dispatch({
  type: 'SET_YEAR'
  payload: 2016
})
```

Тем самым необходимое изменение прослушивается в `redux store`, и в нашем редьюсере `Page` соответственно.

Следовательно, после выполнения `connect(mapStateToProps, mapDispatchToProps)(App)`, мы получили в `App.js` новые свойства (*props*), что наглядно демонстрирует вкладка "React" в `chrome dev tools`.



Добавив `setYear` в свойства `Page.js`, не составит труда использовать необходимый action из компонента, который по прежнему знать ничего не знает о `redux`.

UPDATE [18.03.16]: свойство `innerText` приведенное в коде ниже - нестандартное, поэтому с ним могут возникнуть проблемы в некоторых браузерах. Вместо него, вы можете использовать - `textContent`.

`src/components/Page.js`


```
import React, { PropTypes, Component } from 'react'

export default class Page extends Component {
  onYearBtnClick(e) {
    this.props.setYear(+e.target.innerText)
  }
  render() {
    const { year, photos } = this.props
    return <div>
      <p>
        <button onClick={::this.onYearBtnClick}>2016</button>
        <button onClick={::this.onYearBtnClick}>2015</button>
        <button onClick={::this.onYearBtnClick}>2014</button>
      </p>
      <h3>{year} год</h3>
      <p>У тебя {photos.length} фото.</p>
    </div>
  }
}

Page.propTypes = {
  year: PropTypes.number.isRequired,
  photos: PropTypes.array.isRequired,
  setYear: PropTypes.func.isRequired
}
```

Собственно, код компонента Page по прежнему очень простой. Строка

`::this.onYearBtnClick === this.onYearBtnClick.bind(this)`, и нужна так как React с версии 0.14.x не привязывает `this` к компоненту.

Использование двойного двоеточия - это возможность ES7 ([experimental](#)), которая доступна в babel с настройкой `stage=0` (для тех кто писал код, начиная с раздела "Подготовка" - все уже настроено, смотри файл `.babelrc`)

Глава выдалась достаточно длинной, а хуже всего, что мы написали "кипу" кода, всего лишь для обновления цифры в заголовке. Где профит, как говорится?

Профит обнаружится дальше, когда ваше приложение разрастется. Когда его будет необходимо поддерживать и добавлять новые фичи. За счет **однаправленного** потока данных (юзер кликнул - действие вызвалось - редьюсер изменил состояние - компонент отрисовал изменения) даже в приложении, написанном давно, у вас получится очень быстро разобраться и внести необходимые обновления, которые требует бизнес. К тому же, такой подход отлично работает и для командной работы.

[Исходный код](#)

Константы

Если вынести все page actions в отдельный файл с константами, то в будущем нам удобнее будет писать тесты/работать в команде/поддерживать код. К тому же, таким образом мы не будем отходить от "соглашений" принятых в разработке Flux/Redux приложений.

src/constants/Page.js

```
export const SET_YEAR = 'SET_YEAR'
```

Подключим константу в редьюсер Page и в PageActions

src/reducers/page.js

```
import { SET_YEAR } from '../constants/Page'

const initialState = {
  year: 2016,
  photos: []
}

export default function page(state = initialState, action) {

  switch (action.type) {
    case SET_YEAR: //не забудьте обновить строку на константу
      return { ...state, year: action.payload }

    default:
      return state;
  }
}
```

src/actions/PageActions.js

```
import { SET_YEAR } from '../constants/Page'

export function setYear(year) {

  return {
    type: SET_YEAR, //аналогично, теперь используем константу
    payload: year
  }
}
```

В дальнейшем мы еще добавим констант, не только для компонента `<Page />`, но и для компонента `<User />`, которые мы так же будем объявлять в отдельном файле.

Наводим порядок

Данная глава является своего рода "перекуром". Она затрагивает вопросы стилей и верстки приложения.

Autoprefixer и стили

Добавим в webpack возможность обрабатывать стили, заодно сразу накинув на них возможности autoprefixer'a.

```
npm install style-loader css-loader postcss-loader autoprefixer precss --save-dev
```

P.S. для тех кто пользуется автоматическим способом добавления зависимостей: так как мы добавляем зависимости в `webpack.config` - *npm-install-plugin* не может подтянуть их автоматически.

webpack.config.js

```
var path = require('path')
var webpack = require('webpack')
var NpmInstallPlugin = require('npm-install-webpack-plugin')
var autoprefixer = require('autoprefixer');
var precss = require('precss');

module.exports = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    'webpack-hot-middleware/client',
    'babel-polyfill',
    './src/index'
  ],
  output: {
    path: path.join(__dirname, 'dist'),
    filename: 'bundle.js',
    publicPath: '/static/'
  },
  plugins: [
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.HotModuleReplacementPlugin(),
    new NpmInstallPlugin()
  ],
  module: {
    preLoaders: [
      {
        test: /\.js$/,
        loaders: ['eslint'],
        include: [
          path.resolve(__dirname, "src"),
        ],
      }
    ],
    loaders: [
      {
        loaders: ['react-hot', 'babel-loader'],
        include: [
          path.resolve(__dirname, "src"),
        ],
        test: /\.js$/,
        plugins: ['transform-runtime'],
      },
      {
        test: /\.css$/,
        loader: "style-loader!css-loader!postcss-loader"
      }
    ]
  },
  postcss: function () {
    return [autoprefixer, precss];
  }
}
```

Не забудьте подключить главный файл стилей для всего приложения:

src/index.js

```
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import App from './containers/App'
import './styles/app.css' // <-- импорт стилей
import configureStore from './store/configureStore'

const store = configureStore()

render(
  <Provider store={store}>
    <div className='app'> {/* обернули все в .app */}
      <App />
    </div>
  </Provider>,
  document.getElementById('root')
)
```

Возможно (если вы копируете код, либо тоже любите одинарные кавычки), webpack выдаст ошибку от ESLint. Для использования одинарной кавычки, исправьте правило

```
"jsx-quotes": [1, "prefer-single"]
```

 в файлике *.eslintrc*

Верстка

Верстка и стили не являются темой нашего обучения, поэтому можете просто взглянуть [на исходный код](#), либо сделать как вам хочется.

В реальном приложении, имеет смысл стили для компонентов импортировать в коде самих компонентов, что даст очень большие удобства для переиспользования целых блоков, включая оформление.

Middleware (Усилители). Логгер

Прежде чем мы сможем создавать асинхронные действия, поговорим об усилителях и напишем, обещанный ранее усилитель - *логгер*.

Представляйте усилитель, как нечто стороннее, добавляющее функционал для нашего `store`.

Усилители, это *middleware*. Суть *middleware* функций, взять входные данные, добавить что-то и **передать дальше**.

Например: есть конвейер, по которому движется пальто. На конвейере работают Зина и Людмила. Зина пришивает пуговку, Людмила прикладывает бирку. Внезапно, появляется *middleware* Лена, встает между Зиной и Людмилой и красит пуговку в хипстерский модный цвет. Так как Лена после покраски не уносит пальто с собой, а **передает дальше**, то Людмила как ни в чем не бывало приделывает бирку и пальто готово. Только теперь оно хипстерское. Усиленное.

Для лучшего понимания, предлагаю написать бесполезный усилитель, выдающий `console.log('ping')`, на каждое действие. При этом, мы будем использовать предложенный `redux` метод добавления усилителей с помощью *applyMiddleware*.

Обновим файл конфигурации `store`:

`store/configureStore.js`


```
import { createStore, applyMiddleware } from 'redux'
import rootReducer from '../reducers'
import { ping } from './enhancers/ping' // <!-- подключаем наш enhancer

export default function configureStore(initialState) {
  const store = createStore(
    rootReducer,
    initialState,
    applyMiddleware(ping)) // <!-- добавляем его в цепочку middleware'ов

  if (module.hot) {
    module.hot.accept('../reducers', () => {
      const nextRootReducer = require('../reducers')
      store.replaceReducer(nextRootReducer)
    })
  }

  return store
}
```

Напишем усилитель:

store/enhancers/ping.js

```
/*eslint-disable */
export const ping = store => next => action => {
  console.log('ping')
  return next(action)
}
/*eslint-enable */
```

Боюсь, здесь не обойтись без ES5 версии:

```
var ping = function ping(store) {
  return function (next) {
    return function (action) {
      console.log('ping');
      return next(action);
    };
  };
};
```

Поехали:

- *eslint-disable* - просто выключает проверку этого блока "линтером".
- ping - это функция, которая возвращает функцию. Middleware - это всегда функция, которые обычно возвращают функцию, если только целью middleware не является прервать цепочку вызовов.
- в возвращаемых функциях, благодаря [applyMiddleware](#) у нас становятся

доступными аргументы, которые мы можем использовать во благо приложения:

- `store` - *redux-store* нашего приложения;
- `next` - функция-обертка, которая позволяет продолжить выполнение цепочки;
- `action` - действие, которое было вызвано (как вы помните, вызванные действия - это *store.dispatch*)

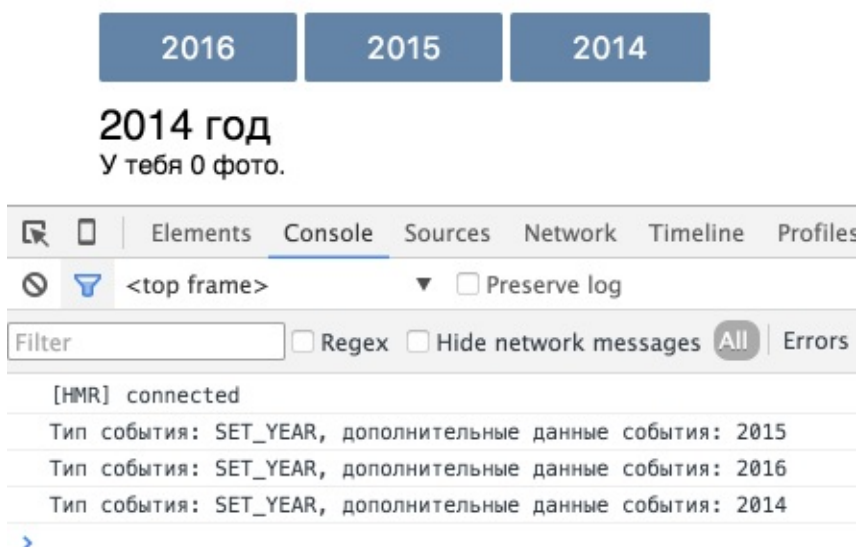
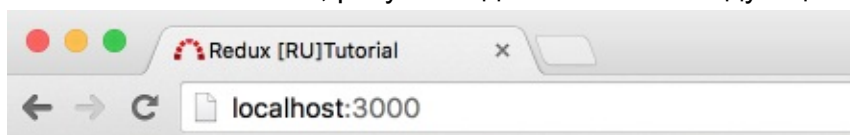
Сейчас, при клике на кнопки, у нас в консоли появляется строка `ping`. Давайте изменим ее, написав простейший логгер: *store/enhancers/ping.js*

```
/*eslint-disable */
export const ping = store => next => action => {
  console.log(`Тип события: ${action.type}, дополнительные данные события: ${action.payload}`)
  return next(action)
}
/*eslint-enable */
```

Я использовал новый строковый синтаксис. В прошлом, наш *console.log* выглядел бы так:

```
console.log('Тип события: ' + action.type + ', дополнительные данные события: ' + action.payload)
```

Покликайте на кнопки, результат должен быть следующим:



Redux-logger

Отбросим наш велосипед и поставим популярный [логгер](#).

```
npm i --save redux-logger
```

Удалите папку enhancers, и измените configureStore.

src/store/configureStore.js

```
import { createStore, applyMiddleware } from 'redux'
import rootReducer from '../reducers'
import createLogger from 'redux-logger'

export default function configureStore(initialState) {
  const logger = createLogger()
  const store = createStore(
    rootReducer,
    initialState,
    applyMiddleware(logger))

  if (module.hot) {
    module.hot.accept('../reducers', () => {
      const nextRootReducer = require('../reducers')
      store.replaceReducer(nextRootReducer)
    })
  }

  return store
}
```

Можете проверить - логгер достаточно информативный и удобен в использовании.

Таким образом, усилители - отличный способ добавить в наш процесс обработки действий некую прослойку с необходимой функциональностью.

Одним из популярнейших усилителей, является [redux-thunk](#), который мы как раз будем использовать для создания асинхронных действий.

[Исходный код](#) на текущий момент.

Асинхронные actions

Давайте представим синхронное действие:

1. Пользователь кликнул на кнопку
2. `dispatch action {type: ТИП_ДЕЙСТВИЯ, payload: доп.данные}`
3. интерфейс обновился

Давайте представим асинхронное действие:

1. Пользователь кликнул на кнопку
2. `dispatch action {type: ТИП_ДЕЙСТВИЯ_ЗАПРОС}`
3. запрос выполнен успешно
 - i. `dispatch action {type: ТИП_ДЕЙСТВИЯ_УСПЕШНО, payload: доп.данные}`
4. запрос выполнен неудачно
 - i. `dispatch action {type: ТИП_ДЕЙСТВИЯ_НЕУДАЧНО, error: true, payload: доп.данные ошибки}`

Благодаря такой схеме, в `reducer`'е мы сможем реализовать подобное:

```
switch(тип_действия)
  case ТИП_ДЕЙСТВИЯ_ЗАПРОС:
    покажи preloader
  case ТИП_ДЕЙСТВИЯ_УСПЕШНО:
    скрой preloader, покажи данные
  case ТИП_ДЕЙСТВИЯ_НЕУДАЧНО:
    скрой preloader, покажи ошибку
```

Как нам известно, действие - это простой объект, который возвращается функцией его создающей (*action creator*).

Убедимся в этом:

`src/actions/PageActions.js`

```
import { SET_YEAR } from '../constants/Page'
export function setYear(year) {
  return {
    type: SET_YEAR,
    payload: year
  }
}
```

Было бы неплохо иметь возможность возвращать не простой объект, а функцию, внутри которой иметь доступ к методу `dispatch`, и вызывать его с необходимым типом действия. Псевдокод, мог бы выглядеть так:

```
export function getPhotos(year) {
  return (dispatch) => {
    dispatch({
      type: GET_PHOTOS_REQUEST
    })

    $.ajax(url)
      .success(
        dispatch({
          type: GET_PHOTOS_SUCCESS,
          payload: response.photos
        })
      )
      .error(
        dispatch({
          type: GET_PHOTOS_FAILURE,
          payload: response.error,
          error: true
        })
      )
  }
}
```

Но вот незадача, actions - это простой объект, и если action creator возвращает не простой объект, а функцию, то это как-то... Подождите! Ведь это именно то, что нам нужно: Если action creator возвращает не простой объект, а функцию - выполни ее, иначе если это простой объект ... тадам, **передай дальше**. Более того, благодаря `applyMiddleware` у нас как раз есть доступный метод `dispatch`! И еще бонусом `getState`.

Отлично, мы только что поняли, что нам нужен еще один усилитель. Такой усилитель уже написан, причем [код его](#) невероятно прост, я даже приведу его здесь:

усилитель: `redux-thunk`

```
function thunkMiddleware({ dispatch, getState }) {
  return next => action =>
    typeof action === 'function' ?
      action(dispatch, getState) :
      next(action);
}

module.exports = thunkMiddleware
```

Нам остается лишь добавить зависимость в наш проект, и убедиться, что у нас `redux` версии, не ниже 3.1.0

```
npm update redux --save
npm install redux-thunk --save
```

Для практики, предлагаю написать следующее:

- по клику на кнопку с номером года
 - меняется год в заголовке
 - ниже (где должны быть фото), появляется текст "Загрузка..."
- после удачной загрузки*
 - убрать текст "Загрузка..."
 - отобразить строку "У тебя XX фото" (зависит, от длины массива, переданного в `action.payload`)

* вместо реального метода загрузки, использовать `setTimeout`, который является удобным для тренировок исполнения асинхронных запросов.

Вы можете попробовать выполнить это задание сами, а потом сравнить его с решением ниже.

Для отображения / скрытия фразы "Загрузка...", используйте в `reducer`'е еще одно свойство у состояния. Например, *fetching*:

```
const initialState = {
  year: 2016,
  photos: [],
  fetching: false
}
```

Решение ниже.

Для начала изменим набор констант:

src/constants/Page.js

```
export const GET_PHOTOS_REQUEST = 'GET_PHOTOS_REQUEST'
export const GET_PHOTOS_SUCCESS = 'GET_PHOTOS_SUCCESS'
```

Далее добавим новый усилитель: *src/store/configureStore.js*

```
import { createStore, applyMiddleware } from 'redux'
import rootReducer from '../reducers'
import createLogger from 'redux-logger'
import thunk from 'redux-thunk' // <-- добавили redux-thunk

export default function configureStore(initialState) {
  const logger = createLogger()
  const store = createStore(
    rootReducer,
    initialState,
    applyMiddleware(thunk, logger)) // <-- добавили его в цепочку перед logger'ом

  if (module.hot) {
    module.hot.accept('../reducers', () => {
      const nextRootReducer = require('../reducers')
      store.replaceReducer(nextRootReducer)
    })
  }

  return store
}
```

Изменим action creator: *src/actions/PageActions.js*

```
import {
  GET_PHOTOS_REQUEST,
  GET_PHOTOS_SUCCESS
} from '../constants/Page'

export function getPhotos(year) {

  return (dispatch) => {
    dispatch({
      type: GET_PHOTOS_REQUEST,
      payload: year
    })

    setTimeout(() => {
      dispatch({
        type: GET_PHOTOS_SUCCESS,
        payload: [1, 2, 3, 4, 5]
      })
    }, 1000)
  }
}
```

Изменим reducer: *src/reducers/page.js*

```
import {
  GET_PHOTOS_REQUEST,
  GET_PHOTOS_SUCCESS
} from '../constants/Page'

const initialState = {
  year: 2016,
  photos: [],
  fetching: false
}

export default function page(state = initialState, action) {

  switch (action.type) {
    case GET_PHOTOS_REQUEST:
      return { ...state, year: action.payload, fetching: true }

    case GET_PHOTOS_SUCCESS:
      return { ...state, photos: action.payload, fetching: false }

    default:
      return state;
  }
}
```

У нас готова логика для обновления состояния (и интерфейса, разумеется). Осталось поправить отображение.

Так как мы переписали и переименовали функцию (setYear -> getPhotos):

src/containers/App.js

```
...
const { getPhotos } = this.props.pageActions

return <div className='row'>
  <Page photos={page.photos} year={page.year} getPhotos={getPhotos} fetching={page.
fetching}/>
...

```

Причем, в `mapDispatchToProps` - нам ничего менять не нужно, так как мы по прежнему присоединяем все `pageActions` в `props` контейнера `<App />`

Обновим соответствующий компонент: *src/components/Page.js*


```
import React, { PropTypes, Component } from 'react'

export default class Page extends Component {
  onYearBtnClick(e) {
    this.props.getPhotos(+e.target.innerText)
  }
  render() {
    const { year, photos, fetching } = this.props
    return <div className='ib page'>
      <p>
        <button className='btn' onClick={::this.onYearBtnClick}>2016</button>{' '}
        <button className='btn' onClick={::this.onYearBtnClick}>2015</button>{' '}
        <button className='btn' onClick={::this.onYearBtnClick}>2014</button>
      </p>
      <h3>{year} год</h3>
      {
        fetching ?
        <p>Загрузка...</p>
        :
        <p>У тебя {photos.length} фото.</p>
      }
    </div>
  }
}

Page.propTypes = {
  year: PropTypes.number.isRequired,
  photos: PropTypes.array.isRequired,
  getPhotos: PropTypes.func.isRequired
}
```

Когда будете проверять работу в браузере, обратите внимание на логгер. Он все так же работает и информативен.

Пока мы писали код для асинхронного запроса, мы НЕ нарушили главные принципы redux-приложения:

1. Мы всегда возвращали новое состояние (новый объект, смотрите *src/reducers/page.js*)
2. Мы строго следовали однонаправленному потоку данных в приложении: *юзер кликнул - возникло действие - редьюсер изменил - компонент отобразил.*

Итого: вы можете сами дописать наше приложение, чтобы оно взаимодействовало с VK, так как все что нужно, это добавить реальный асинхронный запрос (точнее парочку - для логина, и для получения фото). Ложку дегтя добавляет тот факт, что для этого потребуется создать в интерфейсе VK приложение, и выполнять наши запросы с реального сервера, так как VK.API не работает с localhost.

Об этом мы и поговорим в следующей главе.

[Исходный код](#) на данный момент.

Взаимодействуем с VK

Чтобы работать с [VK API](#) вам необходимо будет создать приложение на сайте [vk.com](#), и указать в настройках URL сервера, с которого вы будете выполнять запросы.

Localhost не поддерживается.

Интеграция VK API

Необходимо добавить скрипт *openapi* перед нашей сборкой - *bundle.js*, а так же вызвать *VK.init*

```
<!DOCTYPE html>
<html>
  <head>
    <title>Redux [RU]Tutorial</title>
  </head>
  <body>
    <div id="root" class="container-fluid">
      </div>
      <script src="//vk.com/js/api/openapi.js"></script>
      <script src="/static/bundle.js"></script>
      <script language="javascript">
        VK.init({
          apiId: 5087365
        });
      </script>
    </body>
  </html>
```

Авторизация

Создадим действия для User.

src/actions/UserActions.js

```
import {
  LOGIN_REQUEST,
  LOGIN_SUCCES,
  LOGIN_FAIL
} from '../constants/User'

export function handleLogin() {

  return function(dispatch) {

    dispatch({
      type: LOGIN_REQUEST
    })

    VK.Auth.login((r) => { // eslint-disable-line no-undef
      if (r.session) {
        let username = r.session.user.first_name;

        dispatch({
          type: LOGIN_SUCCES,
          payload: username
        })

      } else {
        dispatch({
          type: LOGIN_FAIL,
          error: true,
          payload: new Error('Ошибка авторизации')
        })
      }
    }, 4); // запрос прав на доступ к photo
  }
}
```

Проверьте список констант:

```
export const LOGIN_REQUEST = 'LOGIN_REQUEST'
export const LOGIN_SUCCES = 'LOGIN_SUCCES'
export const LOGIN_FAIL = 'LOGIN_FAIL'
```

"Приконнектим" в `<App />` `UserActions`, и добавим новые свойства в компонент `<User />`

`src/containers/App.js`

```
import React, { Component } from 'react'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'
import User from '../components/User'
import Page from '../components/Page'
import * as pageActions from '../actions/PageActions'
import * as userActions from '../actions/UserActions'

class App extends Component {
  render() {
    const { user, page } = this.props
    const { getPhotos } = this.props.pageActions
    const { handleLogin } = this.props.userActions

    return <div className='row'>
      <Page photos={page.photos} year={page.year} getPhotos={getPhotos} fetching={page.fetching} />
      <User name={user.name} handleLogin={handleLogin} error={user.error} />
    </div>
  }
}

function mapStateToProps(state) {
  return {
    user: state.user,
    page: state.page
  }
}

function mapDispatchToProps(dispatch) {
  return {
    pageActions: bindActionCreators(pageActions, dispatch),
    userActions: bindActionCreators(userActions, dispatch)
  }
}

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

Обновим reducer user:

src/reducers/user.js

```
import {
  LOGIN_SUCCES,
  LOGIN_FAIL
} from '../constants/User'

const initialState = {
  name: '',
  error: ''
}

export default function user(state = initialState, action) {

  switch(action.type) {
    case LOGIN_SUCCES:
      return { ...state, name: action.payload, error: '' }

    case LOGIN_FAIL:
      return { ...state, error: action.payload.message }

    default:
      return state
  }
}
```

И покажем все это в компоненте `<User />`

src/components/User.js

```
import React, { PropTypes, Component } from 'react'

export default class User extends Component {

  render() {
    const { name, error } = this.props
    let template

    if (name) {
      template = <p>Привет, {name}!</p>
    } else {

      template = <button className='btn' onClick={this.props.handleLogin}>Войти</button>
    }

    return <div className='ib user'>
      {template}
      {error ? <p className='error'> {error}. <br /> Попробуйте еще раз.</p> : ''}
    </div>
  }
}

User.propTypes = {
  name: PropTypes.string.isRequired,
  handleLogin: PropTypes.func.isRequired,
  error: PropTypes.string.isRequired
}
```

Сейчас если кликнуть на "войти" - всплывет VK окно с подтверждением прав доступа (первый раз). После подтверждения прав, вместо кнопки войти появляется надпись "Привет, XXX". При перезагрузке сайта и повторных нажатиях на "войти" - VK окно мгновенно закрывается, а кнопка вновь изменяется на "Привет, XXX". Неплохо бы было проверять "статус", например в *componentWillMount*, но оставляю это на "домашку".

Как всегда, доблестный логгер пишет в консоли - что происходит.

Загрузка фото

Нам нужно практически повторить, все что написано выше, только для блока **Page**.

Можете попробовать сами, используя метод `photos.getAll` из VK API.

Для начала, проверим список констант:

src/constants/Page.js

```
export const GET_PHOTOS_REQUEST = 'GET_PHOTOS_REQUEST'  
export const GET_PHOTOS_SUCCESS = 'GET_PHOTOS_SUCCESS'  
export const GET_PHOTOS_FAIL = 'GET_PHOTOS_FAIL'
```

Напишем немало кода, для загрузки фото:

src/actions/PageActions.js

```
import {  
  GET_PHOTOS_REQUEST,  
  GET_PHOTOS_FAIL,  
  GET_PHOTOS_SUCCESS  
} from '../constants/Page'  
  
let photosArr = []  
let cached = false  
  
function makeYearPhotos(photos, selectedYear) {  
  let createdYear, yearPhotos = []  
  
  photos.forEach((item) => {  
    createdYear = new Date(item.created*1000).getFullYear()  
    if (createdYear === selectedYear ) {  
      yearPhotos.push(item)  
    }  
  })  
  
  yearPhotos.sort((a,b) => b.likes.count-a.likes.count);  
  
  return yearPhotos  
}  
  
function getMorePhotos(offset, count, year, dispatch) {  
  VK.Api.call('photos.getAll', {extended:1, count: count, offset: offset},(r) => { //  
    eslint-disable-line no-undef  
    try {  
      if (offset <= r.response[0] - count) {  
        offset+=200;  
        photosArr = photosArr.concat(r.response)  
        getMorePhotos(offset, count, year, dispatch)  
      } else {  
        let photos = makeYearPhotos(photosArr, year)  
        cached = true  
        dispatch({  
          type: GET_PHOTOS_SUCCESS,  
          payload: photos  
        })  
      }  
    }  
  })  
  catch(e) {  
    dispatch({  
      type: GET_PHOTOS_FAIL,  
      error: true,  
    })  
  }  
}
```



```
    payload: new Error(e)
  })
}

})
}

export function getPhotos(year) {

  return (dispatch) => {
    dispatch({
      type: GET_PHOTOS_REQUEST,
      payload: year
    })

    if (cached) {
      let photos = makeYearPhotos(photosArr, year)
      dispatch({
        type: GET_PHOTOS_SUCCESS,
        payload: photos
      })
    } else {
      getMorePhotos(0, 200, year, dispatch)
    }
  }
}
```

`makeYearPhotos` и `getMorePhotos` можно вынести в папку `utils`, как вспомогательные функции.

Главное здесь, что мы по-прежнему вызываем действия (*dispatch actions*). Все так, как было в самом начале, просто добавилось немного больше логики для получения фото. Алгоритм получения всех фото (да и необходимость получения всех) - оставляю без комментариев. Мне кажется, это приемлемый способ.

Чтобы потестировать показ ошибок, достаточно просто исправить цифру 200 на 2 или 20. VK с любовью вам ответит, что вы мягко-говоря, очень настойчиво обращаетесь к API ;)

Исправив редьюсер и отрисовку в компоненте, мы закончим начатое.

`src/reducers/page.js`

```
import {
  GET_PHOTOS_REQUEST,
  GET_PHOTOS_SUCCESS,
  GET_PHOTOS_FAIL
} from '../constants/Page'

const initialState = {
  year: 2016,
  photos: [],
  fetching: false,
  error: ''
}

export default function page(state = initialState, action) {

  switch (action.type) {
    case GET_PHOTOS_REQUEST:
      return { ...state, year: action.payload, fetching: true, error: '' }

    case GET_PHOTOS_SUCCESS:
      return { ...state, photos: action.payload, fetching: false, error: '' }

    case GET_PHOTOS_FAIL:
      return { ...state, error: action.payload.message, fetching: false }

    default:
      return state;
  }
}
```

src/components/Page.js

```
import React, { PropTypes, Component } from 'react'

export default class Page extends Component {
  onYearBtnClick(e) {
    this.props.getPhotos(+e.target.innerText)
  }
  render() {
    const { year, photos, fetching, error } = this.props
    const years = [2016, 2015, 2014, 2013, 2012, 2011, 2010]
    return <div className='ib page'>
      <p>
        { years.map((item, index) => <button className='btn' key={index} onClick={::this.onYearBtnClick}>{item}</button> )}
      </p>
      <h3>{year} год [{photos.length}]</h3>
      { error ? <p className='error'> Во время загрузки фото произошла ошибка</p> : ''
    }

    {
      fetching ?
      <p>Загрузка...</p>
      :
      photos.map((entry, index) =>
        <div key={index} className='photo'>
          <p><img src={entry.src} /></p>
          <p>{entry.likes.count} ♥</p>
        </div>
      )
    }
  </div>
}

Page.propTypes = {
  year: PropTypes.number.isRequired,
  photos: PropTypes.array.isRequired,
  getPhotos: PropTypes.func.isRequired,
  error: PropTypes.string.isRequired
}
```

Итого: Вы научились выполнять асинхронные запросы и корректно показывать прелоадер, ошибки или успешный результат.

[Исходный код](#) на текущий момент.

P.S. css тоже был слегка подправлен.

Заключение

Спасибо, что вы прочитали данный учебник, если у вас остались вопросы, задавайте их в [твиттере](#), или пишите на почту - maxfarseer@gmail.com с темой "Redux RU".

План обновлений:

- раздел по основам react.js -> написан [туториал](#)
- тестирование
- оптимизация сборки

Как говорится, stay tuned!