М. ПАЦИАНСКИЙ

ОСНОВЫ REDUX

II ИЗДАНИЕ (2018)

REACT: ^16.4.3
REDUX: ^4.0.0

без воды

Содержание

React Redux [RU tutorial] (2-е издание)

Версия от 2018 года включает в себя React `^16.4.1` (без проблем апгрейдится до `16.4.3`) и Redux `^4.0.0`

Так же в 2018м году активность в онлайне выросла в разы, подробнее в разделе "От автора".

В данном учебном курсе вы найдете 2 раздела:

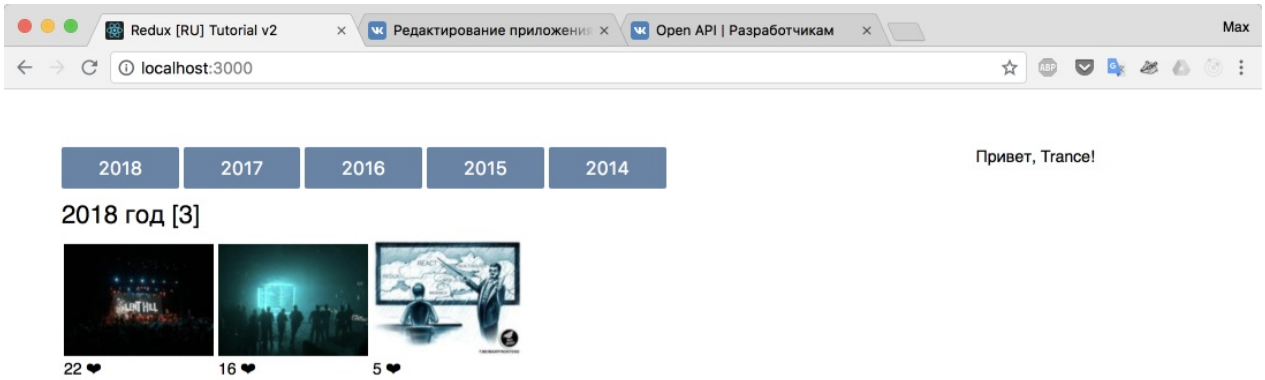
1. Подготовка (сильно похудела во втором издании, так как появился CRA)
2. Теория redux и создание веб-приложения по шагам

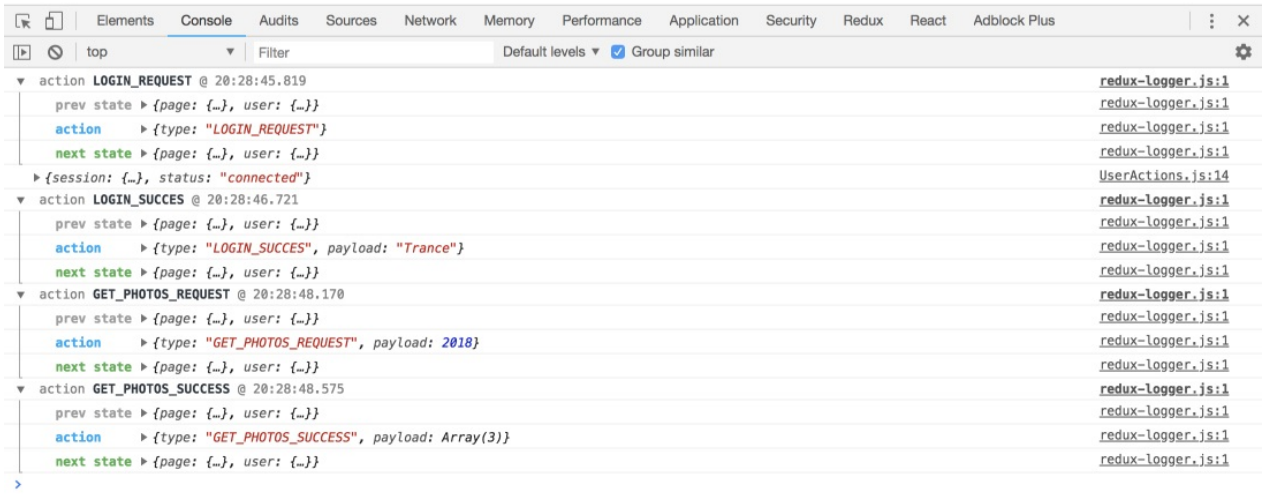
Курс предполагает, что читатель уже знаком с React. Если вы не знакомы, рекомендую для начала ознакомиться с курсом React.js для начинающих.

В результате прохождения курса, вы научитесь:

- Основам создания SPA-приложения на React;
- Грамотно готовить Redux-приложение (однонаправленный поток данных);
- Выполнять асинхронные запросы (прелоадер, обработка ошибок) с помощью стандартного redux-thunk;
- Взаимодействовать со сторонними API (на примере VK API);
- Работать с документацией (по-желанию);
- Оптимизировать перерисовки компонентов;

Результатом изучения будет приложение, которое выведет ваши фото из VK отсортированные по лайкам с фильтром по году.



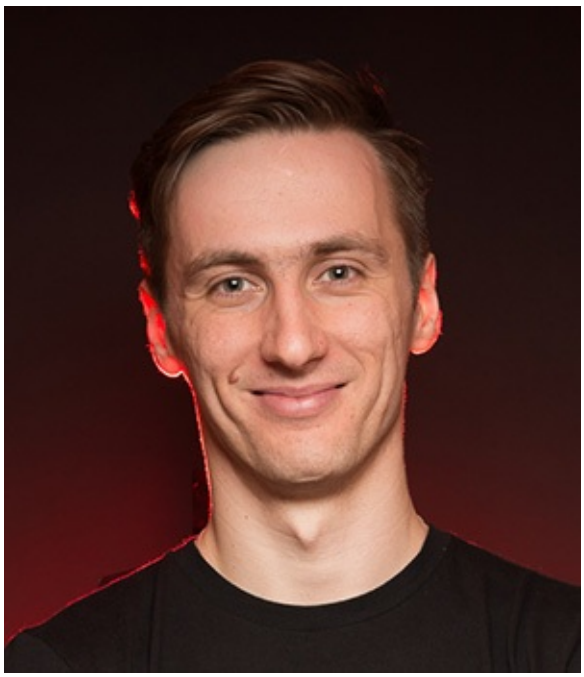


Поддержать проект

Вы можете поддержать проект, мне будет очень приятно.

Если у вас не получается поддержать проект материально, вы можете оставить отзыв в группе vk.

Обо мне



Меня зовут Максим Пацианский, я Frontend-разработчик, стартанул в этой теме с Joomla! сайтов в 2008 году.

Занимаюсь консультированием по React более 2х лет, с момента выхода прошлых учебников.

Подробнее о моем опыте консультирования я писал на хабре.

Напутствие

Пожалуйста, выполняйте код по ходу книги. Ломайте его, "консольте", интересуйтесь.

Полезные ссылки

Мои уроки/вебинары/соц.сети:

- Полноценный учебник "Основы React"
- Расписание стримов и вебинаров (на сайте есть текстовые версии вебинаров)
- Youtube канал с записями вебинаров и стримов
- Группа vkontakte
- Канал в telegram
- Twitter
- Facebook

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

Redux (EN) - документация по Redux (так же есть примеры)

Консультации и платные услуги

С 2016 года, я с удовольствием занимаюсь консультированием 1 на 1, поиском проблем в коде, помощью в подготовке к собеседованию и т.д. Хороший багаж опыта, которым я готов поделиться понятным языком.

Актуальный прайс

Подготовка

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

- Установка create-react-app (CRA) [копия разделов из учебника по основам React]
- настройка VS Code для удобной работы
  - настройка Prettier
  - настройка ESLint

Результатом подготовки, будет следующий код.

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

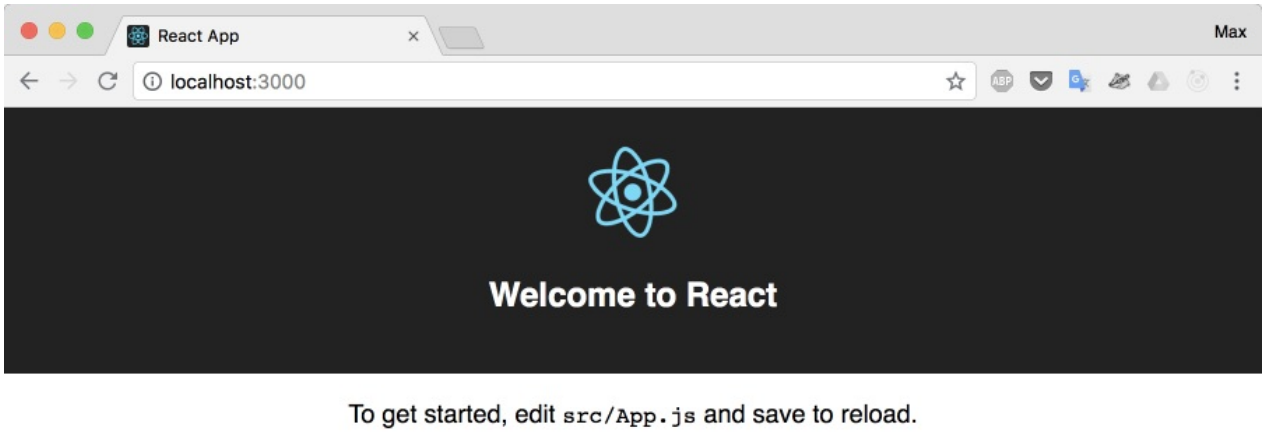
Для всех остальных, я предлагаю за несколько простых шагов настроить удобное рабочее окружение.

Установка и запуск create-react-app

```
npx create-react-app my-app
cd my-app
npm start
```

Если вы не знакомы с данными командами, значит вам нужно поставить себе node.js и ввести их в терминале после.

После запуска мы получим следующую картину в браузере:



И следующую файловую структуру:

```
+-- node_modules (здесь расположены пакеты для работы приложения)
+-- public (здесь расположены публичные файлы, такие как index.html и favicon)
+-- src (здесь живет компонент App)
+-- .gitignore (файл для гита)
+-- package.json (файл с зависимостями проекта)
+-- README.md (описание проекта)
+-- yarn.lock (может быть, а может и не быть - тоже относится к теме зависимостей прое
кта)
```

CRA при каждом изменении в файлах внутри директории src - перезагружает страницу в браузере.

Про import/export задерживаться не будем, так как думаю вы это уже знаете. Если что, есть глава "Приборка и импорты" в учебнике по основам реакта.

Исходный код

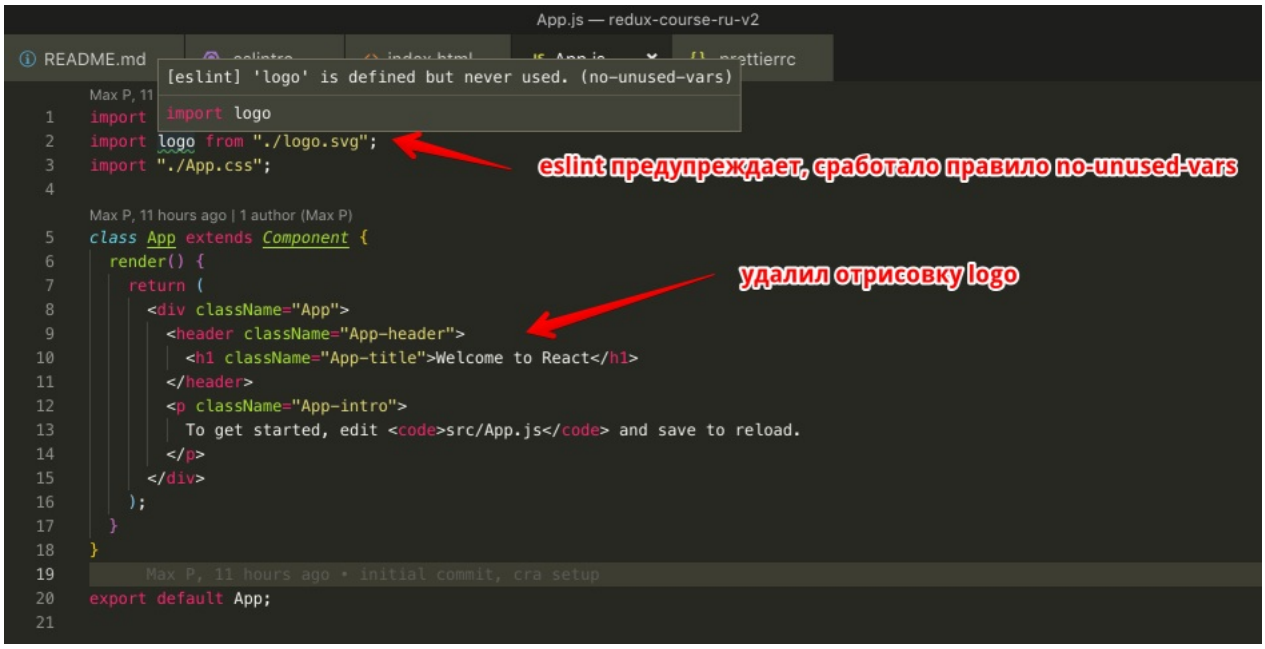
ESLint и Prettier

Кратко о библиотеках:

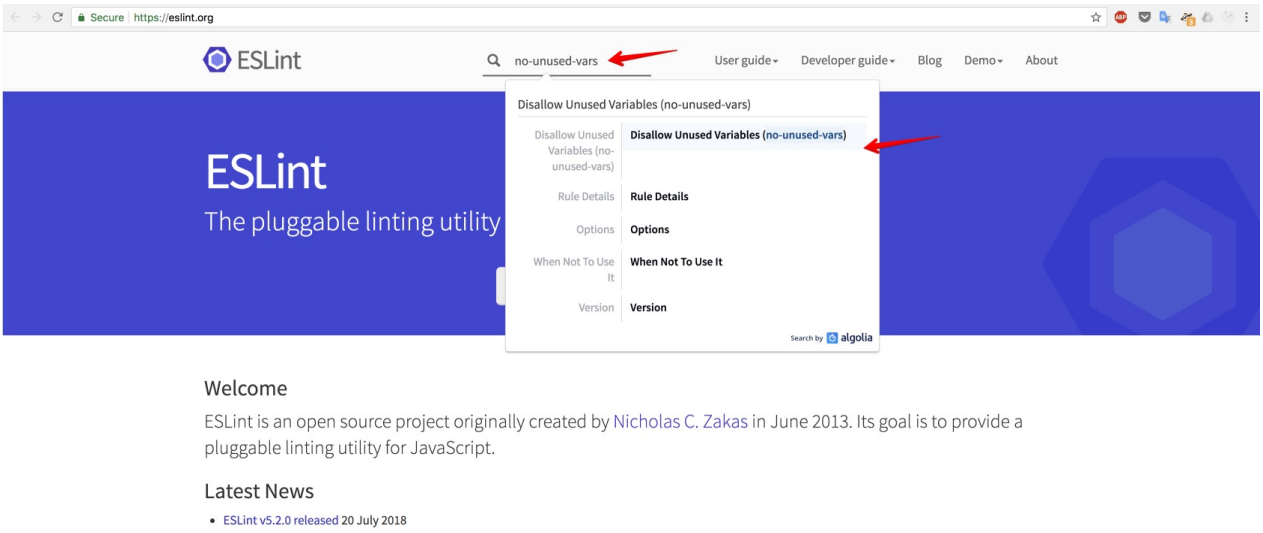
ESLint

Линтер - это помощник по части "здоровья" кода. Вы определяете список правил и в дальнейшем, при настроенном плагине в вашем редакторе, он как Microsoft Word "проферка орфографии" проверяет все, что вы написали.

Например, определили переменную, но нигде не используете? Сработает правило: no-unused-vars (долой неиспользуемые переменные) и переменная будет подчеркнута.



Когда вы видите "подчеркивание", и после наведения видите в скобочках название правила - не нужно бежать гуглить. Нужно идти на сайт eslint.org и там в "поиск" вставлять текст ошибки, будет быстрее.

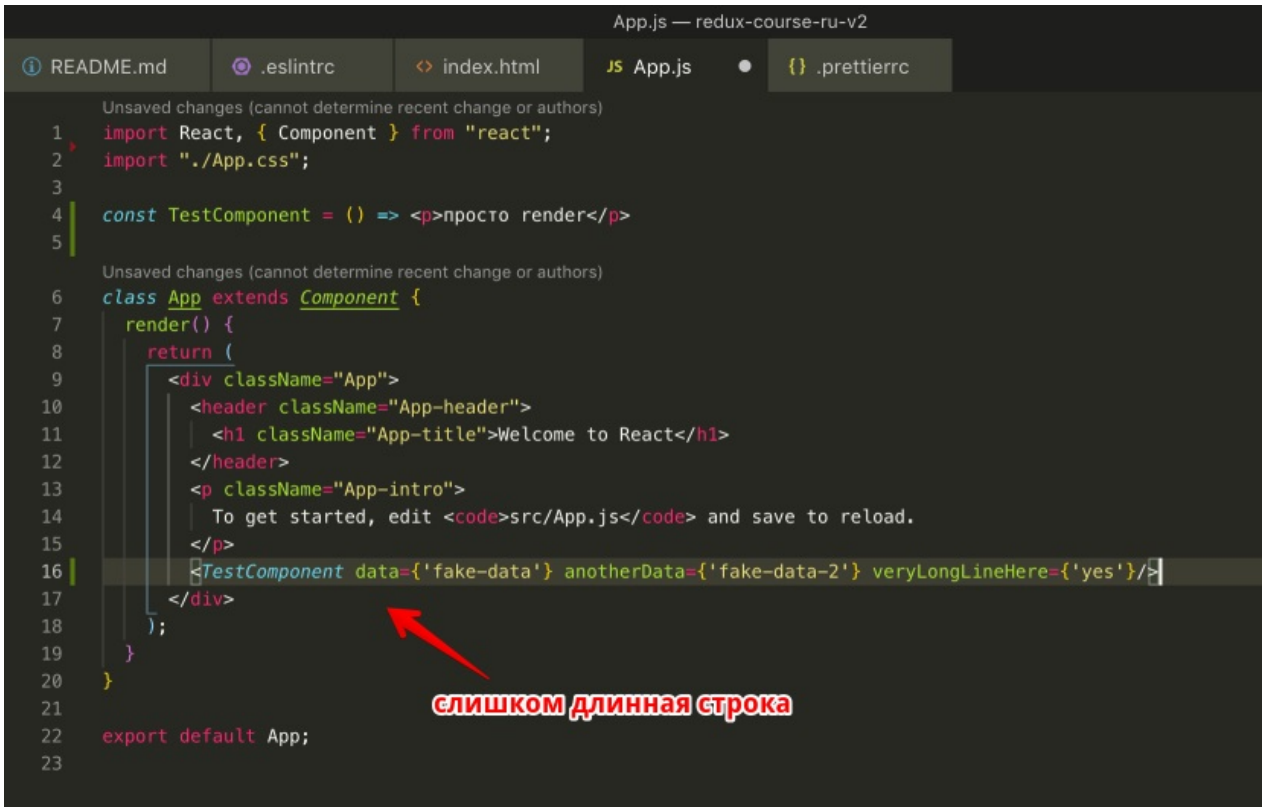


Prettier

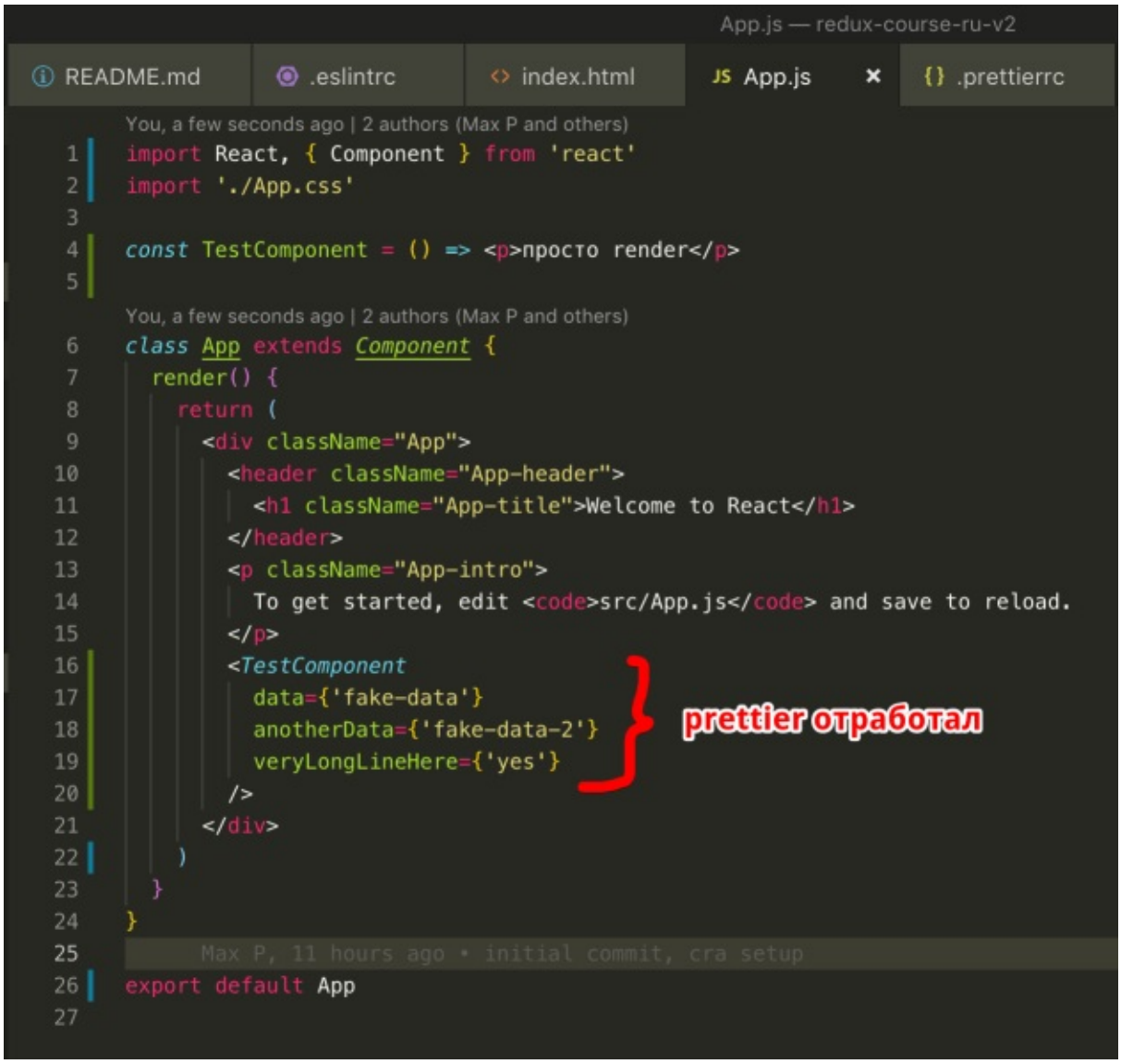
Преттир - это помощник по части оформления кода. Можно писать с пробелами перед именем свойства, кавычками, запятыми в последней строке и тд тп - преттир, настроенный на сохранение или на пре-коммит хук - "перетрясет" ваши файлы и оформит их в соответствии с настройками, которых у него минимум. Это сделано специально, ибо чем меньше настроек, тем меньше конфигураций - когда-нибудь, спор "табы vs пробелы" уйдет в небытие, но кто выиграет?)

Одна из работ "преттира" - форматировать длинные строки.

Было:



Стало:



Я думаю преимущества очевидны, поэтому давайте настроим необходимые ускорители повседневной разработки.

Настройка

Линтер встроен в create-react-app, но для работы в связке с Prettier, а так же для подсветки кода во время написания в VS Code нужна небольшая донастройка.

Для начала установите пакеты:

```
npm install eslint-config-prettier eslint-plugin-prettier prettier lint-staged husky --save-dev
```

Все пакеты в целом понятны зачем, кроме lint-staged и husky

- husky - упрощает работу с git hooks ("пре-коммит" (момент, когда вы собираетесь делать коммит) легко настроить с помощью этой "собаки")
- lint-staged - пакет, который позволяет вам сделать обработку командой из терминала только тех файлов, которые собираются улететь в коммит.

Husky и *lint-staged* - сладкая парочка для борьбы с плохим кодом в нашем репозитории. Например, мы можем настроить, что если ESLint вернул ошибку, то коммит будет автоматически отменен. Вернемся к этому позже.

Итак, **настройка eslint**, создайте следующий файл в корне проекта:

.eslintrc

```
{
  "extends": [
    "react-app",
    "prettier"
  ],
  "rules": {
    "jsx-quotes": [
      1,
      "prefer-double"
    ]
  },
  "plugins": [
    "prettier"
  ]
}
```

Достаточно скромный конфиг, который "наследует" стандартные правила (их много) из *react-app* и *prettier* (это глобальные конфиги, один встроен в create-react-app, второй мы установили посредством пакета eslint-config-prettier)

Затем я переопределил одно правило: jsx-quotes (для имен классов внутри JSX будут ставиться двойные кавычки. Не могу сказать, насколько это важно на сегодняшний день, но раньше у меня были конфликты с преттиром без этого правила).

Вы можете переопределить в списке любые правила, которые вас интересуют. Список можно найти в документации, но проще просто начать работать и по ходу пьесы смотреть на "подчеркивания". Те, которые вас не устраивают - переопределяйте.

Последняя опция в конфиге - использование плагинов. Мы используем плагин prettier (пакет eslint-plugin-prettier), чтобы не было конфликтов между "помощниками" (напоминаю, у нас их два: prettier и eslint).

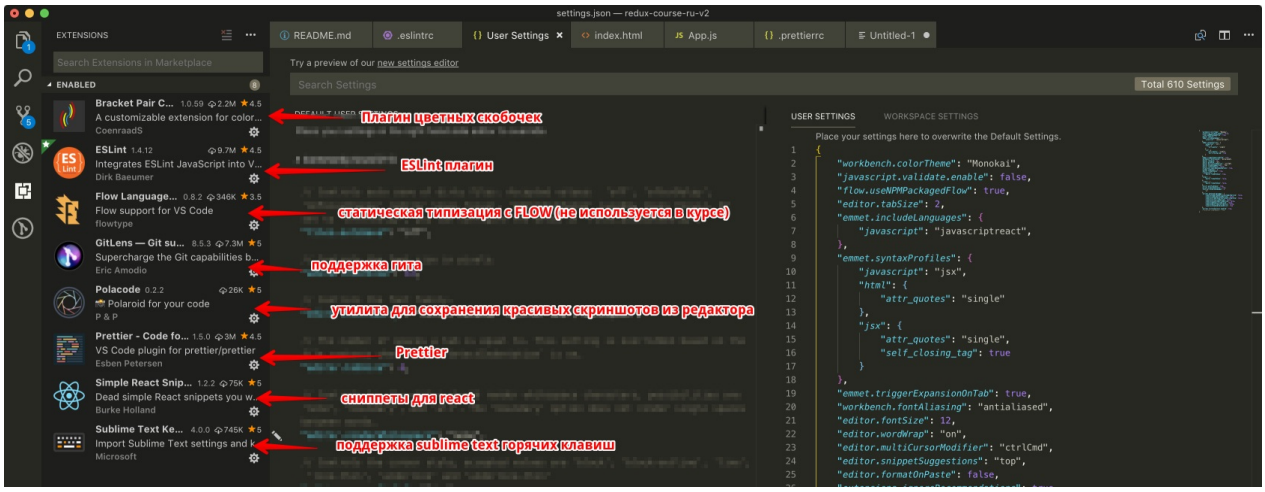
После настройки конфига, вам нужно настроить ваш редактор. Я приведу пример только для Visual Studio Code.

Добавьте в файл с настройками, следующие строки:

```
"editor.formatOnPaste": false,
"editor.formatOnSave": true,
"[javascript]": {
    "editor.formatOnSave": true,
},
"[html]": {
    "editor.formatOnSave": false,
},
"[json]": {
    "editor.formatOnSave": false,
},
"eslint.autoFixOnSave": true,
"eslint.alwaysShowStatus": true,
```

Напоследок, для корректной работы вам потребуется парочка плагинов из маркет-плейса (eslint и prettier).

Мой список плагинов:



Конфиг может быть настроен различными способами, например, взгляните на эти два видео:

- Add ESLint & Prettier to VS Code for a Create React App
- How to Setup VS Code + Prettier + ESLint

Настроим prettier (нам так же нужен конфигурационный файл):

.prettierrc

```
{
  "useTabs": false, // использовать табы? нет (я за пробелы)
  "printWidth": 80, // длина строки - 80
  "tabWidth": 2, // длина "таба" - 2 пробела
  "singleQuote": true, // использовать одинарные кавычки - да!
  "trailingComma": "es5", // запятая в последней строке - да
  "jsxBracketSameLine": false, // закрывающийся jsx в этой же строке
  "parser": "flow", // парсер - flow (пока не важно)
  "semi": false // точка с запятой - нет
}
```

Вот и все настройки. Настройка - parser, вам пока не должна мешать, а что такое trailingComma - пример ниже:

```
const data = {
  name: 'Max',
  city: 'Moscow, // <-- traling comma ("висячая запятая")
}
```

Почему так? Мне это нравится, так как если добавится новое свойство, в *git difference* (изменения в файле) будет только одна строка, вместо двух (в одной добавилась бы запятая, во второй - новое свойство).

На данный момент, если вы будете писать код, у вас уже будет отрабатывать eslint. Так же в момент сохранения, код будет преобразовываться с помощью prettier. Однако, нам еще не хватает настройки пре-коммит хука.

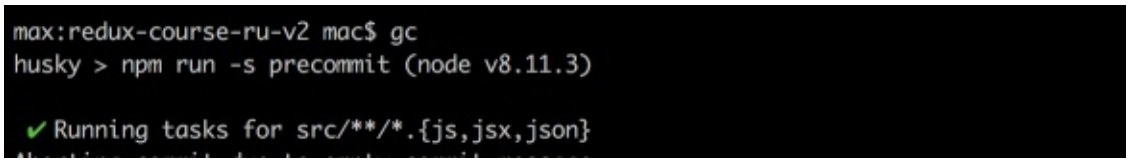
Представьте ситуацию: вы работает с коллегой. Он пишет в блокноте, у него нет никаких "преттиров". Следовательно, чтобы он не закоммитил код, который не отформатирован как вам нужно, мы настраиваем пре-коммит хук. Это значит, в момент коммита, весь `js/jsx/json` код из директории *src*, который он "коммитит" будет преобразован преттиром, так же, как если бы он преобразовался при сохранении в вашем редакторе.

package.json

```
{
  "name": "redux-course-ru-v2",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "react": "^16.4.1",
    "react-dom": "^16.4.1",
    "react-scripts": "1.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test --env=jsdom",
    "eject": "react-scripts eject",
    "precommit": "lint-staged"
  },
  "devDependencies": {
    "eslint-config-prettier": "^2.9.0",
    "eslint-plugin-prettier": "^2.6.2",
    "husky": "^0.14.3",
    "lint-staged": "^7.2.0",
    "prettier": "^1.14.0"
  },
  "lint-staged": {
    "*.{js, jsx}": [
      "prettier --write",
      "git add"
    ]
  }
}
```

В секции scripts добавилась команда *precommit*, и добавилось свойство lint-staged с настройками.

Теперь в момент коммита, в терминале будет похожая ситуация:



Резонный вопрос, у коллеги с блокнотом, у него и ESLint отсутствует же? Верно. Нужно усложнить ему жизнь и "обламывать" коммит, если в нем есть ошибки/предупреждения от ESLint.

Удалите `<TestComponent />` из отрисовки в `<App />`, но оставьте создание переменной.

src/App.js

```
import React, { Component } from 'react'
import './App.css'

const TestComponent = () => <p>просто render</p>

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Welcome to React</h1>
        </header>
        <p className="App-intro">
          To get started, edit <code>src/App.js</code> and save to reload.
        </p>
      </div>
    )
  }
}

export default App
```

Выполните команду в терминале (находясь в директории с проектом):

```
node_modules/.bin/eslint src/
```

Так как я не люблю глобальные зависимости, я использую локально установленный eslint (его установил для нас create-react-app). Чтобы упростить вызов в терминале, можно добавить в секцию *scripts* в *package.json* новую команду:

package.json

```
...
"scripts": {
  "start": "react-scripts start",
  "build": "react-scripts build",
  "test": "react-scripts test --env=jsdom",
  "eject": "react-scripts eject",
  "precommit": "lint-staged",
  "eslint": "node_modules/.bin/eslint src/"
},
...
```

Теперь eslint в терминале можно запускать так: `npm run eslint`. После запуска этой команды, eslint проверит весь src/ на наличие ошибок/предупреждений. Это полезно сделать в начале внедрения "жесткого пре-коммита" и лично исправить все ошибки, чтобы команда научилась на хорошем примере.

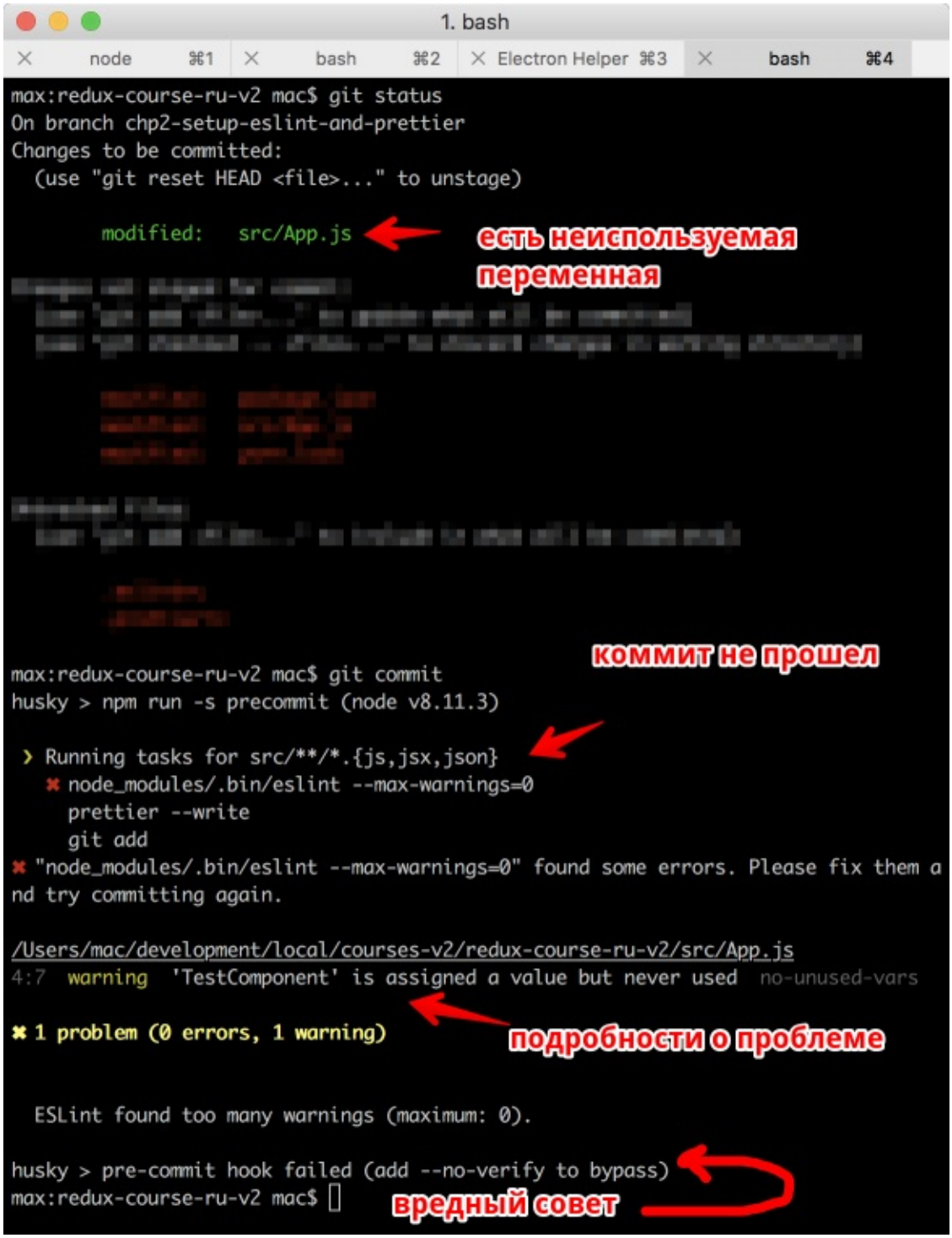
Вернемся к настройке. Изменим lint-stage скрипт в package.json на:

```
"lint-staged": {
  "*.{js, jsx}": [
    "node_modules/.bin/eslint --max-warnings=0",
    "prettier --write",
    "git add"
  ]
}
```

Теперь в момент пре-коммита будет запускаться lint-staged проверка в которой eslint и prettier обработают все файлы, готовящиеся к коммиту.

Что интересно, я настроил еслинт агрессивно (опция `--max-warnings=0`), то есть, даже любое предупреждение прервет коммит.

Проверим:



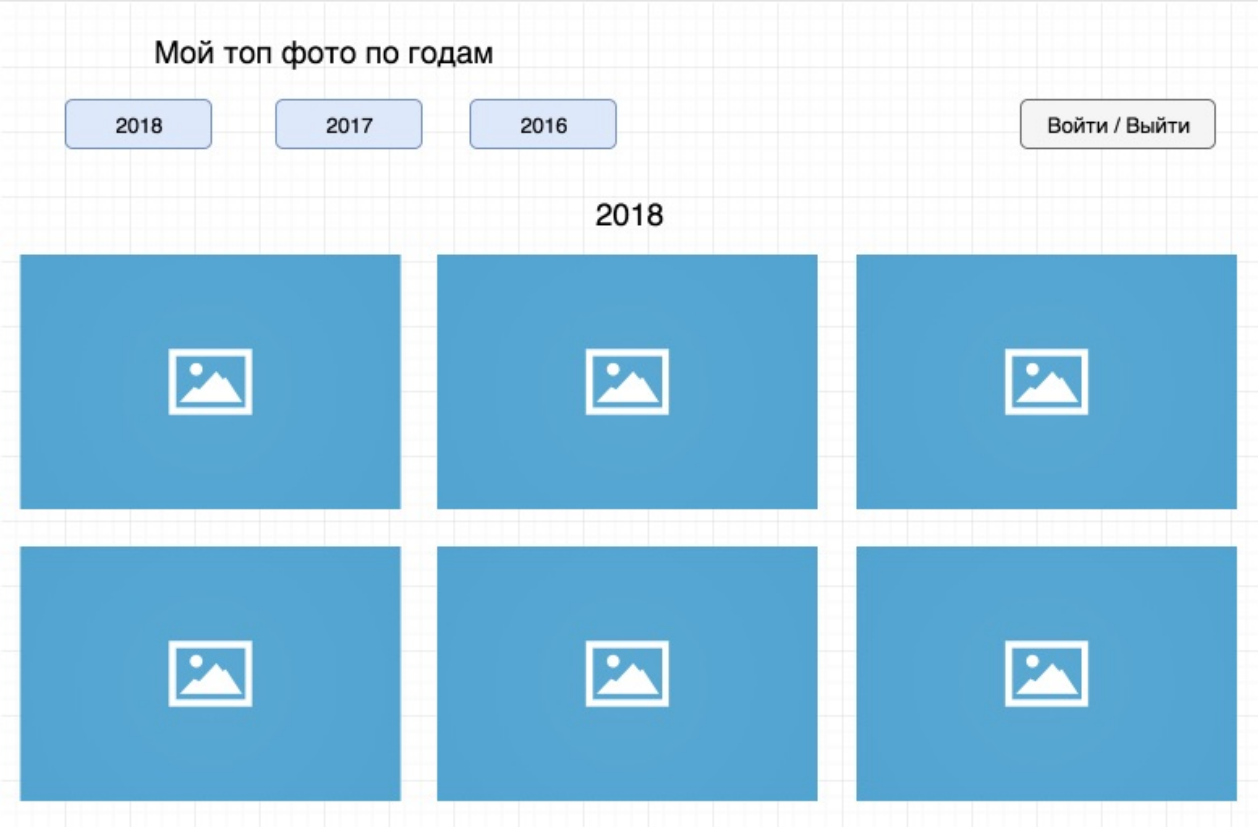
На скрине видно "вредный совет". Да, если добавить `--no-verify` к команде `git commit`, то проверок не будет. Но за это сразу бейте по рукам.

Итого: Настроили ESLint, prettier и pre-commit hook. Очень сильно облегчили жизнь себе и коллегам, кто болеет за единый стиль и чистый код.

Исходный код (без ошибок).

Создание

Я предлагаю по шагам создать одностраничное приложение, с минимумом функций, которое после логина в VK 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 (store) приложения в одном месте;
- однонаправленный поток данных: случился action -> редьюсер по команде "фас" отработал и вернул новое состояние -> компонент(ы) обновились;

Redux вдохновлен Flux методологией и языком программирования Elm

Под капотом, Redux использует старую фичу реакта - **context**, которая обрела вторую жизнь в версии реакта 16.3 - "New context API".

Есть старый context, который использует Redux, и есть новое Context API, не путайте.

Файлы и папки:

Изначально наше приложение в файловом менеджере должно выглядеть так (создайте недостающие директории в src):

```
+-- src
|   +-- actions
|   +-- components
|   +-- containers
|   +-- reducers
|   +-- utils
+-- файлы-от-create-react-app
+-- ...
```

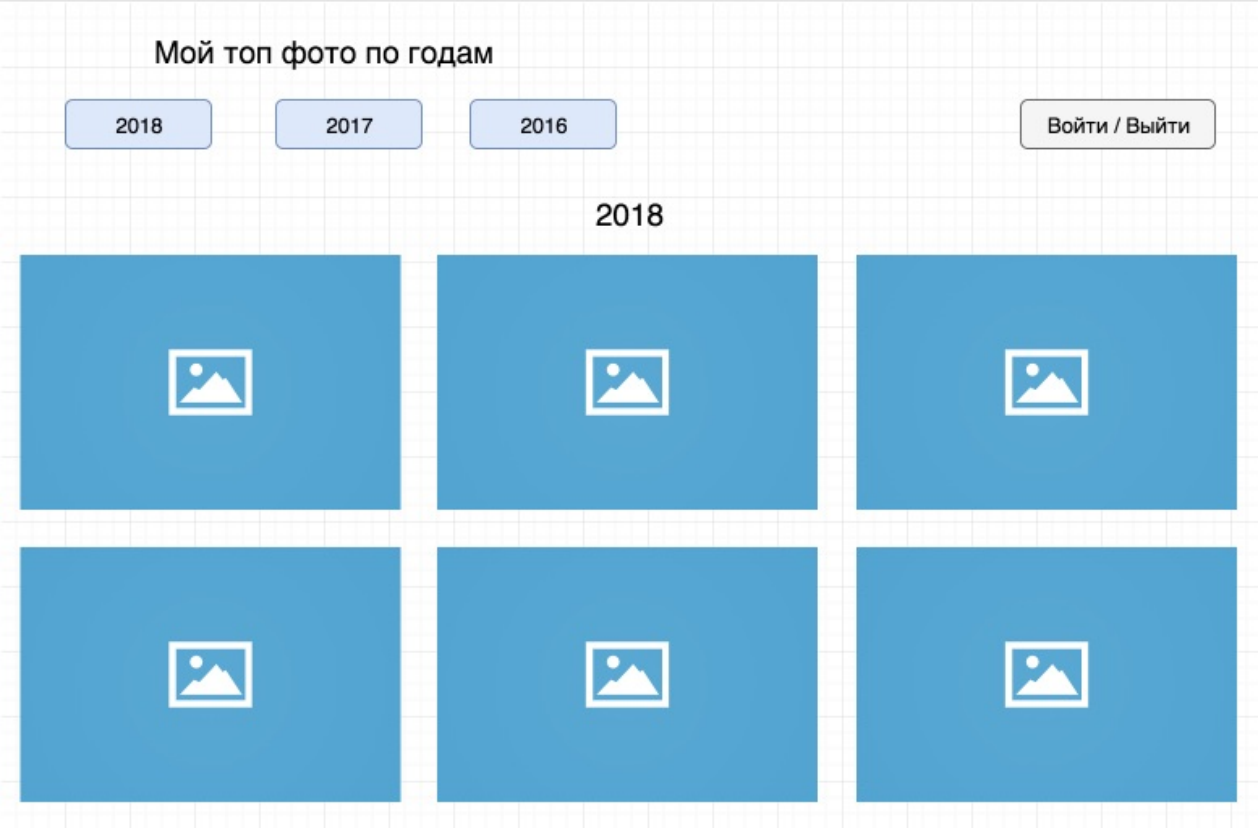
Для обучения мы будем использовать очень распространенный подход организации файлов: деление на контейнеры и компоненты + экшены и редьюсеры в отдельных директориях.

Есть и другие подходы, мне нравится композиция по фичам/страницам (EN).

Основы Redux (теория)

Курс рассчитан на создание приложения по шагам, а это значит максимум практики и минимум теории. Этот 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(ы) (*reducer*) и действия (*actions*), а так же имеет несколько чрезвычайно полезных методов, например:

- `getState()` - позволяет получить состояние приложения;
- `dispatch(action)` - позволяет обновлять состояния, путем вызова ("диспатча") действия;
- `subcribe(listener)` - регистрирует слушателей;

Actions

Actions описывают действия.

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

```
{
    type: 'ЗАГРУЗИ_ФОТО',
    payload: 2018 //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 нашего проекта:

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

Не все любят данный подход с константами, но он был родоначальником, плюс его легко объяснить. К тому же, я до сих пор сторонник этого подхода.

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

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

// я буду использовать синтаксис function внутри actions, так как не вижу смысла
// в изменении его на такую запись:

const getPhotos = (year) => ({
  type: GET_PHOTOS,
  payload: year,
})
```

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

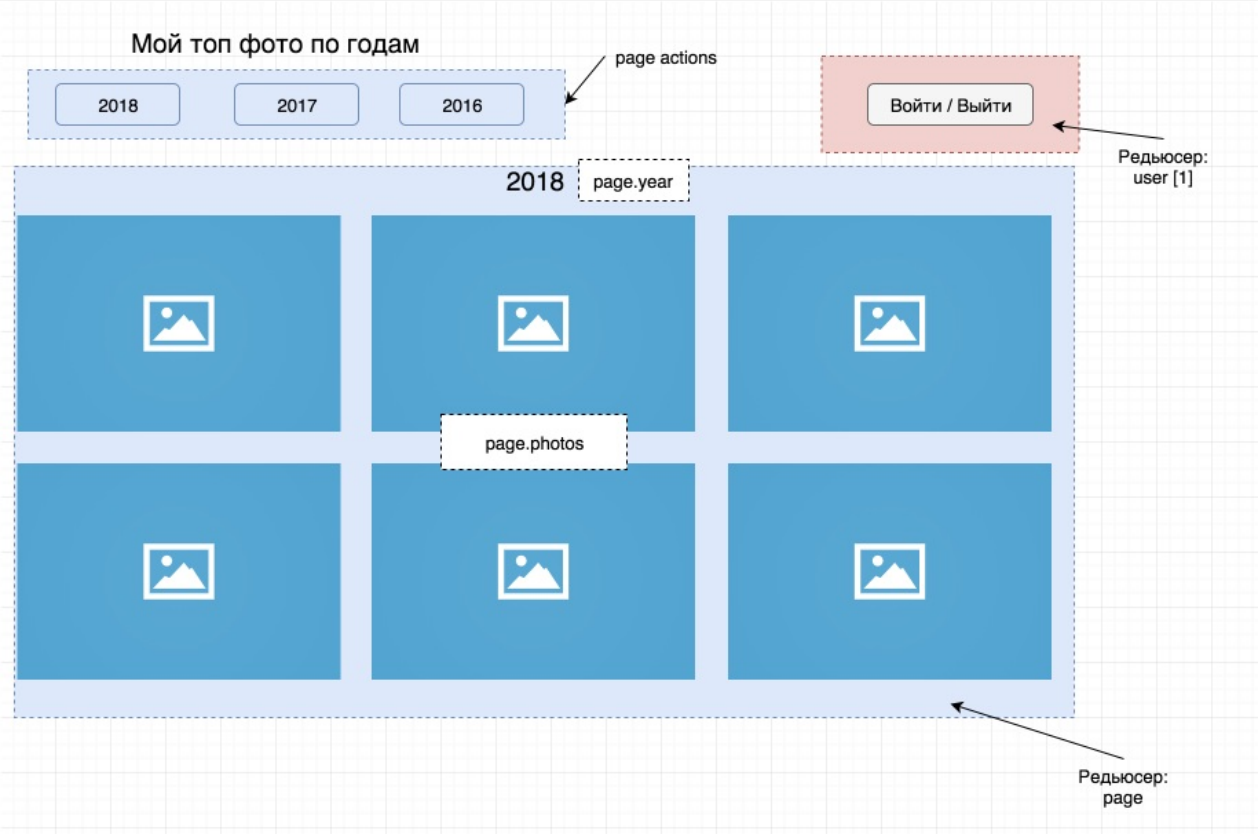
Reducer

"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).

Для того, чтобы научиться комбинировать редьюсеры, мы добавим в приложение reducer - *user*, который просто будет отображать имя, если пользователь залогинился. Ниже на схеме - сноска [1].

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.photos`

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.

Итого: Redux - однонаправленный поток данных в вашем приложении. Случилось действие от юзера - полетел экшен, экшен был пойман редьюсером - изменились пропсы у React-компонента -> компонент перерисовался.

Точка входа

(*Вы можете взять ветку из репозитория, который мы создали в процессе настройки для старта выполнения урока. Практика очень важна.*)

Подтянем Redux и react-redux в наш проект:

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

Точка входа в наше приложение - src/index.js

Обновим его содержимое:

src/index.js

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

import registerServiceWorker from './registerServiceWorker'

import './index.css'

const store = createStore(() => {}, {}) // [1]

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

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

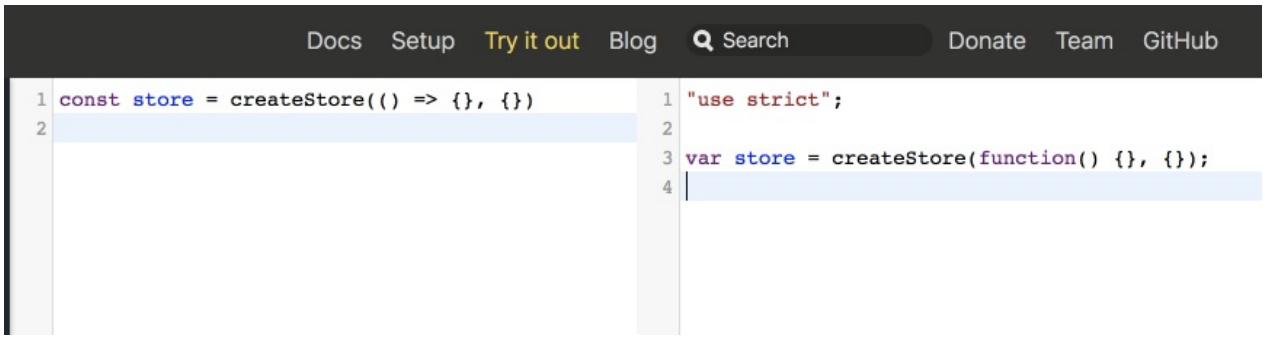
Благодаря этому компоненту, мы сможем получать необходимые данные из store нашего приложения, если воспользуемся вспомогательной функцией `connect`, речь о которой пойдет далее. Сейчас нам и получать нечего, так как store у нас - пустой объект.

Давайте подробнее посмотрим на строку [1]:

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

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

На скриншоте ниже: слева - современный код, справа - старый ES5 код, после преобразования.



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, так как в нем мы добавим позже несколько строк кода, в том числе, добавим усилителей (*enhancers*).

src/store/configureStore.js

```
import { createStore } from 'redux'

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

Поправить импорт в индексе:

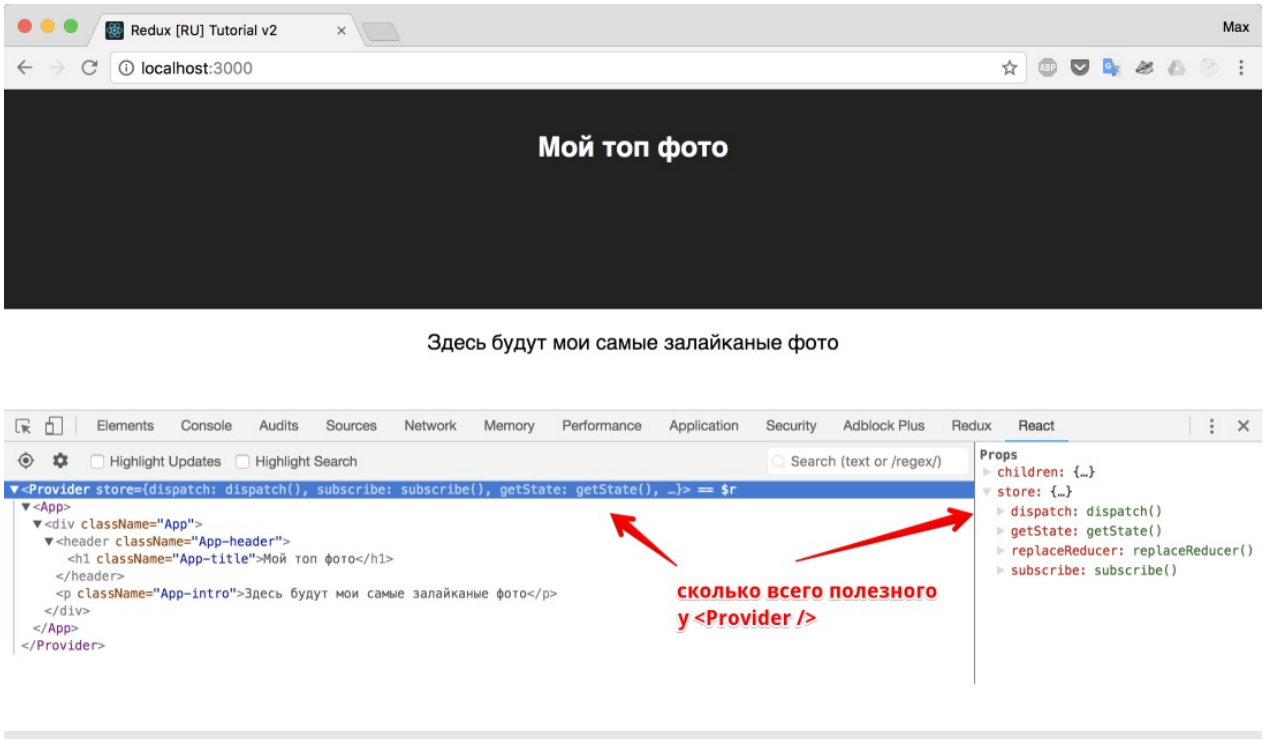
src/index.js

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

import registerServiceWorker from './registerServiceWorker'

import './index.css'

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

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.

Давайте так же исправим *App.js*, чтобы обозначить чем мы тут с вами занимаемся:

```
import React, { Component } from 'react'
import './App.css'

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Мой топ фото</h1>
        </header>
        <p className="App-intro">Здесь будут мои самые залайканые фото</p>
      </div>
    )
  }
}

export default App
```



Итого: мы настроили точку входа для redux-приложения (*src/index.js*), в которой обернули все в `<Provider />`. Так же вынесли для будущего удобства настройку *store* в отдельный файл.

Исходный код.

Создание Reducer

Создадим "корневой редьюсер" (rootReducer).

src/reducers/index.js

```
export const initialState = {
  user: 'Unknown User',
}

export function rootReducer(state = initialState) {
  return state
}
```

В этой функции нечего комментировать. Просто возвращается `{user: 'Unknown User'}` (неизвестный пользователь).

В дальнейшем мы будем комбинировать редьюсеры в корневом редьюсере, но сейчас нам важно отобразить имя юзера (*Unknown User*) в компоненте, чтобы вы не заскучали от чтения.

Главное, что нужно сейчас держать в голове: корневой редьюсер - это и есть представление всего нашего состояния приложения (то есть, всего нашего **store**).

Сконфигурируем store:

src/store/configureStore.js

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

export const store = createStore(rootReducer, initialState)
```

Не забывайте, сигнатура функции createStore:

- первый аргумент - функция-обработчик изменений (редьюсер)
- второй аргумент - начальное состояние

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:

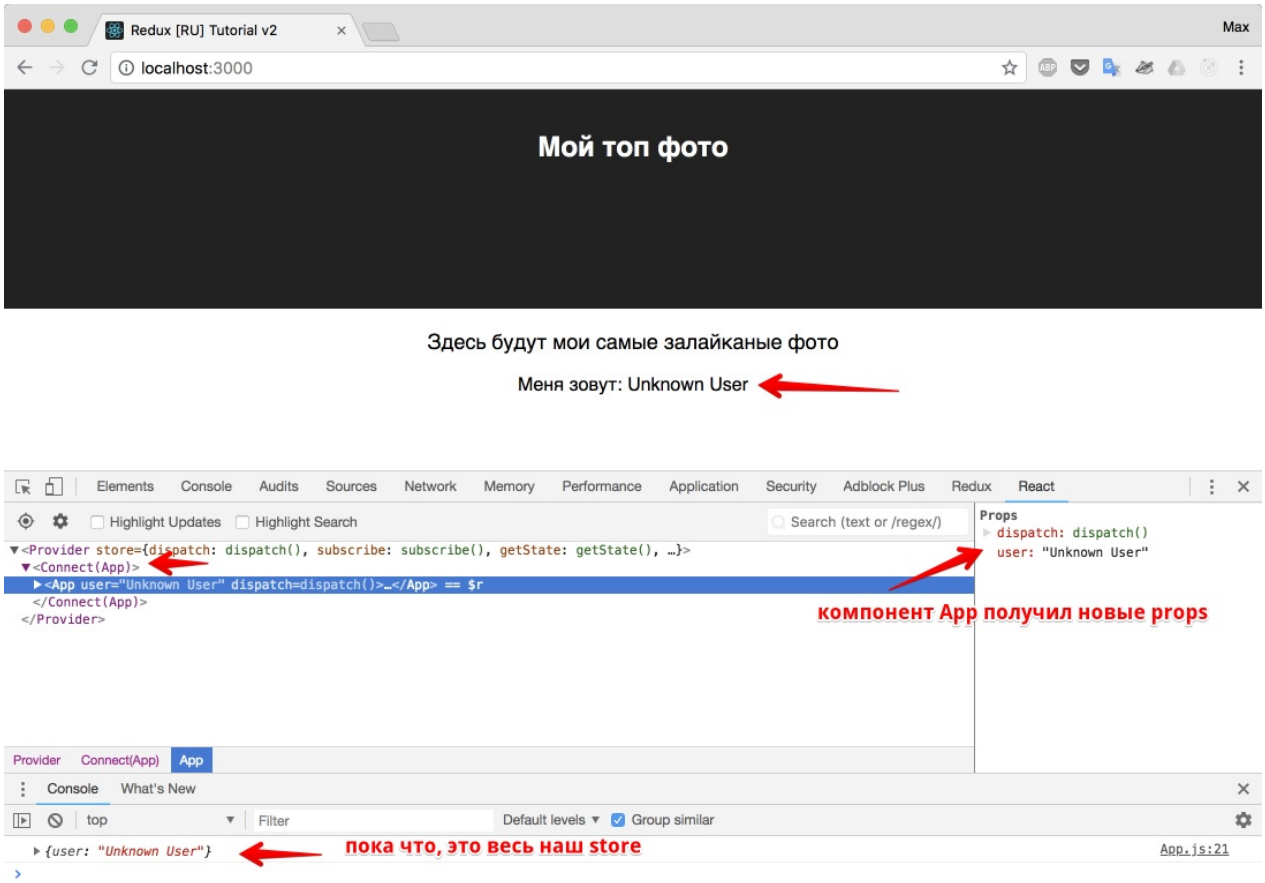
src/App.js

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import './App.css'

class App extends Component {
  render() {
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Мой топ фото</h1>
        </header>
        <p className="App-intro">Здесь будут мои самые залайканые фото</p>
        <p>Меня зовут: {this.props.user}</p> {/* добавлен вывод из props */}
      </div>
    )
  }
}

// приклеиваем данные из store
const mapStateToProps = store => {
  console.log(store) // посмотрим, что же у нас в store?
  return {
    user: store.user,
  }
}

// в наш компонент App, с помощью connect(mapStateToProps)
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.



Взгляните на правую часть скриншота, и вы увидите, что в свойствах (*props*) нашего компонента `<App />` теперь есть метод redux store - **dispatch**, и объект свойств (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

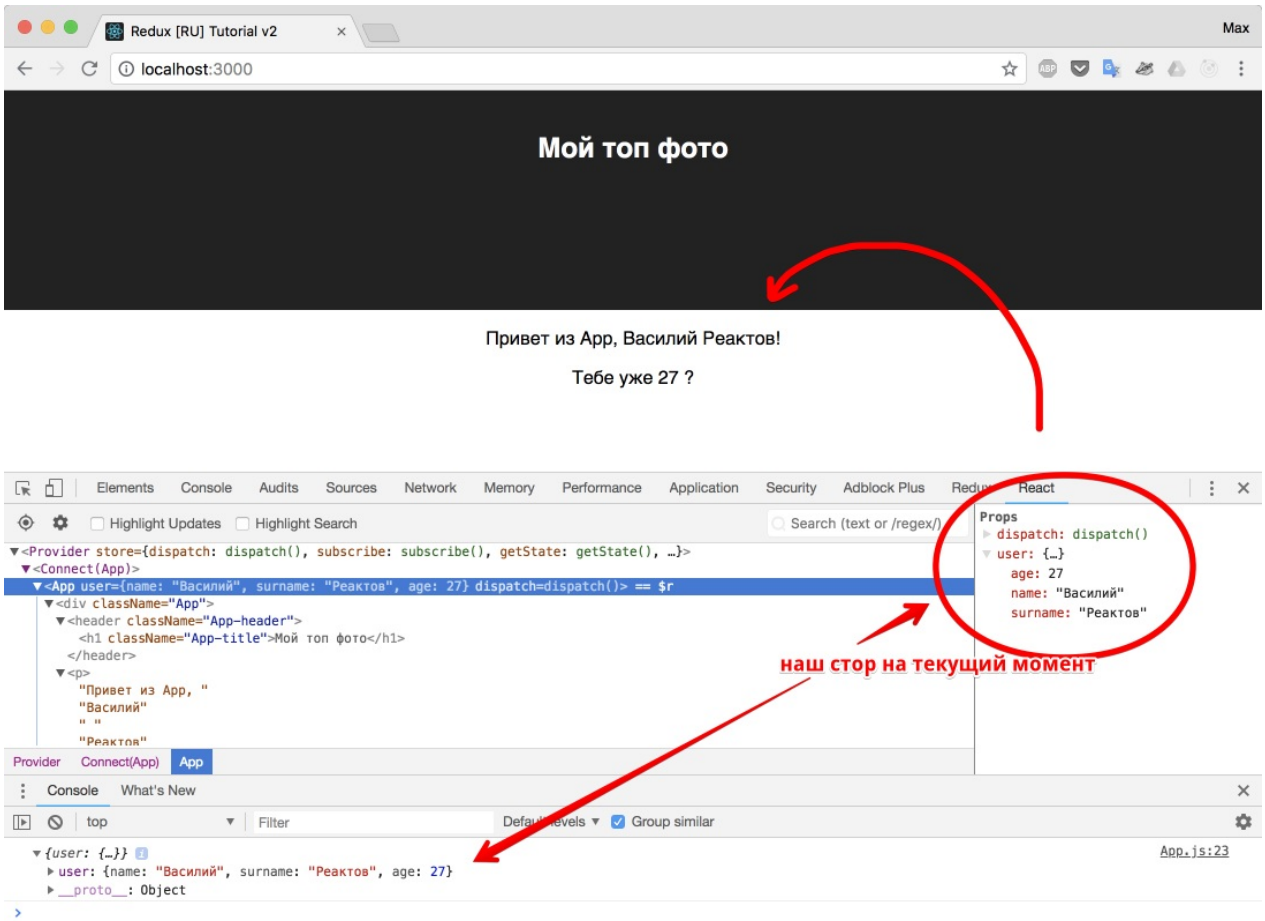
```
export const initialState = {
  user: { // мы вложили в user вместо строки, объект
    name: 'Василий',
    surname: 'Реактов',
    age: 27,
  },
}

export function rootReducer(state = initialState) {
  return state
}
```

затем подкрутим компонент:

src/containers/App.js

```
// ... (импорты)

class App extends Component {
  render() {
    const { name, surname, age } = this.props.user
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Мой топ фото</h1>
        </header>
        <p>
          Привет из App, {name} {surname}!
        </p>
        <p>Тебе уже {age} ?</p>
      </div>
    )
  }
}

// ...  (mapStateToProps и connect - не изменились)
```

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



Итого: мы научились "вытаскивать" данные из стора в компонент, с помощью `connect`.

Исходный код на текущий момент.

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 в следующем уроке.

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 из предыдущих глав):

- user
- page

Создадим их:

src/reducers/user.js

```
const initialState = {
  name: 'Аноним',
}

export function userReducer(state = initialState) {
  return state
}
```

src/reducers/page.js

```
const initialState = {
  year: 2018,
  photos: [],
}

export function pageReducer(state = initialState) {
  return state
}
```

Обновим точку входа для редьюсеров:

src/reducers/index.js

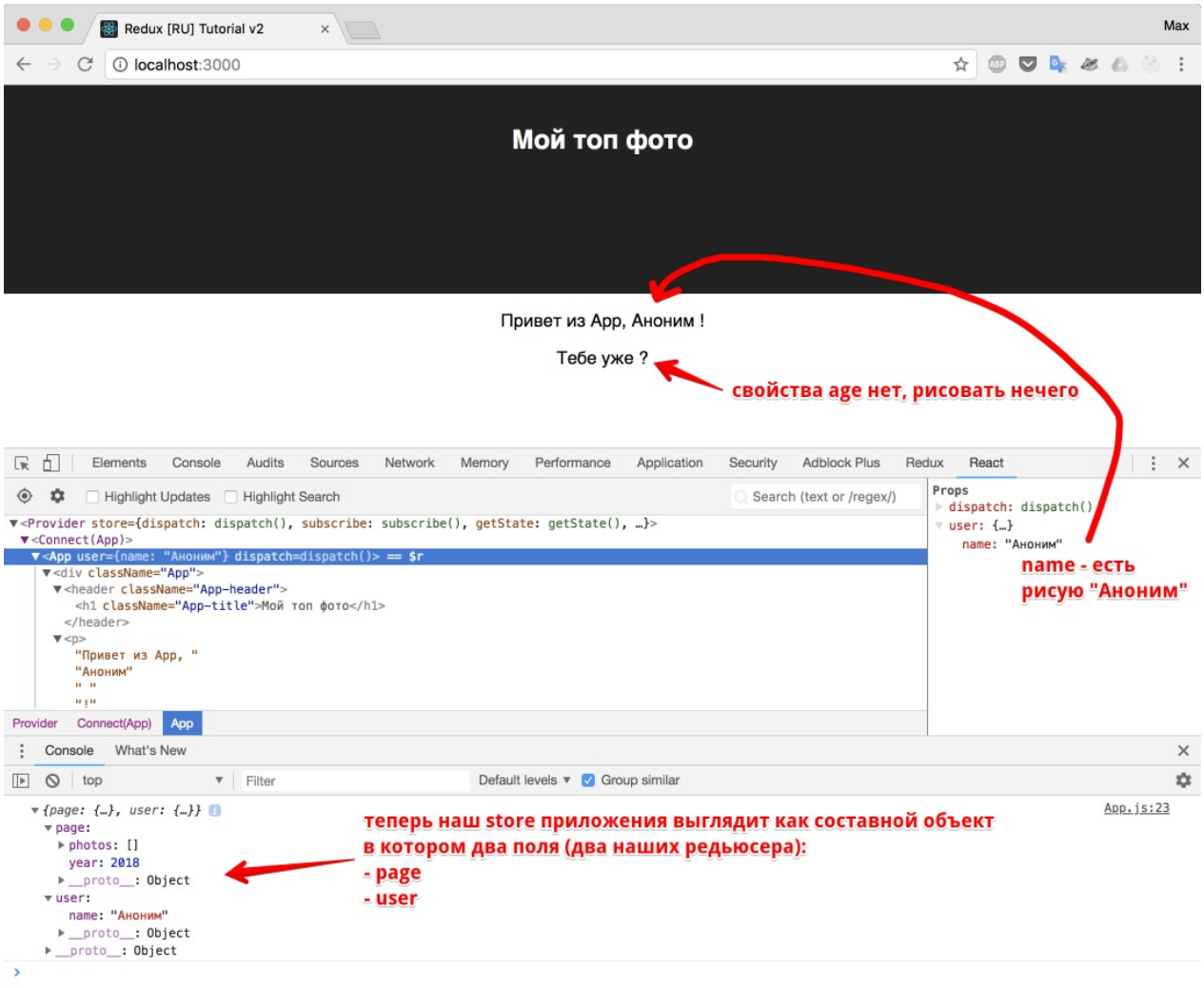
```
import { combineReducers } from 'redux'
import { pageReducer } from './page'
import { userReducer } from './user'

export const rootReducer = combineReducers({
  page: pageReducer,
  user: userReducer,
})
```

Обновим configureStore:

src/store/configureStore.js

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

// удалили "начальное состояние = initial state"
// так как теперь наш редьюсер составной,
// и нам нужны initialState каждого редьюсера.
// Это будет сделано автоматически.
export const store = createStore(rootReducer)
```

Посмотрим что у нас теперь "консолится" в компоненте `<App />`, а так же в React dev tools.



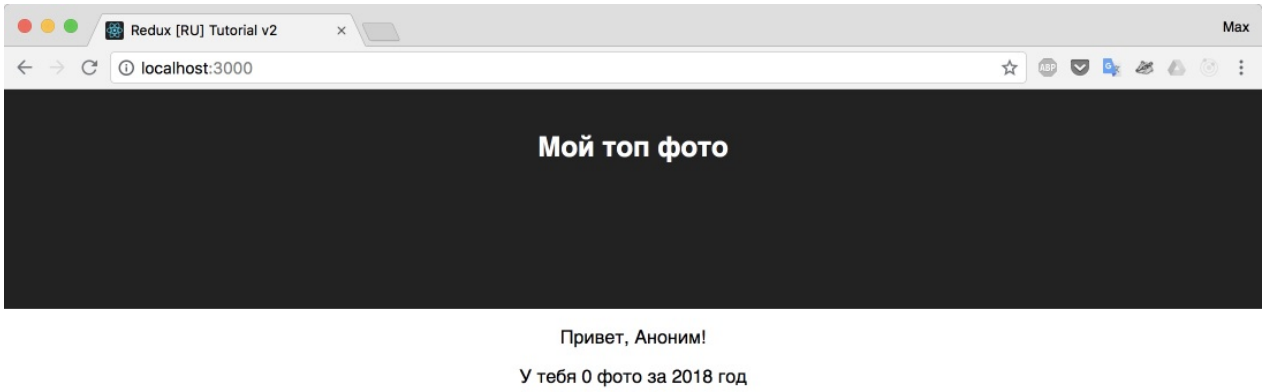
Сейчас в браузере у нас нерабочее приложение. В чем же проблема?

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

```
const mapStateToProps = store => {
  console.log(store)
  return {
    user: store.user,
  }
}
```

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

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

```
...
class App extends Component {
  render() {
    const { user, page } = this.props
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Мой топ фото</h1>
        </header>
        <p>Привет, {user.name}!</p>
        <p>
          У тебя {page.photos.length} фото за {page.year} год
        </p>
      </div>
    )
  }
}

const mapStateToProps = store => {
  console.log(store)
  return {
    user: store.user,
    page: store.page,
  }
}
...
```

Работа функции *mapStateToProps* многих вводит в ступор. В данной функции, мы хотим отрезать от нашего общего пирога (Store) только те кусочки (редьюсеры), которые нам нужны.

Еще можно применить аналогию: мы приклеиваем в props компонента, данные из тех редьюсеров, которые нам требуются.

А если быть более точным, то мы не только получаем в this.props.XXX данные, которым нам нужны, но мы еще и **подписываемся на изменение этих данных**.

После того, как вы знаете о подписке, пора вам раскрыть еще один козырь - когда мы подписываемся только на нужные редьюсеры в компоненте, перерисовка происходит только в случае изменения конкретно этих данных. Если же мы бы подписались просто на весь корневой редьюсер, то не важно в каком бы редьюсере изменились данные - все подписанные на корневой редьюсер компоненты обновились бы.

Опять же, в теории это все абсолютно не "зайдет" не подготовленному читателю. Поэтому на практике мы еще не раз разберем данную информацию.

Итого: 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).

Исходный код на текущий момент.

Полезные ссылки:

- combineReducers (офф.документация)

Контейнеры и компоненты

Прежде чем мы разобьем App.js на компоненты `<User />` и `<Page />` хотелось бы отметить про способ разделения на "компоненты" и "контейнеры", иначе называемый: деление на "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 (*dispatch*) 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!

Установим prop-types и создадим компоненты.

```
npm install --save prop-types
```

src/components/User.js

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

export class User extends React.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 from 'react'
import PropTypes from 'prop-types'

export class Page extends React.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 - это контейнер (так как подключен к redux). 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'

import './App.css'

class App extends Component {
  render() {
    const { user, page } = this.props
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Мой топ фото</h1>
        </header>
        <User name={user.name} />
        <Page photos={page.photos} year={page.year} />
      </div>
    )
  }
}

const mapStateToProps = store => {
  return {
    user: store.user,
    page: store.page,
  }
}

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

Не забудьте так же перенести *App.css* в *src/containers* и поменять подключение `<App />` в index.js:

```
...
import App from './containers/App' // изменили путь
...
```

Так же удалите файл с тестом - *App.test.js*, так как тесты в данный момент не входят в нашу программу, но возможно, будут добавлены в конце в раздел рецептов. Так или иначе, на сайте есть подробнейшая статья тестирование компонентов с помощью jest и enzyme.

Итого: изучили на практике деление на компоненты и контейнеры.

Исходный код

Создание actions

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

В нашем приложении по клику на кнопку года мы должны:

- установить заголовок
- загрузить фото этого года из VK

Сейчас предлагаю рассмотреть установку заголовка, так как 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 начальное состояние (*initial state*)
2. Пользователь нажав кнопку, отправил действие (*dispatch action*)
3. Соответсвующий 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 actions (действия для сущности page):

src/actions/PageActions.js

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

Напоминаю, что поле type - обязательное, а payload - "негласное" соглашение. Немного об этом, можно почитать на английском тут.

Научим редьюсер page реагировать на наше действие:

src/reducers/page.js

```
const initialState = {
  year: 2018,
  photos: [],
}

export function pageReducer(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 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, который был послан в action.payload).

Добавляем вызов actions из компонентов

(возможно, вы будете замечать у меня или в других руководствах, что говорят: стейт (state) приложения - это тоже самое, что и "стор" (store) приложения. Во втором издании учебника я везде стараюсь писать store, чтобы вы не путались со стейтом реакт компонента)

У нас есть *action*, и есть *reducer* готовый изменить *store* приложения. Но наш компонент не знает как обратиться к необходимому действию.

Согласно таблице из прошлого раздела: для изменения данных, наш *компонент* `<Page />`, должен вызывать *callback* из *this.props*, а наш контейнер* `<App />` - 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, мы видим, что с помощью этой функции можно не только подписаться на обновления данных (**mapStateToProps**), но и "прокинуть" наши *actions* в контейнер (**mapDispatchToProps**).

`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 { connect } from 'react-redux'
import { User } from '../components/User'
import { Page } from '../components/Page'
import { setYear } from '../actions/PageActions'

import './App.css'

class App extends Component {
  render() {
    const { user, page, setYearAction } = this.props
    return (
      <div className="App">
        <header className="App-header">
          <h1 className="App-title">Мой топ фото</h1>
        </header>
        <User name={user.name} />
        <Page photos={page.photos} year={page.year} setYear={setYearAction} />
      </div>
    )
  }
}

const mapStateToProps = store => {
  return {
    user: store.user,
    page: store.page,
  }
}

const mapDispatchToProps = dispatch => {
  return {
    setYearAction: year => dispatch(setYear(year)), [1]
  }
}

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

Начнем с разбора `mapDispatchToProps`. Данная функция, первым аргументом получает `dispatch`, а значит мы можем теперь "диспатчить" экшены, которые будут пойманы редьюсером. Еще раз:только те экшены, которые были отправлены с помощью "диспетчера" будут пойманы редьюсером.

Затем мы внутри *mapDispatchToProps* вернули объект, который в итоге приклеится в this.props (так же, как и было в *mapStateToProps*).

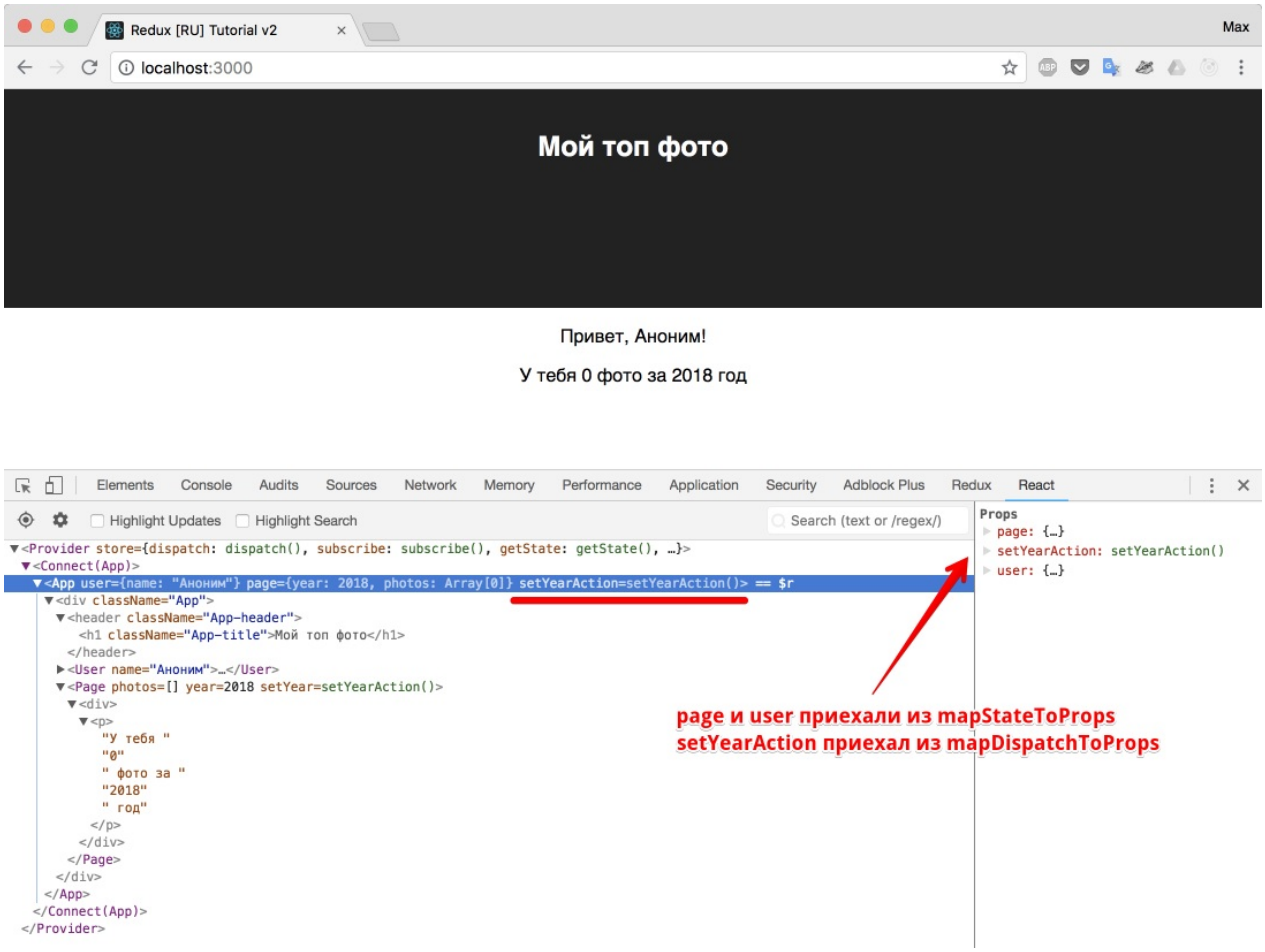
И в конце, мы решили в "приклееном объекте" создать функцию *setYearAction* [1], суть которой сводится к следующему: "диспатчни" импортированный выше *setYear* с переданным годом.

Обычно пишут без добавления Action: `setYear: year => dispatch(setYear(year))`, но я хотел бы уменьшить путаницу для тех людей, кто не силен в основах JavaScript (а зря!) и сейчас может быстро запутаться.

Так же я пишу `return`, для того, чтобы вы могли удобно сконсолить значения аргументов, если вам что-то не понятно. Без return, можно написать так:

```
const mapDispatchToProps = dispatch => ({
  setYearAction: year => dispatch(setYear(year)),
})
```

После выполнения `connect(mapStateToProps, mapDispatchToProps)(App)`, мы получили в `<App />` новые свойства (*props*), что наглядно демонстрирует вкладка "React" в chrome dev tools.



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

Добавим несколько кнопок с годами и обработчик клика на них, в котором будем считывать название года с самой кнопки и отправлять его с помощью экшена `'SET_YEAR'` прямиком в редьюсер `page`.

src/components/Page.js

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

export class Page extends React.Component {
  onBtnClick = e => {
    const year = +e.currentTarget.innerText
    this.props.setYear(year)
  }
  render() {
    const { year, photos } = this.props
    return (
      <div>
        <div>
          <button onClick={this.onBtnClick}>2018</button>
          <button onClick={this.onBtnClick}>2017</button>
          <button onClick={this.onBtnClick}>2016</button>
          <button onClick={this.onBtnClick}>2015</button>
          <button onClick={this.onBtnClick}>2014</button>
        </div>
        <p>
          У тебя {photos.length} фото за {year} год
        </p>
      </div>
    )
  }
}

Page.propTypes = {
  year: PropTypes.number.isRequired,
  photos: PropTypes.array.isRequired,
  setYear: PropTypes.func.isRequired, // добавили новое свойство в propTypes
}
```

Сейчас если кликнуть на кнопку с годом, то в приложении год будет изменяться. Вау?)

Что происходит: по клику на кнопку, вызывается переданное в свойствах функция, в которой диспатчится экшен (с типом `SET_YEAR` и годом). Затем, так как этот экшен был "диспатчнут" он пролетает через все редьюсеры (у нас их два: user и page). Так как в

page есть `case 'SET_YEAR'` - редьюсер возвращает новое состояние, а именно - берет все что было в нашем state (по факту - все что было в данном "куске пирога" от стора связанное с *page*) и возвращает новое значение года:

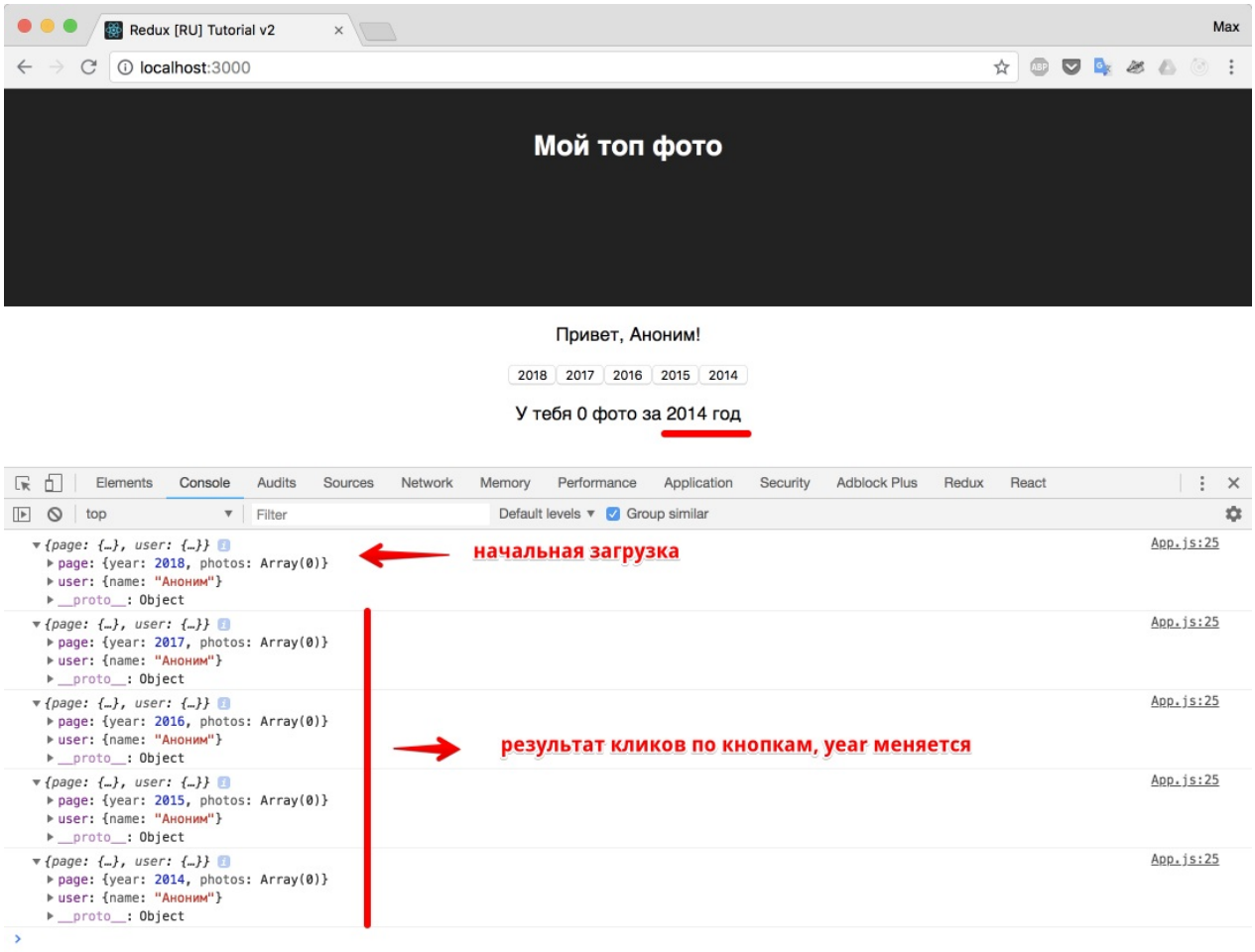
```
case 'SET_YEAR':
  return { ...state, year: action.payload }
```

Затем, так как год изменился, в компоненте `<App />` случилось обновление, так как внутри *mapStateToProps* мы подписаны на изменение данных из редьюсера `page`. Раз случилось обновление, значит перерисовались все потомки и в том числе, в компонент `<Page />` ушло новое значение года.

p.s. в коде было использовано свойство DOM-элемента textContent

p.p.s. можете добавить `console.log(store)` в mapStateToProps и посмотреть есть ли новые данные.

```
const mapStateToProps = store => {
  console.log(store)
  return {
    user: store.user,
    page: store.page,
  }
}
```



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 **однонаправленного** 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.

Итого: мы научились обновлять Redux-приложение правильно: диспатчить экшен и реагировать на экшен в редьюсере.

Исходный код

Константы

Константы одна из первых тем для холиваров. Как их лучше делать, где размещать и так далее. Мы возьмем такой способ: будем определять константу рядом с экшеном.

src/actions/PageActions.js

```
export const SET_YEAR = 'SET_YEAR' // положили строку в константу

export function setYear(year) {
  return {
    type: SET_YEAR, // изменили строку на константу
    payload: year,
  }
}
```

Подключим константу в редьюсер Page

src/reducers/page.js

```
import { SET_YEAR } from '../actions/PageActions'

const initialState = {
  year: 2018,
  photos: [],
}

export function pageReducer(state = initialState, action) {
  switch (action.type) {
    case SET_YEAR: // изменили строку на константу
      return { ...state, year: action.payload }

    default:
      return state
  }
}
```

В дальнейшем мы будем придерживаться такого подхода и добавлять константы для всех типов наших экшенов. Зачем мы это делаем, сказать сложно. Попробую придумать пример: вы решили, что все ваши типы теперь должны быть составными строками `module name/action type`, получается для SET_YEAR будет:

```
const SET_YEAR = 'page/SET_YEAR'
```

При подходе с константами, вам потребуется изменит код лишь в одном месте (в определении константы).

Итого: превратили строковое значение в константу и познакомились с данным подходом организации типов экшенов.

Исходный код.

Наводим порядок

Прежде чем мы начнем собирать воедино паззл из полученных знаний в осмысленное приложение, предлагаю сразу раскидать стили и верстку, так как они не являются темами для обсуждения в подробностях.

Стили

Обычно файл со стилями кладут в то же место, где находится и компонент. У нас же для простоты - все стили будут храниться в index.css

src/index.css

```
/* http://meyerweb.com/eric/tools/css/reset/
   v2.0 | 20110126
   License: none (public domain)
*/

html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video {
  margin: 0;
  padding: 0;
  border: 0;
  font-size: 100%;
  font: inherit;
  vertical-align: baseline;
}
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
  display: block;
}
body {
```

```
  line-height: 1;
  font-family: sans-serif;
}
ol, ul {
  list-style: none;
}
blockquote, q {
  quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
  content: '';
  content: none;
}
table {
  border-collapse: collapse;
  border-spacing: 0;
}

/*end reset*/

h3 {
  font-size: 22px;
  margin: 10px 0 0;
}
.app {
  margin: 50px;
  font: 14px sans-serif;
}
.ib {
  display: inline-block;
}
.page {
  width: 80%;
}
.user {
  width: 20%;
  vertical-align: top;
}

.btn {
  border: none;
  border-radius: 2px;
  display: inline-block;
  height: 36px;
  line-height: 36px;
  font-size: 16px;
  outline: 0;
  padding: 0 2rem;
  text-transform: uppercase;
  vertical-align: middle;
  color: #fff;
  background-color: #6383a8;
```

```
    text-align: center;
    letter-spacing: .5px;
    transition: .2s ease-out;
    cursor: pointer;
}
.btn:hover {
    background-color: #6d8cb0
}
```

Так же, очень популярен подход с использованием styled-components

Во многих проектах, вы можете встретить библиотеку classnames для организации стилей по условию.

Верстка

Здесь двояко: с одной стороны, верстка в реакте та же самая, с другой стороны - верстальщик, который понимает как работает react, стряпает компоненты гораздо чище (старается держать все компоненты "тупыми" и может сам написать простые условия).

Для нашего приложения измененная верстка и стили дадут следующий эффект:

Наконец-то наше приложение стало похоже на схему :)

Так как вопросы верстки и стилей не являются темой нашего обучения, вы можете скопировать исходный код, либо сделать как вам хочется.

Напоминаю: в реальном приложении, лучше держать стили рядом с компонентом, чтобы можно было удобно переиспользовать компоненты между приложениями.

Итого: наше приложение похоже на схему. Автор выдал несколько ссылок на организацию CSS и смылся.

Исходный код.

Middleware (Усилители).

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

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 усилитей 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 const store = createStore(rootReducer, applyMiddleware(ping)) // <-- добавляем
его в цепочку middleware'ов
```

Напишем усилитель:

store/enhancers/ping.js

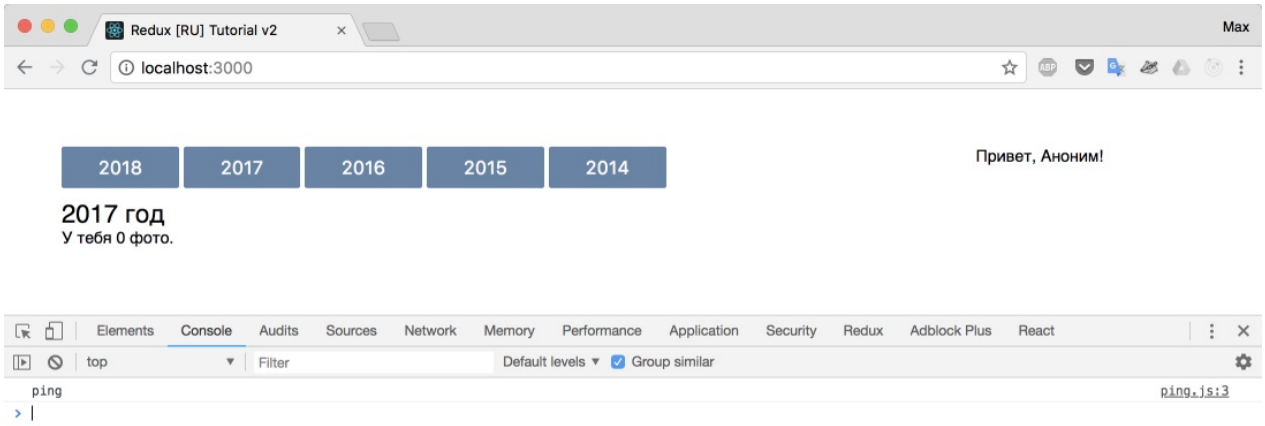
```
/*eslint-disable */
export const ping = store => next => action => {
  console.log('ping')
  return next(action)
}
/*eslint-enable */
```

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 функциях, 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:

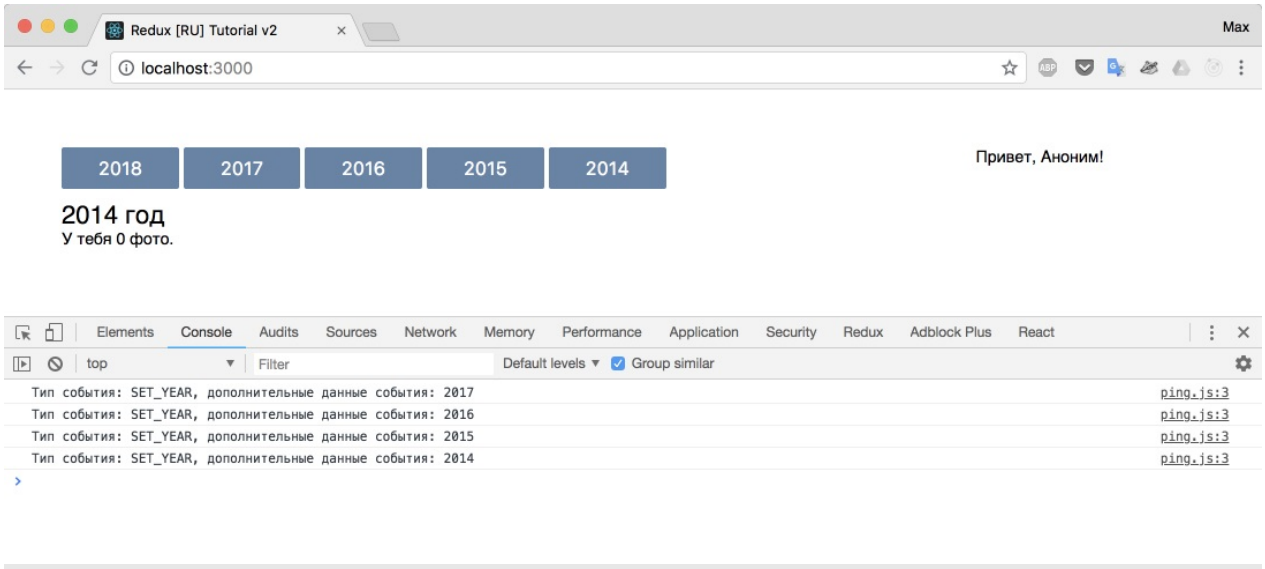
store/enhancers/ping.js

```
/*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: ' + acti
on.payload)
```

Покликайте на кнопки, результат должен быть следующим:



Redux-logger

Отбросим наш велосипед и поставим популярный логгер.

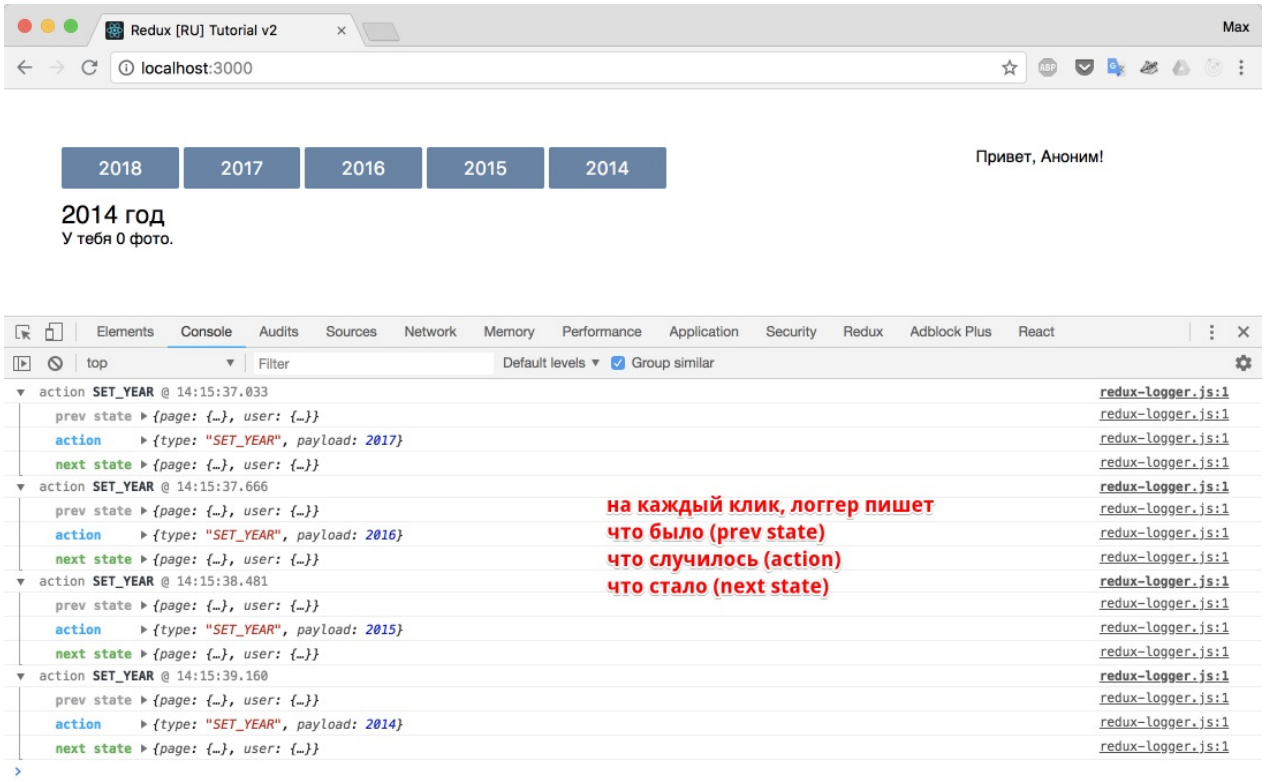
```
npm i --save-dev redux-logger
```

Удалите папку enchancers, и измените configureStore.

src/store/configureStore.js

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

export const store = createStore(rootReducer, applyMiddleware(logger))
```

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 и будем использовать для создания асинхронных действий.

Исходный код на текущий момент.

Асинхронные actions

Давайте представим синхронное действие:

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

Давайте представим асинхронное действие:

- Пользователь кликнул на кнопку
- dispatch action `{type: ТИП_ДЕЙСТВИЯ_ЗАПРОС}`
- запрос выполнился успешно
  - dispatch action `{type: ТИП_ДЕЙСТВИЯ_УСПЕШНО, payload: доп.данные}`
- запрос выполнился неудачно
  - 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

```
export const SET_YEAR = 'SET_YEAR'

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`, чтобы можно было диспатчить события в момент, когда они совершились. 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, мы знаем, что в цепочке middleware 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 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;
```

Нам остается лишь добавить зависимость в наш проект.

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

И добавить redux-thunk в цепочку усилителей перед логгером, так как логгер должен быть последним усилителем в цепочке.

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

export const store = createStore(rootReducer, applyMiddleware(thunk, logger))
```

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, будем 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, *isFetching*:

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

Решение ниже.

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

```
export const GET_PHOTOS_REQUEST = 'GET_PHOTOS_REQUEST'
export const GET_PHOTOS_SUCCESS = 'GET_PHOTOS_SUCCESS'
export function getPhotos(year) {
  return dispatch => {
    // экшен с типом REQUEST (запрос начался)
    // диспатчится сразу, как будто-бы перед реальным запросом
    dispatch({
      type: GET_PHOTOS_REQUEST,
      payload: year,
    })

    // а экшен внутри setTimeout
    // диспатчится через секунду
    // как будто-бы в это время
    // наши данные загружались из сети
    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 '../actions/PageActions'

const initialState = {
  year: 2018,
  photos: [],
  isFetching: false, // изначально статус загрузки - ложь
  // так как он станет true, когда запрос начнет выполнение
}

export function pageReducer(state = initialState, action) {
  switch (action.type) {
    case GET_PHOTOS_REQUEST:
      return { ...state, year: action.payload, isFetching: true }

    case GET_PHOTOS_SUCCESS:
      return { ...state, photos: action.payload, isFetching: 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

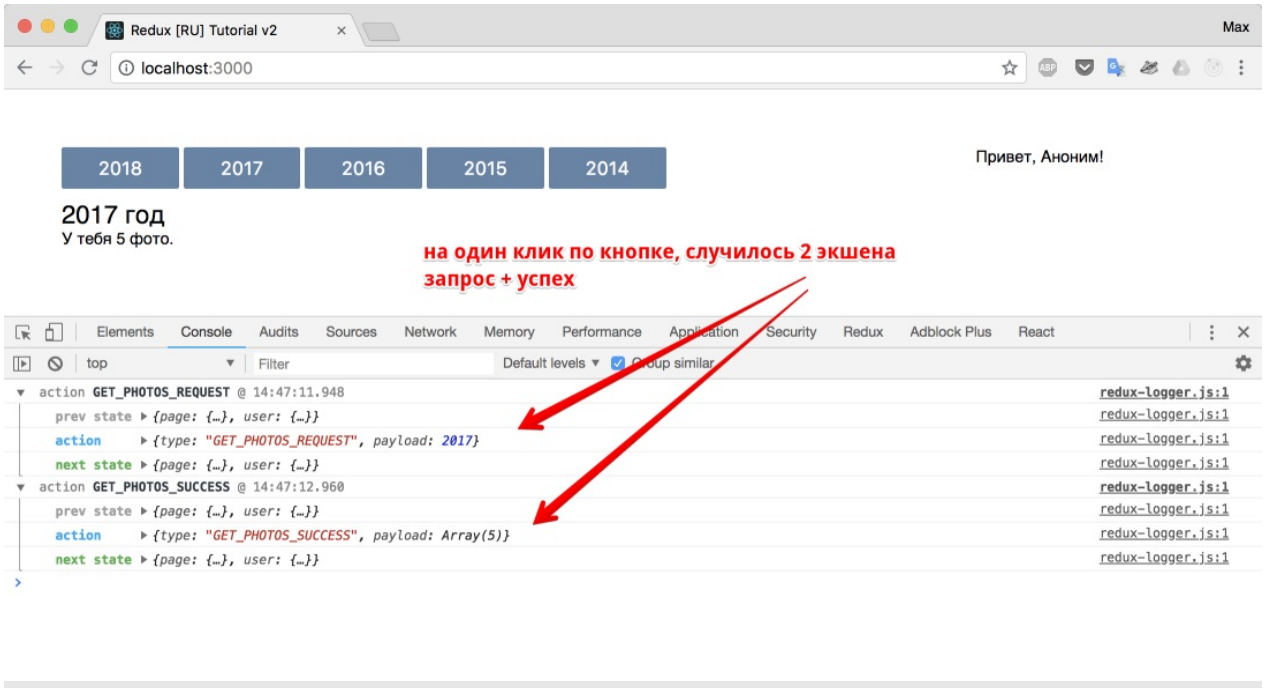
```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { User } from '../components/User'
import { Page } from '../components/Page'
import { getPhotos } from '../actions/PageActions'

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

const mapStateToProps = store => {
  return {
    user: store.user,
    page: store.page,
  }
}

const mapDispatchToProps = dispatch => {
  return {
    getPhotosAction: year => dispatch(getPhotos(year)),
  }
}

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

Обновим соответствующий компонент:

src/components/Page.js

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

export class Page extends React.Component {
  onBtnClick = e => {
    const year = +e.currentTarget.innerText
    this.props.getPhotos(year) // setYear -> getPhotos
  }
  render() {
    const { year, photos, isFetching } = this.props // вытащили isFetching
    return (
      <div className="ib page">
        <p>
          <button className="btn" onClick={this.onBtnClick}>
            2018
          </button>{' '}
          <button className="btn" onClick={this.onBtnClick}>
            2017
          </button>{' '}
          <button className="btn" onClick={this.onBtnClick}>
            2016
          </button>{' '}
          <button className="btn" onClick={this.onBtnClick}>
            2015
          </button>{' '}
          <button className="btn" onClick={this.onBtnClick}>
            2014
          </button>
        </p>
        <h3>{year} год</h3>
        {/* добавили отрисовку по условию */}
        {isFetching ? <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, // setYear -> getPhotos
  // добавили новое свойство - isFetching, причем в propTypes нет boolean, есть bool
  isFetching: PropTypes.bool.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). Для этого придется почитать документацию по работе с VK API.

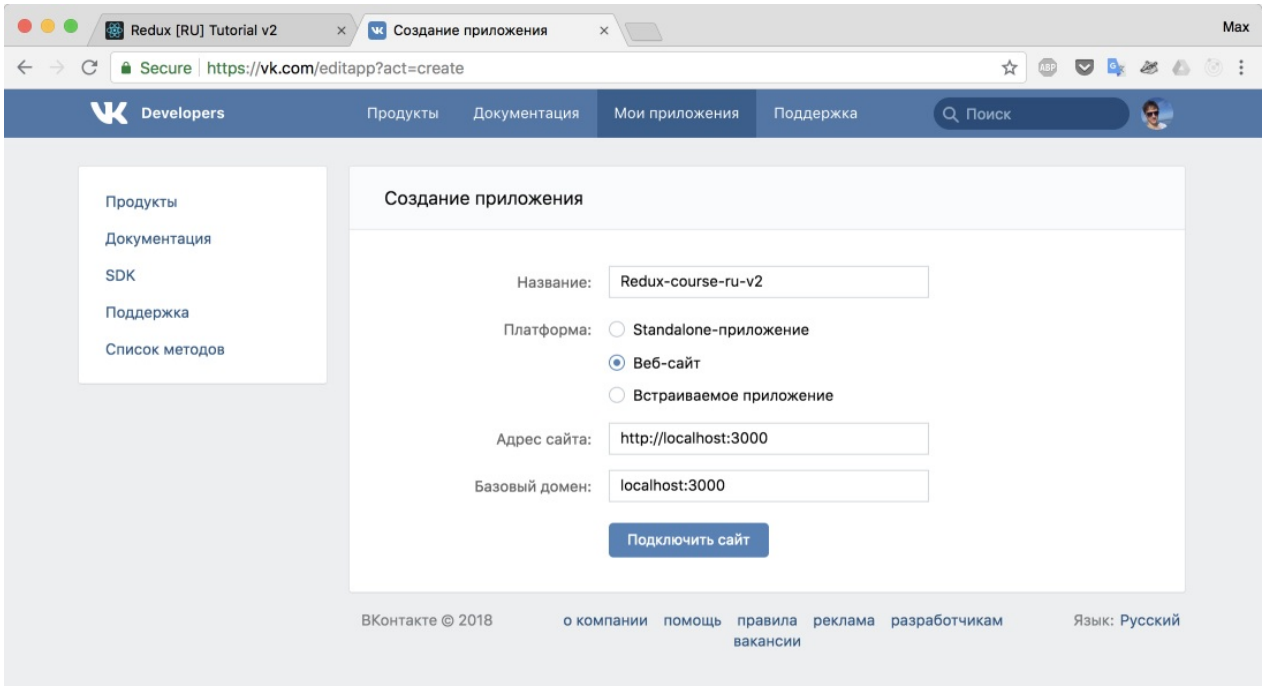
Для тех, кто хочет добить пример поскорее - следующая глава, в которой мы загрузим таки реальные фото из вашего профиля VK.

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.

По адресу https://vk.com/apps?act=manage создайте новое приложение (веб-сайт) и заполните поля как на скриншоте, если используете локалхост и порт 3000.

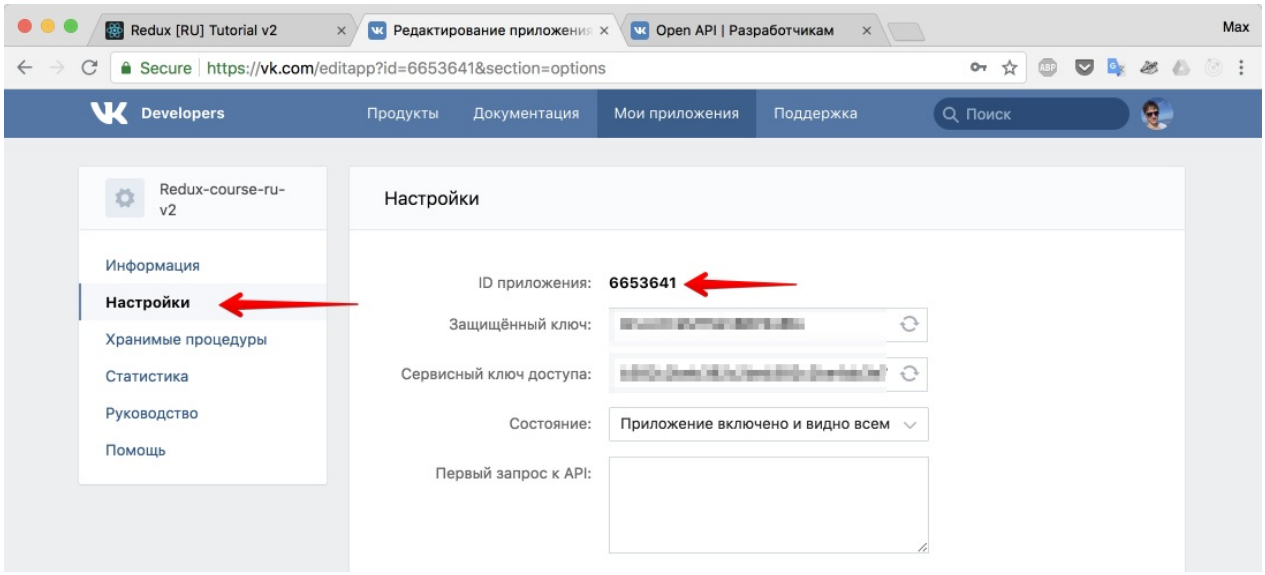


Интеграция VK API

Необходимо добавить скрипт *openapi* (документация), а так же вызвать VK.init

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=
no">
    <meta name="theme-color" content="#000000">
    <link rel="manifest" href="%PUBLIC_URL%/manifest.json">
    <link rel="shortcut icon" href="%PUBLIC_URL%/favicon.ico">
    <title>Redux [RU] Tutorial v2</title>
  </head>
  <body>
    <noscript>
      You need to enable JavaScript to run this app.
    </noscript>
    <div id="root"></div>
    <script src="https://vk.com/js/api/openapi.js?158"></script>
    <script language="javascript">
      VK.init({
        apiId: XXXXXX <!-- ваш номер -->
      });
    </script>
  </body>
</html>
```

Номер приложения можно посмотреть здесь:



Авторизация

Создадим действия для User.

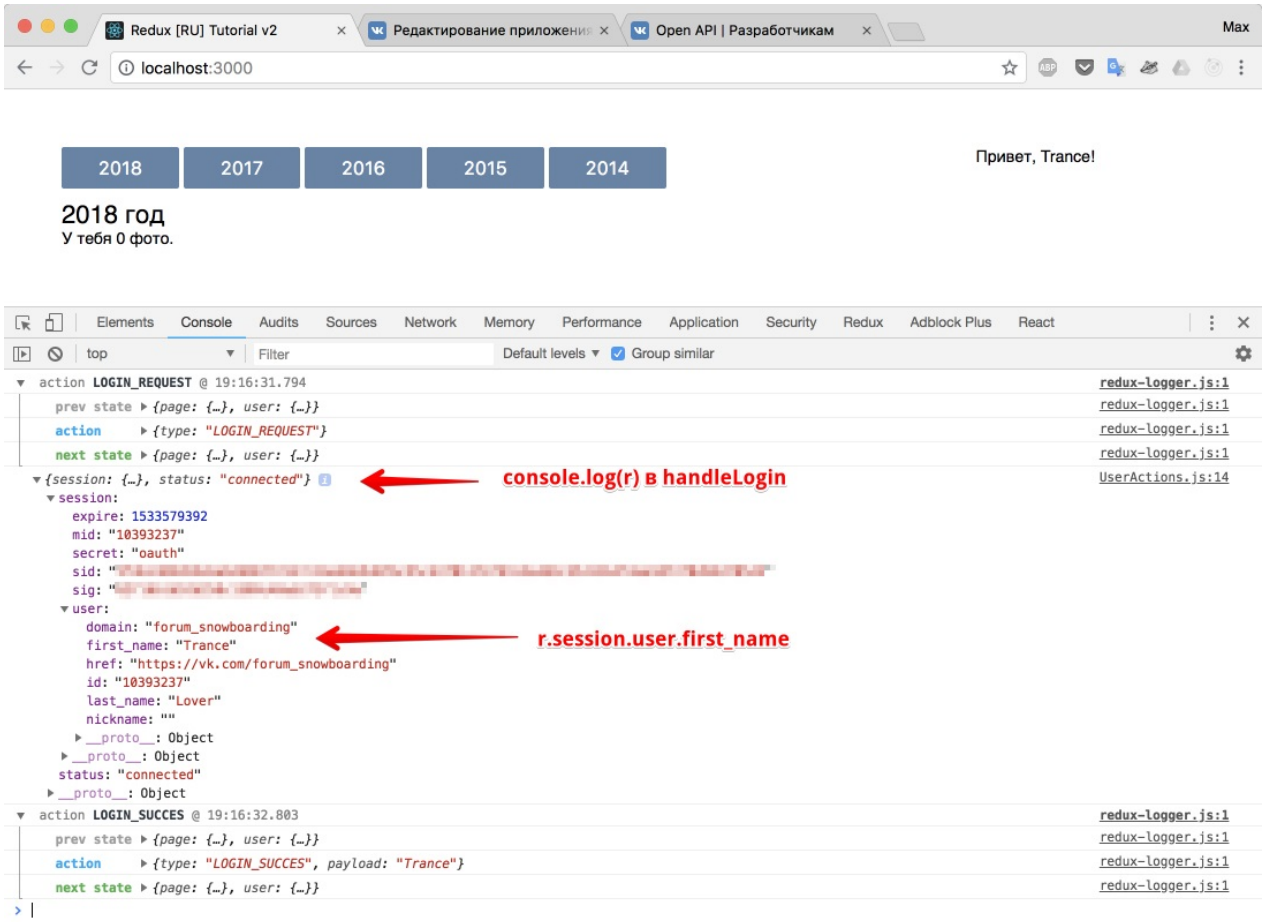
src/actions/UserActions.js

```
export const LOGIN_REQUEST = 'LOGIN_REQUEST'
export const LOGIN_SUCCESS = 'LOGIN_SUCCESS'
export const LOGIN_FAIL = 'LOGIN_FAIL'

export function handleLogin() {
  return function(dispatch) {
    dispatch({
      type: LOGIN_REQUEST,
    })

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

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

Так как загрузка информации из профиля - действие асинхронное, мы использовали проверенную схему из трех действий:

- XXX_REQUEST - диспатчим непосредственно перед стартом реального запроса (для юзера это выглядит, как будто во время запроса)
- XXX_SUCCESS + данные - если все прошло успешно добавляем данные [1]
- XXX_FAIL + ошибка - если что-то пошло не так

[1] Чтобы достать имя пользователя, мы вытащили его из response(r).session. Данные нам предоставил VK, так как мы подтвердили "разрешаю доступ" во всплывающем окне.



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

src/containers/App.js

```
import React, { Component } from 'react'
import { connect } from 'react-redux'
import { User } from '../components/User'
import { Page } from '../components/Page'
import { getPhotos } from '../actions/PageActions'
import { handleLogin } from '../actions/UserActions'

class App extends Component {
  render() {
    // вытащили handleLoginAction из this.props
    const { user, page, getPhotosAction, handleLoginAction } = this.props
    return (
      <div className="app">
        <Page
          photos={page.photos}
          year={page.year}
          isFetching={page.isFetching}
          getPhotos={getPhotosAction}
        />
```

```
        {/* добавили новые props для User */}
        <User
          name={user.name}
          isFetching={user.isFetching}
          error={user.error}
          handleLogin={handleLoginAction}
        />
      </div>
    )
  }
}

const mapStateToProps = store => {
  return {
    user: store.user, // вытащили из стора (из редьюсера user все в переменную thid.pr
ops.user)
    page: store.page,
  }
}

const mapDispatchToProps = dispatch => {
  return {
    getPhotosAction: year => dispatch(getPhotos(year)),
    // "приклеили" в this.props.handleLoginAction функцию, которая умеет диспатчить ha
ndleLogin
    handleLoginAction: () => dispatch(handleLogin()),
  }
}

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

Здесь мы поступили так же, как когда-то для page:

- подписались на кусочек стора (*user*)
- добавили экшен и передали его в *dispatch* в функции *handleLoginAction*
- кусочек стора (*user*) и *handleLoginAction* - стали доступны нам в *this.props*
- в `<User />` передали необходимые свойства

Обновим reducer user:

src/reducers/user.js

```
import { LOGIN_REQUEST, LOGIN_SUCCESS, LOGIN_FAIL } from '../actions/UserActions'

const initialState = {
  name: '',
  error: '', // добавили для сохранения текста ошибки
  isFetching: false, // добавили для реакции на статус "загружаю" или нет
}

export function userReducer(state = initialState, action) {
  switch (action.type) {
    case LOGIN_REQUEST:
      return { ...state, isFetching: true, error: '' }

    case LOGIN_SUCCESS:
      return { ...state, isFetching: false, name: action.payload }

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

    default:
      return state
  }
}
```

В редьюсере есть интересные моменты:

- когда мы начали делать запрос (LOGIN_REQUEST) мы очищаем error. Например, была ошибка, мы стали делать новый запрос - ошибка очистилась;
- если случился LOGIN_SUCCESS - мы в name записываем action.payload (а как вы помните, там мы передаем в строке имя пользователя) и ставим статус загрузки - false (то есть, не загружается, ибо загрузилось);
- если случился LOGIN_FAIL - опять же, загружаю? Нет, значит isFetching - false. Ошибка? Да - запиши в поле error.

Прокачаем `<User />` :

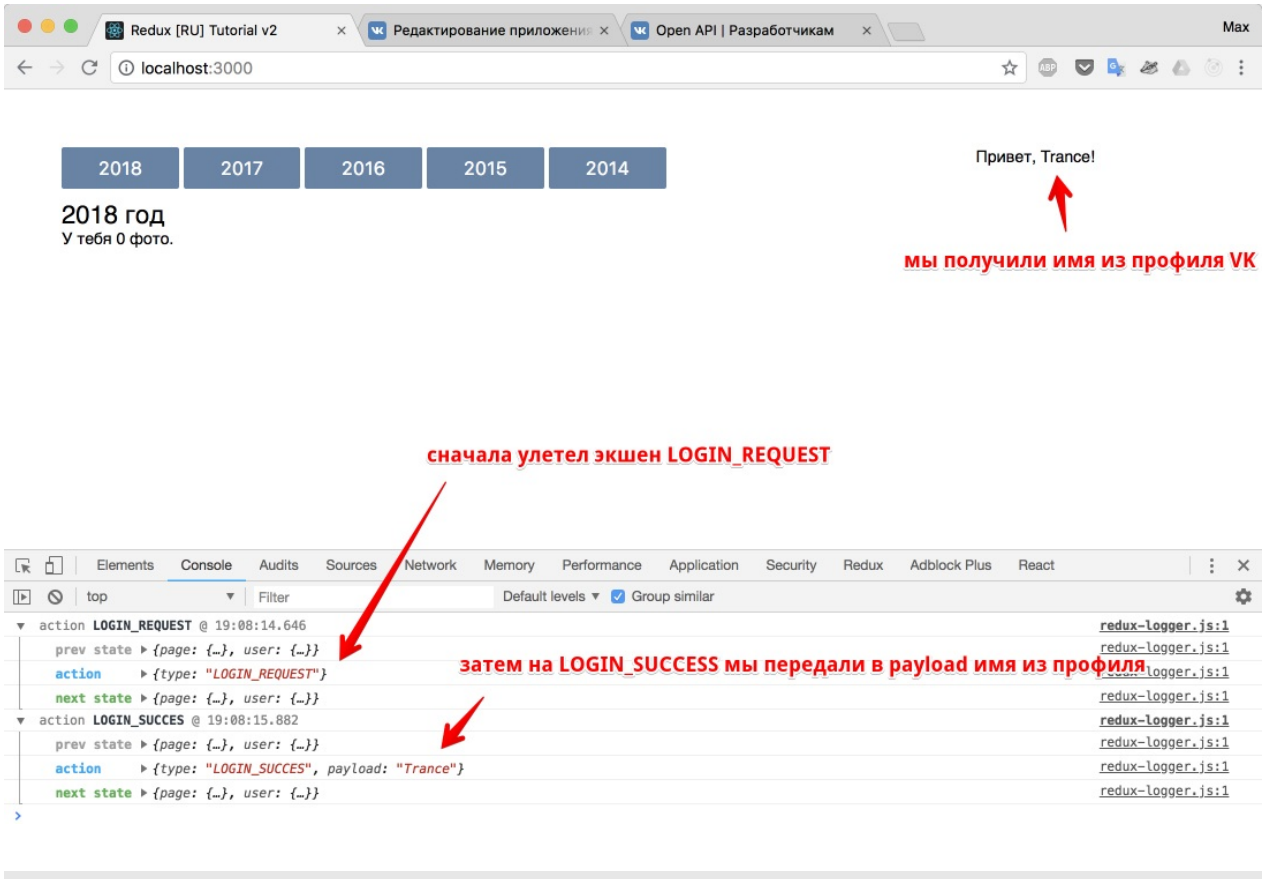
src/components/User.js

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

export class User extends React.Component {
  renderTemplate = () => {
    const { name, error, isFetching } = this.props

    if (error) {
      return <p>Во время запроса произошла ошибка, обновите страницу</p>
    }

    if (isFetching) {
      return <p>Загружаю...</p>
    }

    if (name) {
      return <p>Привет, {name}!</p>
    } else {
      return (
        <button className="btn" onClick={this.props.handleLogin}>
          Войти
        </button>
      )
    }
  }
  render() {
    return <div className="ib user">{this.renderTemplate()}</div>
  }
}

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

В коде компонента `<User />` ничего необычного нет. Рендерим шаблончик (в зависимости от props).

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".

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



Загрузка фото

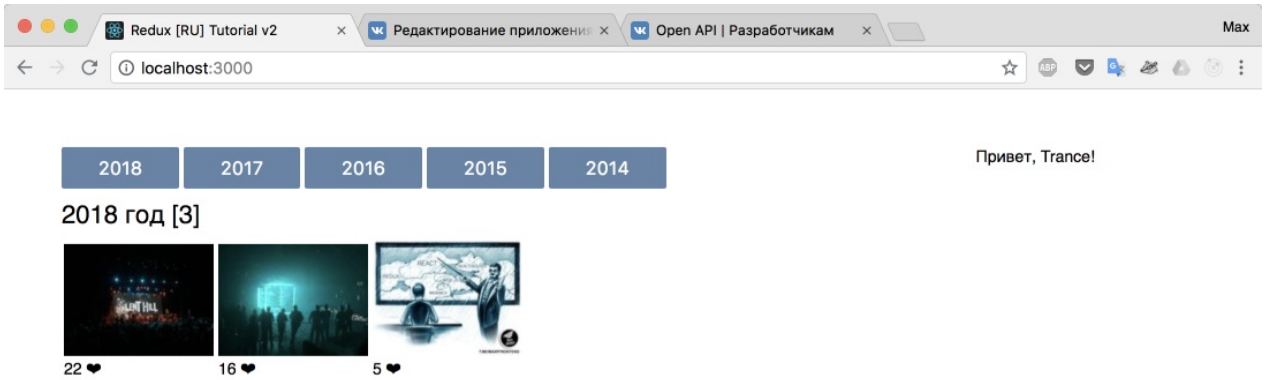
Нам нужно практически повторить, все что написано выше, только для блока **Page**.

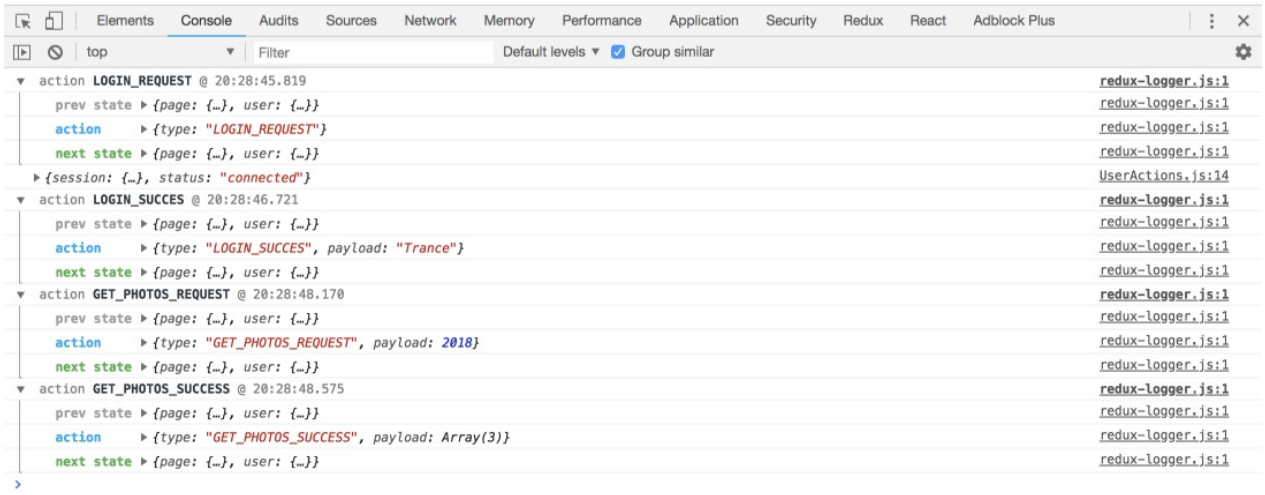
Поэтому, наконец-то появилась самостоятельная задача. Я крайне рекомендую с ней посидеть, так как это практически конец основного материала. Если у вас что-то не получится - вы поймете что нужно закрепить, что перечитать. Не торопитесь смотреть ответ, попробуйте сделать это сами, таким образом вы получите от этого учебника гораздо больше. Да и кайфово это :)

Задача: используя метод photos.getAll вытащите свои фотографии из VK за год, выбранный кнопкой. Отсортируйте их в обратном порядке по лайкам, чтобы самая популярная фото оказалась первой.

После скриншотов есть подсказка: функция, которая делает запрос за фото.

Должно выглядеть следующим образом:





Подсказка: функция для загрузки фото

```
let photosArr = []
let cached = false

function makeYearPhotos(photos, selectedYear) {
  let createdYear,
    yearPhotos = []

  photos.forEach(item => {
    createdYear = new Date(item.date * 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) {
  //eslint-disable-next-line no-undef
  VK.Api.call(
    'photos.getAll',
    { extended: 1, count: count, offset: offset, v: '5.80' },
    r => {
      try {
        photosArr = photosArr.concat(r.response.items)
        if (offset <= r.response.count) {
          offset += 200 // максимальное количество фото которое можно получить за 1 за
прос
          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),
        })
      }
    }
  )
}
```

Так как я не нашел опцию передачи года, то просто выгрузил все фото, по 200 штук за один запрос. Это несколько избыточно, как и тот факт, что мы вызываем функцию `makeYearPhotos`, вместо того чтобы один раз загрузить все фото и "разместить" их по годам. Я оставил код из первого издания учебника, чтобы не усложнять пример.

Решение ниже:

.

.

.

.

Решение:

src/actions/PageActions.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'

let photosArr = []
let cached = false

function makeYearPhotos(photos, selectedYear) {
  let createdYear,
    yearPhotos = []

  photos.forEach(item => {
    createdYear = new Date(item.date * 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) {
  //eslint-disable-next-line no-undef
  VK.Api.call(
    'photos.getAll',
    { extended: 1, count: count, offset: offset, v: '5.80' },
    r => {
      try {
```

```
          photosArr = photosArr.concat(r.response.items)
          if (offset <= r.response.count) {
            offset += 200 // максимальное количество фото которое можно получить за 1 за
прос
            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.

Исправив редьюсер и отрисовку в компоненте, мы закончим начатое.

src/reducers/page.js

```
import {
  GET_PHOTOS_REQUEST,
  GET_PHOTOS_SUCCESS,
  GET_PHOTOS_FAIL,
} from '../actions/PageActions'

const initialState = {
  year: 2018,
  photos: [],
  isFetching: false,
  error: '',
}

export function pageReducer(state = initialState, action) {
  switch (action.type) {
    case GET_PHOTOS_REQUEST:
      return { ...state, year: action.payload, isFetching: true, error: '' }

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

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

    default:
      return state
  }
}
```

src/components/Page.js

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

export class Page extends React.Component {
  onBtnClick = e => {
    const year = +e.currentTarget.innerText
    this.props.getPhotos(year) // setYear -> getPhotos
  }
  renderTemplate = () => {
    const { photos, isFetching, error } = this.props

    if (error) {
      return <p className="error">Во время загрузки фото произошла ошибка</p>
    }

    if (isFetching) {
```

```
        return <p>Загрузка...</p>
      } else {
        return photos.map((entry, index) => ( // [1]
          <div key={index} className="photo">
            <p>
              <img src={entry.sizes[0].url} alt="" />
            </p>
            <p>{entry.likes.count} ❤</p>
          </div>
        ))
      }
    }

    render() {
      const { year, photos } = this.props
      return (
        <div className="ib page">
          <p>
            <button className="btn" onClick={this.onBtnClick}>
              2018
            </button>{' '}
            <button className="btn" onClick={this.onBtnClick}>
              2017
            </button>{' '}
            <button className="btn" onClick={this.onBtnClick}>
              2016
            </button>{' '}
            <button className="btn" onClick={this.onBtnClick}>
              2015
            </button>{' '}
            <button className="btn" onClick={this.onBtnClick}>
              2014
            </button>
          </p>
          <h3>
            {year} год [{photos.length}]
          </h3>
          {this.renderTemplate()}
        </div>
      )
    }
  }

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

[1] - как вы заметили, мы использовали index в качестве ключа для наших div'ов. Запустите пример, попробуйте поменять года. Возможно, вы словите баг, когда у элементов с одинаковым индексом изображение меняется с задержкой. Проблема в том, что мы использовали индекс для элементов, которые изменяются (а индекс-то остается прежним! Ключ в итоге не изменяется, итого реакт "путается").

Чтобы этого избежать, сделайте ключ уникальным (например, для этого у нас есть id в ответе от VK API):

```
if (isFetching) {
  return <p>Загрузка...</p>
} else {
  return photos.map(entry => (
    <div key={entry.id} className="photo">
      <p>
        <img src={entry.sizes[0].url} alt="" />
      </p>
      <p>{entry.likes.count} ❤</p>
    </div>
  ))
}
```

Теперь наш ключ (`key = {entry.id}`) уникальный и бага нет.

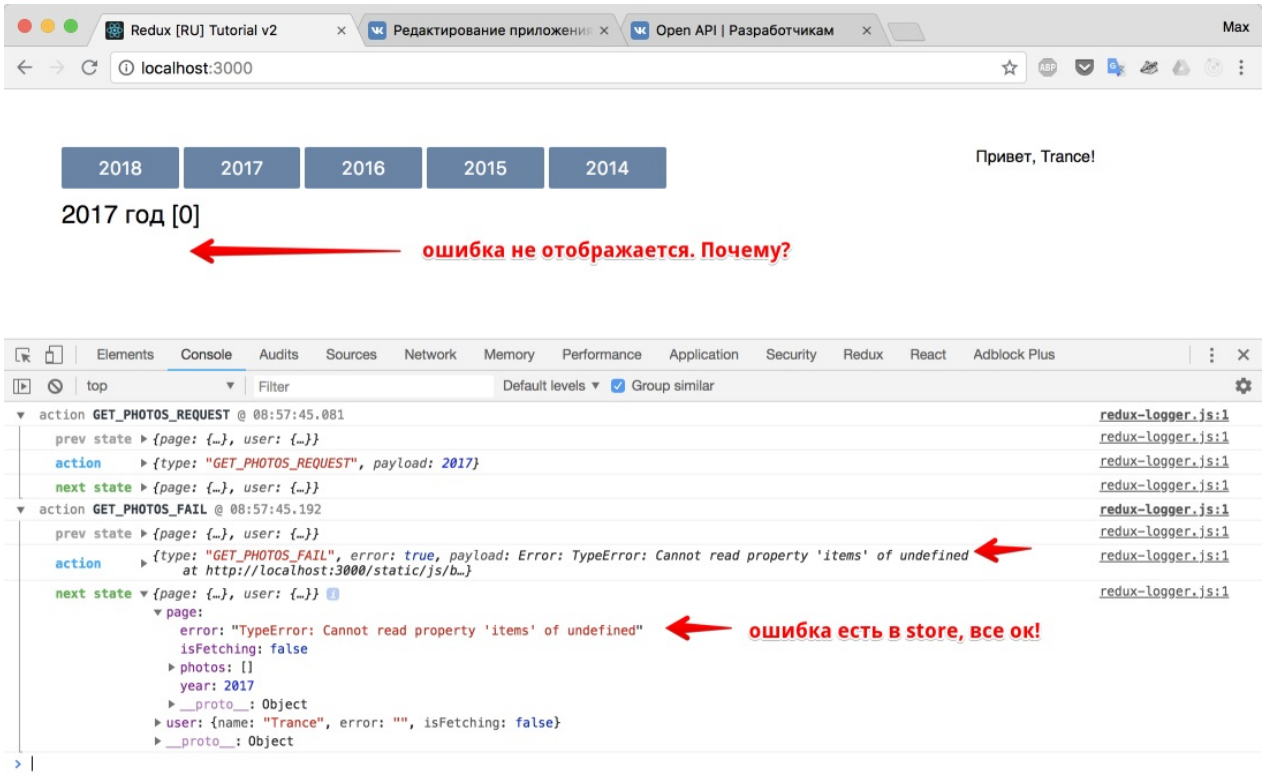
Мини-задачка на внимательность: если сейчас сгенерировать ошибку, то ничего не отобразиться. Как это исправить?

Чтобы проверить ошибку, сделайте в функции запроса фото, поставьте `count: -1` :

src/actions/PageActions.js

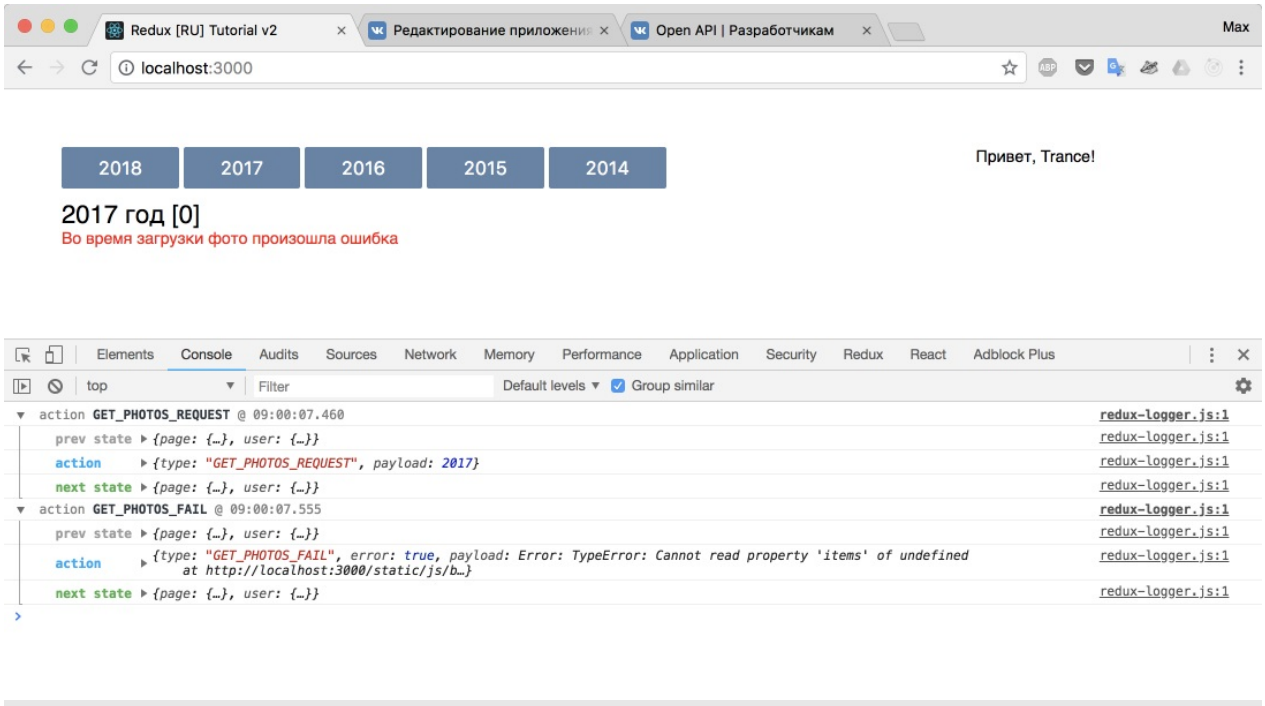
```
...
function getMorePhotos(offset, count, year, dispatch) {
  //eslint-disable-next-line no-undef
  VK.Api.call(
    'photos.getAll',
    { extended: 1, count: -1, offset: offset, v: '5.80' },
    r => {
...
```

Проблема:



Решение:

```
...

class App extends Component {
  render() {
    const { user, page, getPhotosAction, handleLoginAction } = this.props
    return (
      <div className="app">
        {/* добавили error prop для Page */}
        <Page
          photos={page.photos}
          year={page.year}
          isFetching={page.isFetching}
          error={page.error}
          getPhotos={getPhotosAction}
        />
        ...
}

...
```



Итого: закрепили работу с асинхронными запросами.

Исходный код на текущий момент.

P.S. css тоже был слегка подправлен.

Оптимизация. Рефакторинг

В нашем решении есть слабые места:

- некоторые названия переменных избыточны (чтобы было понятно, добавлено Actions у экшенов, которые мы приклеиваем);
- повторяющийся однотипный код (5 кнопок с номером года в `<Page />`);
    - в action улетает текст с кнопки, если текст изменится - код сломается. Проблема: большая связанность. Нужно облегчить.
- возможно существует более простой путь "достать" из вк фото за конкретный год (не рассматриваю это как проблему);
- фраза "Привет, ИМЯ" после обновления страницы заменяется кнопкой "войти", то есть не отображает реальной картины (фотографии у нас при этом доступны для загрузки, то есть мы уже авторизованы);
- после авторизации (или после перезагрузки) было бы неплохо сразу загружать фото для 2018 года, так как юзер видит пустой экран и заголовок 2018;

Можно отнести это к "доработкам". Однако у нас есть место, которое является опасным и о котором я лишь вскользь говорил в учебнике, пора исправится.

Приглашаю вас "убить" главную проблему текущего приложения - **лишние перерисовки компонента** в следующем подразделе.

Остальные проблемы и будущие доработки живут в одноименном разделе.

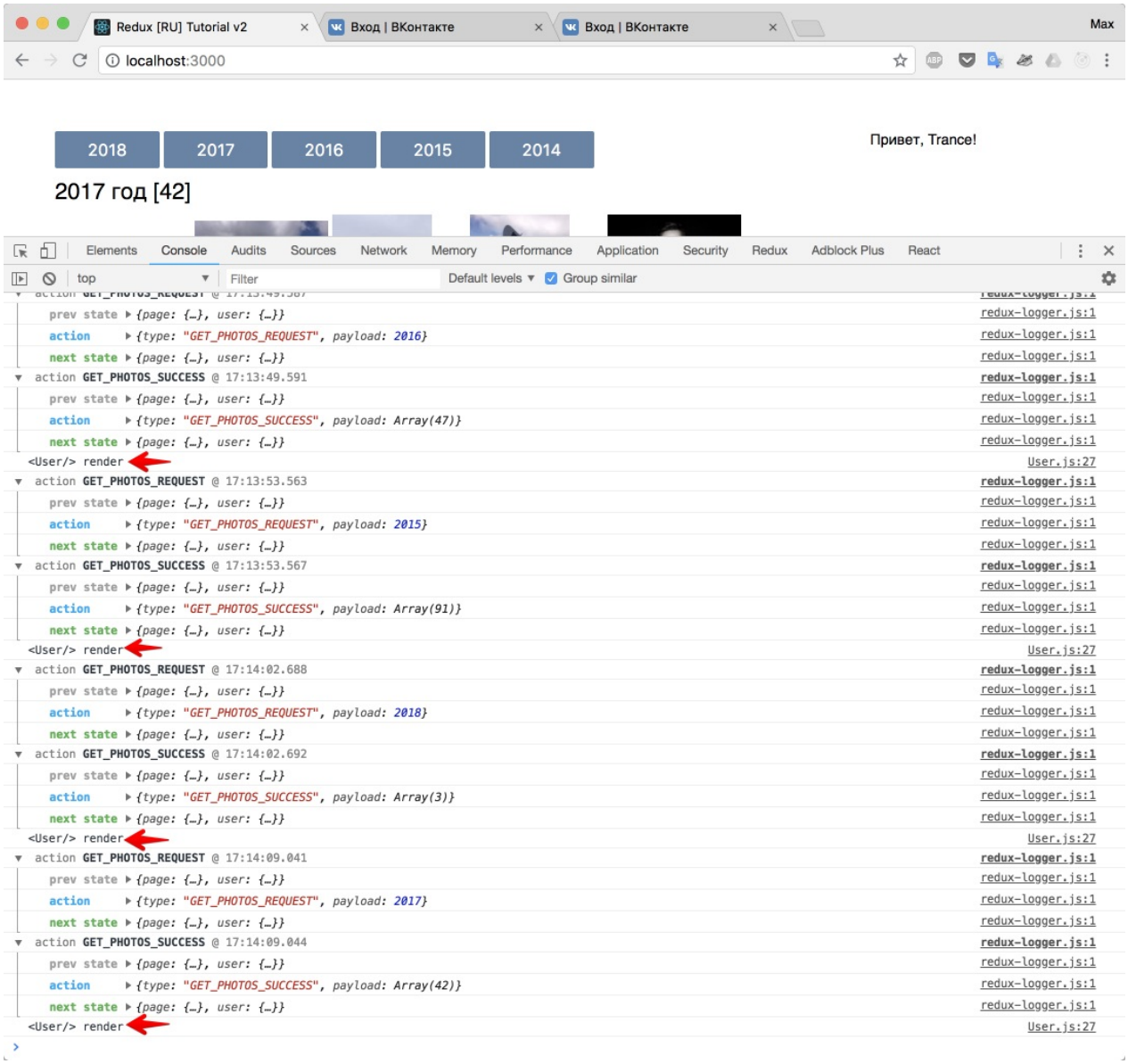
Главная проблема приложения

У нас есть 2 компонента `<User />` и `<Page />`. Мы специально сделали для них два редьюсера, чтобы обновлять их независимо! А у нас? А у нас `<User />` каждый раз обновляется при обновлении `<Page />` и наоборот.

Добавьте console.log в render метод у `<User />`:

src/components/User.js

```
...
render() {
  console.log('<User/> render')
  return <div className="ib user">{this.renderTemplate()}</div>
}
...
```



Перерисовка компонента User происходит постоянно. Это не влияет на производительность нашего мини-приложения, однако, мы не готовы с этим мириться.

Представьте дашборд (панель) с большим количеством виджетов, информация в которых обновляется по событиям от бэкэнда. Если каждый виджет будет перерисовывать полностью весь дашборд, то это будет крайне некрасиво (как для юзера, так и для производительности).

Чтобы такого не было, мы должны каждую отдельную сущность приложения класть в отдельный контейнер.

Будем исправлять, для этого:

- `<App />` становится тупым компонентом, который рендерит 2 контейнера:
  - `<PageContainer />`
  - `<UserContainer />`

Данные контейнеры - просто обертки над нашими компонентами, в которых мы "подключаемся (*connect*) к Redux".

Так же, я сразу заменю название у экшенов внутри mapDispatchToProps: уберу оттуда частичку *Action*.

В остальном, мы просто "разносим" то, что было в `<App />` по раздельным контейнерам.

src/index.js

```
import React from 'react'
import ReactDOM from 'react-dom'
import { Provider } from 'react-redux'
import { store } from './store/configureStore'
import App from './components/App' // изменили путь

import registerServiceWorker from './registerServiceWorker'

import './index.css'

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

src/components/App.js

```
import React, { Component } from 'react'
import UserContainer from '../containers/UserContainer' // изменили импорт
import PageContainer from '../containers/PageContainer' // изменили импорт

class App extends Component {
  render() {
    return (
      <div className="app">
        <PageContainer />
        <UserContainer />
      </div>
    )
  }
}

export default App
```

src/containers/PageContainer.js

```
import React from 'react'
import { connect } from 'react-redux'
import { Page } from '../components/Page'
import { getPhotos } from '../actions/PageActions'

class PageContainer extends React.Component {
  render() {
    const { page, getPhotos } = this.props
    return (
      <Page
        photos={page.photos}
        year={page.year}
        isFetching={page.isFetching}
        error={page.error}
        getPhotos={getPhotos}
      />
    )
  }
}

const mapStateToProps = store => {
  return {
    page: store.page,
  }
}

const mapDispatchToProps = dispatch => {
  return {
    getPhotos: year => dispatch(getPhotos(year)),
  }
}

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

Как вы могли заметить, все что касалось `<Page />` хранится в отдельном контейнере: подписка на часть стора, экшен, пропсы...

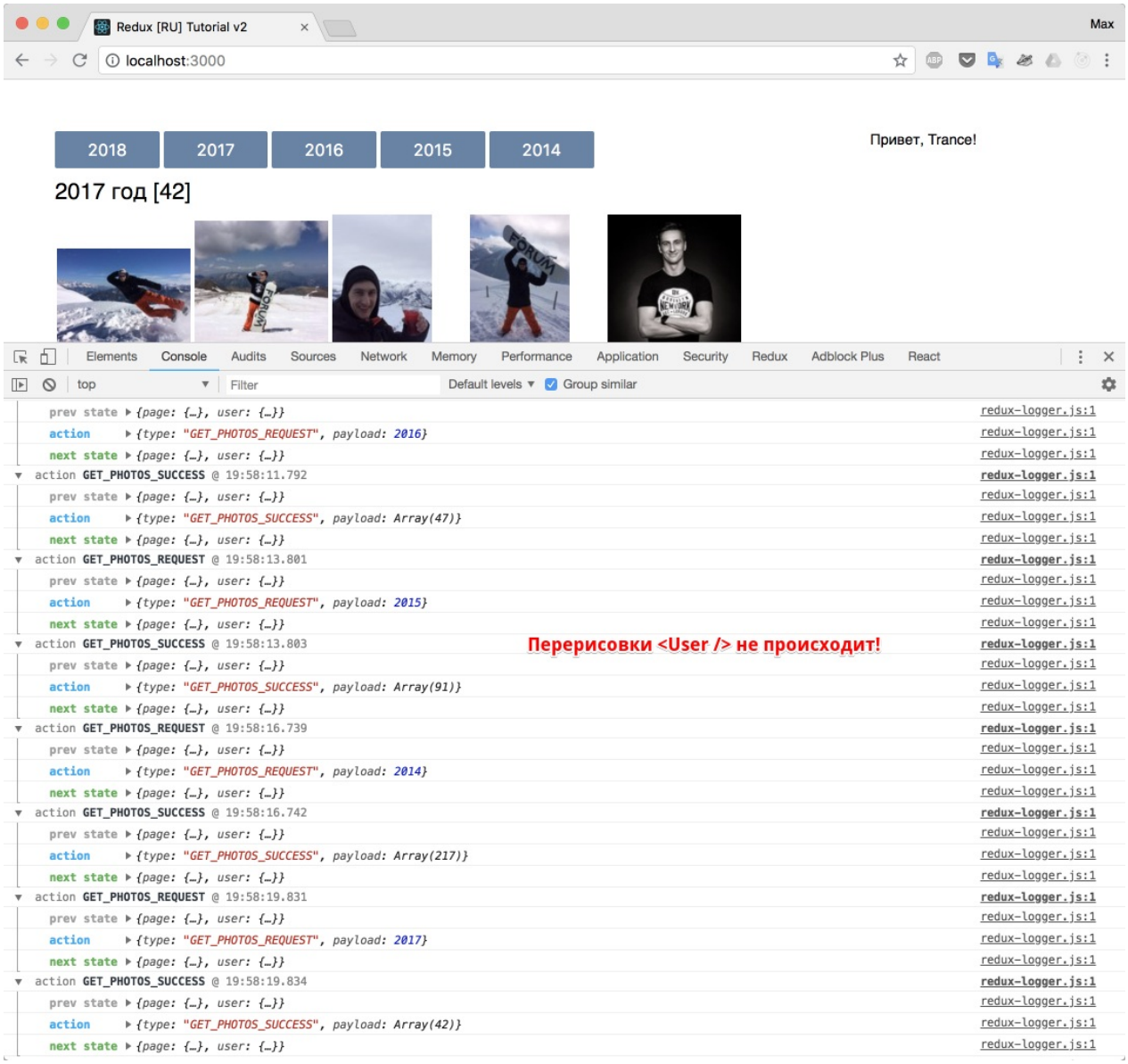
То же самое, делаем для `<UserContainer />`

src/containers/UserContainer.js

```
import React from 'react'
import { connect } from 'react-redux'
import { User } from '../components/User'
import { handleLogin } from '../actions/UserActions'

class UserContainer extends React.Component {
  render() {
    const { user, handleLogin } = this.props
    return (
      <User
        name={user.name}
        error={user.error}
        isFetching={user.isFetching}
        handleLogin={handleLogin}
      />
    )
  }
}

const mapStateToProps = store => {
  return {
    user: store.user,
  }
}

const mapDispatchToProps = dispatch => {
  return {
    handleLogin: () => dispatch(handleLogin()),
  }
}

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

Теперь внимание: в компоненте *App* есть два "независимых компонента". Сейчас при изменении данных в редьюсере для *Page* - *User* перерисовываться не будет. *App* тоже, само собой. *App* у нас вообще не будет перерисовываться более при таком раскладе.

Снова покликаем по кнопкам (*console.log* в `<User />` остался):



Еще раз заострю внимание: мы не просто сделали хорошо, мы сделали супер-хорошо! Render - обычно самая дорогая операция. Вызывать "перерисовку" каждого "кусочка" приложения нужно осознанно. Всегда проверяйте (например, так же банально с помощью `console.log`) сколько раз у вас что рендерится, и нет ли чего лишнего.

Давайте заодно здесь быстренько исправим отрисовку кнопок в `<Page />`

src/components/Page.js

```
...

export class Page extends React.Component {
  onBtnClick = e => {
    ...
  }
  renderButtons = () => {
    const years = [2018, 2017, 2016, 2015, 2014]

    return years.map((item, index) => { // [1]
      return (
        <button key={index} className="btn" onClick={this.onBtnClick}>
          {item}
        </button>
      )
    })
  }
  renderTemplate = () => {
    ...
  }

  render() {
    const { year, photos } = this.props
    return (
      <div className="ib page">
        <p>{this.renderButtons()}</p>
        <h3>
          {year} год [{photos.length}]
        </h3>
        {this.renderTemplate()}
      </div>
    )
  }
}
...
```

(*Добавьте по вкусу щепотку* `margin` *для* `.btn`)

[1] Использовать в данной ситуации *index* для *key* плохо?. В данном случае - не плохо. Напоминаю, что индекс в качестве ключа плохо использовать, когда у вас элементы меняются местами. Справедливости ради, здесь в качестве индекса можно было бы использовать "год", так как главное в индексе - это **уникальность**.

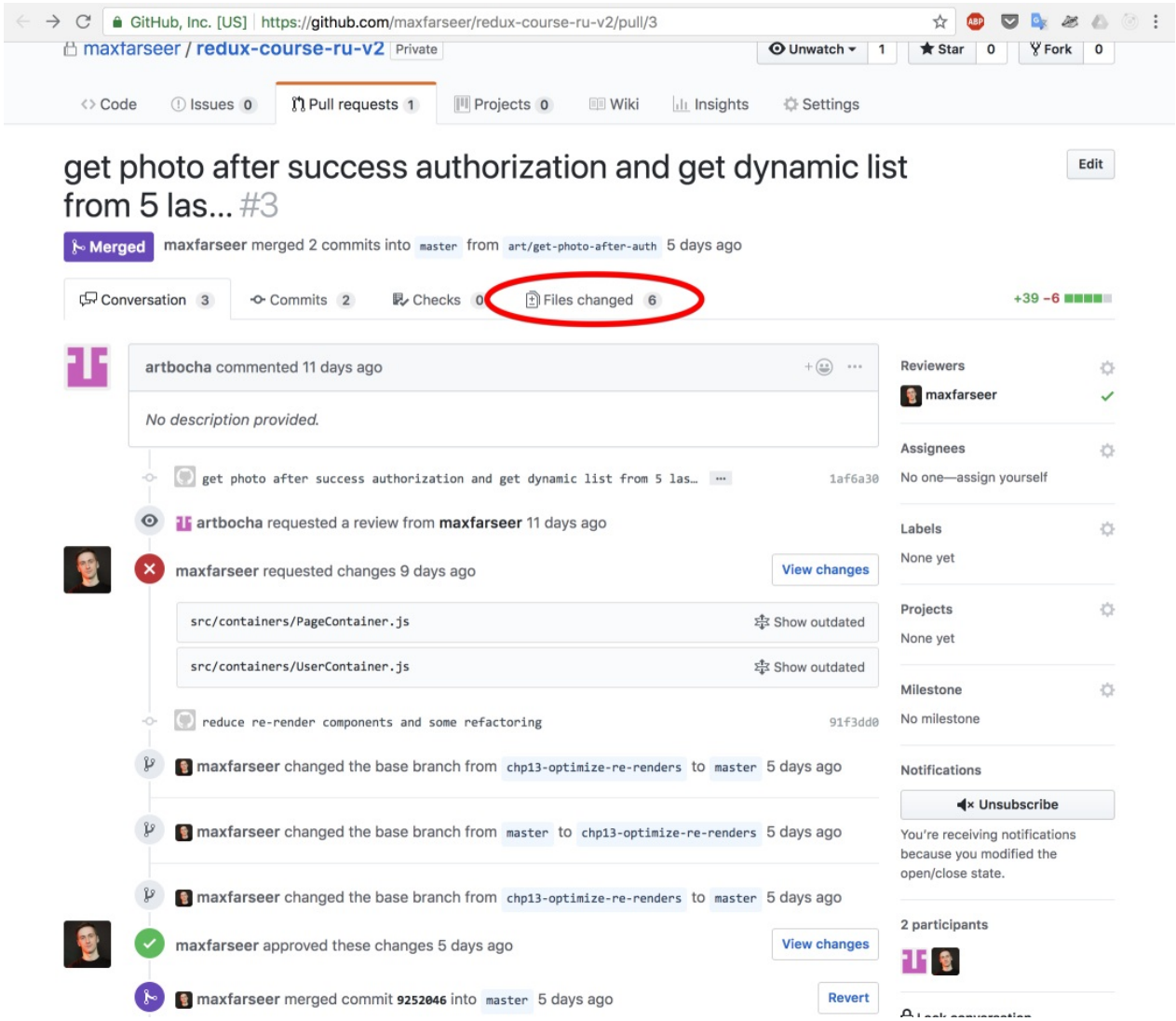
Итого: мы научились бережно относится к перерисовкам, а так же закрепили на практике вопрос зачем разбивать редьюсер на маленькие редьюсеры.

Исходный код.

Доработки

Доработки кладутся в master ветку. К некоторым будут комментарии.

Чтобы легко разобраться в коде, который изменился - открывайте PR #X ссылки, где в каждом pull-request'e видно список измененных файлов (таб - files).



Так же, после того как PR был принят, и если были найдены какие-то проблемы после, то они исправляются только в мастер ветке, поэтому не забывайте заглядывать туда.

- загрузка фото после успешной авторизации (PR #3)

Сделано достаточно просто с помощью callback-функции.

- модальное окно с большим фото в нем (PR #5)

Добавлен пакет react-modal, немного стилей и переделана логика отображения фото.

Появился компонент `<PhotoManager />`, который содержит все фото (компонент `<ListPhoto />`) + модальное окно (одно!). Из `<ListPhoto />` по клику передается url-адрес для большого изображения в компонент `<BigPhoto />`, в котором для прелоадера использован трюк с подгрузкой изображения в `img.onload`.

Больше подробностей на вебинаре (время старта XX) [1]

[1] будет добавлена ссылка, когда выложу запись на YT-канал

Redux-saga версия

Версия с сагой расположена в отдельной ветке.

Есть парочка `TODO:` в коде, можете присылать PR.

Продолжение следует...

В качестве заключения

Я надеюсь вы выполняли код по ходу книги? Если нет, то настало время взять и сделать финальный пример лично.

Прикладываю план по закреплению знаний и прокачке:

0) Ознакомиться с React-router'ом по официальной документации [EN].

0a) По роутингу есть статья на сайте (текст и видео)

1) Тестовое задание #1

2) Тестовое задание #2

Если у вас заблокирован VK, то задания можно найти в github-репозитории. Можете подписаться на репозиторий, чтобы не пропустить следующие.

На уже прошедшие тестовые задания, можно получить code-review платно, стоимость 15$, присылайте запрос на почту: **maxpfrontend@gmail.com** с темой "*Code review тестовое задание X*", где X - номер.

3) Тестирование логики (reducers и actions), включая мок асинхронного запроса.

4) Тестирование компонентов Jest + enzyme

5) Добавьте тесты для второго тестового задания, прокачайте его на свое усмторение

Итого: после выполнения данных шагов самостоятельно, я ответственно заявляю, что вы готовы идти джуном в офис. Осталось прокачать свои софт-скилы (скилы переговоров).

Также, есть видео с вопросами на собеседовании.

Если вам понравилось, вы можете поддержать проект или оставить отзыв.

Полезные ссылки

Мои уроки/вебинары/соц.сети:

- Полноценный учебник "Основы React"
- Расписание стримов и вебинаров (на сайте есть текстовые версии вебинаров)
- Youtube канал с записями вебинаров и стримов
- Группа vkontakte
- Канал в telegram
- Twitter
- Facebook

Спасибо

При создании данного учебника и в процессе его жизни, мне помогали:

- Артем <artbocha> Бочков;