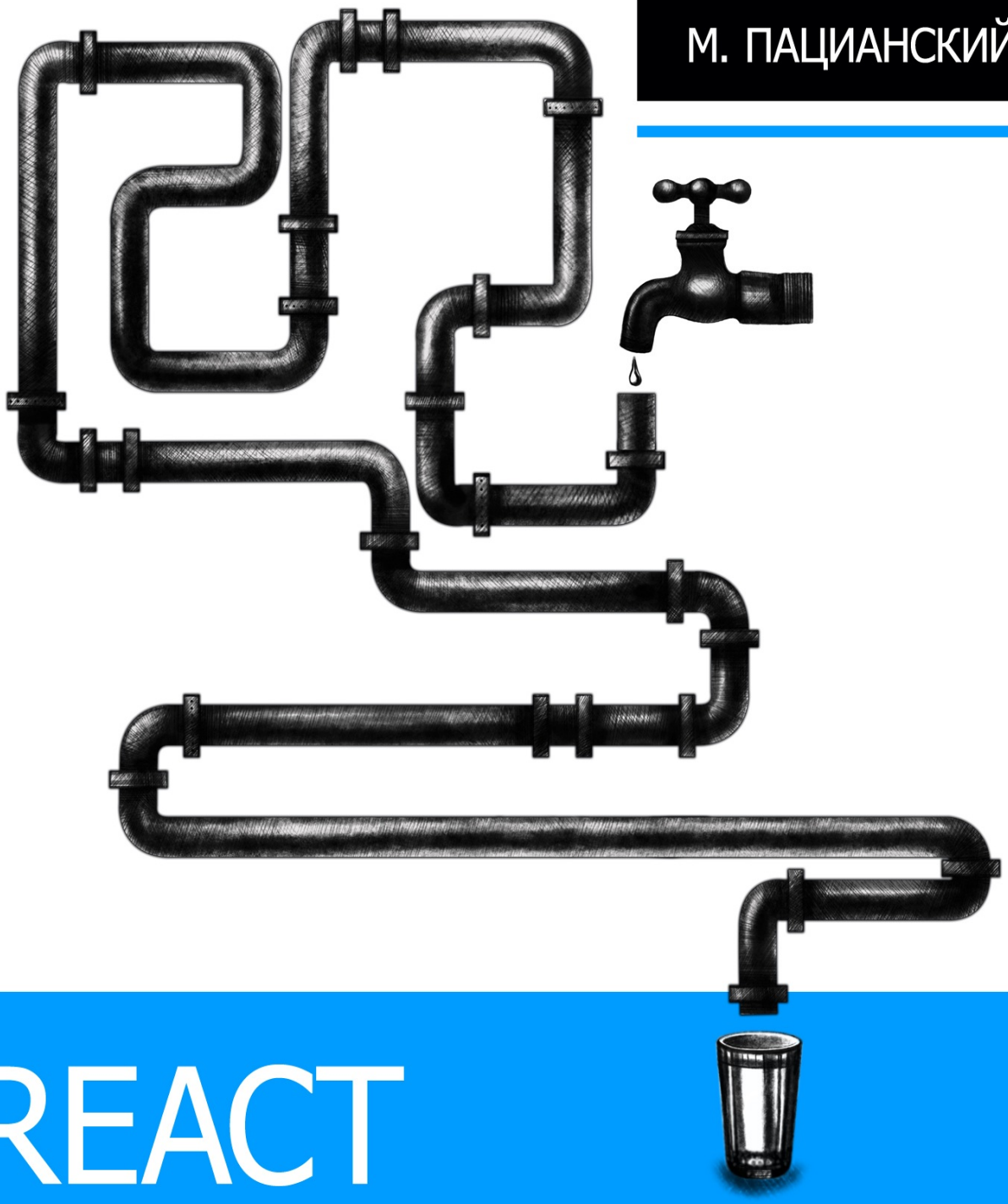
М. ПАЦИАНСКИЙ



REACT +REDUX +REACT-ROUTER

2016

без воды

Table of Contents

Роутинг в react-приложениях

Курс включает в себя 2 части:

1. Подключение и использование react-router'a
2. Использование react-router'a + redux

Преимущества и недостатки

Преимущества данного учебника:

- на русском, текстовый формат + картинки (нет видео)
- практически все разделы содержат исходный код на github
- частично включает в себя переводы официальных туториалов/документации
- использует современные (*апрель 2016*) версии библиотек
- не использует библиотеки для роутинга, кроме *react-router*
- кратко, без воды, полностью раскрывает вопрос роутинга на клиенте

Недостатки:

- некоторые примеры невзрачны
- нет красивого оформления (*html/css*)
- не рассмотрен server-side рендеринг
- нет тестов

От автора

Вопрос **react-router + redux** не раскрыт даже в официальной документации *redux*, поэтому, так как библиотеки развиваются стремительно, я постарался "научить вас рыбачить", а не просто "дать рыбу".

Первая часть [*react-router*] - вряд ли изменится, если **react-router** резко не изменит свое поведение.

Вторая часть [*react-router* + *redux*] - может измениться, либо дополниться примерами.

Вопросы можно задавать в twitter, или на maxfarseer@gmail.com с темой "React-router туториал".

Консультации и платные услуги

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

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

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

Интересные ссылки:

Репозиторий с кодом для уроков

Official docs

Unofficial React Router docs

React router tutorial

Authenticated component (old)

A simple way to route with redux

Заметки и допущения

В курсе переодически встречаются места, когда переменная создается для *наглядности*. Не забывайте, что если мы пытаемся получить доступ к значению один раз - создание переменной неоправданно, а если 2 и более - неплохо бы сделать переменную.

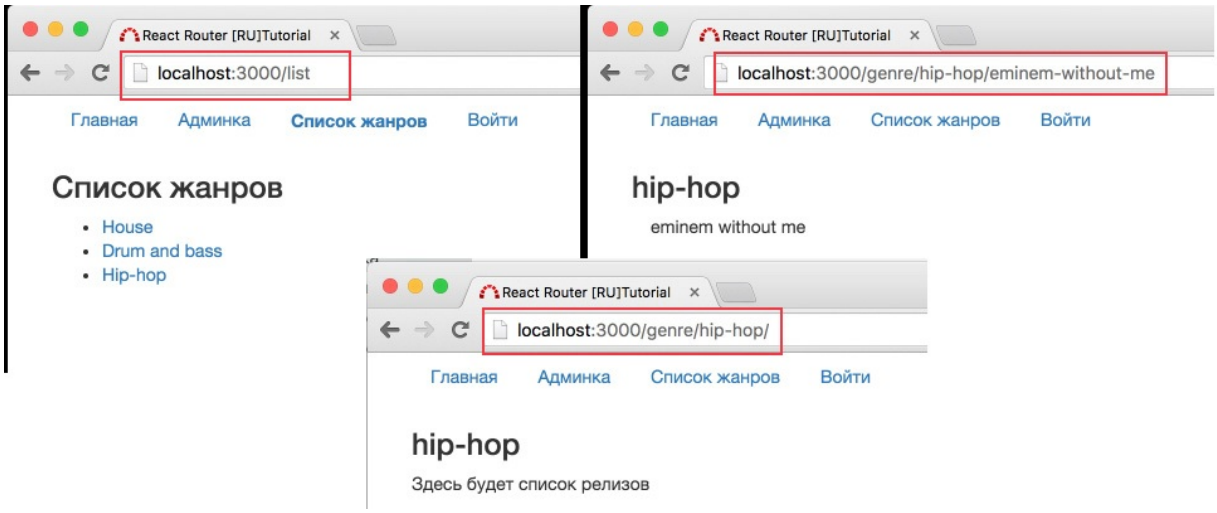
Подготовка

Какие обычно стоят задачи из области "роутинга" ?

1. Доступ к разным "страницам/разделам" - а если быть точным, к разным состояниям приложения, так как мы разрабатываем *SPA* (*single page application* - одностраничное приложение). Для простоты продолжим употреблять слова страница/раздел и подобные.
2. Разделение прав доступа: гости не могут зайти на страницу "/admin", а администратор может.
3. Редиректы.
4. 404 страница

Предлагаю сделать очень примитивный музыкальный каталог:

- есть список всех жанров
- есть страница релиза (в нашем случае - "трэка/песни")
- есть страница релизов данного жанра



Структура сайта

```
/ - главная страница
/list - список жанров
/genre/:genre/ - список релизов данного жанра
/genre/:genre/:release - информация о релизе
/admin - страница администратора
```

Если в URL-адресе есть *параметр* (двоеточие + слово) значит такой адрес является **динамическим**. То есть, может быть таким: `/house/release/avicii-the-nights`, где house - музыкальный жанр, а avicii-the-nights - название релиза.

P.S. в реальном мире, релиз может содержать больше одного трэка. Пример настоящего релиза - здесь.

Начальная структура директорий и некоторых файлов

```
+-- bin
+-- src
|   +-- components
|   +-- containers
|   +-- index.js
+-- webpack
+-- index.html
+-- package.json
+-- server.js
```

Настраиваем dev-окружение

Если вы не хотите заниматься настройкой окружения, а хотите сразу перейти к урокам по react-router'y, возьмите этот исходный код.

Что мы получим в итоге?

- React + перезагрузка страницы при возникновении изменений в коде компонентов (стилей в том числе)
- Bootstrap(и jQuery в качестве примера добавления библиотеки. В учебнике *jQuery* **не используется**)
- SASS
- Возможность писать ES2015/ES7 код
- Линтинг (проверку) нашего кода с помощью ESLint
- Наброски для *development* и *production* версий.

Текст ниже не является обязательным для прочтения, достаточно скачать архив и выполнить `npm install`. В будущем мы установим react-router, поэтому вы точно ничего не "потеряете".

За основу был взят проект redux-easy-boilerplate.

Изменения/Обновления

В процессе написания учебника "случился" отпуск. Да и новые версии выходят постоянно. Я старался обновляться по мере возможностей или в силу необходимых обстоятельств. Ниже будет список.

1. webpack в хроме выдавал слишком много "предупреждений" (из-за source maps). Обновился до `webpack@1.13.0`
2. обновил react и react-dom

```
"react": "^15.0.1",
"react-dom": "^15.0.1",
```

Подробнее о настройке

Все как обычно начинается с *package.json*

```
{
  "name": "react-router-ru-tutorial",
  "version": "1.0.0",
  "description": "React-router RU tutorial",
  "main": "index.js",
  "scripts": {
    "start": "node bin/server.js"
  },
  "author": "Maxim Patsianskiy",
  "license": "MIT",
  "devDependencies": {
    "autoprefixer": "^6.3.3",
    "babel-core": "^6.7.2",
    "babel-eslint": "^5.0.0",
    "babel-loader": "^6.2.4",
    "babel-polyfill": "^6.7.2",
    "babel-preset-es2015": "^6.6.0",
    "babel-preset-react": "^6.5.0",
    "babel-preset-react-hmre": "^1.1.1",
    "babel-preset-stage-0": "^6.5.0",
    "bootstrap-loader": "^1.0.9",
    "bootstrap-sass": "^3.3.6",
    "css-loader": "^0.23.1",
    "eslint": "^2.4.0",
    "eslint-loader": "^1.3.0",
    "eslint-plugin-react": "^4.2.3",
    "estraverse-fb": "^1.3.1",
    "extract-text-webpack-plugin": "^1.0.1",
    "file-loader": "^0.8.5",
    "imports-loader": "^0.6.5",
    "node-sass": "^3.4.2",
    "postcss-import": "^8.0.2",
    "postcss-loader": "^0.8.2",
    "resolve-url-loader": "^1.4.3",
    "sass-loader": "^3.2.0",
    "style-loader": "^0.13.0",
    "url-loader": "^0.5.7",
    "webpack": "^1.12.14",
    "webpack-dev-middleware": "^1.5.1",
    "webpack-dev-server": "^1.14.1",
    "webpack-hot-middleware": "^2.10.0",
    "webpack-merge": "^0.8.3"
  },
  "dependencies": {
    "jquery": "^2.2.1",
    "react": "^0.14.7",
    "react-dom": "^0.14.7"
  }
}
```

Обратите внимание, в списке зависимостей нет ни react-router, ни даже redux, зато есть jQuery и Bootstrap. Собственно, jQuery - это зависимость Bootstrap. Реакт-роутер и редукс будут добавлены позже, как и другие необходимые пакеты.

Начнем распутывать "клубок".

Как видно из *package.json* приложение будем "стартовать" командой `npm start`, которая в свою очередь выполнит `node bin/server.js`

bin/server.js

```
var fs = require('fs');

var babelrc = fs.readFileSync('./.babelrc');
var config;

try {
  config = JSON.parse(babelrc);
} catch (err) {
  console.error('==>     ERROR: Error parsing your .babelrc.');
  console.error(err);
}

require('babel-core/register')(config);
require('../server');
```

Благодаря такой "обертке" мы убьем сразу несколько зайцев:

- сможем выдавать красивое сообщение об ошибке, если не хватает какого-то из babel-preset'ов
- сможем код файла по адресу `../server` относительно текущего (напомню, текущий - это `bin/server.js`) писать на ES2015

Давайте создадим "этот самый файл" по адресу `../server`.

server.js

```
const http = require('http');
const express = require('express');
const app = express();

(function initWebpack() {
  const webpack = require('webpack');
  const webpackConfig = require('./webpack/common.config');
  const compiler = webpack(webpackConfig);

  app.use(require('webpack-dev-middleware')(compiler, {
    noInfo: true, publicPath: webpackConfig.output.publicPath,
  }));

  app.use(require('webpack-hot-middleware')(compiler, {
    log: console.log, path: '/__webpack_hmr', heartbeat: 10 * 1000,
  }));

  app.use(express.static(__dirname + '/'));
})();

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

const server = http.createServer(app);
server.listen(process.env.PORT || 3000, function onListen() {
  const address = server.address();
  console.log('Listening on: %j', address);
  console.log(' -> that probably means: http://localhost:%d', address.port);
});
```

Код выше - это вполне стандартный конфиг сервера на Express.

Отмечу, что здесь мы подключаем конфиг webpack посредством:

```
const webpackConfig = require('./webpack/common.config');
```

что позволит нам удобно разделить конфиг на *production* и *development* версии.

webpack/common.config

```
const path = require('path');
const autoprefixer = require('autoprefixer');
const postcssImport = require('postcss-import');
const merge = require('webpack-merge');

const development = require('./dev.config.js');
const production = require('./prod.config.js');
```

```
require('babel-polyfill').default;

const TARGET = process.env.npm_lifecycle_event;

const PATHS = {
  app: path.join(__dirname, '../src'),
  build: path.join(__dirname, '../dist'),
};

process.env.BABEL_ENV = TARGET;

const common = {
  entry: [
    PATHS.app,
  ],

  output: {
    path: PATHS.build,
    filename: 'bundle.js',
  },

  resolve: {
    extensions: ['', '.jsx', '.js', '.json', '.scss'],
    modulesDirectories: ['node_modules', PATHS.app],
  },

  module: {
    preLoaders: [
      {
        test: /\.js$/,
        loaders: ['eslint'],
        include: [
          path.resolve(__dirname, '../src'),
        ],
      }
    ],
    loaders: [{
      test: /bootstrap-sass\/assets\/javascripts\//,
      loader: 'imports?jQuery=jquery',
    }, {
      test: /\.woff(\?v=\d+\.\d+\.\d+)?$/,
      loader: 'url?limit=10000&mimetype=application/font-woff',
    }, {
      test: /\.woff2(\?v=\d+\.\d+\.\d+)?$/,
      loader: 'url?limit=10000&mimetype=application/font-woff2',
    }, {
      test: /\.ttf(\?v=\d+\.\d+\.\d+)?$/,
      loader: 'url?limit=10000&mimetype=application/octet-stream',
    }, {
      test: /\.otf(\?v=\d+\.\d+\.\d+)?$/,
      loader: 'url?limit=10000&mimetype=application/font-otf',
    }, {
      test: /\.eot(\?v=\d+\.\d+\.\d+)?$/,
```

```
        loader: 'file',
      }, {
        test: /\.svg(\?v=\d+\.\d+\.\d+)?$/,
        loader: 'url?limit=10000&mimetype=image/svg+xml',
      }, {
        test: /\.js$/,
        loaders: ['babel-loader'],
        exclude: /node_modules/,
      }, {
        test: /\.png$/,
        loader: 'file?name=[name].[ext]',
      }, {
        test: /\.jpg$/,
        loader: 'file?name=[name].[ext]',
      }],
    },

    postcss: (webpack) => {
      return [
        autoprefixer({
          browsers: ['last 2 versions'],
        }),
        postcssImport({
          addDependencyTo: webpack,
        }),
      ];
    },
  };

  if (TARGET === 'start' || !TARGET) {
    module.exports = merge(development, common);
  }

  if (TARGET === 'build' || !TARGET) {
    module.exports = merge(production, common);
  }
```

В данном конфиге меня заинтересовали последние строчки. Посмотрите как происходят присваивания в переменную `TARGET`.

Если мы запускаем `npm start` - наш финальный конфиг будет равен текущему конфигу (*webpack/common.config* + *webpack/dev.config*)*.

* под "+" подразумевается "мерж" (слияние) с помощью webpack-merge*

Если же мы запустим `npm build` (у нас нет этого скрипта сейчас), то запустится сборка, которая возьмет текущий конфиг и "смержит" его с *production* конфигом.

Ниже приведен код *dev* и *prod* конфигов. Если интересно - посмотрите различия.

Версия для "разработки".

webpack/dev.config.js

```
const webpack = require('webpack');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  devtool: 'cheap-module-eval-source-map',
  entry: [
    'bootstrap-loader',
    'webpack-hot-middleware/client',
    './src/index',
  ],
  output: {
    publicPath: '/dist/',
  },

  module: {
    loaders: [{
      test: /\.scss$/,
      loader: 'style!css?localIdentName=[path][name]--[local]!postcss-loader!sass',
    }],
  },

  plugins: [
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: '"development"',
      },
      __DEVELOPMENT__: true,
    }),
    new ExtractTextPlugin('bundle.css'),
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.HotModuleReplacementPlugin(),
    new webpack.NoErrorsPlugin(),
    new webpack.ProvidePlugin({
      jQuery: 'jquery',
    }),
  ],
};
```

Версия для "продакшен".

webpack/prod.config.js

```
const webpack = require('webpack');
const ExtractTextPlugin = require('extract-text-webpack-plugin');

module.exports = {
  devtool: 'source-map',

  entry: ['bootstrap-loader/extractStyles'],

  output: {
    publicPath: 'dist/',
  },

  module: {
    loaders: [{
      test: /\.scss$/,
      loader: 'style!css!postcss-loader!sass',
    }],
  },

  plugins: [
    new webpack.DefinePlugin({
      'process.env': {
        NODE_ENV: '"production"',
      },
      __DEVELOPMENT__: false,
    }),
    new ExtractTextPlugin('bundle.css'),
    new webpack.optimize.DedupePlugin(),
    new webpack.optimize.OccurenceOrderPlugin(),
    new webpack.optimize.UglifyJsPlugin({
      compress: {
        warnings: false,
      },
    }),
  ],
};
```

Зачем это в данном туториале? Я посчитал, что раз вы решили прочесть "ручную" настройку - вам будет интересно посмотреть на вариант сборки с использованием dev/prod версий. Если я прав - посмотрите как устроена секция *scripts* здесь.

Ок, немного разобрались. Давайте установим зависимости: `npm install`

Создадим конфигурационные файлы для babel и ESLint:

.babelrc

```
{
  "presets": ["react", "es2015" , "stage-0"],
  "plugins": [],
  "env": {
    "start": {
      "presets": ["react-hmre"]
    }
  }
}
```

.eslintrc

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

Не забывайте про правило jsx-quotes. Если вы предпочитаете использовать двойные кавычки, то оно должно выглядеть так:

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

Список всех правил можно посмотреть на официальном сайте линтера.

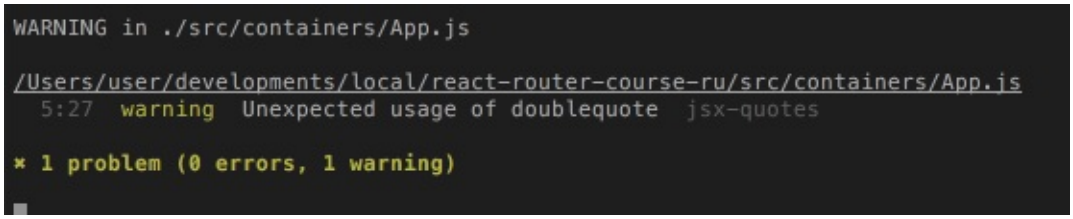
Напомню, обозначения цифр:

- 0 - правило отключено

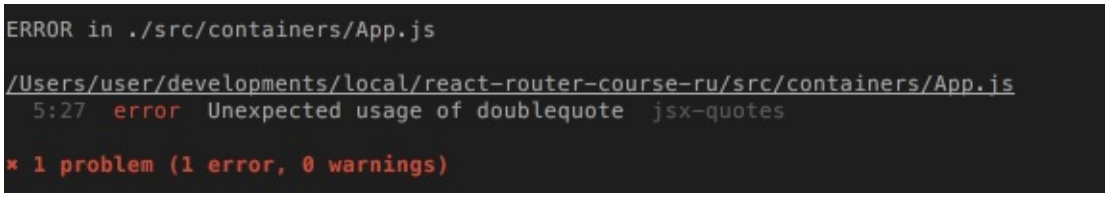
- 1 - правило включено, если нарушение выявлено - выведет warning
- 2 - правило включено, если нарушение выявлено - выведет error

Примеры работы ESLint:

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




```
jsx-quotes: [2, "prefer-single"]
```



Создадим *index.html*, точку входа для скриптов (*index.js*), а так же компонент `<App />`.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>React Router [RU]Tutorial</title>
  </head>
  <body>
    <div id="root">
    </div>

    <script src="dist/bundle.js"></script>
  </body>
</html>
```

src/index.js

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


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

src/containers/App.js

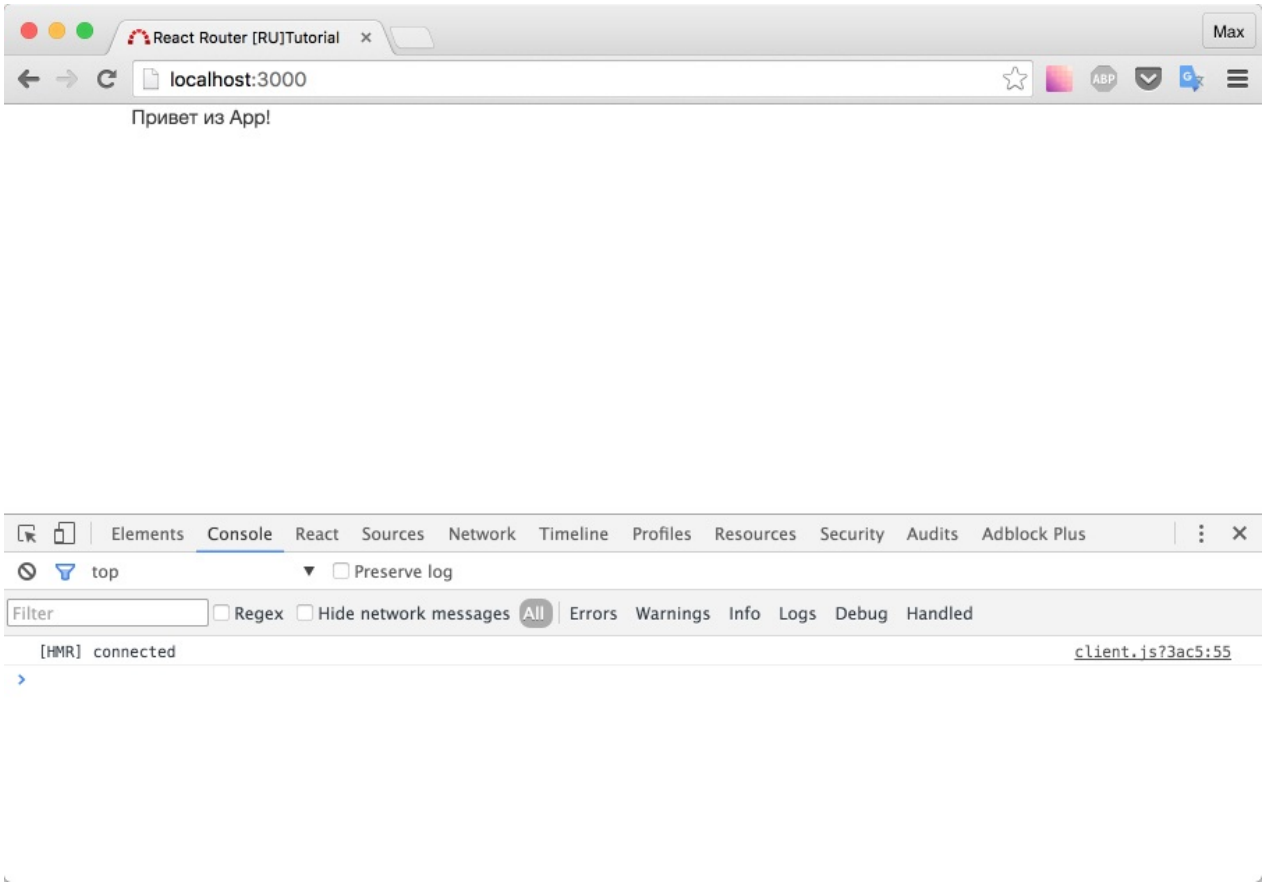
```
import React, { Component } from 'react'

export default class App extends Component {
  render() {
    return <div className='container'>Привет из App!</div>
  }
}
```

Ракета готова к запуску? На старт, внимание... `npm install`.

Теперь точно готова ;)

Проверьте, `npm start` (*каждая первая сборка webpack после его остановки занимает продолжительное время (~6 секунд), последующие "пересборки" значительно быстрее*)

Откройте браузер (обратите внимание так же и на **css**, должен примениться bootstrap-стиль к классу `.container`)



Итого: мы настроили рабочее окружение.

Исходный код на данный момент.

Вопросы:

1. Что хранится в свойстве `process.env.npm_lifecycle_event` ?
2. Где можно посмотреть правила для тонкой настройки ESLint?
3. Что обозначают цифры 0, 1, 2 напротив правила в конфигурационном файле ESLint?

Ответы:

Что хранится в свойстве `process.env.npm_lifecycle_event` ?

- Переменная (variable), которую вы указали в качестве npm **variable**. Подробнее здесь.

Где можно посмотреть правила для тонкой настройки ESLint?

- http://eslint.org/docs/rules/

Что обозначают цифры 0, 1, 2 напротив правила в конфигурационном файле ESLint?

- 0 - правило отключено
- 1 - правило включено, если нарушение выявлено - выведет warning
- 2 - правило включено, если нарушение выявлено - выведет error

Создаем примитивный роутер

Для создания этого раздела я не стал выдумывать "велосипед" и взял пример из офф.документации подправив его под наши нужды.

Что есть роутинг в простом варианте? Соответствие URL-адреса некоему состоянию нашего приложения. Скорее всего: соответствие адреса "рендеру" (*render*) какого-то ключевого компонента.

Сейчас, мы будем изменять лишь часть URL-адреса (после #). Для прослушивания такого рода изменений, воспользуемся событием hashchange.

Создадим три одинаковых компонента:

src/components/Admin.js

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

export default class Admin extends Component {
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Раздел /admin</div>
      </div>
    )
  }
}
```

src/components/Genre.js

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

export default class Genre extends Component {
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Раздел /genre</div>
      </div>
    )
  }
}
```

src/components/Home.js

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

export default class Home extends Component {
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Раздел </div>
      </div>
    )
  }
}
```

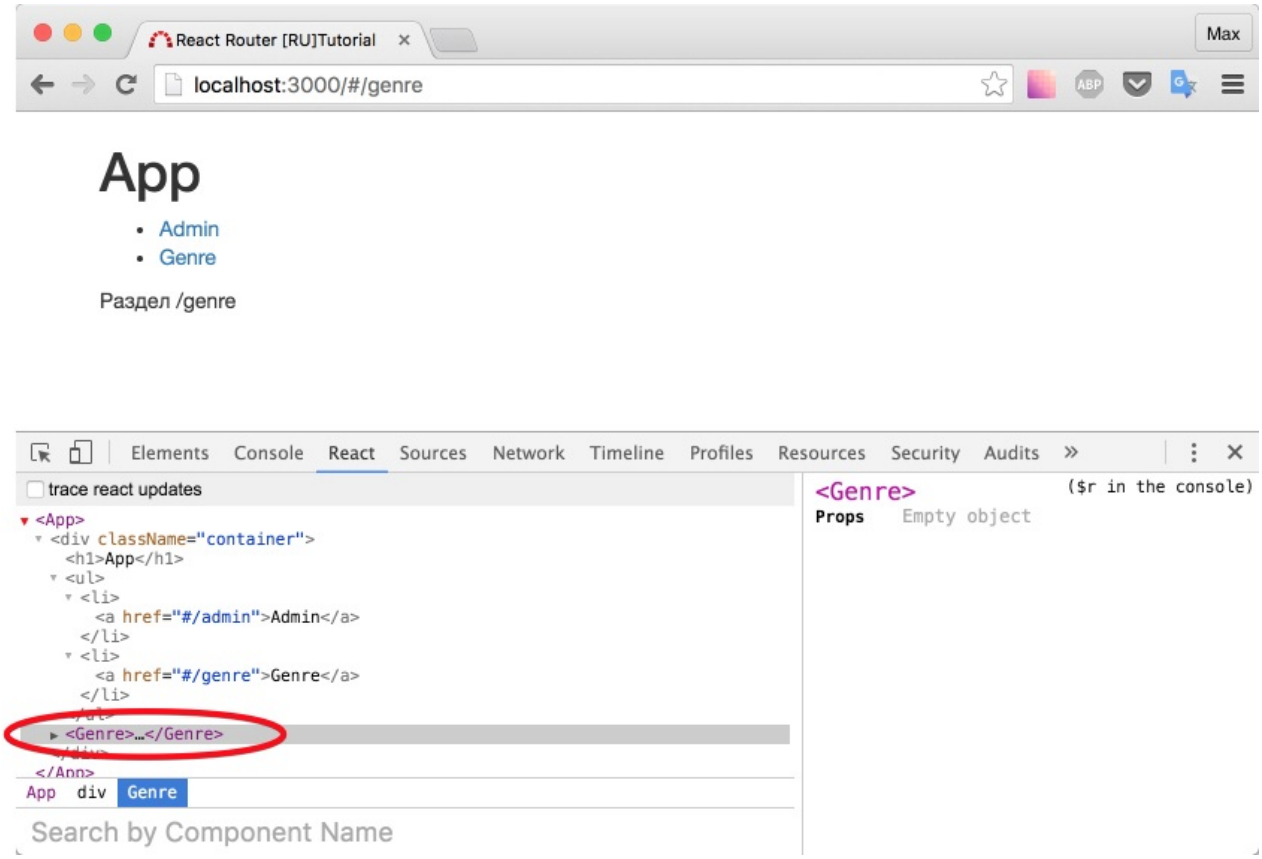
Добавим логики в компонент `<App />`, чтобы если хэш (часть URL-адреса после #) равна `/admin` - показывай содержимое компонента `<Admin />`, если равна `/genre` - `<Genre />`, иначе показывай содержимое `<Home />`.

src/containers/App.js

```
import React, { Component } from 'react'
import Admin from '../components/Admin'
import Genre from '../components/Genre'
import Home from '../components/Home'


export default class App extends Component {
  constructor(props) {
    super(props)
    this.state = {
      route: window.location.hash.substr(1)
    }
  }
  componentDidMount() {
    window.addEventListener('hashchange', () => {
      this.setState({
        route: window.location.hash.substr(1)
      })
    })
  }
  render() {
    let Child

    switch (this.state.route) {
      case '/admin': Child = Admin; break;
      case '/genre': Child = Genre; break;
      default: Child = Home;
    }

    return (
      <div className='container'>
        <h1>App</h1>
        <ul>
          <li><a href='#/admin'>Admin</a></li>
          <li><a href='#/genre'>Genre</a></li>
        </ul>
        <Child />
      </div>
    )

  }
}
```

Покликайте ссылки. Никакой магии в очередной раз.

В момент `componentDidMount` мы подписались на прослушивание события изменения hash-части URL-адреса. Изменяя *state* мы вызываем *ре-рендер* (повторный рендер) компонента и поэтому в переменной **Child** оказывается нужное значение. Таким образом запись `<Child />` превращается в `<Admin />`, `<Genre />` или `<Home />` соответственно.



Обратите внимание, корректная запись именно `<Child />`, а не `<{Child} />`.

```
ERROR in ./src/containers/App.js

/Users/user/developments/local/react-router-course-ru/src/containers/App.js
  37:9  error  Parsing error: Unexpected token

✖ 1 problem (1 error, 0 warnings)


ERROR in ./src/containers/App.js
Module build failed: SyntaxError: /Users/user/developments/local/react-router-course-ru/src/containers/App.js: Unexpected token (37:9)
  35 |           <li><a href='#/genre'>Genre</a></li>
  36 |         </ul>
> 37 |         <{Child} />
     |          ^
  38 |       </div>
  39 |     )
```

Итого: Мы разобрали процесс реализации примитивного роутинга в react-приложении.

Исходный код на данный момент.

Подключаем react-router

Продолжим разбирать страницу официального туториала.

Сейчас код компонента `<App />` не выглядит слишком сложным. Но если развивать идею роутинга, появятся другие URL-адреса, что-то вложится во что-то большее... появится больше ссылок...

Скорее всего, нас ожидает трудноподдерживаемая путаница. Поэтому react-router с одной стороны можно рассматривать как "высокоуровневую" абстракцию, а с другой - полезный "плагин" с массой дополнительных плюшек. Давайте установим его и перепишем наш простой пример, который как обычно, в силу своей простоты, покажется несколько раздутым из-за использования react-router'а.

```
npm i react-router --save
```

src/index.js

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

import { Router, Route, IndexRoute, browserHistory } from 'react-router'

render(
  <Router history={browserHistory}>
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='admin' component={Admin} />
      <Route path='genre' component={Genre} />
    </Route>
  </Router>,
  document.getElementById('root')
)
```

Мы перенесли import компонентов в index.js, а ниже указали роуты. Обратите внимание, на `<IndexRoute />` - так задается роут для "корня" (то есть для */* , еще можно сказать - http://localhost:3000)

Так же, здесь мы использовали одну из главных фишек react-router'а - "вложения" (*nesting*). Мы вложили **admin** и **genre** в */*, поэтому чтобы компоненты были доступны по соответствующим "урлам" (*URL-адресам*), осталось лишь выводить их как "потомков":

src/containers/App.js

```
import React, { Component } from 'react'
import { Link } from 'react-router'

export default class App extends Component {
  render() {
    return (
      <div className='container'>
        <h1>App</h1>
        <ul>
          <li><Link to='/admin'>Admin</Link></li>
          <li><Link to='/genre'>Genre</Link></li>
        </ul>
        {/* добавили вывод потомков */}
        {this.props.children}
      </div>
    )
  }
}
```

Согласно документации, мы также заменили `<a></a>` на компонент реакт-роутера - `<Link />`.

Код реакт-роутера хорошо читается и не изобилует новыми терминами. В следующей главе - продолжим настройку.

Итого: мы подключили react-router, посмотрели как назначается компонент на "корень" сайта, а так же познакомились с компонентом `<Link />`

Исходный код на данный момент.

Задача

Если бы у нас был компонент `<BanList />`, доступный по адресу: *http://localhost:3000/admin/banlist*, как мог бы выглядеть роутер?

Решение

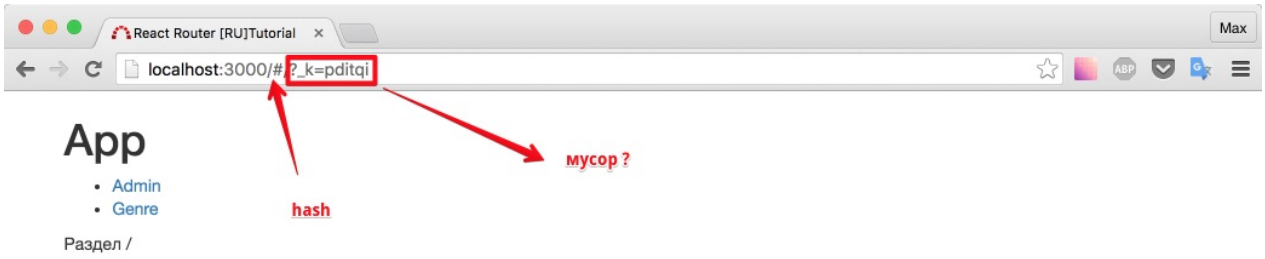
```
...
render(
  <Router history={browserHistory}>
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='admin' component={Admin}>
        <Route path='banlist' component={BanList} />
      </Route>
      <Route path='genre' component={Genre} />
    </Route>
  </Router>,
  document.getElementById('root')
)
...
```

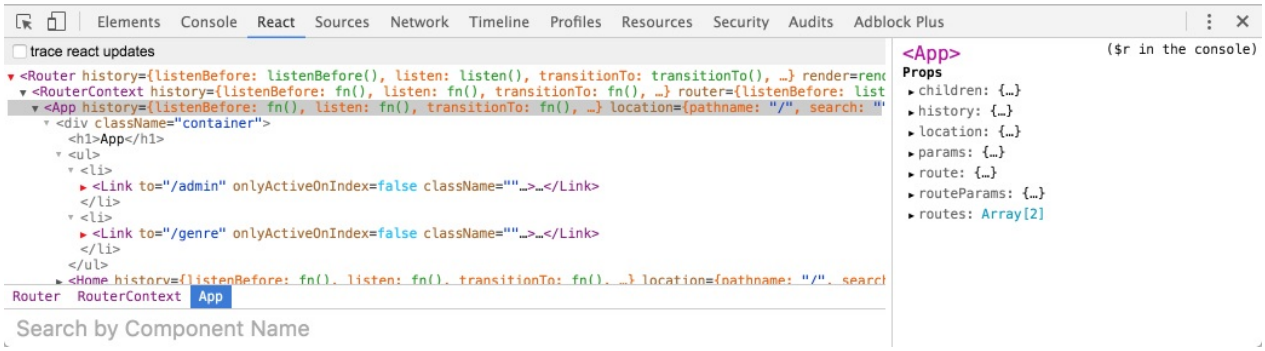
BrowserHistory vs HashHistory

(глава для расширения кругозора)

До текущего времени мы использовали *browserHistory* (смотри *src/index.js*).

Можно использовать и *hashHistory*. Замените в *src/index.js* все вхождения browserHistory на hashHistory, откройте снова `localhost:3000` , видите в адресе #? Покликайте по ссылкам.





"Мусор" в адресе используется так сказать, для того чтобы приблизиться к нативному поведению браузера. Что-то вроде уникального ключа. (дословно здесь (EN)).

Спрашивается, зачем нам *hashHistory* ? Ответ простой - если использовать *hashHistory*, придется меньше телодвижений делать для настройки роутинга на сервере. Давайте вернемся к *browserHistory*, а так же добавим компонент `<NotFound />` .

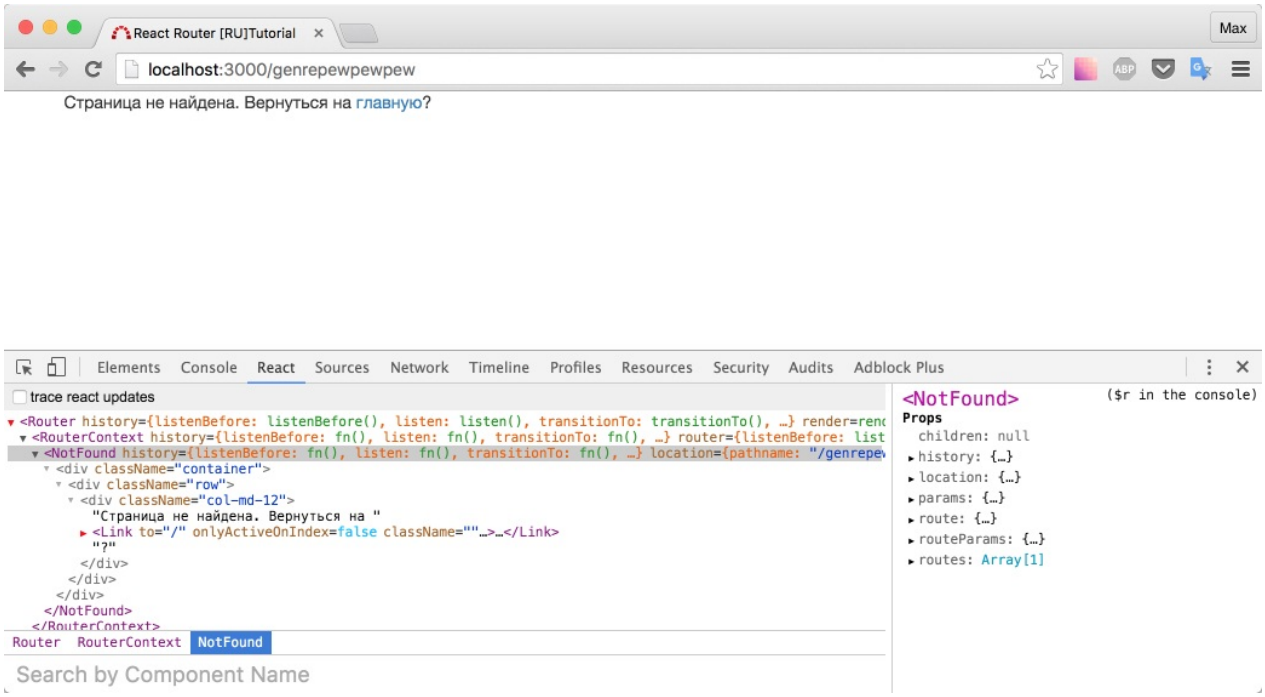
src/components/NotFound.js

```
import React, { Component } from 'react'
import { Link } from 'react-router'

export default class NotFound extends Component {
  render() {
    return (
      <div className='container'>
        <div className='row'>
          <div className='col-md-12'>
            Страница не найдена. Вернуться на <Link to='/'>главную</Link>?
          </div>
        </div>
      </div>
    )
  }
}
```

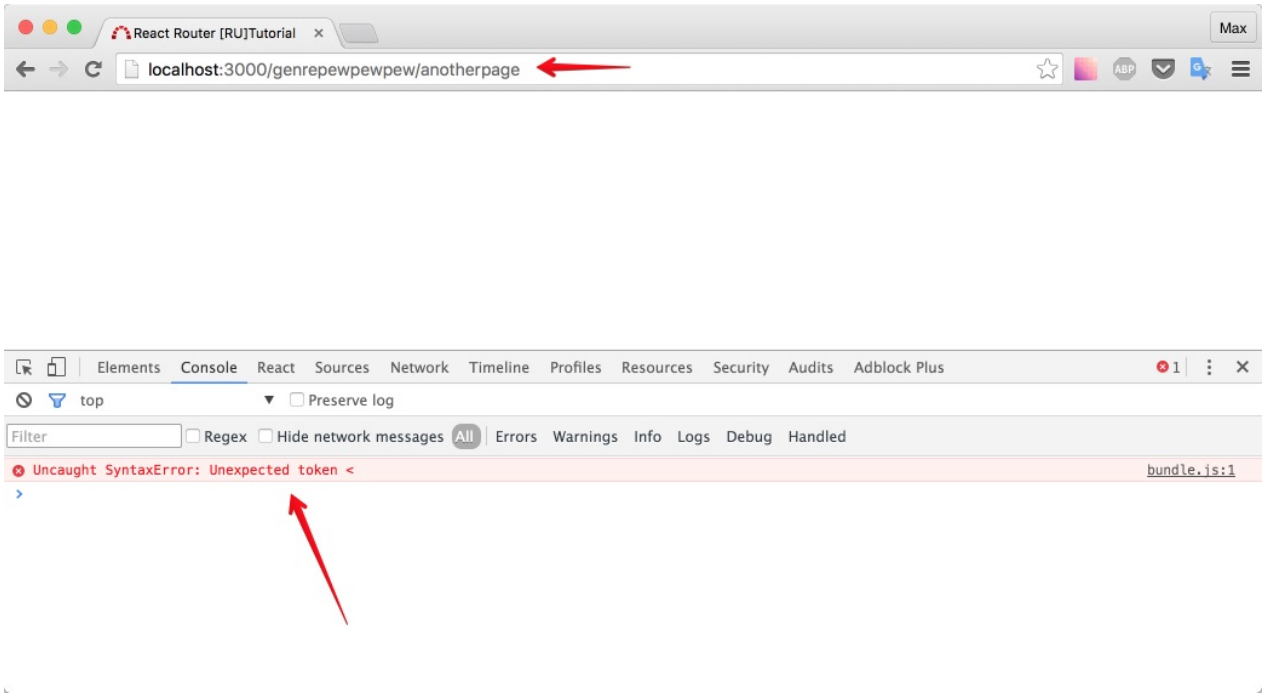
Добавим `<NotFound />` в список роутов:

```
import 'babel-polyfill'
import React from 'react'
import { render } from 'react-dom'
import App from './containers/App'
import Admin from './components/Admin'
import Genre from './components/Genre'
import Home from './components/Home'
import NotFound from './components/NotFound'

import { Router, Route, IndexRoute, browserHistory } from 'react-router'

render(
  <Router history={browserHistory}>
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='admin' component={Admin} />
      <Route path='genre' component={Genre} />
    </Route>
    {/* для всех остальных роутов: показывай NotFound */}
    <Route path='*' component={NotFound} />
  </Router>,
  document.getElementById('root')
)
```

Попробуйте открыть http://localhost:3000/genrepewpewpew

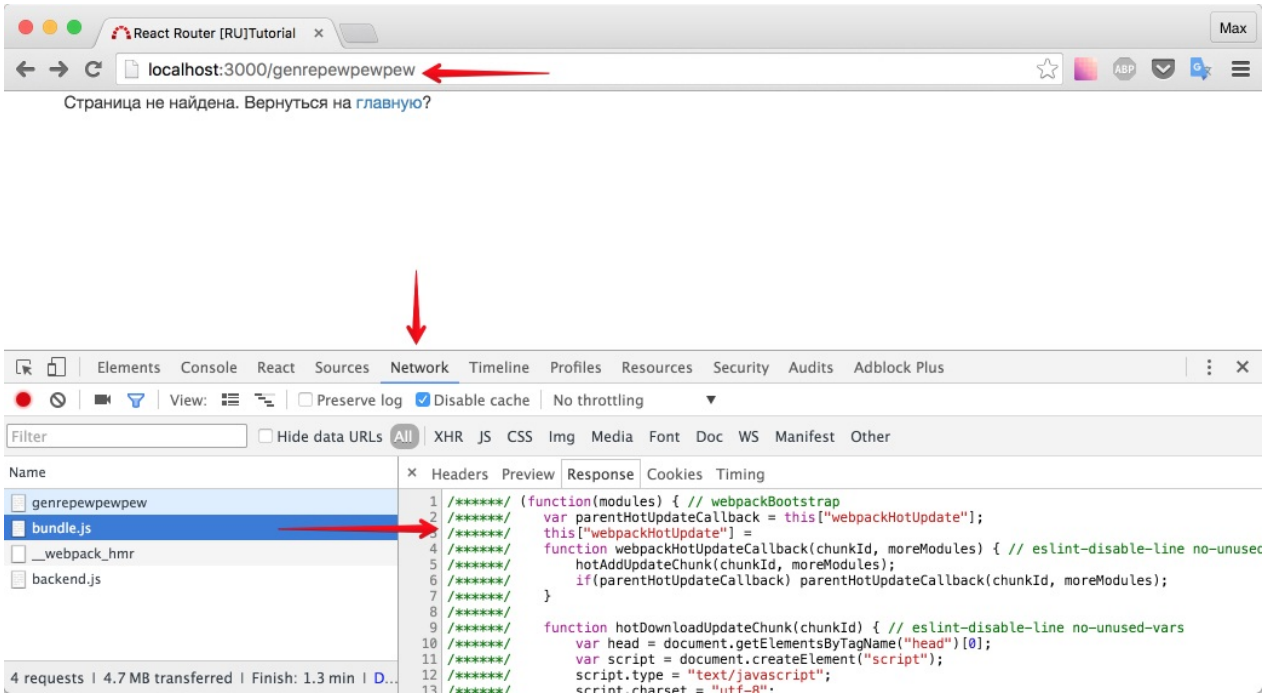


А теперь - http://localhost:3000/genrepewpewpew/anotherpage

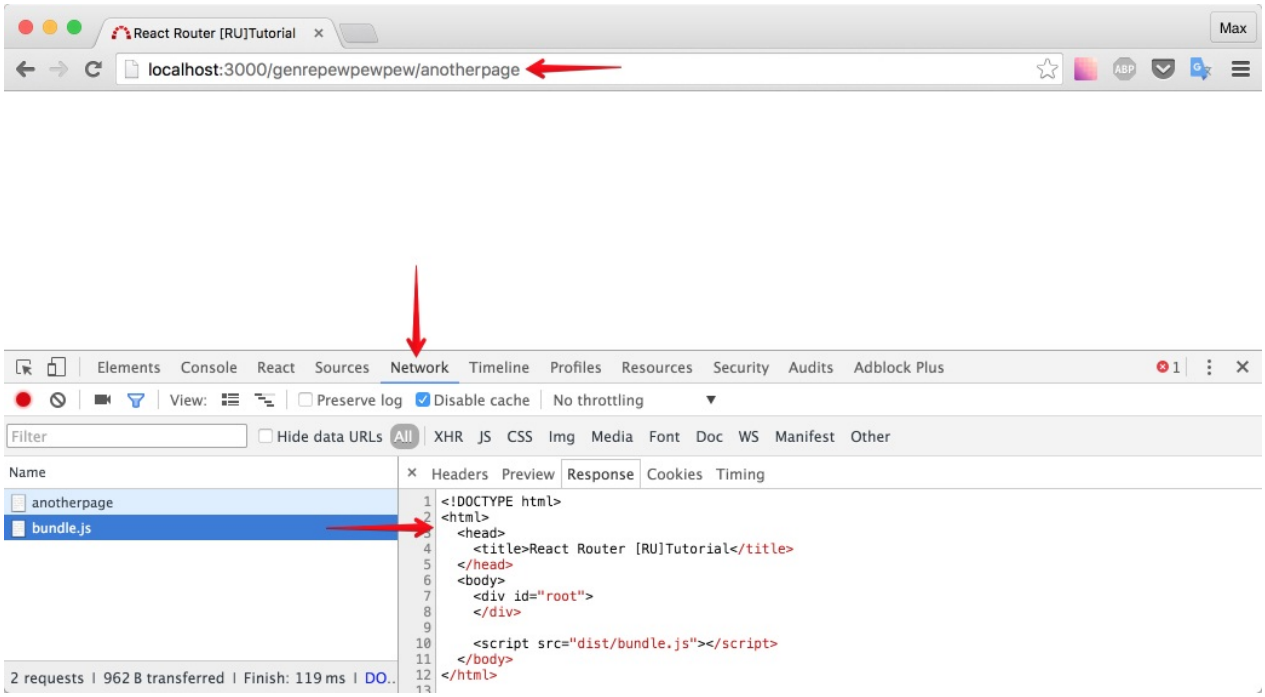


Почему? Зайдите на вкладку Network и проверьте, что отдает сервер внутри *bundle.js*

Корректный *bundle.js* для http://localhost:3000/genrepewpewpew



Некорректный *bundle.js* для http://localhost:3000/genrepewpewpew/anotherpage



Как думаете, в чем проблема?

Ответ кроется во фразе: "нам придется меньше телодвижений делать для настройки роутинга на сервере". Посмотрите сами

server.js

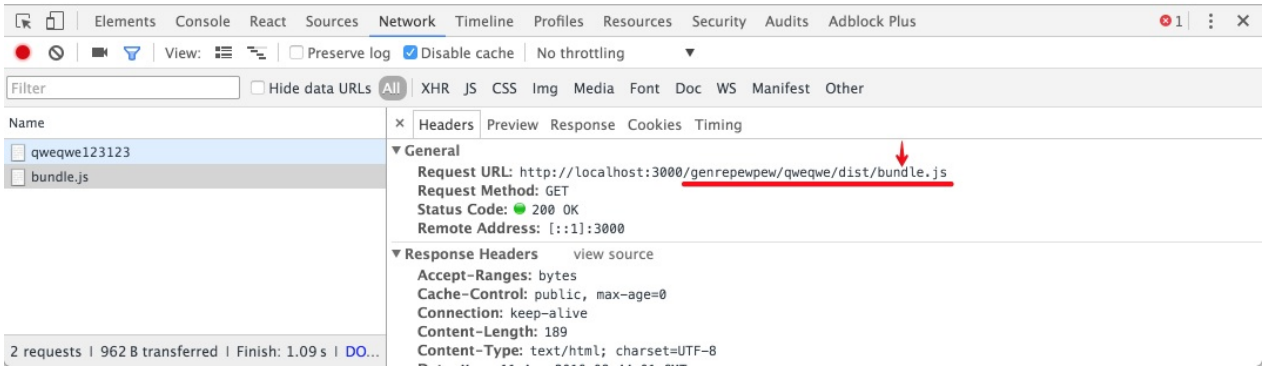
```
app.get(/.*/, function root(req, res) {
  res.sendFile(__dirname + '/index.html');
});
```

Переведем написанное: на все запросы отдавай **index.html**. Код *index.html* предельно прост:

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>React Router [RU]Tutorial</title>
  </head>
  <body>
    <div id="root">
    </div>

    <script src="dist/bundle.js"></script>
  </body>
</html>
```

Тем не менее, здесь допущена ошибка! Используется относительный путь для файла *bundle.js*, снова посмотрим в консоль хрома:



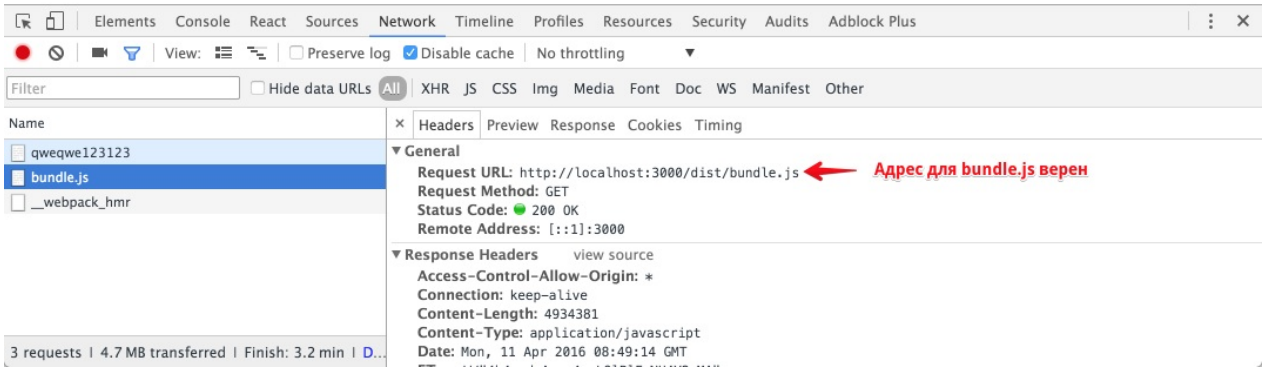
К счастью, такую ошибку легко исправить, и мы по прежнему можем лицезреть красивый вид адресов, используя *browserHistory*.

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>React Router [RU]Tutorial</title>
  </head>
  <body>
    <div id="root">
    </div>
    <!-- добавим / в пути к bundle.js -->
    <script src="/dist/bundle.js"></script>
  </body>
</html>
```

Не забудьте перезапустить webpack, так как *index.html* не находится под наблюдением для пересборки.

Проверьте куда уходит запрос для *bundle.js* сейчас и убедитесь, что путь верен. Убедитесь, что в случае неправильного адреса показывается компонент `<NotFound />`



Альтернативное мнение James K Nelson'a: не нужно использовать pushState (а следовательно browserHistory).

P.S. James предлагает вообще не использовать react-router, но нам же нужно что-то разбирать в этом туториале ;)

Итого: *browserHistory* требует большей поддержки на сервере, но url-адрес выглядит симпатичнее.

Исходный код на данный момент.

Дописываем роуты

Пока я писал этот курс, вышел react 15, давайте обновим сразу react и react-dom.

Далее в курсе будут использованы следующие версии:

```
"react": "^15.0.1",
"react-dom": "^15.0.1",
```

Давайте вспомним, какие роуты мы хотели сделать:

```
/ - главная страница
/list - список жанров
/genre/:genre/ - список релизов данного жанра
/genre/:genre/:release - информация о релизе
/admin - страница администратора
```

Каждый раз, задача будет примерно такой:

- нужно создать компонент (и рендерить в нем детей, если есть дочерние url'ы)
- добавить компонент, как новый Route

Начнем с простого - роут, не имеющий детей - /list

src/components/List.js

```
import React, { Component } from 'react'
import { Link } from 'react-router'

export default class List extends Component {
  render() {
    return (
      <div>
        <div className='row'>
          <div className='col-md-12'>
            <h3> Список жанров </h3>
          </div>
        </div>
        <div className='row'>
          <div className='col-md-12'>
            <ul className='list'>
              <li className='list__item'>
                <Link to='/genre/house/'>House</Link>
              </li>
              <li className='list__item'>
                <Link to='/genre/dnb/'>Drum and bass</Link>
              </li>
              <li className='list__item'>
                <Link to='/genre/hip-hop/'>Hip-hop</Link>
              </li>
            </ul>
          </div>
        </div>
      </div>
    )
  }
}
```

Обновим информацию о роутах: *src/index.js*

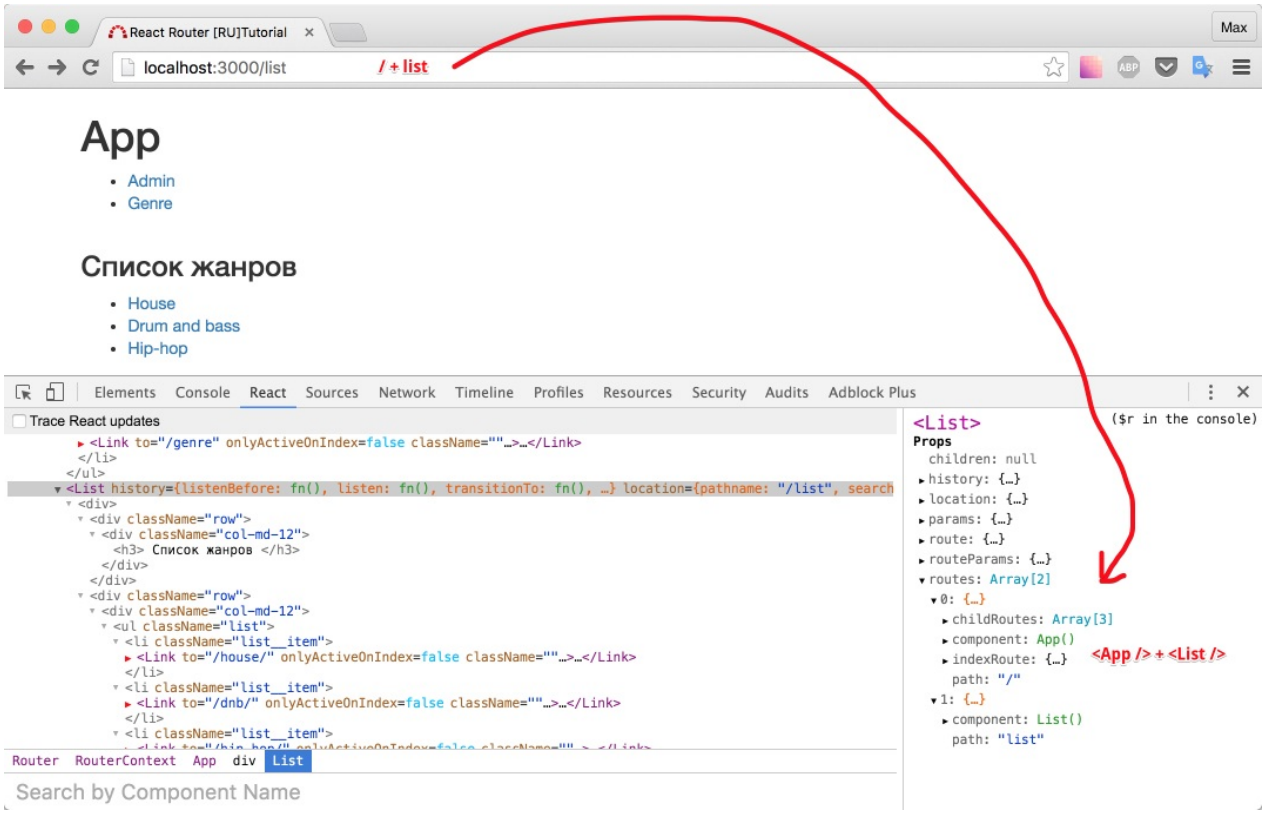
```
...
import List from './components/List'
...

    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='admin' component={Admin} />
      <Route path='genre' component={Genre} />
      <Route path='list' component={List} />
    </Route>
...
```

P.S. в данный момент у нас не поддерживается hot-reload для роутинга. Поэтому придется обновлять браузер по старинке.

Будьте внимательны, я специально привожу не полный код, чтобы вы не страдали *copy-paste*.

`<List />` - дочерний компонент `<App />`, (посмотрите на URL - `/ + list` = две части, в переводе на react-router, это `<App />` + **this.props.children**, и конкретно в нашем случае это `<App />` + `<List />`)

Проверим в браузере:



Обратите внимание, что у `<App />` - `childRoues: Array[3]`, именно этими детьми и являются `<Admin />`, `<Genre />` и `<List />` в данный момент. То есть, снова никакой магии.

Так же, обратите внимание сколько свойств "прокидывает" react-router. Это нам еще пригодится.

Кстати, если сейчас кликнуть по ссылке, например *House*, что отобразит браузер?

Ответ - отобразится компонент `<NotFound />`.

Покликав на ссылки, у нас вырисовываются следующие адреса:

```
http://localhost:3000/genre/house/
http://localhost:3000/genre/dnb/
http://localhost:3000/genre/hip-hop/
```

Для всех этих адресов, требуется отрисовывать один и тот же компонент, который будет отображать список релизов данного жанра. Как быть? Пора познакомиться с **динамическим** адресом.

Динамический роут

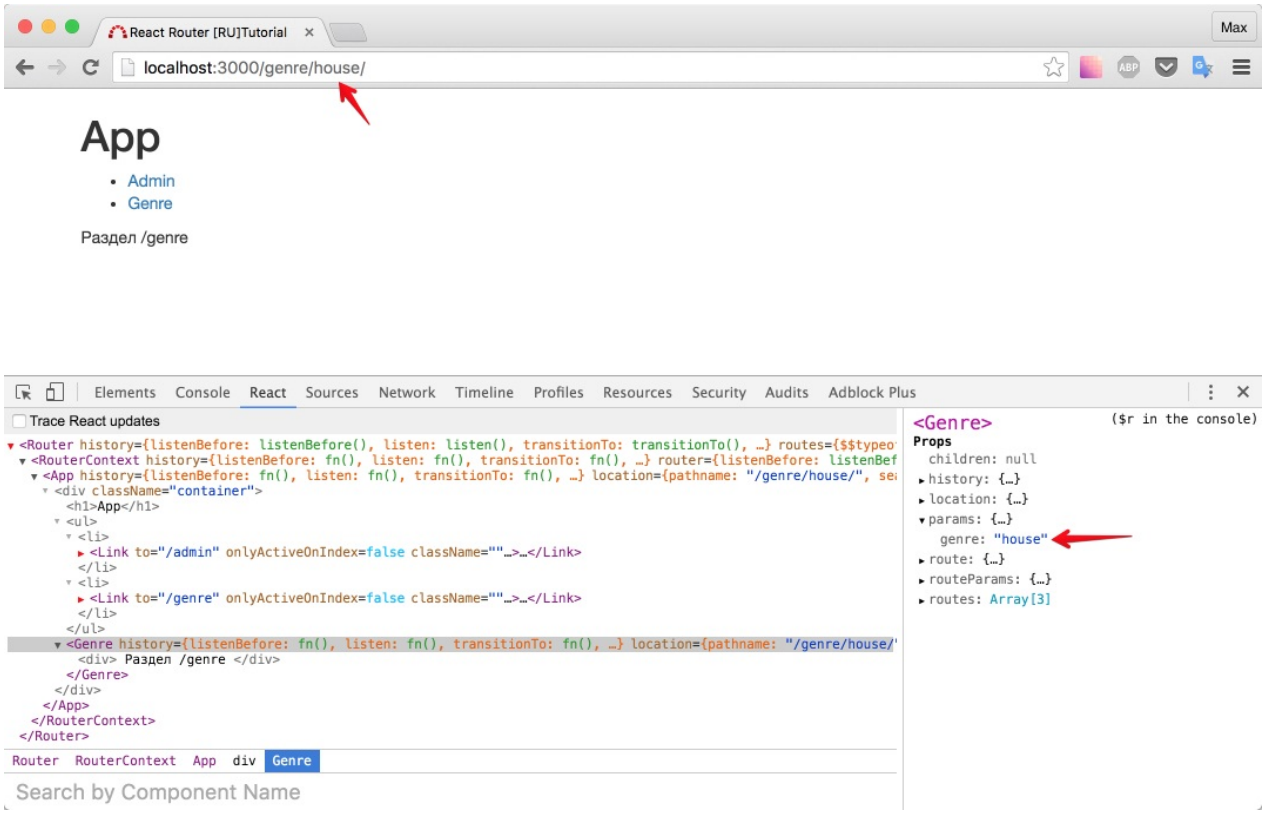
Все очень просто. Сразу к делу. Изменим в *src/index.js*:

```
<Route path='genre' component={Genre} />
```

на

```
<Route path='genre/:genre' component={Genre} />
```

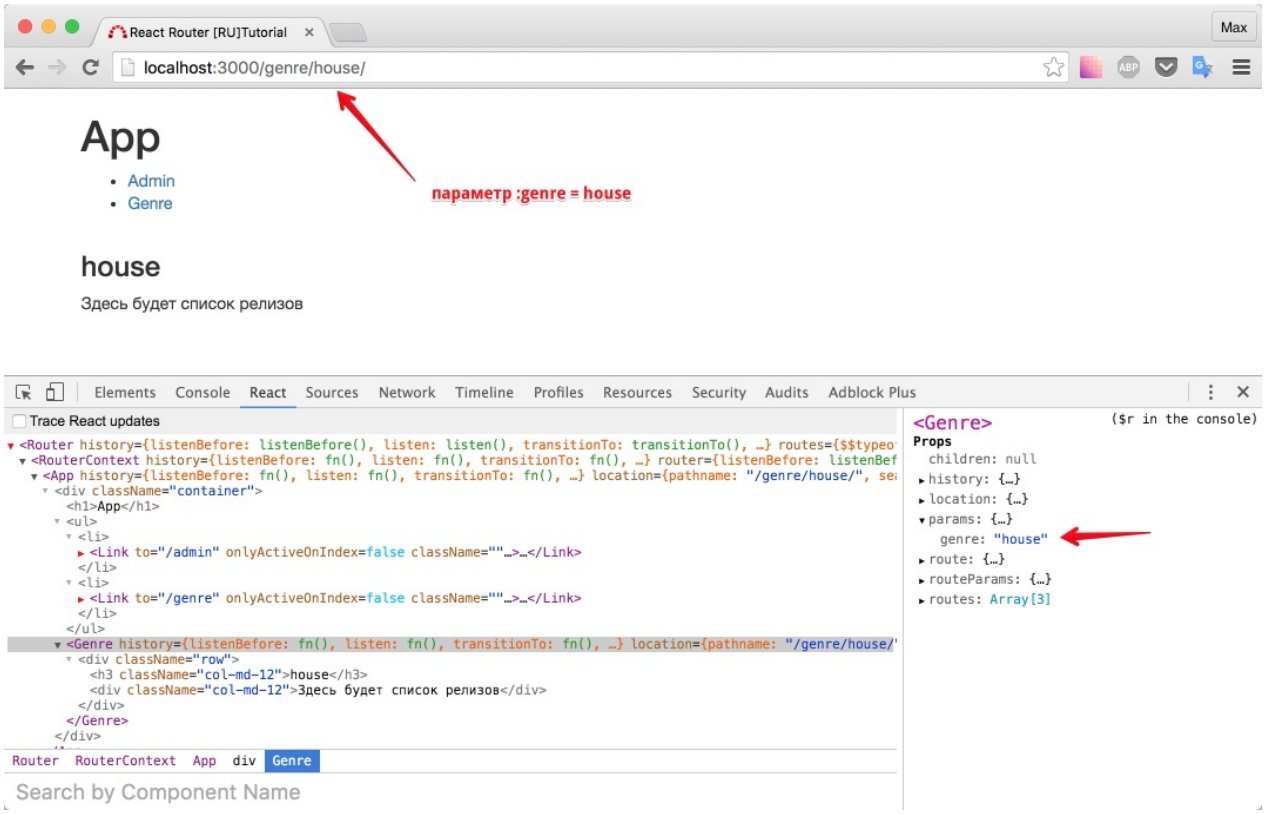
Обновите браузер:



Отображается компонент `<Genre />`, при этом в **props** появилось свойство *params*, давайте сразу возьмем его в оборот:

src/components/Genre.js

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

export default class Genre extends Component {
  render() {
    return (
      <div className='row'>
        <h3 className='col-md-12'>{this.props.params.genre}</h3>
        <div className='col-md-12'>Здесь будет список релизов</div>
      </div>
    )
  }
}
```



Если попробовать ввести вместо *house* другое слово - все так же будет работать.

Будем называть динамическую часть url'а - **параметром**. То есть *:genre* - параметр.

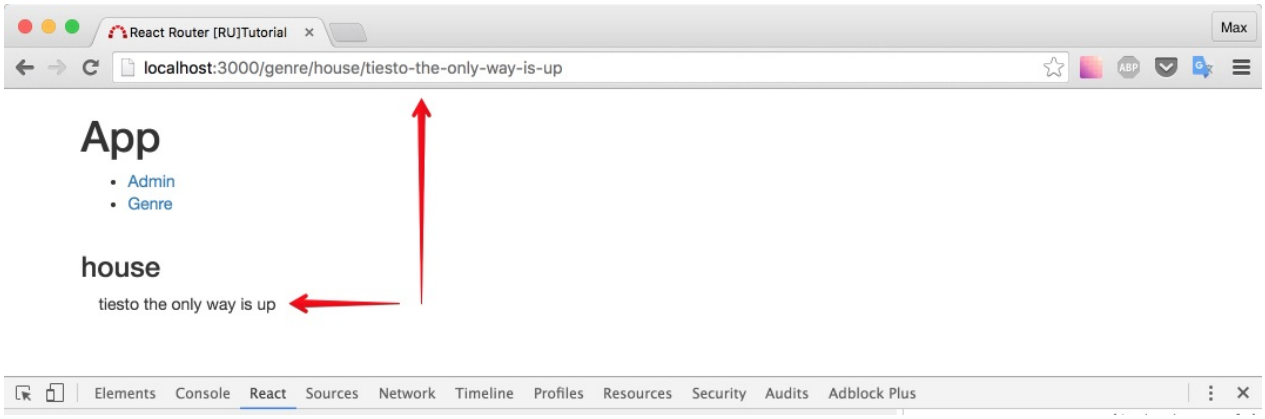
Нет никакого ограничения на количество параметров в адресе. Например:

```
localhost:3000/:param_one/not_param/:param_two/:param_three
```

Задача: создать компонент , который будет доступен по адресу:

```
localhost:3000/genre/house/tiesto-the-only-way-is-up
```

и будет выглядеть так:



Подсказка #1: Необходимо использовать *nesting* (вложения), а значит - потребуется рендерить `this.props.children`

Решение:

src/index.js

```
...

import Release from './components/Release'

...
  <Router history={browserHistory}>
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='admin' component={Admin} />
      <Route path='genre/:genre' component={Genre}>
           {/* добавили новый route */}
        <Route path=':release' component={Release} />
      </Route>
      <Route path='list' component={List} />
    </Route>
    <Route path='*' component={NotFound} />
  </Router>,
...
```

src/components/Genre.js

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

export default class Genre extends Component {
  render() {
    let template;
    {/* если параметр release есть - покажи дочерний компонент */}
    if (this.props.params.release) {
      template = (
        <div className='row'>
          <h3 className='col-md-12'>{this.props.params.genre}</h3>
          <div className='col-md-12'>{this.props.children}</div>
        </div>
      )
    } else {
      template = (
        <div className='row'>
          <h3 className='col-md-12'>{this.props.params.genre}</h3>
          <div className='col-md-12'>Здесь будет список релизов</div>
        </div>
      )
    }

    return template;
  }
}
```

src/components/Release.js

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

export default class Release extends Component {
  render() {
    {/* замени все '-' в параметре (то есть в адресе) на пробелы */}
    const releaseName = this.props.params.release.replace(/-/g,' ');
    return (
      <div className='col-md-12'>
        {releaseName}
      </div>
    )
  }
}
```

С параметром мы можем работать как с любым другим свойством.

Оформление routes

Чем глубже уровень вложение в роутах, тем сложнее сразу понять какому роуту соответствует тот или иной компонент. Поэтому, мы можем указать путь полностью:

```
<Route path='/' component={App}>
  <IndexRoute component={Home} />
  <Route path='/admin' component={Admin} />
  <Route path='/genre/:genre' component={Genre}>
    <Route path='/genre/:genre/:release' component={Release} />
  </Route>
  <Route path='/list' component={List} />
</Route>
```

Так же, давайте вынесем *routes* в отдельный файл.

src/routes.js

```
import React from 'react'
import { Route, IndexRoute } from 'react-router'

import App from './containers/App'
import Admin from './components/Admin'
import List from './components/List'
import Genre from './components/Genre'
import Release from './components/Release'
import Home from './components/Home'
import NotFound from './components/NotFound'

export const routes = (
  <div>
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='/admin' component={Admin} />
      <Route path='/genre/:genre' component={Genre}>
        <Route path='/genre/:genre/:release' component={Release} />
      </Route>
      <Route path='/list' component={List} />
    </Route>
    <Route path='*' component={NotFound} />
  </div>
)
```

Так как *routes* - реакт компонент, и мы используем `<NotFound />` не в качестве дочернего компонента (чтобы не было "шапки"), нам пришлось обернуть роуты в `<div>`. Вы можете сделать иначе, если хотите, чтобы шапка (сейчас это ссылки Admin и Genre) была на всех страницах, включая 404.

Обновим код для подключения роутов:

src/index.js

```
import 'babel-polyfill'
import React from 'react'
import { render } from 'react-dom'
import { Router, browserHistory } from 'react-router'
import { routes } from './routes'

render(
  <Router history={browserHistory} routes={routes} />,
  document.getElementById('root')
)
```

Приберемся

У нас появилась нерабочая ссылка в шапке - Genre. Так как она нам больше не нужна, давайте заменим ее на 'Список жанров' и будем отображать компонент `<List />` по клику на эту ссылку.

Заодно переименуем Admin. Удалим заголовок App и накинем парочку bootstrap классов тэгу `<ul>`.

src/containers/App.js

```
import React, { Component } from 'react'
import { Link } from 'react-router'

export default class App extends Component {
  render() {
    return (
      <div className='container'>
        <ul className='nav nav-pills'>
          <li><Link to='/admin'>Админка</Link></li>
          <li><Link to='/list'>Список жанров</Link></li>
        </ul>
        {this.props.children}
      </div>
    )
  }
}
```

Проверьте в браузере, покликайте по ссылкам. Понажимайте вперед/назад.

Итого: мы рассмотрели как можно создать динамический роут, и как можно использовать динамическую часть в шаблоне компонента.

Исходный код на данный момент (комментарии удалены)

Активная ссылка

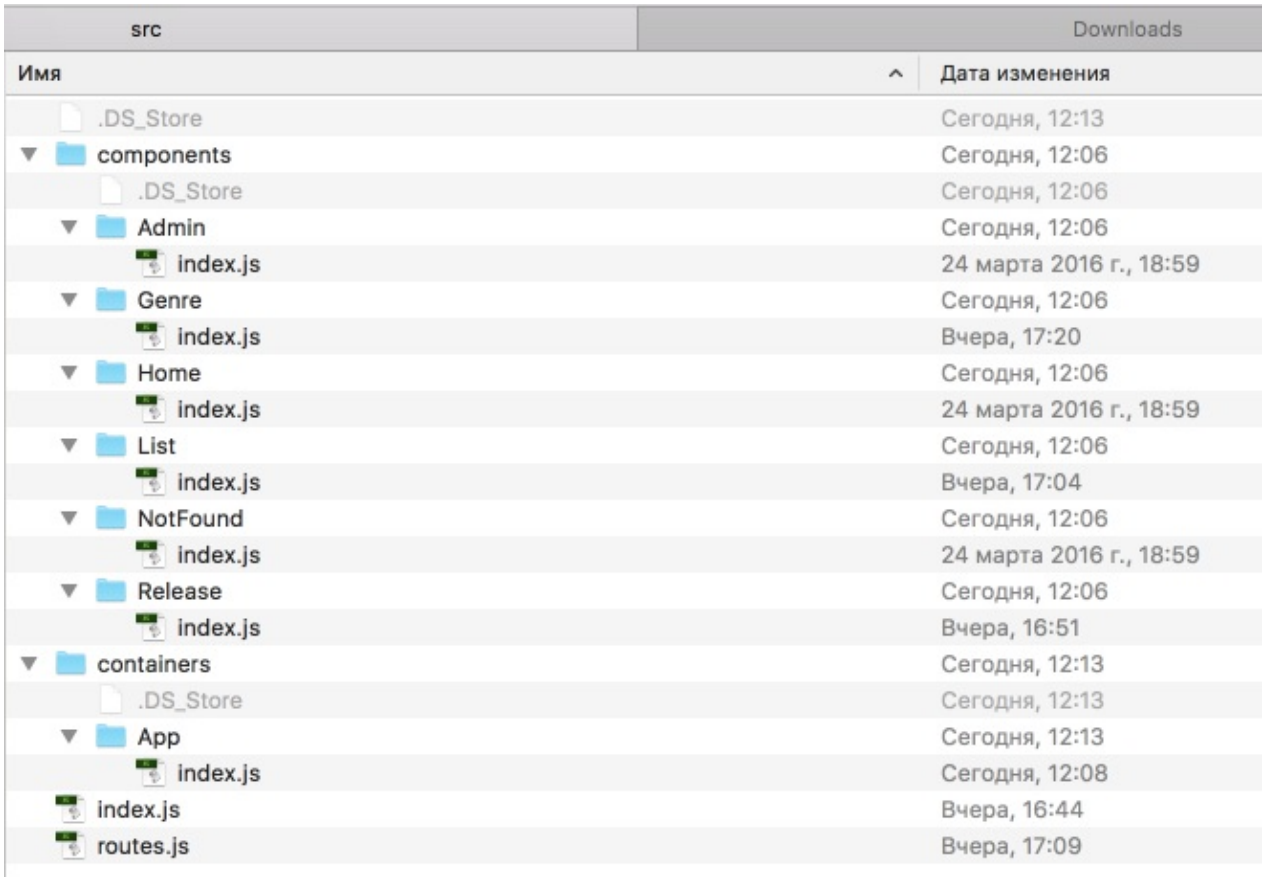
Кто не мечтал подкрасить активную ссылку - тот не делал сайтов (ц) - народная мудрость.

Однако, чтобы подкрасить ссылку, нам нужно куда-то положить стили. Для этого, еще раз взглянем на любой компонент из проекта redux-easy-boilerplate: стили положены вместе с компонентом. Предлагаю делать так же.

Порефакторим

Создайте для всех компонентов свою директорию и положите в нее *index.js* файл. Файлы стилей будем добавлять по мере необходимости.

Необходимо создать директорию с названием, равным названию файла компонента, а сам файл компонента переименовать в index.js В таком случае, не придется даже изменять что-то в подключении роутов.



Создадим файл стилей для компонента (*контейнера*) `<App />`

src/containers/App/styles.scss

```
.active {
  font-weight: bold;
}
```

Чтобы ссылка реагировала на активность, нужно добавить компоненту `<Link />` атрибут **activeClassName**. Например:

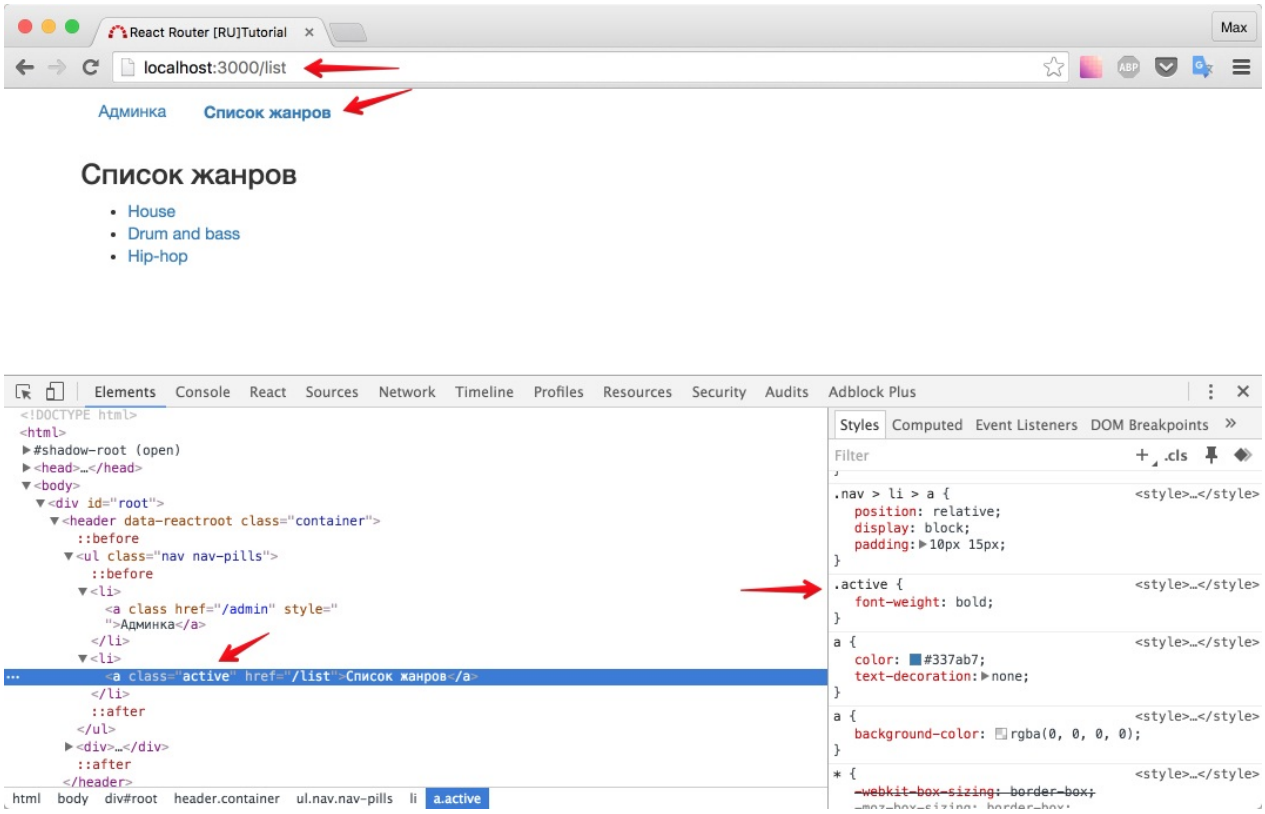
```
<Link to='/list' activeClassName='active'>Список жанров</Link>
```

Важно не забыть импортировать файл стилей. Обработка scss в конфиге вебпака уже прописана, поэтому остается лишь изменить содержимое `<App />`:

src/containers/App/index.js

```
import React, { Component } from 'react'
import { Link } from 'react-router'

import './styles.scss'

export default class App extends Component {
  render() {
    return (
      <div className='container'>
        <ul className='nav nav-pills'>
          <li><Link to='/admin' activeClassName='active'>Админка</Link></li>
          <li><Link to='/list' activeClassName='active'>Список жанров</Link></li>
        </ul>
        {this.props.children}
      </div>
    )
  }
}
```

Проверим?



Туториал (EN) настаивает, чтобы я рассказал вам еще немного. Поэтому я приведу краткий перевод ниже.

Навигационные ссылки (краткий перевод)

Обычно вам не нужно, чтобы ссылки на сайте умели быть активными/не активными. Это требуется только для навигационных ссылок. Чтобы не писать *activeClassName* "ручками", можно сделать обертку над компонентом `<Link />`,

Мы будем использовать *spread-оператор* (троеточие **...** , по-русски здесь). Он склонирует все *props*, а значит и *activeClassName*. То что нам нужно.

Создадим компонент `<NavLink />` (а так же, переместите *styles.scss* из *src/containers/App* в *src/components/NavLink*)

src/components/NavLink/index.js

```
import React, { Component } from 'react'
import { Link } from 'react-router'

import './styles.scss'

export default class NavLink extends Component {
  render() {
    return <Link {...this.props} activeClassName='active'/>
  }
}
```

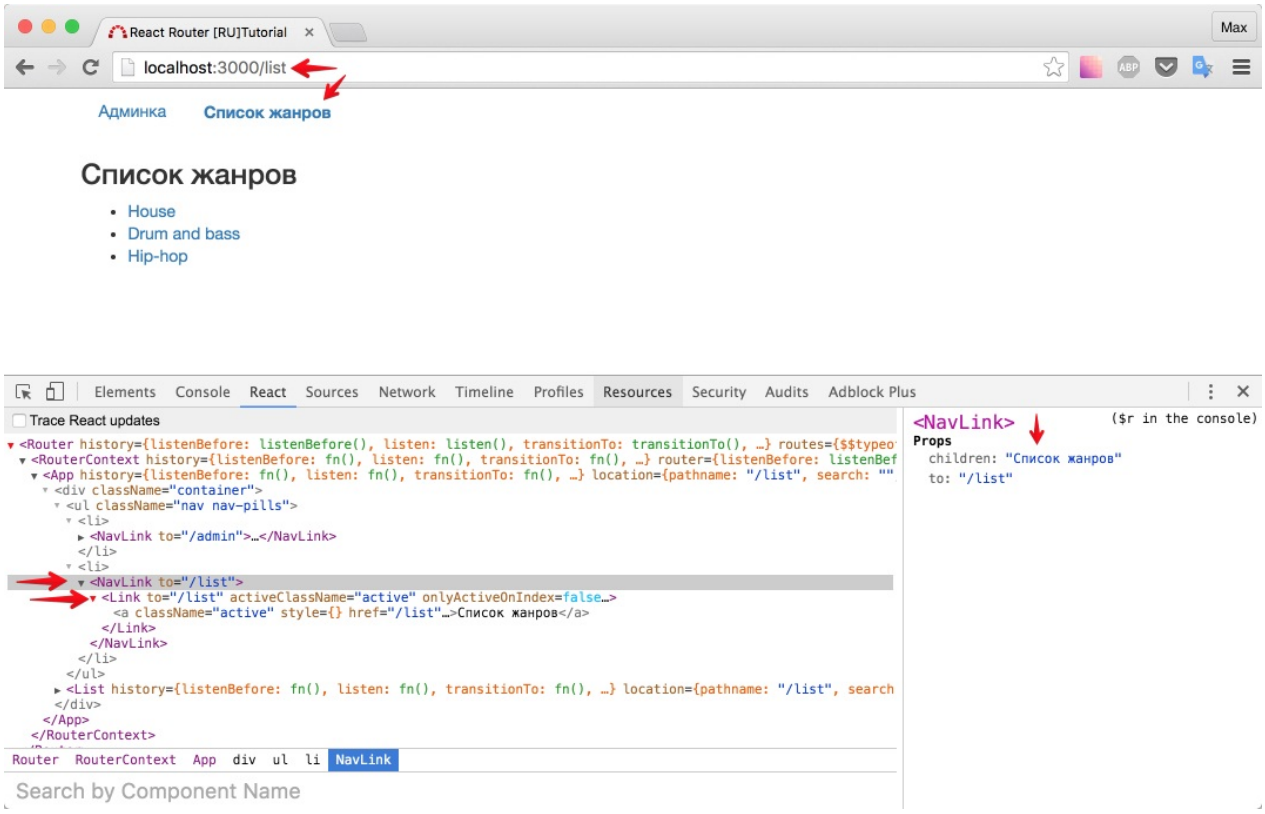
Остается изменить `<App />`:

src/containers/App/index.js

```
import React, { Component } from 'react'
import NavLink from '../../components/NavLink'

export default class App extends Component {
  render() {
    return (
      <div className='container'>
        <ul className='nav nav-pills'>
          <li><NavLink to='/admin'>Админка</NavLink></li>
          <li><NavLink to='/list'>Список жанров</NavLink></li>
        </ul>
        {this.props.children}
      </div>
    )
  }
}
```

(удалили импорт `<Link />`, добавили `<NavLink />`, заменили все вхождения *Link* на *NavLink*)

Посмотрим на результат:



Итого: чтобы ссылка "умела быть активной" - достаточно добавить *activeClassName* для компонента `<Link />`.

P.S. прием с оборачиванием компонента - очень мощный. Возьмите его на вооружение.

Исходный код на данный момент.

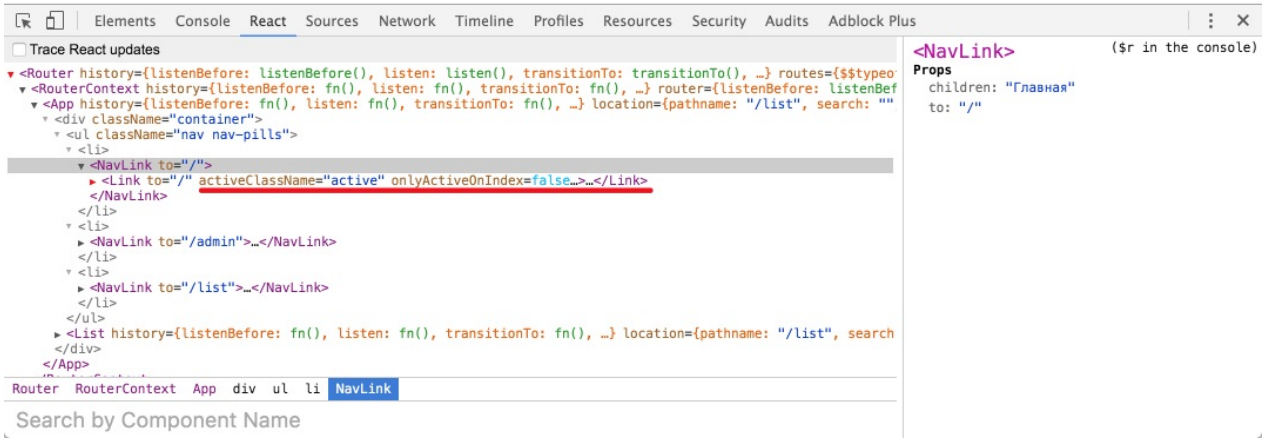
Ссылка на главную

Перед нами простая задача - добавить ссылку на главную. Это навигационная ссылка? Да. Значит используем `<NavLink />`

src/containers/App/index.js

```
import React, { Component } from 'react'
import NavLink from '../../components/NavLink'

export default class App extends Component {
  render() {
    return (
      <div className='container'>
        <ul className='nav nav-pills'>
            {/* добавили ссылку на главную страницу */}
          <li><NavLink to='/'>Главная</NavLink></li>
          <li><NavLink to='/admin'>Админка</NavLink></li>
          <li><NavLink to='/list'>Список жанров</NavLink></li>
        </ul>
        {this.props.children}
      </div>
    )
  }
}
```

Посмотрим в браузере:



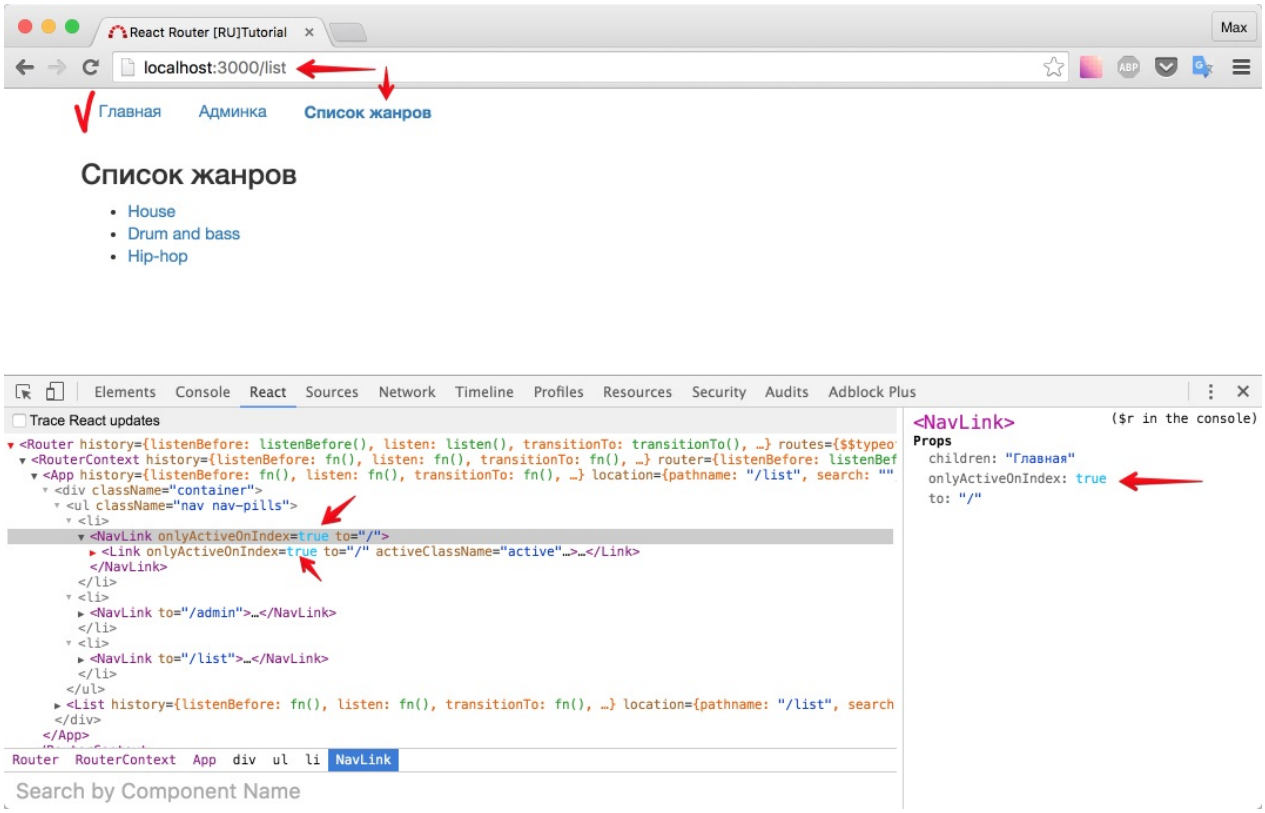
Тот самый случай, когда проблема решается без документации с помощью "инспектора" (вкладки React в консоли хрома). Видим свойство - *onlyActiveOnIndex: false*

Исправляем:

src/containers/App/index.js

```
...
<li><NavLink onlyActiveOnIndex={true} to='/'>Главная</NavLink></li>
...
```

Проверим:



Отлично.

По умолчанию onlyActiveOnIndex - false. Поэтому, так как адрес у нас состоял из двух активных роутов (/ + list) оба компонента `<Link />` получили класс `.active`

В качестве бонуса **вопрос**: мы прописали *onlyActiveOnIndex* для нашего компонента `<NavLink />`, как `<Link />` узнал, что свойство *onlyActiveOnIndex* = *true* ?

Ответ: Все благодаря тому, что мы использовали *spread operator*. Вспомним подробнее строку из файла *src/components/NavLink/index.js*:

```
return <Link {...this.props} activeClassName='active'/>
```

Запись `{...this.props}` означает "возьми все свойства" (возьми все **props**).

Позже мы добавляем ко всем полученным свойствам еще одно - *activeClassName*.

Кстати, если нажать на "Главная" - ссылка подсветится. Это тоже результат работы - onlyActiveOnIndex

Итого: мы научились подсвечивать "ссылку на главную" без ущерба для остальной навигации.

Исходный код на данный момент.

IndexRedirect

Предположим, вам хочется, чтобы пользователь при заходе на сайт, попадал сразу на отображение компонента `<List />` и корректного url-адреса соответственно.

Для этого удалите упоминания об IndexRoute'e и используйте **IndexRedirect**.

Привожу полный код:

src/routes.js

```
import React from 'react'
import { Route, IndexRedirect } from 'react-router'

import App from './containers/App'
import Admin from './components/Admin'
import List from './components/List'
import Genre from './components/Genre'
import Release from './components/Release'
import NotFound from './components/NotFound'

export const routes = (
  <div>
    <Route path='/' component={App}>
      <IndexRedirect to='list' /> {/* INDEX REDIRECT */}
      <Route path='/admin' component={Admin} />
      <Route path='/genre/:genre' component={Genre}>
        <Route path='/genre/:genre/:release' component={Release} />
      </Route>
      <Route path='/list' component={List} />
    </Route>
    <Route path='*' component={NotFound} />
  </div>
)
```

Введите в браузере `http://localhost:3000/` и нажмите *enter*.

Откроется `http://localhost:3000/list`

Что и требовалось доказать. Результаты сохранять не нужно, вернитесь к предыдущей версии файла. Напомню, пользователям git - это можно сделать с помощью:

```
git checkout src/routes.js
```

P.S. для нетерпеливых: если интересно, как происходит *redirect* - используется "хук" на событие *onEnter*. Мы об этом еще не говорили, но вы можете прочитать несколько абзацев на EN здесь.

P.P.S. буквально одной строкой: если вы хотите использовать **редирект** не "главной страницы", то используйте следующий синтаксис:

```
...
<Redirect from='/profile/photos' to='/new_all_photos_page'
/>
...
<Route path='new_all_photos_page' component={NewAllPhotosComponent} />
...
```

Программируем переходы

(конечно, дело касается "переходов по ссылкам")

Давайте рассмотрим очень "выдуманный пример":

На главной странице у нас есть форма, в которой мы можем написать любой жанр. По кнопке - перейти - мы попадаем по адресу:

`http://localhost:3000/genre/жанр_из_формы/`

Динамический роут `/genre/:genre` - как раз позволит нам это.

Практической пользы в этом никакой, но так мы рассмотрим переходы "программно" на деле.

Реальная задача вас ждет уже в следующей главе, но мне бы хотелось, чтобы после решения данного выдуманного примера вы с легкостью решили ее самостоятельно.

Приступим:

src/components/Home/index.js

```
import React, { Component } from 'react'
import { browserHistory } from 'react-router'

export default class Home extends Component {
  handleSubmit(e) {
    e.preventDefault()
    const value = e.target.elements[0].value.toLowerCase()
    browserHistory.push(`/genre/${value}`)
  }
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Раздел /</div>
        <form className='col-md-4' onSubmit={this.handleSubmit}>
          <input type='text' placeholder='genreName'/>
          <button type='submit'>Перейти</button>
        </form>
      </div>
    )
  }
}
```

Все предельно прозрачно: мы использовали *browserHistory* **API**, просто "пушнув" (*pushState*) новую запись в историю.

Замечание: *browserHistory из react-router, использует библиотеку history, которая в свою очередь использует возможности современных браузеров.*

Если мы передадим значение rock - по клику на кнопку нас "перебросит" на:

`http://localhost:3000/genre/rock`

Вопрос валидации здесь не важен, так как цель данного урока не в этом.

Важно отметить, что если вы передадите историю в роутер, **отличную** от того, что используется здесь - приложение сломается. Честно говоря, на практике я не встречался с такой проблемой, но мне нравится ее решение, которое приводится в официальной документации (*ES5 синтаксис*).

Использование this.context для программной навигации

src/components/Home/index.js

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

export default class Home extends Component {
  constructor() {
    super();
    this.handleSubmit = this.handleSubmit.bind(this)
  }
  handleSubmit(e) {
    e.preventDefault()
    const value = e.target.elements[0].value.toLowerCase()
    this.context.router.push(`/genre/${value}`)
  }
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Раздел /</div>
        <form className='col-md-4' onSubmit={this.handleSubmit}>
          <input type='text' placeholder='genreName'/>
          <button type='submit'>Перейти</button>
        </form>
      </div>
    )
  }
}

Home.contextTypes = {
  router: PropTypes.object.isRequired
}
```

Так как, в будущем мы еще столкнемся с использованием **this.context.router** и так как этот вариант является пуленепробиваемым - давайте оставим его.

У меня есть парочка вопросов на знание экосистемы react'a.

1. Зачем используется `this.handleSubmit = this.handleSubmit.bind(this)` ?
2. Будет ли работать `this.context.router.push` если убрать последние три строки (там где проверяются props) ?

Ответы:

1. React с версии 0.14 (предыдущей, перед 15.0) не использует автобиндинг если используются ES2015 классы. Мы должны забиндить *this* явно.
2. Работать не будет, так как `this.context` можно использовать только при наличии проверки *contextTypes*.

Итого: мы можем программно перенаправить браузер пользователя на новый URL. Мы научились использовать this.context используя "классический" (от слова "класс") синтаксис ES2015.

Исходный код на данный момент.

Разделение доступа

В данном разделе, по шагам будет разобран вход на сайт в качестве администратора. Следовательно, мы сделаем недоступным для посещения адрес `localhost:3000/admin`, а так же рассмотрим редирект на "главную" или в "админку" после ввода логина.

Создадим страницу логина.

В качестве "сервера для авторизации", будем использовать localStorage.

Алгоритм простой: вводится логин - кладется в localStorage.

src/components/Login/index.js

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

export default class Login extends Component {
  handleSubmit(e) {
    e.preventDefault()
    const value = e.target.elements[0].value
    window.localStorage.setItem('rr_login', value)
  }
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Пожалуйста, введите логин:</div>
        <form className='col-md-4' onSubmit={this.handleSubmit}>
          <input type='text' placeholder='login'/>
          <button type='submit'>Войти</button>
        </form>
      </div>
    )
  }
}
```

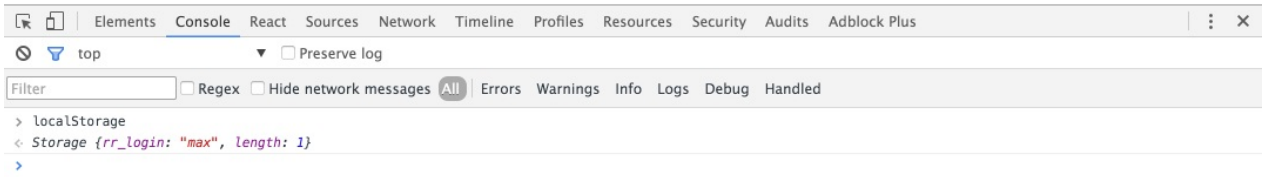
src/routes.js

```
...
import Login from './components/Login'
...
<Route path='/login' component={Login} />
...
```

src/containers/App/index.js

```
...
<li><NavLink to='/login'>Войти</NavLink></li>
...
```

Напоминаю: несмотря на то, что ссылка появилась в шапке (hot-reload работает для компонента `<App />`), страницу все равно нужно перезагрузить (так как hot-reload не работает для обновления списка роутов).



Если ввести новый "логин" - старый перетирается. Данное допущение сейчас не мешает нам проверить работоспособность, а наоборот облегчает проверку.

Нам необходимо закрыть доступ "не админам", для этого потребуется разобрать "события", которые возникают в процессе изменений URL'а.

---

onEnter, onLeave

Предлагаю добавить слово "хук" в словарь. По-моему, хук, не что иное как "действие на событие".

Итак, есть возможность использовать хуки на события onLeave и onEnter. По названию понятно: **onLeave** возникает, когда "роут покинут", а **onEnter** - в момент "захожу на роут".

Представьте адрес:

```
react-site.com/profile/photos/
```

его "роутер-реализацию": (не стоит так говорить в приличном месте)

```
/ + profile + photos
```

и его реализацию компонентами:

`<App /> + <Profile /> + <Photos />`

Представьте, что вы будучи на странице с фото, кликнули на ссылку для перехода на главную страницу. Произойдет:

- *onLeave* на `/profile/photos`
- *onLeave* на `/profile`
- *onEnter* на `/`

И обратная ситуация: вы находитесь на главной, и решили перейти в раздел фото:

- *onLeave* на `/`
- *onEnter* на `/profile`
- *onEnter* на `/profile/photos`

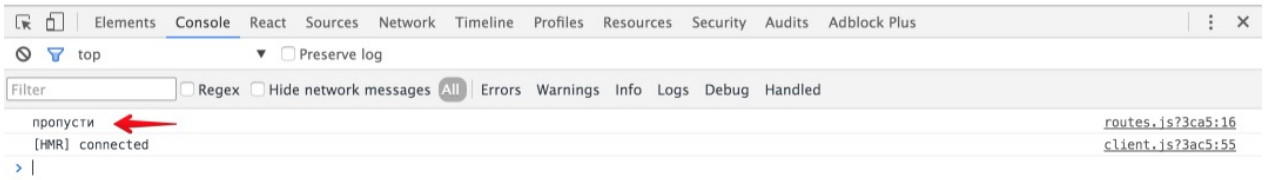
Вернемся к "хукам".

Добавьте в *routes.js* функцию *checkLogin* и непосредственно сам хук.

src/routes.js

```
...
function checkLogin() {
  const login = window.localStorage.getItem('rr_login')
  if (login === 'admin') {
    console.log('пропусти')
  }
}
...
<Route path='/admin' component={Admin} onEnter={checkLogin}/>
...
```

Теперь если вы залогинетесь как **admin** - в консоли браузера вас "пропустят". Причем, заметьте, хук сработает при вводе url напрямую в строке ввода адреса + enter, либо если вы кликнете по ссылке. Так же, если вы уже находитесь в разделе admin и попробуете кликнуть на ссылку "Админка" - *onEnter* не произойдет, а следовательно и хук не сработает. Кажется, разработчики react-router'а постарались на славу.



На всякий случай, напоминаю, что очистить localStorage можно командой `localStorage.clear()`

У хука есть полезные аргументы - **nextState**, **replace**, **callback**. Нам понадобится replace. На пару слов подробнее можно прочитать в офф.документации.

Перепишем функцию *checkLogin*

src/routes.js

```
...
function checkLogin(nextState, replace) {
  const login = window.localStorage.getItem('rr_login')
  if (login !== 'admin') {
    replace('/')
  }
}
...
```

Попробуйте сейчас залогиниться под другим именем: вы не сможете войти на страницу /admin, независимо от того введете ли вы адрес и нажмете enter, или кликните по ссылке.

Страницу API Reference настоятельно рекомендую добавить в закладки.

Использование static method в качестве хука на onEnter

Наш код работает, но функция *checkLogin*, как будто мешается в файле с роутами. Может быть вынести ее в отдельный файл? А вы знаете, есть еще одно интересное решение: использовать static метод класса Admin.

src/components/Admin/index.js

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

export default class Admin extends Component {
  static onEnter(nextState, replace) {
    const login = window.localStorage.getItem('rr_login')
    if (login !== 'admin') {
      replace('/')
    }
  }
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Раздел /admin</div>
      </div>
    )
  }
}
```

Исправим routes.js (привожу полный листинг)

src/routes.js

```
import React from 'react'
import { Route, IndexRoute } from 'react-router'

import App from './containers/App'
import Admin from './components/Admin'
import List from './components/List'
import Genre from './components/Genre'
import Release from './components/Release'
import Home from './components/Home'
import Login from './components/Login'
import NotFound from './components/NotFound'

export const routes = (
  <div>
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      {/* в качестве хука на onEnter - статический метод класса Admin */}
      <Route path='/admin' component={Admin} onEnter={Admin.onEnter}/>
      <Route path='/genre/:genre' component={Genre}>
        <Route path='/genre/:genre/:release' component={Release} />
      </Route>
      <Route path='/list' component={List} />
      <Route path='/login' component={Login} />
    </Route>
    <Route path='*' component={NotFound} />
  </div>
)
```

Все, теперь с чистой совестью можете приступать к задачке на повторение.

Задача: если пользователь ввел admin - после нажатия кнопки "Войти" - направить его на /admin, иначе на /

Подсказка **#1**: Если данная задача вызвала у вас трудность, прочитайте еще раз предыдущую главу.

Решение:

src/components/Login/index.js

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

export default class Login extends Component {
  constructor() {
    super()
    this.handleSubmit = this.handleSubmit.bind(this)
  }
  handleSubmit(e) {
    e.preventDefault()
    const login = e.target.elements[0].value
    window.localStorage.setItem('rr_login', login)

    if (login === 'admin') {
      this.context.router.push('/admin')
    } else {
      this.context.router.push('/')
    }
  }
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Пожалуйста, введите логин:</div>
        <form className='col-md-4' onSubmit={this.handleSubmit}>
          <input type='text' placeholder='login'/>
          <button type='submit'>Войти</button>
        </form>
      </div>
    )
  }
}

Login.contextTypes = {
  router: PropTypes.object.isRequired
}
```

Итого: мы познакомились с возможностью "вклиниваться" в процесс роутинга. Разобрали рабочую ситуацию: как ограничить доступ юзеру в раздел администратора. Закрепили знания по программной навигации.

Исходный код на данный момент.

Подтверждение перехода

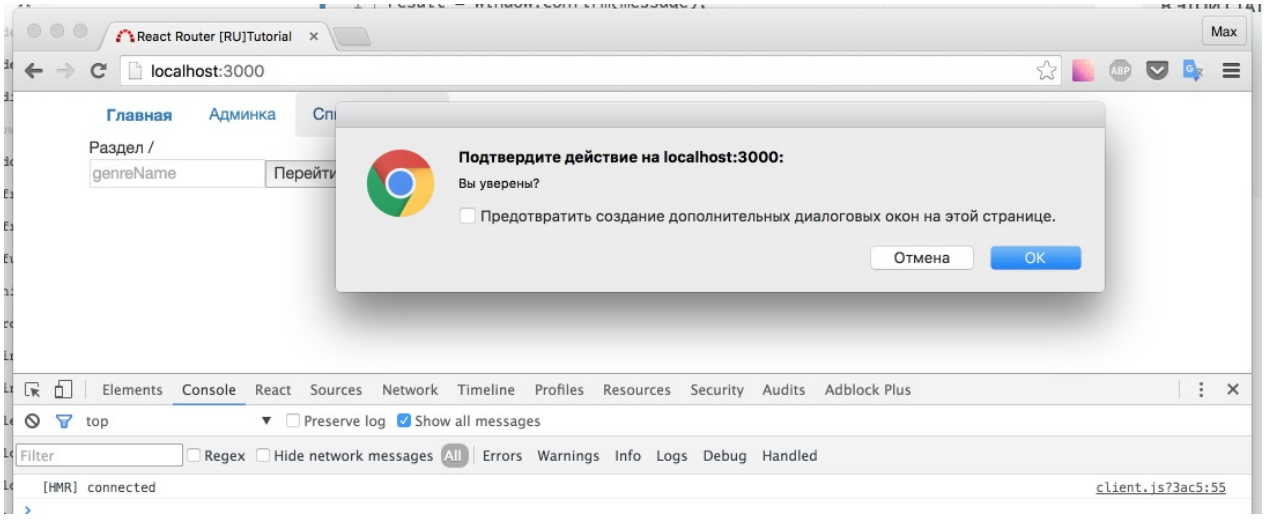
Страница *Home/index.js* - уже зарекомендовала себя в качестве страницы для "выдуманных примеров". Сделаем еще один.

Представьте ситуацию, что вам нужно спросить у пользователя подтверждение о переходе. Самый распространенный вариант - редактирование каких-то данных -> клик на ссылку -> "вы уверены что хотите перейти, все изменения будут утеряны" -> да/нет.

В нашем, упрощенном примере, мы просто покажем confirm окно, на любой переход с главной страницы.

src/components/Home/index.js

```
...
componentDidMount() {
    this.context.router.setRouteLeaveHook(this.props.route, this.routerWillLeave)
}
routerWillLeave() {
    let answer = window.confirm('Вы уверены?')
    if (!answer) return false
}
...
```



Так же, можно использовать более краткую запись в функции *routerWillLeave*:

```
routerWillLeave() {
    return 'Вы уверены?'
}
```

Ссылка на документацию (EN)

Я надеюсь, что всем понятно: вместо стандартного confirm-окна, мы можем показать свои формочки и прочее... Главное, что мы должны **вернуть false**, если не хотим перехода.

Обратите внимание: если вы попробуете, находясь на главной странице, вписать адрес вручную и нажать enter - переход произойдет без вопросов.

Итого: научились предотвращать нежелательный переход по клику.

Исходный код на данный момент.

Итого по роутингу

Если вы не планируете использовать redux - на этом разделе для вас заканчивается курс по роутингу.

Подведем итоги. Мы научились:

- изменять URL-адрес и отрисовывать нужные компоненты в соответствии с этими изменениями
- использовать основные "плюшки" react-router'а: вложения (*nesting*) и параметры в адресе (*:param*)
- ограничивать доступ к странице (на примере обработки события *onEnter*)
- предотвращать нежелательный уход со страницы
- подкрашивать активные ссылки (заодно рассмотрели мощный прием - обертка над компонентом (*wrap component*))
- программно изменять роут

В качестве бонуса:

Необязательный параметр в адресе

Перевод кода из документации:

```
<Route path="/hello/:name">         // совпадет с /hello/michael and /hello/ryan
<Route path="/hello(/:name)">       // совпадет с /hello, /hello/michael, and /hello/r
yan
<Route path="/files/*.*">           // совпадет с /files/hello.jpg and /files/hello.ht
ml
<Route path="/**/*.jpg">            // совпадет с /files/hello.jpg and /files/path/to/
file.jpg
```

Кроме данного бонуса, в официальных гайдах (EN) еще много чего интересного, например, если вы используете *require.ensure*, вам будет полезен материал - Dynamic Routing (EN)

Так же рекомендую изучить примеры.

Для тех, кто хочет идти дальше и использовать redux + react-router: давайте приберемся и вперед.

Приборка заключается в изменение компонента `<Home />`

src/components/Home/index.js

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

export default class Home extends Component {
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Добро пожаловать на курс по redux + react-router</d
iv>
      </div>
    )
  }
}
```

P.S. приятная особенность: вы уже прошли не половину учебника, а процентов восемьдесят пять. Финал близко.

Подключаем redux

Если вы не знакомы с redux, вам поможет этот туториал

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

(вы всегда можете проверить версии в *package.json*, если что-то вдруг перестало работать. "Мир реакт" активно развивается)

Нам необходимо подговиться для решения следующей задачи:

Пользователь кликает на "Админка".

- Если пользователь залогинен - пропускаем его на страницу
- Если пользователь не залогинен - отправлям его на страницу логина

При чем здесь *redux*? Разве такую задачу мы уже не решили с помощью хука на *onEnter?*

Это отличный вопрос! Делайте так, как необходимо для приложения, а не просто тащите в проект все модные технологии. Тем не менее, у нашей задачи есть свои цели:

- во время логина мы сделаем задержку (как будто ждем ответ от сервера), и лишь когда пришел ответ - перенаправим браузер пользователя на /admin
- роутинг будет выполняться посредством **store.dispatch**

В итоге, решив данную задачу, у вас будет четкое понимание как сделать любой подобный запрос с редиректом, ведь по факту все сводится к одному и тому же: когда получен ответ - сделай редирект.

(*код ниже выполнять не обязательно, так как можно взять исходный код в конце раздела.*)

Для начала работы с нашим приложением, необходимо внести множество изменений. Для тех, кто знаком с **redux** - ничего нового. Тезисно и частично с кодом привожу здесь, так как, возможно вы читаете в формате книги вдали от ПК и хотите повторить какие-то моменты связанные с подключением redux в проект:

- Обернем `<Router />` в `<Provider />`

src/index.js

```
...
import { Provider } from 'react-redux'
import configureStore from './store/configureStore'
...

const store = configureStore()

render(
  <Provider store={store}>
    <Router history={browserHistory} routes={routes} />
  </Provider>,
  document.getElementById('root')
)
```

- создаем *константы* для пользователя

src/constants/User.js

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

- создаем *actions* для пользователя

src/actions/UserActions.js

```
/* eslint-disable no-unused-vars */

import {
  LOGIN_REQUEST,
  LOGIN_FAIL,
  LOGIN_SUCCESS,
  LOGOUT_SUCCESS
} from '../constants/User'

export function login(payload) {
  // TODO
  return {
    type: LOGIN_REQUEST
  }
}

export function logout() {
  return {
    type: LOGOUT_SUCCESS
  }
}

/* eslint-enable no-unused-vars */
```

- создаем *reducer* для пользователя

src/reducers/user.js

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

const initialState = JSON.parse(window.localStorage.getItem('rr_user')) || {}

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

  switch (action.type) {

    case LOGIN_REQUEST:
      // TODO
      return {}

    case LOGIN_SUCCESS:
      // TODO
      return {}

    case LOGIN_FAIL:
      // TODO
      return {}

    case LOGOUT_SUCCESS:
      // TODO
      return {}

    default:
      return state
    }
}
```

- Сразу создадим файл с главным редьюсером, где в будущем сможем "скомбинировать" больше редьюсеров (на данный момент у нас только один)

src/reducers/index.js

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

export const rootReducer = combineReducers({
  user,
  popup
})
```

- конфигурируем *store*

- добавляем несколько *middleware* (logger - для логов и thunk - для асинхронных запросов)

src/sotre/configureStore

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

export default function configureStore() {
  const store = compose(
    applyMiddleware(thunkMiddleware),
    applyMiddleware(createLogger())
  )(createStore)(rootReducer)

  if (module.hot) {
    // Enable Webpack hot module replacement for reducers
    module.hot.accept('../reducers', () => {
      const nextRootReducer = require('../reducers').rootReducer
      store.replaceReducer(nextRootReducer)
    });
  }

  return store
}
```

- превратим "тупой компонет" `<Login />` в "умный компонент" `<LoginPage />` (то есть подключим его (*connect*))

src/containers/LoginPage/index.js

```
import React, { Component } from 'react'
import { bindActionCreators } from 'redux'
import { connect } from 'react-redux'
import * as UserActions from '../../actions/UserActions'

export class LoginPage extends Component {
  handleSubmit(e) {
    e.preventDefault()
    this.props.actions.login({name: e.target.elements[0].value})
  }
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>
          <form className='form-inline' onSubmit={::this.handleSubmit}>
            <input className='form-control' type='text' placeholder='login'/>{' '}
            <button className='btn btn-primary' type='submit'>Войти</button>
          </form>
        </div>
      </div>
    )
  }
}

function mapStateToProps() {
  return {}
}

function mapDispatchToProps(dispatch) {
  return {
    actions: bindActionCreators(UserActions, dispatch)
  }
}

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

Обратите внимание, что мы стали передавать объект user

```
{
    name: логин
}
```

До этого момента, мы передавали строку с именем. Так же немного изменилось оформление компонента.

В *routes.js* изменили подключение `<Login />` на `<LoginPage />`

В будущем вы сможете добавить юзерам роль в проекте и так далее.

На всякий случай, напоминаю, что:

```
::this.handleSubmit
```

Эквивалентно

```
this.handleSubmit.bind(this)
```

Итого: мы подключили redux, создав необходимый каркас для дальнейшей деятельности.

Исходный код на текущий момент. Не забудьте скачать новые зависимости:

```
npm install
```

и очистить *localStorage*

```
localStorage.clear() //выполняется из консоли хрома, например
```

store.dispatch редирект

Данная глава неспроста называется store.dispatch редирект. Пожалуй, факт, что мы должны выполнить редирект с помощью **store.dispatch** - является основополагающим в этом подходе.

Чего мы добьемся в таком случае?

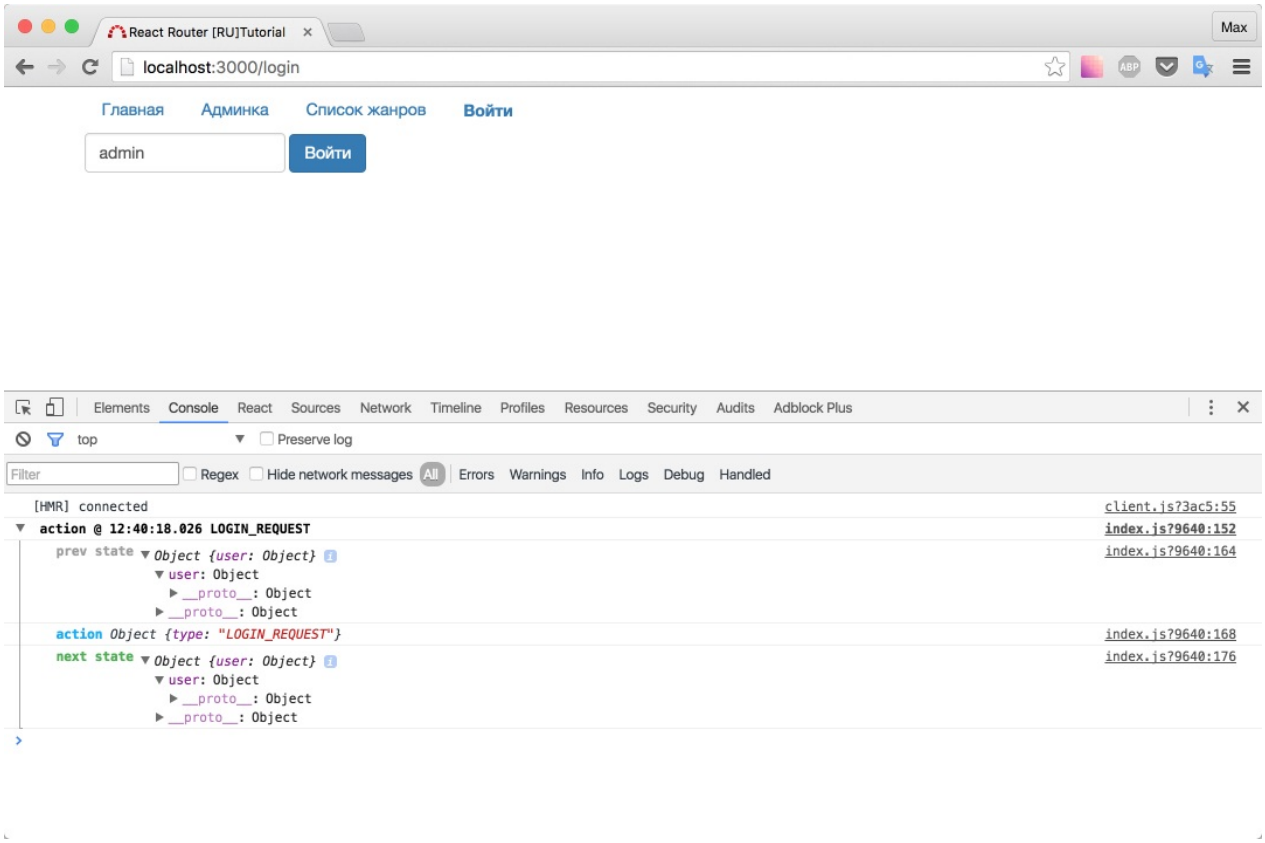
Мы не нарушим однонаправленный (и самое важное, *четко контролируемый нами*) поток данных в приложении. По сути, весь роутинг - это действия (*actions*).

Большинство библиотек (redux-router, react-router-redux) так и поступают. В них, каждое действие можно увидеть в логах.

Почему здесь, я не хочу разбирать эти отличные библиотеки? Потому что, я хочу показать вам как сделать редирект "вручную", чтобы вы четко понимали как это работает. Да, мы потеряем связь с *redux-devtools* (в данном курсе не используется), да мы не будем "синхронизировать" данные роутера в *store*... Но так ли это важно и необходимо?

Давайте просто будем использовать react-router напрямую. А в лог писать, только те действия с роутингом, которые нам действительно необходимы.

Напоминаю, что предыдущий исходный код, дал следующий результат: если мы пытаемся залогиниться - логируется LOGIN_REQUEST и ничего не происходит.



Для начала, оформим все необходимое для имитации логина. В "заглушке" будем, помимо свойства name, добавлять свойство *isAuthenticated*

src/actions/UserActions.js

```
...
export function login(payload) {
  return (dispatch) => {

    dispatch({
      type: LOGIN_REQUEST
    })

    setTimeout(() => {
      dispatch({
        type: LOGIN_SUCCESS,
        payload: {
          name: payload.name,
          isAuthenticated: true
        }
      })
    },2000)
  }
}
...
```

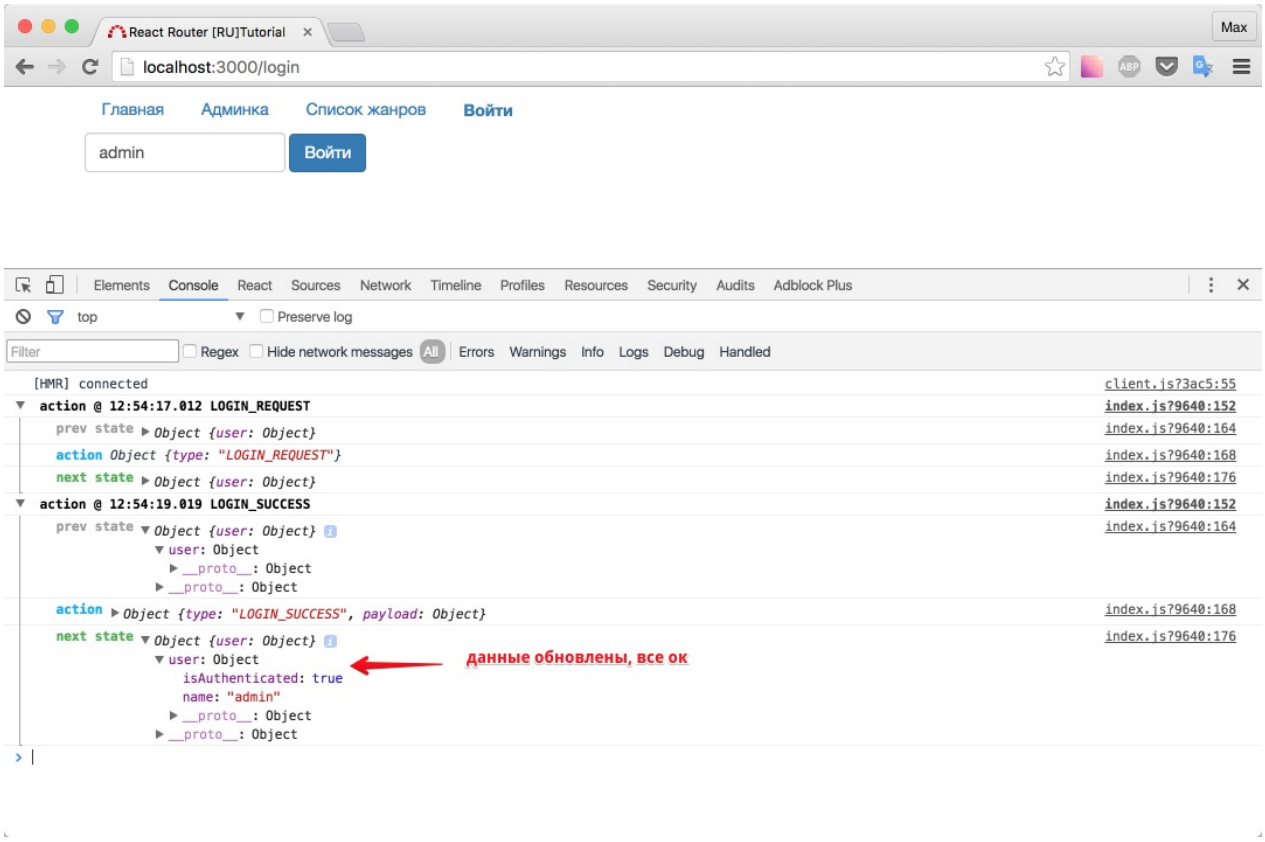
В редьюсере, соответственно, будем корректно обрабатывать измененный *action*:

src/reducers

```
...
case LOGIN_SUCCESS:
      return {...state, name: action.payload.name, isAuthenticated: action.payload.isA
uthenticated}
...
```

На всякий случай: ... в начале и в конце = "другой код выше / ниже", а ... в строке return - spread operator

Проверим:



LocalStorage в данный момент никак не используем.

Немного порассуждаем: если бы у нас была возможность "диспатчить" редирект, где бы мы это сделали? Очевидно, что после LOGIN_SUCCESS в таймауте.

Проблема в том, что мы не можем вызывать редирект, с помощью store.dispatch

Вопрос: как это исправить?

Ответ: написать middleware.

Задача: требуется написать middleware, который в случае: action.type = РОУТИНГ, выполнял бы переход (с помощью browserHistory, разумеется).

Дополнение **#1**: Было бы здорово, если бы мы имели возможность указывать push или replaceState метод.

Решение (*псевдо-код*):

```
import browserHistory
import КОНСТАНТА_РОУТИНГА

export const redirect = store => next => action => {
  if (action.type === КОНСТАНТА_РОУТИНГА) {
    browserHistory[МЕТОД](куда_перенаправить)
  }

  return next(action)
}
```

Предлагаю решить самостоятельно, а после сверится с кодом ниже.

Решение: создание middleware для роутинга

src/constants/Routing.js

```
export const ROUTING = 'ROUTING'
```

src/middlewares/redirect.js

```
import { browserHistory } from 'react-router'

import {
  ROUTING
} from '../constants/Routing'

export const redirect = store => next => action => { //eslint-disable-line no-unused-v
ars
  if (action.type === ROUTING) {
    browserHistory[action.payload.method](action.payload.nextUrl)
  }

  return next(action)
}
```

События связанные с роутингом мы не будем обрабатывать редьюсером.

Обновим *actionCreator* (функцию-создатель действия):

src/actions/UserActions.js

```
...
import {
  ROUTING
} from '../constants/Routing'

export function login(payload) {
  return (dispatch) => {

    dispatch({
      type: LOGIN_REQUEST
    })

    setTimeout(() => {
      dispatch({
        type: LOGIN_SUCCESS,
        payload: {
          name: payload.name,
          isAuthenticated: true
        }
      })

      dispatch({
        type: ROUTING,
        payload: {
          method: 'push', //или, например, replace
          nextUrl: '/admin'
        }
      })
    },2000)
  }
}
...
```

Вопрос: где мы должны добавить middleware в цепочку других middleware'ов ?

Ответ: там, где настраивается объект *store*

src/store/configureStore.js

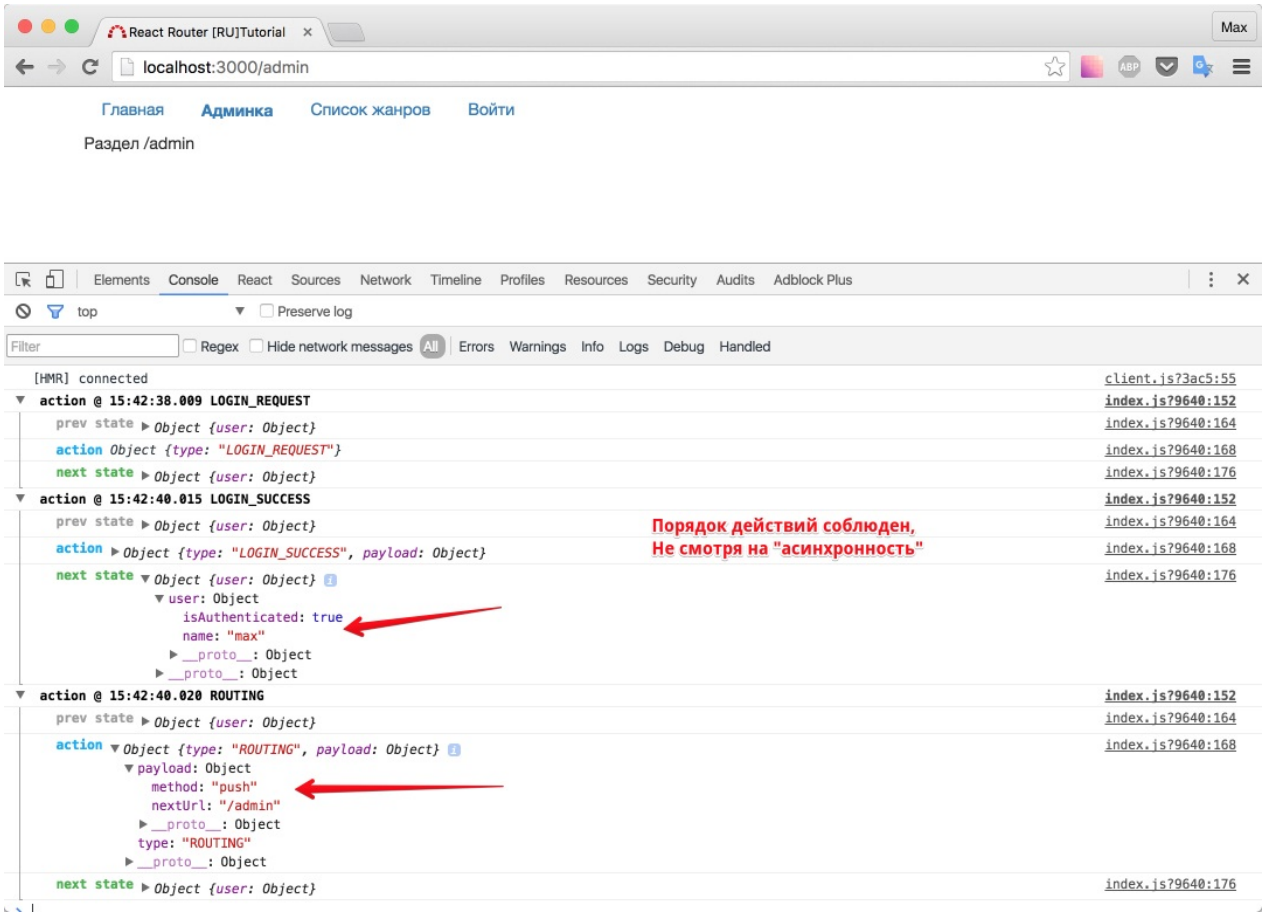
```
...
import { redirect } from '../middlewares/redirect'

export default function configureStore() {
  const store = compose(
    applyMiddleware(thunkMiddleware),
    applyMiddleware(createLogger()),
    applyMiddleware(redirect) // добавили редирект middleware
  )(createStore)(rootReducer)
...
```

Очистим метод *onEnter* у компонента `<Admin />`:

src/components/Admin/index.js

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

export default class Admin extends Component {
  static onEnter() {
    // nothing
  }
  render() {
    return (
      <div className='row'>
        <div className='col-md-12'>Раздел /admin</div>
      </div>
    )
  }
}
```

Откройте в браузере `http://localhost:3000/login` и попробуйте залогиниться (введите любое имя):



Теперь нажмите "назад" - вы должны оказаться на странице `/login`, так как мы использовали метод **push**. Данный метод добавляет страницу в историю браузера. **replace** же, не добавляет страницу в историю браузера, но тем не менее url меняет. Скоро мы поработаем с ним.

Не забывайте про возможность пользователя нажимать "назад/вперед".

Итог: мы научились делать редирект посредством **store.dispatch**, но сломали *onEnter hook*, теперь страница `/admin` доступна всегда. Исправим это в следующем уроке.

Исходный код на данный момент.

"Закрытый" компонент

Для урока, я использовал этот пример.

В конце прошлого раздела у нас осталась проблема - все имеют доступ к */admin*. Мы уже решали ее в первой части с помощью хука на *onEnter*. Но для связки redux + react-router мне нравится подход с "закрытым" компонентом.

Помните, мы делали `<NavLink />` обертку? Как я уже писал - это очень мощный прием. Сейчас мы воспользуемся им [приемом] еще раз.

AuthenticatedComponent

Начнем с размышлений и псевдокода:

Закрытый_компонент.js

```
...
render() {
    this.props.юзер_может_видеть_это
    ? <Компонент_который_закрыт + его props />
    : ничего
}
...
```

Что ж, выглядит "заумно"? На деле - воспользуемся старой доброй возможностью *прокидывать аргументы в функцию*. А так же, сразу сделаем нашу обертку "приконекченной" (*connect*).

src/containers/AuthenticatedComponent/index.js

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

export default function requireAuthentication(Component) {

  class AuthenticatedComponent extends Component {
    render() {
      return (
        <div>
          {this.props.user.isAuthenticated === true
            ? <Component {...this.props} />
            : null
          }
        </div>
      )
    }
  }

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

  return connect(mapStateToProps)(AuthenticatedComponent)
}
```

В *render* методе мы проверяем - есть ли у *user* право видеть данный компонент, и если да - показываем. Если нет - null.

Обновим роуты:

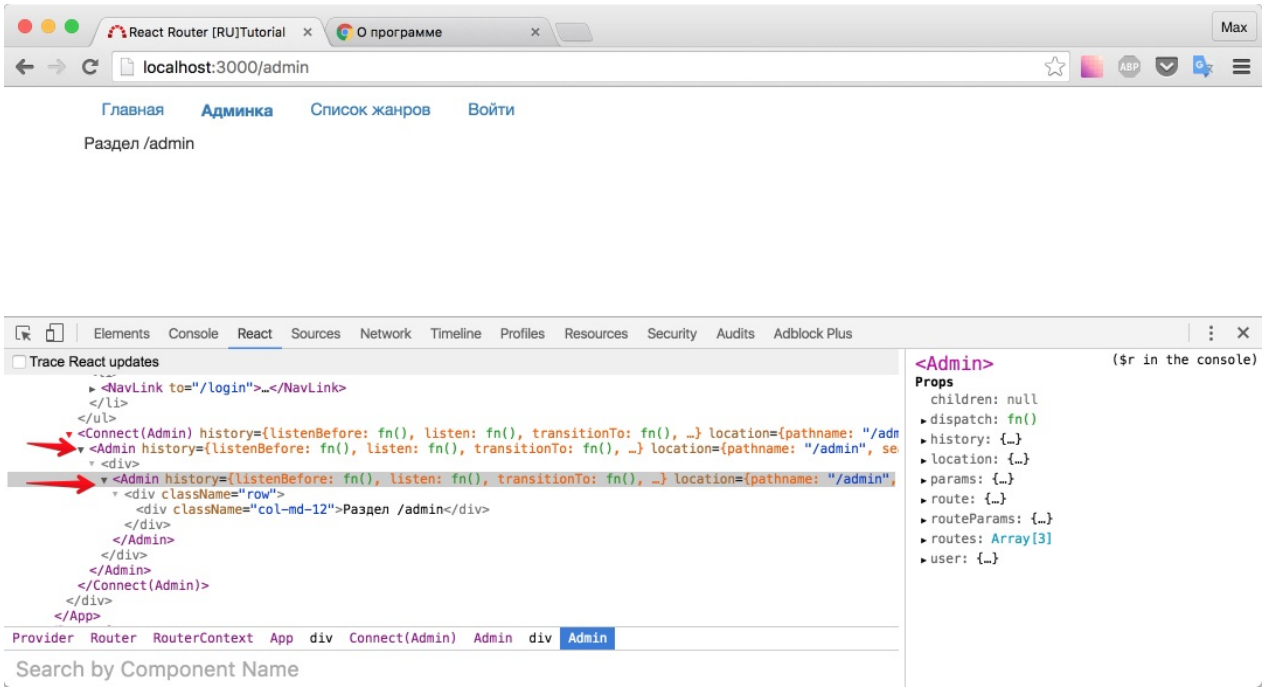
src/routes.js

```
...
import requireAuthentication from './containers/AuthenticatedComponent'

...
    <Route path='/' component={App}>
      <IndexRoute component={Home} />
      <Route path='/admin' component={requireAuthentication(Admin)} /> // Admin -> <Co
mponent {...this.props} />
...
```

Опять же, почувствуйте все удобства "оборачивания": нам вообще не нужно ничего делать с компонентом `<Admin />`.

Перекур. Интересный момент. На дворе 22.04.2016 и мой хром версии - 50.0.2661.75 (64-bit) + React dev tools последней версии.

Давайте проверим как работает наш подход. Зайдите на `/login`, залогиньтесь и после редиректа посмотрите в инспектор:



Выглядит не так, как мы ожидали. Вероятно, должно было быть так:

```
connect(AuthenticatedComponent)
>> AuthenticatedComponent
>> >> Admin
```

Но у нас:

```
connect(Admin)
>> Admin
>> >> Admin
```

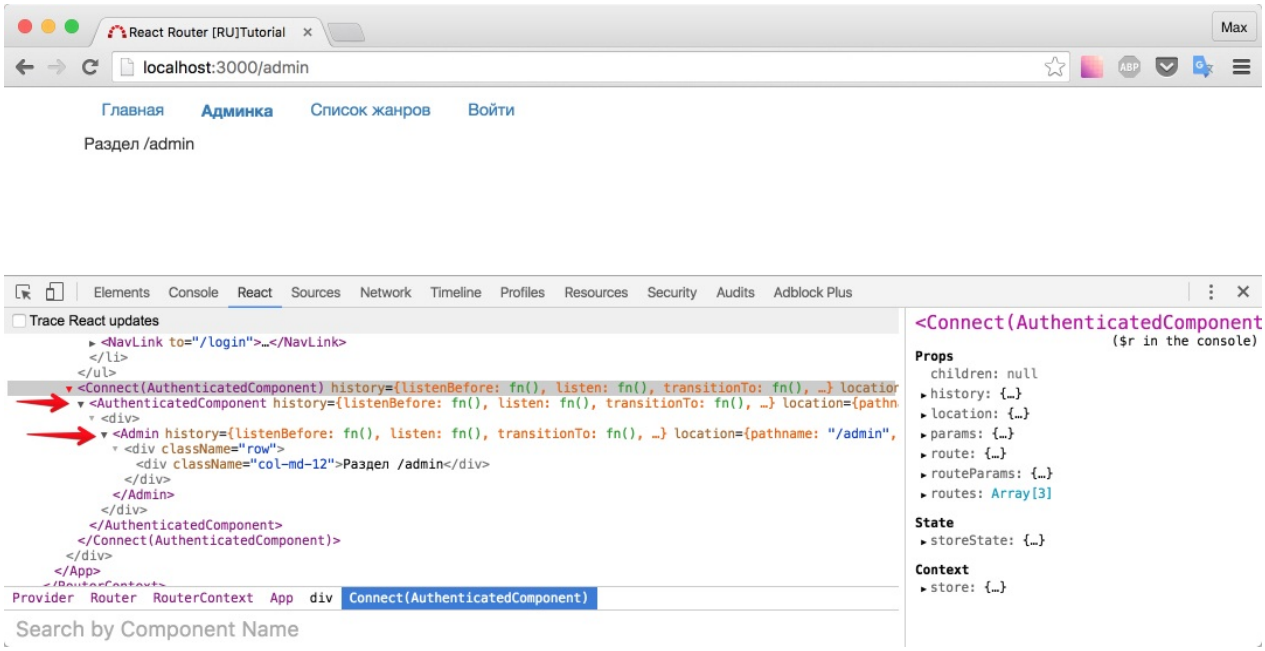
До текущего момента мы сделали очень много работы, чтобы все работало так как мы ожидаем, не так ли? Если мы пишем console.log - у нас возникает один console.log (камень в огород Angular), если мы хотим чтобы "роутинг" был частью потока данных - мы и это делаем в соответствии с документацией. Тогда в чем дело?

Посмотрите внимательно на код **AuthenticatedComponent**. У нас и аргумент функции *Component* и класс наследует *Component*. Вероятно, проблема здесь. Давайте просто импортируем чуть-чуть иначе, а название аргумента сохраним.

src/containers/AuthenticatedComponent/index.js

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

export default function requireAuthentication(Component) {

  class AuthenticatedComponent extends React.Component {
    render() {
      return (
        <div>
          {this.props.user.isAuthenticated === true
            ? <Component {...this.props} />
            : null
          }
        </div>
      )
    }
  }

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

  return connect(mapStateToProps)(AuthenticatedComponent)
}
```

Проверим:



Ожидаемо и предсказуемо. Перекур завершен.

P.S. было "extends Component", стало "extends React.Component" (конечно, строка импорта тоже изменилась)

Редирект на страницу логина, если пользователь не аутентифицирован

Мы почти закончили. На данный момент:

- после логина - редирект
- при попытке неавторизованному пользователю зайти на `/admin` - пусто (вместо *null* можете выводить "403 - доступ запрещен", либо можете создать компонент `<NoAccess />` ...)

Тем не менее, по условию задачи: если пользователь неавторизован - перенаправь его на `/login`.

Для этого воспользуемся lifecycle-методами (методами жизненого цикла): *componentWillMount* и *componentWillReceiveProps*

src/containers/AuthenticatedComponent/index.js

```
import React from 'react'
import { connect } from 'react-redux'
import { ROUTING } from '../../constants/Routing'

export function requireAuthentication(Component) {

  class AuthenticatedComponent extends React.Component {
    componentWillMount() {
      this.checkAuth(this.props.user)
    }
    componentWillReceiveProps(nextProps) {
      this.checkAuth(nextProps.user)
    }
    checkAuth(user) {
      if (!user.isAuthenticated) {
        this.props.dispatch({
          type: ROUTING,
          payload: {
            method: 'replace',
            nextUrl: '/login'
          }
        })
      }
    }
    render() {
      return (
        <div>
          {this.props.user.isAuthenticated === true
            ? <Component {...this.props} />
            : null
          }
        </div>
      )
    }
  }

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

  return connect(mapStateToProps)(AuthenticatedComponent)
}
```

Поскольку, *AuthenticatedComponent* присоединен, у нас есть возможность использовать `this.props.dispatch`.

Для истории браузера используется метод *replace*, так как нам это подходит лучше (после проверки, замените на push и походите "Назад/Вперед").

Измените на *replace* и в *UserActions*.

src/actions/UserActions.js

```
...
dispatch({
    type: ROUTING,
    payload: {
      method: 'replace',
      nextUrl: '/admin'
    }
})
...
```

Проверьте в браузере.

Итого: мы рассмотрели еще один пример "оборачивания" компонента, тем самым получив необходимый для решения задачи редирект.

P.S. вы можете использовать "оборачивание", для скрытия каких-то элементов на странице, а не только всей страницы целиком.

Исходный код на данный момент.

Итого по react-router + redux

Спасибо, что вы дочитали до конца. Подведем небольшой итог, того, что не сделано:

- нет разделения прав доступа
- страницы выглядят ужасно (ни картинок, ни стилей толком)
- надпись "войти" не меняется на "выйти" и подобное...

На правах автора учебника, я могу сказать: "а это на самостоятельное задание".

Так как сама "техника" описана достаточно подробно, я думаю у вас все получится. Возможно, если будет время я добавлю сюда решение.

По ролям, хочу натолкнуть ваши "гугления" на использование побитовых операторов.

Если у вас есть вопросы или пожелания по учебнику - вы всегда можете задать их мне в твиттере или на почту.

Данный мини-учебник является третьим в серии "**Без воды**". Ссылки на первые два:

1. React.js для начинающих
2. Туториал по redux на русском