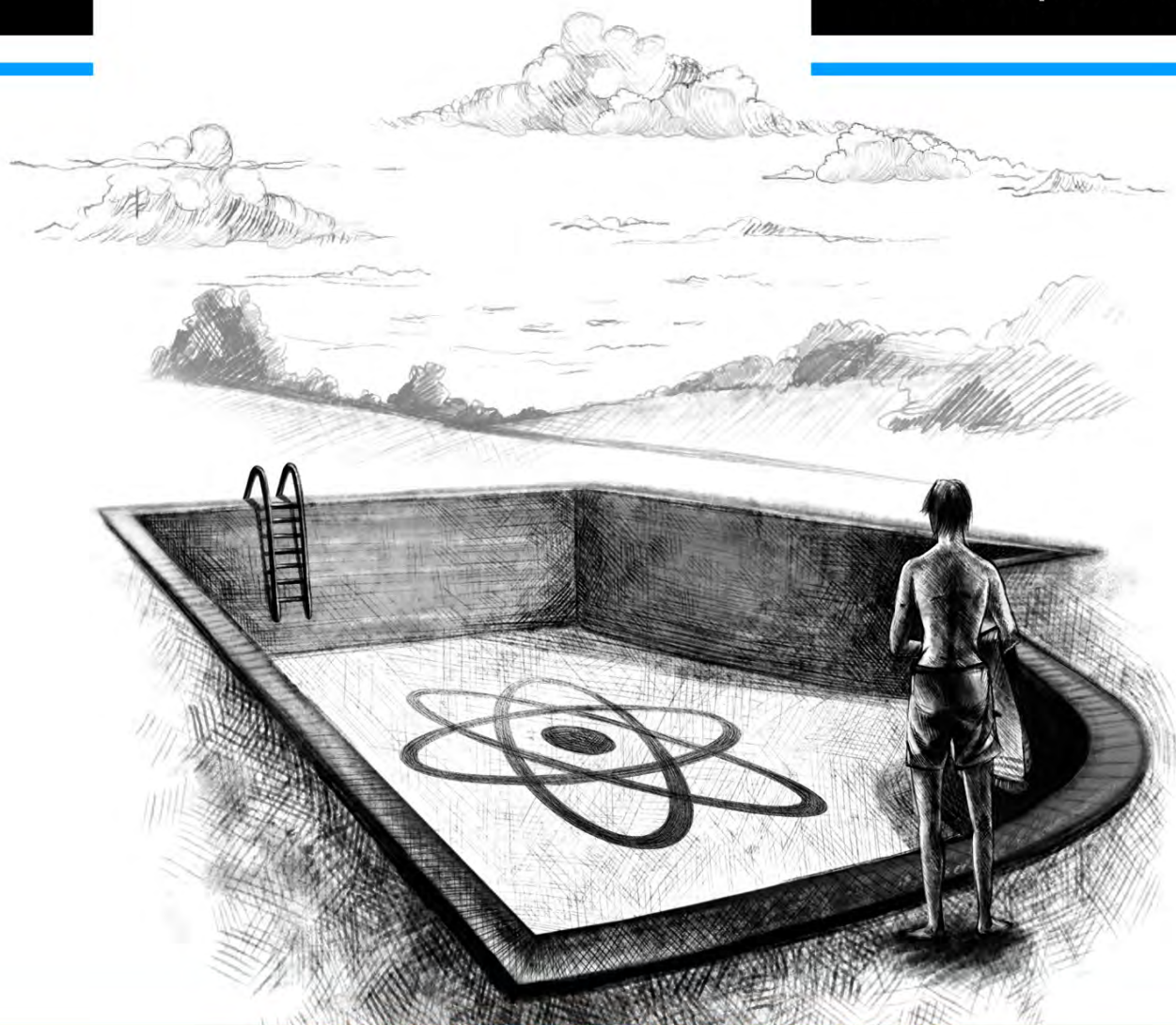


М. ПАЦИАНСКИЙ



REACT.JS ДЛЯ НАЧИНАЮЩИХ

2016



Table of Contents

Вступление	1.1
Подготовка	1.2
Подключаем react	1.3
Создание компонента	1.4
Использование props	1.5
If-else, тернарный оператор	1.6
Порефакторим...	1.7
React.propTypes	1.8
Использование state	1.9
Продвинутое использование	1.9.1
Работа с input	1.9.2
Жизненный цикл компонента	1.10
Работа с формой	1.11
Добавить новость	1.11.1

React.js курс для начинающих

В данном курсе разбираются основы React.js

Результатом курса будет небольшое приложение новостей, в котором можно добавить новость, а так же посмотреть у новости "подробнее".

После прочтения курса, вы научитесь:

1. Создавать компоненты, учитывая *propTypes*
2. Грамотно использовать *props* и *state* компонента
3. Работать с формой
4. Работать с react dev tools
5. Рефакторить и быть лучше ;)

В тексте курса часто встречаются небольшие задачки, а так же приводится их решение.

Для успешного прохождения курса, вам потребуются знания:

1. HTML/CSS
2. Javascript (ну или хотя бы jQuery, если вы понимаете, что \$ всего лишь функция...)

В процессе изучения, нам потребуется локальный сервер. Для этого мы воспользуемся *node.js* и *express*. Знание этих технологий не обязательно, достаточно будет просто "скопировать/вставить", либо воспользоваться своим локальным сервером, например на Apache.

Рекомендую после изучения начального курса посмотреть в сторону Flux-подхода, а именно - Redux.

Полезные ссылки

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

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

Подготовка

React.js это всего лишь библиотека. Не фреймворк.

Как и любая другая библиотека, реакт добавляется на страницу с помощью тэга

```
script .
```

Так как в современном js "рулят" модули и различного рода преобразования/сжимания и так далее - *react* отлично дружит с *webpack*, *babel* и прочими. Для простоты курса, мы будем работать с реактом, как с обычной библиотекой, например, jQuery.

Кстати, библиотека jQuery для работы реакта не нужна. Но нужны некоторые вспомогательные библиотечки, которые предоставляет сам реакт, либо его друзья в лице babel.

Еще раз поясню: *цель данного курса - просто добавить реакт на страницу и "начать писать на нем"*. Пусть все будет в одном не сжатом файле, сейчас важно *взять и сделать*. Модульность, супер-умная перезагрузка страницы без F5 и т.д. - *все потом*.

Первым делом [скачайте react](#).

Далее создайте структуру файлов и папок. Все нескопированные файлы, пока оставьте пустыми.

```
+-- js
|   +-- react
|       +-- react-dom.js (скопируйте из архива из папки build)
|       +-- react.js (скопируйте из архива из папки build)
|   +-- app.js
+-- index.html
+-- package.json
+-- server.js
```

Локальный сервер

Вы можете работать с *localhost* как вам привычно. Либо взять решение, которое рассматривается ниже:

Необходимо создать локальный сервер - `server.js` , чтобы работала вкладка в консоли - *React Developer Tools*, которая поможет нам отлаживать наше React-приложение.

Прежде чем мы это сделаем, создайте файл `package.json` , который является инструкцией/описанием для нашего проекта.

package.json

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

Не забудьте поменять автора ;)

Локальный сервер

Создадим с помощью Node.js* и express локальный сервер.

* если у вас не установлен Node.js, то вам необходимо скачать и установить его с официального сайта (<https://nodejs.org/en/>). Если вы используете Windows, для выполнения команд ниже рекомендую консоль - [ConEmu](#).

Добавим express в наш проект.

```
npm install express --save-dev
```

При использовании флага `--save-dev` , пакет *express* добавится в список зависимостей нашего проекта. Вы можете посмотреть как изменился файл `package.json` , чтобы убедиться в этом.

server.js

```
var express = require('express');
var app = express();

app.set('port', (process.env.PORT || 3000));

app.use('/', express.static(__dirname));

app.listen(app.get('port'), function() {
  console.log('Server started: http://localhost:' + app.get('port') + '/');
});
```

Заполните файл index.html

index.html

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

Запустите наш сервер `node server.js`

Так же, сервер можно запускать с помощью команды `npm start`, это возможно, потому что у нас в файле `package.json` есть секция *scripts*, в которой указана команда *start*. Такой вариант использовать удобно, когда для запуска сервера, вам приходится писать гораздо больше, например:

```
node server.js -option1 -option2 CONST=qweqwe ...
```

Достаточно будет указать полную команду в файле-инструкции, и затем удобно использовать ее с помощью сокращенного варианта. Да, по сути это *alias* (сокращения) для вашего проекта.

Что ж, убедитесь, что у вас в браузере появилась строка "Привет, я #root", и если все хорошо, в следующем разделе мы подключим react и начнем его использовать.

[Исходный код](#) для данного раздела.

Подключаем react

Как уже было сказано, реакт - просто библиотека, которая так же просто подключается. Мы уже скачали *react* и *react-dom*, давайте добавим их на страницу.

Так же, сразу подключим наш *app.js*, в котором будем писать код.

index.html

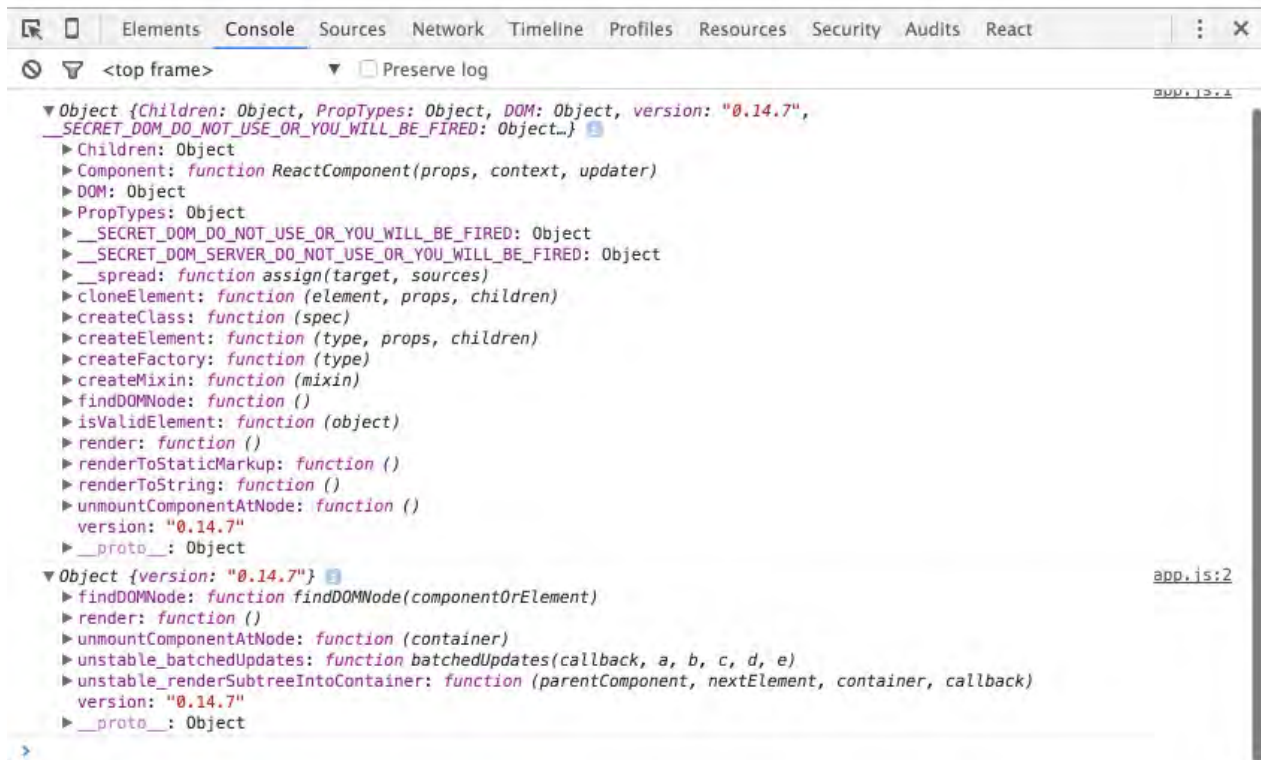
```
<!DOCTYPE html>
<html>
  <head>
    <title>React [RU] Tutorial</title>
  </head>
  <body>
    <div id="root">Привет, я #root</div>

    <script src="js/react/react.js"></script>
    <script src="js/react/react-dom.js"></script>
    <script src="js/app.js"></script>
  </body>
</html>
```

js/app.js

```
console.log(React);
console.log(ReactDOM);
```

Посмотрим в браузер:



Отлично. Предлагаю вывести что-нибудь с помощью react.

Изменим app.js:

js/app.js

```
ReactDOM.render(
  React.createElement('h1', null, 'Привет, Мир!'),
  document.getElementById('root')
);
```

Bay! Мы только что добавили заголовок (`<h1></h1>`) с помощью react.

Мы использовали функцию `ReactDOM.render`, которая принимает первым аргументом - реакт-компонент, а вторым - элемент DOM дерева, куда мы хотим добавить react.

Вообще, зачем мы добавили react на нашу страницу? Наверное, потому что где-то слышали, что он быстрый. Почему же?

React при изменениях DOM-дерева, старается использовать минимально-возможные воздействия. React использует "**виртуальный DOM**" (удаляет/изменяет/добавляет элементы и т.д.) для того, чтобы в реальный DOM за "один присест" добавить все изменения. Как известно, операции с DOM-деревом самые дорогостоящие. Поэтому "интеллектуальный" подход реакта к ним - та самая "киллер-фича".

Данное объяснение считаю очень "общим", но, чтобы не потеряться в теории, этого сейчас вполне достаточно.

Вернемся к нашему коду в файле *app.js*

С таким синтаксисом, я про `React.createElement('h1', null, 'Привет, Мир!')`, , реакт явно не завоевал бы мир, поэтому команда разработчиков решила писать разметку в javascript-коде и назвала такой синтаксис `JSX` .

У такой техники есть ярые противники, но, мне кажется, пользы от использования реакта гораздо больше, чем от "правильного" разделения на *разметку* и *скрипты*.

Исправим наш код. *js/app.js*

```
ReactDOM.render(  
  <h1>Hello, world!</h1>,  
  document.getElementById('root')  
)
```

Поскольку React - javascript библиотека, а мы пишем JSX синтаксис внутри обычного .js файла, нам необходимо как-то научить браузер превращать

```
<h1>Hello, world!</h1>
```

в

```
React.createElement('h1', null, 'Привет, Мир!')
```

С этим отлично справляется **babel**. Так как у нас нет никакого "сборщика" кода, нам необходимо добавить еще один **скрипт** на страницу. Скачайте его, либо подключите через CDN. Я скачаю и положу его в папку *js/react*.

Так же, для нашего скрипта нужно указать атрибут `type='text/babel'`

index.html

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

    <script src="js/react/react.js"></script>
    <script src="js/react/react-dom.js"></script>
    <script src="js/react/browser.min.js"></script>
    <script type="text/babel" src="js/app.js"></script>
  </body>
</html>
```

Итого: мы можем писать привычную нам HTML разметку внутри "реакт-кода".

Важно понимать - никакой магии нет. Babel превращает HTML * разметку в Javascript код.

**Правильнее это называть JSX разметкой, вы скоро увидите небольшие отличия.*

[Исходный код](#) на данный момент.

Создание компонента

В прошлом разделе, я упомянул, что ReactDOM.render принимает react-компонент (далее буду называть просто "компонент") и DOM-элемент, в который мы хотим "примонтировать" наше приложение.

```
<h1>Hello, world!</h1>
```

 - как ни странно, это примитивный компонент.

Пока ничего интересного, но давайте представим такой псевдо-код:

```
var photos = ['images/cat.jpg', 'images/dog.jpg', 'images/owl.jpg']

ReactDOM.render(
  <App>
    <Photos photos=photos />
    <LastNews />
    <Comments />
  </App>,
  document.getElementById('root')
);
```

Что примечательного в псевдо-коде? Он очень хорошо читается, ведь очевидно, что наше приложение (App) отображает: фото (кошка, собака, сова), новости (какие-нибудь) и комментарии (тоже какие-нибудь).

Хочу вас обрадовать. React.js код выглядит практически так же. Он отлично читается, так как деление на компоненты позволяет отлично структурировать код.

Давайте создадим примитивный компонент.

js/App.js

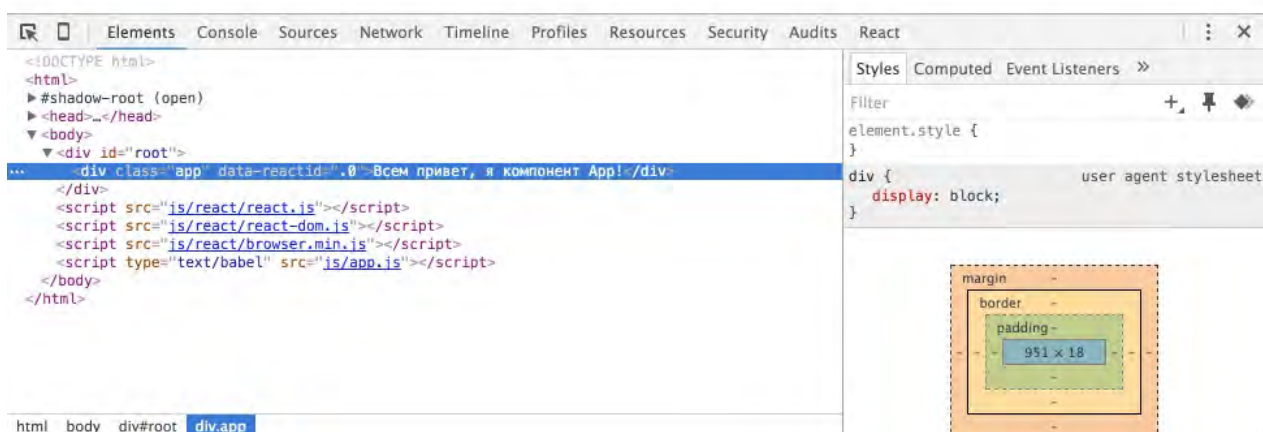

```

var App = React.createClass({
  render: function() {
    return (
      <div className="app">
        Всем привет, я компонент App!
      </div>
    );
  }
});

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

```

Посмотрите в браузер, вы увидите разметку, которую мы указали в методе Render, нашего компонента App.



Конечно, кое-что добавил уже сам реакт, например `data-reactid=".0"`, это нормально.

Слово `class` в javascript является зарезервированным, поэтому внутри JSX синтаксиса необходимо писать **`className`**.

В ReactDOM.render мы передали целиком наш компонент, без свойств.

Запись `<App />` эквивалентна `<App></App>`.

Итого: у нас есть компонент `<App />`

Давайте разовьем идею, и добавим что-нибудь внутрь App, и отобразим это как ни в чем не бывало.

js/App.js

```
var News = React.createClass({
  render: function() {
    return (
      <div className="news">
        К сожалению, новостей нет.
      </div>
    );
  }
});

var App = React.createClass({
  render: function() {
    return (
      <div className="app">
        Всем привет, я компонент App! Я умею отображать новости.
        <News />
      </div>
    );
  }
});

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

Давайте, вновь взглянем на код и поищем примечательные места.

Во-первых - мы никак не изменили код внутри ReactDOM.render. Компонент `<App />`, содержит сейчас в себе другой компонент. Но при этом, это никак не влияет на "рендер" всего нашего приложения.

Во-вторых, как уже было сказано - компонент `<App />` содержит в себе компонент `<News />`. Да-да! Так же, как если бы это был просто дочерний `<div></div>` элемент.

В-третьих, наш компонент `<News />` такой же примитивный, как и App, и содержит всего один (обязательный!) метод `render`.

Задача на понимание происходящего: создайте компонент `<Comments />` и сделайте, чтобы он отображался после новостей. Текст компонента: "Нет новостей - комментировать нечего."

Решение для задачи всегда публикуется ниже по тексту, и обычно содержит сначала подсказки, а потом код всего решения.

Решение:

js/App.js

```
var News = React.createClass({
  render: function() {
    return (
      <div className="news">
        К сожалению, новостей нет.
      </div>
    );
  }
});

var Comments = React.createClass({
  render: function() {
    return (
      <div className="comments">
        Нет новостей - комментировать нечего
      </div>
    );
  }
});

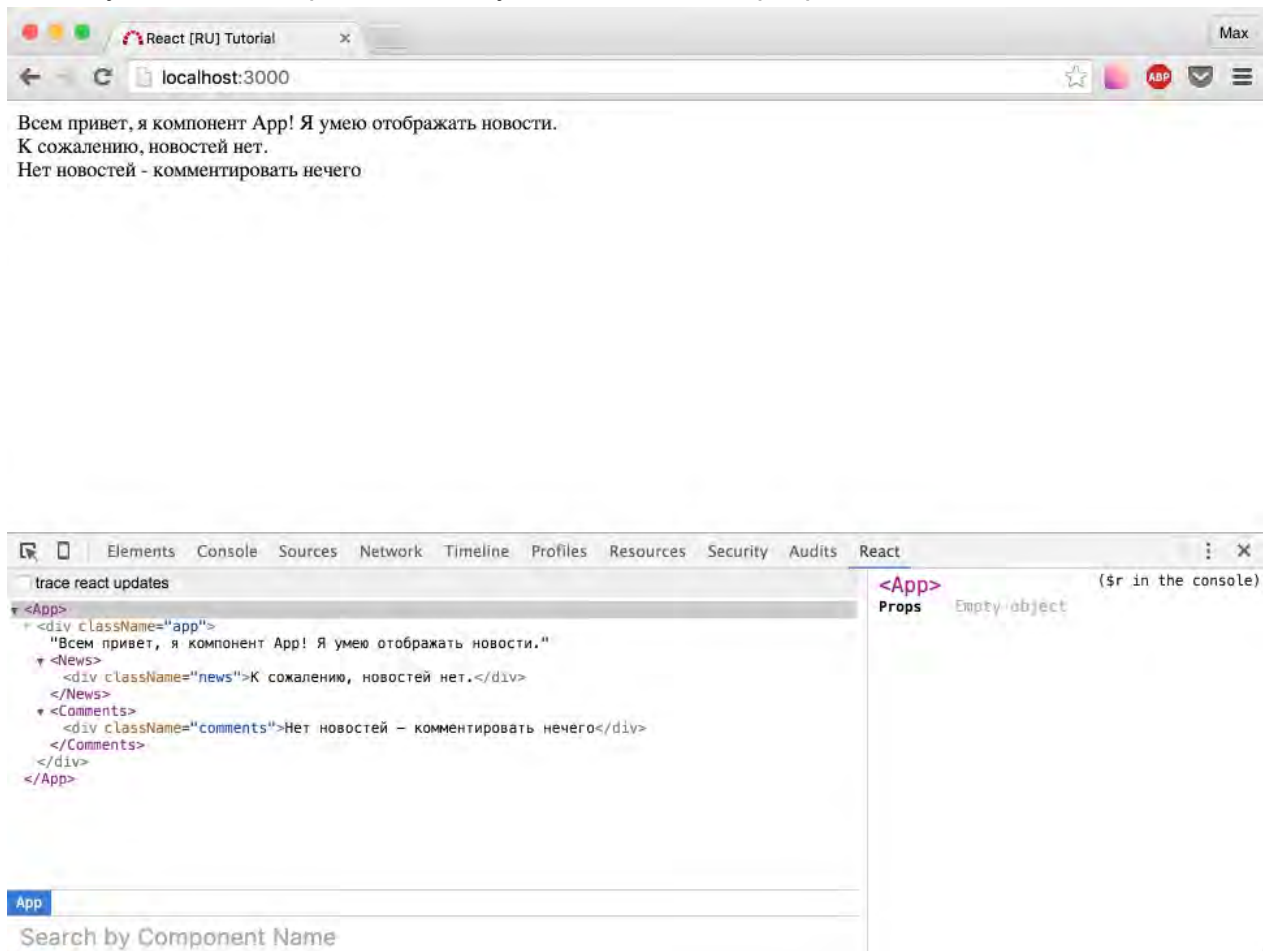
var App = React.createClass({
  render: function() {
    return (
      <div className="app">
        Всем привет, я компонент App! Я умею отображать новости.
        <News />
        <Comments />
      </div>
    );
  }
});

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

Откройте браузер, посмотрите разметку страницы. Все как мы написали + data-react атрибуты.

Ощущается некоторое неудобство, при использовании стандартного способа просмотра в консоли. Предлагаю вам установить react devtools (плагин для [хрома](#), плагин для [мозилы](#)).

После установки, откройте вкладку React в консоли разработчика.



Пытливый читатель уже заметил окошечко "Props". Ок, об этом и поговорим в следующей главе.

[Исходный код](#) на данный момент.

Использование props

У каждого компонента могут быть свойства. Они хранятся в `this.props`, и передаются компоненту как атрибуты.

Общий вид:

```
var value1 = {name: Garry, surname: Potter};  
<MyComponent data={value1} eshe_odno_svoistvo={[1,2,3,4,5]} />
```

В свойство можно передать любой javascript примитив, объект, переменную и даже выражение. Значение свойства должно быть взято в фигурные скобки.

Значения доступны через `this.props.ИМЯ_СВОЙСТВА`

В нашем случае, мы получим:

- `this.props.data` - объект `{name: Garry, surname: Potter}`
- `this.props.eshe_odno_svoistvo` - массив `[1, 2, 3, 4, 5]`

`this.props` используются **только** для чтения!

Давайте создадим несколько новостей для нашего приложения.

Добавьте в начало файла `js/app.js`

```
var my_news = [  
  {  
    author: 'Саша Печкин',  
    text: 'В четверг, четвертого числа...'  
  },  
  {  
    author: 'Просто Вася',  
    text: 'Считаю, что $ должен стоить 35 рублей!'  
  },  
  {  
    author: 'Гость',  
    text: 'Бесплатно. Скачать. Лучший сайт - http://localhost:3000'  
  }  
];
```

И измените строку с подключением компонента News следующим образом:

```

var App = React.createClass({
  render: function() {
    return (
      <div className="app">
        Всем привет, я компонент App! Я умею отображать новости.
        <News data={my_news} /> { /*добавили свойство data */}
        <Comments />
      </div>
    );
  }
});

```

Обратите внимание, комментарии внутри JSX пишутся в фигурных скобках: `{ /* текст комментария */ }`

Так же прошу заметить, JSX это не весь js-код, который содержится в *App.js*, грубо говоря JSX - это HTML-разметка + переменные/выражения. Поэтому в остальных местах можно писать комментарии привычным для вас способом (`//... или /*...*/`)

Откройте в консоли вкладку react (конечно, предварительно обновив страницу).

The screenshot shows a web browser at localhost:3000 displaying the output of the App component: "Всем привет, я компонент App! Я умею отображать новости.", "К сожалению, новостей нет.", and "Нет новостей - комментировать нечего". Below the browser, the React DevTools component inspector is open, showing the component tree and the props for the selected `<News>` component. Red arrows and text annotations highlight key details:

- A red arrow points to the `data` prop in the `<News data={my_news} />` tag in the component tree, with the text: **мы добавили массив новостей в свойства**.
- A red arrow points to the closing tag `</News>` in the component tree, with the text: **комментарий превратился в пустую строку**.
- A red arrow points to the props panel on the right, with the text: **свойства отображаются здесь**.

The props panel for `<News>` shows:

```

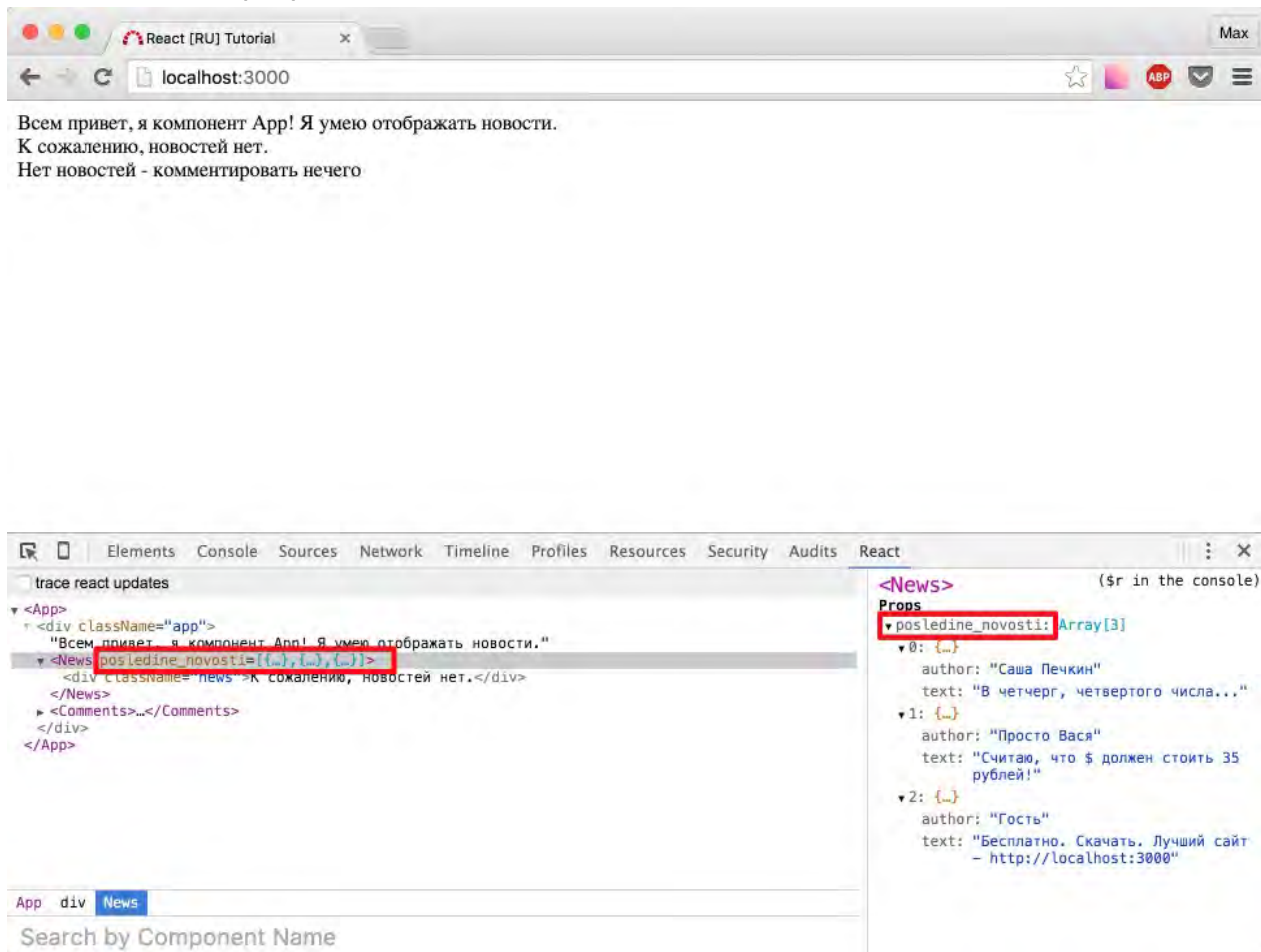
Props
data: Array[3]
  0: {
    author: "Саша Печкин"
    text: "В четверг, четвертого числа..."
  }
  1: {
    author: "Просто Вася"
    text: "Считаю, что $ должен стоить 35 рублей!"
  }
  2: {
    author: "Гость"
    text: "Бесплатно. Скачать. Лучший сайт - http://localhost:3000"
  }

```

Напомню: в данный момент, мы добавили свойство *data* в наш компонент `<News />`. Необязательно было называть свойство так, можно было написать, например:

```
<News posledine_novosti={my_news} />
```

Тогда в консоли разработчика, это выглядело бы так:



Сейчас у нашего компонента есть свойство, в котором лежат наши новости, но компонент не умеет их отображать. Это легко исправить.

Представим HTML разметку для наших новостей:

```
<div class="news">
  <p class="news__author">Саша Печкин:</p>
  <p class="news__text">В четверг, четвертого числа...</p>
</div>
```

Вопрос: У нас есть разметка для одного элемента данных, есть данные целиком (массив `my_news`). Как отобразить эти данные?

Ответ: Нужно создать шаблон, пройтись по всем переменным в массиве с новостями и подставить значения.

Когда вам нужно отобразить переменную в шаблоне разметки - она так же оборачивается в фигурные скобки. На практике это выглядит проще, чем в теории, поэтому давайте представим, как может выглядеть наша JSX разметка:

```
this.props.data.map(function(item, index) {  
  return (  
    <div key={index}>  
      <p className="news__author">{item.author}</p>  
      <p className="news__text">{item.text}</p>  
    </div>  
  )  
})
```

1. Мы использовали метод массивов - *Map*. Если вы незнакомы с ним, [прочитайте документацию](#).
2. Мы обернули разметку внутри *return* в корневой элемент `<div>`. Мы могли бы обернуть ее в любой другой элемент, главное, что нужно **запомнить** - внутри *return* всегда должен возвращаться DOM-узел (то есть, что угодно, обернутое в родительский тэг).
3. Мы использовали у родительского элемента атрибут **key** (`<div key={index}>`). Если объяснить предельно просто: реакту нужна уникальность, чтобы все его механизмы работали корректно. По "ключу" он будет понимать с каким именно дочерним узлом вы работаете и какому родителю он принадлежит. По поводу использования индекса в качестве ключа в конце урока есть ценное замечание.
4. Мы использовали в шаблоне, значения переменных + текст, например `<p className="news__author">{item.author}</p>` могло бы быть представлено в нативном js-коде как `<p className="news__author">'+item.author+': '</p>` (пустая строка + значение переменной + двоеточие)

В результате работы функции *map*, мы получили новый массив, состоящий из необходимых нам реакт-элементов. Это и есть решение нашей задачи, нам осталось лишь сохранить этот массив в переменной, например `newsTemplate`, и в *render* функции компонента `<News />` вернуть разметку + "переменную-шаблон".

js/app.js

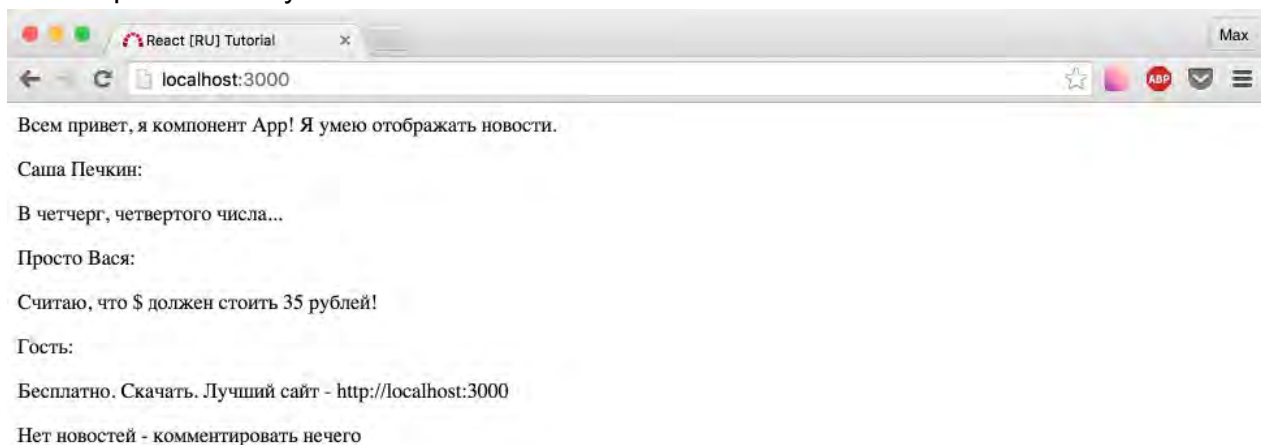

```

...
var News = React.createClass({
  render: function() {
    var data = this.props.data;
    var newsTemplate = data.map(function(item, index) {
      return (
        <div key={index}>
          <p className="news__author">{item.author}</p>
          <p className="news__text">{item.text}</p>
        </div>
      )
    })

    return (
      <div className="news">
        {newsTemplate}
      </div>
    );
  }
});
...

```

Посмотрим что получилось:



Напоследок, мне не хочется, но придется вновь разрушить магию. Давайте сконсолим, *newsTemplate*. Просто добавьте `console.log` перед `return`'ом метода *render* компонента

```
<News /> .
```

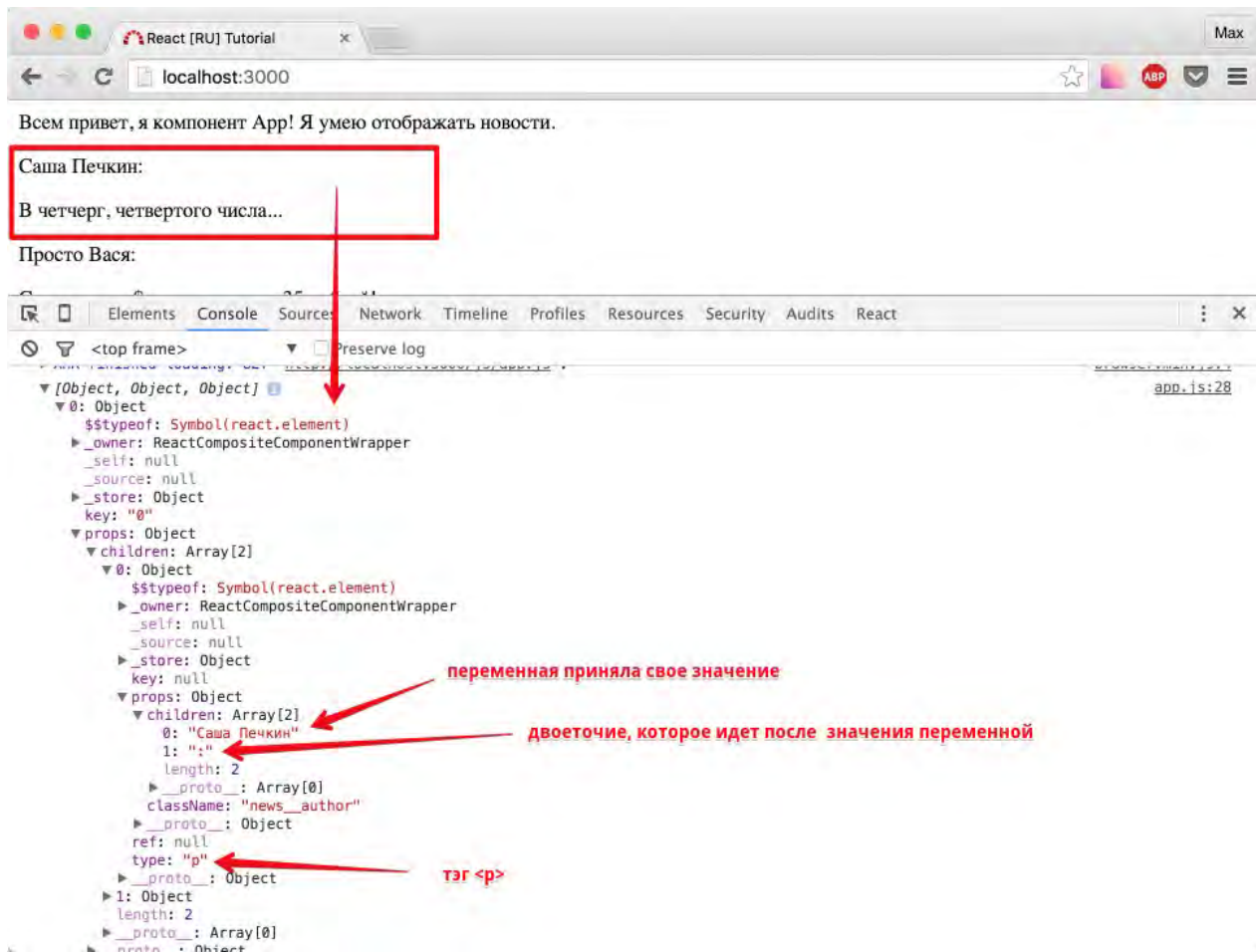
js/app.js

```
...
var News = React.createClass({
  render: function() {
    var data = this.props.data;
    var newsTemplate = data.map(function(item, index) {
      return (
        <div key={index}>
          <p className="news__author">{item.author}</p>
          <p className="news__text">{item.text}</p>
        </div>
      )
    })

    console.log(newsTemplate);

    return (
      <div className="news">
        {newsTemplate}
      </div>
    );
  }
});
...
```

Взглянем в консоль:



Объект, у объекта свойства... все как обычно в мире javascript.

Итого: мы научились отображать свойства компонента.

[Исходный код](#) на текущий момент здесь. Не забудьте удалить console.log.

P.S. Здесь и в продолжении всего курса в коде для отображения массива новостей используется `key = {index}`. Обратите внимание на следующую ветку [комментариев](#) (спасибо *DeLaVega* и *geakstr*).

If-else, тернарный оператор

Помните, у нас была фраза "новостей нет"? Хорошо бы ее отображать, если новостей действительно нет.

Для начала, научимся отображать общее количество новостей, допустим внизу, после списка новостей.

Как бы сказал участник игры "Угадай JS" - я напишу это за одну строку. Что скажете вы? Подсказка:

```
var News = React.createClass({
  render: function() {
    var data = this.props.data;
    var newsTemplate = data.map(function(item, index) {
      return (
        <div key={index}>
          <p className="news__author">{item.author}</p>
          <p className="news__text">{item.text}</p>
        </div>
      )
    })

    return (
      <div className="news">
        {newsTemplate}
        ЭТА_СТРОКА_ЗДЕСЬ
      </div>
    );
  }
});
```

Ответ:

```
<strong>Всего новостей: {data.length}</strong>
```

Поиграйтесь с переменной *my_news*. Сделайте ее пустым массивом, добавьте/удалите элементы. Пообновляйте страницу. Количество новостей должно работать корректно.

Вернемся к нашей задаче. Алгоритм прост:

Создаем переменную *newsTemplate*, если новости есть - в переменную по-прежнему будем передавать результат работы функции *map*, иначе - будем передавать сразу разметку.

Компонент News:

```
var News = React.createClass({
  render: function() {
    var data = this.props.data;
    var newsTemplate;

    if (data.length > 0) {
      newsTemplate = data.map(function(item, index) {
        return (
          <div key={index}>
            <p className="news__author">{item.author}</p>
            <p className="news__text">{item.text}</p>
          </div>
        )
      })
    } else {
      newsTemplate = <p>К сожалению новостей нет</p>
    }

    return (
      <div className="news">
        {newsTemplate}
        <strong>Всего новостей: {data.length}</strong>
      </div>
    );
  }
});
```

Неплохо, но если нет новостей - зачем нам показывать, что всего новостей 0? Давайте решим это с помощью CSS класса *.none*, который будем добавлять если новостей нет.

Для этого создайте CSS файл с содержимым:

css/app.css

```
.none {
  display: none !important;
}
```

и подключите его в *index.html*

index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>React [RU] Tutorial</title>
    <link rel="stylesheet" href="css/app.css">
  </head>
  <body>
    <div id="root"></div>

    <script src="js/react/react.js"></script>
    <script src="js/react/react-dom.js"></script>
    <script src="js/react/browser.min.js"></script>
    <script type="text/babel" src="js/app.js"></script>
  </body>
</html>
```

С классом `.none` все вновь решается в одну строку.

Измените строку про количество новостей следующим образом:

```
<strong className={data.length > 0 ? '' : 'none'}>Всего новостей: {data.length}</strong>
```

Проще простого: *есть новости ? 'пустой класс' : 'класс .none'*

Для работы с классами, когда их становится больше и условия становятся сложнее, можно использовать [classNames \(NPM пакет\)](#). Но сейчас в этом нет необходимости.

Итого: если вам нужно отобразить что-то в зависимости от условий, делайте это так же, как если бы react не был подключен. Для удобства, мы использовали *переменную-шаблон*, которую объявили **заранее**, а затем в зависимости от условия сохраняли в нее необходимую разметку.

[Исходный код](#) на данный момент.

P.S. [официальная документация про If-else внутри JSX](#)

Порефакторим...

Для начала, удалите вовсе компонент `<Comments />` (и `var Comments = ...` соответственно).

Далее, давайте представим: у наших новостей появляются какие-то дополнительные поля, пользователь начинает взаимодействовать с ними, например "пометить как прочитанное" и так далее. Нам было бы удобно, чтобы каждая новость была представлена компонентом.

Задача: `<News />` должен рендерить список компонентов `<Article />`. Каждый компонент `<Article />` должен получать соответствующие данные, например: первый экземпляр получит первый элемент массива, второй - второй и так далее.

То есть, раньше в `map` мы возвращали JSX-разметку. Но мы так же можем возвращать и компонент.

Попробуйте сами, а потом смотрите решение ниже.

Подсказка #1: `if-else` нашего компонента `<News />`

```
if (data.length > 0) {
  newsTemplate = data.map(function(item, index) {
    return (
      <div key={index}>
        <Article data={item} />
      </div>
    )
  })
} else {
  newsTemplate = <p>К сожалению новостей нет</p>
}
```

Подсказка #2 (по сути решение задачи): компонент `<Article />`

```
var Article = React.createClass({
  render: function() {
    var author = this.props.data.author,
        text = this.props.data.text;

    return (
      <div className="article">
        <p className="news__author">{author}</p>
        <p className="news__text">{text}</p>
      </div>
    )
  }
});
```

Что любопытно, больше не изменилось ни-че-го.

Замените фразу "всем привет, я компонент App..." на обычный `<h3>` заголовок - "Новости".

```
var App = React.createClass({
  render: function() {
    return (
      <div className="app">
        <h3>Новости</h3>
        <News data={my_news} />
      </div>
    );
  }
});
```

Добавьте красоты (CSS) по вкусу, либо возьмите мой вариант:

`css/app.css`


```

.none {
  display: none !important;
}

body {
  background: rgba(0, 102, 255, 0.38);
  font-family: sans-serif;
}

p {
  margin: 0 0 5px;
}

.article {
  background: #FFF;
  border: 1px solid rgba(0, 89, 181, 0.82);
  width: 600px;
  margin: 0 0 5px;
  box-shadow: 2px 2px 5px -1px rgb(0, 81, 202);
  padding: 3px 5px;
}

.news__author {
  text-decoration: underline;
  color: #007DDC;
}

.news__count {
  margin: 10px 0 0 0;
  display: block;
}

```

С новыми стилями, код сценария выглядит так:

js/app.js

```

var my_news = [
  {
    author: 'Саша Печкин',
    text: 'В четчерг, четвертого числа...'
  },
  {
    author: 'Просто Вася',
    text: 'Считаю, что $ должен стоить 35 рублей!'
  },
  {
    author: 'Гость',
    text: 'Бесплатно. Скачать. Лучший сайт - http://localhost:3000'
  }
];

var Article = React.createClass({

```

```

render: function() {
  var author = this.props.data.author,
      text = this.props.data.text;

  return (
    <div className='article'>
      <p className='news__author'>{author}</p>
      <p className='news__text'>{text}</p>
    </div>
  )
}
});

var News = React.createClass({
  render: function() {
    var data = this.props.data;
    var newsTemplate;

    if (data.length > 0) {
      newsTemplate = data.map(function(item, index) {
        return (
          <div key={index}>
            <Article data={item} />
          </div>
        )
      })
    } else {
      newsTemplate = <p>К сожалению новостей нет</p>
    }

    return (
      <div className='news'>
        {newsTemplate}
        <strong className={'news__count ' + (data.length > 0 ? '' : 'none')}>Всего новос
тей: {data.length}</strong>
      </div>
    );
  }
});

var App = React.createClass({
  render: function() {
    return (
      <div className='app'>
        <h3>Новости</h3>
        <News data={my_news} />
      </div>
    );
  }
});

ReactDOM.render(
  <App />,

```

```
document.getElementById('root')  
);
```

Обратите внимание, я добавил несколько классов, один из них я добавил к тэгу **strong**. Как вы помните, у нас там было условие:

```
<strong className={data.length > 0 ? '': 'none'}>Всего новостей: {data.length}</strong>
```

Так как атрибут *class* (либо в JSX - *className*) у элемента принимает строку, то можно представить элемент с несколькими классами:

```
<div className={ 'class1' + 'class2' + 'class3' }>text</div>
```

В таком случае к нашему элементу применится ... класс "**class1class2class3**", что естественно, так как мы забыли пробелы. Наверное, мы хотели сделать так:

```
<div className={ 'class1 ' + 'class2 ' + 'class3' }>text</div>
```

Теперь у нас будет элемент с классами: class1, class2, class3

Именно поэтому, мы и получили следующее выражение:

```
<strong className={'news__count ' + (data.length > 0 ? '': 'none')}>Всего новостей: {d  
ata.length}</strong>
```

Посмотрим, что вышло в итоге:

The screenshot shows a web browser window with the address bar at `localhost:3000`. The page title is "React [RU] Tutorial". The main content area has a light blue background and displays three news items, each in a white box with a blue border:

- Саша Печкин:**
В четчерг, четвертого числа...
- Просто Вася:**
Считаю, что \$ должен стоить 35 рублей!
- Гость:**
Бесплатно. Скачать. Лучший сайт - <http://localhost:3000>

Below the news items, it says "Всего новостей: 3".

The bottom of the image shows the React DevTools component inspector. The "Elements" tab is active, showing the component tree:

```
<App>  
  <div className="app">  
    <h3>Новости</h3>  
    <News data={[...], [...], [...]}>...</News>  
  </div>  
</App>
```

The "Props" tab shows an empty object: `Props Empty object`. The "Search by Component Name" input field is visible, with "App" entered.

Либо:

The screenshot shows a web browser window with the address bar at `localhost:3000`. The page has a blue background and contains the text "Новости" and "К сожалению новостей нет". The browser's developer tools are open, showing the DOM tree and the CSS styles for the selected element.

DOM Tree:

```
<!DOCTYPE html>
<html>
  >#shadow-root (open)
  ><head>_</head>
  ><body>
    ><div id="root">
      ><div class="app" data-reactid=".0">
        ><h3 data-reactid=".0.0">Новости</h3>
        ><div class="news" data-reactid=".0.1">
          ><p data-reactid=".0.1.0">К сожалению новостей нет</p>
          ><strong class="news__count none" data-reactid=".0.1.1">
            ><span data-reactid=".0.1.1.0">Всего новостей: </span>
            ><span data-reactid=".0.1.1.1">0</span>
          </strong>
        </div>
      </div>
    </div>
  </body>
</html>
```

Styles:

```
element.style {
}

.news__count {
  margin: 10px 0 0 0;
  display: block;
}

.none {
  display: none !important;
}

strong, b {
}
```

Исходный код на данный момент.

React.propTypes

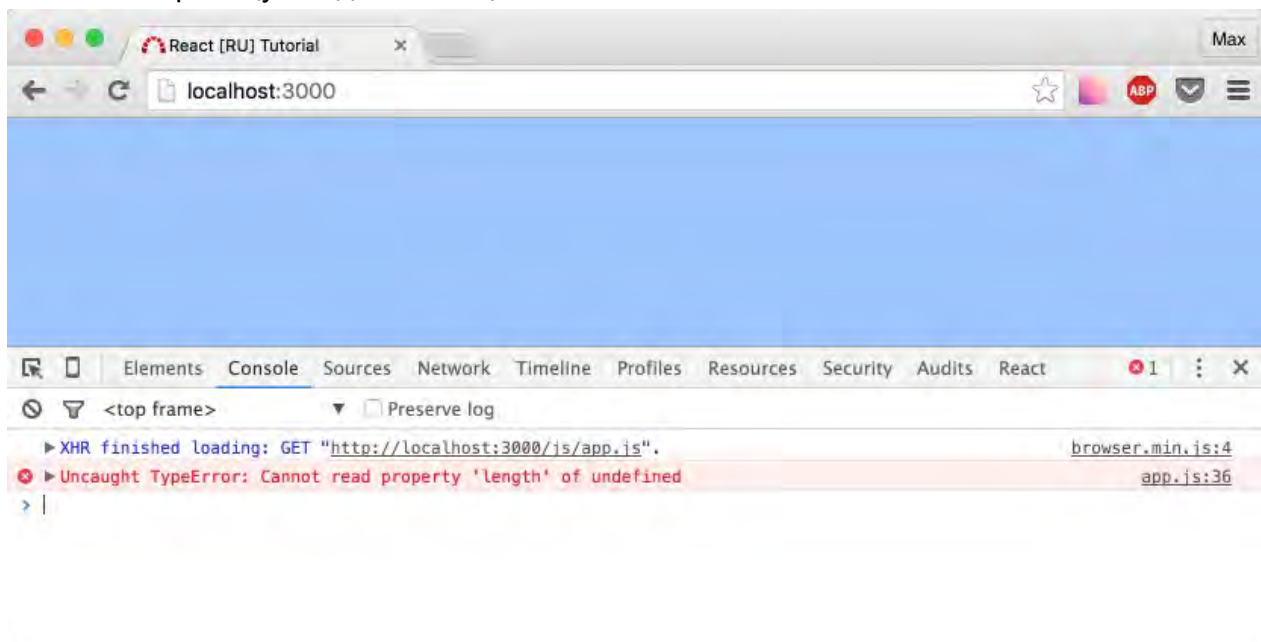
(скучный, но небольшой теоретический перекур)

Перед выполнением данного урока, не забывайте, что *PropTypes* не работает с *production* версией реакта. Эта фишка только для разработки, так как валидация - дорогая операция.

Давайте сломаем наш код:

```
var App = React.createClass({
  render: function() {
    return (
      <div className='app'>
        <h3>Новости</h3>
        <News /> {/* удалили data = {my_news} */}
      </div>
    );
  }
});
```

Обновим страницу - видим сообщение об ошибке.



В принципе, все понятно - мы пытаемся вызвать метод `map` у `undefined`. У примитива `undefined`, как известно никаких методов нет - ошибка, получите распишитесь. Хорошо, что кода мало и мы быстро выяснили в чем проблема. Еще лучше, что есть

возможность улучшить наше положение, добавив *propTypes* - специальное свойство, которое будет "валидировать" наш компонент.

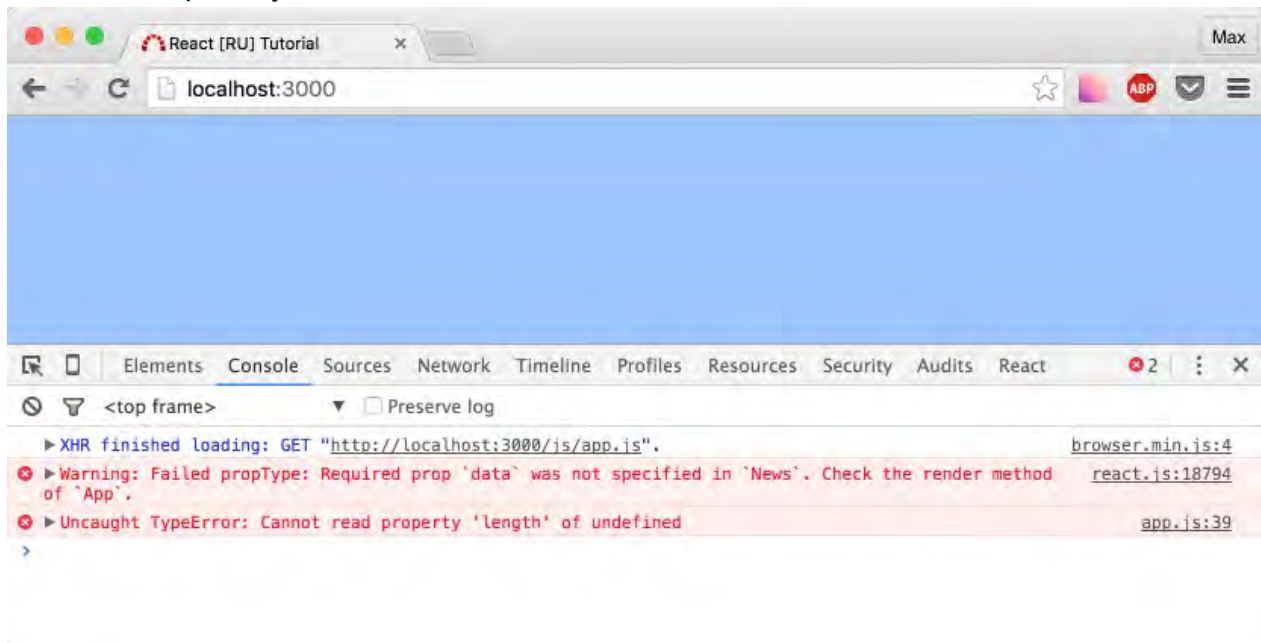
Внесите изменения в компонент `<News />`

```
var News = React.createClass({
  propTypes: {
    data: React.PropTypes.array.isRequired
  },
  render: function() {
    var data = this.props.data;
    var newsTemplate;

    if (data.length > 0) {
      newsTemplate = data.map(function(item, index) {
        return (
          <div key={index}>
            <Article data={item} />
          </div>
        )
      })
    } else {
      newsTemplate = <p>К сожалению новостей нет</p>
    }

    return (
      <div className='news'>
        {newsTemplate}
        <strong className={'news__count ' + (data.length > 0 ? '' : 'none')}>Всего новос
стей: {data.length}</strong>
      </div>
    );
  }
});
```

Обновите страницу:



Гораздо лучше! Исходя из текста ошибки нам сразу понятно куда копать: в render методе App, не указано свойство data, которое ожидается в News. Восстановим свойство data.

```
var App = React.createClass({
  render: function() {
    return (
      <div className='app'>
        <h3>Новости</h3>
        <News data={my_news}/>
      </div>
    );
  }
});
```

Вновь все работает, и наша консоль чиста.

Подробнее о propTypes

Приведу выдержку из [офф. документации](#):


```
React.createClass({
  propTypes: {
    // Вы можете указать, каким примитивом должно быть свойство
    optionalArray: React.PropTypes.array,
    optionalBool: React.PropTypes.bool,
    optionalFunc: React.PropTypes.func,
    optionalNumber: React.PropTypes.number,
    optionalObject: React.PropTypes.object,
    optionalString: React.PropTypes.string,

    // Вы можете указать, что свойство может быть одним из ...
    optionalUnion: React.PropTypes.oneOfType([
      React.PropTypes.string,
      React.PropTypes.number,
      React.PropTypes.instanceOf(Message)
    ]),

    // Вы можете указать, конкретную структуру объекта свойства
    optionalObjectWithShape: React.PropTypes.shape({
      color: React.PropTypes.string,
      fontSize: React.PropTypes.number
    }),

    // Вы можете указать, что свойство ОБЯЗАТЕЛЬНО
    requiredFunc: React.PropTypes.func.isRequired,

    // Если нужно указать, что свойство просто обязательно, и может быть любым примитивом
    requiredAny: React.PropTypes.any.isRequired,

  }
});
```

Согласно этому листингу, мы можем перевести правило, указанное в компоненте

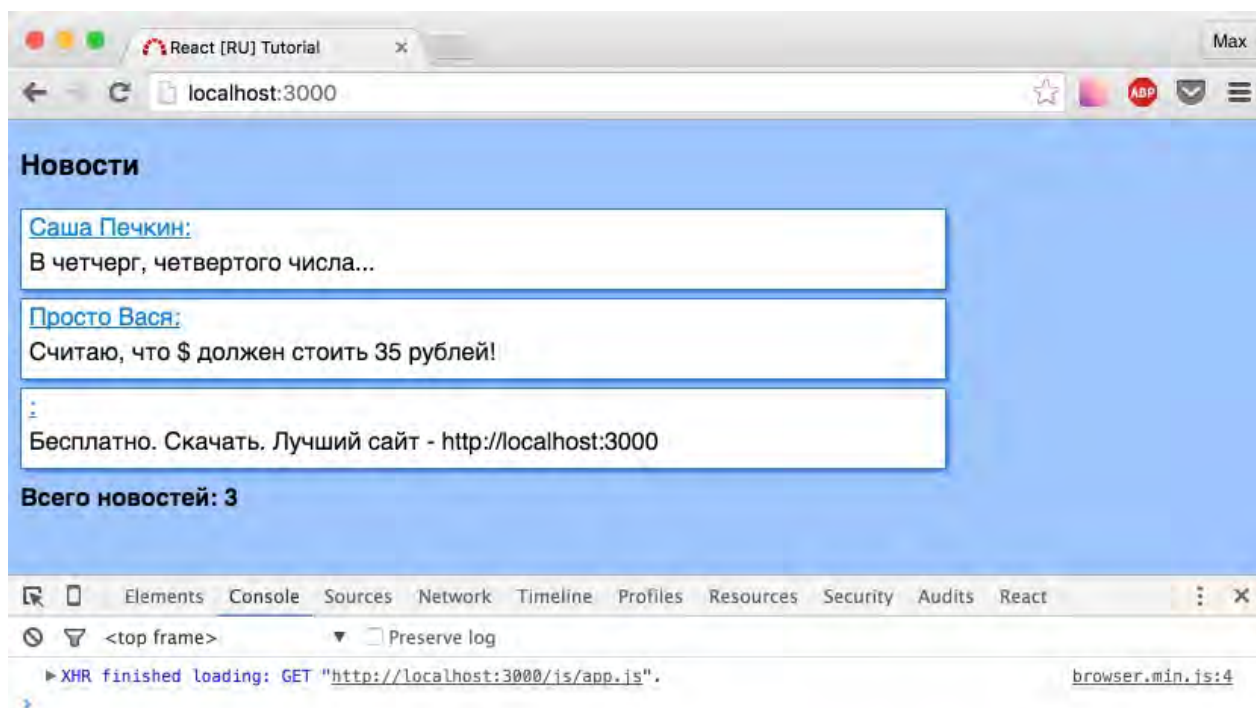
```
<News /> :
```

`React.PropTypes.array.isRequired` - свойство должно быть массивом и оно обязательно должно быть!

Я вижу по глазам некоторых (да-да, вижу), что все это какая-то бесполезная лабуда. И так понятно - есть ошибка, есть возможность тыкнуть на нее в дебаггере и посмотреть. Специально для вас, следующая ситуация: удалите из массива `my_news` автора, например в третьем элементе:

```
var my_news = [  
  {  
    author: 'Саша Печкин',  
    text: 'В четчерг, четвертого числа...'  
  },  
  {  
    author: 'Просто Вася',  
    text: 'Считаю, что $ должен стоить 35 рублей!'  
  },  
  {  
    text: 'Бесплатно. Скачать. Лучший сайт - http://localhost:3000'  
  }  
];
```

Посмотрим результат:



Никаких ошибок. Но наше приложение **не работает** так как надо. Кто виноват? Реакт? Backend-программист который прислал нам такие данные?

Программист, может и виноват. Но реакт точно нет. У нас получилось, что в `this.props.data.author` содержится `undefined` (переменная не определена). Поэтому react так и поступил, и показал нам "ничего" (на скриншоте это только "двоеточие").

Новости

Саша Печкин:
В четчерг, четвертого числа...

Просто Вася:
Считаю, что \$ должен стоить 35 рублей!

Бесплатно. Скачать. Лучший сайт - http://localhost:3000

Всего новостей: 3

React DevTools Component Inspector:

```

<Article>
  Props
  data: {
    text: "Бесплатно. Скачать. Лучший сайт - http://localhost:3000"
    author: undefined
  }

```

data.author === undefined

Такую ошибку сложно отловить.

Добавьте `propTypes` в компонент `<Article />`

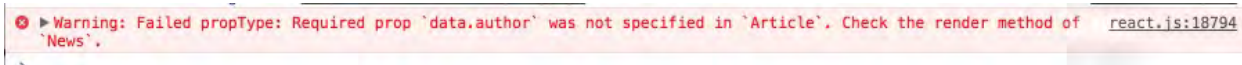
```

var Article = React.createClass({
  propTypes: {
    data: React.PropTypes.shape({
      author: React.PropTypes.string.isRequired,
      text: React.PropTypes.string.isRequired
    })
  },
  render: function() {
    var author = this.props.data.author,
        text = this.props.data.text;

    return (
      <div className='article'>
        <p className='news__author'>{author}</p>
        <p className='news__text'>{text}</p>
      </div>
    )
  }
});

```

В таком случае вы получите сообщение об ошибке:



Warning: Failed propType: Required prop `data.author` was not specified in `Article`. Check the render method of `News`.

The screenshot shows a red warning message in a console. The text of the warning is: "Warning: Failed propType: Required prop `data.author` was not specified in `Article`. Check the render method of `News`." The file path "react.js:18794" is visible on the right side of the warning box.

Разве это не прекрасно?

[Исходный код](#) на данный момент.

P.S. Не забудьте вернуть автора ;)

Использование state

Вернемся от теории к практике, покликаем по ссылкам-кнопочкам, поизменяем свойства компонентов...

Упс, простите. Как вы помните, свойства (*this.props*) следует использовать только для чтения, а для динамических свойств нужно использовать так называемое "состояние" (*state*).

Итак, встречайте - **this.state** ;)

Так как мне нужно в этом разделе сохранить минимум теории и больше практики, сразу перейдем к делу. Предлагаю вместе решить следующую задачу: *у новости есть ссылка "подробнее", по клику на которую - бинго, текст новости целиком.*

Начнем с изменения данных:

```
var my_news = [
  {
    author: 'Саша Печкин',
    text: 'В четчерг, четвертого числа...',
    bigText: 'в четыре с четвертью часа четыре чёрненьких чумазеньких чертёнка чертили чёрными чернилами чертёж.'
  },
  {
    author: 'Просто Вася',
    text: 'Считаю, что $ должен стоить 35 рублей!',
    bigText: 'А евро 42!'
  },
  {
    author: 'Гость',
    text: 'Бесплатно. Скачать. Лучший сайт - http://localhost:3000',
    bigText: 'На самом деле платно, просто нужно прочитать очень длинное лицензионное соглашение'
  }
];
```

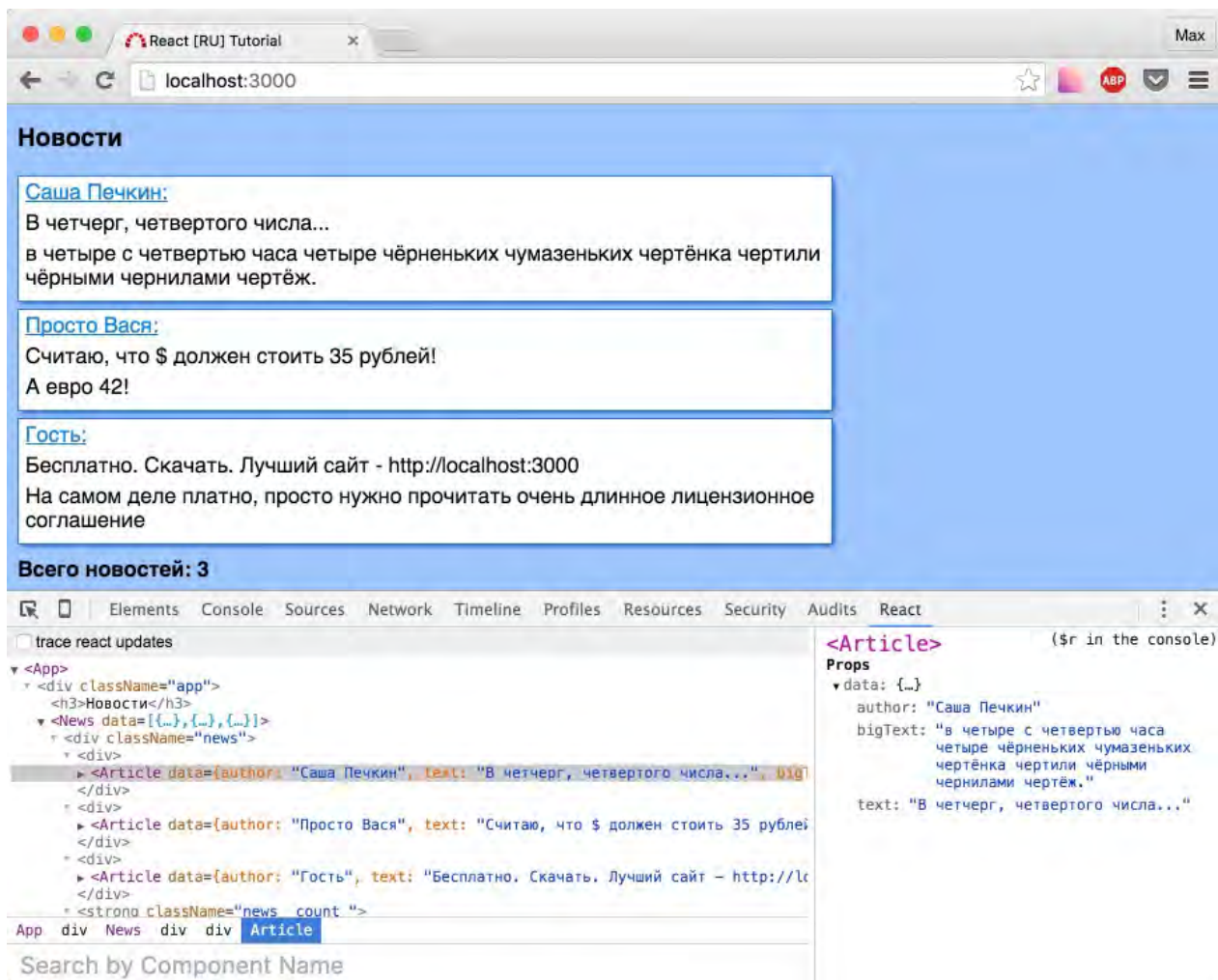
Затем, научимся отображать полный текст новости сразу после вводного текста:

```
var Article = React.createClass({
  propTypes: {
    data: React.PropTypes.shape({
      author: React.PropTypes.string.isRequired,
      text: React.PropTypes.string.isRequired,
      bigText: React.PropTypes.string.isRequired
    })
  },
  render: function() {
    var author = this.props.data.author,
        text = this.props.data.text,
        bigText = this.props.data.bigText;

    return (
      <div className='article'>
        <p className='news__author'>{author}</p>
        <p className='news__text'>{text}</p>
        <p className='news__big-text'>{bigText}</p>
      </div>
    )
  }
});
```

Опять же, больше ничего изменять не нужно. Данные корректно отобразятся.

Проверим...



Отлично, можно продолжать работу: добавим ссылку - "подробнее". Приведу фрагмент кода:

```

    ...
    return (
      <div className='article'>
        <p className='news__author'>{author}</p>
        <p className='news__text'>{text}</p>
        <a href="#" className='news__readmore'>Подробнее</a>
        <p className='news__big-text'>{bigText}</p>
      </div>
    )
    ...

```

Проверьте и если все ок - мы готовы к работе с состоянием компонента.

getInitialState

Если вы определяете какое-то изменяемое свойство в компоненте, необходимо указать начальное состояние (в терминологии react.js - *initial state*). Для этого, у компонента указывается метод *getInitialState*:

```
var Article = React.createClass({
  propTypes: {
    data: React.PropTypes.shape({
      author: React.PropTypes.string.isRequired,
      text: React.PropTypes.string.isRequired,
      bigText: React.PropTypes.string.isRequired
    })
  },
  getInitialState: function() {
    return {
      visible: false
    };
  },
  render: function() {
    var author = this.props.data.author,
        text = this.props.data.text,
        bigText = this.props.data.bigText;

    return (
      <div className='article'>
        <p className='news__author'>{author}</p>
        <p className='news__text'>{text}</p>
        <a href="#" className='news__readmore'>Подробнее</a>
        <p className='news__big-text'>{bigText}</p>
      </div>
    )
  }
});
```

Посмотрим в консоли на вкладке React:

The screenshot shows a web browser at localhost:3000 displaying a news application titled "Новости". It lists three news items, each with an author, text, and a "Подробнее" link. Below the browser, the React DevTools component inspector is open, showing the component tree on the left and the props/state for the selected `<Article>` component on the right. The `visible` property in the state is highlighted with a red circle.

Новости

Саша Печкин:
В четчерг, четвертого числа...
[Подробнее](#)
в четыре с четвертью часа четыре чёрненьких чумазеньких чертёнка чертили чёрными чернилами чертёж.

Просто Вася:
Считаю, что \$ должен стоить 35 рублей!
[Подробнее](#)
А евро 42!

Гость:
Бесплатно. Скачать. Лучший сайт - http://localhost:3000
[Подробнее](#)

React DevTools Component Inspector

<Article> (\$r in the console)

Props

- data: {...}

State

- visible: false

В состоянии (в *state*) появилось свойство. Будем использовать его в момент render'a.

```

var Article = React.createClass({
  propTypes: {
    data: React.PropTypes.shape({
      author: React.PropTypes.string.isRequired,
      text: React.PropTypes.string.isRequired,
      bigText: React.PropTypes.string.isRequired
    })
  },
  getInitialState: function() {
    return {
      visible: false
    };
  },
  render: function() {
    var author = this.props.data.author,
        text = this.props.data.text,
        bigText = this.props.data.bigText,
        visible = this.state.visible; // считываем значение переменной из состояния ко
мпонента

    return (
      <div className='article'>
        <p className='news__author'>{author}</p>
        <p className='news__text'>{text}</p>

        /* для ссылки readmore: не показывай ссылку, если visible === true */
        <a href="#" className={'news__readmore ' + (visible ? 'none': '')}>Подробнее</a>

        /* для большо текста: не показывай текст, если visible === false */
        <p className={'news__big-text ' + (visible ? '' : 'none')}>{bigText}</p>
      </div>
    )
  }
});

```

Можно проверить в браузере, я лишь заострю внимание, что мы вновь использовали выражение в качестве значения CSS-класса у элемента (не забывайте *пробел*), а так же использовали одну и ту же переменную состояния в **двух** местах: теперь, когда мы научимся изменять состояние по клику (чуть ниже), если показывается большой текст - ссылка подробнее, наоборот, прячется.

Обработка кликов - onClick

Чтобы обработать клик, нам необходимо указать атрибут *onClick* у элемента.

В качестве обработчика у нас будет функция, которая изменяет состояние. Для изменения состояния, нужно **обязательно** использовать метод **setState**, а не просто перезаписывать значение переменной.

```
var Article = React.createClass({
  propTypes: {
    data: React.PropTypes.shape({
      author: React.PropTypes.string.isRequired,
      text: React.PropTypes.string.isRequired,
      bigText: React.PropTypes.string.isRequired
    })
  },
  getInitialState: function() {
    return {
      visible: false
    };
  },
  readmoreClick: function(e) {
    e.preventDefault();
    this.setState({visible: true});
  },
  render: function() {
    var author = this.props.data.author,
        text = this.props.data.text,
        bigText = this.props.data.bigText,
        visible = this.state.visible;

    return (
      <div className='article'>
        <p className='news__author'>{author}</p>
        <p className='news__text'>{text}</p>
        <a href="#" onClick={this.readmoreClick} className='news__readmore ' + (visible ? 'none': '')>Подробнее</a>
        <p className='news__big-text ' + (visible ? '': 'none')>{bigText}</p>
      </div>
    )
  }
});
```

Проверьте в браузере, кликните на ссылку "подробнее".

--

Итого:

Для сохранения динамических свойств, используется состояние (*state*) компонента.

Для обработки клика, используется свойство *onClick* + функция-обработчик.

Существуют и другие стандартные события, которые работают по такому же принципу.

Полный список [здесь](#), а мы будем знакомиться с ними по мере необходимости.

Для изменения состояния, используется метод **setState**, который принимает объект с аргументами, которые нужно изменить. Например, у нас есть состояние:

```
...
getInitialState: function() {
  return {
    visible: false,
    rating: 0,
    eshe_odno_svoistvo: 'qweqwe'
  };
}
...
```

Чтобы изменить рейтинг, нужно написать следующий setState:

```
this.setState({rating: 100500})
```

Чтобы изменить все три свойства:

```
this.setState({
  rating: 100500,
  visible: true,
  eshe_odno_svoistvo: 'привет'
})
```

Так же у *setState* есть возможность указать *callback* функцию, которая будет вызвана после того, как новое состояние "установится".

```
...
readmoreClick: function(e) {
  e.preventDefault();
  this.setState({visible: true}, function() {
    alert('Состояние изменилось');
  });
},
...
```

Напоследок, немного о стиле кода.

У ссылки "подробнее" получилась длинная строка, кому-то может понравится такой вариант:

```
...
return (
  <div className='article'>
    <p className='news__author'>{author}</p>
    <p className='news__text'>{text}</p>
    <a href="#"
      onClick={this.readmoreClick}
      className={'news__readmore ' + (visible ? 'none': '')}>
        Подробнее
    </a>
    <p className={'news__big-text ' + (visible ? '' : 'none')}>{bigText}</p>
  </div>
)
...
```

Я не люблю длинные строки, поэтому оставлю второй вариант.

[Исходный код](#) на данный момент.

Продвинутое использование state

В этом разделе мы посмотрим как изменение *state* влияет на компонент и немного "зацепим" *stateless* архитектуру.

Изменение state вызывает render компонента

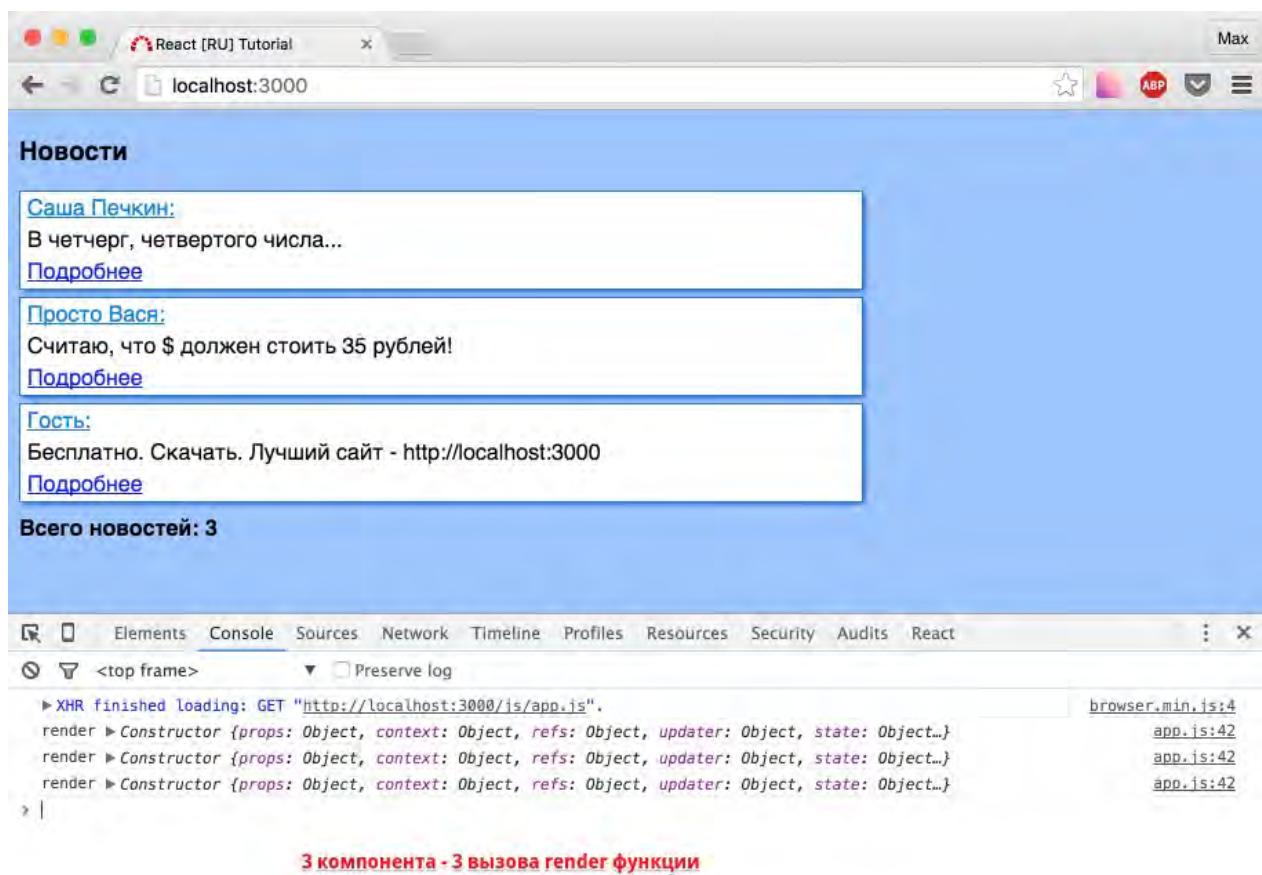
Все указано в подзаголовке, предлагаю нам в этом убедиться:

фрагмент компонента `<Article />` :

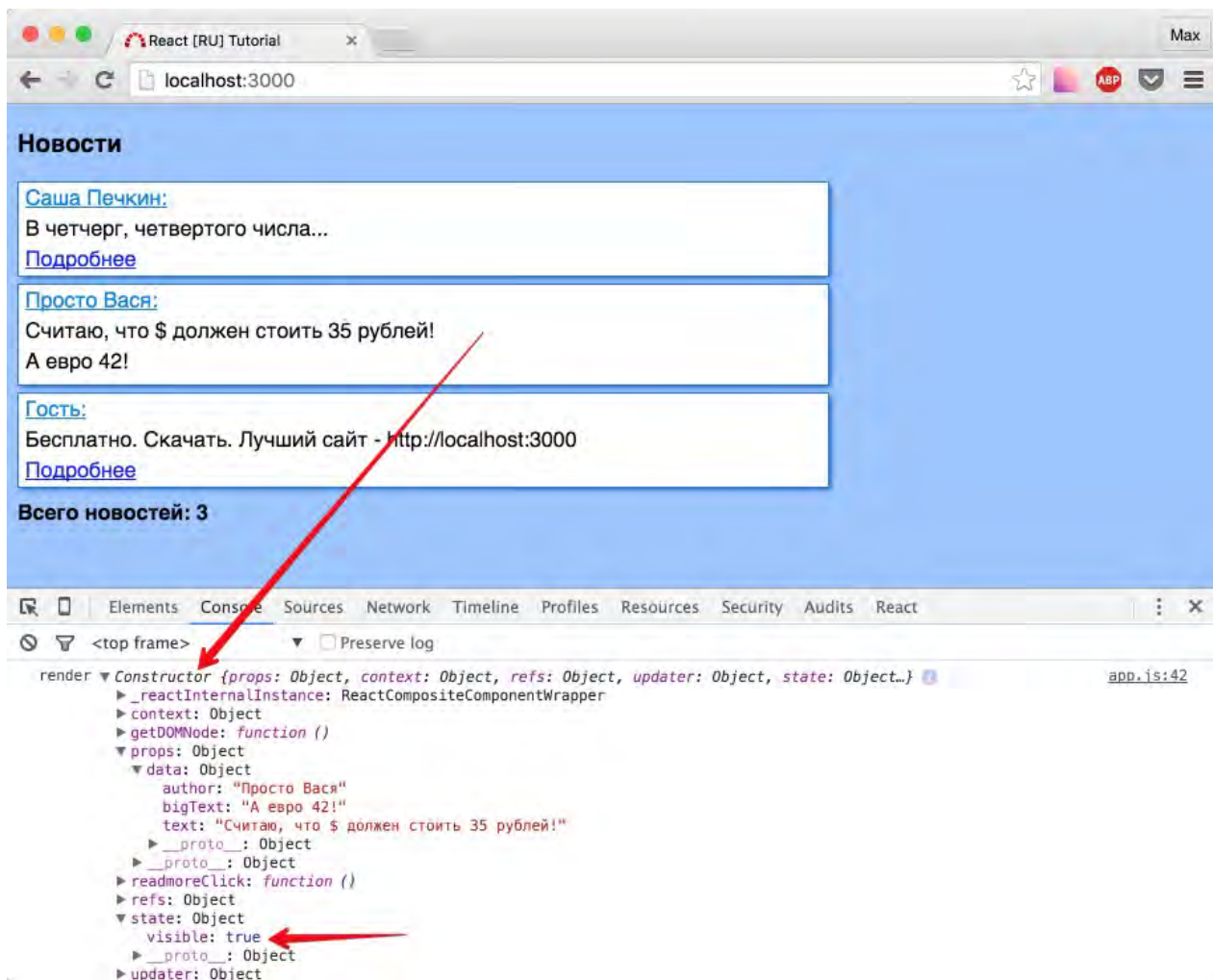
```
...
render: function() {
  var author = this.props.data.author,
      text = this.props.data.text,
      bigText = this.props.data.bigText,
      visible = this.state.visible;

  console.log('render', this); //добавили console.log

  return (
    <div className='article'>
      <p className='news__author'>{author}</p>
      <p className='news__text'>{text}</p>
      <a href="#"
        onClick={this.readmoreClick}
        className={'news__readmore ' + (visible ? 'none': '')}>
        Подробнее
      </a>
      <p className={'news__big-text ' + (visible ? '' : 'none')}>{bigText}</p>
    </div>
  )
}
...
```



Очистите консоль, и нажмите подробнее на любой из новостей:



Убедились? Запомните *первое правило*: **нельзя** вызывать `setState` в `render`: реакт видит изменилось состояние - начинает перерисовывать компонент - видит что изменилось состояние - начинает перерисовывать компонент...

Второе правило: `render` - дорогостоящая операция, поэтому внимательно относитесь к тому, где вы вызываете `setState`, и что это за собой влечет. Банальные `console.log`'и могут вам в этом помочь.

Очевидно, что если перерисовывается родительский компонент, то будут перерисованы и все дочерние компоненты. Дальше мы с вами пройдем разные "стадии жизни" компонента, и убедимся, что во время его "перерисовки" могут выполняться разные дорогостоящие операции и даже аjax-запросы. Пока что, просто убедимся, что вызов `setState` родителя - перерисует дочерние компоненты. Для этого предлагаю создать обработчик `onClick` на фразе "Всего новостей".

Попробуйте сами.

Задача: Необходимо добавить компоненту `<News />` свойство состояния - `counter`, в котором хранится количество кликов по фразе. То есть обычный автоинкремент.

В решении важно использовать `this.setState({counter: ++this.state.counter})` , об этом мы подробно поговорим после решения, которое представлено ниже как обычно в виде подсказок и полностью.

Подсказка **#1**: добавьте метод *getInitialState* в компонент `<News />` для создания начального состояния.

```
...
getInitialState: function() {
  return {
    counter: 0
  }
}
...
```

Подсказка **#2**: добавьте обработчик *onClick* с функцией, которая будет увеличивать счетчик (следовательно изменять *state*, следовательно вызывать *this.setState...*).

Решение: Полный код компонента `<News />`

```
var News = React.createClass({
  propTypes: {
    data: React.PropTypes.array.isRequired
  },
  getInitialState: function() {
    return {
      counter: 0
    }
  },
  onTotalNewsClick: function() {
    this.setState({counter: ++this.state.counter });
  },
  render: function() {
    var data = this.props.data;
    var newsTemplate;

    if (data.length > 0) {
      newsTemplate = data.map(function(item, index) {
        return (
          <div key={index}>
            <Article data={item} />
          </div>
        )
      })
    } else {
      newsTemplate = <p>К сожалению новостей нет</p>
    }

    return (
      <div className='news'>
        {newsTemplate}
        <strong
          className={'news__count ' + (data.length > 0 ? '':'none')}
          onClick={this.onTotalNewsClick}>
          Всего новостей: {data.length}
        </strong>
      </div>
    );
  }
});
```

Проверьте в браузере. Если вы не удаляли console.log из компонента `<Article />` - на каждый клик по фразе "Всего новостей", в консоли будет появляться по 3 "перерисовки".

Поговорим о: `this.setState({counter: ++this.state.counter })`

Почему же, было важно использовать именно префиксную запись ++, а не постфиксную? Сначала вспомним теорию:

- ++ перед переменной (префикс) - сначала увеличивает ее на 1, а потом возвращает значение;
- ++ после переменной (постфикс) - сначала вернет значение, а потом увеличит значение переменной;

В таком случае, мы должны были бы потерять всего 1 клик, не так ли? Проверьте в консоли следующим образом: откройте вкладку React в консоли, выберите компонент `<News />`, начните кликать на фразу "Всего новостей"

- Изменяется ли значение counter, если используется префиксная запись?

Да, изменяется:

The screenshot shows a web browser at localhost:3000 displaying a news application. The page has a blue header with the word "Новости" (News). Below it, there are three news items, each with an author, a text snippet, and a "Подробнее" (More) link. At the bottom, it says "Всего новостей: 3" (Total news: 3). Below the browser window, the React DevTools component inspector is open. The left pane shows the component tree with the `<News data={...}>` component selected, highlighted by a red arrow and the text "выберите компонент". The right pane shows the props and state of the selected component. The state is `{counter: 7}`. A red arrow points to the `counter` value, with a red text box stating: "в момент клика по фразе 'Всего новостей' - значение будет изменяться динамически" (at the moment of clicking on the phrase 'Total news' - the value will change dynamically).

- Изменяется ли значение counter, если используется постфиксная запись?

Нет, не изменяется вообще. (убедитесь сами)

Ответ кроется в [официальной документации](#):

setState() - не изменяет *this.state* немедленно, а создает очередь изменений состояния. Доступ к *this.state* после вызова метода, потенциально может вернуть имеющееся (что равносильно - бывшее) значение.

Самое время крикнуть - верните мои деньги назад, и уйти... Но, не все так плачевно. Теперь вы знаете об этой особенности, и будете если что вооружены. Зачем так сделано? Вероятно, для оптимизации работы библиотеки в целом.

Вообще *state* у компонентов используется не часто. С появлением [flux](#) - подхода, комьюнити стало перемещаться на сторону *stateless* подхода, когда *state* не используется вообще (за исключением редких моментов). Мой любимый игрок данного лагеря - **Redux**, о котором я тоже написал [подробное руководство](#) на русском.

Почему сейчас мы не изучаем Redux, стоит ли бросить все и прочитать другой tutorial? Определенно - нет. Продолжайте изучение данного курса.

[Исходный код](#) с `console.log`'ами и обработчиком кликов на фразе "Всего новостей".

Работа с input

Сперва приберемся:

Удалим лишние `console.log`'и, удалим обработчик `onTotalNewsClick`.

Затем создадим компонент - `<TestInput />`, который будет просто отрисовывать (render) - input перед списком новостей.

Полный листинг нашего кода после данных манипуляций:

js/app.js

```
var my_news = [
  {
    author: 'Саша Печкин',
    text: 'В четчерг, четвертого числа...',
    bigText: 'в четыре с четвертью часа четыре чёрненьких чумазеньких чертёнка чертили чёрными чернилами чертёж.'
  },
  {
    author: 'Просто Вася',
    text: 'Считаю, что $ должен стоить 35 рублей!',
    bigText: 'А евро 42!'
  },
  {
    author: 'Гость',
    text: 'Бесплатно. Скачать. Лучший сайт - http://localhost:3000',
    bigText: 'На самом деле платно, просто нужно прочитать очень длинное лицензионное соглашение'
  }
];

var Article = React.createClass({
  propTypes: {
    data: React.PropTypes.shape({
      author: React.PropTypes.string.isRequired,
      text: React.PropTypes.string.isRequired,
      bigText: React.PropTypes.string.isRequired
    })
  },
  getInitialState: function() {
    return {
      visible: false
    };
  },
  readmoreClick: function(e) {
    e.preventDefault();
    this.setState({visible: true});
  }
});
```

```

    },
    render: function() {
      var author = this.props.data.author,
          text = this.props.data.text,
          bigText = this.props.data.bigText,
          visible = this.state.visible;

      return (
        <div className='article'>
          <p className='news__author'>{author}</p>
          <p className='news__text'>{text}</p>
          <a href="#"
            onClick={this.readmoreClick}
            className={'news__readmore ' + (visible ? 'none': '')}>
            Подробнее
          </a>
          <p className={'news__big-text ' + (visible ? '' : 'none')}>{bigText}</p>
        </div>
      )
    }
  });

var News = React.createClass({
  propTypes: {
    data: React.PropTypes.array.isRequired
  },
  getInitialState: function() {
    return {
      counter: 0
    }
  },
  render: function() {
    var data = this.props.data;
    var newsTemplate;

    if (data.length > 0) {
      newsTemplate = data.map(function(item, index) {
        return (
          <div key={index}>
            <Article data={item} />
          </div>
        )
      })
    } else {
      newsTemplate = <p>К сожалению новостей нет</p>
    }

    return (
      <div className='news'>
        {newsTemplate}
        <strong
          className={'news__count ' + (data.length > 0 ? '' : 'none')}>Всего новостей: {
data.length}</strong>

```

```
    </div>
  );
}
});

// --- добавили test input ---
var TestInput = React.createClass({
  render: function() {
    return (
      <input className='test-input' value='введите значение' />
    );
  }
});

var App = React.createClass({
  render: function() {
    return (
      <div className='app'>
        <h3>Новости</h3>
        <TestInput /> { /* добавили вывод компонента */ }
        <News data={my_news} />
      </div>
    );
  }
});

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

Напомню про комментарии:

Первый комментарий, добавлен с помощью `//`, так как данный комментарий не находится внутри JSX. А второй - находится, следовательно имеет вид `{/* комментарий */}`.

Вообще, код сейчас не работает (но это не из-за комментария). Давайте посмотрим на ошибку внимательно:

```
Warning: Failed form propTypes: You provided a `value` prop to a form field without an `onChange` handler. This will render a read-only field. If the field should be mutable use `defaultValue`. Otherwise, set either `onChange` or `readOnly`. Check the render method of `TestInput`. react.js:18794
```

Вы предоставили свойство value для поля, у которого нет onChange обработчика. Поэтому отрисовано поле только для чтения. Если поле должно быть изменяемо, используйте defaultValue. Либо установите onChange или readOnly. Проверьте render метод компонента TestInput.

Не могу не любить react за такие подробные сообщения об ошибках.

А вы кстати попробуйте сейчас изменить значение инпута. Ничего не выйдет. Здесь у нас есть два пути, и первый нам известный - использовать какое-нибудь свойство *state* в качестве динамически изменяемого значения инпута.

Controlled components (контролируемые компоненты)

Для вызова *setState*, будем использовать событие *onChange*. Работа с ним не отличается от работы с *onClick* или другими любыми событиями. Главное - передать функцию-обработчик.

Не торопитесь, давайте подумаем еще раз:

1. Нам нужно передать функцию обработчик, которая будет изменять какую-то переменную состояния.
2. Значит нам нужно создать начальное состояние (*getInitialState*).
3. Если у нас есть переменная состояния компонента, значит мы хотим, чтобы именно она была в качестве *value* у нашего инпута.

Сможете сделать сами? Если да - отлично, если нет - решение ниже.

Подсказка **#1**: так может выглядеть функция-обработчик

```
...
onChangeHandler: function(e) {
  this.setState({myValue: e.target.value})
},
...
```

Решение:


```
var TestInput = React.createClass({
  getInitialState: function() {
    return {
      myValue: ''
    };
  },
  onChangeHandler: function(e) {
    this.setState({myValue: e.target.value})
  },
  render: function() {

    return (
      <input
        className='test-input'
        value={this.state.myValue}
        onChange={this.onChangeHandler}
        placeholder='введите значение'
      />
    );
  }
});
```

У нас есть *placeholder* - "введите значение", который будет показываться в момент загрузки страницы, так как наше начальное состояние input'a - пустая строка. При изменении, мы устанавливаем в переменную *myValue* - то что введено в input. Следовательно - input корректно изменяется.

Обычно, мы хотим по клику отправлять значения инпута...

Задача: По клику на кнопку - показывать alert с текстом инпута.

Попробуйте сами.

Подсказка #1:

Вам необходимо сделать: добавить кнопку, на кнопку "повесить" обработчик *onClick*, в функции обработчике считывать значение `this.state.myValue` .

Подсказка #2:

Так как нам необходимо рендерить больше одного элемента, нужно обернуть их в родительский элемент, например в `<div></div>`

Решение:

```
var TestInput = React.createClass({
  getInitialState: function() {
    return {
      myValue: ''
    };
  },
  onChangeHandler: function(e) {
    this.setState({myValue: e.target.value})
  },
  onBtnClickHandler: function() {
    alert(this.state.myValue);
  },
  render: function() {
    return (
      <div>
        <input
          className='test-input'
          value={this.state.myValue}
          onChange={this.onChangeHandler}
          placeholder='введите значение'
        />
        <button onClick={this.onBtnClickHandler}>Показать alert</button>
      </div>
    );
  }
});
```

Предлагаю добавить отступы для .test-input:

css/app.css

```
...
.test-input {
  margin: 0 5px 5px 0;
}
...
```

После добавления отступа в данном коде ничего не раздражает. Или нет? Как думаете, что здесь может расстроить борца за оптимизацию?

Ответ: каждый раз, после любого изменения у нас вызывается *setState*, а значит - полная перерисовка компонента. Не очень приятно. Опять же, чуть больше логики в момент создания компонента и в пору будет расстроиться от "отзывчивого" поля ввода.

Поэтому, наш выбор - это второй путь. Неконтролируемый компонент!

Uncontrolled Components (неконтролируемый компонент)

Главное отличие **неконтролируемого** компонента от **контролируемого** в том, что у него нет обработчика изменений, а значит нет постоянных вызовов `setState` и перерисовок.

Для того чтобы считать значение такого компонента используется вспомогательная функция вспомогательной библиотеки ReactDOM - `ReactDOM.findDOMNode`, а для того, чтобы можно было найти с помощью нее элемент, используется атрибут **ref**.

Для неконтролируемого компонента требуется указывать `defaultValue`.

Начнем по порядку:

1. Удалим обработчик `onChange`
2. Удалим `getInitialState`
3. Укажем `defaultValue` = пустая строка (`defaultValue=''`) вместо `value`
4. Добавим атрибут `ref`, назовем его `myTestInput`

```
...
<input
  className='test-input'
  defaultValue=''
  placeholder='введите значение'
  ref='myTestInput'
/>
...
```

Обновите страницу, попробуйте ввести значение. Работает? Работает!

Теперь нам нужно, научиться считывать значение: перепишите `onBtnClickHandler` следующим образом:

```
onBtnClickHandler: function() {
  alert(ReactDOM.findDOMNode(this.refs.myTestInput).value);
},
```

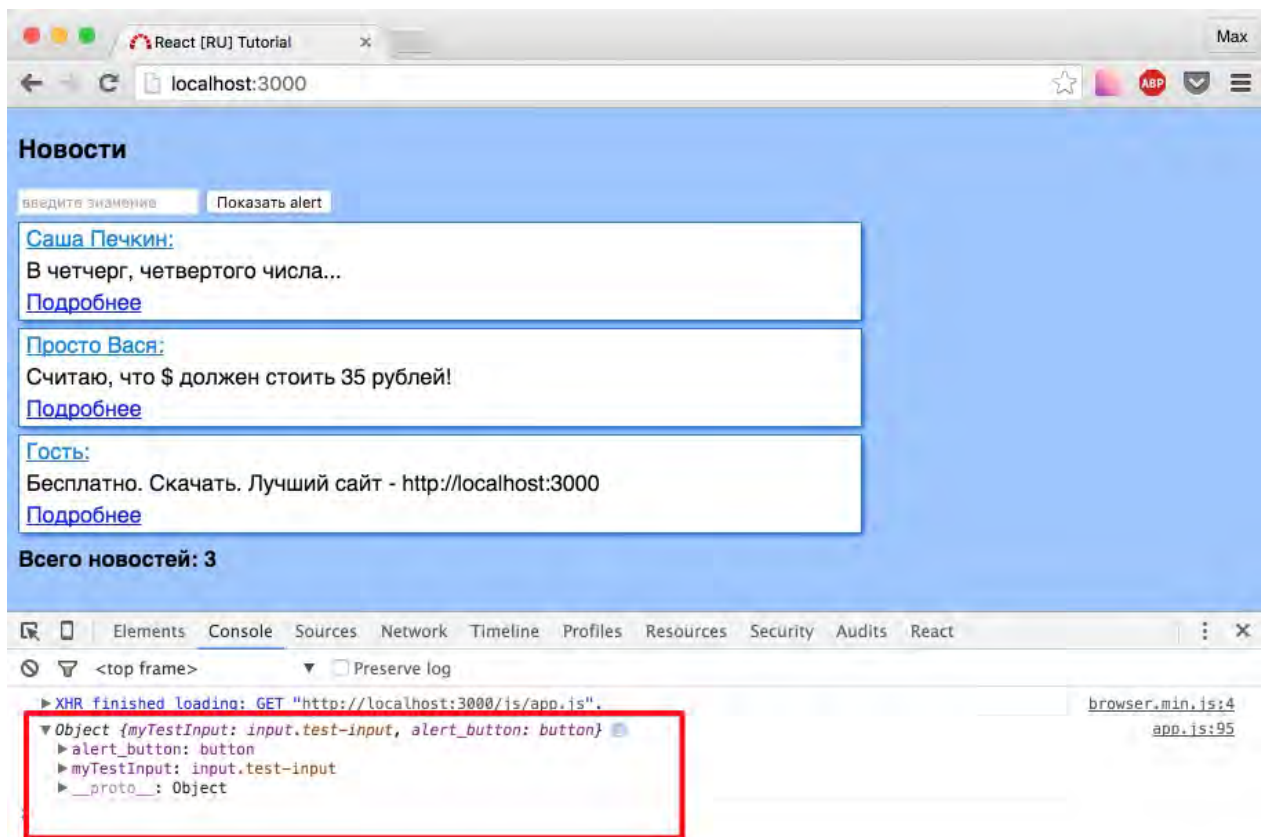
Метод `ReactDOM.findDomNode` принимает ссылку (`this.refs.МОЙ_ЭЛЕМЕНТ`) и возвращает **нативный** DOM элемент. Для тех кто в танке: `$('.my-input')` - возвращает jQuery обертку над элементом. У jQuery обертки обычно больше методов и свойств.

Напоследок, давайте поконсолим значения, посмотрим что к чему.

Добавьте атрибут *ref* для кнопки рядом с инпутом. А затем в обработчике *onBtnClickHandler* сконсольте *this.refs*

```
var TestInput = React.createClass({
  onBtnClickHandler: function() {
    console.log(this.refs);
    alert(ReactDOM.findDOMNode(this.refs.myTestInput).value);
  },
  render: function() {
    return (
      <div>
        <input
          className='test-input'
          defaultValue=''
          placeholder='введите значение'
          ref='myTestInput'
        />
        <button onClick={this.onBtnClickHandler} ref='alert_button'>Показать alert</button>
      </div>
    );
  }
});
```

В браузере:



Как видите, *this.refs* содержит все *refs* компонента. Если попробовать раскрыть свойства какого-нибудь из них - реакт ругнется. Общий посыл: не нужно лезть в DOM. Он прав, обычно такой необходимости нет. А для нашего случая как раз и созданы **refs**.

В качестве "итога" на этот раз, вам необходимо прочитать первые 3 абзаца из раздела **Uncontrolled Components**. Сейчас, все должно быть понятно!

За этот урок, мы научились с вами не вызывать дорогой *setState* и *render* на "каждый чих".

P.S. конечно, в данном случае никакого выигрыша в производительности нет. Оба подхода хорошо работают.

Вариант с контролируемыми и неконтролируемыми компонентами, работа с *defaultValue* и *state* являются одинаковыми для всех элементов форм.

Очень рекомендую посмотреть страницу документации на англ.языке по [элементам форм](#)

[Исходный код](#) на данный момент (включая *alert* и *console.log*).

Жизненный цикл компонента

Любимая фраза этого учебника "давайте представим задачу":

Мы отрисовали компонент, в котором есть `input`, и хотим чтобы фокус установился в него. Когда я первый раз подумал "как это сделать", даже не придумал что и ответить.

Хорошо, допустим я знаю, что могу достучаться до DOM элемента через `this.refs`, но в какой момент стучаться?

Какие вообще "моменты" есть?

Lifecycle methods

У каждого компонента, есть жизненный цикл (*lifecycle*): компонент будет примонтирован, компонент отрисовался, компонент будет удален и так далее...

У всех этих фаз есть методы, так называемые *lifecycle-methods*. Полный [список](#) в документации выглядит крайне просто, предлагаю вам в конце урока еще раз его посмотреть, а пока хватит информации и здесь:

- *componentWillMount* - компонент будет примонтирован. В данный момент у нас нет возможности посмотреть DOM элементы.
- *componentDidMount* - компонент примонтировался. В данный момент у нас есть возможность использовать *refs*, а следовательно это то самое место, где мы хотели бы указать установку фокуса. Так же, таймауты, аjax-запросы и взаимодействие с другими библиотеками стоит обрабатывать **здесь**.

Этот метод подходит для решения нашей задачи:

```
var TestInput = React.createClass({
  componentDidMount: function() { //ставим фокус в input
    ReactDOM.findDOMNode(this.refs.myTestInput).focus();
  },
  onBtnClickHandler: function() {
    console.log(this.refs);
    alert(ReactDOM.findDOMNode(this.refs.myTestInput).value);
  },
  render: function() {
    return (
      <div>
        <input
          className='test-input'
          defaultValue=''
          placeholder='введите значение'
          ref='myTestInput'
        />
        <button onClick={this.onBtnClickHandler} ref='alert_button'>Показать alert</button>
      </div>
    );
  }
});
```

Принцип прежний: мы находим DOM-узел, считываем его свойство / вызываем его нативный метод, в данном случае - вызываем метод `focus()`.

- *componentWillReceiveProps* - компонент получает новые *props*. Этот метод не вызывается в момент первого render'a. В официальной документации очень хороший пример, пожалуй скопирую его:

```
componentWillReceiveProps: function(nextProps) {
  this.setState({
    likesIncreasing: nextProps.likeCount > this.props.likeCount
  });
}
```

Обратите внимание: в этот момент, старые *props* доступны как *this.props*, а новые *props* доступны в виде **nextProps** аргумента функции.

Так же, если вы вызываете *setState* внутри этого метода - **не будет** вызван дополнительный *render*.

- *shouldComponentUpdate* - должен ли компонент обновиться? На самом деле, обычно реакт сам отлично разбирается. Но иногда ручное управление позволяет существенно ускорить работу в "узких местах". С этим методом нужно работать очень аккуратно.

- *componentWillUpdate* - вызывается прямо перед *render*, когда новые *props* и *state* получены. В этом методе нельзя вызывать *setState*.
- *componentDidUpdate* - вызывается сразу после *render*. Не вызывается в момент первого *render*'а компонента.
- *componentWillUnmount* - вызывается сразу перед тем, как компонент будет удален из DOM.

Конечно, в [документации](#) все описано немного подробнее. Я рекомендую с ней ознакомиться.

Главная мысль данного урока: у компонента есть стадии жизни, "в которые можно писать код". Да, пусть я выступаю здесь как "плохой программист", который советует вам писать свои велосипеды на разных стадиях жизни компонента, но именно таким образом вы быстро освоитесь.

Если же вы принадлежите к "правильному" типу программистов - пожалуйста, вот все lifecycle-методы. Выучите, перечитайте, осознайте - и пишите код без багов ;)

Итого: существует несколько lifecycle-методов, благодаря которым мы уже почти перестали "лазить" в DOM, а если и делаем это, то осознанно.

[Исходный код](#) на данный момент.

Работа с формой

В данном уроке мы превратим наш *input* в форму добавления новости. Научимся работать с чекбоксами, `disabled` атрибутом кнопки и прочими стандартными для такой задачи вещами.

Результатом добавления новости, пока что, вновь, будет *alert* с текстом новости.

Переименуйте `<TestInput />` в `<Add />`, и рендерите в нем следующую форму: автор (*input*), текст новости (*textarea*), "я согласен с правилами" (*checkbox*), "показать alert" (*button*).

Попутно изменим названия классов, удалим лишние обработчики и переместим компонент `<Add />` перед заголовком "Новости".

```
var Add = React.createClass({
  componentDidMount: function() {
    ReactDOM.findDOMNode(this.refs.author).focus();
  },
  onBtnClickHandler: function(e) {
    e.preventDefault();
  },
  render: function() {
    return (
      <form className='add cf'>
        <input
          type='text'
          className='add__author'
          defaultValue=''
          placeholder='Ваше имя'
          ref='author'
        />
        <textarea
          className='add__text'
          defaultValue=''
          placeholder='Текст новости'
          ref='text'
        ></textarea>
        <label className='add__checkrule'>
          <input type='checkbox' defaultChecked={false} ref='checkrule' />Я согласен с
правилами
        </label>
        <button
          className='add__btn'
          onClick={this.onBtnClickHandler}
          ref='alert_button'>
          Показать alert
        </button>
      </form>
    );
  }
});
```

Если вы не против моего оформления, можете взять стили для компонента `<Add />` :

```

.add {
  margin: 0 5px 5px 0;
  width: 210px;
  border: 1px dashed rgba(0, 89, 181, 0.82);
  padding: 5px;
}
.add__author, .add__text, .add__btn, .add__checkrule {
  display: block;
  margin: 0 0 5px 0;
  padding: 5px;
  width: 94%;
  border: 1px solid rgba(0, 89, 181, 0.82);
}
.add__checkrule {
  border: none;
  font-size: 12px;
}
.add__btn {
  box-sizing: content-box;
  color: #FFF;
  text-transform: uppercase;
  background: #007DDC;
}
.add__btn:disabled {
  background: #CCC;
  color: #999;
}

```

Отключим кнопку "показать alert", если не отмечен чекбокс. Здесь есть 2 варианта - использовать *state* или не использовать. Для нашей задачи никаких проблем с производительностью не будет, если мы будем использовать *state*. Так даже лучше, чем "лезть в DOM". Но я все же приведу оба решения на всякий случай.

Решение 1: без использования *state*.

Добавьте инпуту обработчик *onChange*, и добавьте функцию обработчик.

```

...
onCheckRuleClick: function(e) {
  ReactDOM.findDOMNode(this.refs.alert_button).disabled = !e.target.checked;
},
...
<label className='add__checkrule'>
  <input type='checkbox' defaultChecked={false} ref='checkrule' onChange={this.onCheckRuleClick}/>Я согласен с правилами
</label>
...

```

Проверьте в браузере. При первой загрузке, кнопка активна, хотя чекбокс не стоит. Хех, решается на *HTML* ;) Добавьте атрибут *disabled* кнопке.

```
<button
  className='add__btn'
  onClick={this.onBtnClickHandler}
  ref='alert_button'
  disabled>
  Показать alert
</button>
```

Решение 2: с использованием *state*

- добавьте *getInitialState* компоненту `<Add />` ;
- удалите *defaultChecked* из инпута;
- сделайте атрибут *disabled* у кнопки равным значению из *state*;
- измените функцию обработчик;

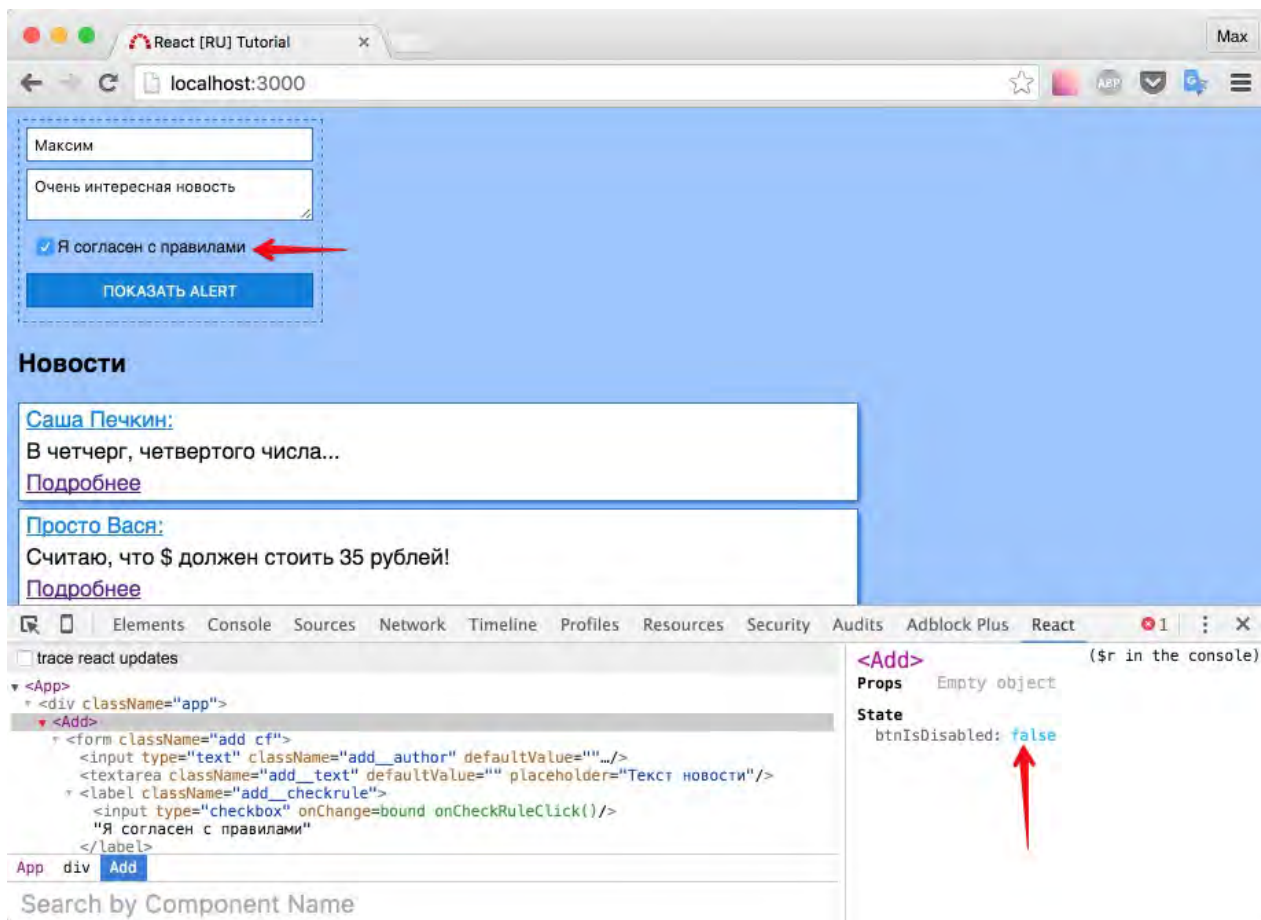
Приведу полный код компонента `<Add />`

```

var Add = React.createClass({
  getInitialState: function() { //устанавливаем начальное состояние (state)
    return {
      btnIsDisabled: true
    };
  },
  componentDidMount: function() {
    ReactDOM.findDOMNode(this.refs.author).focus();
  },
  onBtnClickHandler: function(e) {
    e.preventDefault();
  },
  onCheckRuleClick: function(e) {
    this.setState({btnIsDisabled: !this.state.btnIsDisabled}); //устанавливаем значени
е в state
  },
  render: function() {
    return (
      <form className='add cf'>
        <input
          type='text'
          className='add__author'
          defaultValue=''
          placeholder='Ваше имя'
          ref='author'
        />
        <textarea
          className='add__text'
          defaultValue=''
          placeholder='Текст новости'
          ref='text'
        ></textarea>
        <label className='add__checkrule'>
          <input type='checkbox' ref='checkrule' onChange={this.onCheckRuleClick}/>Я с
огласен с правилами
        </label>

        { /* берем значение для disabled атрибута из state */}
        <button
          className='add__btn'
          onClick={this.onBtnClickHandler}
          ref='alert_button'
          disabled={this.state.btnIsDisabled}
        >
          Показать alert
        </button>
      </form>
    );
  }
});

```



Какое решение выбрать?

В данном случае я за второй вариант. Для меня большим злом является необходимость работать с DOM, чем "рендер" компонента `<Add />` на каждый клик по чекбоксу. В данный момент я могу гарантировать, что никакой проблемы с производительностью не будет. Но заметьте, для `input`'а я все же оставил по прежнему схему работы через DOM. Путано? Я руководствовался тем, что клик по чекбоксу - событие не частое, а возможно единичное. `<input />` и `<textarea></textarea>` - более вредные в моем понимании. События `onChange` происходят в них слишком часто.

Работайте как вам нравится, просто знайте, что есть два варианта.

Для добавления новости нам осталось сформировать текст, который будет показываться в `alert`. Я думаю, эта задача вам точно под силу. Решение ниже.

Решение:

Достаточно всего лишь подкорректировать функцию `onBtnClickHandler`

```
onBtnClickHandler: function(e) {  
  e.preventDefault();  
  var author = ReactDOM.findDOMNode(this.refs.author).value;  
  var text = ReactDOM.findDOMNode(this.refs.text).value;  
  alert(author + '\n' + text);  
},
```

Блокировка кнопки, если не все поля заполнены

Обычно требуется блокировать кнопку так же в случае каких-то проблем в валидации данных. Самая первая из них - если необходимые поля пустые - не отправлять форму.

Представим, что у нас оба поля являются обязательными. Как бы решалась такая задача без react? Вероятно у нас была бы функция *validate*, которая вызывалась бы на каждое изменение в проверяемых полях. Нужно было бы генерировать и прослушивать событие...

Думаю, вы поняли намек. Здесь без *state* не обойтись, и это точно то место, где следует использовать именно **состояние**.

Попробуйте сами, а потом сверьтесь с решением.

Задача: если в поле "автор" или "текст" не введено ничего (либо пробелы) - кнопка "показать alert" должна быть недоступной.

Подсказка **#1:** для удаления пробелов используйте стандартный метод `trim()`

Подсказка **#2:** вам потребуется больше переменных, так же лучше переименовать переменную *btnIsDisabled*...

```
...  
getInitialState: function() { //устанавливаем начальное состояние (state)  
  return {  
    agreeNotChecked: true,  
    authorIsEmpty: true,  
    textIsEmpty: true  
  };  
},  
...
```

Подсказка **#3:** объявите обработчики на *onChange*


```

...
onAuthorChange: function(e) {
  if (e.target.value.trim().length > 0) {
    this.setState({authorIsEmpty: false})
  } else {
    this.setState({authorIsEmpty: true})
  }
},
onTextChange: function(e) {
  if (e.target.value.trim().length > 0) {
    this.setState({textIsEmpty: false})
  } else {
    this.setState({textIsEmpty: true})
  }
},
...
<input
  type='text'
  className='add__author'
  onChange={this.onAuthorChange}
  placeholder='Ваше имя'
  ref='author'
/>
<textarea
  className='add__text'
  onChange={this.onTextChange}
  placeholder='Текст новости'
  ref='text'
></textarea>
...

```

Если вас смущает дублирование кода - спокойствие, позже порефакторим.

Подсказка **#4**: у атрибута `disabled` проверяйте выполнение условия: *если хотя бы одно из свойств состояния (`agreeNotChecked`, `authorIsEmpty`, `textIsEmpty`) имеет значение `true` - кнопка выключается.*

```

<button
  className='add__btn'
  onClick={this.onBtnClickHandler}
  ref='alert_button'
  disabled={agreeNotChecked || authorIsEmpty || textIsEmpty}
>
  Показать alert
</button>

```

Решение: код компонента `<Add />` полностью:

```

var Add = React.createClass({

```

```

getInitialState: function() { //устанавливаем начальное состояние (state)
  return {
    agreeNotChecked: true,
    authorIsEmpty: true,
    textIsEmpty: true
  };
},
componentDidMount: function() {
  ReactDOM.findDOMNode(this.refs.author).focus();
},
onBtnClickHandler: function(e) {
  e.preventDefault();
  var author = ReactDOM.findDOMNode(this.refs.author).value;
  var text = ReactDOM.findDOMNode(this.refs.text).value;
  alert(author + '\n' + text);
},
onCheckRuleClick: function(e) {
  this.setState({agreeNotChecked: !this.state.agreeNotChecked}); //устанавливаем зна
чение в state
},
onAuthorChange: function(e) {
  if (e.target.value.trim().length > 0) {
    this.setState({authorIsEmpty: false})
  } else {
    this.setState({authorIsEmpty: true})
  }
},
onTextChange: function(e) {
  if (e.target.value.trim().length > 0) {
    this.setState({textIsEmpty: false})
  } else {
    this.setState({textIsEmpty: true})
  }
},
render: function() {
  var agreeNotChecked = this.state.agreeNotChecked,
      authorIsEmpty = this.state.authorIsEmpty,
      textIsEmpty = this.state.textIsEmpty;
  return (
    <form className='add cf'>
      <input
        type='text'
        className='add__author'
        onChange={this.onAuthorChange}
        placeholder='Ваше имя'
        ref='author'
      />
      <textarea
        className='add__text'
        onChange={this.onTextChange}
        placeholder='Текст новости'
        ref='text'
      ></textarea>
    </form>
  );
}

```

```

    <label className='add__checkrule'>
      <input type='checkbox' ref='checkrule' onChange={this.onCheckRuleClick}/>Я с
огласен с правилами
    </label>

    <button
      className='add__btn'
      onClick={this.onBtnClickHandler}
      ref='alert_button'
      disabled={agreeNotChecked || authorIsEmpty || textIsEmpty}
    >
      Показать alert
    </button>
  </form>
);
}
});

```

Избавляемся от дублирования кода

Дублирование кода в функциях *onAuthorChange* и *onTextChange* - это плохо. Хорошо было бы создать одну функцию, которая принимала бы аргумент - *fieldName* и изменяла бы соответствующую переменную в *state* согласно нашей логике.

Как передать аргумент в функцию? Что насчет такого варианта:

```

<input
  type='text'
  className='add__author'
  onChange={this.onFieldChange('имя_переменной')}
  placeholder='Ваше имя'
  ref='author'
/>

```

Откровенно плохой вариант. Функция сразу же выполнится - так как указаны ().

Нам нужно передать именно функцию, а не **результат ее выполнения**. На помощь приходит метод `bind()`.

Верной строкой для *onChange* в таком случае будет:

Для input'a:

```

onChange={this.onFieldChange.bind(this, 'authorIsEmpty')}

```

Для textarea:

```
onChange={this.onChange.bind(this, 'textIsEmpty')}
```

Напишем саму функцию *onFieldChange*:

```
...
onFieldChange: function(fieldName, e) {
  if (e.target.value.trim().length > 0) {
    this.setState({[''+fieldName]:false})
  } else {
    this.setState({[''+fieldName]:true})
  }
},
...
```

Я надеюсь, не возникает вопросов, откуда взялся аргумент *fieldName*, почему аргумент 'e' теперь второй? (если что, это все *bind* виноват, он "прокинул" нашу переменную в функцию первым аргументом, можно было прокинуть еще...)

Так как в переменной у нас строка, мы не можем передать ее напрямую в *setState* в качестве названия поля объекта.

Возможно вам покажется более читаемым такой вариант:

```
...
onFieldChange: function(fieldName, e) {
  var next = {};
  if (e.target.value.trim().length > 0) {
    next[fieldName] = false;
    this.setState(next);
  } else {
    next[fieldName] = true;
    this.setState(next);
  }
},
...
```

Разницы между ними нет, в этом "противостоянии" я выбираю первый вариант.

Итого: мы разобрали пару стандартных способов блокировать нажатие кнопки на форме. Мы опять не взяли событие *onSubmit* для формы и ограничились лишь событием *onClick*.

По традиции - [исходный код](#) на данный момент.

Добавить новость

Что такое добавление новости?

1. Это форма, в которую мы вводим необходимые данные.
2. Это "лента новостей", которая отображает наши данные.

После создания новости должно генерироваться событие, на которое должна быть подписана новостная лента. Иначе, как лента узнает о том, что нужно показать новую новость?

Очевидно, что нам нужна какая-то "система событий", чтобы научить компонент `<Add />` генерировать событие, а компонент `<News />` отображать. Так как компонент `<News />` имеет статичный атрибут `data`, было бы логично предположить, что в `data` нужно "скидывать" динамически сформированную переменную. Конечно же, это место где нужно использовать `state`.

Подытожим:

1. компонент `<App />` должен иметь переменную в `state` - "новости", которую передавать в компонент `<News />`.
2. Компонент `<App />` должен уметь слушать событие "добавлена новость".
3. Компонент `<App />` так же должен уметь отписываться от прослушивания события.

```
var App = React.createClass({
  getInitialState: function() {
    return {
      news: my_news
    };
  },
  componentDidMount: function() {
    /* Слушай событие "Создана новость"
       если событие произошло, обнови this.state.news
    */
  },
  componentWillUnmount: function() {
    /* Больше не слушай событие "Создана новость" */
  },
  render: function() {
    console.log('render');
    return (
      <div className='app'>
        <Add />
        <h3>Новости</h3>
        <News data={this.state.news} />
      </div>
    );
  }
});
```

Мы добавили переменную в *state* (с начальным состоянием новостей - массивом `my_news`), ее же стали передавать в качестве свойств, для компонента `<News />`. А также, "как будто подписались" на прослушивание события в момент примонтирования компонента, и "как будто отписались" в момент "перед удалением компонента". В нашем примере, компонент `<App />` не может быть удален, но тем не менее "не забывать отписываться" - хорошая практика.

На втором берегу, у нас компонент `<Add />`, который должен уметь генерировать событие в обработчике `onBtnClickHandler`.

Кстати, переименуйте кнопку "показать alert" -> "Добавить новость", так как мы уже почти готовы!

Глобальная система событий

Напомню, у нас возник вопрос о необходимости взаимодействия двух компонентов. Эти компоненты не состоят в отношении родитель-потомок.

Предлагаю воспользоваться решением [EventEmitter](#), для этого скачайте/добавьте библиотеку ([.min версия](#)) в `index.html`, перед `app.js`.

index.html

```

<!DOCTYPE html>
<html>
  <head>
    <title>React [RU] Tutorial</title>
    <link rel="stylesheet" href="css/app.css">
  </head>
  <body>
    <div id="root"></div>

    <script src="js/react/react.js"></script>
    <script src="js/react/react-dom.js"></script>
    <script src="js/react/browser.min.js"></script>
    <script src="js/EventEmitter.js"></script>
    <script type="text/babel" src="js/app.js"></script>
  </body>
</html>

```

Теперь мы можем в *app.js* добавить глобальную переменную:

```

...
window.ee = new EventEmitter();
...

```

Благодаря которой, можем генерировать событие в обработчике *onBtnClickHandler* компонента `<Add />`

```

...
onBtnClickHandler: function(e) {
  e.preventDefault();
  var author = ReactDOM.findDOMNode(this.refs.author).value;
  var text = ReactDOM.findDOMNode(this.refs.text).value;

  var item = [{
    author: author,
    text: text,
    bigText: '...'
  }];

  window.ee.emit('News.add', item);
},
...

```

`window.ee.emit('News.add', item);` = сгенерируй событие 'News.add' и передай в качестве данных - *item*.

И наконец, благодаря *window.ee* мы можем подписываться/отписываться в `<App />` :


```
...
componentDidMount: function() {
  var self = this;
  window.ee.addListener('News.add', function(item) {
    var nextNews = item.concat(self.state.news);
    self.setState({news: nextNews});
  });
},
componentWillUnmount: function() {
  window.ee.removeListener('News.add');
},
...
```

`window.ee.addListener` - принимает в качестве аргументов имя события и функцию-обработчик. Чтобы внутри функции-обработчика (*callback*) использовать `this` - мы сохранили его чуть выше в переменную *self*.

Интересный момент: `var nextNews = item.concat(self.state.news);`

Мы создали **новый** массив, в котором первым элементом поставили новую новость, чтобы она была верхней в списке.

Кстати, было бы неплохо частично очищать форму после добавления новости: а именно, удалять текст новости, но оставлять "чекбокс" и автора. Внесите эти изменения в `onBtnClickHandler`:

```
...
onBtnClickHandler: function(e) {
  e.preventDefault();
  var textEl = ReactDOM.findDOMNode(this.refs.text);

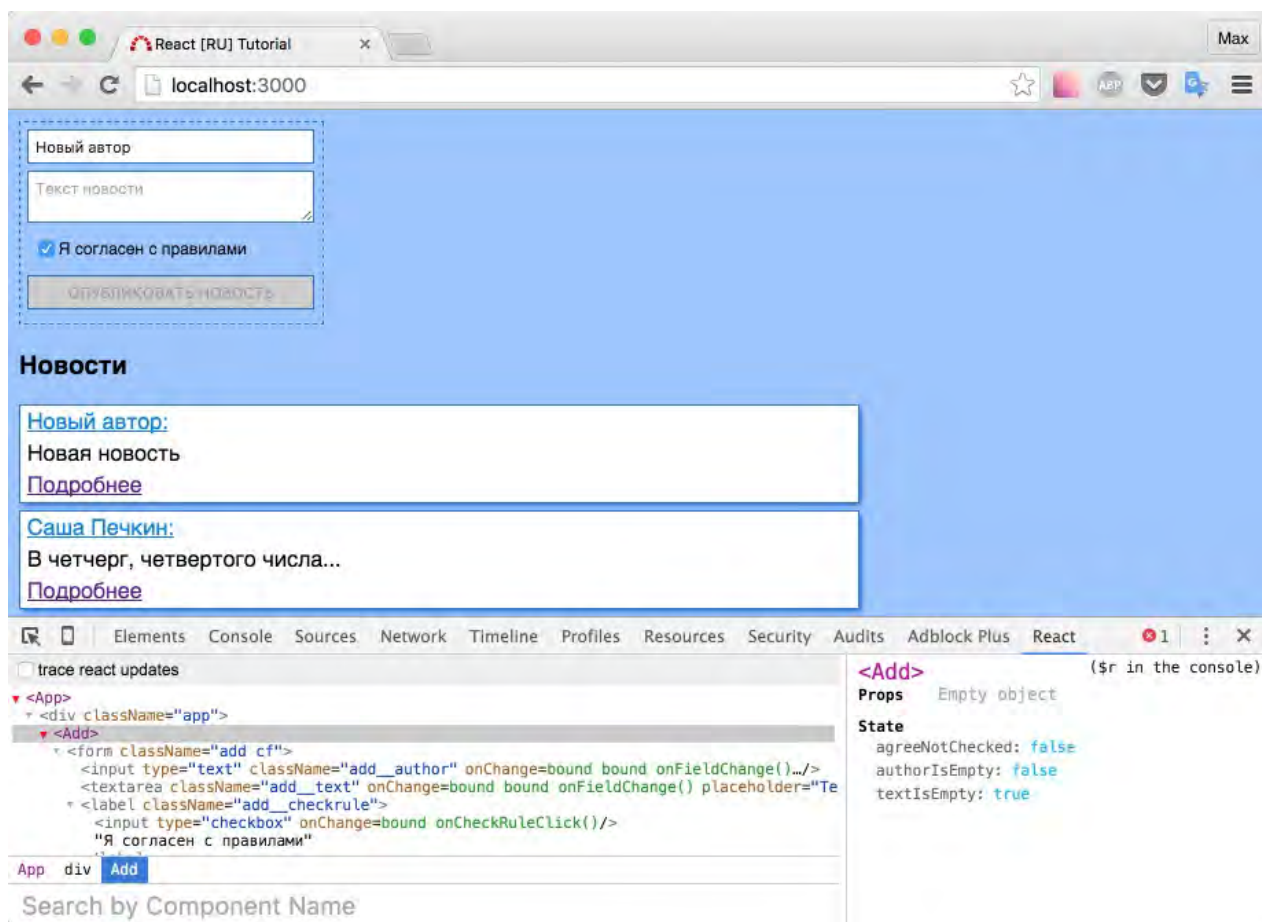
  var author = ReactDOM.findDOMNode(this.refs.author).value;
  var text = textEl.value;

  var item = [{
    author: author,
    text: text,
    bigText: '...'
  }];

  window.ee.emit('News.add', item);

  textEl.value = '';
  this.setState({textIsEmpty: true});
},
...
```

Удобно, что кнопка "дизейблится" (*disable*) после очистки.



Кхм, итоговый сценарий:

`js/app.js`

```

'use strict';

var my_news = [
  {
    author: 'Саша Печкин',
    text: 'В четчерг, четвертого числа...',
    bigText: 'в четыре с четвертью часа четыре чёрненьких чумазеньких чертёнка чертили чёрными чернилами чертёж.'
  },
  {
    author: 'Просто Вася',
    text: 'Считаю, что $ должен стоять 35 рублей!',
    bigText: 'А евро 42!'
  },
  {
    author: 'Гость',
    text: 'Бесплатно. Скачать. Лучший сайт - http://localhost:3000',
    bigText: 'На самом деле платно, просто нужно прочитать очень длинное лицензионное соглашение'
  }
];

window.ee = new EventEmitter();

```

```
var Article = React.createClass({
  propTypes: {
    data: React.PropTypes.shape({
      author: React.PropTypes.string.isRequired,
      text: React.PropTypes.string.isRequired,
      bigText: React.PropTypes.string.isRequired
    })
  },
  getInitialState: function() {
    return {
      visible: false
    };
  },
  readmoreClick: function(e) {
    e.preventDefault();
    this.setState({visible: true});
  },
  render: function() {
    var author = this.props.data.author,
        text = this.props.data.text,
        bigText = this.props.data.bigText,
        visible = this.state.visible;

    return (
      <div className='article'>
        <p className='news__author'>{author}</p>
        <p className='news__text'>{text}</p>
        <a href="#"
          onClick={this.readmoreClick}
          className={'news__readmore ' + (visible ? 'none': '')}>
          Подробнее
        </a>
        <p className={'news__big-text ' + (visible ? '' : 'none')}>{bigText}</p>
      </div>
    )
  }
});

var News = React.createClass({
  propTypes: {
    data: React.PropTypes.array.isRequired
  },
  getInitialState: function() {
    return {
      counter: 0
    }
  },
  render: function() {
    var data = this.props.data;
    var newsTemplate;

    if (data.length > 0) {
```

```

newsTemplate = data.map(function(item, index) {
  return (
    <div key={index}>
      <Article data={item} />
    </div>
  )
})
} else {
  newsTemplate = <p>К сожалению новостей нет</p>
}

return (
  <div className='news'>
    {newsTemplate}
    <strong
      className={'news__count ' + (data.length > 0 ? '':'none')} >Всего новостей: {
data.length}</strong>
    </div>
  );
}
});

var Add = React.createClass({
  getInitialState: function() {
    return {
      agreeNotChecked: true,
      authorIsEmpty: true,
      textIsEmpty: true
    };
  },
  componentDidMount: function() {
    ReactDOM.findDOMNode(this.refs.author).focus();
  },
  onBtnClickHandler: function(e) {
    e.preventDefault();
    var textEl = ReactDOM.findDOMNode(this.refs.text);

    var author = ReactDOM.findDOMNode(this.refs.author).value;
    var text = textEl.value;

    var item = [{
      author: author,
      text: text,
      bigText: '...'
    }];

    window.ee.emit('News.add', item);

    textEl.value = '';
    this.setState({textIsEmpty: true});
  },
  onCheckRuleClick: function(e) {
    this.setState({agreeNotChecked: !this.state.agreeNotChecked});
  },
});

```

```

    },
    onFieldChange: function(fieldName, e) {
      if (e.target.value.trim().length > 0) {
        this.setState({[''+fieldName]:false})
      } else {
        this.setState({[''+fieldName]:true})
      }
    },
    render: function() {
      var agreeNotChecked = this.state.agreeNotChecked,
          authorIsEmpty = this.state.authorIsEmpty,
          textIsEmpty = this.state.textIsEmpty;
      return (
        <form className='add cf'>
          <input
            type='text'
            className='add__author'
            onChange={this.onFieldChange.bind(this, 'authorIsEmpty')}
            placeholder='Ваше имя'
            ref='author'
          />
          <textarea
            className='add__text'
            onChange={this.onFieldChange.bind(this, 'textIsEmpty')}
            placeholder='Текст новости'
            ref='text'
          ></textarea>
          <label className='add__checkrule'>
            <input type='checkbox' ref='checkrule' onChange={this.onCheckRuleClick}/>Я с
огласен с правилами
          </label>

          <button
            className='add__btn'
            onClick={this.onBtnClickHandler}
            ref='alert_button'
            disabled={agreeNotChecked || authorIsEmpty || textIsEmpty}
          >
            Опубликовать новость
          </button>
        </form>
      );
    }
  });

var App = React.createClass({
  getInitialState: function() {
    return {
      news: my_news
    };
  },
  componentDidMount: function() {
    var self = this;

```

```
window.ee.addListener('News.add', function(item) {
  var nextNews = item.concat(self.state.news);
  self.setState({news: nextNews});
});
},
componentWillUnmount: function() {
  window.ee.removeListener('News.add');
},
render: function() {
  console.log('render');
  return (
    <div className='app'>
      <Add />
      <h3>Новости</h3>
      <News data={this.state.news} />
    </div>
  );
}
});

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

Ужасно длинный файл. Так хочется разбить его на кусочки, использовать ES6 синтаксис, модули, сжимать его в конце-концов...

Не торопите события! На данный момент было важно показать вам именно работу с реактом, как с *"просто еще одной библиотекой"*. Организация кода "на современный лад" - входит в мои планы. Быть может, когда вы будете читать этот текст, соответствующая глава уже будет написана.

Полезная ссылка по вопросу организации взаимодействия между компонентами <http://stackoverflow.com/questions/21285923/reactjs-two-components-communicating/31563614#31563614>

Итого: Мы научили компоненты совместной работе. Посмотрели на реализацию *EventEmitter* для браузера.

Данный урок хорош тем, что он показывает подход к реализации глобальной системы событий в React.js

Мне очень симпатичен *Redux*, который элегантно решает эту проблему. По нему тоже есть [подробный tutorial на русском](#). Рекомендую к изучению.

[Исходный код](#) на данный момент.

