Учебный курс по React

Перевод опубликован в блоге компании RUVDS (оригинал).

Читать онлайн (много полезных комментариев):

- Часть 1: обзор курса, причины популярности React, ReactDOM и JSX
- Часть 2: функциональные компоненты
- Часть 3: файлы компонентов, структура проектов
- Часть 4: родительские и дочерние компоненты
- Часть 5: начало работы над TODO-приложением, основы стилизации
- Часть 6: о некоторых особенностях курса, JSX и JavaScript
- Часть 7: встроенные стили
- Часть 8: продолжение работы над TODO-приложением, знакомство со свойствами компонентов
- Часть 9: свойства компонентов
- Часть 10: практикум по работе со свойствами компонентов и стилизации
- Часть 11: динамическое формирование разметки и метод массивов map
- Часть 12: практикум, третий этап работы над TODO-приложением
- Часть 13: компоненты, основанные на классах
- Часть 14: практикум по компонентам, основанным на классах, состояние компонентов
- Часть 15: практикумы по работе с состоянием компонентов
- Часть 16: четвёртый этап работы над TODO-приложением, обработка событий
- Часть 17: пятый этап работы над TODO-приложением, модификация состояния компонентов
- Часть 18: шестой этап работы над TODO-приложением
- Часть 19: методы жизненного цикла компонентов
- Часть 20: первое занятие по условному рендерингу
- Часть 21: второе занятие и практикум по условному рендерингу
- Часть 22: седьмой этап работы над TODO-приложением, загрузка данных из внешних источников
- Часть 23: первое занятие по работе с формами
- Часть 24: второе занятие по работе с формами
- Часть 25: практикум по работе с формами
- Часть 26: архитектура приложений, паттерн Container/Component
- Часть 27: курсовой проект
- Часть 28: современные возможности React, идеи проектов, заключение

Две полезных кликабельных картинки для наших читателей :)

Оглавление

Учебный курс по React, часть 1: обзор курса, причины популярности React, ReactDOM и JSX

Занятие 1. Обзор курса и рекомендации по его освоению

Оригинал

Добро пожаловать на курс «Основы React». Меня зовут Боб Зиролл, я расскажу вам о том, как создавать фронтенд-проекты, используя один из самых популярных в мире веб-фреймворков. Я работаю в области компьютерного образования уже много лет, в частности, сейчас руководжу организацией учебного процесса в V School.

О процессе освоения курса
За годы разработки учебных курсов, направленных на то, чтобы помочь всем желающим быстро осваивать сложные вещи, я разработал собственный подход к обучению, о котором, думаю, полезно будет рассказать.

Для начала хочу отметить, что самый лёгкий и результативный способ изучить что угодно заключается в том, чтобы не жалеть сил и времени на практику. Если вы хотите научиться программировать — то чем раньше вы начнёте делать что-то сами, и чем чаще будете это делать — тем выше ваши шансы на успех.

Обычно, когда я ввожу в курс дела учащихся V School, я привожу им следующий пример из собственной жизни. Недавно меня потянуло на работу с деревом. Я читал книги, смотрел бесчисленные видео на YouTube, мне дарили инструменты. Но я не мог сделать ничего достойного до тех пор, пока не взял инструменты в руки. Только многие часы, потраченные на работу пилой и наждачной бумагой, на склеивание и свинчивание деталей, позволили мне приблизиться к цели. Собственно говоря, по такому же принципу устроено и освоение всего, чего угодно. Хотите изучить React? Пишите код.

Большинство занятий этого курса содержат упражнения. Ожидается, что вы постараетесь выполнять их самостоятельно. Если же вы, ознакомившись с заданием для самостоятельной работы, тут же перейдёте к описанию его решения, то вы, на самом деле, выберете самый сложный способ изучения React. Кроме того, не ждите, пока вам предложат попрактиковаться — берите инициативу на себя и пробуйте всё, о чём узнаёте. Старайтесь как можно больше самостоятельно работать с кодом. В частности, когда вы уже немного освоите React — создавайте нечто такое, что вам интересно, испытывайте всё, что вам любопытно испытать. Это позволит вам избежать такой неприятности, как «tutorial hell».

Ещё один важный момент моего подхода заключается в интервальном обучении и в повторении пройденного материала. Это — важнейшие вещи, которые позволяют по-настоящему запомнить то, чему вы учитесь. Не рекомендуется с головой бросаться в изучение курса. Это — путь в тот самый «tutorial hell». При таком подходе возникает ощущение, как будто вы и правда что-то узнали, а в реальности же вы просто не в состоянии запомнить то, что «изучили».

Поэтому, продвигаясь по материалам, делайте частые перерывы. Речь идёт не о периодических перерывах в 5-10 минут, а кое о чём более масштабном. Изучите пару принципов, попрактикуйтесь в их использовании, а затем денёк передохните. Когда вы вернётесь к курсу, будет очень полезно повторить уже изученные материалы. Конечно, при таком подходе на то, чтобы освоить курс, уйдёт больше времени, но это чрезвычайно благотворно скажется на вашем обучении.

Теперь давайте в двух словах обсудим то, чему вы научитесь, освоив этот курс.

Состав курса и предварительные требования
Вот перечень основных тем курса:

- Компоненты. Говоря о React, нельзя избежать обсуждения концепции компонентов. Компоненты в React — это основной строительный блок для создания фрагментов HTML-кода, подходящих для повторного использования. И практически всё остальное, о чём мы будем говорить, имеет отношение к тому, как использовать эти компоненты для построения веб-приложений.
- JSX. Это — синтаксическое расширение JavaScript, которое позволяет создавать компоненты, используя возможности HTML и JavaScript.
- Стилизация компонентов. Стилизация позволяет придать компонентам привлекательный внешний вид.
- Свойства и обмен данными в приложении. Свойства используются для передачи данных между компонентами.
- Состояние. Механизмы состояния компонентов используются для хранения данных в приложении и для управления ими.
- Обработка событий. События позволяют наладить интерактивные взаимоотношения с пользователями приложений.
- Методы жизненного цикла компонентов. Эти методы позволяют программисту влиять на различные события, происходящие с компонентами.
- Загрузка данных из внешних API с использованием протокола HTTP.
- Работа с формами.

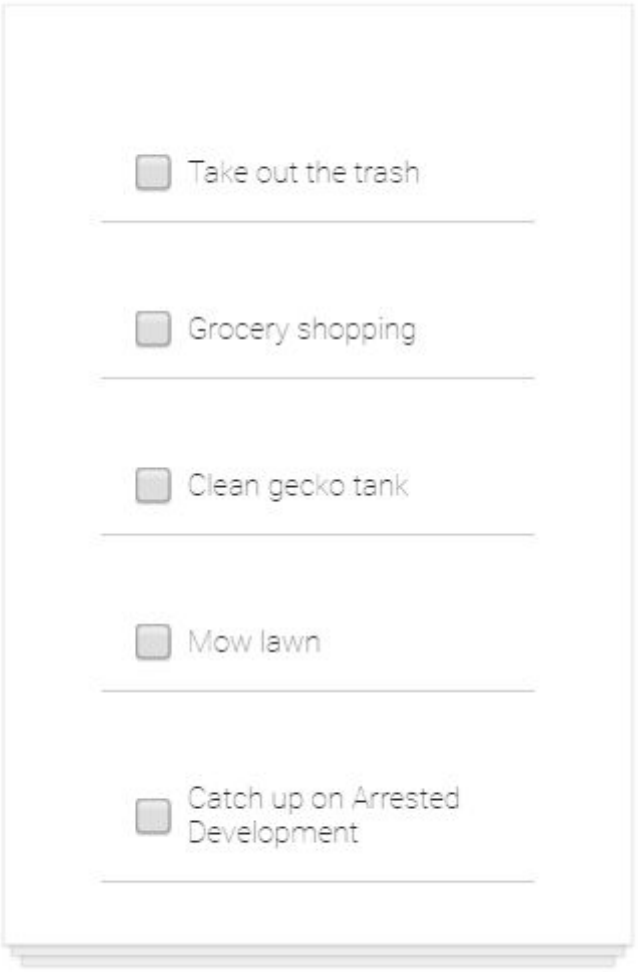
Для того чтобы продуктивно заниматься по этому курсу, вам нужно знать HTML, CSS и JavaScript (ES6).

Занятие 2. Учебные проекты

Оригинал

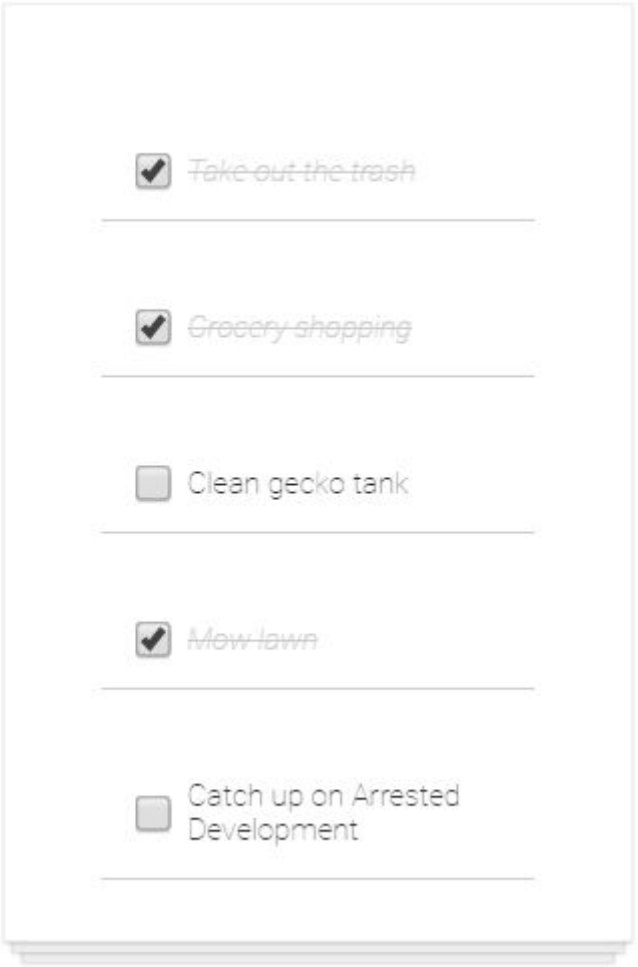
В процессе прохождения этого курса вы будете разрабатывать учебные проекты. Взглянем на некоторые из них.

Нашей первой разработкой будет стандартное TODO-приложение.



TODO-приложение

Может и выглядит оно скучновато, но в ходе его разработки будет задействовано множество возможностей, о которых мы будем говорить в курсе. По элементам списка дел можно будет щелкать, отмечая их как завершённые, и видеть, как меняется их внешний вид.



Отмеченные дела в TODO-приложении

А вот — наш курсовой проект — генератор мемов.

Генератор мемов

При работе с этим приложением в поля `Top Text` и `Bottom Text` вводят тексты, которые, соответственно, попадут в верхнюю и нижнюю часть изображения. По нажатию на кнопку `Gen` программа случайным образом выбирает изображение мема из соответствующего API и добавляет к нему текст. Вот пример работы этого приложения

Готовый мем

Занятие 3. Зачем нужен React и почему его стоит изучать?

[Оригинал]

Зачем использовать React, если можно разработать веб-проект на чистом JavaScript? Если вы интересуетесь веб-разработкой, то, возможно, слышали о том, что React позволяет создавать очень быстрые приложения, производительность которых превышает то, что достижимо с использованием лишь JavaScript. Это достигается за счёт использования в React технологии, называемой Virtual DOM. Мы не будем вдаваться в подробности о Virtual DOM, если вы хотите познакомиться с этой технологией поближе, можете посмотреть это видео.

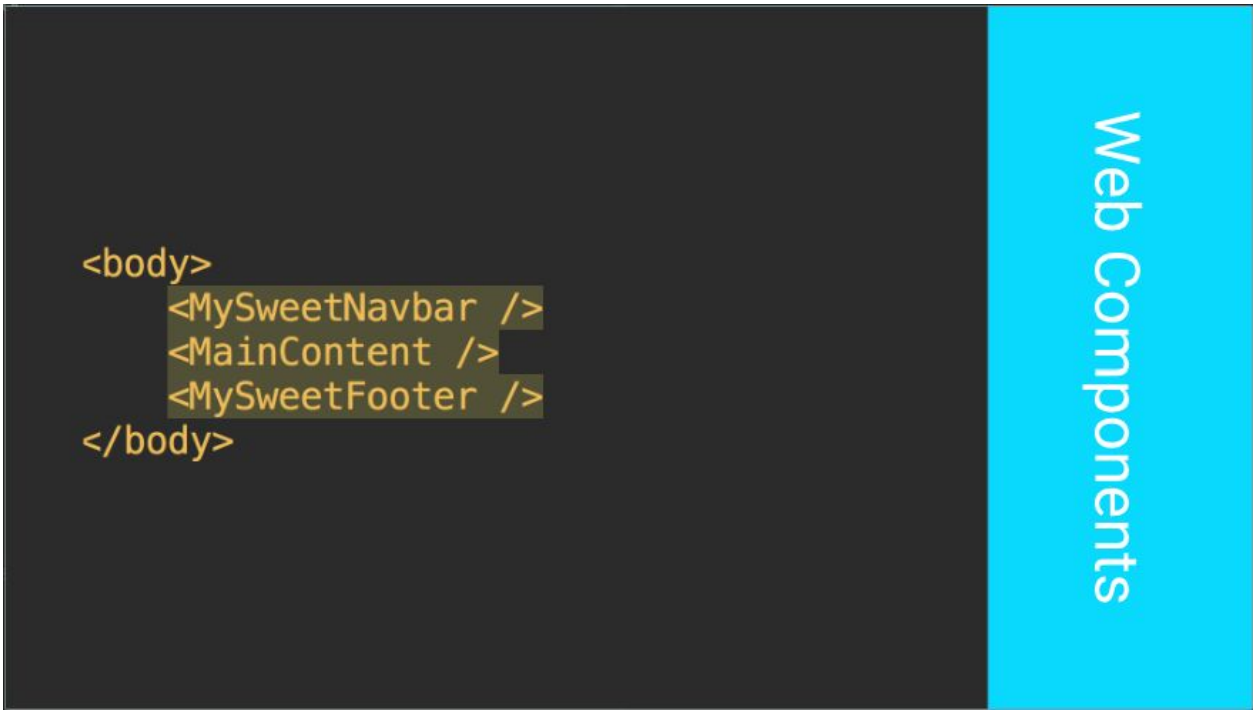
Сейчас нам достаточно знать о том, что Virtual DOM помогает веб-приложениям работать гораздо быстрее, чем если бы при их разработки использовался обычный JS. Ещё одно по-настоящему замечательное преимущество, которое даёт нам React — это возможность создавать веб-компоненты, подходящие для повторного использования. Рассмотрим пример.

У нас имеется стандартный элемент `navbar` (навигационная панель) из библиотеки Bootstrap.

Навигационная панель

Если вы раньше не пользовались Bootstrap, то знайте, что это просто CSS-библиотека, которая даёт веб-разработчику красиво оформленные элементы. Тут примерно четыре десятка строк кода, всё это выглядит довольно громоздко, ориентироваться в таком коде непросто. Если включить всё это в состав HTML-страницы, на которой и так имеется много всего, код такой страницы окажется попросту перегруженным различными конструкциями.

Веб-компоненты React позволяют брать фрагменты HTML-кода, оформлять их в виде самостоятельных компонентов, и, вместо того, чтобы добавлять на страницу эти фрагменты, включать в состав страниц нечто вроде особых HTML-тегов, указывающих на них. В нашем случае, вместо того, чтобы добавлять на страницу сорок строк HTML-разметки, достаточно включить в её состав компонент, содержащий эту разметку. У нас он называется `MySweetNavbar`.



Компонентный подход к формированию веб-страниц

Назвать такой компонент можно как угодно. Как видно, разметку страницы, основанную на компонентах, гораздо легче читать. Разработчик сразу видит общую структуру такой страницы. В данном случае, как можно понять из содержимого тега `<body>`, в верхней части страницы находится навигационная панель (`MySweetNavbar`), в середине размещено основное содержимое (`MainContent`), а в нижней части страницы имеется подвал (`MySweetFooter`).

Кроме того, компоненты не только улучшают структуру кода HTML-страниц. Они ещё и подходят для повторного использования. Как быть, если на нескольких страницах нужна одна и та же навигационная панель? Как быть, если такие панели на разных страницах немного отличаются друг от друга? Что делать, если всё та же панель используется на множестве страниц, а в неё нужно внести некое изменение? Без использования компонентного подхода трудно дать достойные ответы на эти и на многие другие вопросы.

Ещё одной причиной популярности React можно считать тот факт, что разработкой и поддержкой этой библиотеки занимается Facebook. Это, по меньшей мере, означает, что React на постоянной основе, занимаются квалифицированные программисты. Популярность React, и то, что проект это опенсорсный, опубликованный на GitHub, означает ещё и то, что вклад в проект делают множество

сторонних разработчиков. Всё это позволяет говорить о том, что React, в обозримом будущем, будет жить и развиваться.

Говоря о React, и, в частности, о том, почему эту библиотеку стоит изучать, нельзя не вспомнить об огромном рынке труда, связанном с этой библиотекой. В наши дни React-специалисты пользуются устойчивым спросом. Если вы изучаете React с целью найти работу в сфере фронтенд-разработки — это означает, что вы на правильном пути.

Занятие 4. Среда разработки, ReactDOM и JSX

Оригинал

Здесь мы поговорим о том, как создать простейшее React-приложение с использованием ReactDOM и затронем некоторые ключевые моменты, касающиеся JSX. Но, прежде чем приступать к работе с кодом, поговорим о том, где этот код запускать.

Среда разработки
Для того чтобы экспериментировать с React-кодом, пожалуй, лучше всего будет развернуть полноценную локальную среду разработки. Для того чтобы это сделать, вы можете обратиться к недавно опубликованному нами материалу React.js: понятное руководство для начинающих, в частности, к его разделу **Практика разработки React-приложений**. А именно, для того чтобы приступить к экспериментам, нужно, с помощью `create-react-app`, создать новое приложение, после чего запустить локальный сервер разработки и приступить к редактированию кода. Если речь идёт о простейших примерах, то их код можно вводить прямо в стандартный файл `index.js`, убрав из него имеющийся в нём код или закомментировав его.

Содержимое файла `index.html` в проекте, создаваемом `create-react-app`, соответствует его содержимому в примерах, которые будут приводиться в данном курсе. В частности, речь идёт о наличии на странице элемента `<div>` с идентификатором `root`.

Ещё один вариант, который обычно подходит для каких-то совсем простых экспериментов, заключается в использовании онлайн-платформ наподобие codepen.io. Например, вот демонстрационный проект React-приложения Дэна Абрамова. Суть подготовки Codepen-проекта к экспериментам с React заключается в подключении к нему библиотек `react` и `react-dom` (это можно сделать, щёлкнув по кнопке `Settings` в верхней части страницы, перейдя в появившемся окне в раздел `JavaScript` и подключив к проекту, предварительно найдя их с помощью встроенной системы поиска, нужные библиотеки).

Вполне возможно, что вам, для экспериментов, будет удобно пользоваться возможностями Scrimba. Для этого можете просто открыть страницу соответствующего занятия. Ссылки на эти страницы можно найти ниже заголовков с номерами и названиями занятий.

Первая программа
Надо отметить, что в наших примерах будут использоваться возможности ES6 (ES2015), поэтому вам весьма желательно в них ориентироваться. В частности, для того чтобы импортировать в проект библиотеку `react`, служит такая конструкция:

```
import React from "react"
```

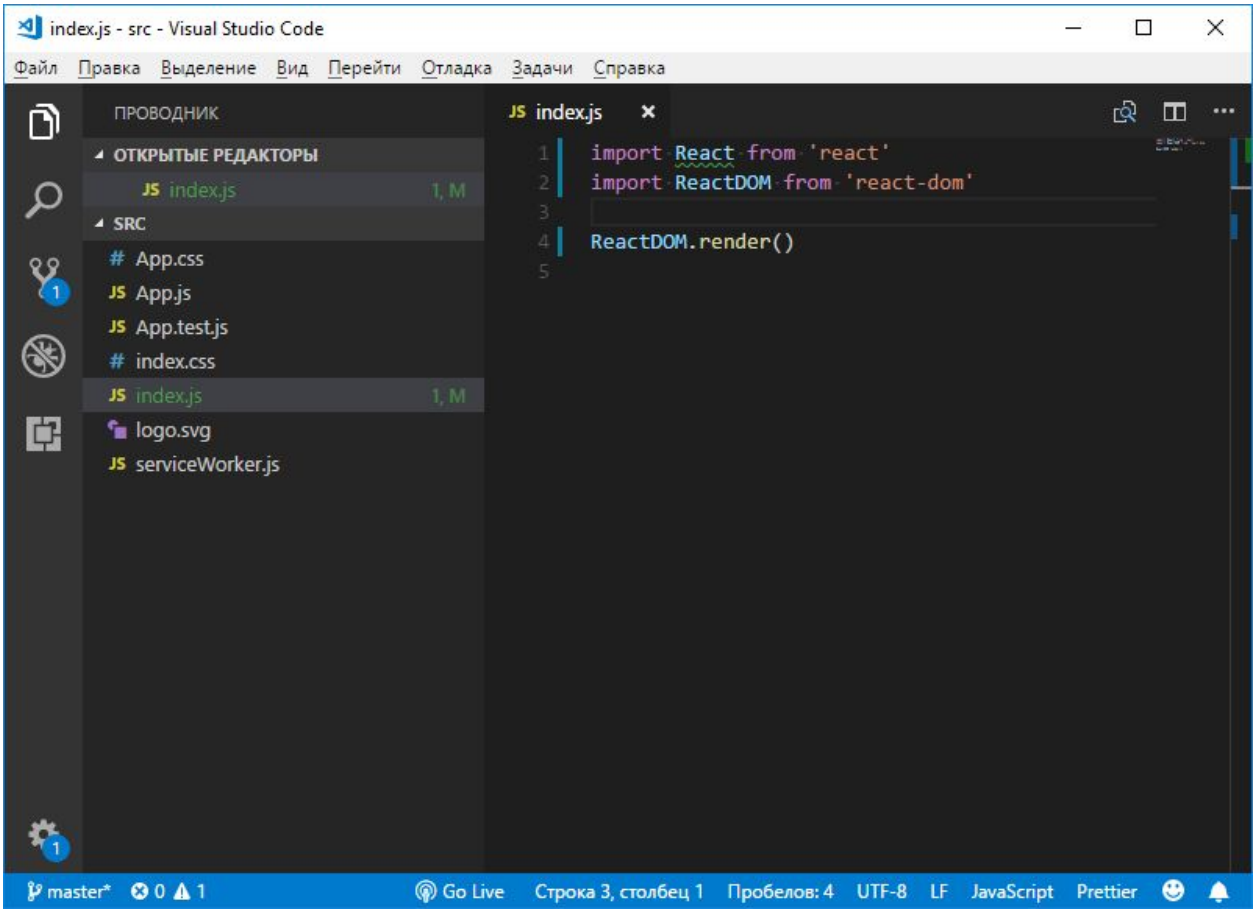
А так можно импортировать библиотеку `react-dom`:

```
import ReactDOM from "react-dom"
```

Теперь воспользуемся методом `render() ReactDOM` для того чтобы вывести что-то на экран:

```
ReactDOM.render()
```

Если вы решите использовать для экспериментов проект, созданный средствами `create-react-app`, то сейчас его файл `index.js` (открытый в VSCode), будет выглядеть так, как показано на следующем рисунке.



Ввод кода в index.js

Если у вас запущен сервер разработки и в браузере открыта страница `http://localhost:3000/`, то вы, сохранив такой `index.js`, вы увидите там сообщения об ошибках. Это, на данном этапе работы, совершенно нормально, так как мы пока не сообщили системе о том, что и куда мы хотим вывести командой `render()`.

На самом деле, сейчас пришло время разобраться с тем кодом, который мы только что написали. А именно, здесь мы импортировали в проект React, потом — ReactDOM — для того, чтобы возможностями этой библиотеки можно было бы воспользоваться для вывода чего-то на экран.

Метод `render()` принимает два аргумента. Первый будет тем, что мы хотим вывести, а второй — будет тем местом, куда мы хотим что-то вывести. Если это записать в виде псевдокода, то получится следующее:

```
ReactDOM.render(ЧТО ВЫВОДИТЬ, КУДА ВЫВОДИТЬ)
```

То, что мы хотим вывести, должно быть каким-то образом привязано к некоей HTML-странице. Тот код, который мы будем писать, будет превращён в HTML-элементы, которые и попадут на страницу.

Вот как эта страница может выглядеть.

```
<html>

    <head>
```

```
        <link rel="stylesheet" href="style.css">
    </head>
    <body>
        <div id="root"></div>
        <script src="index.js"></script>
    </body>
</html>
```

Тут имеются все базовые элементы HTML-страницы, включая тег `<link>` и тег `<script>`.

Если вы пользуетесь `create-react-app`, то страница `index.html` будет выглядеть немного иначе. В частности, в её коде нет команды импорта `index.js`. Дело в том, что при сборке проекта связь `index.html` и `index.js` осуществляется автоматически.

Обратите внимание на элемент `<div>` с идентификатором `root`. Между открывающим и закрывающим тегами этого элемента React разместит всё, что мы создадим. Этот элемент можно считать контейнером для всего нашего приложения.

Если теперь вернуться к файлу `index.js` и к методу `render() ReactDOM`, его вторым аргументом, местом, куда надо выводить данные, будет указание на элемент `<div>` с идентификатором `root`. Тут мы воспользуемся обычным JavaScript, после чего второй аргумент метода Render будет выглядеть так:

```
ReactDOM.render(ЧТО ВЫВОДИТЬ, document.getElementById("root"))
```

При таком подходе метод `render()` берёт первый аргумент и выводит то, что он описывает, в указанное место. Теперь займёмся этим первым аргументом. Начнём с простого элемента `<h1>`. И, как это обычно бывает при написании первой программы, добавим в него текст `Hello world!`:

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

Если теперь обновить страницу браузера, то на ней будет выведен, в качестве заголовка первого уровня, заданный текст.

Результат работы первой программы

Тут у вас может возникнуть вопрос о том, почему это мы помещаем описание HTML-элемента в то место, где ожидается аргумент JavaScript-метода. Ведь JavaScript, столкнувшись с чем-то вроде `<h1>Hello world!</h1>`, вероятно, решит, что первый символ этого выражения представляет собой оператор «меньше», дальше, очевидно, идёт имя переменной, потом — оператор сравнения «больше».

JavaScript не распознает в этой последовательности символов HTML-элемент, да он и не должен этого делать.

Разработчики React создали не только библиотеку, но и особый язык, который называется JSX. JSX очень похож на разновидность HTML. Дальше вы увидите, что практически весь JSX-код почти полностью совпадает с формируемой с его помощью HTML-разметкой. Различия между JSX и HTML, конечно, есть, и мы их постепенно обсудим.

Мы ввели довольно простую и короткую инструкцию, но в недрах React при её выполнении происходит много всего интересного. Так, эта инструкция преобразуется в её версию на JavaScript, осуществляется формирование HTML-кода, хотя в детали этого процесса мы тут не вдаёмся. Именно поэтому нам надо импортировать в проект не только `react-dom`, но и `react`, так как библиотека React — это именно то, что позволяет нам пользоваться JSX и сделать так, чтобы JSX-конструкции работали так, как ожидается. Если из нашего примера убрать строку `import React from "react"`, сохранить файл скрипта и обновить страницу, будет выведено сообщение об ошибке. В частности, `create-react-app` сообщит нам о том, что без доступа к React пользоваться JSX нельзя (`'React' must be in scope when using JSX  react/react-in-jsx-scope`).

Дело в том, что даже хотя в нашем примере React напрямую не используется, библиотека применяется для работы с JSX.

Ещё кое-что, касающееся работы с JSX, на что я хочу обратить ваше внимание, заключается в том, что нельзя рендерить JSX-элементы, следующие друг за другом. Предположим, что после элемента `<h1>` нужно вывести элемент `<p>`. Попробуем воспользоваться такой конструкцией:

```
ReactDOM.render(<h1>Hello world!</h1><p>This is a paragraph</p>,
document.getElementById("root")) //неправильно
```

Если после этого обновить страницу — будет выведено сообщение об ошибке (в `create-react-app` это выглядит как `Parsing error: Adjacent JSX elements must be wrapped in an enclosing tag`). Суть этой ошибки заключается в том, что такие элементы должны быть обёрнуты в какой-то другой элемент. То, что получится в итоге, должно выглядеть как один элемент с двумя вложенными в него элементами.

Для того чтобы наш пример заработал, обернём код `<h1>Hello world!</h1><p>This is a paragraph</p>` в элемент `<div>`:

```
ReactDOM.render(<div><h1>Hello world!</h1><p>This is a paragraph</p></div>,
document.getElementById("root"))
```

Если теперь обновить страницу, то всё будет выглядеть так, как ожидается.

Два HTML-элемента на странице

Для того чтобы привыкнуть к JSX, потребуется некоторое время, но после этого его использование окажется гораздо легче и удобнее, чем работа с HTML-элементами с использованием стандартных

средств JavaScript. Например, для того чтобы стандартными средствами описать элемент `<p>` и настроить его содержимое, понадобится примерно следующее:

```
var myNewP = document.createElement("p")

myNewP.innerHTML = "This is a paragraph."
```

Потом нужно подключить его к элементу, который уже есть на странице. Это — образец императивного программирования, а то же самое, благодаря JSX, можно делать в декларативном стиле.

Занятие 5. Практикум. ReactDOM и JSX

Оригинал

На прошлом занятии вы познакомились с ReactDOM и JSX, а теперь пришло время применить полученные знания на практике.

Все практические задания мы будем оформлять следующим образом. Сначала, в разделе с заголовком **Задание**, будет дано само задание, и, возможно, в разделе **Подсказки**, будут даны краткие рекомендации по его выполнению. Затем будет следовать раздел **Решение**. Рекомендуется приложить все усилия к тому, чтобы выполнить задание самостоятельно, а потом уже разбираться с приведённым решением.

Если вы чувствуете, что не справляетесь — вернитесь к предыдущему занятию, повторите соответствующий материал и попробуйте снова.

Задание
Напишите React-приложение, которое выводит на страницу маркированный список (тег `<ul>`). Этот список должен содержать три элемента (`<li>`) с любым текстом.

Подсказки
Сначала надо импортировать в проект необходимые библиотеки, а потом воспользоваться одной из них для вывода элементов, сформированных с помощью некоего JSX-кода, на страницу.

Решение
Для начала нужно импортировать в файл необходимые библиотеки. А именно — речь идёт о библиотеке `react`, и, так как мы собираемся выводить что-то на страницу, о библиотеке `react-dom`.

```
import React from "react"

import ReactDOM from "react-dom"
```

После этого надо воспользоваться методом `render()` объекта `ReactDOM`, передав ему описание элемента, который мы хотим вывести на страницу и указав место на странице, куда должен быть выведен этот элемент.

```
ReactDOM.render(

  <ul>

    <li>1</li>

    <li>2</li>

    <li>3</li>

  </ul>,
```

```
  document.getElementById("root")
)
```

Первый аргумент — это описание маркированного списка, второй — элемент страницы, в который он должен попасть — тег `<div>` с идентификатором `root`. HTML-код можно записать и в одну строку, но лучше оформить его так, как в нашем варианте решения.

Вот полный код решения:

```
import React from "react"
import ReactDOM from "react-dom"

ReactDOM.render(
  <ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>,
  document.getElementById("root")
)
```

Со временем мы поговорим о том, как, используя аккуратные конструкции, выводить с помощью метода `render()` большие объёмы HTML-разметки.

Учебный курс по React, часть 2: функциональные компоненты

Занятие 6. Функциональные компоненты

Оригинал

На предыдущем практическом занятии мы говорили о том, что необязательно размещать весь JSX-код, формирующий HTML-элементы, в аргументе метода `ReactDOM.render()`. В нашем случае речь идёт о маркированном списке, таком, описание которого представлено ниже.

```
import React from "react"
import ReactDOM from "react-dom"

ReactDOM.render(
  <ul>
    <li>1</li>
    <li>2</li>
```

```
    <li>3</li>
  </ul>,
  document.getElementById("root")
)
```

Представьте себе, что надо вывести, пользуясь таким же подходом, целую веб-страницу, на которой имеются сотни элементов. Если это сделать, то нормально поддерживать такой код будет практически невозможно. Когда мы говорили о причинах популярности React, одной из них была поддержка этой библиотекой компонентов, подходящих для повторного использования. Сейчас мы поговорим о том, как создавать функциональные компоненты React.

Эти компоненты называются «функциональными» из-за того, что создают их, конструируя особые функции.

Создадим новую функцию и дадим ей имя `MyApp`:

```
import React from "react"
import ReactDOM from "react-dom"

function MyApp() {

}

ReactDOM.render(
  <ul>
    <li>1</li>
    <li>2</li>
    <li>3</li>
  </ul>,
  document.getElementById("root")
)
```

Имя функции сделано именно таки по той причине, что здесь используется схема именования функций-конструкторов. Их имена (фактически — имена компонентов) записываются в верблюжьем стиле — первые буквы слов, из которых они состоят, делаются заглавными, в том числе — первая буква первого слова. Вам следует строго придерживаться этого соглашения об именовании подобных функций.

Функциональные компоненты устроены довольно просто. А именно, в теле функции должна быть команда, возвращающая JSX-код, который и представляет соответствующий компонент.

В нашем примере достаточно взять код маркированного списка и организовать возврат этого кода из функционального компонента. Вот как это может выглядеть:

```
function MyApp() {

  return <ul>

    <li>1</li>

    <li>2</li>

    <li>3</li>

  </ul>

}
```

И хотя в данном случае всё будет работать так, как ожидается, то есть, команда `return` вернёт весь этот код, рекомендуется заключать подобные конструкции в круглые скобки и применять ещё одно соглашение, принятое в React при форматировании кода программ. Оно заключается в том, что отдельные элементы помещают на отдельных строках и соответствующим образом выравнивают. В результате применения вышеописанных идей код нашего функционального компонента будет выглядеть так:

```
function MyApp() {

  return (

    <ul>

      <li>1</li>

      <li>2</li>

      <li>3</li>

    </ul>

  )

}
```

При таком подходе возвращаемая из компонента разметка оказывается очень похожей на обычный HTML-код.

Теперь, в методе `ReactDOM.render()`, можно создать экземпляр нашего функционального компонента, передав его этому методу в качестве первого аргумента и заключив его в JSX-тег.

```
import React from "react"

import ReactDOM from "react-dom"


function MyApp() {

  return (

    <ul>
```

```
      <li>1</li>
      <li>2</li>
      <li>3</li>
    </ul>
  )
}

ReactDOM.render(
  <MyApp />,
  document.getElementById("root")
)
```

Можно заметить, что тут использован самозакрывающийся тег. В некоторых случаях, когда нужно создавать компоненты, имеющие более сложную структуру, подобные конструкции строятся иначе, но пока мы будем пользоваться именно такими самозакрывающимися тегами.

Если обновить страницу, сформированную средствами вышеприведённого кода, то её внешний вид будет таким же, каким он был до выноса разметки маркированного списка в функциональный компонент.

Разметка, которую возвращают функциональные компоненты, подчиняется тем же правилам, которые мы рассматривали в применении к первому параметру метода `ReactDOM.render()`. То есть — нельзя, чтобы в ней присутствовали JSX-элементы, следующие друг за другом. Попытка поместить в предыдущем примере после элемента `<ul>` любой другой элемент, скажем — `<ol>`, приведёт к ошибке. Избежать этой проблемы можно, например, просто обернув всё, что возвращает функциональный компонент, в элемент `<div>`.

Возможно, вы уже начали ощущать то, какие мощные возможности даёт нам использование функциональных компонентов. В частности, речь идёт о создании собственных компонентов, которые содержат фрагменты JSX-кода, представляющие собой описание HTML-разметки, которая окажется на веб-странице. Такие компоненты можно компоновать друг с другом.

В нашем примере имеется компонент, выводящий простой HTML-список, но, по мере того, как мы будем создавать всё более сложные приложения, мы будем разрабатывать компоненты, выводящие другие созданные нами компоненты. В итоге всё это будет превращаться в обычные HTML-элементы, но иногда для формирования этих элементов понадобятся, возможно, десятки собственных компонентов.

В итоге, когда мы будем создавать всё больше компонентов, мы будем размещать их в отдельных файлах, но пока вам важно освоить то, что мы только что обсудили, привыкнуть к функциональным компонентам. В процессе прохождения курса вы будете создавать всё более и более сложные файловые структуры.

На этом занятии мы разобрали основы функциональных компонентов, а на следующем применим полученные знания на практике.

Занятие 7. Практикум. Функциональные компоненты

[Оригинал]

Задание

1. Подготовьте базовый React-проект.
2. Создайте функциональный компонент `MyInfo`, формирующий следующие HTML-элементы:
    a. Элемент `<h1>` с вашим именем.
    b. Текстовый абзац (элемент `<p>`), содержащий ваш краткий рассказ о себе.
    c. Список, маркированный (`<ul>`) или нумерованный (`<ol>`), с перечнем трёх мест, которые вам хотелось бы посетить.
3. Выведите экземпляр компонента `MyInfo` на веб-страницу.

Дополнительное задание
Стилизуйте элементы страницы, самостоятельно узнав о том, как это сделать (поищите в Google). Надо отметить, что о стилизации компонентов мы в этом курсе ещё поговорим.

Решение
Тут нас устроит такая же HTML-страница, которой мы пользовались ранее. Файл с React-кодом тоже будет выглядеть вполне стандартно. А именно, импортируем в него библиотеки, создадим каркас функционального компонента `MyInfo` и вызовем метод `render()` объекта `ReactDOM`, передав ему компонент, который надо вывести на страницу, и ссылку на элемент страницы, в котором должен быть выведен этот компонент. На данном этапе работы код будет выглядеть так:

```
import React from "react"

import ReactDOM from "react-dom"


function MyInfo() {


}


ReactDOM.render(<MyInfo />, document.getElementById("root"))
```

Теперь нужно вернуть из `MyInfo` JSX-код, формирующий HTML-разметку в соответствии с заданием. Вот полный код решения.

```
import React from "react"

import ReactDOM from "react-dom"


function MyInfo() {

  return (

    <div>

      <h1>Bob Ziroll</h1>

      <p>This is a paragraph about me...</p>
```

```
      <ul>
        <li>Thailand</li>
        <li>Japan</li>
        <li>Nordic Countries</li>
      </ul>
    </div>
  )
}

ReactDOM.render(
  <MyInfo />,
  document.getElementById("root")
)
```

Обратите внимание на то, что конструкция, возвращаемая из `MyInfo`, заключена в круглые скобки, и на то, что элементы, которые нужно вывести, находятся внутри вспомогательного элемента `<div>`.

Учебный курс по React, часть 3: файлы компонентов, структура проектов

Занятие 8. Файлы компонентов, структура React-проектов

Оригинал

Файлы компонентов

Если предположить, что вы выполняли задание из предыдущего практического занятия, используя стандартный проект, созданный `create-react-app`, то сейчас в нём задействован файл `index.html` из папки `public`, содержимое которого нас устраивает, и файл `index.js` из папки `src`, в котором мы пишем код. В частности, `index.js` выглядит сейчас примерно так:

```
import React from "react"
import ReactDOM from "react-dom"

function MyInfo() {
  return (
    <div>
      <h1>Bob Ziroll</h1>
      <p>This is a paragraph about me...</p>
      <ul>
```

```
        <li>Thailand</li>
        <li>Japan</li>
        <li>Nordic Countries</li>
      </ul>
    </div>
  )
}

ReactDOM.render(
  <MyInfo />,
  document.getElementById("root")
)
```

Обратите внимание на то, что код функционального компонента `MyInfo` содержится именно в этом файле. Как вы помните, React позволяет создавать множество компонентов, в этом кроется одна из его сильных сторон. Понятно, что разместить код большого количества компонентов в одном файле, хотя и технически реализуемо, на практике означает большие неудобства. Поэтому код компонентов, даже небольших по объёму, обычно оформляют в виде отдельных файлов. Именно такому подходу и рекомендуется следовать при разработке React-приложений.

Файлам компонентов дают имена, соответствующие именам компонентов, код которых эти файлы хранят. Размещают их, в случае с `create-react-app`, в той же папке `src`, где расположен файл `index.js`. При таком подходе файл с компонентом `MyInfo` получит имя `MyInfo.js`.

Создадим файл `MyInfo.js` и перенесём в него код компонента `MyInfo`, удалив его из `index.js`.

На данном этапе работы `index.js` будет выглядеть так:

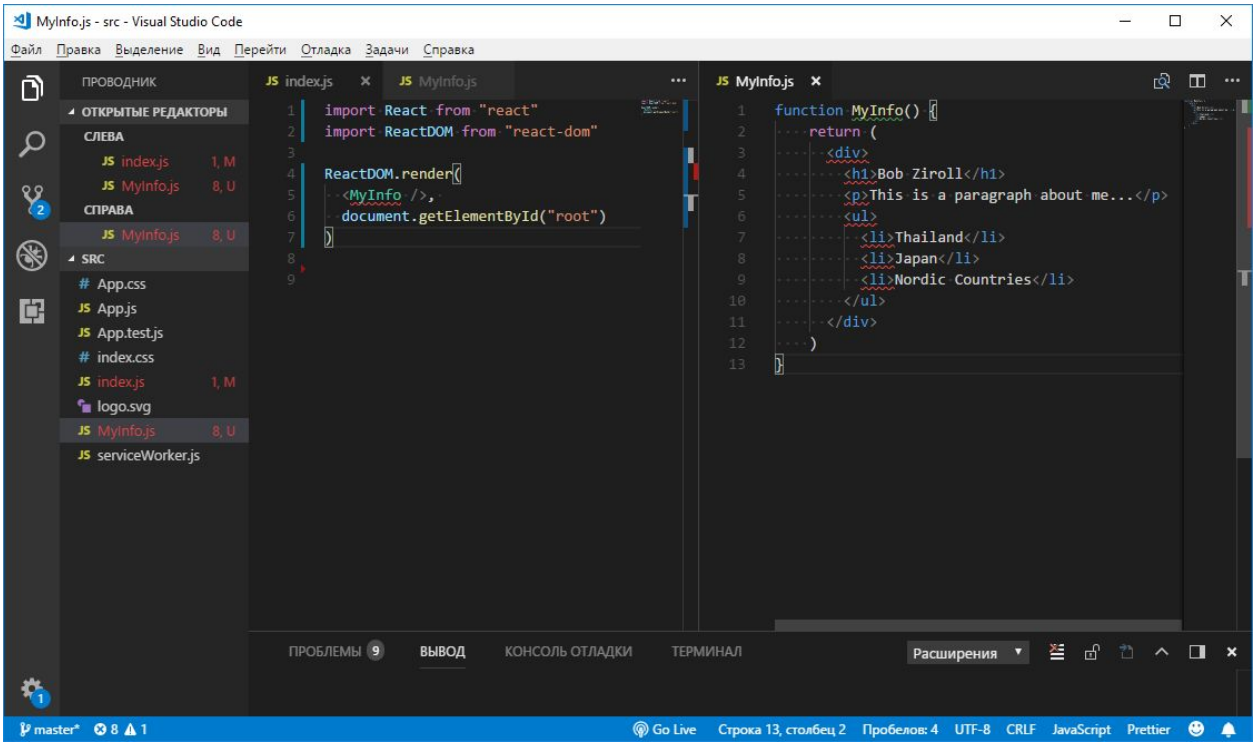
```
import React from "react"
import ReactDOM from "react-dom"

ReactDOM.render(
  <MyInfo />,
  document.getElementById("root")
)
```

Код `MyInfo.js` будет таким:

```
function MyInfo() {
    return (
```

```
      <div>
        <h1>Bob Ziroll</h1>
        <p>This is a paragraph about me...</p>
        <ul>
          <li>Thailand</li>
          <li>Japan</li>
          <li>Nordic Countries</li>
        </ul>
      </div>
    )
}
```

Вот как всё это выглядит в VSCode.



Перенос кода компонента в новый файл

Код компонента из `index.js` мы перенесли, но получившаяся у нас сейчас структура пока ещё неработоспособна.

Во-первых, вспомните — для чего в `index.js` нужна команда `import React from "react"`, даже с учётом того, что мы напрямую к `React` здесь не обращаемся. Причина этого в том, что без импорта `React` не будут работать механизмы этой библиотеки, в частности — JSX. Именно благодаря этой команде импорта мы могли, на предыдущих занятиях, передавать методу `ReactDOM.render()` JSX-код и выводить HTML-разметку, создаваемую на его основе, на страницу. Всё это значит, что в файле `MyInfo.js` нам тоже нужно импортировать React. Это — обычная практика для файлов компонентов.

Во-вторых, нам нужно сделать так, чтобы функцией `MyInfo` из файла `MyInfo.js` можно было бы воспользоваться в других файлах приложения. Её для этого нужно экспортировать. Тут используются возможности стандарта ES6. В результате обновлённый код `MyInfo.js` принимает следующий вид:

```
import React from "react"


function MyInfo() {
    return (
      <div>
        <h1>Bob Ziroll</h1>
        <p>This is a paragraph about me...</p>
        <ul>
          <li>Thailand</li>
          <li>Japan</li>
          <li>Nordic Countries</li>
        </ul>
      </div>
    )
}


export default MyInfo
```

Теперь поработаем над файлом `index.js`. А именно, нам нужно, чтобы компонент `MyInfo` был бы доступен в этом файле. Сделать его доступным в `index.js` можно, импортировав его.

Что если попробовать записать команду импорта компонента по образцу команд импорта `react` и `react-dom` в `index.js`? Например, внесём в файл такую команду импорта:

```
import MyInfo from "MyInfo.js" // неправильно
```

Система, увидев такую команду, в частности, опираясь на то, что в ней отсутствуют сведения об относительном пути к файлу, будет искать зависимость проекта — модуль с именем, заданным при вызове этой команды (вот как устанавливать зависимости в проекты, созданные средствами `create-react-app`; эти зависимости потом можно импортировать в React-проекты так же, как была импортирована библиотека React). Такого модуля она не найдёт, в результате команда импорта не сработает. Поэтому команду импорта файла нужно переписать с указанием пути к нему. В данном случае нас устроит указание на текущую директорию (`./`) и команда импорта приобретёт следующий вид:

```
import MyInfo from "./MyInfo.js" // правильно
```

Кроме того, если говорить о команде `import`, важно учитывать, что она подразумевает то, что с её помощью импортируются JavaScript-файлы. То есть расширение `.js` вполне можно убрать, и команда `import`, приобретя вид, показанный ниже, не утратит работоспособности.

```
import MyInfo from "./MyInfo" // правильно
```

Обычно подобные команды импорта JS-файлов записывают именно так.

Вот полный код файла `index.js`.

```
import React from "react"

import ReactDOM from "react-dom"


import MyInfo from "./MyInfo"


ReactDOM.render(

  <MyInfo />,

  document.getElementById("root")

)
```

Структура проекта

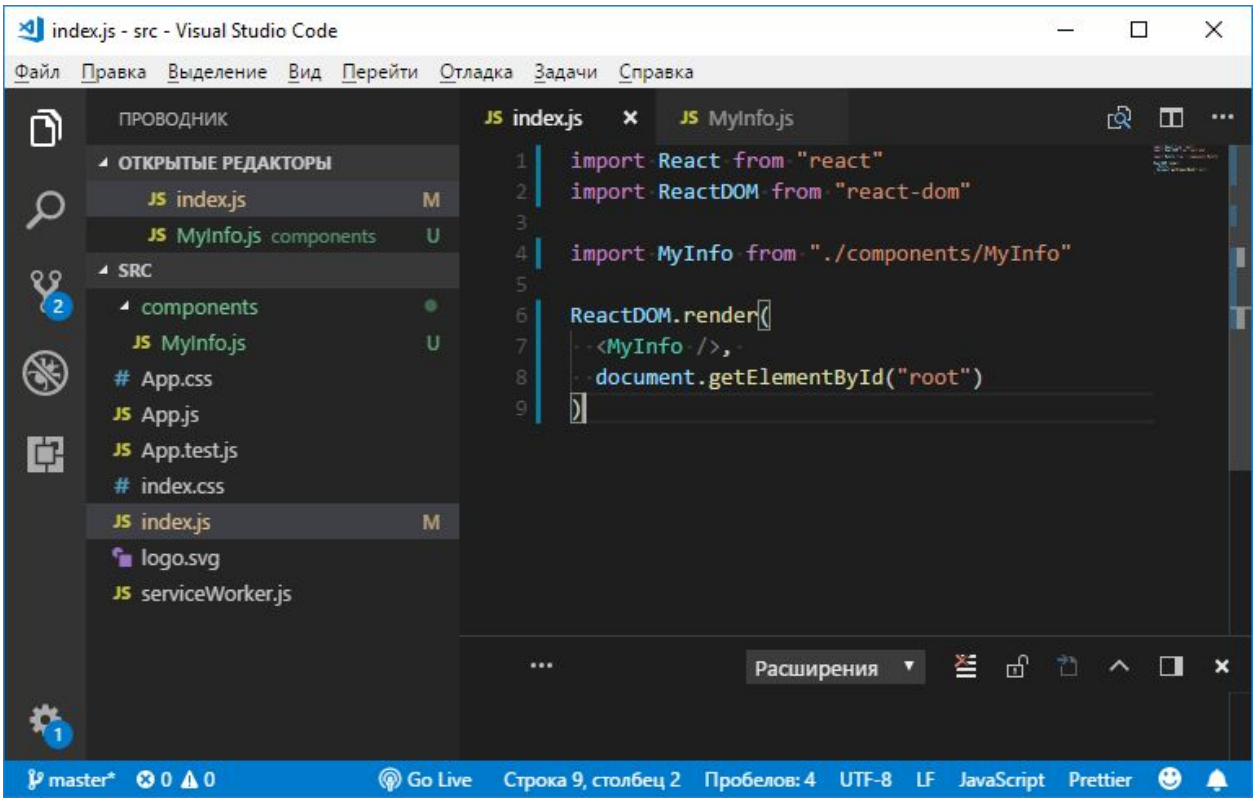
При росте размеров и сложности React-проектов очень важно поддерживать в хорошем состоянии их структуру. В нашем случае, хотя наш проект сейчас и невелик, можно, в папке `src`, создать папку `components`, предназначенную для хранения файлов с кодом компонентов.

Создадим такую папку и переместим в неё файл `MyInfo.js`. После этого нужно будет отредактировать команду импорта этого файла в `index.js`.

А именно, сейчас путь к `MyInfo.js` указывает на то, что этот файл находится там же, где и `index.js`, но на самом деле файл этот теперь находится в папке `components`, находящейся в той же папке, что и `index.js`. В результате относительный путь к нему в `index.js` будет выглядеть так: `./components/MyInfo`. Вот каким будет обновлённый код `index.js`:

```
import React from "react"

import ReactDOM from "react-dom"


import MyInfo from "./components/MyInfo"


ReactDOM.render(

  <MyInfo />,

  document.getElementById("root")

)
```

А вот как всё это выглядит в VSCode.



Папка для хранения компонентов и импорт компонента из этой папки в VSCode

На самом деле, одна папка `components`, предназначенная для размещения кода всех компонентов — это пример чрезвычайно упрощённой структуры проекта. В реальных проектах, для обеспечения удобства работы с большим количеством сущностей, используются гораздо более сложные структуры папок. То, какими именно будут эти структуры, зависит от нужд проекта и от личных предпочтений программиста.

Рекомендуется поэкспериментировать со всем тем, о чём вы сегодня узнали. Например, можете попробовать переместить файл `MyInfo.js` в какую-нибудь папку и посмотреть — что из этого выйдет, можно попытаться переименовать его, поменять в нём какой-то код. Когда в ходе подобных экспериментов правильная работа проекта будет нарушена — полезно будет разобраться в проблеме и снова привести проект в работоспособное состояние.

Учебный курс по React, часть 4: родительские и дочерние компоненты

Занятие 9. Родительские и дочерние компоненты

Оригинал

Сегодня мы поговорим о родительских и дочерних компонентах. Использование подобных конструкций сделает наше приложение гораздо более сложным, чем в случае, когда в нём был всего один компонент, выводимый в DOM, такой, как `MyInfo`. Вместо этой простой структуры в приложении может присутствовать сложная иерархия компонентов, которая, в итоге, преобразуется в JSX-элементы.

Начнём с уже знакомого вам шаблона приложения. Для того чтобы вспомнить пройденное, можете, по памяти, в пустом файле `index.js`, написать код для вывода на страницу заголовка первого уровня с текстом `Hello World!` средствами React. Вот как может выглядеть подобный код:

```
import React from "react"
```

```
import ReactDOM from "react-dom"


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

В прошлый раз там, где в вышеприведённом коде находится описание элемента `<h1>`, присутствовал код для вывода компонента `MyInfo`. Теперь же мы собираемся создать компонент `App` и вывести его. Для этого нам понадобится код следующего вида:

```
import React from "react"
import ReactDOM from "react-dom"


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

Компонент `App` будет точкой входа в наше приложение. Вероятно, вы уже заметили, что в коде предыдущего примера кое-чего не хватает. Это действительно так — мы пока не импортировали сюда `App`. Сделаем это:

```
import React from "react"
import ReactDOM from "react-dom"


import App from "./App"


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

Но такой код всё ещё остаётся нерабочим. Нам нужен файл компонента `App` (`App.js`), расположенный в той же папке, что и `index.js`. Именно к такому файлу мы обращаемся в команде импорта `import App from "./App"`. Напомним, что имена компонентов React записываются в верблюжьем стиле и начинаются с заглавной буквы. Создадим нужный нам файл и опишем в нём компонент `App`. Попытайтесь сделать это самостоятельно. А именно — напишите код, благодаря которому компонент `App` выведет на страницу текст `Hello again`.

Вот как выглядит этот код:

```
import React from "react"


function App(){
  return (
    <h1>Hello again</h1>
  )
}


export default App
```

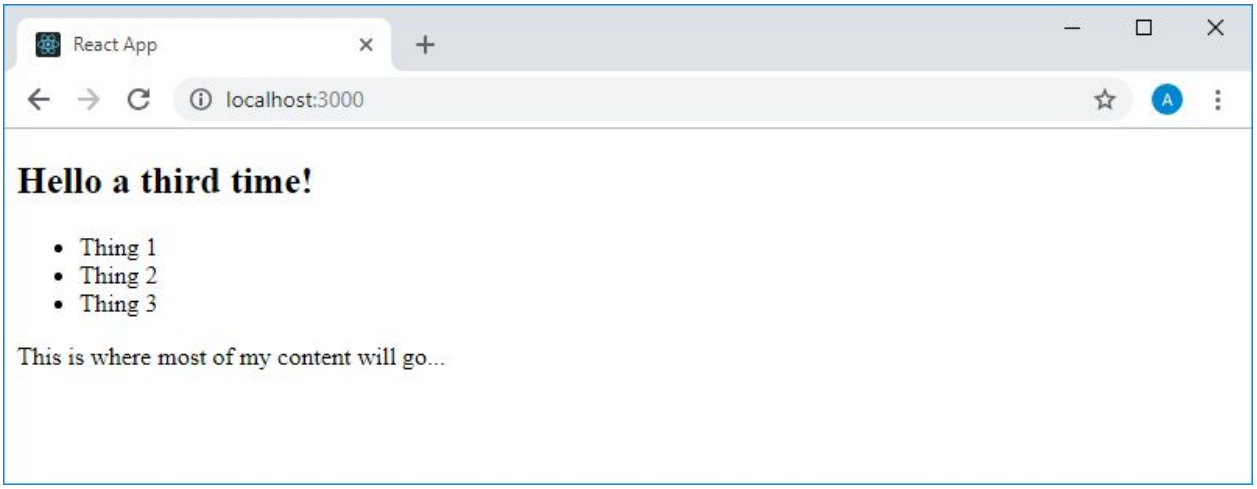
Тут функция `App` возвращает единственный элемент, но напомним, что из подобных функций можно возвращать и более сложные структуры. Самое главное — не забывать о том, что возвратить можно лишь один элемент, который, если, на самом деле, вывести нужно несколько элементов, представляет собой контейнер, включающий их в себя. Например, вот как будет выглядеть возврат разметки, описывающей заголовок первого уровня и маркированный список:

```
import React from "react"


function App(){
  return (
    <div>
      <h1>Hello a third time!</h1>
      <ul>
        <li>Thing 1</li>
        <li>Thing 2</li>
        <li>Thing 3</li>
      </ul>
    </div>
  )
}


export default App
```

Возможно, сейчас мы решим, что то, что формируется средствами компонента `App`, должно представлять собой некий веб-сайт. У него будет навигационный блок (`<nav></nav>`) и область основного содержимого (`<main></main>`). Это решение приведёт к формированию следующего кода:

```
import React from "react"


function App(){
  return (
    <div>
      <nav>
        <h1>Hello a third time!</h1>
        <ul>
          <li>Thing 1</li>
          <li>Thing 2</li>
          <li>Thing 3</li>
        </ul>
      </nav>
      <main>
        <p>This is where most of my content will go...</p>
      </main>
    </div>
  )
}


export default App
```

Вот как всё это будет выглядеть в браузере.

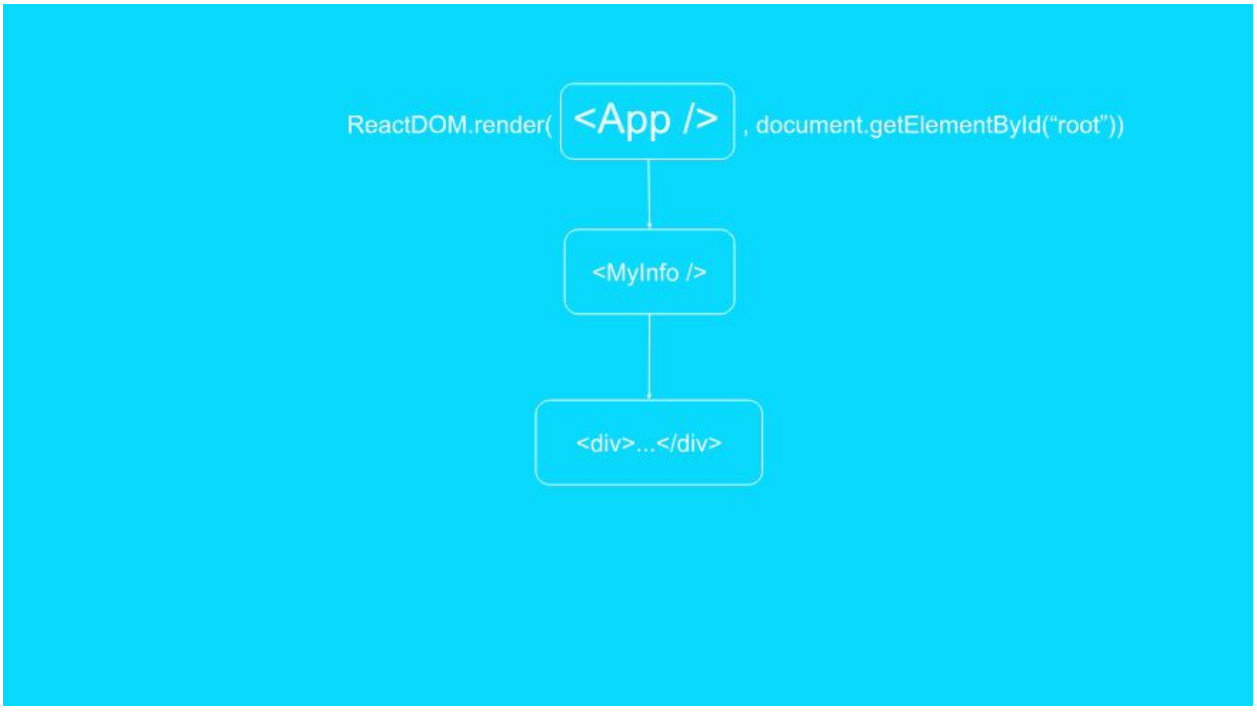


Приложение в браузере

Тут ещё можно стилизовать список для того, чтобы он стал больше похожим на навигационную панель.

Можно заметить, что код компонента уже стал довольно большим. Это идёт вразрез с целью, ради которой мы используем React. Ранее мы говорили о том, что фрагменты HTML-кода можно представлять в виде отдельных компонентов, а в нашем компоненте всё свалено в одну кучу. Поэтому сейчас мы создадим компоненты для каждого самостоятельного фрагмента разметки.

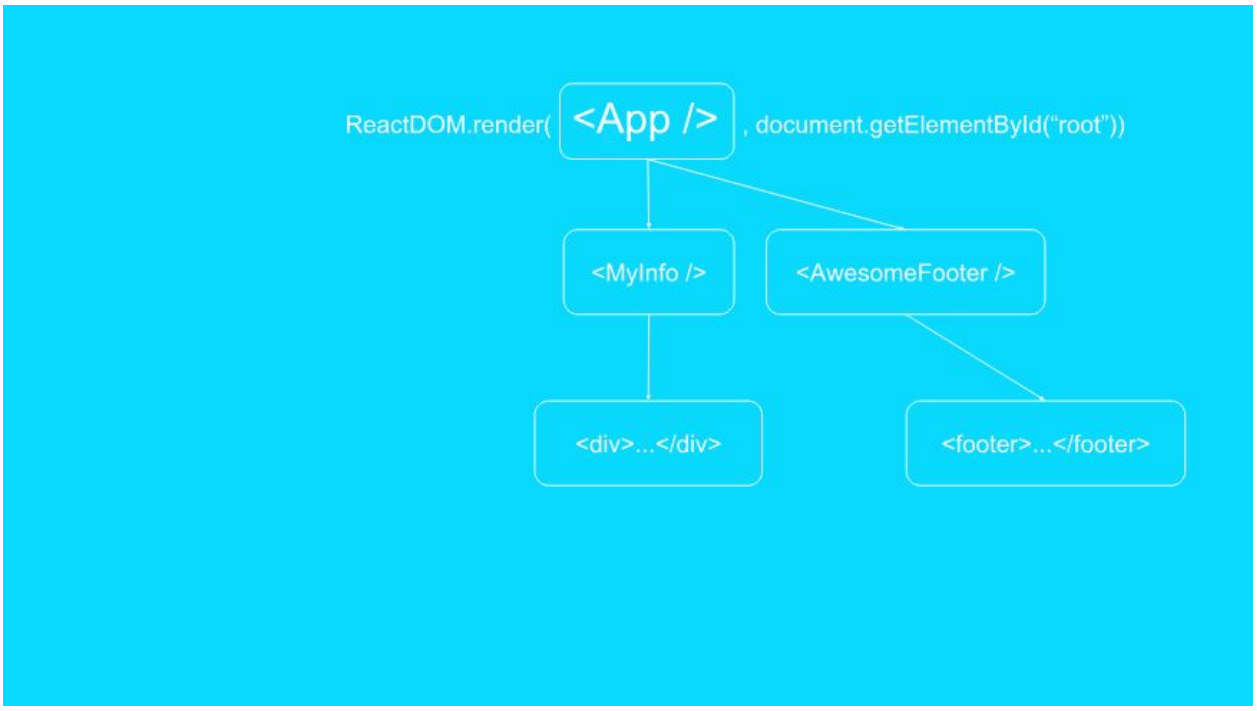
Взгляните на эту схему для того, чтобы лучше разобраться в том, о чём идёт речь.



*Компонент App выводит компонент MyInfo, выводящий элемент *

Тут мы выводим на страницу компонент `App`. При этом компонент `App` решает вывести ещё один компонент — `MyInfo`. А уже компонент `MyInfo` выводит некий JSX-элемент. Обратите внимание на разницу между понятиями «компонент» и «элемент». Элементы — это сущности, которые превращаются в обычный HTML-код. Так, в элементе, представленном в нижней части схемы, используется простой тег `<div>`, его имя начинается с маленькой буквы, что говорит нам о том, что это — обычный элемент, а не один из созданных нами компонентов. С другой стороны, имя `MyInfo` начинается с большой буквы. Это помогает понять, что перед нами — компонент.

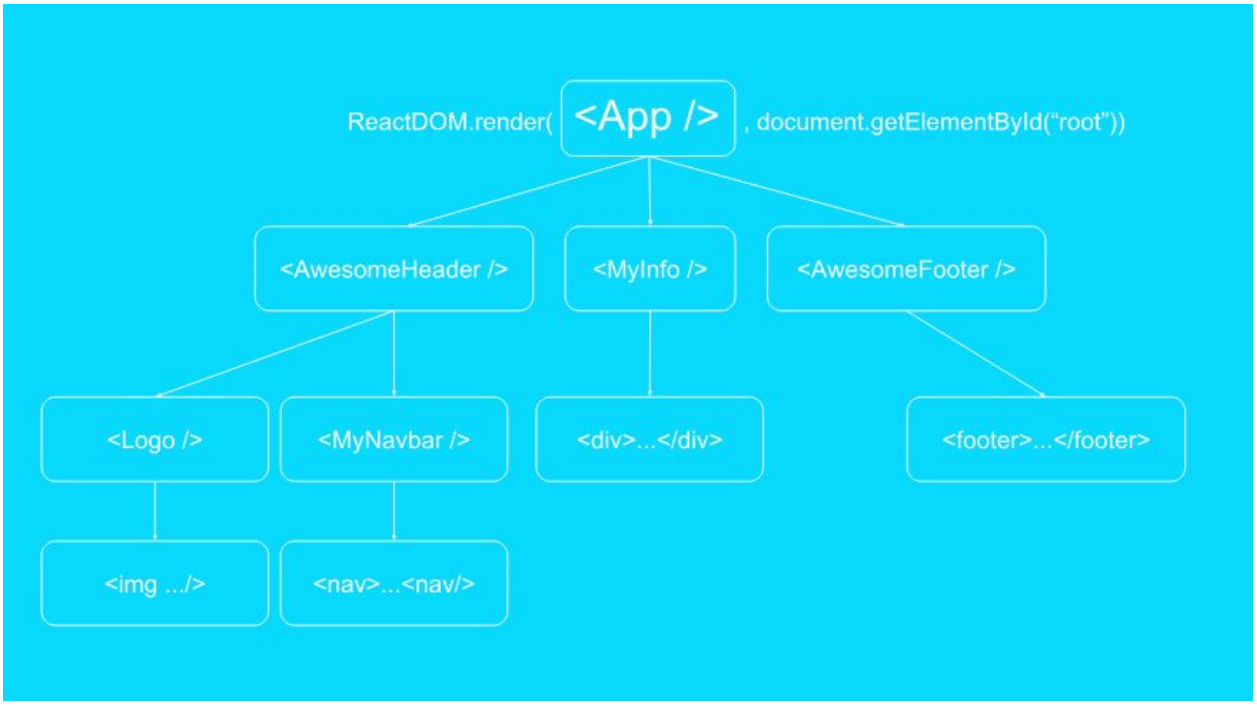
Вы могли слышать о том, что DOM (Document Object Model, объектная модель документа) часто называют «деревом». Корневым элементом этого дерева является элемент `<html>`. В нашем случаем корневым элементом дерева, представленного на схеме, является компонент `App`. Возможности этого компонента не ограничиваются выводом другого компонента, `MyInfo` в нашем случае. Он может, например, вывести ещё один компонент, представляющий собой «подвал», нижнюю часть страницы. Скажем, этот компонент будет носить имя `AwesomeFooter`.



Компонент App выводит два компонента

Этот компонент, в свою очередь, может вывести элемент `<footer>`, который будет содержать HTML-код нижней части страницы.

Если у нас имеется «подвал» страницы, то она может содержать и «шапку», оформляющую её верхнюю часть.



Компонент App выводит три компонента

Верхняя часть страницы представлена на нашей схеме компонентом `AwesomeHeader`. Такие имена этим компонентам даны для того, чтобы не путать их с элементами. Компонент `AwesomeHeader`, как и компонент `App`, может выводить не только JSX-разметку, но и другие компоненты. Например, это может быть компонент `NavBar`, представляющий собой навигационную панель, и компонент `Logo`, выводящий логотип. А эти компоненты уже выведут обычные элементы — такие, как `<img />` и `<nav>`.

По мере рассмотрения этой схемы вы можете заметить, что React-приложение, в ходе его развития, может становиться всё более и более сложным. И то, что мы тут рассмотрели, на самом деле, представляет собой пример крайне простой структуры приложения.

Теперь давайте создадим в нашем учебном приложении компонент, который будет представлять собой «подвал» страницы.

Для этого создадим, в той же папке, где находится файл `index.js`, новый файл. Назовём его `Footer.js` и поместим в него следующий код:

```
import React from "react"


function Footer() {
    return (
        <footer>
            <h3>This is my footer element</h3>
        </footer>
    )
}


export default Footer
```

Обратите внимание на то, что имя функционального компонента начинается с большой буквы (`Footer`), а имя элемента (`<footer>`) — с маленькой. Как уже было сказано, это помогает отличать элементы от компонентов.

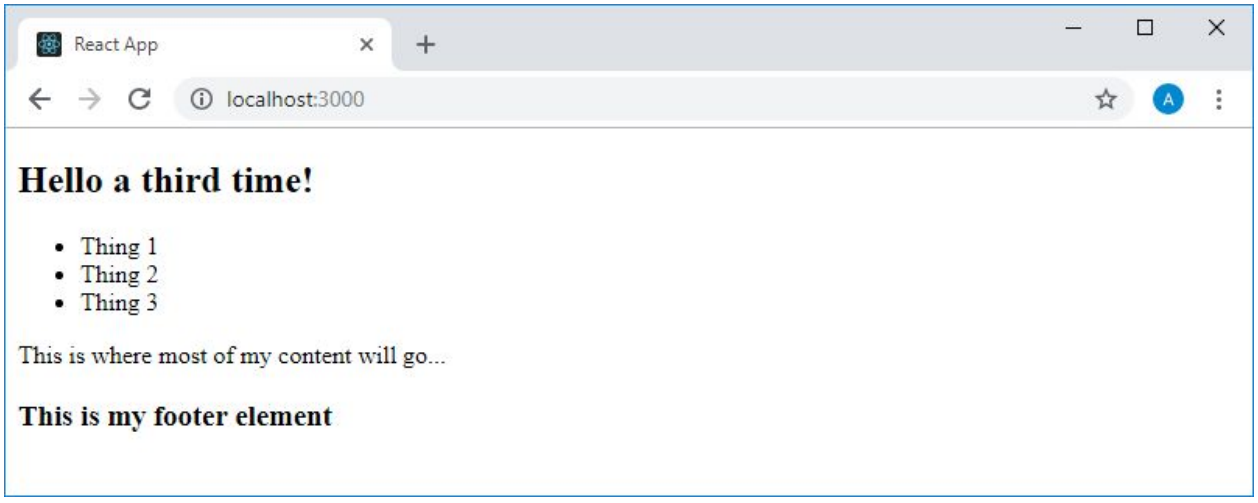
Если теперь обновить страницу, то разметка, формируемая компонентом `Footer`, в её нижней части выведена не будет. Это совершенно ожидаемо, так как для того, чтобы её вывести, нужно внести соответствующие изменения в код компонента `App`.

А именно, речь идёт о том, что в коде файла компонента `App` нужно импортировать компонент `Footer` и создать его экземпляр. Отредактируем код файла `App.js`:

```
import React from "react"

import Footer from "./Footer"


function App(){
```

```
  return (
    <div>
      <nav>
        <h1>Hello a third time!</h1>
        <ul>
          <li>Thing 1</li>
          <li>Thing 2</li>
          <li>Thing 3</li>
        </ul>
      </nav>
      <main>
        <p>This is where most of my content will go...</p>
      </main>
      <Footer />
    </div>
  )
}


export default App
```

Теперь страница, которую формирует приложение, будет выглядеть так, как показано ниже.



Компонент App выводит в нижней части страницы разметку, формируемую другим компонентом — Footer

Можно заметить, что сейчас в коде, выводимом компонентом `App`, имеется странная смесь из обычных JSX-элементов с компонентами. Куда лучше было бы, если бы то, что выводит компонент `App`, было бы

похоже на нечто вроде оглавления книги, чтобы в нём присутствовали, в основном, компоненты. А именно, речь идёт о том, чтобы код компонента выглядел бы примерно так:

```
import React from "react"

import Footer from "./Footer"


function App() {

    return (

        <div>

            <Header />

            <MainContent />

            <Footer />

        </div>

    )

}


export default App
```

Если приложение имеет подобную структуру (в нашем случае, так как файлы компонентов `Header` и `MainContent` пока не созданы, код работать не будет), то описание элементов, формирующих различные части страницы, будет находиться в файлах соответствующих компонентов. При этом компоненты, импортируемые в компонент `App`, могут содержать другие вложенные компоненты. Так могут быть сформированы довольно обширные структуры, размеры которых определяются нуждами конкретного приложения.

Здесь мы поговорили о том, как работать с вложенными компонентами. Вы вполне можете попробовать на практике то, что только что узнали, приведя проект, файл `App.js` которого выглядит так, как показано выше, в рабочее состояние.

Занятие 10. Практикум. Родительские и дочерние компоненты

Оригинал

Задание
Создайте React-приложение с нуля. Выведите на страницу корневой компонент `App` (определённый в отдельном файле). Внутри этого компонента выведите следующие компоненты:

1. `Navbar`
2. `MainContent`
3. `Footer`

Компоненты, выводимые `App`, должны быть описаны в отдельных файлах, каждый из них должен выводить какие-нибудь JSX-элементы.

Решение

В качестве основы для решения этой задачи используется стандартный проект, создаваемый средствами `create-react-app` (если вы не знаете как такой проект создать — взгляните на этот материал). Тут используется стандартный `index.html`.

Код файла `index.js`:

```
import React from "react"

import ReactDOM from "react-dom"


import App from "./App"


ReactDOM.render(

  <App />,

  document.getElementById("root")

)
```

Вот код файла `App.js`. Обратите внимание на то, что для хранения файлов компонентов мы будем использовать папку `components`.

```
import React from "react"


import Header from "./components/Header"

import MainContent from "./components/MainContent"

import Footer from "./components/Footer"


function App() {

    return (

        <div>

            <Header />

            <MainContent />

            <Footer />

        </div>

    )

}
```

```
export default App
```

Код файла `Header.js`:

```
import React from "react"


function Header() {
    return (
        <header>This is the header</header>
    )
}


export default Header
```

Код файла `MainContent.js`:

```
import React from "react"


function MainContent() {
    return (
        <main>This is the main section</main>
    )
}


export default MainContent
```

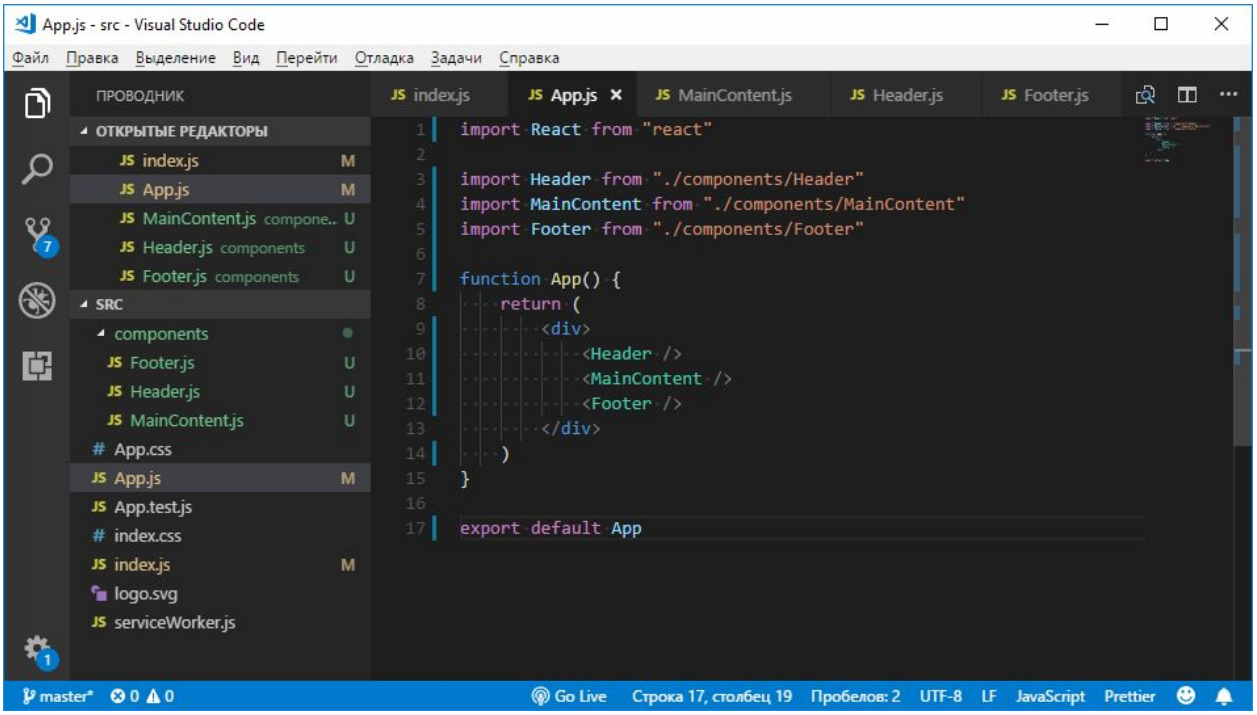
Код файла `Footer.js`:

```
import React from "react"


function Footer() {
    return (
        <footer>This is the footer</footer>
    )
}
```

```
export default Footer
```

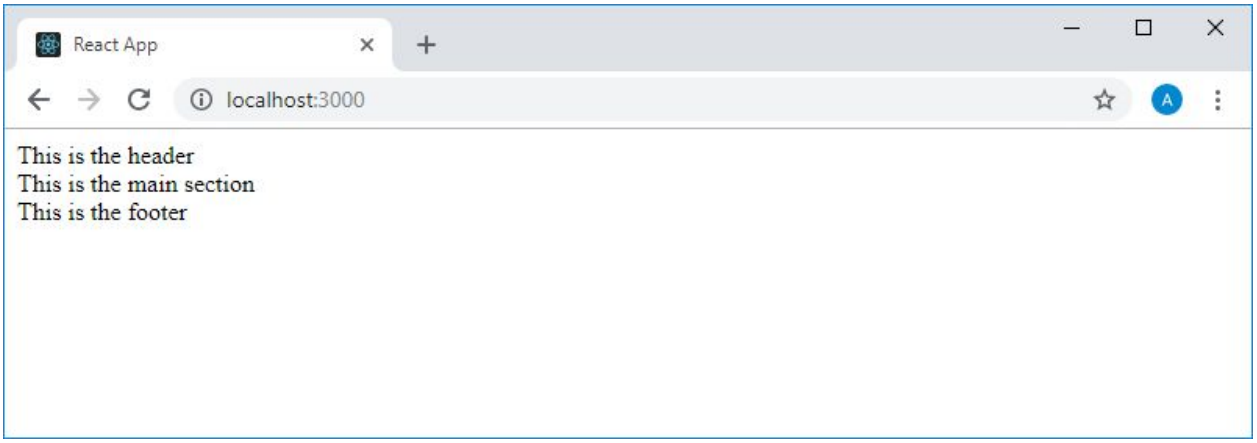
Работу с компонентами вы можете организовать так, как вам будет удобнее. То есть, можно, например, сначала написать в файле `App.js` весь необходимый код, выполняющий импорт компонентов и вывод их экземпляров, а потом создать файлы компонентов. Можно сделать всё наоборот — сначала создать файлы компонентов с кодом, а потом уже работать над файлом `App.js`. Самое главное, чтобы в итоге получилось работающее приложение.

Вот как выглядит проект в VSCode.



Проект в VSCode

А вот как выглядит страница, которую сформировало это приложение.



Страница приложения в браузере

Наше React-приложение работает, но то, что оно выводит на страницу, выглядит как-то совсем неинтересно. Исправить это можно, стилизовав содержимое страницы.

Учебный курс по React, часть 5: начало работы над TODO-приложением, основы стилизации

Занятие 11. Практикум. TODO-приложение. Этап №1

Оригинал

На этом занятии мы начнём работу над нашим первым учебным проектом — TODO-приложением. Подобные занятия будут оформлены как обычные практикумы. Сначала вам будет дано задание, для выполнения которого будет необходимо ориентироваться в ранее изученном материале, после чего будет представлено решение.

Мы будем работать над этим приложением довольно долго, поэтому, если вы пользуетесь `create-react-app`, рекомендуется создать для него отдельный проект.

Задание

- Создайте новое React-приложение.
- Выведите на страницу компонент `<App />` средствами файла `index.js`.
- Компонент `<App />` должен формировать код для вывода 3-4 флажков с каким-нибудь текстом, следующим после них. Текст может быть оформлен с помощью тегов `<p>` или `<span>`. То, что у вас получится, должно напоминать список дел, в который уже внесены некие записи.

Решение
Код файла `index.js`:

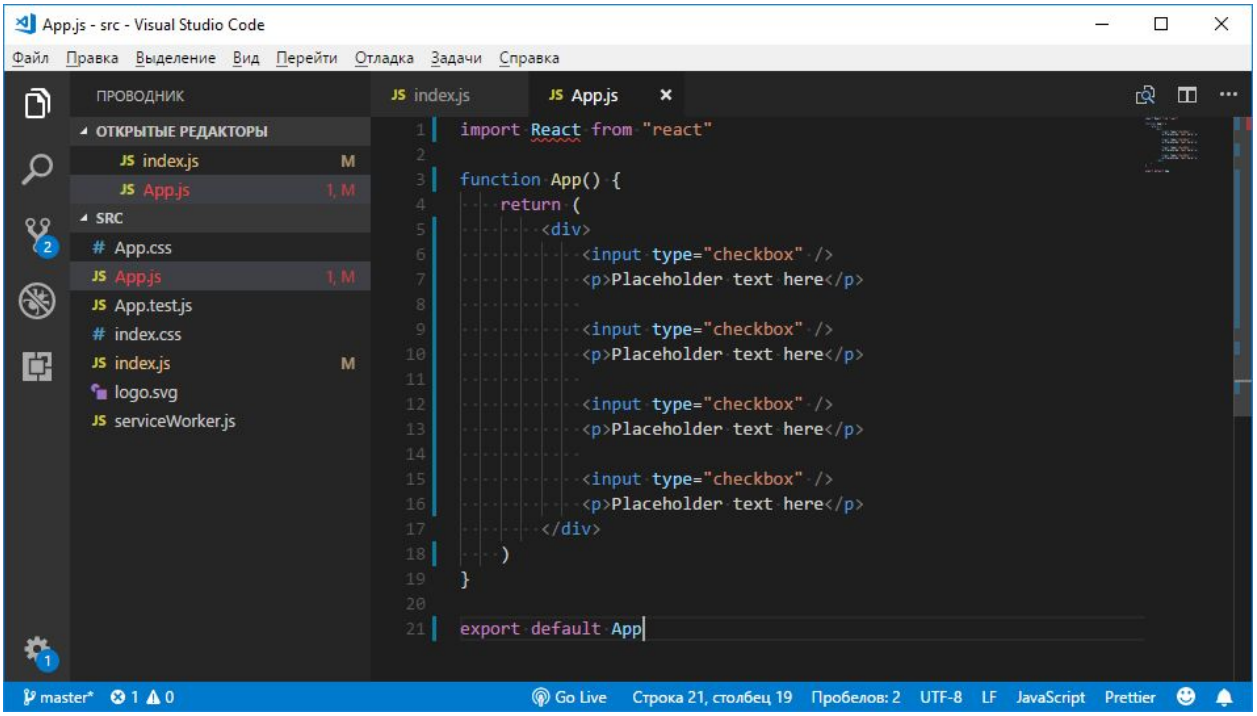
```
import React from 'react'

import ReactDOM from 'react-dom'


import App from './App'


ReactDOM.render(

    <App />,

    document.getElementById('root')

)
```

Код файла `App.js`:

```
import React from "react"


function App() {

    return (

        <div>

            <input type="checkbox" />
```

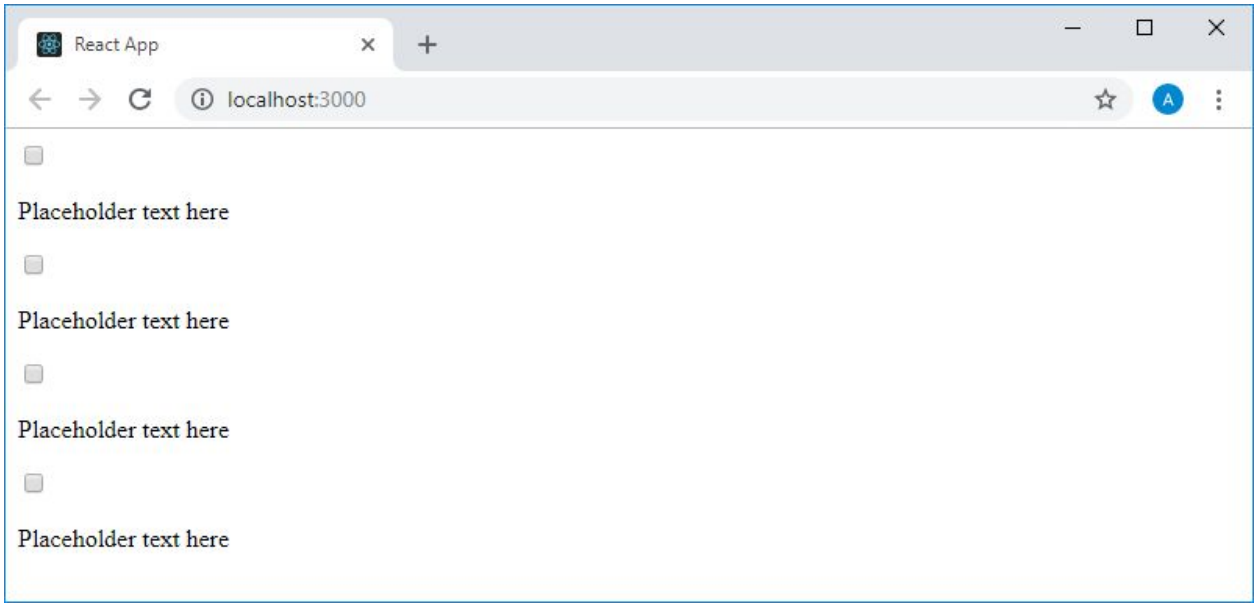
```
            <p>Placeholder text here</p>

            <input type="checkbox" />
            <p>Placeholder text here</p>

            <input type="checkbox" />
            <p>Placeholder text here</p>

            <input type="checkbox" />
            <p>Placeholder text here</p>
        </div>
    )
}

export default App
```

Вот как на данном этапе работы выглядит стандартный проект `create-react-app` в VSCode.



Проект в VSCode

Вот что наше приложение выводит сейчас на страницу.



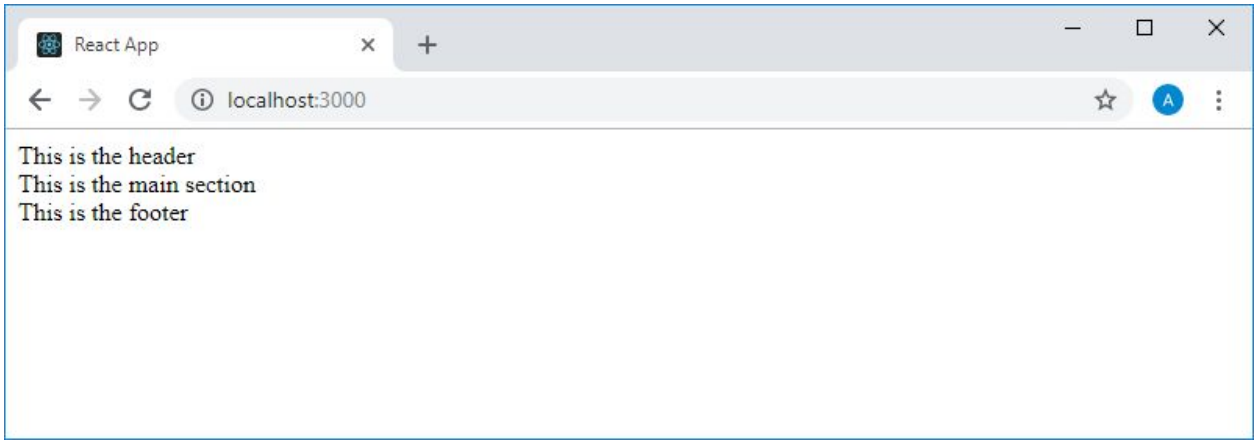
Внешний вид приложения в браузере

Сегодня мы сделали первый шаг на пути к TODO-приложению. Но то, что выводит это приложение на экран, выглядит не так уж и приятно, как и то, что оказывалось на страницах в ходе прошлых наших экспериментов. Поэтому на следующем занятии мы займёмся стилизацией элементов.

Занятие 12. Стилизация в React с использованием CSS-классов

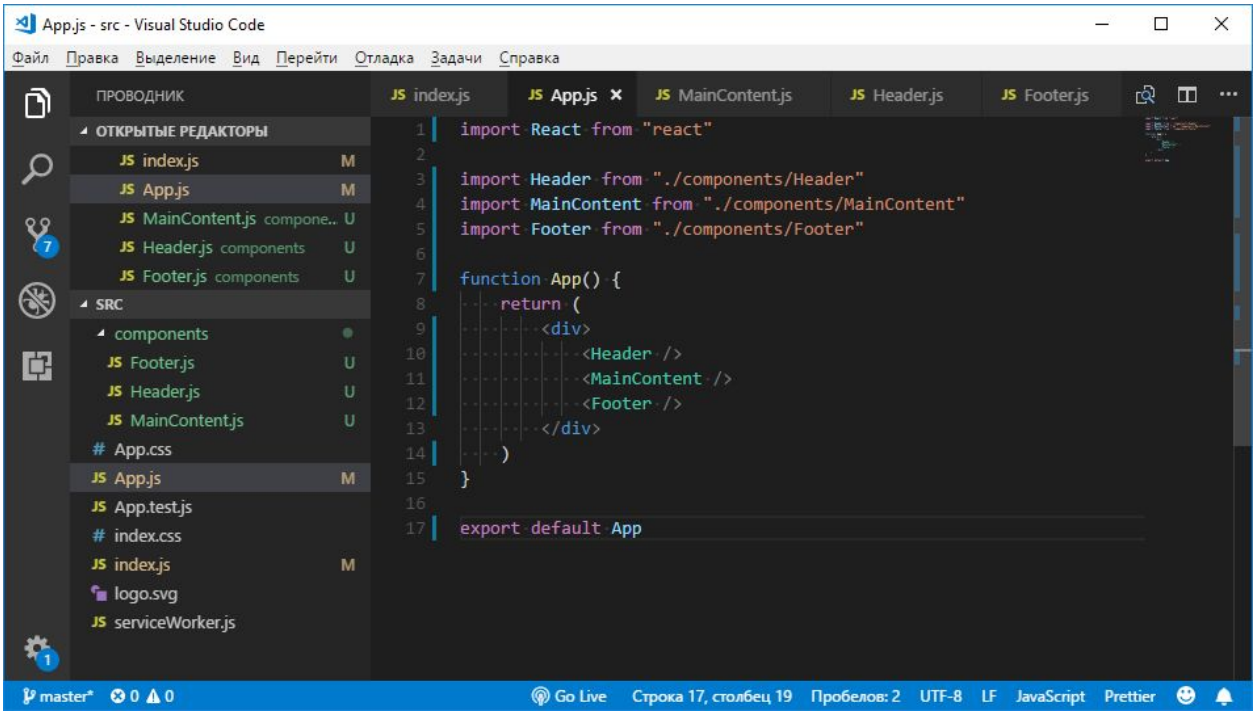
Оригинал

Сейчас мы будем работать над тем приложением, которое было создано в результате выполнения практикума на занятии 10. Вот как выглядело то, что выводило на экран это приложение.



Страница приложения в браузере

Мы хотели бы стилизовать элементы страницы. В React существует множество подходов к стилизации. Мы используем сейчас тот подход, с принципами которого вы, вероятно, уже знакомы. Он заключается в применении CSS-классов и CSS-правил, назначаемых этим классам. Взглянем на структуру этого проекта и подумаем о том, каким элементам нужно назначить классы, которые будут использоваться для их стилизации.



Проект в VSCode

Файл `index.js` ответственен за рендеринг компонента `App`. Компонент `App` выводит элемент `<div>`, в котором содержатся три других компонента — `Header`, `MainComponent` и `Footer`. А каждый из этих компонентов просто выводит по одному JSX-элементу с текстом. Именно в этих компонентах мы и будем заниматься стилизацией. Поработаем над компонентом `Header`. Напомним, что на данном этапе работы его код выглядит так:

```
import React from "react"


function Header() {
    return (
        <header>This is the header</header>
    )
}


export default Header
```

Обычно, когда работают с HTML-кодом и хотят назначить некоему элементу класс, это делается с помощью атрибута `class`. Предположим, мы собираемся назначить такой атрибут элементу `<header>`. Но тут нельзя забывать о том, что мы работаем не с HTML-кодом, а с JSX. И здесь атрибут `class` мы использовать не можем (на самом деле, воспользоваться атрибутом с таким именем можно, но делать так не рекомендуется). Вместо этого используется атрибут с именем `className`. Во многих публикациях говорится о том, что причина этого заключается в том, что `class` — это зарезервированное ключевое слово `JavaScript`. Но, на самом деле, JSX использует обычный API JavaScript для работы с DOM. Для доступа к элементам с использованием этого API применяется уже знакомая вам конструкция следующего вида:

```
document.getElementById("something")
```

Для того чтобы добавить к элементу новый класс, поступают так:

```
document.getElementById("something").className += " new-class-name"
```

В похожей ситуации удобнее пользоваться свойством элементов `classList`, в частности, из-за того, что у него есть удобные методы для добавления и удаления классов, но в данном случае это неважно. А важно то, что тут применяется свойство `className`.

Собственно говоря, нам, для назначения классов JSX-элементам, достаточно знать о том, что там, где в HTML используется ключевое слово `class`, в JSX надо пользоваться ключевым словом `className`.

Назначим элементу `<header>` класс `navbar`:

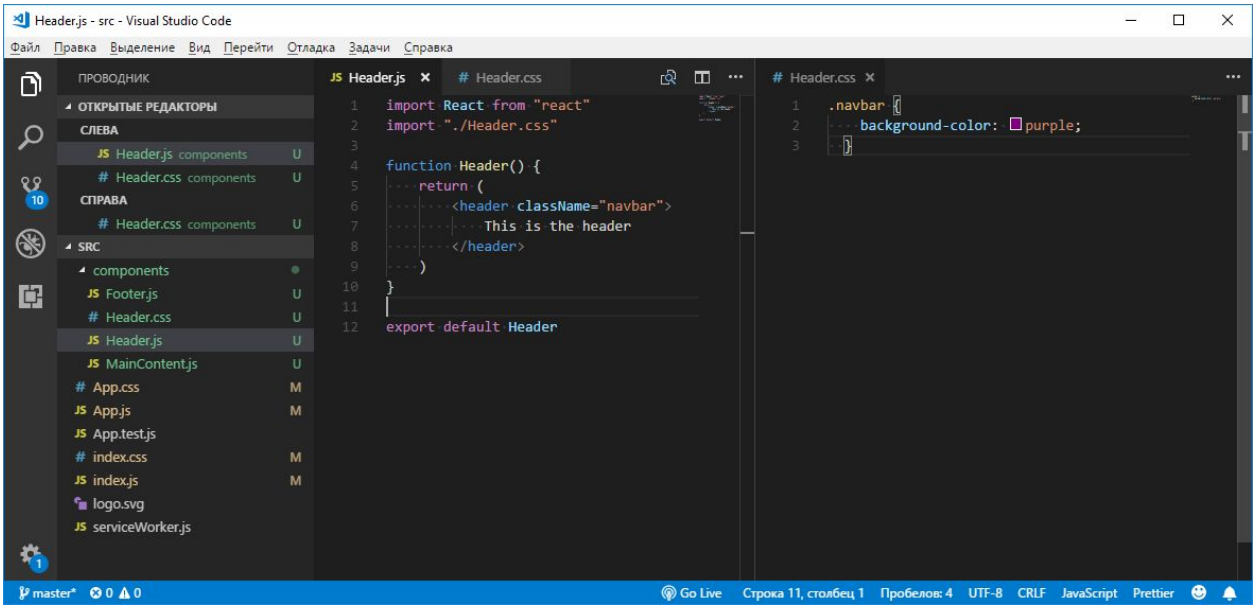
```
import React from "react"


function Header() {
    return (
        <header className="navbar">This is the header</header>
    )
}


export default Header
```

Теперь, в папке `components`, создадим файл `Header.css`. Поместим в него следующий код:

```
.navbar {
  background-color: purple;
}
```

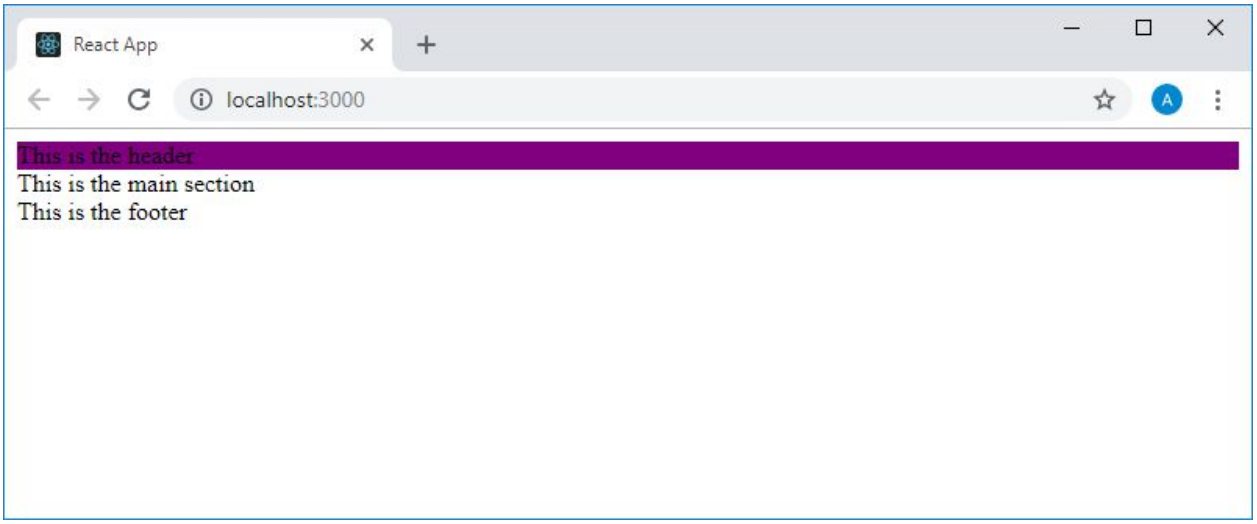
Теперь подключим этот файл в `Header.js` командой `import "./Header.css"` (этой командой, расширяющей возможности стандартной команды `import`, мы сообщаем бандлеру Webpack, который используется в проектах, созданных средствами `create-react-app`, о том, что хотим использовать `Header.css` в `Header.js`).

Вот как это будет выглядеть в VSCode.



Файл стилей и его подключение в VSCode

Всё это приведёт к тому, что внешний вид самой верхней строки, выводимой приложением на страницу, изменится.



Изменение стиля верхней строки

Тут мы использовали крайне простой стиль. Заменим содержимое файла `Header.css` на следующее:

```
.navbar {
    height: 100px;
    background-color: #333;
    color: whitesmoke;
    margin-bottom: 15px;
    text-align: center;
    font-size: 30px;
    display: flex;
```

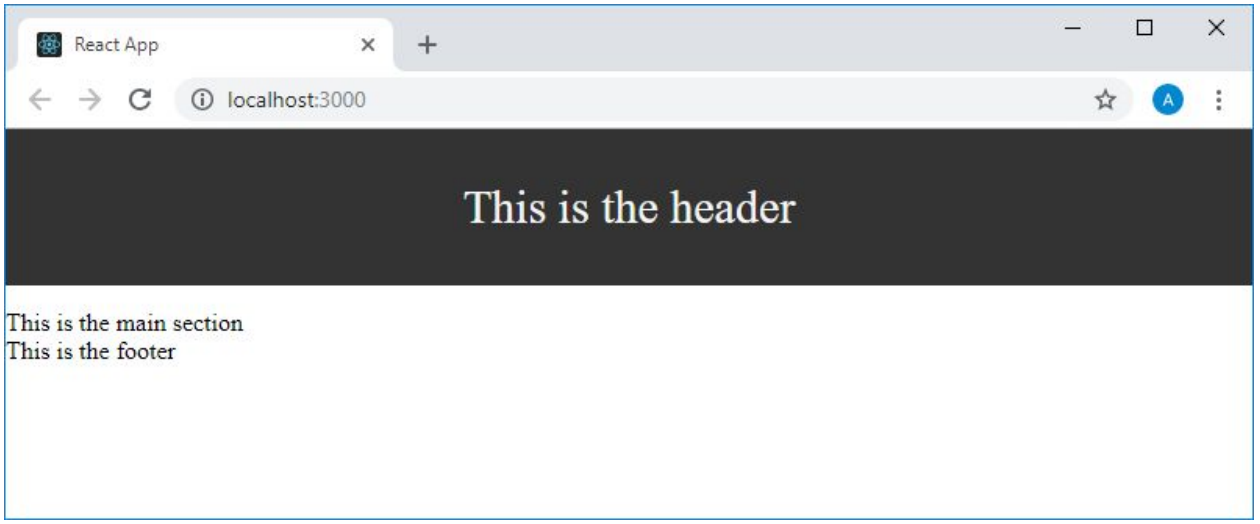
```
    justify-content: center;

    align-items: center;

}
```

Кроме того, откроем уже существующий в проекте файл `index.css` и поместим в него следующий стиль:

```
body {

    margin: 0;

}
```

Подключим этот файл в `index.js` командой `import "./index.css"`

В результате страница приложения примет вид, показанный на следующем рисунке.



Изменение стиля страницы

Обратите внимание на то, что стили, заданные в `index.css`, повлияли на все элементы страницы. Если вы для экспериментов, пользуетесь, например, неким онлайн-редактором, там работа с файлами стилей может быть организована по-особенному. Например, в таком редакторе может присутствовать единственный стандартный файл стилей, автоматически подключаемый к странице, CSS-правила, описанные в котором, будут применяться ко всем элементам страницы. В нашем простом примере вполне можно было бы поместить все стили в `index.css`.

Собственно говоря, предполагая, что вы уже знакомы с CSS, можно сказать, что здесь используется в точности такой же CSS-код, который применяется для стилизации обычных HTML-элементов. Главная особенность, о которой надо помнить, стилизуя элементы с помощью классов в React, заключается в том, что вместо атрибута элемента `class`, применяемого в HTML, используется `className`.

Кроме того, нужно отметить, что классы можно назначать лишь элементам React — таким, как `<header>`, `<p>` или `<h1>`. Если попытаться назначить имя класса экземпляру компонента — наподобие `<Header />` или `<MainContent />`, система будет вести себя совсем не так, как можно ожидать. Об этом мы ещё поговорим. Пока же просто запомните, что классы в React-приложениях назначают элементам, а не компонентам.

Вот ещё кое-что, что в начале работы с React может показаться вам сложным. Речь идёт о стилизации элементов, которые имеют разный уровень иерархии на странице. Скажем, это происходит в тех случаях, когда для стилизации используются технологии CSS Flexbox или CSS Grid. В подобных

случаях, например, при использовании Flex-вёрстки, нужно знать о том, какая сущность является flex-контейнером, и о том, какие сущности являются flex-элементами. А именно, может быть непросто понять — как именно стилизовать элементы, к каким именно элементам применять те или иные стили. Предположим, например, что элемент `<div>` из нашего компонента `App` должен быть flex-контейнером:

```
import React from "react"


import Header from "./components/Header"

import MainContent from "./components/MainContent"

import Footer from "./components/Footer"


function App() {

    return (

        <div>

            <Header />

            <MainContent />

            <Footer />

        </div>

    )

}


export default App
```

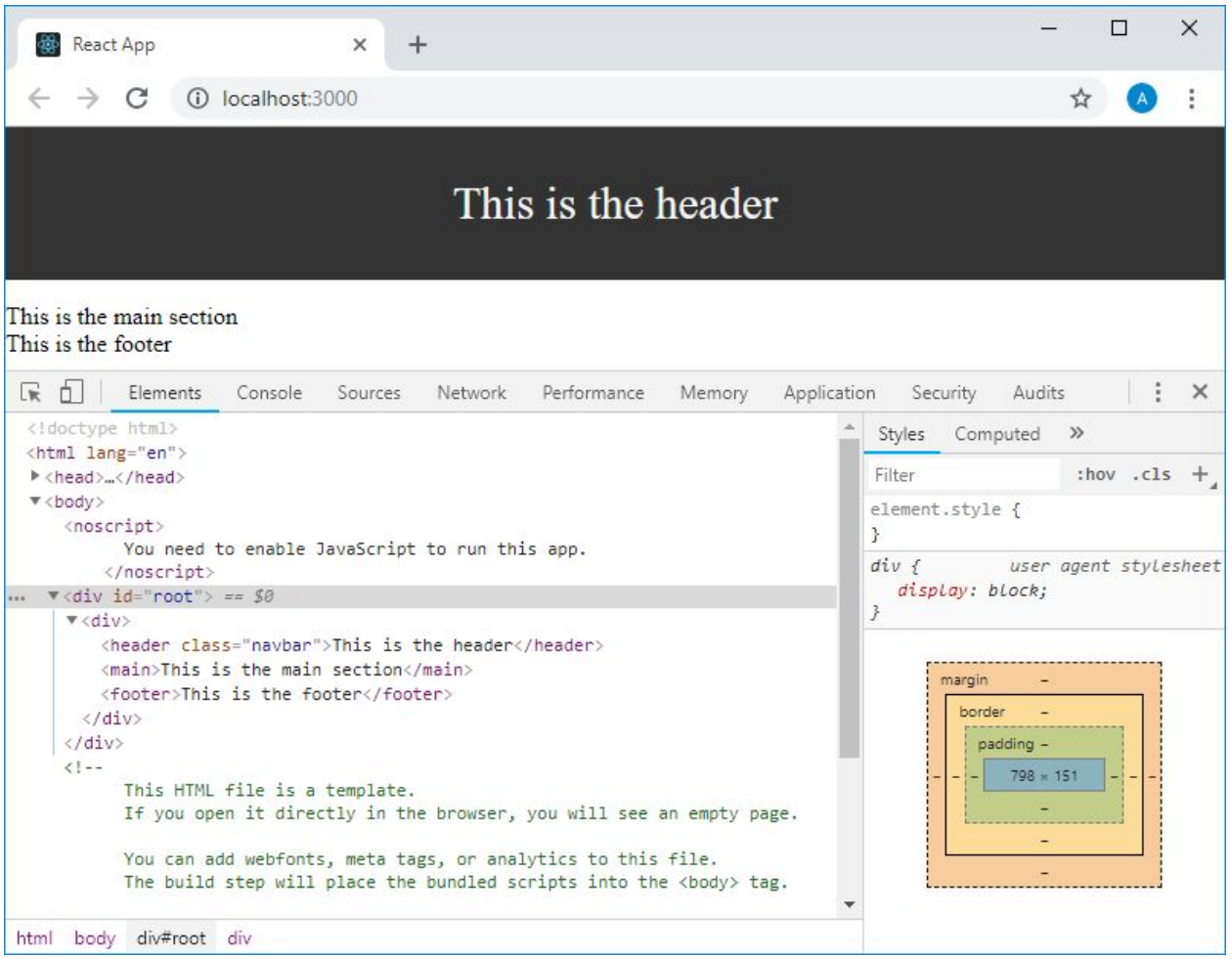
При этом flex-элементы выводятся средствами компонентов `Header`, `MainContent` и `Footer`. Взглянем, например, на код компонента `Header`:

```
import React from "react"

import "./Header.css"


function Header() {

    return (

        <header className="navbar">

            This is the header

        </header>

    )
```

```
}


export default Header
```

Элемент `<header>` должен быть прямым потомком элемента `<div>` из компонента `App`. Его и надо стилизовать как flex-элемент.

Для того чтобы разобраться в стилизации подобных конструкций, нужно учитывать то, каким будет HTML-код, сформированный приложением. Откроем вкладку `Elements` инструментов разработчика Chrome для той страницы, которая выводится в браузере при работе с проектом, созданным средствами `create-react-app`.



Код страницы

Элемент `<div>` с идентификатором `root` — это тот элемент страницы `index.html`, к которому мы обращаемся в методе `ReactDOM.render()`, вызываемом в файле `index.js`. В него выводится разметка, формируемая компонентом `App`, а именно — следующий элемент `<div>`, который содержит элементы `<header>`, `<main>` и `<footer>`, формируемые соответствующими компонентами.

То есть, анализируя код React-приложения, приведённый выше, можно считать, что конструкция `<Header />` в компоненте `App` заменяется на конструкцию `<header className="navbar">This is the header</header>`. Понимание этого факта позволяет использовать сложные схемы стилизации элементов страниц.

На этом мы завершаем первое знакомство со стилизацией React-приложений. Рекомендуется поэкспериментировать с тем, что вы только что узнали. Например — попробуйте самостоятельно стилизовать элементы, выводимые компонентами `<MainContent />` и `<Footer />`.

Учебный курс по React, часть 6: о некоторых особенностях курса, JSX и JavaScript

Занятие 13. О некоторых особенностях курса

Оригинал

Прежде чем мы продолжим занятия, мне хотелось бы немного рассказать о некоторых особенностях кода, который я демонстрирую в этом курсе. Вы могли обратить внимание на то, что в коде обычно не используются точки с запятой. Например, как видите, в примерах, подобных следующему, их нет:

```
import React from "react"

import ReactDOM from "react-dom"


function App() {

    return (

        <h1>Hello world!</h1>

    )

}


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

Возможно, вы привыкли ставить точки с запятой везде, где это возможно. Тогда, например, первые две строчки предыдущего фрагмента кода выглядели бы так:

```
import React from "react";

import ReactDOM from "react-dom";
```

Я же недавно решил, что буду обходиться без них, в результате у меня и получается такой код, который вы видите в примерах. Конечно, в JavaScript есть конструкции, в которых без точек с запятой не обойтись. Скажем, при описании цикла for, синтаксис которого выглядит так:

```
for ([инициализация]; [условие]; [финальное выражение])выражение
```

Но в большинстве случаев без точек с запятой в конце строк обойтись можно. И их отсутствие в коде никак не нарушит его работу. На самом деле, вопрос использования в коде точек с запятой — это вопрос личных предпочтений программиста.

Ещё одна особенность кода, который я пишу, заключается в том, что хотя ES6 технически позволяет использовать стрелочные функции в тех случаях, когда функции объявляют с использованием ключевого слова `function`, я этим не пользуюсь.

Например, код, приведённый выше, можно переписать так:

```
import React from "react"

import ReactDOM from "react-dom"


const App = () => <h1>Hello world!</h1>


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

Но я к подобному не привык. Полагаю, что стрелочные функции чрезвычайно полезны в определённых случаях, в которых особенности этих функций не мешают правильной работе кода. Например, тогда, когда обычно пользуются анонимными функциями, или когда пишут методы классов. Но я предпочитаю пользоваться традиционными функциями. Многие, при описании функциональных компонентов, пользуются стрелочными функциями. Я согласен с тем, что у такого подхода есть преимущества перед использованием традиционных конструкций. При этом я не стремлюсь навязывать какой-то определённый способ объявления функциональных компонентов.

Занятие 14. JSX и JavaScript

Оригинал

На следующих занятиях мы будем говорить о встроенных стилях. Прежде чем перейти к этим темам, нам нужно уточнить некоторые особенности взаимодействия JavaScript и JSX. Вы уже знаете, что, пользуясь возможностями React, мы можем, из обычного JavaScript-кода, возвращать конструкции, напоминающие обычную HTML-разметку, но являющиеся JSX-кодом. Такое происходит, например, в коде функциональных компонентов.

Что если имеется некая переменная, значение которой нужно подставить в возвращаемый функциональным компонентом JSX-код?

Предположим, у нас есть такой код:

```
import React from "react"

import ReactDOM from "react-dom"


function App() {

  return (

    <h1>Hello world!</h1>

  )

}


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

Добавим в функциональный компонент пару переменных, содержащих имя и фамилию пользователя.

```
function App() {
```

```
  const firstName = "Bob"

  const lastName = "Ziroll"


  return (
    <h1>Hello world!</h1>
  )
}
```

Теперь мы хотим, чтобы то, что возвращает функциональный компонент, оказалось бы не заголовком первого уровня с текстом `Hello world!`, а заголовком, содержащим приветствие вида `Hello Bob Ziroll!`, которое сформировано с использованием имеющихся в компоненте переменных.

Попробуем переписать то, что возвращает компонент, так:

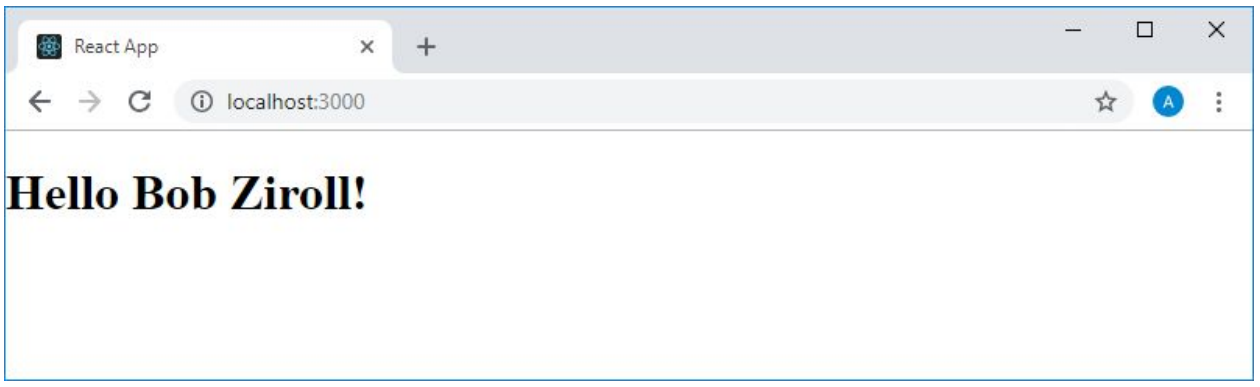
```
<h1>Hello firstName + " " + lastName!</h1>
```

Если взглянуть на то, что появится на странице после обработки подобного кода, то окажется, что выглядит это не так, как нам нужно. А именно, на страницу попадёт текст `Hello firstName + " " + lastName!`. При этом, если для запуска примера используется стандартный проект, созданный средствами `create-react-app`, нас предупредят о том, что константам `firstName` и `lastName` назначены значения, которые нигде не используются. Правда, это не помешает появлению на странице текста, который представляет собой в точности то, что было возвращено функциональным компонентом, без подстановки вместо того, что нам казалось именами переменных, их значений. Имена переменных в таком виде система считает обычным текстом.

Зададимся вопросом о том, как воспользоваться возможностями JavaScript в JSX-коде. На самом деле, сделать это довольно просто. В нашем случае достаточно лишь заключить то, что должно быть интерпретировано как JavaScript-код, в фигурные скобки. В результате то, что возвращает компонент, будет выглядеть так:

```
<h1>Hello {firstName + " " + lastName}!</h1>
```

При этом на страницу попадёт текст `Hello Bob Ziroll!`. В этих фрагментах JSX-кода, выделенных фигурными скобками, можно использовать обычные JavaScript-конструкции. Вот как выглядит в браузере то, что выведет этот код:



Страница, разметка которой сформирована средствами JSX и JavaScript

Так как при работе со строками в современных условиях, в основном, применяются возможности ES6, перепишем код с их использованием. А именно, речь идёт о шаблонных литералах, оформляемых с

помощью обратных кавычек (` `). Такие строки могут содержать конструкции вида `${выражение}`. Стандартное поведение шаблонных литералов предусматривает вычисление содержащихся в фигурных скобках выражений и преобразование того, что получится, в строку. В нашем случае это будет выглядеть так:

```
<h1>Hello {`${firstName} ${lastName}`}!</h1>
```

Обратите внимание на то, что имя и фамилия разделены пробелом, который интерпретируется здесь как обычный символ. Результат выполнения этого кода будет таким же, как было показано выше. В общем-то, самое главное, что вы должны сейчас понять, заключается, в том, что то, что заключено в фигурные скобки, находящиеся в JSX-коде — это обычный JS.

Рассмотрим ещё один пример. А именно, перепишем наш код так, чтобы, если его вызывают утром, он выводил бы текст `Good morning`, если днём — `Good afternoon`, а если вечером — `Good night`. Для начала напишем программу, которая сообщает о том, который сейчас час. Вот код функционального компонента `App`, который решает эту задачу:

```
function App() {

  const date = new Date()



  return (

    <h1>It is currently about {date.getHours() % 12} o'clock!</h1>

  )

}
```

Тут создан новый экземпляр объекта `Date`. В JSX используется JavaScript-код, благодаря которому мы узнаём, вызвав метод `date.getHours()`, который сейчас час, после чего, вычисляя остаток от деления этого числа на `12`, приводим время к 12-часовому формату. Похожим образом можно, проверив время, сформировать нужную нам строку. Например, это может выглядеть так:

```
function App() {

  const date = new Date()

  const hours = date.getHours()

  let timeOfDay



  if (hours < 12) {

    timeOfDay = "morning"

  } else if (hours >= 12 && hours < 17) {

    timeOfDay = "afternoon"

  } else {

    timeOfDay = "night"
```

```
  }


  return (
    <h1>Good {timeOfDay}!</h1>
  )
}
```

Тут имеется переменная `timeOfDay`, а анализируя текущее время с помощью конструкции `if`, мы узнаём время дня и записываем его в эту переменную. После этого мы используем переменную в возвращаемом компонентом JSX-коде.

Как обычно, рекомендуется поэкспериментировать с тем, что мы сегодня изучили.

Учебный курс по React, часть 7: встроенные стили

Занятие 15. Встроенные стили

Оригинал

В конце прошлого занятия мы создали компонент, который анализирует время суток и выводит соответствующее приветствие. Вот полный код файла `index.js`, реализующий этот функционал:

```
import React from "react"
import ReactDOM from "react-dom"


function App() {
  const date = new Date()
  const hours = date.getHours()
  let timeOfDay


  if (hours < 12) {
    timeOfDay = "morning"
  } else if (hours >= 12 && hours < 17) {
    timeOfDay = "afternoon"
  } else {
    timeOfDay = "night"
  }
```

```
  return (
    <h1>Good {timeOfDay}!</h1>
  )
}


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

Теперь нам нужно стилизовать то, что этот код выводит на страницу. При этом мы собираемся использовать здесь подход, который отличается от рассмотренной ранее стилизации элементов с применением CSS-классов. А именно, речь идёт о применении HTML-атрибута `style`. Посмотрим, что произойдёт, если воспользоваться такой конструкцией:

```
<h1 style="color: #FF8C00">Good {timeOfDay}!</h1>
```

На самом деле — ничего хорошего. Текст на страницу не попадёт, вместо этого будет выведено сообщение об ошибке. Суть его сводится к тому, что тут, при настройке стилей, ожидается не строковое значение, а объект, содержащий пары вида `ключ: значение`, где ключами являются имена CSS-свойств, а значениями — их значения.

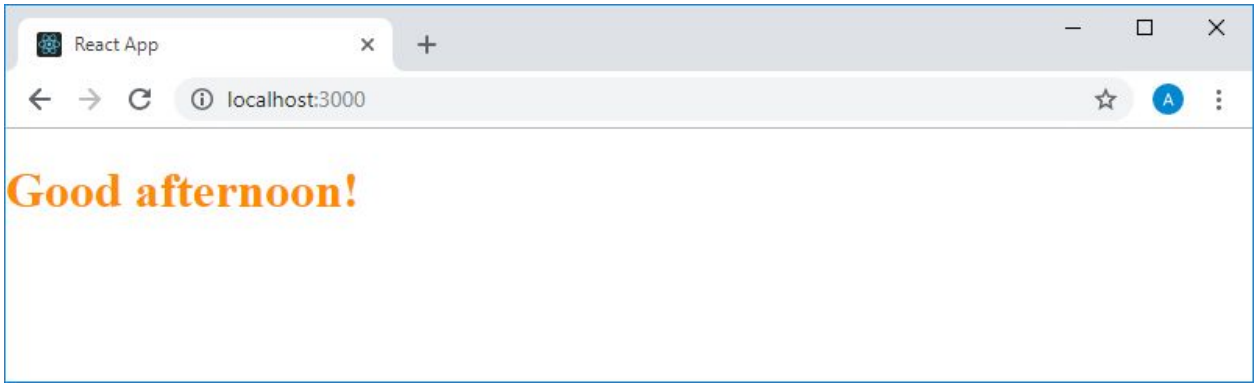
Пытаясь пользоваться HTML-атрибутами в JSX-коде, мы не должны забывать о том, что то, с чем мы работаем, хотя и похоже на обычный HTML-код, им не является. В результате то, как тут будет выглядеть та или иная конструкция, может отличаться от того, что принято в HTML. В данном случае нам нужен обычный JavaScript-объект, содержащий описание стилей. Вооружившись этой идеей, перепишем вышеприведённый фрагмент кода так:

```
<h1 style={color: "#FF8C00"}>Good {timeOfDay}!</h1>
```

Так, к сожалению, наш код тоже не заработает. В результате его выполнения снова выведется сообщение об ошибке, правда, не такое, как в прошлый раз. Оно теперь сообщает о том, что там, где система может ожидать фигурную скобку, она находит что-то другое. Для того чтобы решить эту проблему, нужно вспомнить о том, о чём мы говорили на предыдущем занятии. А именно — о том, что JavaScript-код, встраиваемый в JSX, должен быть заключён в фигурные скобки. Те фигурные скобки, которые уже имеются в нашем коде, используются для описания объектного литерала, а не для выделения JS-кода. Исправим это:

```
<h1 style={{color: "#FF8C00"}}>Good {timeOfDay}!</h1>
```

Теперь компонент формирует на странице именно то, что нужно.



Стилизованный текст, выводимый на страницу компонентом

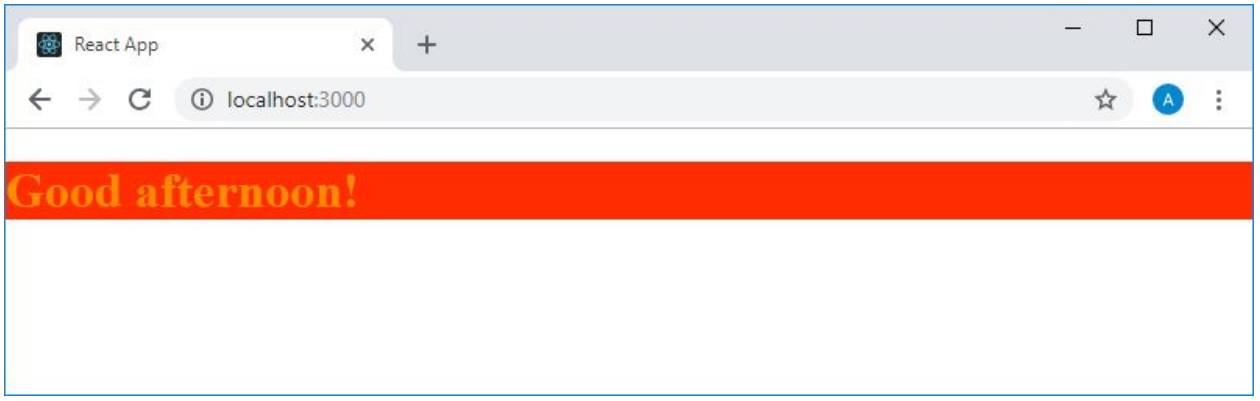
Что если мы решим продолжить стилизацию этого текста? Для этого нам нужно вспомнить о том, что стили мы описываем в JS-объекте, а это значит, что в этот объект надо добавить дополнительные пары вида `ключ: значение`. Например, попытаемся стилизовать таким образом фон текста, использовав CSS-свойство `background-color` и дополним код так:

```
<h1 style={{color: "#FF8C00", background-color: "#FF2D00"}}>Good
{timeOfDay}!</h1>
```

Такая конструкция приведёт к сообщению об ошибке. Дело тут в том, что стили мы описываем с помощью обычного JS-объекта, а в JavaScript переменные и имена свойств объектов (идентификаторы) не могут содержать символ «-», тире. На самом деле, это ограничение можно обойти, например, заключив имя свойства объекта в кавычки, но в нашем случае это к делу не относится. Мы в подобных ситуациях, когда имена свойств CSS содержат тире, этот символ убираем и делаем первую букву слова, следующего за ним, заглавной. Несложно заметить, что при таком подходе имена свойств CSS будут записываться в верблюжьем стиле — так же, как в JavaScript принято записывать имена переменных, состоящие из нескольких слов. Перепишем код:

```
<h1 style={{color: "#FF8C00", backgroundColor: "#FF2D00"}}>Good
{timeOfDay}!</h1>
```

Посмотрим на результаты его работы.



Стилизованный текст, выводимый на страницу компонентом

В процессе стилизации текста код объекта со стилями становится всё длиннее. Работать с ним неудобно. Если попытаться разбить этот код на несколько строк, ничего хорошего тоже не получится. Поэтому мы вынесем описание объекта со стилями из JSX-кода, создав константу с именем `styles`, записав в неё объект и использовав её имя в JSX. В результате у нас получится следующее:

```
const styles = {
    color: "#FF8C00",
    backgroundColor: "#FF2D00"
  }

  return (
    <h1 style={styles}>Good {timeOfDay}!</h1>
  )
```

Этот код работает в точности так же, как и вышеописанный, но такой подход оказывается очень удобным, когда возникает необходимость в добавлении в объект новых стилей. Это не приводит к разрастанию кода, возвращаемого компонентом.

Как видите, сейчас значения CSS-свойств задаются в объекте `styles` в виде строк. При работе с этим объектом стоит учесть некоторые особенности, которые, в частности, касаются свойств, значения которых задаются в виде чисел. Например, это свойство `fontSize` (выглядящее как `font-size` в CSS). Так, это свойство можно задать в виде обычного числа, а не строки, заключённой в кавычки. Например, вполне допустима такая конструкция:

```
const styles = {

    color: "#FF8C00",

    backgroundColor: "#FF2D00",

    fontSize: 24

}
```

Здесь число `24` будет интерпретировано как размер шрифта, указываемый в пикселях. Если единицу измерения нужно указать в явном виде — нам снова нужно будет пользоваться строковыми значениями свойств. Например, следующий фрагмент кода аналогичен, в плане воздействия на размер шрифта, предыдущему, но единица измерения размера тут указана в явном виде:

```
const styles = {

    color: "#FF8C00",

    backgroundColor: "#FF2D00",

    fontSize: "24px"

}
```

Здесь мы указали размер в пикселях, но при необходимости в подобных конструкциях можно использовать и другие единицы измерения.

Говоря о встроенных стилях нельзя не упомянуть и об ограничениях этого подхода. Так, если в стили нужно добавить префиксы браузеров, это может оказаться немного более сложной задачей чем добавление других стилей. А вот что-то наподобие псевдоклассов, таких как `:hover`, не поддерживается. Если вам это понадобится на данном этапе освоения React — лучше всего будет воспользоваться стилизацией элементов с использованием CSS-классов. А в будущем вам, вероятно, удобнее всего будет пользоваться для подобных целей специализированными библиотеками вроде styled-components. Но сейчас мы ограничимся встроенными стилями и стилизацией элементов с помощью CSS-классов.

Возможно, после того, как вы узнали об этом ограничении встроенных стилей, вы зададитесь вопросом о том, зачем пользоваться ими, если CSS-классы позволяют добиться того же эффекта и обладают более обширными возможностями. Одна из причин использования встроенных стилей в React заключается в том, что такие стили можно формировать динамически. При этом то, каким будет тот или иной стиль, определяется средствами JavaScript-кода. Перепишем наш пример так, чтобы цвет текста менялся бы в зависимости от времени суток, в которое выводится сообщение.

Вот полный код компонента, в котором используется динамическое формирование стилей.

```
function App() {
  const date = new Date()
  const hours = date.getHours()
  let timeOfDay
  const styles = {
    fontSize: 30
  }

  if (hours < 12) {
    timeOfDay = "morning"
    styles.color = "#04756F"
  } else if (hours >= 12 && hours < 17) {
    timeOfDay = "afternoon"
    styles.color = "#2E0927"
  } else {
    timeOfDay = "night"
    styles.color = "#D90000"
  }

  return (
    <h1 style={styles}>Good {timeOfDay}!</h1>
  )
}
```

Обратите внимание на то, что объявление константы `styles` теперь находится перед блоком `if`. В объекте, определяющем стиль, установлен лишь размер шрифта надписи — `30` пикселей. Затем в объект добавляют свойство `color`, значение которого зависит от времени суток. Напомним, что речь идёт о совершенно обычном объекте JavaScript, а такие объекты поддерживают добавление и изменение свойств после их создания. После того, как стиль сформирован, он применяется при выводе текста. Для того чтобы быстро протестировать все ветви условного оператора, можно, при инициализации константы `date`, передать конструктору объекта типа `Date` желаемые дату и время. Например, это может выглядеть так:

```
const date = new Date(2018, 6, 31, 15)
```

Собственно говоря, смысл этого всего заключается в том, что динамические данные могут воздействовать на то, как выглядят элементы, формируемые компонентами. Это открывает перед разработчиком большие возможности.

Учебный курс по React, часть 8: продолжение работы над TODO-приложением, знакомство со свойствами компонентов

Занятие 16. Практикум. TODO-приложение. Этап №2

Оригинал

Задание

1. Выполняя предыдущий практикум, вы создали React-приложение, компонент `App` которого выводит набор пар элементов — флажков (элементов `<input type="checkbox" />`) и их описаний (элементов `<p> </p>`). Оформите элементы этого набора в виде самостоятельного компонента — `<TodoItem />` и используйте его для формирования списка в компоненте `App`. При этом пока не обращайте внимания на то, что все элементы этого списка будут выглядеть одинаково (позже мы поговорим о том, как наполнить их разными данными).
2. Стилизуйте страницу так, как вам захочется, используя CSS-файлы, встроенные стили, или комбинацию из этих методов стилизации React-приложений.

Решение

Здесь предполагается, что вы продолжаете работу над приложением, основанном на стандартном проекте, который создан средствами `create-react-app`. Вот каким был код компонента `App` до выполнения задания.

```
import React from "react"


function App() {

    return (

        <div>

            <input type="checkbox" />

            <p>Placeholder text here</p>


            <input type="checkbox" />

            <p>Placeholder text here</p>


            <input type="checkbox" />

            <p>Placeholder text here</p>


            <input type="checkbox" />
```

```
            <p>Placeholder text here</p>
        </div>
    )
}

export default App
```

Создадим, в той же папке, в которой находится этот файл, файл `TodoItem.js`, в котором будет храниться код компонента `TodoItem`. Теперь импортируем этот файл в файл `App.js` следующей командой:

```
import TodoItem from "./TodoItem"
```

Сделать это можно и позже, когда придёт время пользоваться кодом компонента `TodoItem` (ещё пока не написанным). Этим кодом мы сейчас и займёмся. Вот каким он будет:

```
import React from "react"


function TodoItem() {
    return (
        <div>
            <input type="checkbox" />
            <p>Placeholder text here</p>
        </div>
    )
}

export default TodoItem
```

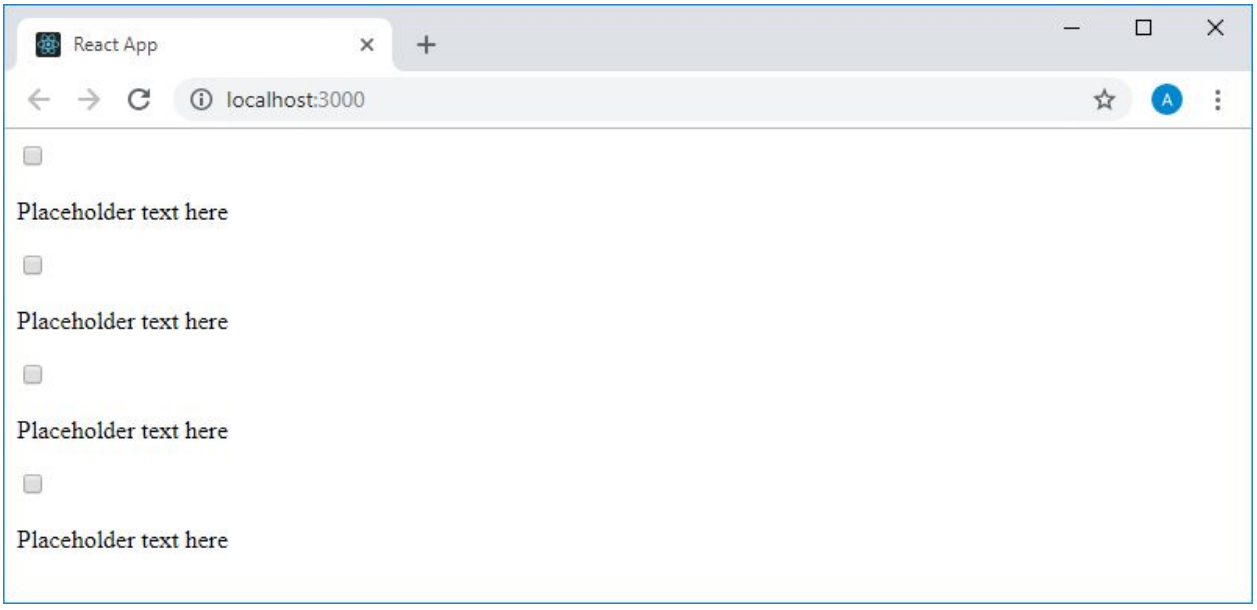
Обратите внимание на две вещи. Во-первых — этот компонент возвращает два элемента — поэтому они обёрнуты в элемент `<div>`. Во-вторых — то, что он возвращает, представляет собой копию одной из пар элементов флажок/описание из файла `App.js`.

Теперь мы снова переходим к файлу `App.js` и, вместо пар флажок/описание, используем в возвращаемой им разметке экземпляры компонента `TodoItem`:

```
import React from "react"
import TodoItem from "./TodoItem"


function App() {
```

```
    return (
        <div>
            <TodoItem />
            <TodoItem />
            <TodoItem />
            <TodoItem />
        </div>
    )
}


export default App
```

В результате страница, которую формирует приложение, будет выглядеть так, как показано ниже.



Внешний вид приложения в браузере

Собственно говоря, её внешний вид, в сравнении с предыдущим вариантом, не изменился, но то, что теперь для формирования пар элементов используется компонент, открывает перед нами большие возможности, которыми мы воспользуемся позже.

Теперь выполним второе задание, стилизовав приложение с использованием CSS-классов. Для этого приведём код компонента `App` к следующему виду:

```
import React from "react"

import TodoItem from "./TodoItem"


function App() {
```

```
    return (
        <div className="todo-list">
            <TodoItem />
            <TodoItem />
            <TodoItem />
            <TodoItem />
        </div>
    )
}

export default App
```

Тут мы назначили имя класса для элемента `<div>`. Похожим образом поработаем с кодом компонента `TodoItem`:

```
import React from "react"

function TodoItem(){

    return (
        <div className="todo-item">
            <input type="checkbox" />
            <p>Placeholder text here</p>
        </div>
    )
}

export default TodoItem
```

Теперь подключим CSS-файл `index.css`, который уже имеется в проекте, так как он создан средствами `create-react-app`, в файле `index.js`:

```
import React from "react"
import ReactDOM from "react-dom"
```

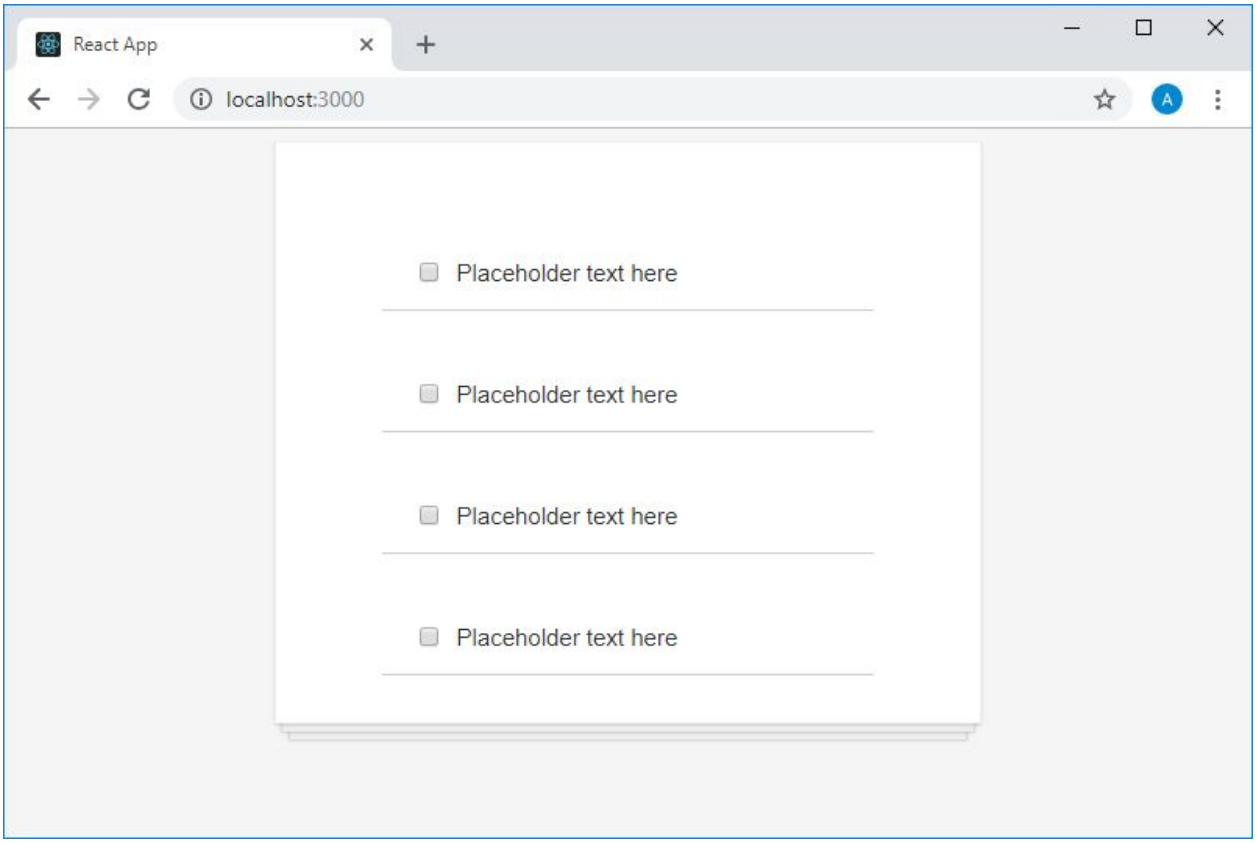
```
import "./index.css"


import App from "./App"


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

Добавим в `index.css` следующее описание стилей:

```
body {
    background-color: whitesmoke;
}


.todo-list {
    background-color: white;
    margin: auto;
    width: 50%;
    display: flex;
    flex-direction: column;
    align-items: center;
    border: 1px solid #efefef;
    box-shadow:
    /* The top layer shadow */
        0 1px 1px rgba(0,0,0,0.15),
            /* The second layer */
        0 10px 0 -5px #eee,
            /* The second layer shadow */
        0 10px 1px -4px rgba(0,0,0,0.15),
            /* The third layer */
        0 20px 0 -10px #eee,
```

```
            /* The third layer shadow */
        0 20px 1px -9px rgba(0,0,0,0.15);
    padding: 30px;
}

.todo-item {
    display: flex;
    justify-content: flex-start;
    align-items: center;
    padding: 30px 20px 0;
    width: 70%;
    border-bottom: 1px solid #cecece;
    font-family: Roboto, sans-serif;
    font-weight: 100;
    font-size: 15px;
    color: #333333;
}

input[type=checkbox] {
    margin-right: 10px;
    font-size: 30px;
}

input[type=checkbox]:focus {
    outline: 0;
}
```

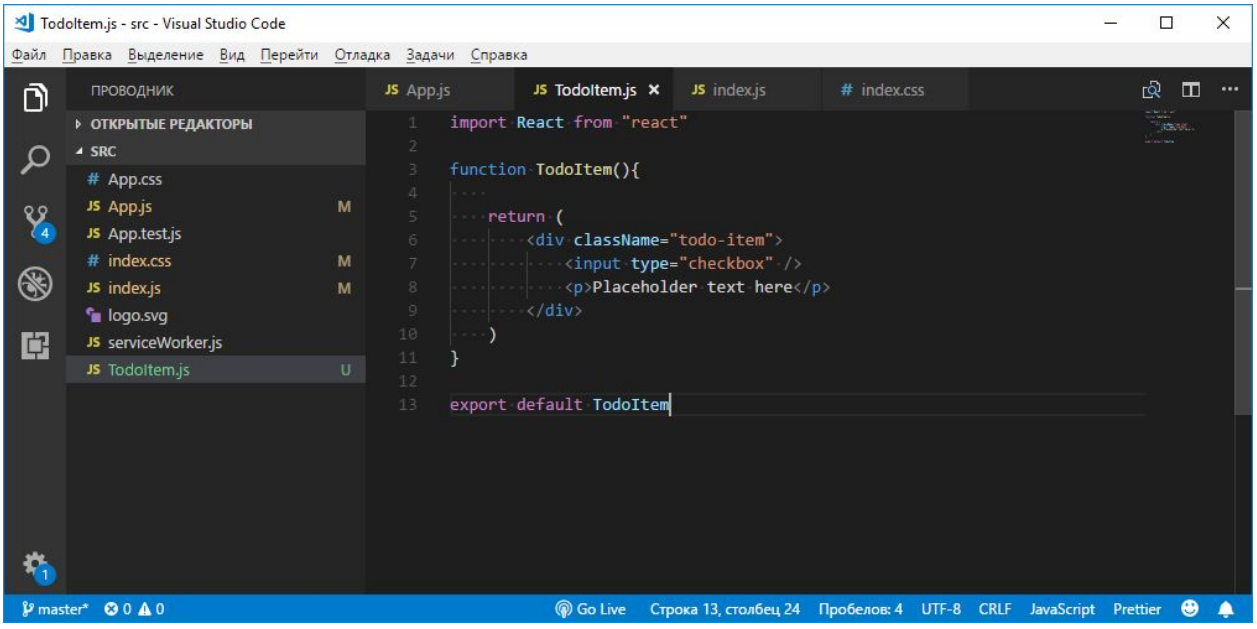
Вот как теперь будет выглядеть страница приложения в браузере.



Внешний вид приложения в браузере

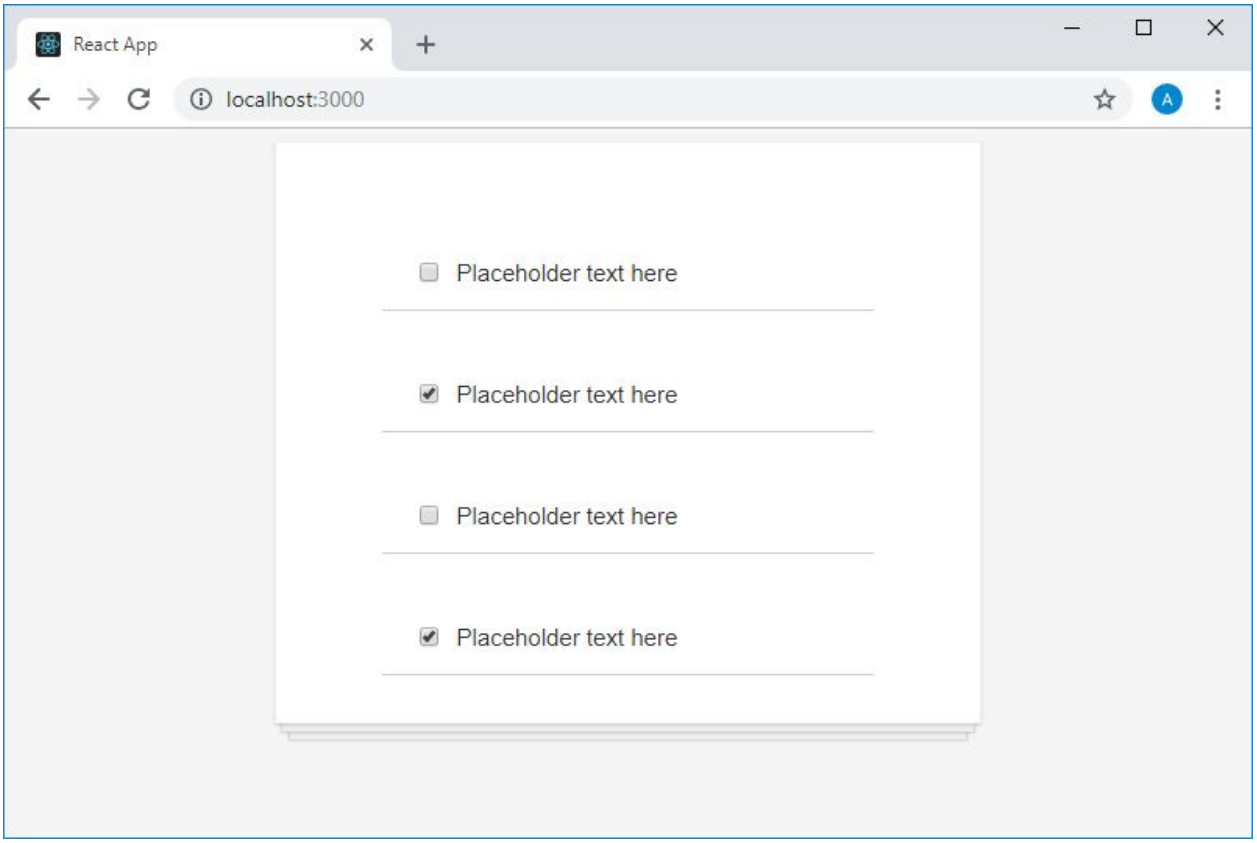
Вы можете самостоятельно проанализировать и отредактировать эти стили.

Если говорить об особенностях кода, применяемого при стилизации, обратите внимание на то, что для назначения классов элементам тут используется ключевое слово `className`, и на то, что React поддерживает встроенные стили. Вот как на данном этапе выглядит проект нашего приложения в VSCode.



Проект приложения в VSCode

Если сейчас поэкспериментировать с тем, что у нас получилось, окажется, что флажки реагируют на воздействия пользователя.



Флажки реагируют на воздействия пользователя

Но при этом код приложения ничего не знает о тех изменениях, которые с этими флажками происходят. Если коду будет известно о том, что происходит в интерфейсе приложения, это позволит нам организовать его реакцию на различные события. Например, элемент списка дел, в котором установлен флажок, указывающий на то, что соответствующая задача выполнена, можно как-то изменить. О том, как это сделать, мы поговорим на следующих занятиях.

Занятие 17. Свойства, часть 1. Атрибуты HTML-элементов

Оригинал

Поговорим о концепции свойств в React. Начнём с примера HTML-кода некоей страницы:

```
<html>

    <head></head>

    <body>

        <a>This is a link</a>

        <input />

        <img />

    </body>

</html>
```

Как видите, тут нет ничего, относящегося к React. Перед нами — обычная HTML-разметка. Обратите внимание на три элемента, присутствующих в теле страницы, описываемой этой разметкой: `<a>`, `<input />` и `<img />`, и подумайте о том, что с ними не так.

Проблема тут заключается в том, что все эти элементы не выполняют свойственных им функций. Ссылка, описываемая тегом `<a>`, никуда не ведёт. Этому тегу нужно назначить атрибут (свойство) `href`, содержащий некий адрес, по которому будет осуществлён переход при щелчке по ссылке. Та же проблема характерна и для тега `<img />` из нашего примера. Ему не назначен атрибут `src`, задающий изображение, локальное, или доступное по URL, которое будет выводить этот элемент. В результате оказывается, что для обеспечения правильной работы элементов `<a>` и `<img>` необходимо, соответственно, задать их свойства `href` и `src`. Если говорить об элементе `<input>`, то он, без настройки его атрибутов, выведет на страницу поле ввода, но в таком виде его обычно не используют, настраивая его свойства `placeholder`, `name`, `type`. Последнее свойство, например, позволяет кардинально менять внешний вид и поведение элемента `<input>`, превращая его из поля для ввода текста во флажок, в радиокнопку, или в кнопку для отправки формы. Надо отметить, что термины «атрибут» и «свойство» мы используем здесь как взаимозаменяемые.

Отредактировав вышеприведённый код, мы можем привести его к следующему виду:

```
<html>

    <head></head>

    <body>

        <a href="https://google.com">This is a link</a>

        <input placeholder="First Name" name="" type=""/>

        <img src=""/>

    </body>

</html>
```

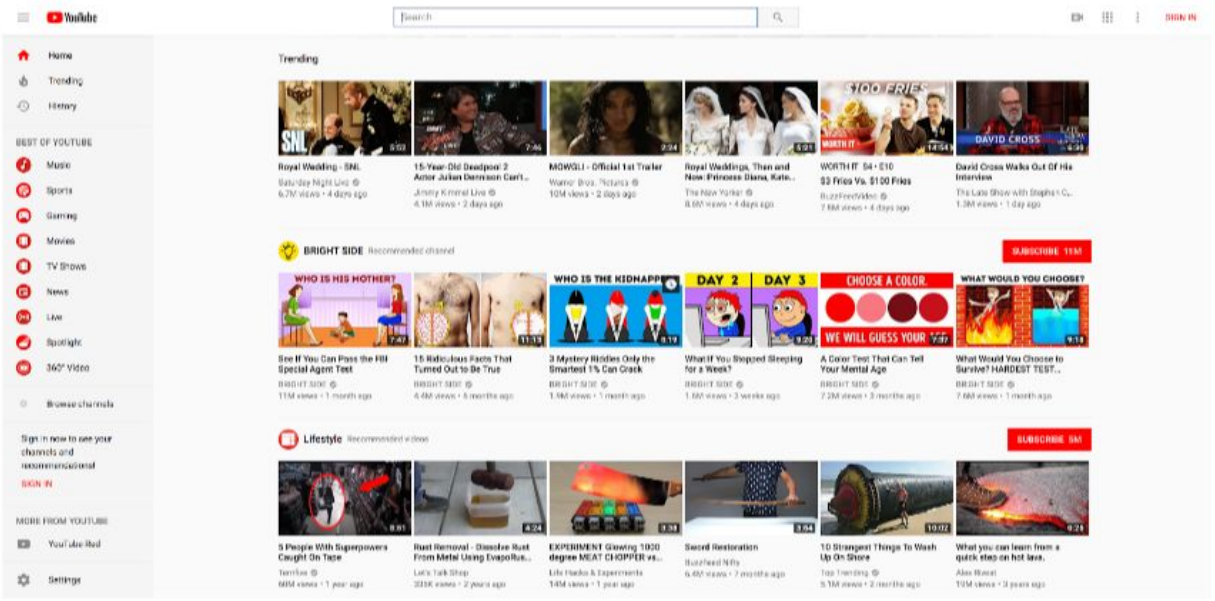
Он и в таком виде пока не вполне нормален, но тут мы, по крайней мере, задали значения для некоторых атрибутов HTML-элементов, и обозначили места, куда можно ввести значения для ещё некоторых атрибутов.

Собственно говоря, если вышеописанная концепция атрибутов HTML-элементов вам понятна, то вы без труда разберётесь и с концепцией свойств компонентов React. А именно, речь идёт о том, что мы, в React-приложениях, можем использовать компоненты собственной разработки, а не только стандартные HTML-теги. При работе с компонентами мы можем назначать им свойства, которые, при обработке их в компонентах, способны менять их поведение. Например — такие свойства позволяют настраивать внешний вид компонентов.

Занятие 18. Свойства, часть 2. Компоненты, подходящие для повторного использования

Оригинал

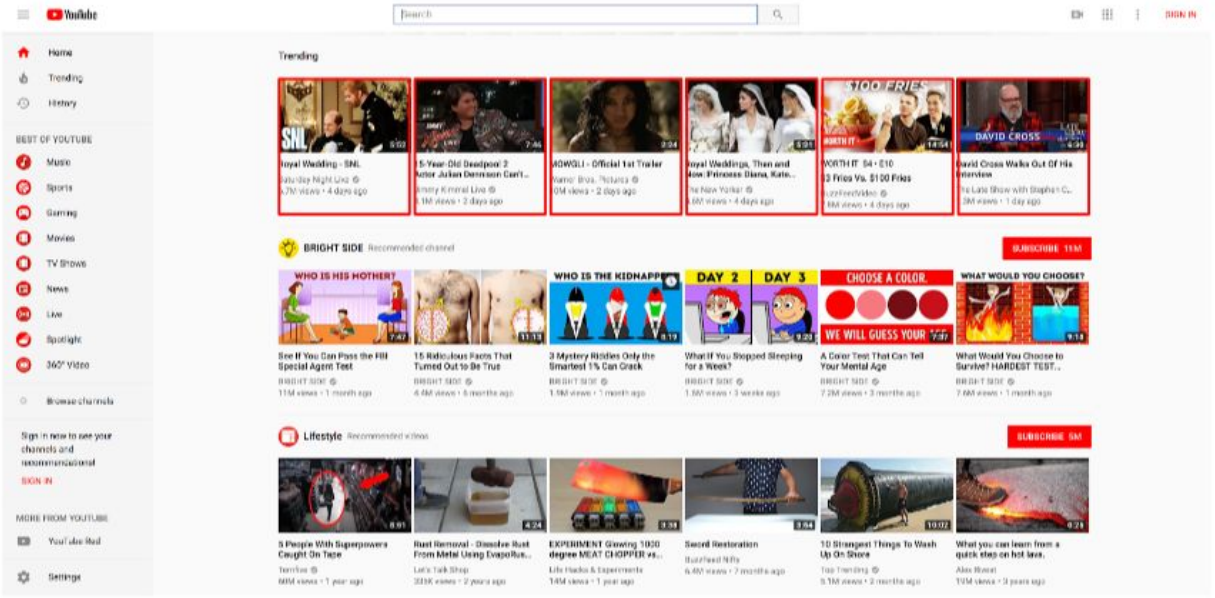
Прежде чем мы перейдём к разговору об использовании свойств в React, рассмотрим ещё одно концептуальное понятие. Взглянем на домашнюю страницу YouTube.



Домашняя страница YouTube

Уверен, что на этой странице React не использует, так как Google занимается развитием фреймворка Angular, но принципы, которые мы рассмотрим на этом примере, универсальны.

Подумайте о том, как подобную страницу можно было бы построить, используя возможности React. Пожалуй, первое, на что вы обратите внимание — это то, что данную страницу можно разделить на фрагменты, представленные самостоятельными компонентами. Например, несложно заметить, что элементы, в которых выводится информация о видеоклипах, выделенные на рисунке ниже, очень похожи друг на друга.



Домашняя страница YouTube, элементы, похожие друг на друга

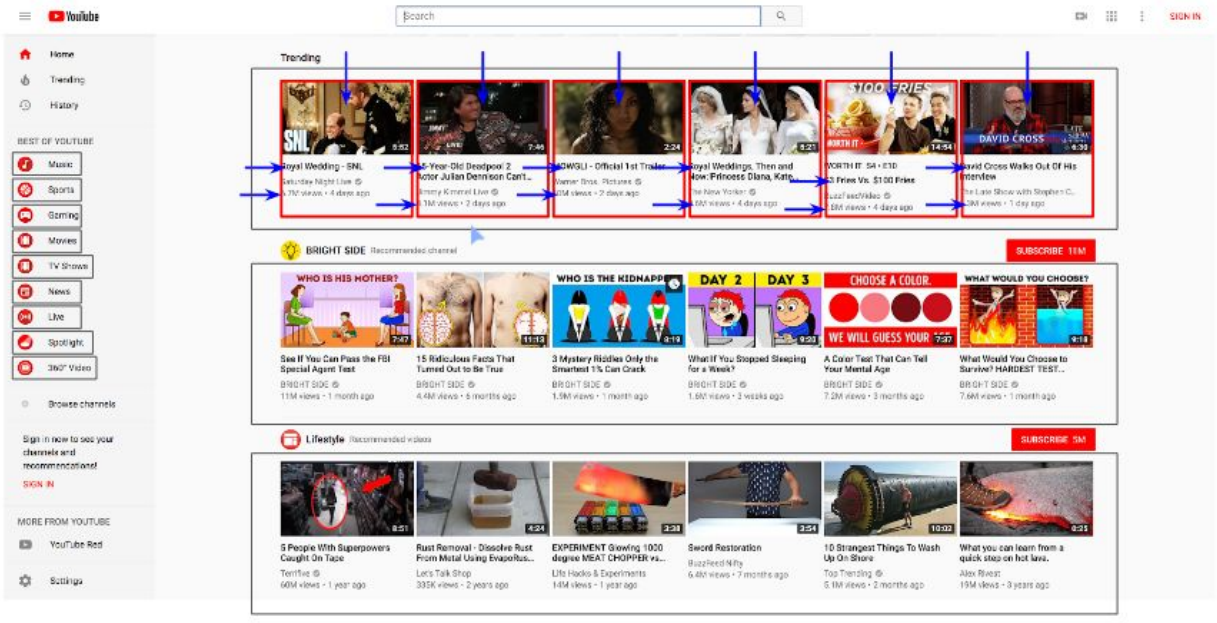
Если присмотреться к этим элементам, окажется, что в верхней части каждого из них имеется изображение, и то, что все эти изображения имеют одинаковый размер. У каждого из этих элементов есть заголовок, оформленный полужирным шрифтом и находящийся сразу под изображением. Каждый

элемент содержит сведения о количестве просмотров соответствующего видео, о дате его публикации. В правом нижнем углу каждого изображения, присутствующего на элементе, имеются сведения о длительности соответствующего видеоклипа.

Вполне понятно то, что тот, кто создал эту страницу, не занимался чем-то вроде копирования, вставки и модифицирования кода для представления каждого из выведенных на ней элементов. Если бы подобная страница была создана средствами React, то можно было бы представить себе, что карточки видеоклипов являются экземплярами некоего компонента, скажем, `<VideoTile />`. При этом такой компонент включает в себя некоторое количество других компонентов, представляющих собой изображение, заголовок, сведения о длительности клипа, и другие элементы карточки видеоклипа.

Вышесказанное приводит нас к мысли о том, что для формирования подобной страницы был разработан единственный компонент, представляющий собой карточку видеоклипа. При этом на страницу было выведено множество экземпляров этого компонента, каждый из которых отображает уникальные данные. То есть — при разработке подобного компонента нужно предусмотреть возможность изменения неких свойств, вроде URL изображения, влияющих на его внешний вид и поведение. Собственно говоря, именно этому и посвящено наше следующее занятие. Но, прежде чем мы к нему перейдём, мне хотелось бы, чтобы вы освоились с идеей использования компонентов в React.

Вспомните о том занятии, на котором мы обсуждали родительские и дочерние компоненты, и о том, что компоненты могут формировать структуры, обладающие большой глубиной вложенности. Например, в нашем случае, на странице можно выделить горизонтальные группы карточек видеоклипов, вероятно, расположенные таким образом с помощью некоего служебного компонента для вывода списков элементов. Этими элементами являются карточки видеоклипов, которые, в свою очередь, выводят некое количество других элементов, представляющих собой сведения о конкретном клипе.



Домашняя страница YouTube, родительские и дочерние компоненты

Кроме того, в левой части страницы можно видеть вертикальную группу похожих друг на друга элементов. Вероятнее всего, они являются экземплярами одного и того же компонента. При этом у каждого из них имеется собственное изображение и текст.

То, что современные фреймворки для разработки веб-интерфейсов, такие, как React, Angular или Vue, позволяют, один раз создав компонент, многократно использовать его, задавая его свойства, является одной из причин популярности этих фреймворков. Это значительно облегчает и ускоряет разработку.

Учебный курс по React, часть 9: свойства компонентов

Занятие 19. Свойства компонентов в React

Оригинал

Создадим новый проект средствами `create-react-app` и изменим код нескольких стандартных файлов из папки `src`.

Вот код файла `index.js`:

```
import React from "react"

import ReactDOM from "react-dom"

import "./index.css"


import App from "./App"


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

Вот стили, которые описаны в файле `index.css`:

```
body {
  margin: 0;
}


.contacts {
  display: flex;
  flex-wrap: wrap;
}


.contact-card {
  flex-basis: 250px;
  margin: 20px;
}
```

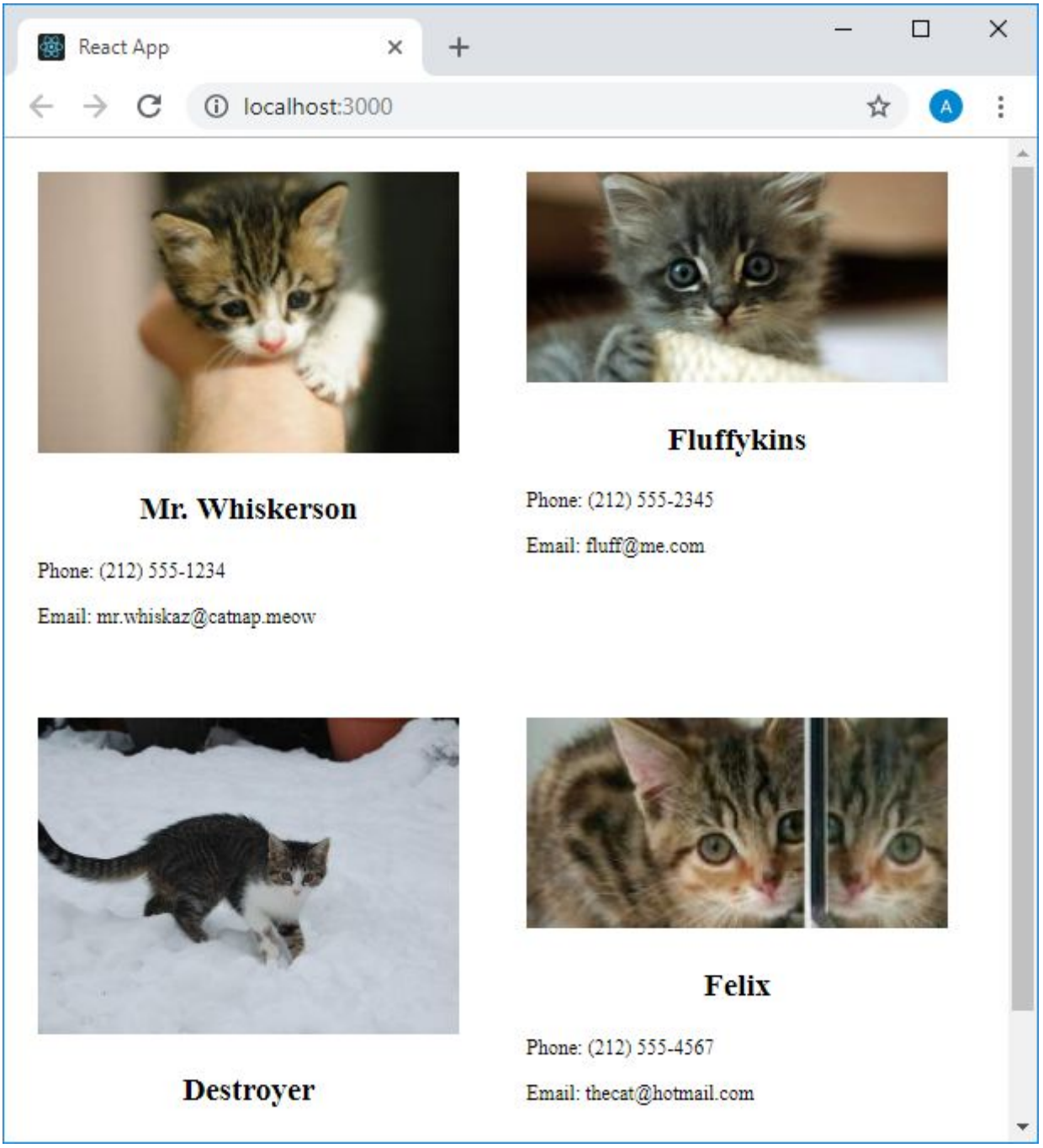
```
.contact-card > img {
  width: 100%;
  height: auto;
}

.contact-card > h3 {
  text-align: center;
}

.contact-card > p {
  font-size: 12px;
}
```

Вот код, находящийся в файле `App.js`:

```
import React from "react"

function App() {
    return (
        <div className="contacts">
            <div className="contact-card">
                <img src="http://placekitten.com/300/200"/>
                <h3>Mr. Whiskerson</h3>
                <p>Phone: (212) 555-1234</p>
                <p>Email: mr.whiskaz@catnap.meow</p>
            </div>

            <div className="contact-card">
                <img src="http://placekitten.com/400/200"/>
                <h3>Fluffykins</h3>
                <p>Phone: (212) 555-2345</p>
                <p>Email: fluff@me.com</p>
```

```
            </div>

            <div className="contact-card">
                <img src="http://placekitten.com/400/300"/>
                <h3>Destroyer</h3>
                <p>Phone: (212) 555-3456</p>
                <p>Email: ofworlds@yahoo.com</p>
            </div>

            <div className="contact-card">
                <img src="http://placekitten.com/200/100"/>
                <h3>Felix</h3>
                <p>Phone: (212) 555-4567</p>
                <p>Email: thecat@hotmail.com</p>
            </div>
        </div>
    )
}

export default App
```

Вот как будет выглядеть это приложение в браузере.



Страница приложения в браузере

Проанализировав код и внешний вид приложения можно прийти к выводу о том, что для вывода карточек со сведениями о животных хорошо было бы использовать особые компоненты. Сейчас эти элементы формируются средствами компонента `App`. Учитывая же то, о чём мы говорили на предыдущих занятиях, можно пойти и дальше — подумать об универсальном компоненте, который можно настраивать, передавая ему атрибуты или свойства.

В нашем приложении имеются карточки с изображениями кошек, их именами и контактными сведениями их владельцев (а может — и их самих) — телефоном и адресом электронной почты. Для того чтобы создать компонент, который в дальнейшем станет основой для всех подобных карточек, можно взять один из фрагментов разметки, возвращаемой компонентом `App`. Например — такой:

```
<div className="contact-card">

    <img src="http://placekitten.com/300/200"/>

    <h3>Mr. Whiskerson</h3>
```

```
    <p>Phone: (212) 555-1234</p>

    <p>Email: mr.whiskaz@catnap.meow</p>

</div>
```

`App` возвращает четыре подобных блока, каждый из них можно было бы использовать для создания самостоятельного компонента, но такой подход нас не устраивает. Поэтому создадим один компонент, который станет основой всех карточек, выводимых приложением. Для этого создадим в папке `src` новый файл компонента — `ContactCard.js` и поместим в него код, который возвращает первый элемент `<div>`, возвращаемый компонентом `App`, код которого приведён выше. Вот каким будет код компонента `ContactCard`:

```
import React from "react"


function ContactCard() {

    return (

        <div className="contact-card">

            <img src="http://placekitten.com/300/200"/>

            <h3>Mr. Whiskerson</h3>

            <p>Phone: (212) 555-1234</p>

            <p>Email: mr.whiskaz@catnap.meow</p>

        </div>

    )

}


export default ContactCard
```

Ясно, что если создать несколько экземпляров этого компонента, то все они будут содержать одни и те же данные, так как эти данные жёстко заданы в коде компонента. А нам хотелось бы, чтобы, при создании разных экземпляров этого компонента, можно было бы настраивать выводимые им данные. Речь идёт о том, чтобы компоненту можно было бы передавать некие свойства, которыми он потом сможет воспользоваться.

Мы работаем с функциональными компонентами, которые представляют собой обычные JS-функции, в которых, благодаря использованию библиотеки React, можно использовать особые конструкции. Как известно, функции могут принимать аргументы, хотя их можно использовать и без аргументов. Аналогией нашего компонента `ContactCard`, в том виде, в котором он сейчас существует, может стать такая вот простая функция, которая, ничего не принимая, просто возвращает сумму двух чисел:

```
function addNumbers() {

    return 1 + 1

}
```

Её можно использовать для того, чтобы узнать сумму чисел 1 и 1, но, например, для того, чтобы сложить 1 и 2, используя функции, которые не принимают никаких входных данных, нам пришлось бы писать новую функцию. Совершенно очевидно то, что такой подход приведёт к огромным неудобствам при необходимости сложения разных чисел, поэтому в подобной ситуации будет разумным создать универсальную функцию для сложения чисел, которая принимает два числа и возвращает их сумму:

```
function addNumbers(a, b) {

    return a + b

}
```

То, что возвращает такая функция, будет зависеть от того, какие аргументы ей передали при вызове. Создавая React-компоненты мы можем пойти точно таким же путём.

Импортируем в файл `App.js` компонент `ContactCard` и вернём четыре его экземпляра, не удаляя пока код, который формирует карточки на странице приложения:

```
import React from "react"

import ContactCard from "./ContactCard"


function App() {

    return (

        <div className="contacts">

            <ContactCard />

            <ContactCard />

            <ContactCard />

            <ContactCard />


            <div className="contact-card">

                <img src="http://placekitten.com/300/200"/>

                <h3>Mr. Whiskerson</h3>

                <p>Phone: (212) 555-1234</p>

                <p>Email: mr.whiskaz@catnap.meow</p>

            </div>


            <div className="contact-card">

                <img src="http://placekitten.com/400/200"/>

                <h3>Fluffykins</h3>
```

```
                <p>Phone: (212) 555-2345</p>
                <p>Email: fluff@me.com</p>
            </div>

            <div className="contact-card">
                <img src="http://placekitten.com/400/300"/>
                <h3>Destroyer</h3>
                <p>Phone: (212) 555-3456</p>
                <p>Email: ofworlds@yahoo.com</p>
            </div>

            <div className="contact-card">
                <img src="http://placekitten.com/200/100"/>
                <h3>Felix</h3>
                <p>Phone: (212) 555-4567</p>
                <p>Email: thecat@hotmail.com</p>
            </div>
        </div>
    )
}

export default App
```

Теперь поработаем над кодом, используемым для создания экземпляров компонента `ContactCard`. Создавая обычные HTML-элементы, мы можем настраивать их атрибуты, влияющие на их поведение и внешний вид. Имена этих атрибутов жёстко заданы стандартом. В случае с компонентами можно воспользоваться точно таким же подходом, с той только разницей, что имена атрибутов мы придумываем сами, и сами же решаем — как именно они будут использованы в коде компонента.

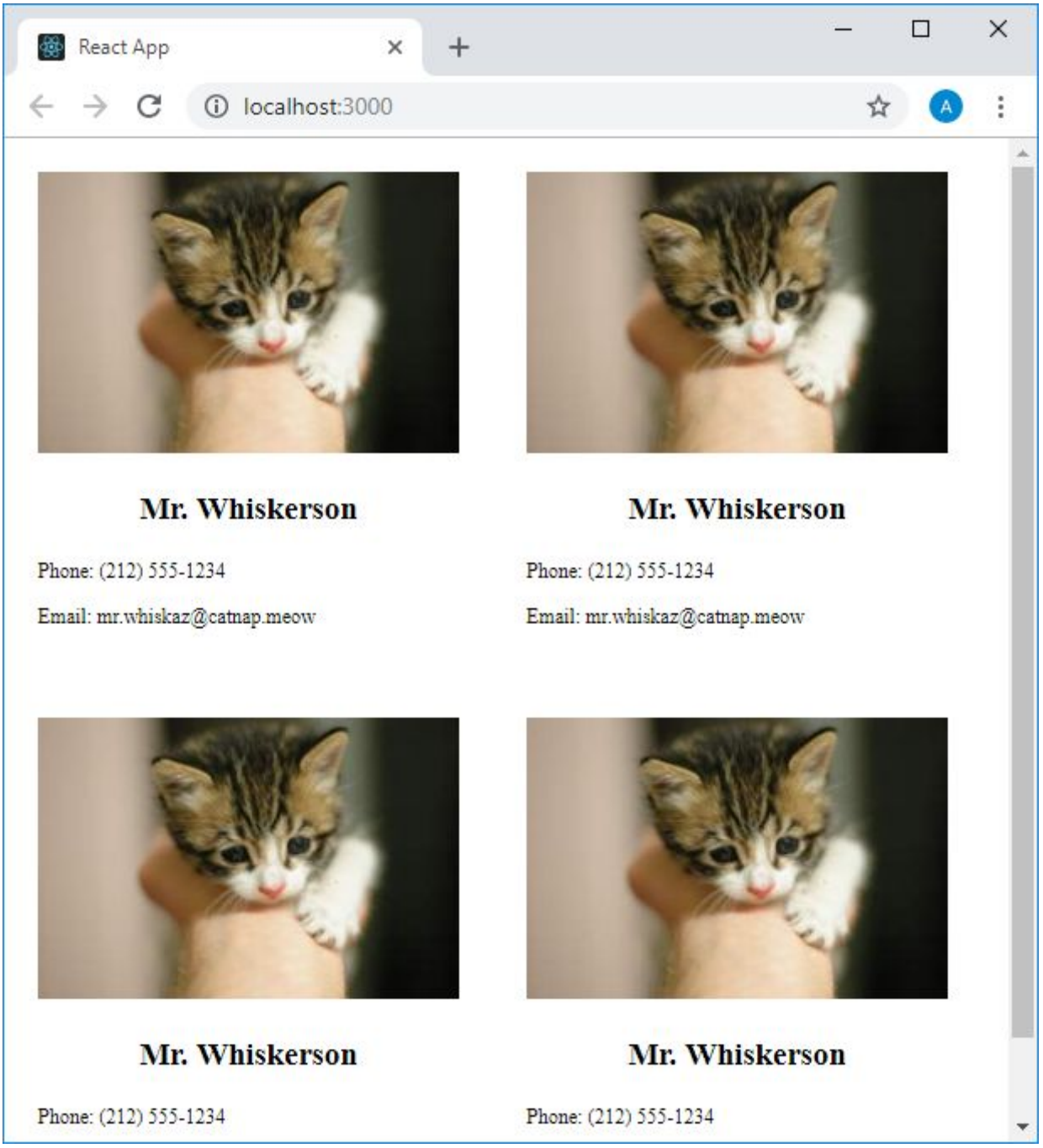
Каждая из карточек содержит четыре фрагмента информации, которые, от карточки к карточке, могут меняться. Это — изображение кошки и её имя, а также телефон и адрес электронной почты. Пусть имя кошки будет содержаться в свойстве `name`, адрес изображения — в свойстве `imgURL`, телефон — в свойстве `phone`, а адрес электронной почты — в свойстве `email`.

Зададим эти свойства экземплярам компонентов `ContactCard` и, по мере переноса данных из кода, который уже имеется в `App`, будем удалять соответствующие его фрагменты. В результате код компонента `App` будет выглядеть так:

```
import React from "react"
import ContactCard from "./ContactCard"

function App() {
    return (
        <div className="contacts">
            <ContactCard
                name="Mr. Whiskerson"
                imgUrl="http://placekitten.com/300/200"
                phone="(212) 555-1234"
                email="mr.whiskaz@catnap.meow"
            />

            <ContactCard
                name="Fluffykins"
                imgUrl="http://placekitten.com/400/200"
                phone="(212) 555-2345"
                email="fluff@me.com"
            />

            <ContactCard
                name="Destroyer"
                imgUrl="http://placekitten.com/400/300"
                phone="(212) 555-3456"
                email="ofworlds@yahoo.com"
            />

            <ContactCard
                name="Felix"
                imgUrl="http://placekitten.com/200/100"
```

```
                phone="(212) 555-4567"
                email="thecat@hotmail.com"
            />


        </div>
    )
}


export default App
```

Правда, одной только передачи свойств компоненту недостаточно для того, чтобы они были бы в нём использованы. Страница, которая будет сформирована вышеприведённым компонентом `App`, будет содержать четыре одинаковых карточки, данные которых заданы в коде компонента `ContactCard`, который пока не знает о том, что ему делать с переданными ему свойствами.



Данные карточек жёстко заданы в коде, компонент не умеет работать с переданными ему свойствами

Поэтому сейчас пришло время поговорить о том, как компонент `ContactCard` может работать со свойствами, передаваемыми ему при создании его экземпляров.

Приступим к решению этой задачи, указав, при объявлении функции `ContactCard`, что она принимает параметр `props`. При этом код компонента будет выглядеть так:

```
import React from "react"


function ContactCard(props) {

    return (

        <div className="contact-card">
```

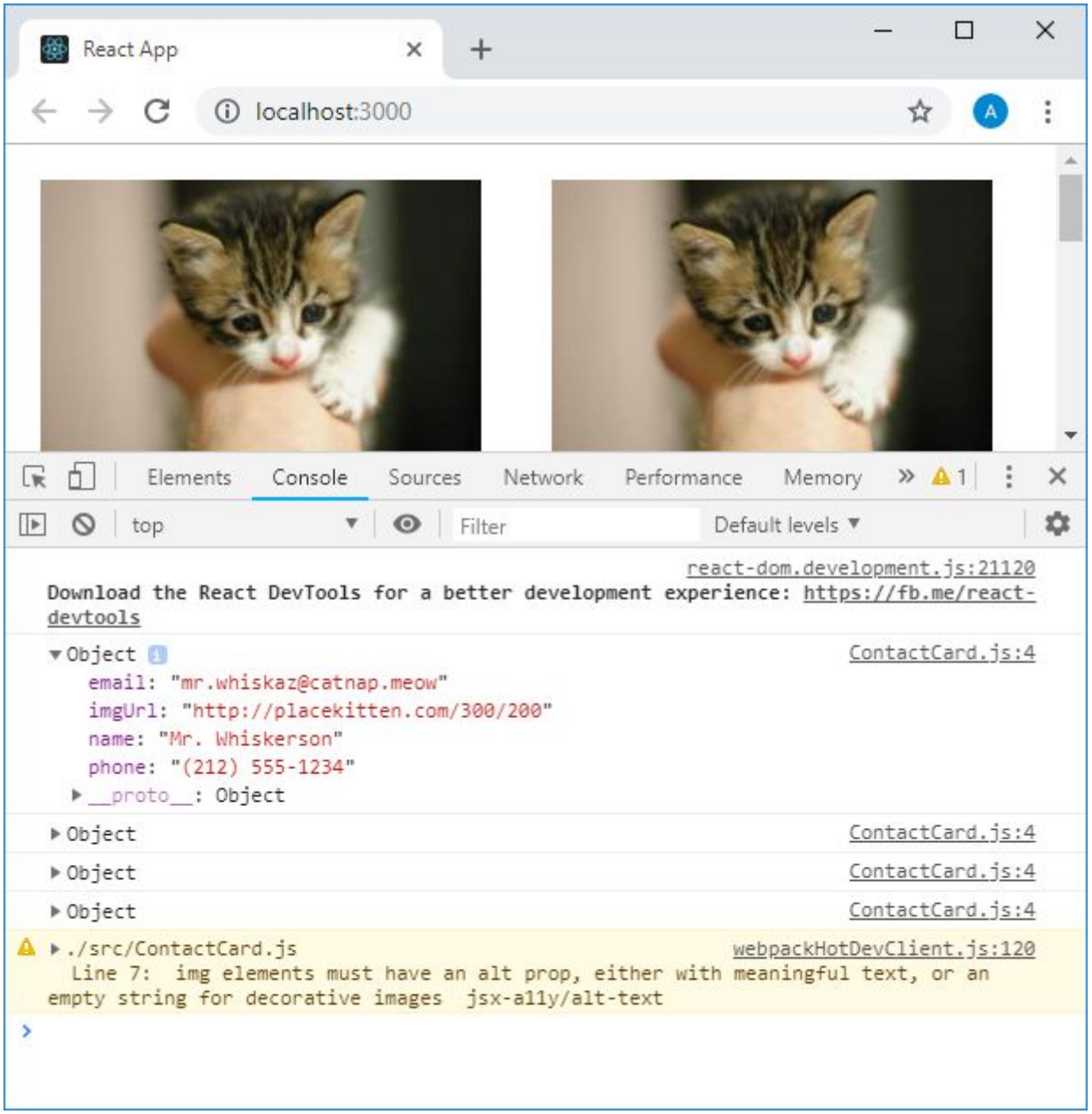
```
            <img src="http://placekitten.com/300/200"/>
            <h3>Mr. Whiskerson</h3>
            <p>Phone: (212) 555-1234</p>
            <p>Email: mr.whiskaz@catnap.meow</p>
        </div>
    )
}

export default ContactCard
```

На самом деле, этот параметр можно назвать как угодно, но в React принято называть его именно `props`, и те свойства, о которых мы тут говорим, часто называют просто «props».

Параметр `props` — это объект. Свойствами этого объекта являются свойства, переданные компоненту при создании его экземпляра. То есть, например, в нашем объекте `props` будет свойство `props.name`, содержащее имя кошки, переданное компоненту при создании его экземпляра. Кроме того, у него будут свойства `props.imgUrl`, `props.phone`, `props.email`. Для того чтобы в этом убедиться, добавим в начало функции `ContactCard` команду `console.log(props)`.

```
import React from "react"

function ContactCard(props) {
    console.log(props)
    return (
        <div className="contact-card">
            <img src="http://placekitten.com/300/200"/>
            <h3>Mr. Whiskerson</h3>
            <p>Phone: (212) 555-1234</p>
            <p>Email: mr.whiskaz@catnap.meow</p>
        </div>
    )
}

export default ContactCard
```

Это позволит вывести объект `props`, получаемый компонентом, в консоль.



Объект props в консоли

Тут можно видеть вывод четырёх объектов из `ContactCard.js`. Их именно столько из-за того, что мы создаём четыре экземпляра компонента `ContactCard`.

Всё это даёт нам возможность использовать в коде компонента, вместо жёстко заданных значений, то, что передано ему при создании его экземпляра, доступное в виде свойств объекта `props`.

Что если мы попытаемся воспользоваться свойством `props.imgUrl` так:

```
<img src=props.imgUrl/>
```

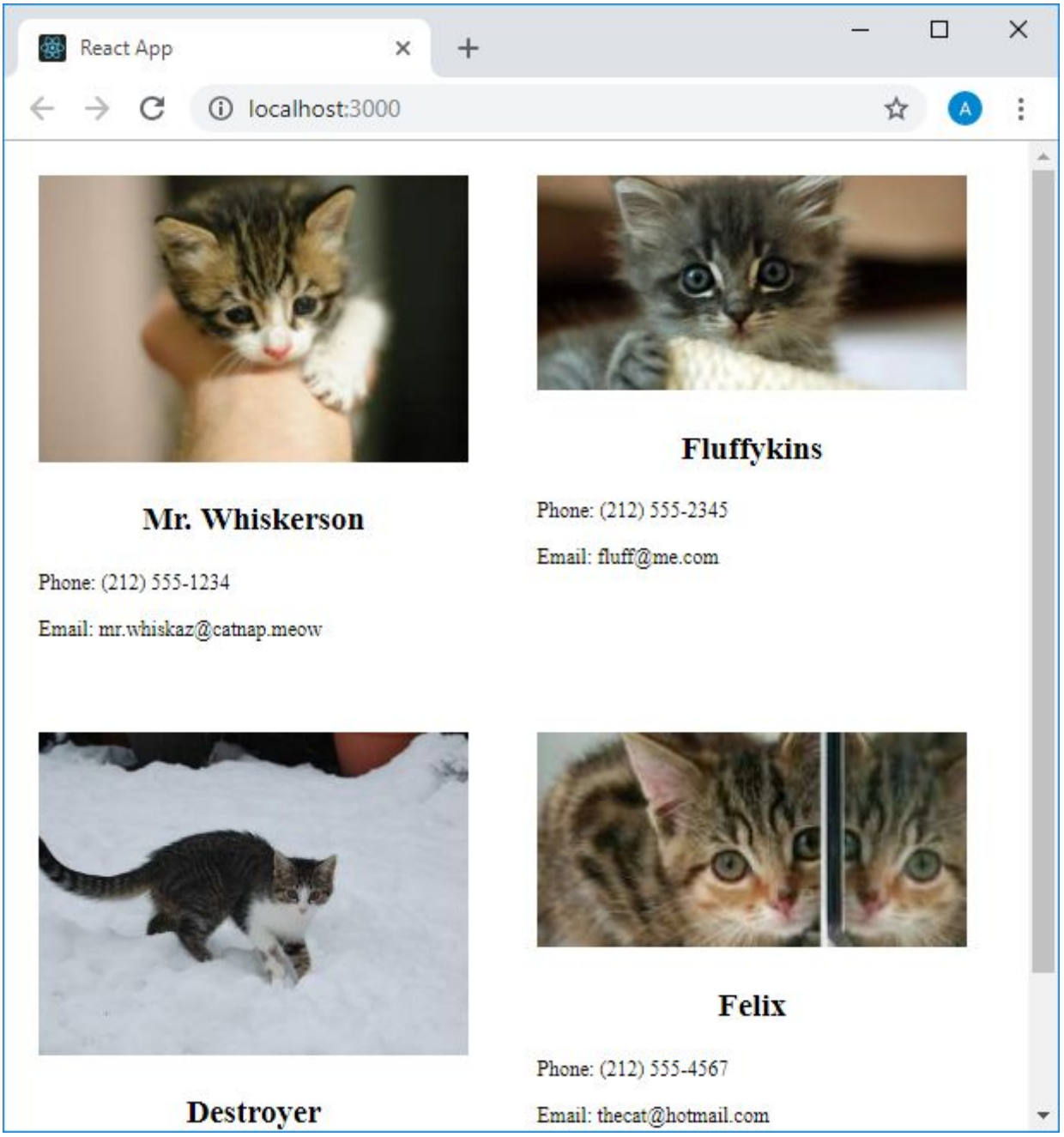
На первый взгляд такая конструкция может сработать, но вспомним о том, что тут нам нужно использовать сущность из JavaScript в JSX-коде. О том, как это делается, мы говорили на одном из предыдущих занятий. А именно, в нашем случае свойство объекта нужно заключить в фигурные скобки:

```
<img src={props.imgUrl}/>
```

Переработаем по такому же принципу другие элементы, возвращаемые компонентом, после чего его код примет следующий вид:

```
import React from "react"


function ContactCard(props) {

    return (

        <div className="contact-card">

            <img src={props.imgUrl}/>

            <h3>{props.name}</h3>

            <p>Phone: {props.phone}</p>

            <p>Email: {props.email}</p>

        </div>

    )

}


export default ContactCard
```

Обратите внимание на то, что в полях для вывода телефона и адреса электронной почты мы оставили тексты `Phone:` и `Email:` с пробелами, следующими за ними, так как эти тексты используются во всех компонентах. Если теперь взглянуть на страницу приложения, то можно заметить, что она содержит четыре разных карточки.



Страница, сформированная с использованием универсального компонента

Наш компонент принимает всего четыре свойства. Что если некоему компоненту нужно будет, например, передать 50 свойств? Пожалуй, передавать каждое такое свойство отдельной строкой, как это сделано в компоненте `App`, будет неудобно. В таких случаях можно воспользоваться другим способом передачи свойств компонентам. Он заключается в том, что, при создании экземпляра компонента, ему передаётся не список свойств, а объект со свойствами. Вот как это может выглядеть на примере первого компонента:

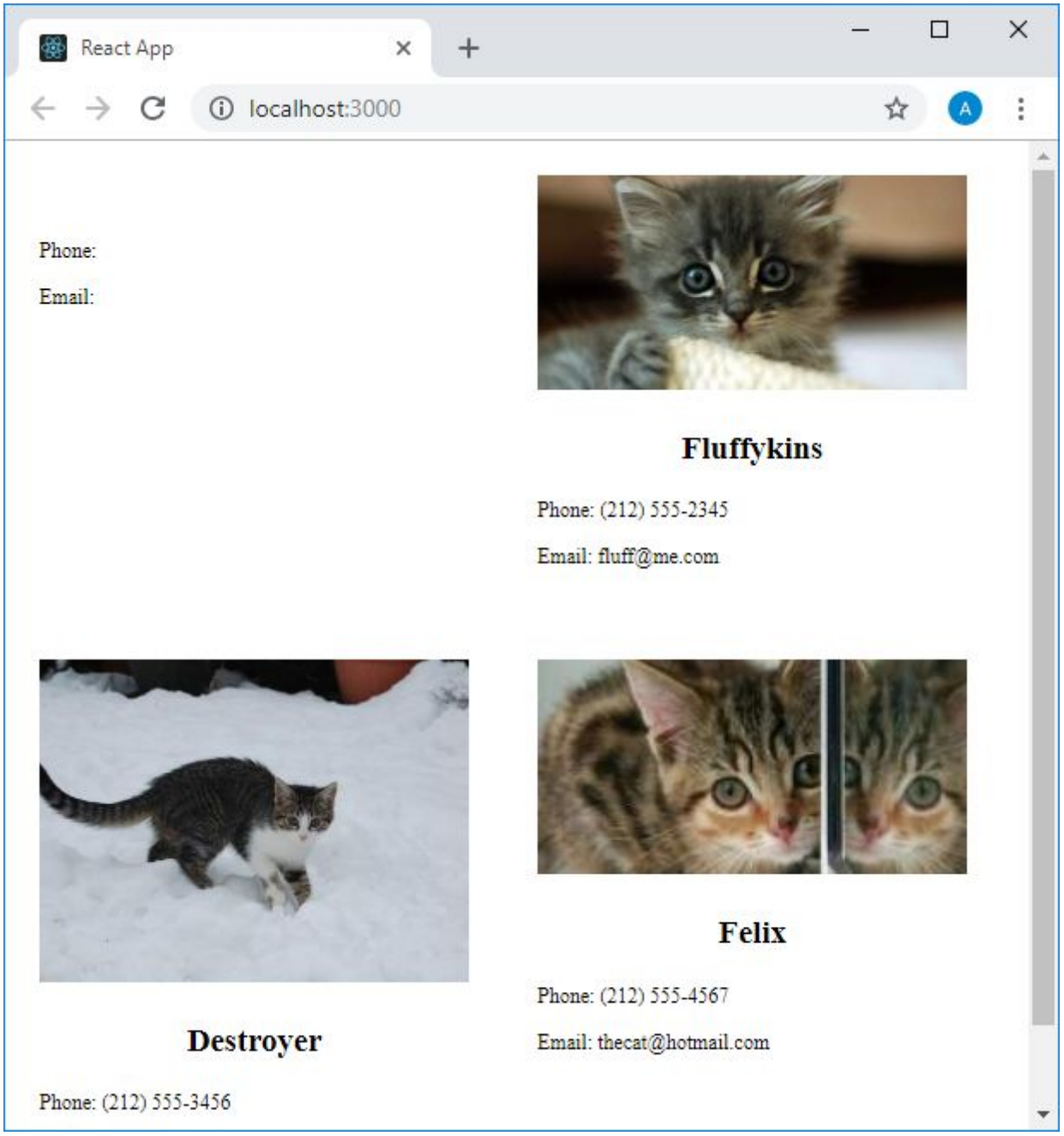
```
import React from "react"

import ContactCard from "./ContactCard"


function App() {

    return (
```

```
<div className="contacts">
    <ContactCard
        contact={{
          name: "Mr. Whiskerson",
          imgUrl: "http://placekitten.com/300/200",
          phone: "(212) 555-1234",
          email: "mr.whiskaz@catnap.meow"
        }}
    />

    <ContactCard
        name="Fluffykins"
        imgUrl="http://placekitten.com/400/200"
        phone="(212) 555-2345"
        email="fluff@me.com"
    />

    <ContactCard
        name="Destroyer"
        imgUrl="http://placekitten.com/400/300"
        phone="(212) 555-3456"
        email="ofworlds@yahoo.com"
    />

    <ContactCard
        name="Felix"
        imgUrl="http://placekitten.com/200/100"
        phone="(212) 555-4567"
        email="thecat@hotmail.com"
    />
```

```
        </div>
    )
}


export default App
```

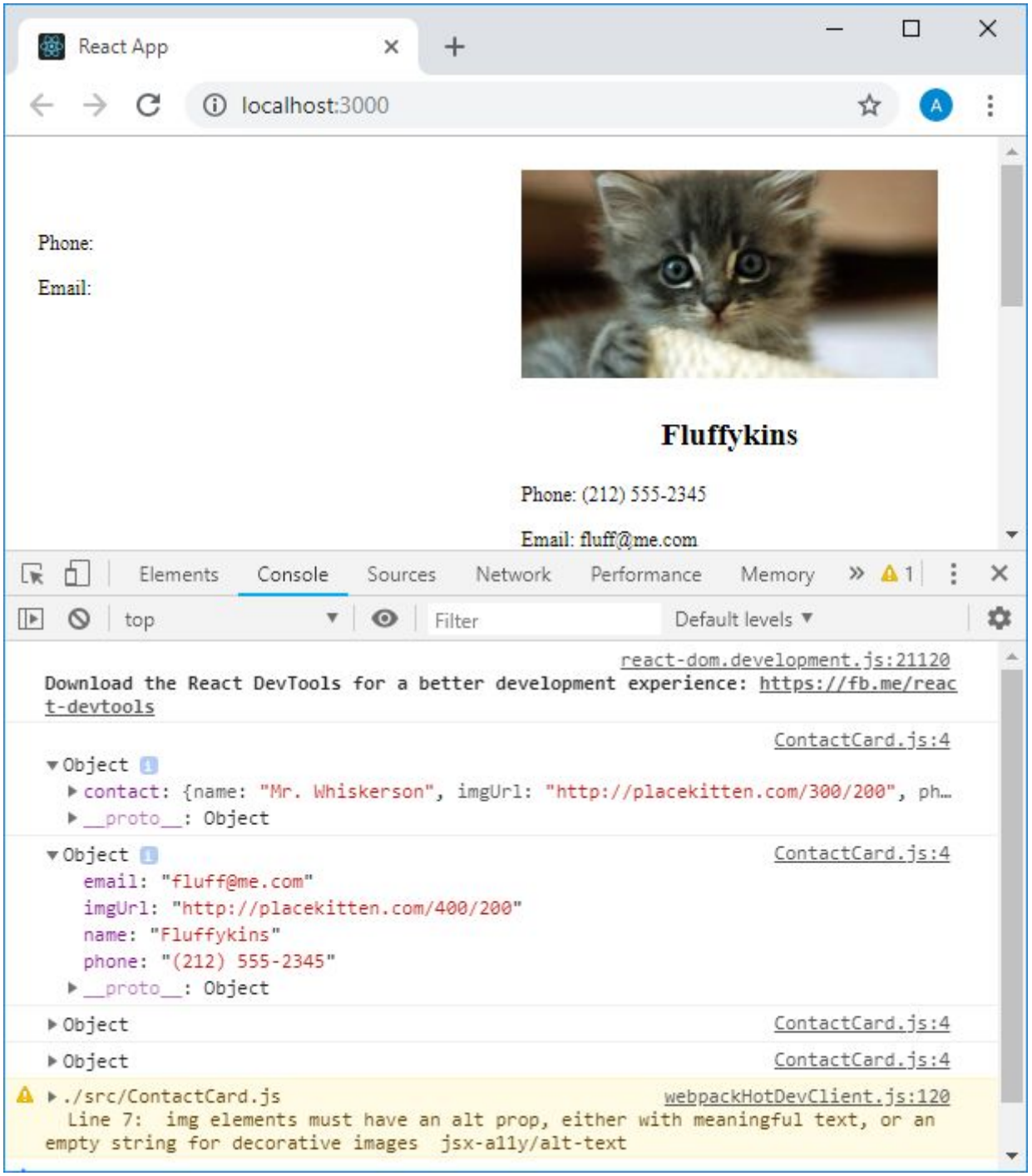
Нельзя сказать, что этот подход значительно сократил количество кода, используемого для описания экземпляра компонента. Дело в том, что свойства, передаваемые компоненту, всё так же жёстко заданы в коде, хотя мы и передаём компоненту лишь один объект. Преимущества этого подхода можно будет ощутить в ситуациях, когда данные для компонента получают из неких внешних источников. Например — из JSON-файла.

В ходе модификации кода компонента `App`, используемого для создания первого экземпляра компонента `ContactCard`, правильная работа приложения была нарушена. Вот как теперь будет выглядеть его страница.



Нарушение правильной работы приложения

Как это можно это исправить? Для того чтобы в этом разобраться, полезно будет проанализировать происходящее с помощью команды `console.log(props)`.



Анализ объекта props

Как видно, объект `props` первого компонента отличается от такого же объекта второго и следующих компонентов.

В компоненте `ContactCard` мы пользуемся объектом `props` исходя из предположения о том, что у него есть свойства `name`, `imgUrl` и прочие подобные. Здесь же первый компонент получает лишь одно свойство — `contact`. Это приводит к тому, что у объекта `props` оказывается лишь одно свойство — `contact`, являющееся объектом, а в коде компонента работа с подобной структурой не предусмотрена.

Перевести наш компонент на модель использования лишь одного свойства-объекта `contact`, содержащего другие свойства, довольно просто. Для этого, например, для доступа к свойству `name`, достаточно воспользоваться конструкцией вида `props.contact.name` в коде компонента. Аналогичные конструкции позволяют правильно работать с другими нужными нам свойствами.

Переработаем код компонента с учётом передачи ему единственного свойства-объекта `contact`, содержащего другие свойства:

```
import React from "react"


function ContactCard(props) {
    console.log(props)
    return (
        <div className="contact-card">
            <img src={props.contact.imgUrl}/>
            <h3>{props.contact.name}</h3>
            <p>Phone: {props.contact.phone}</p>
            <p>Email: {props.contact.email}</p>
        </div>
    )
}


export default ContactCard
```

Первый компонент теперь должен будет выводиться нормально, но мы этого, на данном этапе работы над проектом, не увидим, так как система сообщит нам о множестве ошибок, связанных с тем, что несколько экземпляров компонента `ContactCard`, создаваемые в компоненте `App`, не получают свойство-объект `contact`. При выполнении кода это свойство будет иметь значение `undefined`. В результате производится попытка обратиться к некоему свойству значения `undefined`, что и приводит к возникновению ошибки. Исправим это, переработав код компонента `App`, ответственный за формирование компонентов `ContactCard`:

```
import React from "react"
import ContactCard from "./ContactCard"


function App() {
    return (
        <div className="contacts">
            <ContactCard
                contact={{
                  name: "Mr. Whiskerson",
```

```
      imgUrl: "http://placekitten.com/300/200",
      phone: "(212) 555-1234",
      email: "mr.whiskaz@catnap.meow"
    }}
/>

<ContactCard
    contact={{
      name: "Fluffykins",
      imgUrl: "http://placekitten.com/400/200",
      phone: "(212) 555-2345",
      email: "fluff@me.com"
    }}
/>

<ContactCard
    contact={{
      name: "Destroyer",
      imgUrl: "http://placekitten.com/400/300",
      phone: "(212) 555-3456",
      email: "ofworlds@yahoo.com"
    }}
/>

<ContactCard
    contact={{
      name: "Felix",
      imgUrl: "http://placekitten.com/200/100",
      phone: "(212) 555-4567",
      email: "thecat@hotmail.com"
```

```
                }}

            />


        </div>

    )

}


export default App
```

Теперь страница приложения будет выглядеть так же, как раньше.

Как обычно, рекомендуется самостоятельно поэкспериментировать с изученными сегодня концепциями для того, чтобы лучше их усвоить. Например — можете поработать с кодом, добавить новые свойства, передаваемые компоненту, и попытаться использовать их в компоненте.

Учебный курс по React, часть 10: практикум по работе со свойствами компонентов и стилизации

Занятие 20. Практикум. Свойства компонентов, стилизация

Оригинал

Задание

1. Создайте новый проект React-приложения.
2. Выведите на странице приложения компонент `App`, код которого должен находиться в отдельном файле.
3. Компонент App должен выводить 5 компонентов `Joke`, содержащих анекдоты. Выведите эти компоненты так, как вам хочется.
4. Каждый компонент `Joke` должен принимать и обрабатывать свойство `question`, для основной части анекдота, и свойство `punchLine` — для его ключевой фразы.

Дополнительное задание
Некоторые анекдоты целиком состоят из ключевой фразы. Например: «It’s hard to explain puns to kleptomaniacs because they always take things literally». Подумайте над тем, как компонент `Joke` может вывести лишь переданное ему свойство `punchLine`, в том случае, если свойство `question` не задано. Поэкспериментируйте со стилизацией компонентов.

Решение

Основное задание

Файл `index.js` будет выглядеть вполне привычно:

```
import React from "react"

import ReactDOM from "react-dom"


import App from "./App"
```

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

Вот код файла `App.js`:

```
import React from "react"
import Joke from "./Joke"

function App() {
    return (
        <div>
            <Joke
                question="What's the best thing about Switzerland?"
                punchLine="I don't know, but the flag is a big plus!"
            />

            <Joke
                question="Did you hear about the mathematician who's afraid of
negative numbers?"
                punchLine="He'll stop at nothing to avoid them!"
            />

            <Joke
                question="Hear about the new restaurant called Karma?"
                punchLine="There’s no menu: You get what you deserve."
            />

            <Joke
                question="Did you hear about the actor who fell through the
floorboards?"
                punchLine="He was just going through a stage."
            />
```

```
            <Joke
                question="Did you hear about the claustrophobic astronaut?"
                punchLine="He just needed a little space."
            />

        </div>
    )
}

export default App
```

Обратите внимание на то, что, так как файл компонента `Joke` расположен в той же папке, что и файл компонента `App`, мы импортируем его командой `import Joke from "./Joke"`. Из `App` мы возвращаем несколько элементов, поэтому весь вывод нужно обернуть в некий тег, например — в тег `<div>`. Экземплярам компонента мы передаём свойства `question` и `punchLine`.

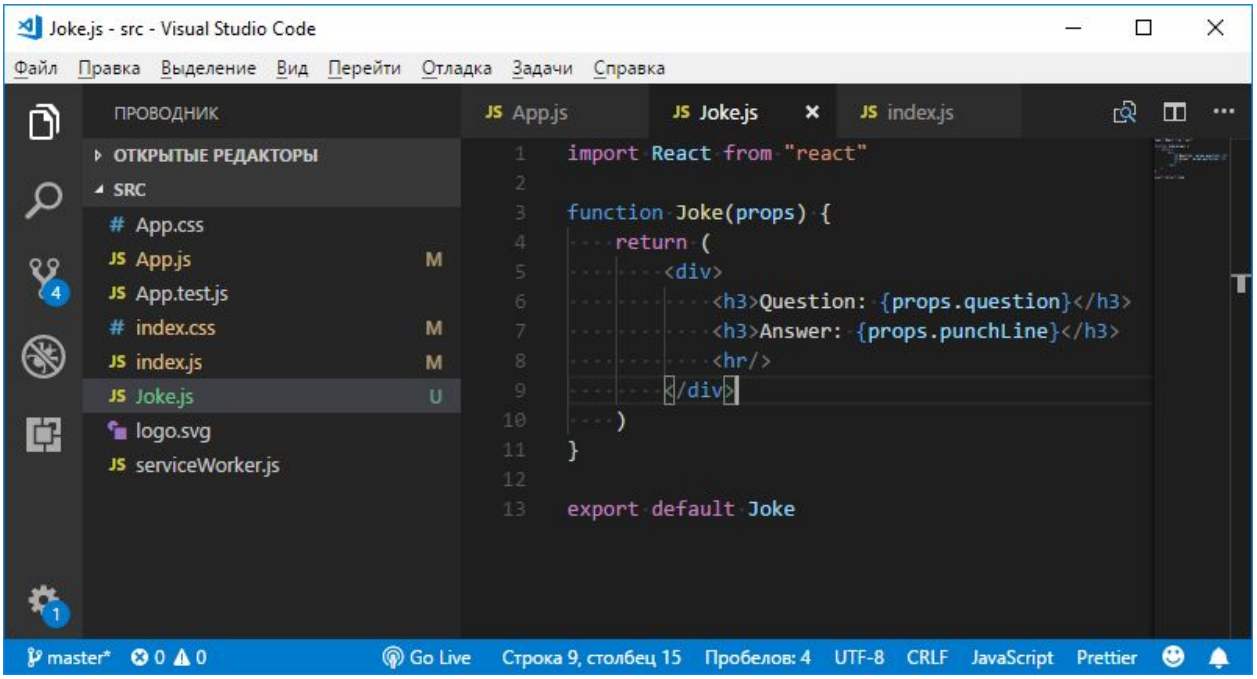
Вот код файла `Joke.js`:

```
import React from "react"

function Joke(props) {
    return (
        <div>
            <h3>Question: {props.question}</h3>
            <h3>Answer: {props.punchLine}</h3>
            <hr/>
        </div>
    )
}

export default Joke
```

Здесь, при объявлении функции `Joke`, мы указываем параметр `props`. Напомним, что именно такое имя используется по сложившейся традиции. На самом деле оно может быть любым, но лучше называть его именно `props`.

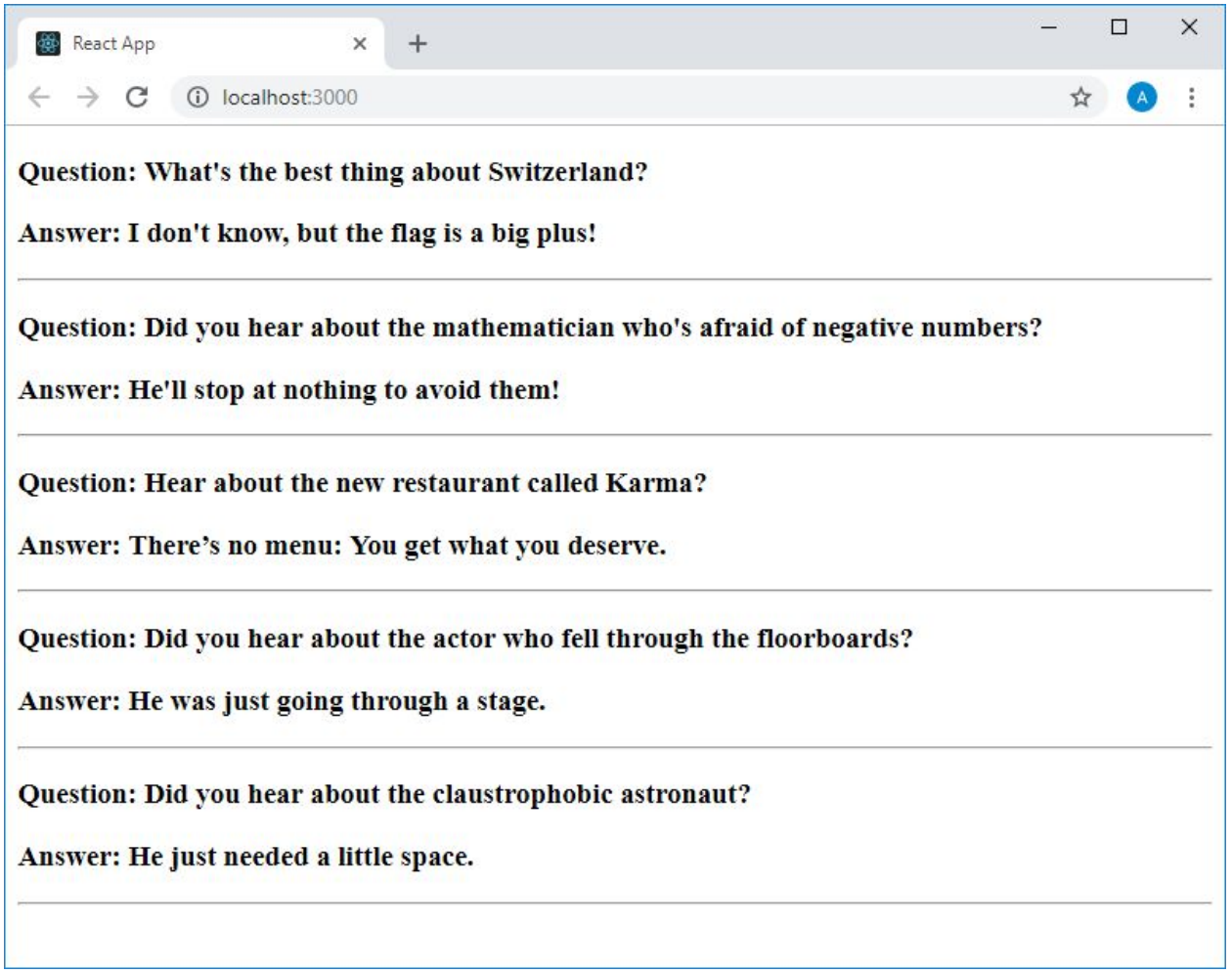
Из компонента мы возвращаем несколько элементов — поэтому они заключены в тег `<div>`. С помощью конструкций `props.question` и `props.punchLine` мы обращаемся к свойствам, переданным экземпляру компонента при его создании. Эти свойства становятся свойствами объекта `props`. Они заключены в фигурные скобки из-за того, что JavaScript-код, используемый в JSX-разметке, нужно оформлять фигурными скобками. Иначе система примет имена переменных за обычный текст. После пары элементов `<h3>`, в одном из которых выводится основной текст анекдота, а в другой — его ключевая фраза, находится элемент `<hr/>`, описывающий горизонтальную линию. Такие линии будут выводиться после каждого анекдота, разделяя их.

Вот как выглядит проект приложения в VSCode.



Приложение в VSCode

Вот страница приложения.



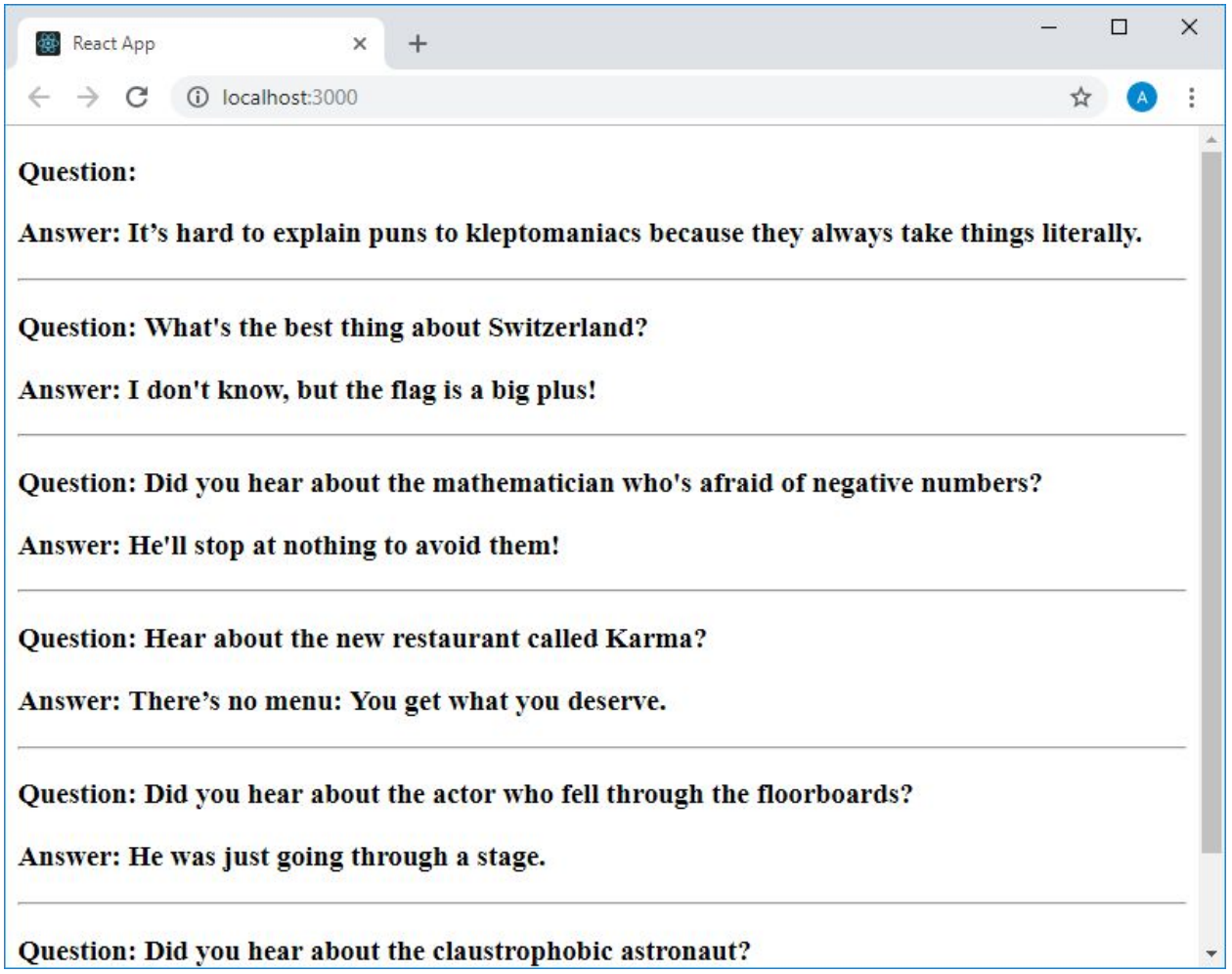
Страница приложения в браузере

Дополнительное задание

Напомним, что основная цель дополнительного задания заключается в том, чтобы организовать правильный вывод анекдотов, которые целиком состоят из ключевой фразы. Это выражается в том, что, при создании экземпляра компонента `Joke`, ему передают лишь свойство `punchLine`, а свойство `question` не передают. Создание экземпляра подобного компонента выглядит так:

```
<Joke

    punchLine="It's hard to explain puns to kleptomaniacs because they always take things literally."

/>
```

Если поместить этот код в верхнюю часть кода, возвращаемого компонентом `App`, то страница приложения примет следующий вид.



Неправильно сформированная страница приложения

Очевидно, проблема тут заключается в том, что, хотя компоненту не передано свойства `question`, он выводит текст, предваряющий основную часть каждого анекдота, после которого ничего уже не выводится.

Забегая вперёд отметим, что в будущих частях курса мы поговорим об условном рендеринге. С помощью этого подхода к рендерингу можно эффективно решать задачи, подобные нашей. Пока же мы попытаемся воспользоваться средствами стилизации страниц. А именно, сделаем так, чтобы, если компоненту не передаётся свойство `question`, соответствующий фрагмент возвращаемой им JSX-разметки не отображался бы на странице. Вот полный код компонента `Joke`, в котором реализован один из возможных подходов решения нашей проблемы средствами CSS:

```
import React from "react"


function Joke(props) {

    return (

        <div>

            <h3 style={{display: props.question ? "block" : "none"}}>Question:
{props.question}</h3>

            <h3>Answer: {props.punchLine}</h3>
```

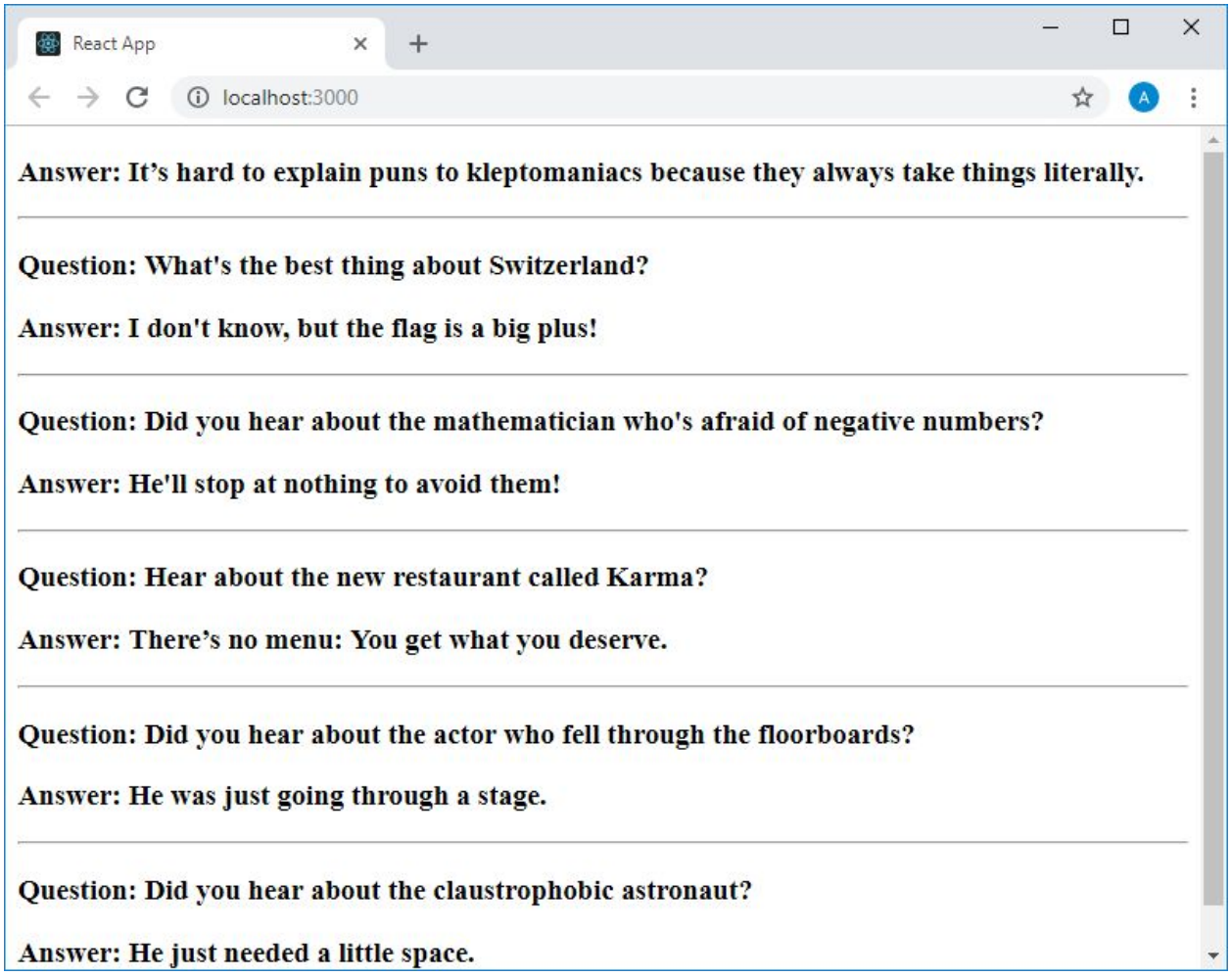
```
            <hr/>
        </div>
    )
}


export default Joke
```

Первому элементу `<h3>` мы назначаем стиль, который определяется в процессе создания экземпляра компонента на основе наличия в объекте свойства `props.question`. Если это свойство в объекте есть, элемент принимает стиль `display: block` и выводится на страницу, если нет — `display: none` и на страницу не выводится. К тому же эффекту приведёт и использование такой конструкции:

```
<h3 style={{display: !props.question && "none"}}>Question:
{props.question}</h3>
```

Здесь стиль `display: none` назначается элементу в том случае, если у объекта `props` нет свойства `question`, в противном случае свойству `display` не назначается ничего.

Теперь страница приложения в браузере будет выглядеть так, как показано ниже.



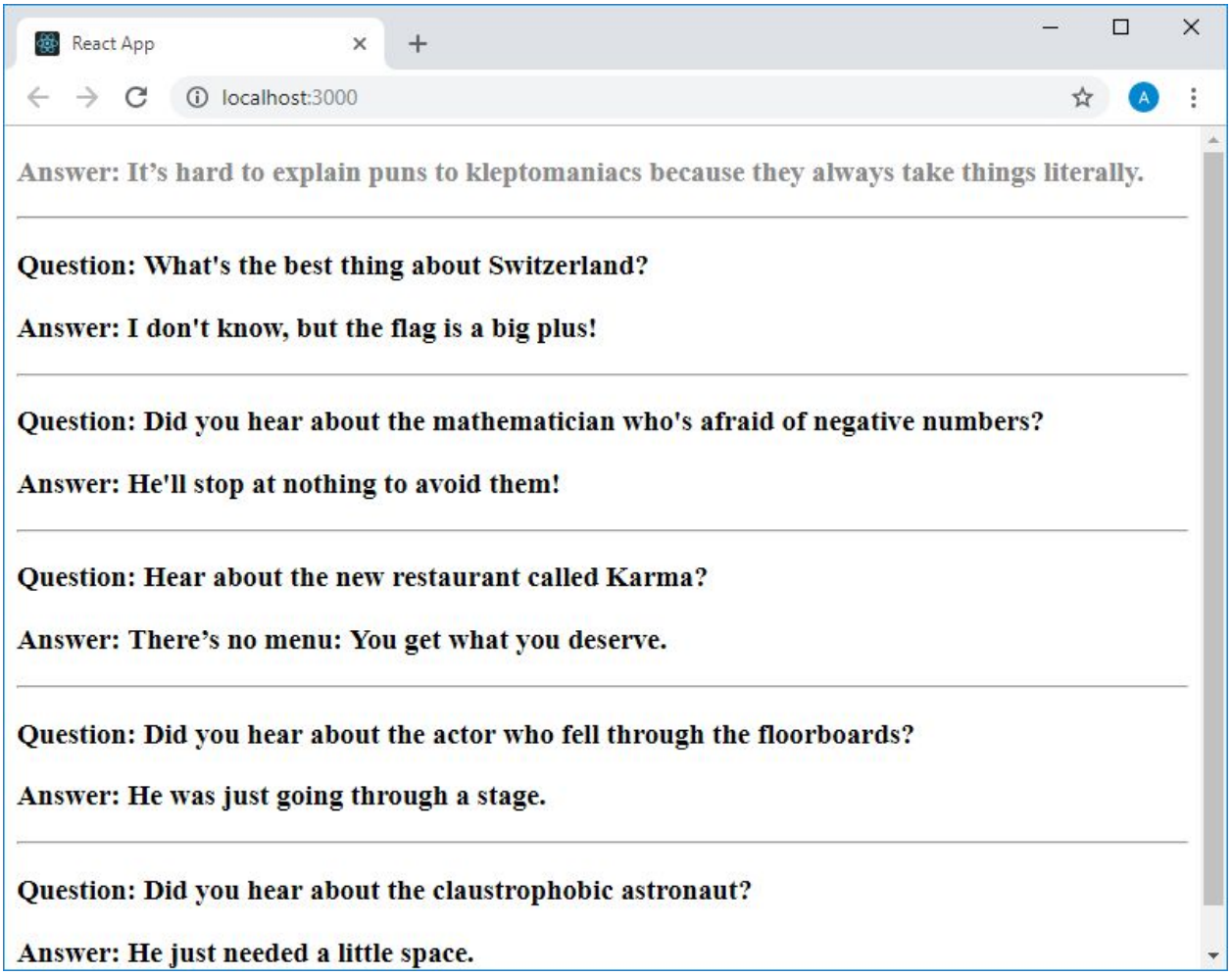
Правильная обработка компонентом ситуации, в которой ему не передают свойство question

Можно заметить, что все элементы, формируемые компонентом `Joke`, выглядят одинаково. Подумаем над тем, как выделить те из них, которым передаётся лишь свойство `punchLine`. Решим эту задачу,

используя встроенные стили, и тот подход, который мы рассмотрели выше. Вот обновлённый код компонента `Joke`:

```
import React from "react"


function Joke(props) {

    return (

        <div>

            <h3 style={{display: !props.question && "none"}}>Question:
{props.question}</h3>

            <h3 style={{color: !props.question && "#888888"}}>Answer:
{props.punchLine}</h3>

            <hr/>

        </div>

    )

}


export default Joke
```

А вот как выглядит то, что теперь выводится на страницу приложения.



Стилизация элемента, который отличается от других

Теперь, после того, как мы поработали над компонентом `Joke`, он стал более универсальным и лучше подходящим для повторного использования.

Учебный курс по React, часть 11: динамическое формирование разметки и метод массивов map

Занятие 21. Динамическое формирование разметки и метод массивов map

Оригинал

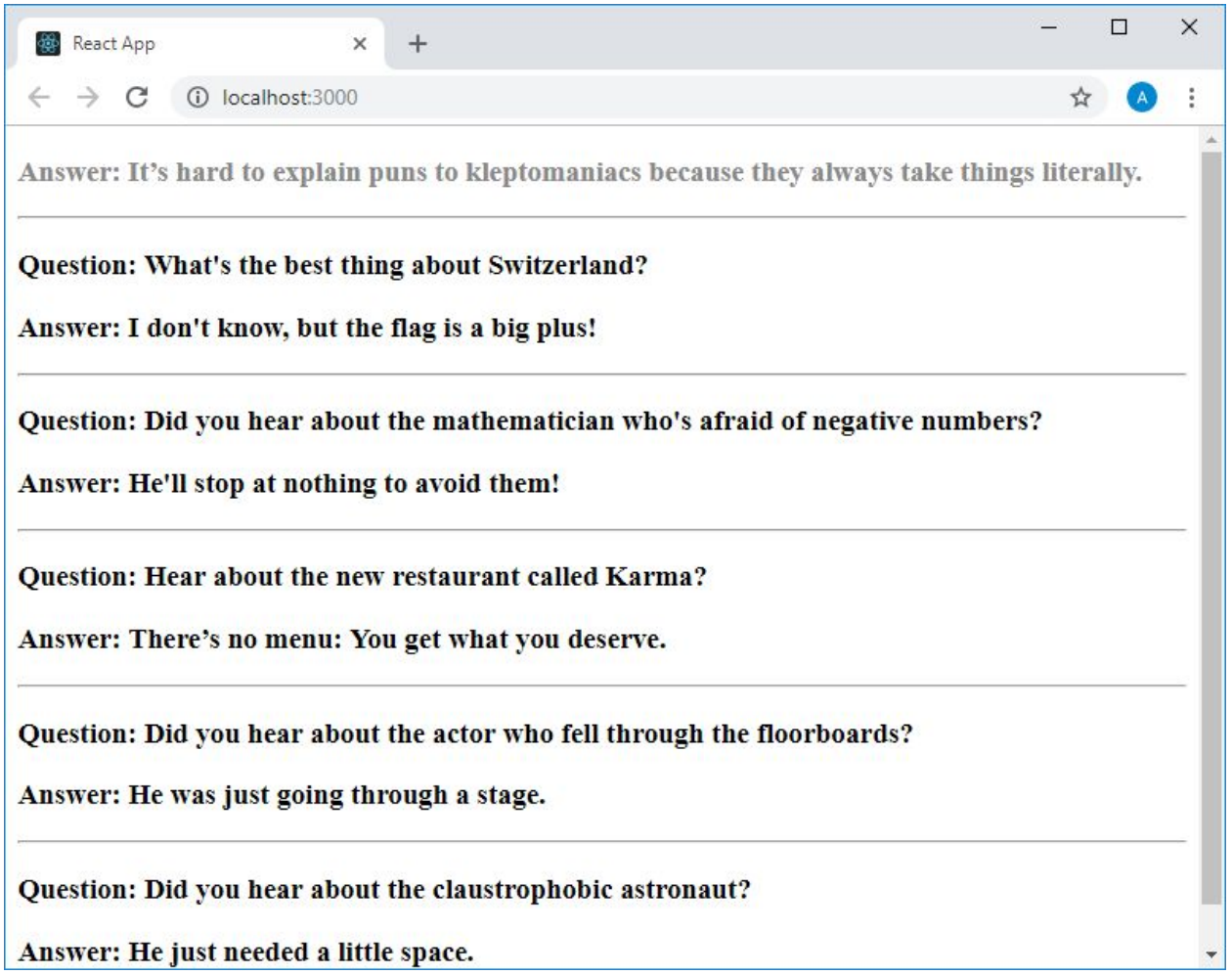
Продолжим работу с того момента, на котором мы остановились, выполняя предыдущее практическое задание. Напомним, что тогда код файла `App.js` выглядел следующим образом:

```
import React from "react"


import Joke from "./Joke"


function App() {

    return (

        <div>
```

```
            <Joke punchLine="It’s hard to explain puns to kleptomaniacs because
they always take things literally." />


            <Joke

                question="What's the best thing about Switzerland?"

                punchLine="I don't know, but the flag is a big plus!"

            />


            <Joke

                question="Did you hear about the mathematician who's afraid of
negative numbers?"

                punchLine="He'll stop at nothing to avoid them!"

            />


            <Joke

                question="Hear about the new restaurant called Karma?"

                punchLine="There’s no menu: You get what you deserve."

            />


            <Joke

                question="Did you hear about the actor who fell through the
floorboards?"

                punchLine="He was just going through a stage."

            />


            <Joke

                question="Did you hear about the claustrophobic astronaut?"

                punchLine="He just needed a little space."

            />


        </div>

    )
```

```
}


export default App
```

Компонент `App` выводит набор компонентов `Joke`. Вот как на данном этапе работы выглядит страница приложения.



Страница приложения

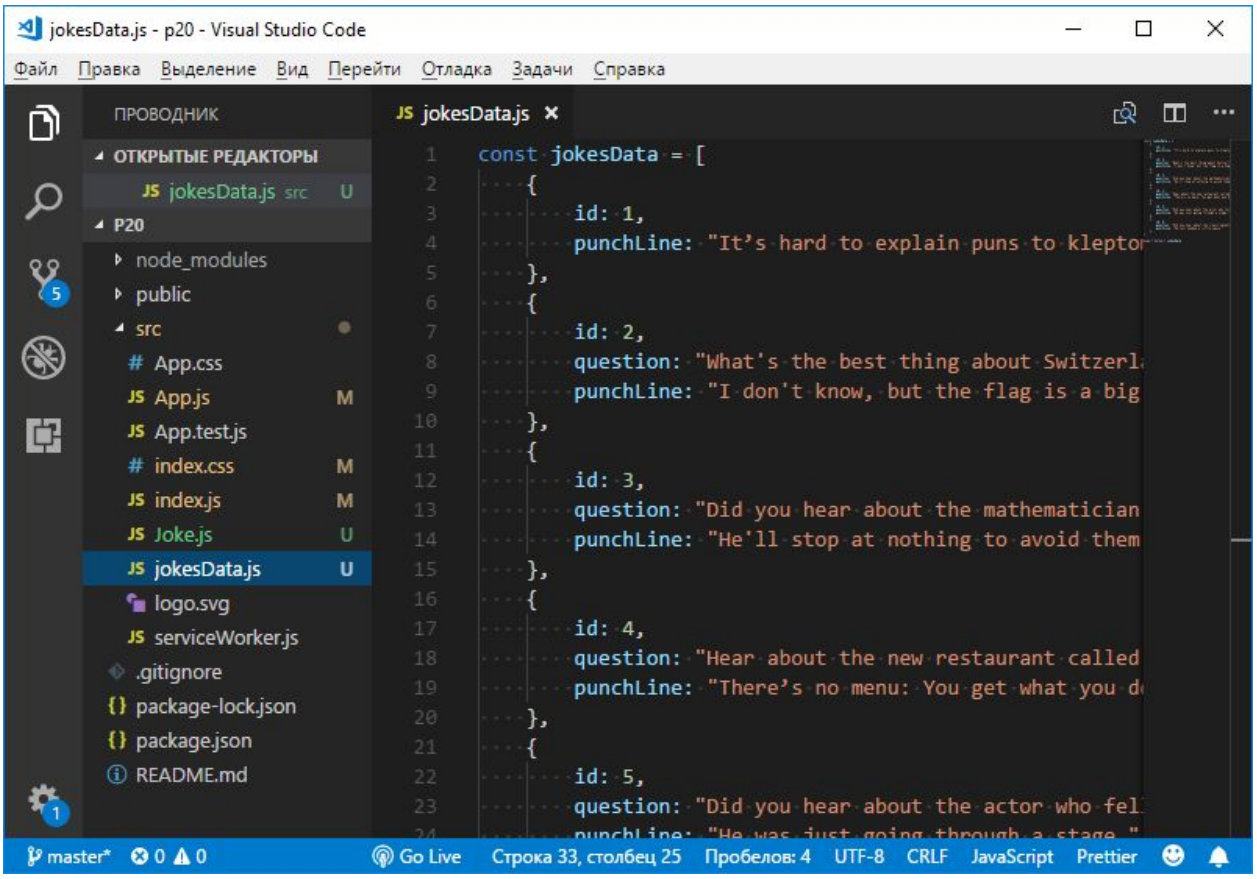
Некоторым из этих компонентов передаются свойства `question` и `punchLine`, а некоторым — только `punchLine`. Сейчас значения этих свойств заданы в коде создания экземпляров компонента `Joke` в виде обычного текста. В реальности же основной объём данных, которые выводят на страницы React-приложений, поступает в приложение в результате выполнения HTTP-запросов к неким API. Эти API поддерживаются средствами серверов, которые берут информацию из баз данных, оформляют её в виде JSON-кода и отправляют этот код клиентским частям приложений. Мы пока ещё не дошли до такого уровня, чтобы выполнять запросы к API, поэтому сейчас мы, в роли источника данных, воспользуемся файлом с данными, которые могли бы быть получены в результате разбора JSON-ответа сервера. А именно, это будет файл `jokesData.js` со следующим содержимым:

```
const jokesData = [

    {

        id: 1,
```

```
        punchLine: "It’s hard to explain puns to kleptomaniacs because they always take things literally."
    },
    {
        id: 2,
        question: "What's the best thing about Switzerland?",
        punchLine: "I don't know, but the flag is a big plus!"
    },
    {
        id: 3,
        question: "Did you hear about the mathematician who's afraid of negative numbers?",
        punchLine: "He'll stop at nothing to avoid them!"
    },
    {
        id: 4,
        question: "Hear about the new restaurant called Karma?",
        punchLine: "There’s no menu: You get what you deserve."
    },
    {
        id: 5,
        question: "Did you hear about the actor who fell through the floorboards?",
        punchLine: "He was just going through a stage."
    },
    {
        id: 6,
        question: "Did you hear about the claustrophobic astronaut?",
        punchLine: "He just needed a little space."
    }
]
```

```
export default jokesData
```

Этот файл будет расположен в директории `src` нашего проекта.



Новый файл в папке src

Фактически, в нём содержится массив объектов. Похожий массив можно получить, разобрав JSON-данные, полученные от некоего API. Мы экспортируем из этого файла массив `jokesData`. При необходимости мы можем импортировать этот файл в компонент, в котором он нужен, и представить себе, что работаем мы не с данными, взятыми из файла, а с тем, что вернуло нам некое API.

Теперь, когда у нас есть массив исходных данных, подумаем о том, как превратить эти данные в набор экземпляров React-компонентов.

Многие разработчики говорят о том, что благодаря освоению React они лучше изучили JavaScript. Причина подобного заключается в том, что действия, подобные тому, о котором мы будем сейчас говорить, в других фреймворках, вроде Angular и Vue, выполняются с помощью каких-то особых средств. А в React подобное делается с помощью обычного JavaScript.

В частности, мы планируем пользоваться некоторыми стандартными методами массивов, являющимися функциями высшего порядка. Эти методы могут, в качестве аргументов, принимать функции, описываемые программистами. Именно эти функции определяют то, что вызов того или иного стандартного метода сделает с элементами массива.

Предположим, у нас есть числовой массив:

```
const nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Мы можем обработать этот массив с помощью стандартного метода массивов `map()`, передав ему некую функцию, которая задаёт порядок преобразования элементов этого массива. В нашем случае этой функции будут передаваться, по одному, числа из этого массива. Функция может делать с ними всё, что угодно, после чего то, что она возвратит, попадёт в новый массив, в элемент, индекс которого

соответствует индексу обрабатываемого элемента. Если нам нужно сформировать новый массив, элементы которого представляют собой элементы исходного массива, умноженные на 2, то выглядеть это будет так:

```
const doubled = nums.map(function(num) {

    return num * 2

})
```

Проверим работу этого кода:

```
console.log(doubled) // [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Если раньше вы не встречались с методами массивов — такими, как `map()`, `filter()`, `reduce()` и другие — рекомендуется с ними разобраться.

Здесь мы, для автоматического формирования списка экземпляров компонента, будем пользоваться методом `map()`.

Вернёмся к нашему примеру. Импортируем в файл `App.js` файл `jokesData.js`. Делается это так:

```
import jokesData from "./jokesData"
```

После этого, в коде программы, мы сможем работать с массивом `jokesData`. А именно, мы собираемся воспользоваться методом `map()`. Вот как будет выглядеть «заготовка» этого метода.

```
jokesData.map(joke => {



})
```

Обратите внимание на то, что мы здесь передаём методу `map()` стрелочную функцию. В нашем случае это позволяет сделать код компактнее. Так как функция принимает всего один параметр (`joke`), мы, при её объявлении, можем обойтись без круглых скобок.

Из функции, передаваемой методу `map()`, мы хотим вернуть новый экземпляр компонента `Joke`, которому переданы свойства `question` и `punchLine` поступившего в неё элемента массива `jokesData`. Вот как это может выглядеть:

```
jokesData.map(joke => {

    return (

    <Joke question={joke.question} punchLine={joke.punchLine} />

    )

})
```

Этот код можно сократить, если учесть два факта. Во-первых, `return` возвращает лишь один элемент, поэтому можно поместить этот элемент сразу после `return`, избавившись от круглых скобок.
Во-вторых, стрелочная функция содержит лишь операцию возврата некоего значения, поэтому при объявлении такой функции можно обойтись и без ключевого слова `return` и без фигурных скобок.
Кроме того, вспомним о том, что в результате работы метода `map()` формируется новый массив. Этот массив нужно где-то сохранить. Все эти рассуждения приводят нас к следующему:

```
const jokeComponents = jokesData.map(joke => <Joke question={joke.question} punchLine={joke.punchLine} />)
```

В константе `jokeComponents` теперь будет содержаться массив, каждый элемент которого представляет собой описание экземпляра компонента `Joke` с переданными ему свойствами `question` и `punchLine`.

Что нам теперь делать с этим массивом компонентов? React позволяет очень удобно работать с такими массивами. А именно, речь идёт о том, что такой массив можно использовать в JSX-коде. Вот как теперь будет выглядеть код файла `App`:

```
import React from "react"


import Joke from "./Joke"

import jokesData from "./jokesData"


function App() {

    const jokeComponents = jokesData.map(joke => <Joke question={joke.question} punchLine={joke.punchLine} />)




    return (

        <div>

            {jokeComponents}

        </div>

    )

}


export default App
```

Страница приложения после этого будет выглядеть так же, как и прежде, однако в консоли браузера можно будет увидеть следующее предупреждение:

```
Warning: Each child in an array or iterator should have a unique "key" prop.

Check the render method of `App`. See https://fb.me/react-warning-keys for more information.

    in Joke (at App.js:7)

    in App (at src/index.js:6)
```

Смысл его сводится к тому, что у элементов массива должно быть уникальное свойство `key`. Мы не будем вдаваться в подробности, касающиеся того, почему React ожидает наличия уникального свойства `key` у повторяющихся компонентов. Нам достаточно учесть тот факт, что выполняя операции массового создания экземпляров компонентов, наподобие той, которую мы только что выполняли с помощью метода `map()`, экземплярам нужно передавать свойство `key`. При этом такое свойство можно передать и самому экземпляру компонента, и, например, тегу `<div>`, в который заключён код компонента. Это особой роли не играет.

Итак, свойству `key` нужно назначить некое уникальное значение. Как правило, в объектах данных, получаемых из API, имеются некие идентификаторы (свойства наподобие `id`). Главное для нас — их уникальность. Например, мы могли бы назначить свойству `key` значение `joke.question` — все тексты в этих свойствах в нашем приложении уникальны. Но мы поступим иначе. Вспомните о том, как выглядят объекты с данными из массива, который мы экспортировали из файла `jokesData.js`. Вот его фрагмент:

```
const jokesData = [

    {

        id: 1,

        punchLine: "It’s hard to explain puns to kleptomaniacs because they
always take things literally."

    },

    {

        id: 2,

        question: "What's the best thing about Switzerland?",

        punchLine: "I don't know, but the flag is a big plus!"

    },

...

]
```

У каждого объекта есть свойство `id`, уникальность которого мы поддерживаем самостоятельно. Именно значения таких свойств и можно использовать в роли значений для свойства `key`.

Теперь код создания массива экземпляров компонента в `App.js` примет следующий вид:

```
const jokeComponents = jokesData.map(joke => <Joke key={joke.id}
question={joke.question} punchLine={joke.punchLine} />)
```

Если внести это изменение в код, взглянуть на страницу приложения в браузере и проверить содержимое консоли, то окажется, что уведомление о свойстве `key` исчезло.

После всех преобразований, которым мы подвергли проект, внешний вид страницы приложения не изменился. Однако код компонента `App` стал гораздо короче и понятней, а данные для формирования списка компонентов берутся теперь из чего-то, сильно напоминающего внешний источник данных. Именно по такой схеме работают настоящие приложения.

Кроме того надо отметить, что ядром вышеописанной модификации кода стало использование стандартного метода массивов `map()`. Мы пользовались методикой формирования списка экземпляров

компонента `Joke` в компоненте `App`, но ничто не мешает нам, при необходимости, воспользоваться таким же подходом и в компоненте `Joke`, который может, на основе переданных ему данных, формировать собственный список экземпляров некоего компонента.

При этом, как мы уже говорили, среди стандартных методов массивов можно найти и другие интересные инструменты. Например, метод `sort()` можно использовать для сортировки элементов массивов по некоему признаку. Метод `filter()` можно использовать для отбора только тех элементов массива, которые соответствуют неким критериям. Всё это применимо и при работе с массивами, содержащими экземпляры компонентов.

Если хотите — можете поэкспериментировать с этими методами. Скажем, попытайтесь воспользоваться методом `filter()` и убрать из вывода, формируемого компонентом `App`, те экземпляры компонента `Joke`, свойство `question` которых не превышает заданную длину. Или сделайте так, чтобы в вывод попали бы только компоненты, для которых задано и свойство `question`, и свойство `punchLine`.

Учебный курс по React, часть 12: практикум, третий этап работы над TODO-приложением

Занятие 22. Практикум. Динамическое формирование наборов компонентов

Оригинал

Задание
Взяв за основу стандартный проект React-приложения, формируемый `create-react-app`, и воспользовавшись приведённым ниже кодом файла с данными `vschoolProducts.js`, создайте приложение, которое выводит на странице список компонентов `Product`, формируемый с использованием стандартного метода массива `.map()` на основе данных из `vschoolProducts.js`.

Не забудьте, создавая экземпляры компонента, передавать им атрибут `key` с уникальным идентификатором, иначе система выдаст соответствующее предупреждение.

В процессе решения задачи можете пользоваться следующей заготовкой файла `App.js`:

```
import React from "react"

import productsData from "./vschoolProducts"


function App() {

  return (

    <div>


    </div>

  )

}
```

```
export default App
```

Вот код файла `vschoolProducts.js`:

```
const products = [

    {

        id: "1",

        name: "Pencil",

        price: 1,

        description: "Perfect for those who can't remember things! 5/5 Highly
recommend."

    },

    {

        id: "2",

        name: "Housing",

        price: 0,

        description: "Housing provided for out-of-state students or those who
can't commute"

    },

    {

        id: "3",

        name: "Computer Rental",

        price: 300,

        description: "Don't have a computer? No problem!"

    },

    {

        id: "4",

        name: "Coffee",

        price: 2,

        description: "Wake up!"

    },

    {

        id: "5",
```

```
        name: "Snacks",
        price: 0,
        description: "Free snacks!"
    },
    {
        id: "6",
        name: "Rubber Duckies",
        price: 3.50,
        description: "To help you solve your hardest coding problems."
    },
    {
        id: "7",
        name: "Fidget Spinner",
        price: 21.99,
        description: "Because we like to pretend we're in high school."
    },
    {
        id: "8",
        name: "Sticker Set",
        price: 14.99,
        description: "To prove to other devs you know a lot."
    }
]


export default products
```

Решение

Вот код файла `App.js`:

```
import React from "react"
import Product from "./Product"
import productsData from "./vschoolProducts"
```

```
function App() {

    const productComponents = productsData.map(item => <Product key={item.id}
product={item}/>)



    return (

        <div>

            {productComponents}

        </div>

    )

}



export default App
```

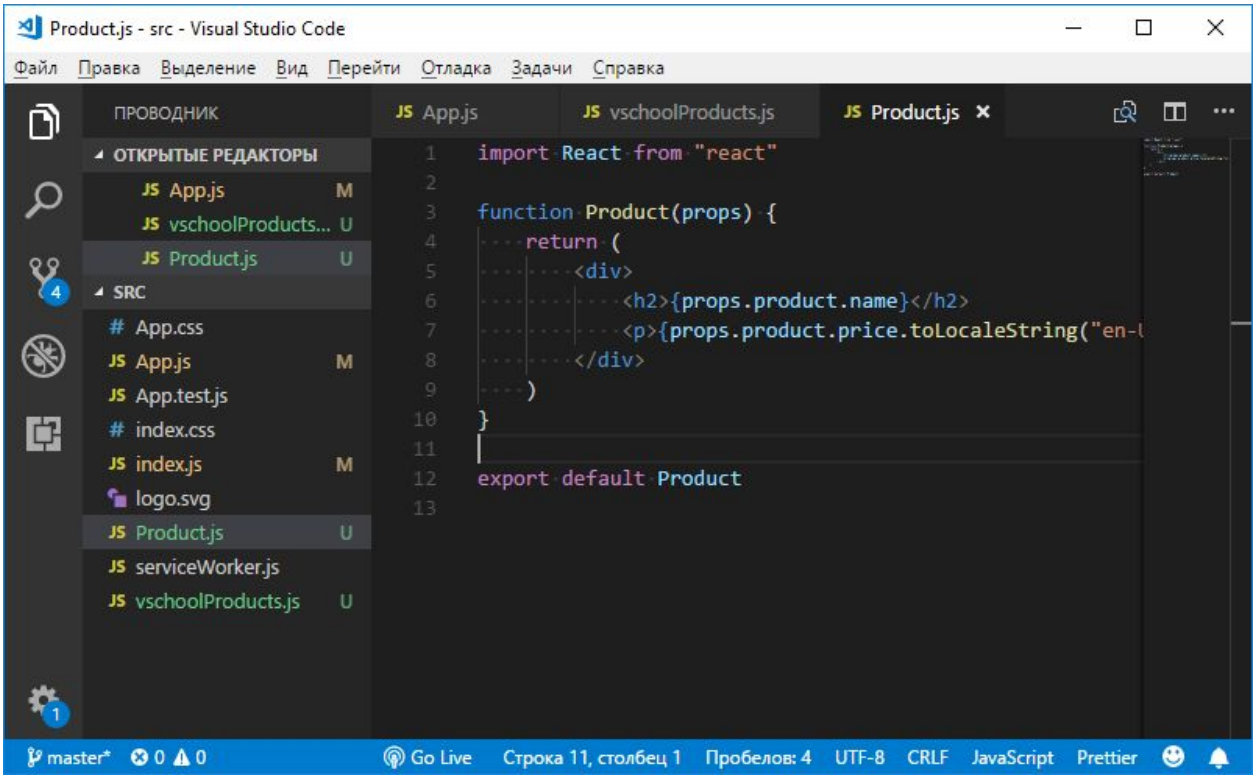
Обратите внимание на то, что свойство `id` объектов из файла с данными выводить на экран необязательно. Это свойство пригодилось нам для задания атрибута `key` создаваемых средствами метода `.map()` экземпляров компонента `Product`.

Вот код файла `Product.js`:

```
import React from "react"



function Product(props) {

    return (

        <div>

            <h2>{props.product.name}</h2>

            <p>{props.product.price.toLocaleString("en-US", { style:
"currency", currency: "USD" })} - {props.product.description}</p>

        </div>

    )

}



export default Product
```

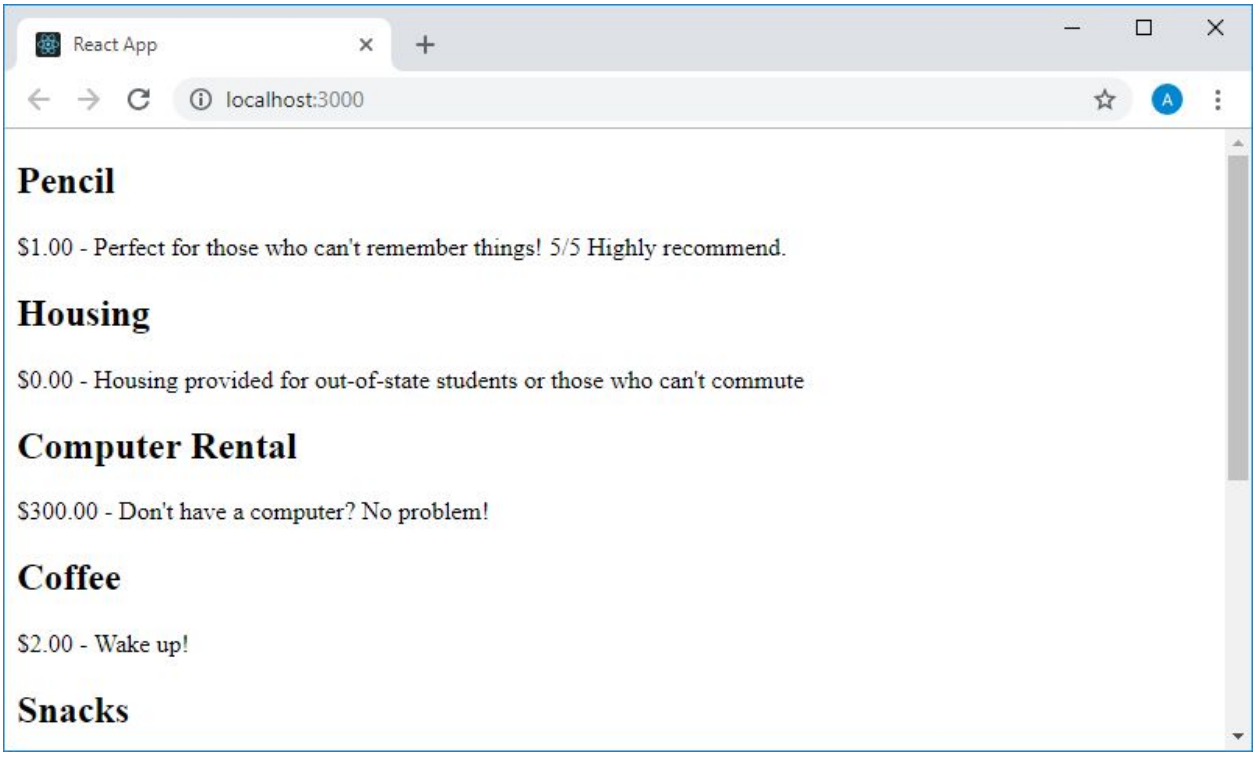
Здесь мы, при выводе свойства, содержащего цену товара, которое видно в компоненте как `props.product.price`, используем метод `toLocaleString()`, средствами которого форматируем сумму товара.

Вот как выглядит проект приложения в VSCode.



Приложение в VSCode

А вот страница приложения в браузере.



Страница приложения в браузере

Обратите внимание на то, в каком виде представлена стоимость товаров.

Занятие 23. Практикум. TODO-приложение. Этап №3

Оригинал

Здесь мы продолжаем работу над TODO-приложением, которой занимались здесь и здесь. В частности, здесь вам будет предложено применить знания о динамическом формировании списков компонентов для создания списка дел, выводимого приложением.

Задание
Используя файл с данными о делах `todosData.js`, содержимое которого приведено ниже, создайте список компонентов `TodoItem` и выведите этот список в компоненте `App`. Обратите внимание на то, что вам нужно будет модифицировать код компонента `TodoItem` таким образом, чтобы он мог бы выводить передаваемые ему свойства.

Вот содержимое файла `todosData.js`:

```
const todosData = [
    {
        id: 1,
        text: "Take out the trash",
        completed: true
    },
    {
        id: 2,
        text: "Grocery shopping",
        completed: false
    },
    {
        id: 3,
        text: "Clean gecko tank",
        completed: false
    },
    {
        id: 4,
        text: "Mow lawn",
        completed: true
    },
    {
        id: 5,
        text: "Catch up on Arrested Development",
```

```
        completed: false
    }
]


export default todosData
```

Решение

Вот код файла `App.js`:

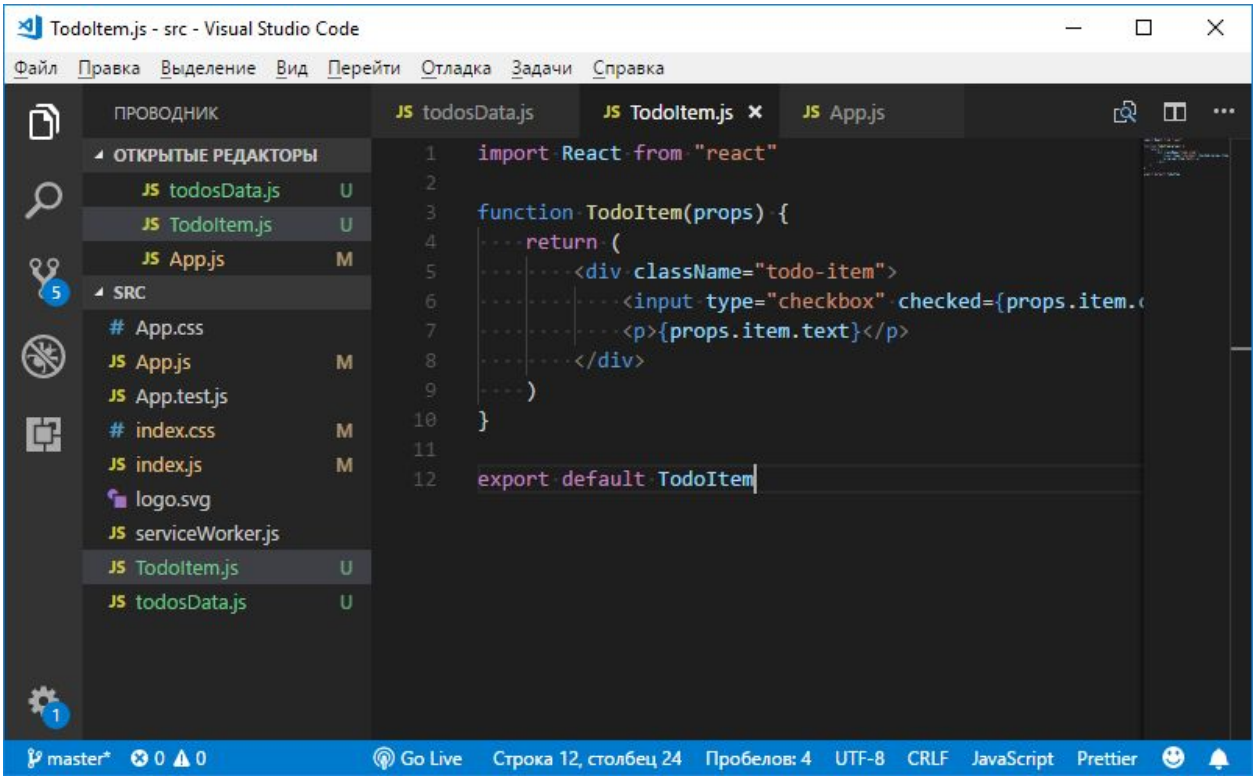
```
import React from "react"
import TodoItem from "./TodoItem"
import todosData from "./todosData"


function App() {
    const todoItems = todosData.map(item => <TodoItem key={item.id}
item={item}/>)


    return (
        <div className="todo-list">
            {todoItems}
        </div>
    )
}


export default App
```

Вот код файла `TodoItem.js`:

```
import React from "react"


function TodoItem(props) {
    return (
        <div className="todo-item">
            <input type="checkbox" checked={props.item.completed}/>
            <p>{props.item.text}</p>
```

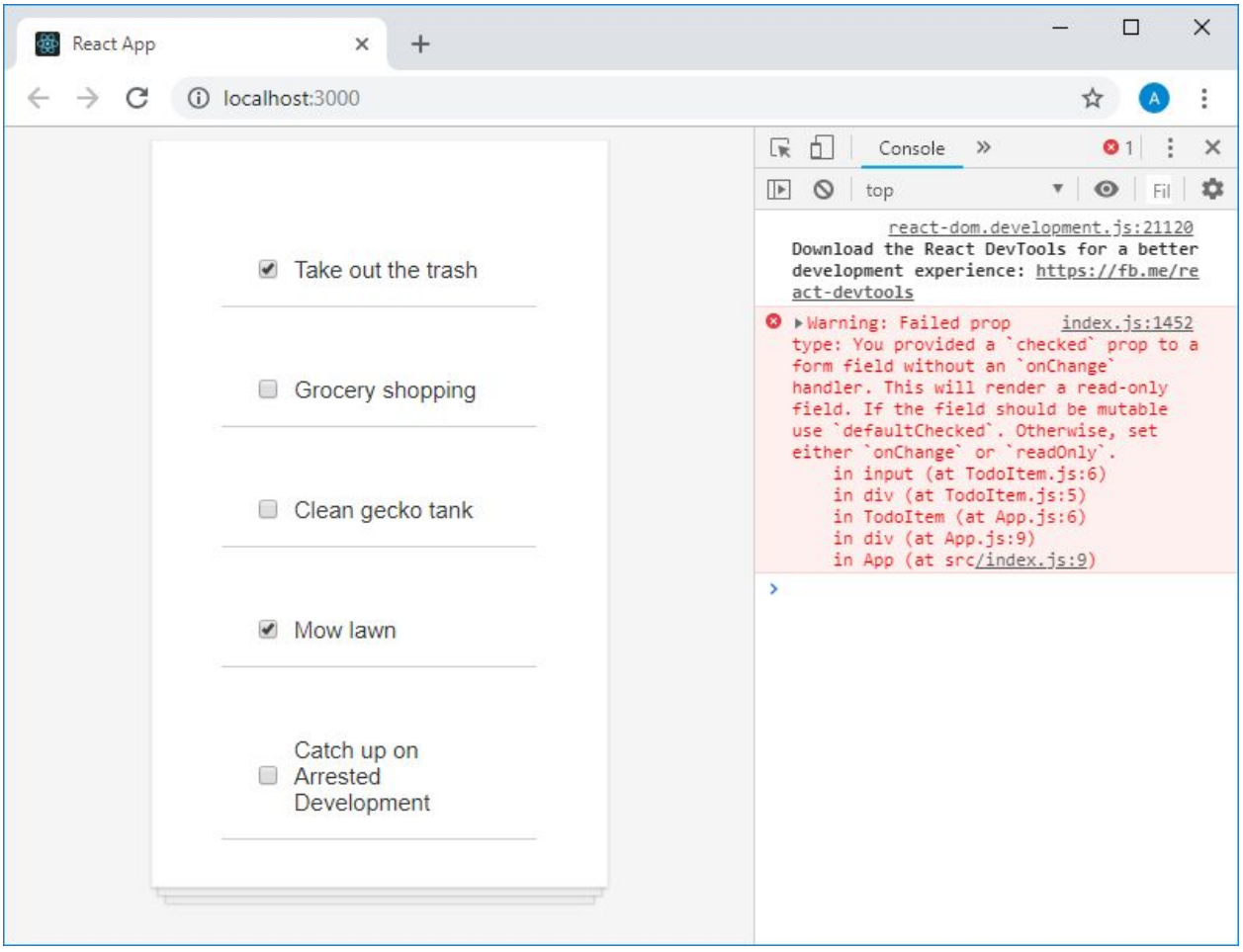
```
        </div>
    )
}


export default TodoItem
```

Вот проект приложения в VSCode.



Приложение в VSCode

Надо отметить, что после завершения работ, предусмотренных этим практическим занятием, мы, несмотря на то, что оснастим приложение новыми возможностями, нарушим его функционал. В частности, речь идёт о состоянии флажков. Флажки, для настройки состояния которых использовалось свойство `props.item.completed`, установленное в значение `true`, будут установлены, флажки, для которых использовалось это же свойство, установленное в `false`, будут сняты. Но если щёлкнуть мышью по флажку на странице приложения — ничего не произойдёт, так как мы неправильно настроили соответствующий HTML-элемент. В консоли можно видеть предупреждение об этом.



Страница приложения в браузере и предупреждение в консоли

Позже мы будем говорить о формах и исправим наше учебное приложение при продолжении работы над ним.

Учебный курс по React, часть 13: компоненты, основанные на классах

Занятие 24. Компоненты, основанные на классах

Оригинал

Если вы, до того, как начали осваивать этот учебный курс, изучали React по материалам каких-то других курсов, у вас может возникнуть вопрос по поводу того, что здесь мы пользуемся функциональными компонентами. Дело в том, что во многих других курсах эта тема либо не освещается, либо о функциональных компонентах говорят как о чём-то таком, в чём нет особой необходимости. Некоторые авторы идут ещё дальше и говорят о том, что функциональные компоненты лучше не использовать, отдавая предпочтение компонентам, основанным на классах. Это, по их мнению, избавляет программиста от ненужной нагрузки. Я же полагаю, что любому, кто изучает React, полезно будет увидеть полную картину и узнать о популярных в наши дни подходах по работе с компонентами. В частности, сейчас актуально направление, в соответствии с которым везде, где это возможно, используют функциональные компоненты, а компоненты, основанные на классах — лишь там, где они действительно необходимы. При этом надо отметить, что всё это — лишь рекомендации. Каждый разработчик сам решает как именно он будет конструировать свои приложения.

Когда я веду курсы по React, я предпочитаю начинать с функциональных компонентов, так как функции — понятные конструкции. Одного взгляда на функциональный компонент достаточно для того, чтобы понять, какие именно действия он выполняет. Скажем, вот код функционального компонента, который

представляет собой обычную функцию, возвращающую элемент `<div>`, содержащий элемент `<h1>` с неким текстом.

```
function App() {

    return (

        <div>

            <h1>Code goes here</h1>

        </div>

    )

}
```

Но, по мере того, как мы углубляемся в изучение React, знакомимся с его возможностями, оказывается, что функциональные компоненты не способны предложить нам всё то, что может понадобиться нам от React-компонентов. Поэтому сегодня мы поговорим о компонентах, основанных на классах. А именно, начнём с создания компонента, основанного на классе, который выполняет те же действия, что и вышеприведённый функциональный компонент. А на следующих занятиях мы коснёмся тех дополнительных возможностей, которые дают нам компоненты, основанные на классах. В частности, речь идёт о возможности работы с состоянием компонентов и с методами их жизненного цикла.

Преобразуем функциональный компонент в компонент, основанный на классе. Если вы не особенно хорошо знакомы с ключевым словом `class`, появившемся в ES6, и с возможностями, которые оно открывает перед разработчиками, рекомендуется уделить некоторое время на то, чтобы познакомиться с классами поближе.

Описание компонента, основанного на классах, начинается с ключевого слова `class`. Затем идёт имя компонента, составляемое по тем же правилам, что и имена функциональных компонентов. При этом после конструкции наподобие `class App` будет идти не нечто вроде фигурной скобки, а конструкция вида `extends React.Component`. После неё ставится пара фигурных скобок, в которых будет описано тело класса.

Классы в JavaScript представляют собой надстройку над традиционной моделью прототипного наследования. Сущность конструкции `class App extends React.Component` сводится к тому, что мы объявляем новый класс и указываем на то, что его прототипом должен быть `React.Component`. Наличие у нашего компонента этого прототипа позволяет пользоваться в этом компоненте всеми теми полезными возможностями, которые имеются в `React.Component`.

Итак, на данном этапе работы над компонентом, основанном на классах, его код выглядит так:

```
class App extends React.Component {



}
```

У компонента, основанного на классах, должен быть, по меньшей мере, один метод. Это — метод `render()`. Данный метод должен возвращать то же самое, что мы обычно возвращаем из функциональных компонентов. Вот как выглядит полный код компонента, основанного на классах, реализующего те же возможности, что и вышеприведённый функциональный компонент.

```
class App extends React.Component {
```

```
    render() {
        return (
            <div>
                <h1>Code goes here</h1>
            </div>
        )
    }
}
```

Работают с компонентами, основанными на классах так же, как с функциональными компонентами. То есть, в нашем случае достаточно заменить код функционального компонента на новый код и приложение будет работать так же, как и прежде.

Поговорим о методе `render()`. Если, перед формированием элементов, возвращаемых этим методом, нужно выполнить некие вычисления, их выполняют именно в этом методе, перед командой `return`. То есть, если у вас есть некий код, определяющий порядок формирования визуального представления компонента, этот код нужно поместить в метод `render`. Например, тут можно выполнить настройку стилей в том случае, если вы пользуетесь встроенными стилями. Здесь же будет и код, реализующий механизм условного рендеринга, и другие подобные конструкции.

Если вы знакомы с классами, вы можете создать собственный метод и разместить код, готовящий компонент к визуализации, в нём, после чего вызвать этот метод в методе `render`. Выглядит это так:

```
class App extends React.Component {

    yourMethodHere() {

    }

    render() {
        const style = this.yourMethodHere()
        return (
            <div>
                <h1>Code goes here</h1>
            </div>
        )
    }
}
```

А именно, тут мы исходим из предположения о том, что в методе `yourMethodHere()` производится формирование стилей, а то, что он возвращает, записывается в константу `style`, объявленную в методе `render()`. Обратите внимание на то, что для обращения к нашему собственному методу используется ключевое слово `this`. Позже мы поговорим об особенностях этого ключевого слова, но пока остановимся на представленной здесь конструкции.

Теперь поговорим о том, как в компонентах, основанных на классах, работать со свойствами, передаваемыми им при создании их экземпляров.

При использовании функциональных компонентов мы объявляли соответствующую функцию с параметром `props`, представляющим собой объект, в который попадало то, что передавалось компоненту при создании его экземпляра. Выглядит это так:

```
function App(props) {

    return (

        <div>

            <h1>{props.whatever}</h1>

        </div>

    )

}
```

При работе с компонентом, основанном на классе, то же самое выглядит так:

```
class App extends React.Component {

    render() {

        return (

            <div>

                <h1>{this.props.whatever}</h1>

            </div>

        )

    }

}
```

Учебный курс по React, часть 14: практикум по компонентам, основанным на классах, состояние компонентов

Занятие 25. Практикум. Компоненты, основанные на классах

Оригинал

Задание
Ниже представлен код, который нужно поместить в файл `index.js` стандартного React-приложения, созданного средствами `create-react-app`. Преобразуйте функциональные компоненты, которые вы

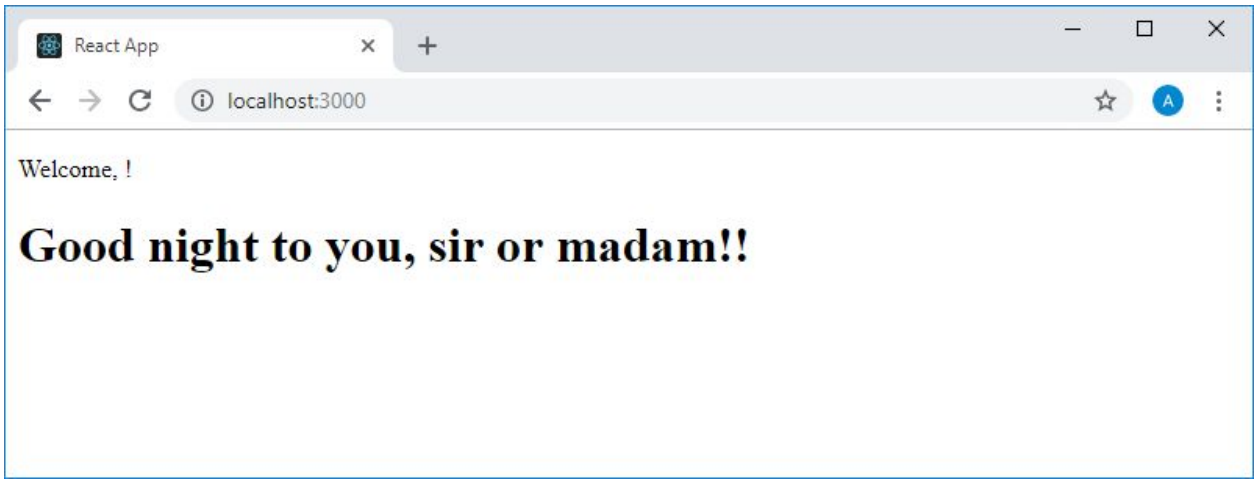
встретите в этом коде, в компоненты, основанные на классах, и, кроме того, найдите и исправьте небольшую ошибку.

Код файла `index.js`:

```
import React from "react"
import ReactDOM from "react-dom"


// #1
function App() {
    return (
        <div>
            <Header />
            <Greeting />
        </div>
    )
}


// #2
function Header(props) {
    return (
        <header>
            <p>Welcome, {props.username}!</p>
        </header>
    )
}


// #3
function Greeting() {
    const date = new Date()
    const hours = date.getHours()
    let timeOfDay
```

```
    if (hours < 12) {
        timeOfDay = "morning"
    } else if (hours >= 12 && hours < 17) {
        timeOfDay = "afternoon"
    } else {
        timeOfDay = "night"
    }
    return (
        <h1>Good {timeOfDay} to you, sir or madam!</h1>
    )
}

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

Решение
Для начала посмотрим на то, что выдаёт приложение в его исходном виде, открыв его в браузере.



Страница исходного приложения в браузере

Видно, что верхняя строка, которая выводится на странице, выглядит неправильно. После запятой, следующей за Welcome, очевидно, должно быть нечто вроде имени пользователя.

Если проанализировать код приложения, то окажется, что эту строчку выводит компонент `Header`, ожидая получить свойство `username`, задаваемое при создании его экземпляра. Экземпляр этого компонента создаётся в компоненте `App`. Выяснив это, мы сможем исправить ту самую ошибку, о которой шла речь в задании.

Надо отметить, что компоненты обычно размещают в разных файлах, но в данном случае мы описали их все в одном файле.

Приступим к преобразованию функционального компонента `App` в компонент, основанный на классе. Для этого достаточно привести его код к такому виду:

```
class App extends React.Component {

    render() {

        return (

            <div>

                <Header username="vschool"/>

                <Greeting />

            </div>

        )

    }

}
```

Перед именем компонента теперь идёт ключевое слово `class`, дальше следует команда `extends React.Component`, после чего, в фигурных скобках, описывается тело класса. Здесь должен присутствовать метод `render()`, возвращающий то, что мы возвращали из функционального компонента. По такому же принципу перерабатываются и другие компоненты. Обратите внимание на конструкцию `<Header username="vschool"/>`. Здесь мы передаём компоненту `Header` свойство `username` со значением `vschool`, исправляя тем самым ошибку, которая имеется в исходном приложении.

Как вы уже знаете, компонент `Header` ожидает получение свойства `username`, и в функциональном компоненте доступ к этому свойству осуществляется с помощью конструкции `props.username` (`props` в данном случае — это аргумент функции, описывающей компонент). В компонентах, основанных на классах то же самое выглядит как `this.props.username`. Вот переработанный код компонента `Header`:

```
class Header extends React.Component {

    render() {

        return (

            <header>

                <p>Welcome, {this.props.username}!</p>

            </header>

        )

    }

}
```

Третий компонент, `Greeting`, немного отличается от других. Дело в том, что в нём, перед командой `return`, выполняются некоторые вычисления. При преобразовании его в компонент, основанный на классе, эти вычисления нужно поместить в метод `render()` до команды возврата. Вот код переработанного компонента `Greeting`:

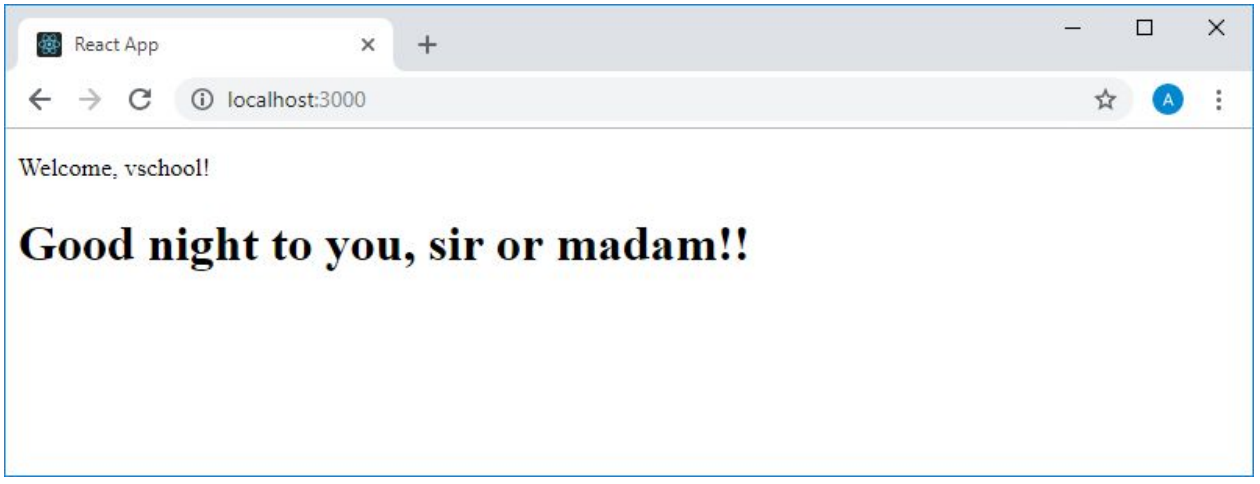
```
class Greeting extends Component {
```

```
    render() {
        const date = new Date()
        const hours = date.getHours()
        let timeOfDay

        if (hours < 12) {
            timeOfDay = "morning"
        } else if (hours >= 12 && hours < 17) {
            timeOfDay = "afternoon"
        } else {
            timeOfDay = "night"
        }
        return (
            <h1>Good {timeOfDay} to you, sir or madam!</h1>
        )
    }
}
```

Обратите внимание на то, что при объявлении этого компонента использована такая конструкция: `class Greeting extends Component`. Часто так делают ради краткости кода, но для того, чтобы это сработало, нам нужно доработать команду импорта `react`, приведя её к такому виду:

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

Вот как выглядит страница переработанного приложения в браузере.



Страница переработанного приложения в браузере

Собственно говоря, выглядит она так же, как и страница исходного приложения, а единственное заметное различие между этими страницами заключается в том, что теперь в первой строчке выводится переданное компоненту `Header` имя пользователя.

Вот полный код переработанного файла `index.js`:

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

import ReactDOM from "react-dom"


// #1

class App extends React.Component {

    render() {

        return (

            <div>

                <Header username="vschool"/>

                <Greeting />

            </div>

        )

    }

}


// #2

class Header extends React.Component {

    render() {

        return (

            <header>

                <p>Welcome, {this.props.username}!</p>

            </header>

        )

    }

}


// #3
```

```
class Greeting extends Component {

    render() {

        const date = new Date()

        const hours = date.getHours()

        let timeOfDay


        if (hours < 12) {

            timeOfDay = "morning"

        } else if (hours >= 12 && hours < 17) {

            timeOfDay = "afternoon"

        } else {

            timeOfDay = "night"

        }

        return (

            <h1>Good {timeOfDay} to you, sir or madam!</h1>

        )

    }

}


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

Если выполнение этого практического задания не вызвало у вас сложностей — замечательно. Если же вы пока ещё не вполне освоились с компонентами, основанными на классах — уделите время на эксперименты с ними. Например, можете снова переделать компоненты, основанные на классах, в функциональные компоненты, а потом выполнить обратное преобразование.

Занятие 26. Состояние компонентов

Оригинал

Состояние (state) — это невероятно важная концепция React. Если компонент нуждается в хранении каких-либо собственных данных и в управлении этими данными (в отличие от ситуации, когда данные ему передаёт родительский компонент, используя механизм свойств), используют состояние компонента. Сегодня мы рассмотрим основные понятия, касающиеся состояния компонентов.

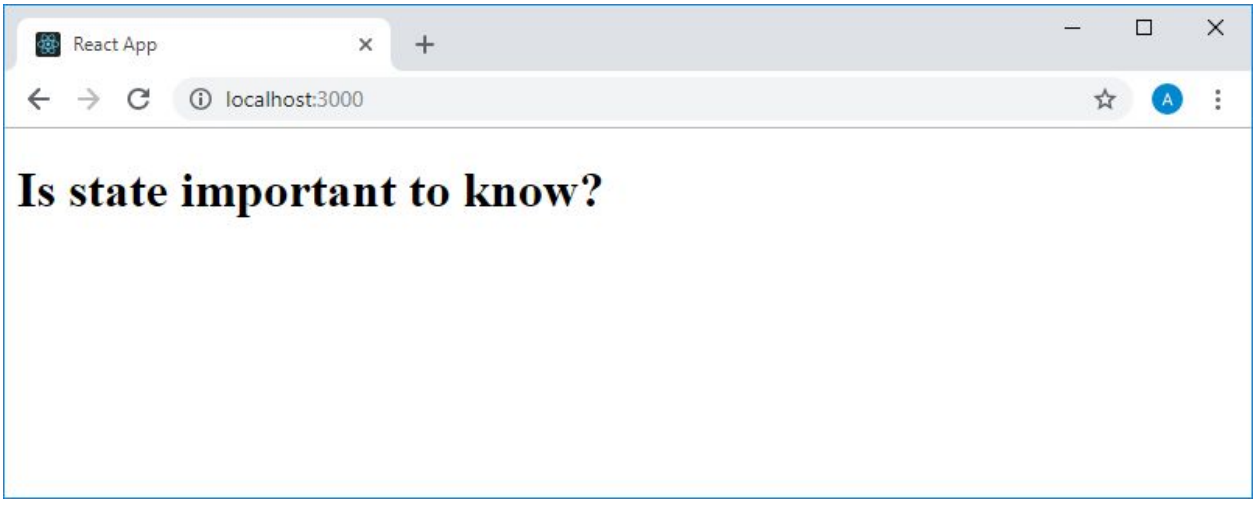
Состояние — это всего лишь данные, которыми управляет компонент. В частности, это означает, что компонент может эти данные менять. При этом уже знакомые нам свойства, получаемые компонентом от родительского компонента, компонент-получатель изменить не может. Они, в соответствии с документацией React, иммутабельны (неизменяемы). Например, если попытаться, в компоненте,

основанном на классе, использовать конструкцию наподобие `this.props.name = "NoName"` — мы столкнёмся с сообщением об ошибке.

Нужно отметить, что если некоему компоненту нужно работать с состоянием, то это должен быть компонент, основанный на классе. Поговорим о том, как оснастить компонент состоянием, начав работу со следующего фрагмента кода, представляющего собой содержимое файла `App.js` стандартного проекта, созданного средствами `create-react-app`:

```
import React from "react"


class App extends React.Component {
    render() {
        return (
            <div>
                <h1>Is state important to know?</h1>
            </div>
        )
    }
}


export default App
```

Вот как выглядит страница приложения в браузере.



Страница приложения в браузере

Для того чтобы оснастить компонент состоянием сначала нужно создать конструктор класса. Он выглядит как метод класса `constructor()`. После этого код компонента будет выглядеть так:

```
class App extends React.Component {
```

```
    constructor() {


    }


    render() {
        return (
            <div>
                <h1>Is state important to know?</h1>
            </div>
        )
    }
}
```

`Constructor()` — это специальный метод, встроенный в JavaScript, который предназначен для создания и инициализации объектов, основанных на классах. Собственно говоря, если при создании объекта нужно что-то инициализировать, соответствующие операции выполняются именно в методе `constructor()`.

Первое, что нужно сделать в коде конструктора — выполнить вызов конструктора родительского класса. Делается это с помощью функции `super()`. В конструкторе родительского класса могут выполняться некие операции по инициализации, результаты выполнения которых пригодятся и нашему объекту. Вот как теперь будет выглядеть конструктор нашего класса:

```
constructor() {
    super()
}
```

Теперь, для того, чтобы оснастить компонент состоянием, нам нужно, в конструкторе, добавить к экземпляру класса свойство `state`. Это свойство является объектом:

```
constructor() {
    super()
    this.state = {}
}
```

Здесь мы инициализировали его пустым объектом. Работать с состоянием в коде компонента можно, используя конструкцию `this.state`. Добавим в состояние новое свойство:

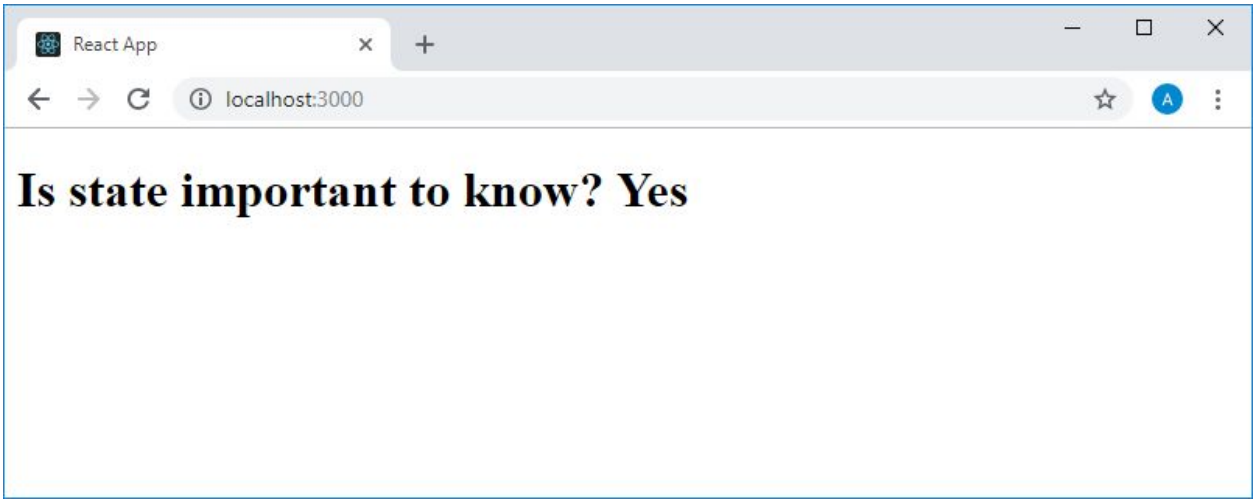
```
constructor() {
    super()
    this.state = {
```

```
        answer: "Yes"
    }
}
```

Подумаем теперь о том, как воспользоваться тем, что хранится в состоянии, в коде. Вспомним о том, что компонент выводит на экран вопрос `Is state important to know?`. В состоянии хранится ответ на этот вопрос. Для того чтобы добавить этот ответ после вопроса, нужно поступить так же, как мы обычно поступаем, добавляя JavaScript-конструкции в JSX-код. А именно, надо добавить в конец строки конструкцию `{this.state.answer}`. В результате полный код компонента будет выглядеть так:

```
class App extends React.Component {
    constructor() {
        super()
        this.state = {
            answer: "Yes"
        }
    }

    render() {
        return (
            <div>
                <h1>Is state important to know? {this.state.answer}</h1>
            </div>
        )
    }
}
```

А вот как будет выглядеть страница приложения в браузере.



Страница приложения в браузере

Тут хотелось бы отметить, что состояние, которое получает компонент при инициализации, можно менять в процессе работы компонента. Кроме того, компоненты могут передавать состояние компонентам-потомкам с помощью уже известного вам механизма работы со свойствами. Например, в нашем случае, если предположить, что имеется некий компонент `ChildComponent`, данные из состояния ему можно передать так:

```
<ChildComponent answer={this.state.answer}/>
```

Пока мы не будем подробно говорить о том, как менять данные, хранящиеся в состоянии компонента. Отметим лишь, что при вызове метода `setState()`, который используется для решения этой задачи, будет изменено не только состояние компонента, но и обновлены данные состояния, переданные через механизм свойств его дочерним компонентам. Кроме того, изменение состояния приведёт к тому, что данные из состояния, отображающиеся на странице приложения, автоматически изменятся.

Учебный курс по React, часть 15: практикумы по работе с состоянием компонентов

Занятие 27. Практикум. Состояние компонентов, отладка

Оригинал

Задание
Проанализируйте представленный ниже код класса `App` из файла `App.js` стандартного React-приложения, созданного средствами `create-react-app`. Этот код неполон, в нём есть ошибки.

```
import React from "react"


class App extends Component() {

    return (

        <div>

            <h1>{this.state.name}</h1>

            <h3>{this.state.age} years old</h3>
```

```
        </div>
    )
}


export default App
```

У компонента `App`, основанного на классе, нет конструктора, его состояние не инициализировано, но при формировании того, что он возвращает, подразумевается наличие у него состояния с некими данными.

Ваша задача заключается в том, чтобы привести этот код в работоспособное состояние.

Если вы столкнётесь с неким неизвестным вам сообщением об ошибке — не спешите заглядывать в решение. Попытайтесь самостоятельно, например, внимательно вчитавшись в код и поискав сведения о проблеме в интернете, выяснить причину ошибки и её исправить. Отладка кода — ценнейший навык, который обязательно пригодиться вам при работе над реальными проектами.

Решение
Тело класса похоже на тело функционального компонента. В нём имеется лишь команда `return`, но в компонентах, основанных на классах, эта команда используется в методе `render()`, а не в теле класса. Исправим это.

```
import React from "react"

class App extends Component() {
    render() {
        return (
            <div>
                <h1>{this.state.name}</h1>
                <h3>{this.state.age} years old</h3>
            </div>
        )
    }
}


export default App
```

Если продолжить анализ кода, поглядывая на сообщения об ошибках, выводимые в браузере, можно понять, что хотя конструкция `class App extends Component` выглядит вполне нормально, с тем, к чему мы обращаемся по имени `Component`, всё ещё что-то не так. Проблема заключается в том, что в команде импорта, `import React from "react"`, мы импортируем в проект лишь `React`, но не `Component`. То есть, нам надо либо отредактировать эту команду, приведя её к виду `import React, {Component} from "react"`, тогда при создании класса мы сможем воспользоваться уже

существующим кодом, либо переписать объявление класса в таком виде: `class App extends React.Component`. Мы остановимся на первом варианте. Теперь код компонента выглядит так:

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


class App extends Component() {
    render() {
        return (
            <div>
                <h1>{this.state.name}</h1>
                <h3>{this.state.age} years old</h3>
            </div>
        )
    }
}


export default App
```

Правда, на этом проблемы не заканчиваются. Приложение не работает, в браузере появляется сообщение об ошибке `TypeError: Cannot set property 'props' of undefined`, нам сообщают, что что-то не так с первой строкой объявления класса.

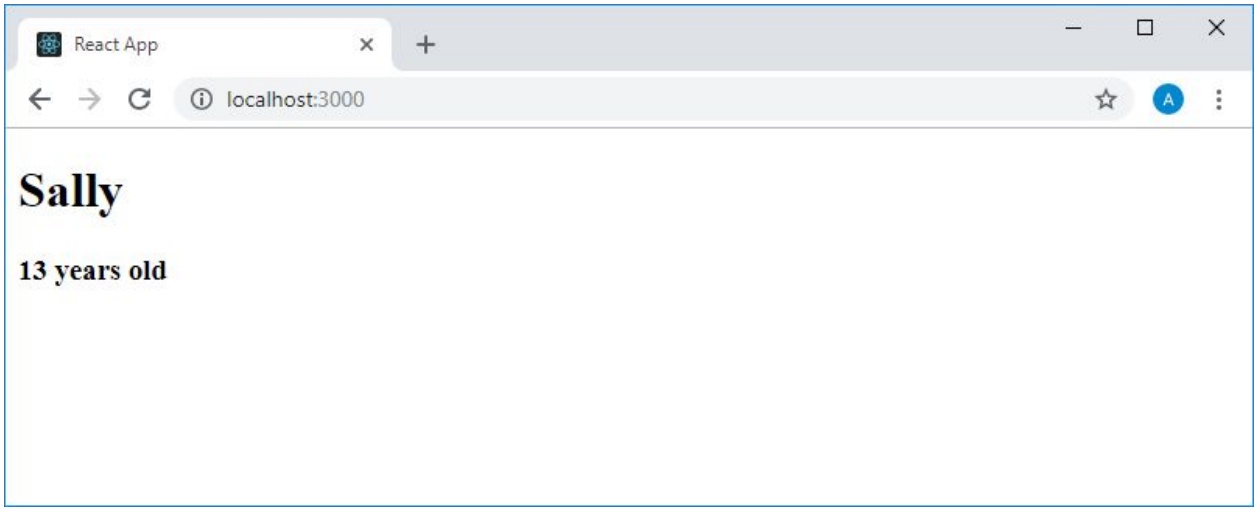
Дело тут в том, что при объявлении компонента, который, как мы уже поняли, является компонентом, который основан на классе, после имени родительского класса стоит пара круглых скобок. Они тут не нужны, это не функциональный компонент, поэтому от них мы избавимся. Теперь код приложения оказывается более или менее работоспособным, компонент, основанный на классе, уже не выглядит совершенно неправильным, но система продолжает сообщать нам об ошибках. Теперь сообщение об ошибке выглядит так: `TypeError: Cannot read property 'name' of null`. Очевидно, оно относится к попытке использования данных, хранящихся в состоянии компонента, которое пока не инициализировано. Поэтому теперь мы создадим конструктор класса, не забыв вызвать в нём `super()`, и инициализируем состояние компонента, добавив в него значения, с которыми компонент пытается работать.

Вот готовый рабочий код компонента `App`:

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


class App extends Component {
    constructor() {
        super()
```

```
        this.state = {
            name: "Sally",
            age: 13
        }
    }

    render() {
        return (
            <div>
                <h1>{this.state.name}</h1>
                <h3>{this.state.age} years old</h3>
            </div>
        )
    }
}

export default App
```

Вот как будет выглядеть страница приложения в браузере.



Страница приложения в браузере

Занятие 28. Практикум. Состояние компонентов, работа с данными, хранящимися в состоянии

Оригинал

На этом практическом занятии у вас будет ещё один шанс поработать с состоянием компонентов.

Задание

Ниже приведён код функционального компонента. Взяв его за основу, сделайте следующее:

1. Преобразуйте его так, чтобы у компонента было бы состояние.
2. В состоянии компонента должно присутствовать свойство `isLoggedIn`, хранящее логическое значение `true`, если пользователь вошёл в систему, и `false` — если нет (в нашем случае никаких механизмов «входа в систему» здесь нет, соответствующее значение нужно установить вручную, при инициализации состояния).
3. Постарайтесь сделать так, чтобы, если пользователь в систему вошёл, компонент выводил бы текст `You are currently logged in`, а если нет — то текст `You are currently logged out`. Это задание является необязательным, если у вас возникнут трудности при его выполнении, можете, например, поискать идеи в интернете.

Вот код файла `App.js`:

```
import React from "react"



function App() {

    return (

        <div>

            <h1>You are currently logged (in/out)</h1>

        </div>

    )

}



export default App
```

Решение

В нашем распоряжении имеется функциональный компонент. Для того чтобы оснастить его состоянием, его надо преобразовать в компонент, основанный на классе и инициализировать его состояние. Вот как выглядит код преобразованного компонента:

```
import React from "react"



class App extends React.Component {

    constructor() {

        super()

        this.state = {

            isLoggedIn: true

        }
```

```
    }


    render() {
        return (
            <div>
                <h1>You are currently logged (in/out)</h1>
            </div>
        )
    }
}


export default App
```

Проверим работоспособность приложения.

Приложение в браузере

На самом деле, если вы самостоятельно дошли до этого момента, это значит, что вы усвоили данный в курсе материал, посвящённый компонентам, основанным на классах и состоянию компонентов. Теперь займёмся необязательным заданием.

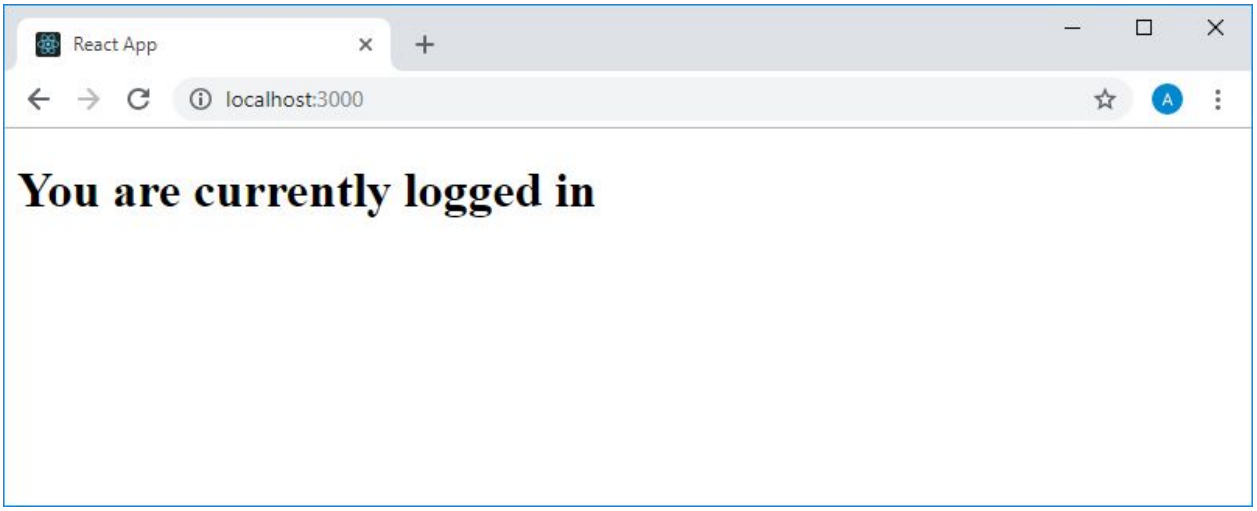
В сущности, то, что нужно сделать для выполнения этого задания, мы будем обсуждать на занятии, которое посвящено условному рендерингу, так что здесь мы немного забегаем вперёд. Итак, мы собираемся объявить и инициализировать переменную, которая будет содержать строку `in` или `out` в зависимости от того, что именно хранится в свойстве состояния `isLoggedIn`. Работать с этой переменной мы будем в методе `render()`, сначала анализируя данные и записывая в неё нужное значение, а потом используя её в возвращаемом компонентом JSX-коде. Вот что у нас в итоге получилось:

```
import React from "react"
```

```
class App extends React.Component {
    constructor() {
        super()
        this.state = {
            isLoggedIn: true
        }
    }

    render() {
        let wordDisplay
        if (this.state.isLoggedIn === true) {
            wordDisplay = "in"
        } else {
            wordDisplay = "out"
        }
        return (
            <div>
                <h1>You are currently logged {wordDisplay}</h1>
            </div>
        )
    }
}

export default App
```

Обратите внимание на то, что переменная `wordDisplay` — это обычная локальная переменная, объявленная в методе `render()`, поэтому для обращения к ней внутри этого метода достаточно лишь указать её имя.

Вот как теперь будет выглядеть страница приложения



Страница приложения в браузере

Свойство состояния `isLoggedIn` установлено в значение `true`, поэтому на странице выводится текст `You are currently logged in`. Для вывода текста `You are currently logged out` нужно поменять, в коде компонента, значение `isLoggedIn` на `false`.

Надо отметить, что эту задачу можно решить и другими способами. Но код, который у нас получился, понятен и работоспособен, поэтому мы остановимся на нём, хотя, например, учитывая то, что `isLoggedIn` — логическая переменная, условие `if (this.state.isLoggedIn === true)` можно переписать как `if (this.state.isLoggedIn)`.

Учебный курс по React, часть 16: четвёртый этап работы над TODO-приложением, обработка событий

Занятие 29. Практикум. TODO-приложение. Этап №4

Оригинал

Задание
В прошлый раз мы загружали список дел для приложения из JSON-файла, а потом, проходясь по полученному массиву, формировали, с помощью метода `map()`, набор компонентов. Нам хотелось бы модифицировать эти данные. А это мы сможем сделать только в том случае, если предварительно загрузим их в состояние компонента.

Сегодняшнее задание заключается в том, чтобы вы преобразовали компонент `App` в компонент с состоянием и загрузили бы импортированные данные о делах в состояние этого компонента.

Решение
Вспомним уже имеющийся в нашем проекте код компонента `App`:

```
import React from "react"

import TodoItem from "./TodoItem"

import todosData from "./todosData"


function App() {
```

```
    const todoItems = todosData.map(item => <TodoItem key={item.id}
item={item}/>)


    return (
        <div className="todo-list">
            {todoItems}
        </div>
    )
}


export default App
```

Для того чтобы получить возможность модифицировать данные из списка дел нам нужно чтобы то, что сейчас хранится в `todosData`, было бы помещено в состояние компонента `App`.

Решая эту задачу, мы сначала должны преобразовать функциональный компонент `App` в компонент, основанный на классе. Потом нам нужно загрузить данные из `todosData` в состояние и, формируя список компонентов `TodoItem`, обходить уже не массив `todosData`, а массив с такими же данными, хранящийся в состоянии. Вот как это будет выглядеть:

```
import React from "react"
import TodoItem from "./TodoItem"
import todosData from "./todosData"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            todos: todosData
        }
    }


    render() {
        const todoItems = this.state.todos.map(item => <TodoItem key={item.id}
item={item}/>)
```

```
        return (
            <div className="todo-list">
                {todoItems}
            </div>
        )
    }
}


export default App
```

Надо отметить, что после всех этих преобразований внешний вид приложения не изменился, но выполнив их, мы подготовили его к дальнейшей работе над ним.

Занятие 30. Обработка событий в React

Оригинал

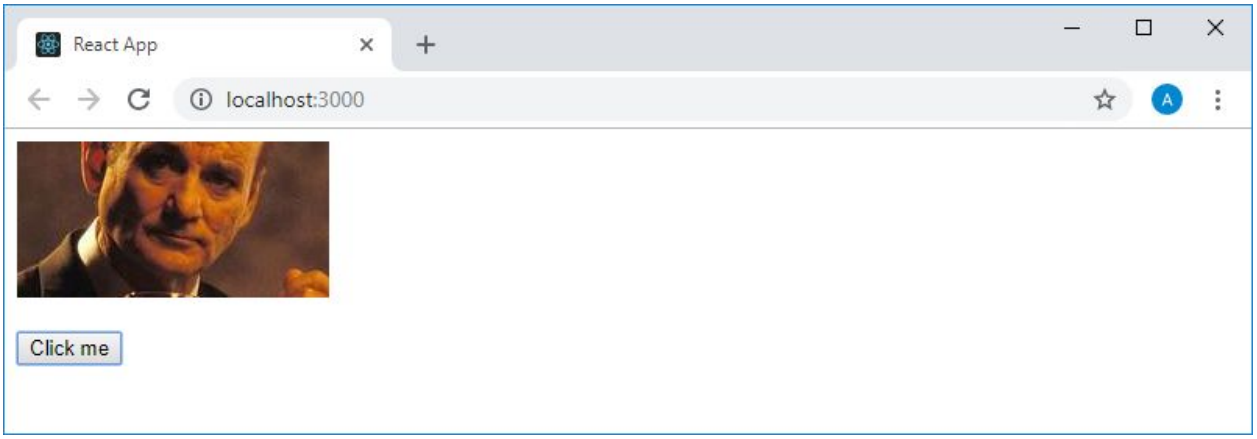
Обработка событий — это то, что является движущей силой веб-приложений, и то, что отличает их от простых статических веб-сайтов. Обработка событий в React устроена довольно просто, она очень похожа на то, как события обрабатываются в обычном HTML. Так, например, в React имеются обработчики событий `onClick` и `onSubmit`, которые сходны с аналогичными механизмами HTML, представленными в виде `onclick` и `onsubmit`, не только в плане имён (в React, правда, их имена формируются с использованием верблюжьего стиля), но и в том, как именно с ними работают.

Здесь мы будем рассматривать примеры, экспериментируя со стандартным приложением, создаваемым средствами `create-react-app`, файл компонента `App` которого содержит следующий код:

```
import React from "react"


function App() {
    return (
        <div>
            <img src="https://www.fillmurray.com/200/100"/>
            <br />
            <br />
            <button>Click me</button>
        </div>
    )
}
```

```
export default App
```

Вот как выглядит наше приложение в браузере.



Страница приложения в браузере

Прежде чем мы сможем серьёзно говорить о модификации состояния компонентов с помощью метода `setState()`, нам нужно разобраться с событиями и с обработкой событий в React. Механизмы обработки событий позволяют пользователю приложения взаимодействовать с ним. Приложение же может реагировать, например, на события `click` или `hover`, выполняя при возникновении этих событий некие действия.

Обработка событий в React, на самом деле, устроена довольно просто. Если вам знакомы стандартные механизмы HTML, используемые для назначения элементам управления обработчиков событий, наподобие обработчика события `onclick`, то вы сразу же увидите сходство с этими механизмами того, что предлагает нам React.

Например, для того, чтобы средствами HTML сделать так, чтобы по нажатию на некую кнопку выполнялась бы какая-то функция, можно воспользоваться такой конструкцией (при условии существования и доступности этой функции):

```
<button onclick="myFunction()">Click me</button>
```

В React, как уже было сказано, обработчики событий имеют имена, составленные по правилам верблюжьего стиля, то есть `onclick` превратится здесь в `onClick`. То же самое справедливо и для обработчика события `onMouseOver`, и для других обработчиков. Причина подобного изменения заключается в том, что здесь используется подход к именованию сущностей, обычный для JavaScript.

Поработаем теперь с нашим кодом и сделаем так, чтобы кнопка реагировала бы на щелчки по ней. Вместо того чтобы передавать обработчику код для вызова функции в виде строки, мы передаём имя функции в фигурных скобках. Заготовка соответствующего фрагмента нашего кода будет теперь выглядеть так:

```
<button onClick={}>Click me</button>
```

Если вы взглянете на код компонента `App`, который мы используем в этом примере, вы заметите, что там пока не объявлена функция, которую планируется вызывать при нажатии на кнопку. В общем-то, прямо сейчас мы вполне можем обойтись анонимной функцией, объявленной прямо в коде, описывающем кнопку. Вот как это будет выглядеть:

```
<button onClick={() => console.log("I was clicked!")}>Click me</button>
```

Теперь при нажатии на кнопку в консоль попадёт текст `I was clicked!`.

Того же эффекта можно добиться, объявив самостоятельную функцию и приведя код файла компонента к следующему виду:

```
import React from "react"


function handleClick() {

    console.log("I was clicked")

}


function App() {

    return (

        <div>

            <img src="https://www.fillmurray.com/200/100"/>

            <br />

            <br />

            <button onClick={handleClick}>Click me</button>

        </div>

    )

}


export default App
```

Для того чтобы ознакомиться с полным списком событий, поддерживаемых React, загляните на эту страницу документации.

Теперь попытайтесь оснастить наше приложение новой возможностью. А именно — сделайте так, чтобы при наведении мыши на изображение в консоль выводилось бы какое-нибудь сообщение. Для этого вам нужно найти подходящее событие в документации и организовать его обработку.

На самом деле, решить эту задачу можно разными способами, мы продемонстрируем её решение, основанное на событии `onMouseOver`. При возникновении этого события мы будем выводить в консоль сообщение. Вот как будет теперь выглядеть наш код:

```
import React from "react"


function handleClick() {

    console.log("I was clicked")
```

```
}


function App() {

    return (

        <div>

            <img onMouseOver={() => console.log("Hovered!")}
src="https://www.fillmurray.com/200/100"/>

            <br />

            <br />

            <button onClick={handleClick}>Click me</button>

        </div>

    )

}


export default App
```

Обработка событий даёт в руки программиста огромные возможности, которые, конечно же, не ограничиваются выводом сообщений в консоль. В дальнейшем мы поговорим о том, как обработка событий, совмещённая с возможностями по изменению состояния компонентов, позволит нашим приложениям решать возлагаемые на них задачи.

Как обычно — рекомендуем уделить некоторое время на то, чтобы поэкспериментировать с тем, что вы сегодня узнали.

Учебный курс по React, часть 17: пятый этап работы над TODO-приложением, модификация состояния компонентов

Занятие 31. Практикум. TODO-приложение. Этап №5

Оригинал

Задание
Запуская наше Todo-приложение, вы могли заметить, что в консоль выводится уведомление, которое указывает на то, что мы, настроив свойство `checked` элемента в компоненте `TodoItem`, не предусмотрели механизм для взаимодействия с этим элементом в виде обработчика события `onChange`. При работе с интерфейсом приложения это выражается в том, что флажки, выводимые на странице, нельзя устанавливать и снимать.

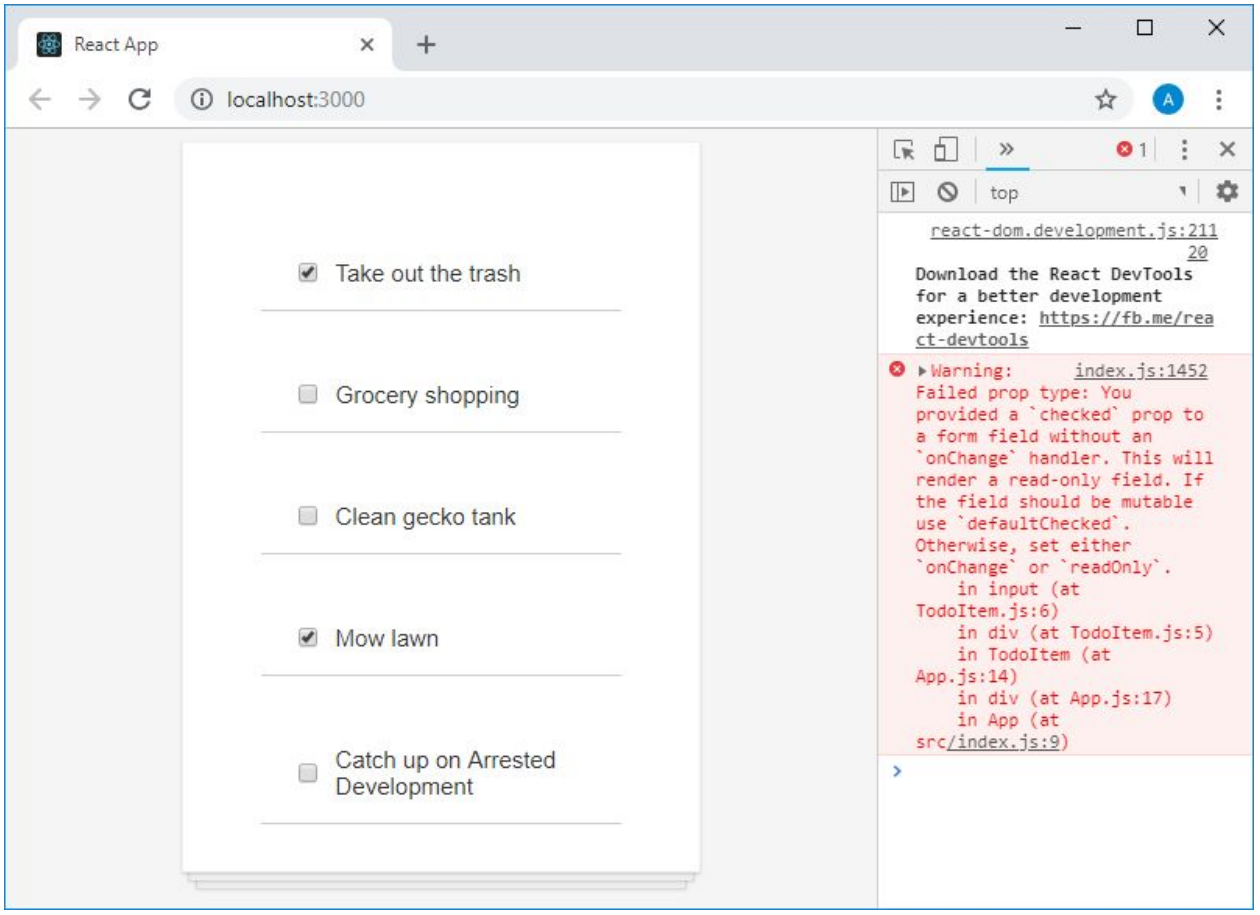
Здесь вам предлагается оснастить элемент типа `checkbox` компонента `TodoItem` обработчиком события `onChange`, который, на данном этапе работы, достаточно представить в виде функции, которая что-то выводит в консоль.

Решение

Вот как сейчас выглядит код компонента `TodoItem`, который хранится в файле `TodoItem.js`:

```
import React from "react"


function TodoItem(props) {
    return (
        <div className="todo-item">
            <input type="checkbox" checked={props.item.completed}/>
            <p>{props.item.text}</p>
        </div>
    )
}


export default TodoItem
```

Вот что выводится в консоль при запуске приложения.



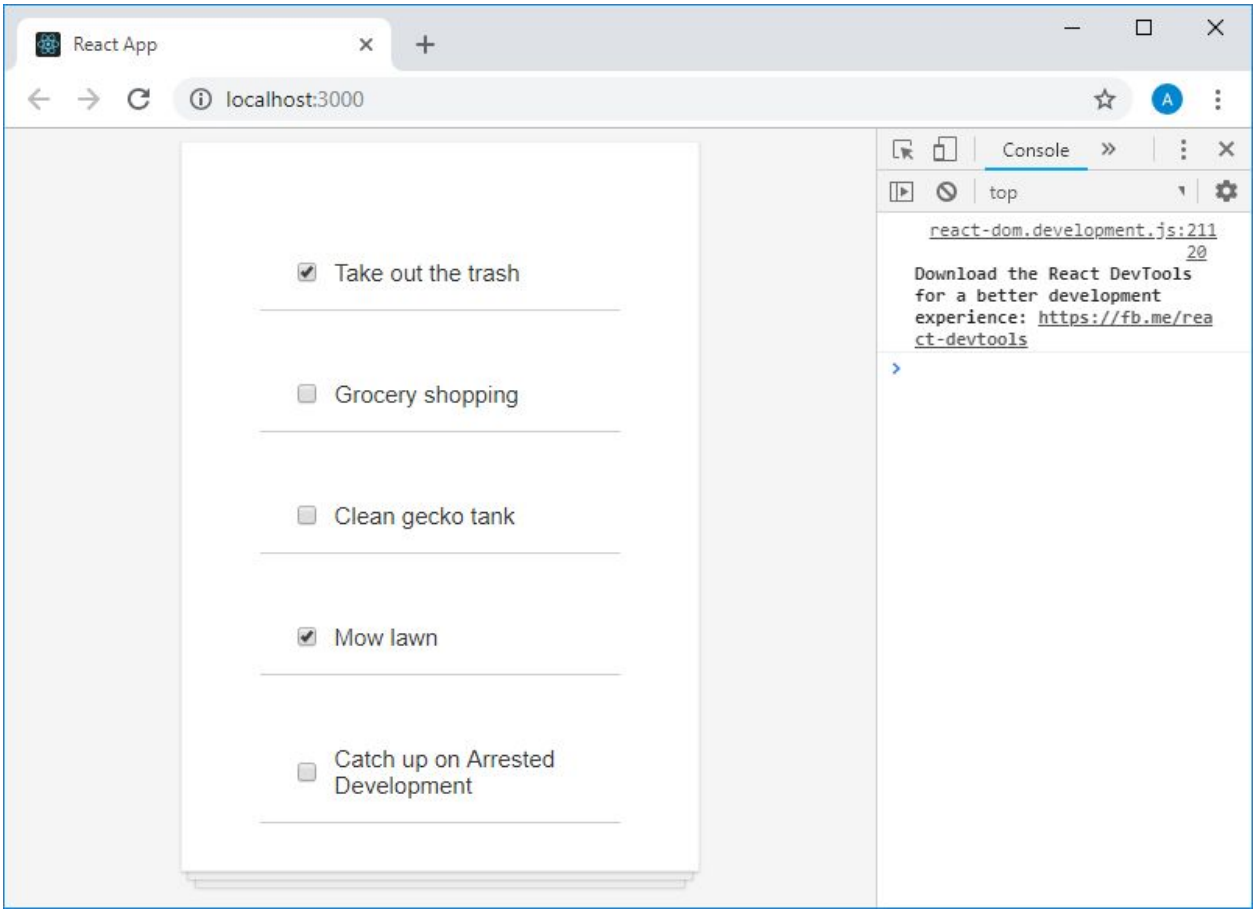
Уведомление в консоли

При этом флажки на наши воздействия не реагируют.

Для того чтобы избавиться от этого уведомления и подготовить проект к дальнейшей работе, достаточно назначить обработчик события `onChange` элементу `checkbox`. Вот как это выглядит в коде:

```
import React from "react"


function TodoItem(props) {
    return (
        <div className="todo-item">
            <input
                type="checkbox"
                checked={props.item.completed}
                onChange={() => console.log("Changed!")}
            />
            <p>{props.item.text}</p>
        </div>
    )
}


export default TodoItem
```

Здесь мы, в качестве обработчика, используем простую функцию, которая выводит в консоль слово `Checked!`. При этом щелчки по флажкам не приводят к изменению их состояния, но уведомление из консоли, как можно видеть на следующем рисунке, исчезает.



Флажки всё ещё не работают, но уведомление из консоли исчезло

Это небольшое изменение, внесённое в приложение, позволит нам, после того, как мы разберёмся с изменением состояния компонентов, сделать так, чтобы флажки работали правильно.

Занятие 32. Изменение состояния компонентов

Оригинал

Начнём работу со стандартного приложения, создаваемого с помощью `create-react-app`, в файле `App.js` которого содержится такой код:

```
import React from "react"


class App extends React.Component {

    constructor() {

        super()

        this.state = {

            count: 0

        }

    }
```

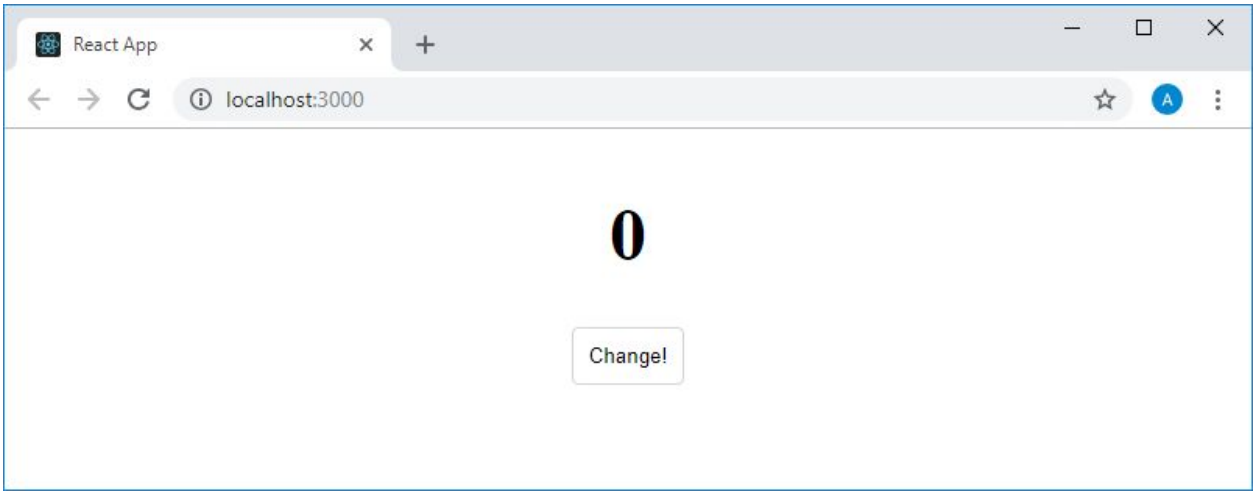
```
    render() {
        return (
            <div>
                <h1>{this.state.count}</h1>
                <button>Change!</button>
            </div>
        )
    }
}

export default App
```

В файле стилей `index.css`, который подключён в файле `index.js`, содержится следующее описание стилей:

```
div {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
}

h1 {
    font-size: 3em;
}

button {
    border: 1px solid lightgray;
    background-color: transparent;
    padding: 10px;
    border-radius: 4px;
}
```

```
button:hover {
    cursor: pointer;
}

button:focus {
    outline:0;
}
```

На данном этапе работы приложение выглядит так, как показано на следующем рисунке.



Страница приложения в браузере

Сегодня мы будем говорить о том, как менять состояние компонентов. Если у компонента есть состояние, это позволяет, инициализировав его, хранить в нём какие-то данные. Но если бы состояние нельзя было бы изменять, то от его наличия у компонента не было бы особенной пользы, хранение в нём данных не особенно сильно отличалось бы от, например, жёсткого задания их в коде компонента.

Поговорим о приложении, на примере которого мы будем рассматривать работу с состоянием компонента. Компонент `App`, код которого представлен выше, представляет собой компонент, основанный на классе. Это вполне очевидно, так как нам нужно, чтобы у этого компонента было бы состояние. В коде компонента мы используем конструктор. В нём мы, как всегда, вызываем метод `super()` и инициализируем состояние, записывая в него свойство `count` и присваивая ему начальное значение `0`. В методе `render()` мы выводим заголовок первого уровня, представляющий значение свойства `count` из состояния компонента, а также кнопку со словом `Change!`. Всё это отформатировано с помощью стилей.

Если, на данном этапе работы над приложением, открыть его в браузере и щёлкнуть по кнопке, то ничего, естественно, не произойдёт. Нам же нужно чтобы щелчок по кнопке менял бы состояние компонента, воздействуя на его свойство `count`. При этом мы уже изучили методику обработки событий в React, и наша задача сводится к тому, чтобы создать механизм, который, реагируя на щелчок по кнопке, меняет свойство состояния `count`.

Приступим к решению нашей задачи, оснастив кнопку обработчиком события `onClick`, который, для начала, будет просто выводить что-нибудь в консоль.

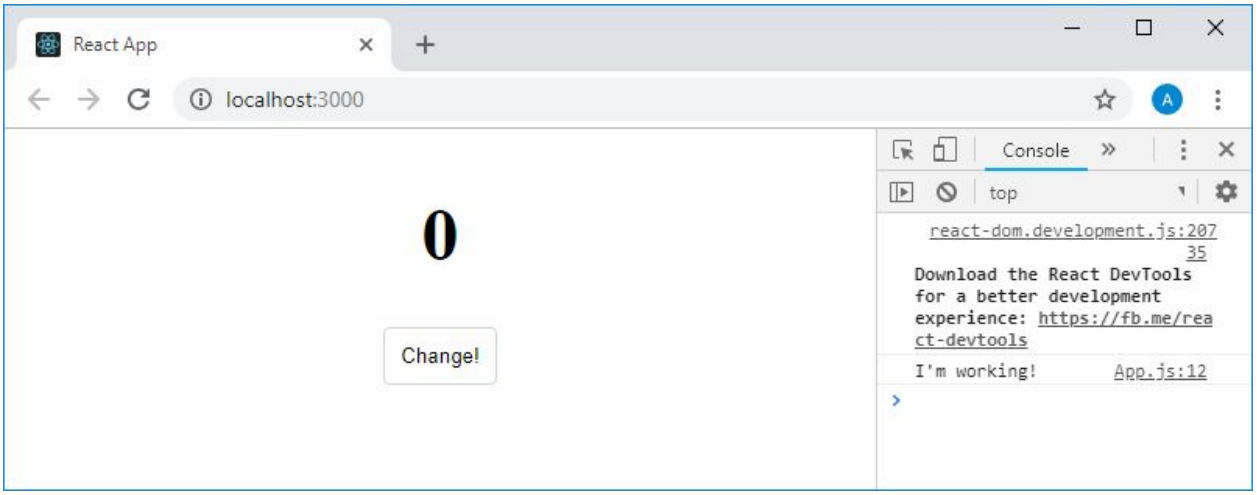
Для этого мы добавим в класс компонента новый метод. Назвать его можно как угодно, но подобные методы принято называть так, чтобы их имена указывали бы на обрабатываемые ими события. В

результате мы, так как мы собираемся с его помощью обрабатывать событие `click`, назовём его `handleClick()`. Вот как теперь будет выглядеть код компонента `App`.

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            count: 0
        }
    }

    handleClick() {
        console.log("I'm working!")
    }

    render() {
        return (
            <div>
                <h1>{this.state.count}</h1>
                <button onClick={this.handleClick}>Change!</button>
            </div>
        )
    }
}

export default App
```

Обратите внимание на то, что обращаясь к этому методу из `render()`, мы используем конструкцию вида `this.handleClick`.

Теперь, если щёлкнуть по кнопке, в консоль попадёт соответствующее сообщение.



Щелчок по кнопке вызывает метод класса

Сейчас давайте сделаем так, чтобы щелчок по кнопке увеличивал бы число, выводимое над ней, то есть, модифицировал бы состояние компонента. Может быть, попробовать поменять состояние компонента напрямую, в методе `handleClick()`? Скажем, что если переписать этот метод так:

```
handleClick() {

    this.state.count++

}
```

Нужно сразу сказать, что так с состоянием компонентов в React не работают. Попытка выполнения подобного кода вызовет ошибку.

Состояние компонента можно сравнить с одеждой, которую носит человек. Если он хочет переодеться, то он не перешивает и не перекрашивает одежду, не снимая с себя, а снимает её и надевает что-то другое. Собственно говоря, именно так работают и с состоянием компонентов.

Возможно, вы помните о том, что мы говорили о специальном методе, используемом для модификации состояния, доступном в компонентах, основанных на классах благодаря тому, что они расширяют класс `React.Component`. Это — метод `setState()`. Его используют в тех случаях, когда нужно изменить состояние компонента. Этим методом можно пользоваться по-разному.

Вспомним о том, что состояние представляет собой объект. Попробуем передать методу `setState()` объект, который заменит состояние. Перепишем метод `handleClick()` так:

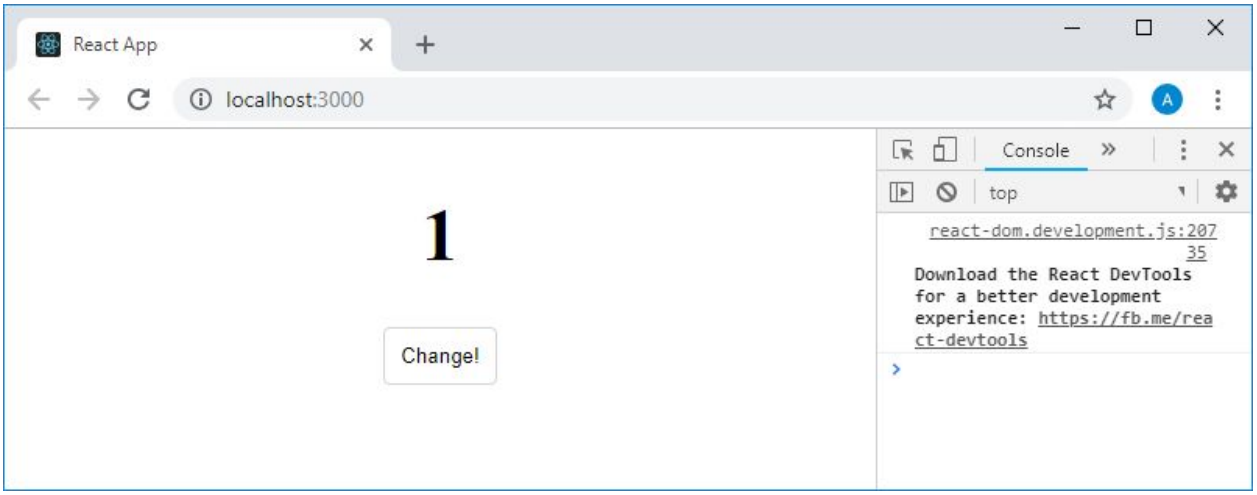
```
handleClick() {

    this.setState({ count: 1 })

}
```

Попытка воспользоваться таким методом вызовет такую ошибку: `TypeError: Cannot read property 'setState' of undefined`. На самом деле, то о чём мы сейчас говорим, вызывает множество споров в среде React-разработчиков, и сейчас я собираюсь показать вам очень простой способ решения этой проблемы, который, на первый взгляд, может показаться необычным.

Речь идёт о том, что каждый раз, создавая метод класса (`handleClick()` в нашем случае), в котором планируется использовать метод `setState()`, этот метод нужно связать с `this`. Делается это в конструкторе. Код компонента после этой модификации будет выглядеть так:

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            count: 0
        }
        this.handleClick = this.handleClick.bind(this)
    }

    handleClick() {
        this.setState({ count: 1 })
    }

    render() {
        return (
            <div>
                <h1>{this.state.count}</h1>
                <button onClick={this.handleClick}>Change!</button>
            </div>
        )
    }
}

export default App
```

Теперь после нажатия на кнопку `Change!` над ней появится число 1, сообщений об ошибках выводиться не будет.



Нажатие на кнопку модифицирует состояние

Правда, кнопка у нас получилась «одноразовой». После первого щелчка по ней `0` меняется на `1`, а если щёлкнуть по ней ещё раз — ничего уже не произойдёт. В общем-то, это и неудивительно. Код, вызываемый при щелчке по кнопке, делает своё дело, каждый раз меняя состояние на новое, правда, после первого же щелчка по кнопке новое состояние, в котором в свойстве `count` хранится число `1`, не будет отличаться от старого. Для того чтобы решить эту проблему, рассмотрим ещё один способ работы с методом `setState()`.

Если нас не интересует то, каким было предыдущее состояние компонента, то этому методу можно просто передать объект, который заменит состояние. Но часто бывает так, что новое состояние компонента зависит от старого. В нашем случае это означает, что мы, опираясь на значение свойства `count`, которое хранится в предыдущей версии состояния, хотим прибавить к этому значению `1`. В случаях, когда для изменения состояния нужно быть в курсе того, что в нём хранилось ранее, методу `setState()` можно передать функцию, которая, в качестве параметра, получает предыдущую версию состояния. Назвать этот параметр можно как угодно, в нашем случае это будет `prevState`. Заготовка этой функции будет выглядеть так:

```
handleClick() {

    this.setState(prevState => {



    })

}
```

Можно подумать, что в подобной функции достаточно просто обратиться к состоянию с помощью конструкции вида `this.state`, но такой подход нас не устроит. Поэтому важно, чтобы эта функция принимала бы предыдущую версию состояния компонента.

Функция должна возвращать новую версию состояния. Вот как будет выглядеть метод `handleClick()`, решающий эту задачу:

```
handleClick() {

    this.setState(prevState => {

        eturn {

            count: prevState.count + 1
```

```
        }
    })
}
```

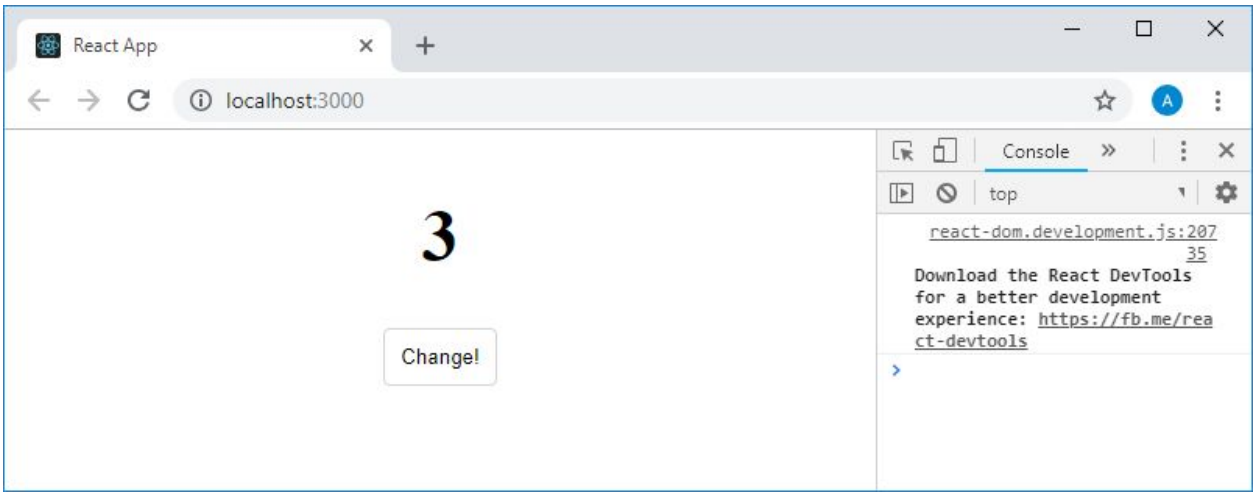
Обратите внимание на то, что для получения нового значения свойства `count` мы используем конструкцию `count: prevState.count + 1`. Можно подумать, что тут подойдёт и конструкция вида `count: prevState.count++`, но оператор `++` приводит к модификации переменной, к которой он применяется, это будет означать попытку модификации предыдущей версии состояния, поэтому им мы здесь не пользуемся.

Полный код файла компонента на данном этапе работы будет выглядеть так:

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            count: 0
        }
        this.handleClick = this.handleClick.bind(this)
    }

    handleClick() {
        this.setState(prevState => {
            return {
                count: prevState.count + 1
            }
        })
    }



    render() {
        return (
            <div>
```

```
                <h1>{this.state.count}</h1>
                <button onClick={this.handleClick}>Change!</button>
            </div>
        )
    }
}

export default App
```

Теперь каждый щелчок по кнопке увеличивает значение счётчика.



Каждый щелчок по кнопке увеличивает значение счётчика

То, с чем мы только что разобрались, открывает для нас огромные возможности в сфере разработки React-приложений.

Ранее мы говорили о том, что родительский компонент может, через механизм свойств, передавать свойства из собственного состояния дочерним компонентам. Если React обнаружит изменение состояния родительского компонента, он выполнит повторный рендеринг дочернего компонента, которому передаётся это состояние. Выглядит это как вызов метода `render()`. В результате дочерний компонент будет отражать новые данные, хранящиеся в состоянии родительского компонента.

Учебный курс по React, часть 18: шестой этап работы над TODO-приложением

Занятие 33. Практикум. TODO-приложение. Этап №6

Оригинал

Задание
На этом практическом занятии мы продолжим работу над Todo-приложением, сделаем так, чтобы действия пользователя влияли бы на состояние компонента. Речь идёт о том, чтобы у нас появилась возможность отмечать пункты списка дел как выполненные или невыполненные. Ниже приведён код

компонента `App`, некоторые заготовки и комментарии, имеющиеся в котором, призваны помочь вам в выполнении задания. Собственно говоря, вот что вам предлагается сегодня сделать:

1. Создайте в компоненте `App` обработчик события, который реагирует на изменения флажков (речь идёт о событии `onChange`) и соответствующим образом меняет состояние приложения. Пожалуй, это — самая сложная часть сегодняшнего задания. Для того чтобы с ней справиться — обратите внимание на комментарии и заготовки, представленные в коде.
2. Передайте соответствующий метод компоненту `TodoItem`.
3. В компоненте `TodoItem` создайте механизм, который, при возникновении события `onChange`, вызывает переданный экземпляру компонента метод и передаёт ему идентификатор (`id`) дела, которому соответствует флажок, по которому щёлкнул пользователь.

Вот код компонента `App`:

```
import React from "react"

import TodoItem from "./TodoItem"

import todosData from "./todosData"


class App extends React.Component {

    constructor() {

        super()

        this.state = {

            todos: todosData

        }

        this.handleChange = this.handleChange.bind(this)

    }


    handleChange(id) {

        // Обновите состояние так, чтобы у элемента с заданным id свойство

        // completed поменялось бы с false на true (или наоборот).

        // Помните о том, что предыдущую версию состоянию менять не следует.

        // Вместо этого нужно вернуть новую версию состояния, содержащую
изменения.

        // (Подумайте о том, как для этого использовать метод массивов map.)

    }


    render() {
```

```
        const todoItems = this.state.todos.map(item => <TodoItem key={item.id}
item={item}/>)


        return (
            <div className="todo-list">
                {todoItems}
            </div>
        )
    }
}


export default App
```

Решение

Для начала создадим простой механизм проверки вызова метода `handleChange()`. А именно — приведём его к такому виду:

```
handleChange(id) {
    console.log("Changed", id)
}
```

Теперь мы реализуем то, что нужно сделать в соответствии с пунктами 2 и 3 задания. То есть — создадим связь между щелчком по флажку и вызовом метода `handleChange()` с передачей ему `id` этого флажка.

Для того чтобы передать экземпляру компонента `TodoItem` ссылку на `handleChange()`, мы можем поступить так же, как поступали передавая ему свойства и переписать код создания списка компонентов так:

```
const todoItems = this.state.todos.map(item => <TodoItem key={item.id}
item={item} handleChange={this.handleChange}/>)
```

Обратите внимание на то, что свойство `handleChange`, которое будет доступно компонентам `TodoItem`, содержит ссылку на метод `handleChange` экземпляра компонента `App`. В компоненте `TodoItem` к этому методу можно обратиться так же, как и к другим передаваемым ему свойствам. На данном этапе работы код `TodoItem` выглядит так:

```
import React from "react"


function TodoItem(props) {
    return (
        <div className="todo-item">
```

```
            <input
                type="checkbox"
                checked={props.item.completed}
                onChange={() => console.log("Changed!")}
            />
            <p>{props.item.text}</p>
        </div>
    )
}

export default TodoItem
```

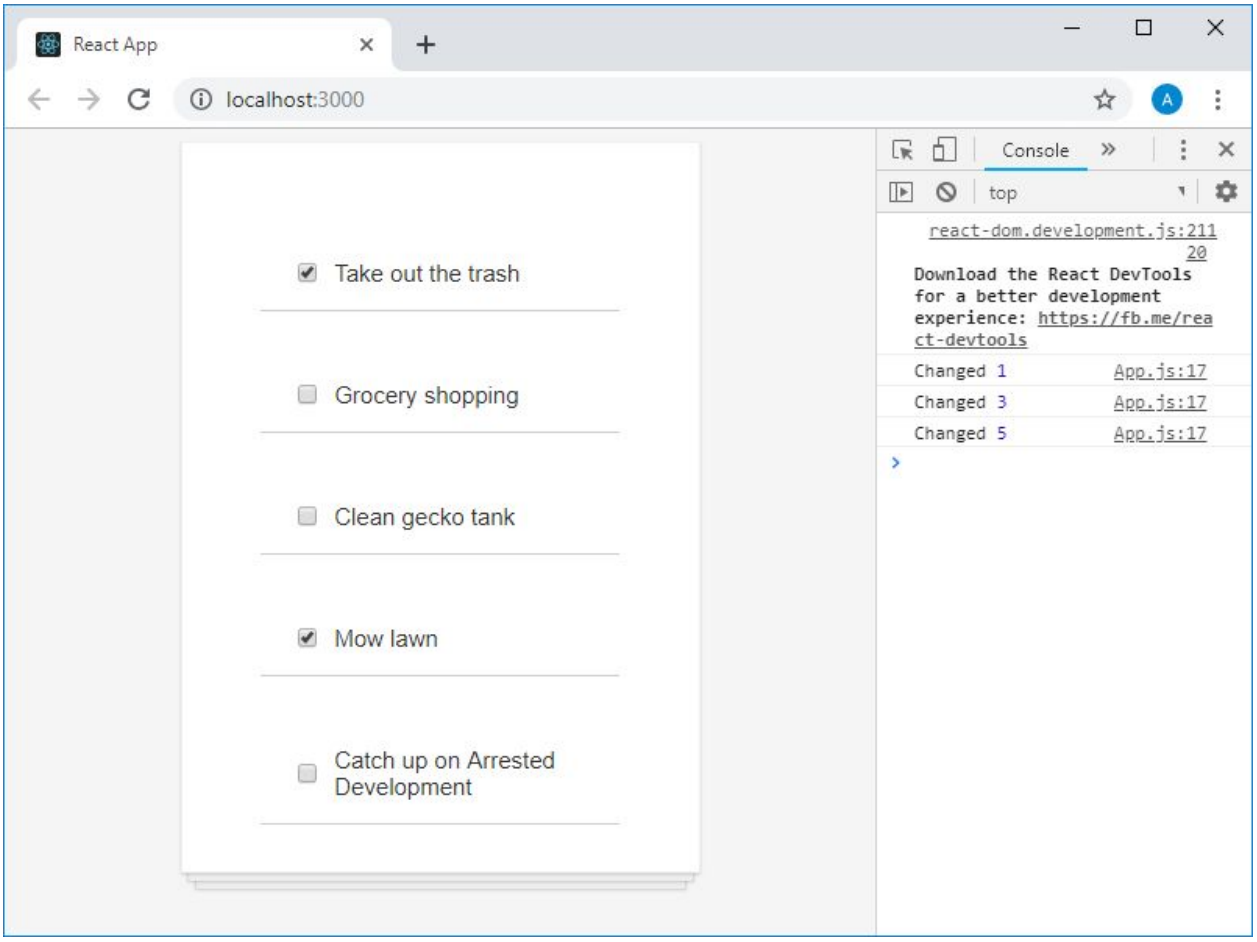
Для вызова метода `handleChange` в коде компонента можно воспользоваться конструкцией вида `props.handleChange()`. При этом данному методу нужно передать `id` элемента. Обработчик события `onChange` принимает объект события. Нам, для вызова метода `handleChange()`, этот объект не требуется. Перепишем код, в котором мы назначаем элементу обработчик события `onChange`, так:

```
onChange={(event) => props.handleChange(props.item.id)}
```

Здесь мы вызываем метод `handleChange()`, передавая ему идентификатор элемента, взятый из переданных ему свойств, из другой функции, которая принимает объект события. Так как мы этот объект здесь не используем, код можно переписать так:

```
onChange={() => props.handleChange(props.item.id)}
```

Теперь попробуем запустить приложение и, открыв консоль, пощёлкать по флажкам.



Проверка метода handleChange()

В консоль попадают сообщения, содержащие идентификаторы флажков, по которым мы щёлкаем. Но флажки пока не меняют внешний вид, так как в методе `handleChange()` ещё не реализован механизм изменения состояния компонента. В результате мы только что справились со второй и третьей частями задания и теперь можем приступить к работе над его первой частью, пожалуй, самой интересной из всех, касающейся работы с состоянием.

Для начала нам нужно решить вопрос, касающийся того, что в состоянии хранится массив, некий элемент которого, в ответ на щелчок по флажку, должен претерпеть изменения, но мы при этом не должны модифицировать массив, хранящийся в старой версии состояния. То есть, например, нельзя просто пройтись по уже имеющемуся в состоянии массиву объектов, найти нужный элемент и изменить его свойство `completed`. Нам нужно, чтобы, после изменения состояния, был бы сформирован новый массив, один из элементов которого будет изменён, а остальные останутся такими же, какими были раньше. Одним из подходов к формированию такого массива является использование метода массивов `map()`, упомянутого в комментариях к заданию. Код мы будем писать в методе `setState()`. Приведём код метода `handleChange()` из компонента `App` к следующему виду:

```
handleChange(id) {

    this.setState(prevState => {



    })

}
```

Теперь, с помощью метода `map()`, пройдёмся по массиву `prevState.todos` и поищем в нём нужный нам элемент, то есть тот, `id` которого передано методу `handleChange()`, после чего изменим его

свойство `completed`. Метод `map()` возвращает новый массив, который и будет использоваться в новом состоянии приложения, поэтому мы запишем этот массив в константу. Вот как это выглядит:

```
handleChange(id) {
    this.setState(prevState => {
        const updatedTodos = prevState.todos.map(todo => {
            if (todo.id === id) {
                todo.completed = !todo.completed
            }
            return todo
        })
        return {
            todos: updatedTodos
        }
    })
}
```

Здесь, в ходе обработки массива с помощью `map()`, если обнаруживается элемент, `id` которого равен `id`, переданному методу `handleChange()`, значение свойства `completed` этого элемента меняется на противоположное (`true` на `false` и наоборот). После этого, независимо от того, был ли изменён элемент, `map()` возвращает этот элемент. Он попадает в новый массив (представленный здесь константой `updatedTodos`) под тем же индексом, под которым соответствующий элемент был в массиве `todos` из предыдущей версии состояния. После того, как будет просмотрен весь массив и будет полностью сформирован массив `updatedTodos`, этот массив используется в качестве значения свойства `todos` объекта, возвращаемого методом `setState()`, который представляет собой новую версию состояния.

Если запустить приложение теперь, то можно обнаружить, что флажки реагируют на наши воздействия. Это говорит о том, что щелчки по ним меняют состояние приложения, после чего производится их повторный рендеринг с использованием новых данных.

Учебный курс по React, часть 19: методы жизненного цикла компонентов

Занятие 34. Методы жизненного цикла компонентов, часть 1

Оригинал

Одной из особенностей разработки React-приложений является тот факт, что мы пишем довольно простой JavaScript-код, который приводит в действие внутренние механизмы React и тем самым даёт нам замечательные возможности по разработке интерфейсов приложений и по работе с данными. При этом компоненты, которыми мы пользуемся, в течение своего жизненного цикла, проходят через определённые этапы. Часто то, что происходит с компонентом в приложении, сравнивают с жизнью человека. Люди рождаются, живут, в их жизни случаются некие значимые события, после чего они умирают. Компоненты React в этом похожи на людей, так как они тоже «рождаются», «живут» и

«умирают». Работая с компонентами, мы можем реагировать на то, что с ними происходит, благодаря методам их жизненного цикла, которые вызываются в особенные моменты их «жизни».

Недавно команда разработчиков React признала устаревшими три метода жизненного цикла компонентов. Мы эти методы, всё же, рассмотрим, так как ими всё ещё можно пользоваться, и так как они могут встретиться вам в существующем коде. Кроме того, в React были добавлены два новых метода жизненного цикла компонентов, о которых мы поговорим на следующем занятии.

Мы рассмотрим только самые важные методы, которые наиболее актуальны для тех, кто только приступил к изучению React. Когда же вы продолжите осваивать эту библиотеку, вы сможете поэкспериментировать и с другими методами.

Вот хороший материал, посвящённый методам жизненного цикла компонентов React, которые были актуальны до выхода версии React 16.3. Здесь, в публикации из официального блога разработчиков React, можно узнать об изменениях, которые произошли в React 16.3.

Теперь начнём разговор о методах жизненного цикла компонентов React, с которыми вам придётся встречаться чаще всего.

Мы, как всегда, будем пользоваться здесь демонстрационным проектом. В данном случае мы начинаем со стандартного проекта, созданного средствами create-react-app, в файле `App.js` которого содержится следующий код:

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


class App extends Component {
    constructor() {
        super()
        this.state = {}
    }

    render() {
        return (
            <div>
                Code goes here
            </div>
        )
    }
}

export default App
```

Для начала давайте взглянем на метод, которым вы, работая с компонентами, основанными на классах, уже пользовались. Это — метод `render()`. Часто его, говоря о методах жизненного цикла компонентов, не упоминают. Думаю, что этот метод, если сравнивать компонент с человеком, можно сравнить с одеванием перед выходом на улицу. Задачей этого метода является определение того, что будет выведено на экран, то есть того, как будет выглядеть компонент. Метод `render()` в процессе жизни компонента может быть вызван множество раз. Так, когда React определяет, что что-то, относящееся к компоненту, изменилось, наподобие состояния или свойств, то есть что-то такое, что может повлиять на внешний вид компонента, React может вызвать этот метод. Это можно сравнить, если продолжить аналогию с людьми, с тем, что человек может решить переодеться. Например для того, чтобы, после рабочего дня, подготовиться к некоему праздничному мероприятию.

Теперь рассмотрим ещё один метод жизненного цикла компонентов — `componentDidMount()`. Этот метод объявляют, точно так же, как и любые другие методы компонентов, основанных на классах, в теле класса компонента:

```
componentDidMount() {

}
```

Этот метод вызывается в момент жизненного цикла компонента, который можно сравнить с «рождением» этого компонента. Этот метод срабатывает один раз после того, как компонент будет смонтирован (вставлен) в дерево DOM. При этом, например, если после изменения неких данных, влияющих на внешний вид компонента, будет выполнен его повторный рендеринг, метод `componentDidMount()` вызван не будет. Происходит это из-за того, что при выполнении подобных операций не производится изъятие компонента из дерева DOM и его последующее повторное включение в состав дерева.

Метод `componentDidMount()` обычно используют для выполнения обращений к неким API, в случаях, когда разработчику нужны данные из внешних источников. Предположим, компонент, который мы тут рассматриваем, на самом деле называется `TodoList` и представляет собой компонент, формирующий список дел в Todo-приложении. Метод `componentDidMount()` такого компонента может выполнять загрузку материалов из серверной базы данных, необходимых для корректного вывода на страницу хранящегося на сервере списка дел. В результате после того, как монтирование компонента завершено, мы, в методе `componentDidMount()`, можем загрузить данные, необходимые для правильного отображения компонента на странице. Мы ещё поговорим о загрузке данных, необходимых компонентам, а пока же можете запомнить, что это — наиболее часто встречающийся вариант использования метода `componentDidMount()`.

Следующий метод жизненного цикла компонентов, который мы обсудим, называется `componentWillReceiveProps()`. Этот метод можно сравнить с тем, что происходит, когда кто-то получает от кого-то подарок. Так, компонент может получать свойства от родительского компонента. Каждый раз, когда компонент принимает свойства, вызывается этот метод. При этом данный метод вызывается каждый раз, когда родительский компонент передаёт свойства дочернему компоненту, а не только тогда, когда это случается в первый раз. Например, если родительский компонент решит поменять свойства, переданные дочернему компоненту, то мы, в методе `componentWillReceiveProps()`, сможем, например, проверить, отличаются ли новые свойства от тех, что уже были переданы компоненту. Дело в том, что если новые свойства не отличаются от старых, это значит, что их поступление ничего не меняет, а значит — мы можем, выяснив это, больше ничего не делать. Если же новые свойства отличаются от старых, мы можем выполнить некие действия. Обычно этот метод объявляют в теле класса компонента в таком виде:

```
componentWillReceiveProps(nextProps) {

}
```

Тут обычно используется, в качестве имени параметра, `nextProps`, но назвать этот параметр можно как угодно. Для того чтобы сравнить некое конкретное свойство, которое уже было передано компоненту, с тем, что уже было ему передано ранее, и принять решение о дальнейших действиях, можно воспользоваться такой конструкцией:

```
componentWillReceiveProps(nextProps) {

    if (nextProps.whatever !== this.props.whatever) {

        // сделать тут что-то важное

    }

}
```

Обычно этот метод используют именно так.

Однако, как уже было сказано, после выхода React 16.3 некоторые методы жизненного цикла компонентов были признаны устаревшими, и `componentWillReceiveProps()` — это один из таких методов.

До выхода React 17 этими устаревшими методами всё ещё можно пользоваться, хотя лучше этого не делать. Если без рассматриваемого метода никак не обойтись — его нужно назвать `UNSAFE_componentWillReceiveProps()`. После выхода React 17 имя метода `componentWillReceiveProps()` не будет означать ничего особенного.

Об этом методе полезно знать для того, чтобы иметь возможность понимать устаревший код, но при разработке современных React-приложений он уже использоваться не будет.

Ещё один интересный метод жизненного цикла компонентов называется `shouldComponentUpdate()`. Он, если продолжить сравнение компонента с человеком, напоминает момент, когда человек размышляет о том, надо ли ему переодеться или нет. В обычных условиях, если React не вполне уверен в том, надо ли повторно отрендерить компонент, он его, на всякий случай, всё же отрендерит. При этом неважно — нужно ли это, в соответствии с логикой приложения, или нет. Это приводит к тому, что React повторно рендерит компоненты даже в тех случаях, когда ничего, имеющего отношение к компоненту, не меняется. Подобное может привести к замедлению приложения, так как по такому принципу React обрабатывает все компоненты, входящие в состав приложения. Метод `shouldComponentUpdate()` даёт разработчику возможность оптимизировать приложение. Здесь можно реализовать некую логику, помогающую выяснить необходимость обновления компонента. Этот метод обычно объявляют так:

```
shouldComponentUpdate(nextProps, nextState) {

    // вернуть true если компонент нуждается в обновлении

    // вернуть false в противном случае

}
```

При этом из этого метода, если компонент нуждается в повторном рендеринге, с учётом новых свойств и состояния, нужно вернуть `true`. В противном случае из него нужно вернуть `false`. Собственно говоря, возврат `false` из этого метода приводит к тому, что обновление компонента не выполняется и приложение работает быстрее, но, делая это, нужно быть уверенным в том, что компоненту действительно не требуется повторный рендеринг. Если же компоненту нужно обновиться, а этот метод вернул `false` — это приведёт к появлению ошибок, с которыми будет сложно бороться.

Ещё один метод жизненного цикла компонентов, о котором мы поговорим, называется `componentWillUnmount()`. Этот метод знаменует собой окончание «жизни» компонента — тот момент, когда он удаляется из дерева DOM и исчезает с экрана. Этот метод, в основном, используется для того, чтобы освобождать ресурсы, занятые компонентом и навести перед его удалением порядок. Например, если в методе `componentDidMount()` было настроено нечто вроде прослушивателя событий, благодаря которому, когда пользователь прокручивает страницу, выполняется некий код, именно в `componentWillUnmount()` можно удалить такой прослушиватель событий. На самом же деле, у этого метода есть множество вариантов применения, которые направлены на то, чтобы убрать из приложения всё, что окажется ненужным после исчезновения компонента.

Вот полный код нашего компонента `App`, в который добавлены методы жизненного цикла:

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


class App extends Component {
    constructor() {
        super()
        this.state = {}
    }


    componentDidMount() {
        // загрузить данные, необходимые для корректного отображения компонента
    }


    componentWillReceiveProps(nextProps) {
        if (nextProps.whatever !== this.props.whatever) {
            // сделать тут что-то важное
        }
    }


    shouldComponentUpdate(nextProps, nextState) {
        // вернуть true если компонент нуждается в обновлении
        // вернуть false в противном случае
    }


    componentWillUnmount() {
```

```
        // навести порядок после удаления компонента
        // (например - убрать прослушиватели событий)
    }

    render() {
        return (
            <div>
                Code goes here
            </div>
        )
    }
}

export default App
```

На этом мы завершаем данное занятие, хотя надо отметить, что методы жизненного цикла компонентов React не ограничиваются теми, что мы сегодня рассмотрели.

Занятие 35. Методы жизненного цикла компонентов, часть 2

Оригинал

Как уже было сказано на предыдущем занятии, когда вышел React 16.3, было сообщено о том, что три метода жизненного цикла компонентов устарели. Это методы `componentWillMount()`, `componentWillReceiveProps()` и `componentWillUpdate()`. Также было сообщено о появлении двух новых методов. Это — статический метод `getDerivedStateFromProps()` и метод `getSnapshotBeforeUpdate()`. Нельзя сказать, что эти методы сыграют важную роль на будущих занятиях этого курса, но мы, несмотря на это, здесь с ними ознакомимся.

Экспериментировать будем в том же проекте, который мы использовали в прошлый раз.

Вот как выглядит объявление метода `getDerivedStateFromProps()`:

```
static getDerivedStateFromProps(props, state) {

}
```

Обратите внимание на ключевое слово `static` перед именем метода. Он, на основании принятых им свойств, должен возвратить обновлённое состояние. Он используется в тех случаях, когда некий компонент должен принимать входящие свойства, получаемые им от компонента-родителя, и настраивать своё состояние, основываясь на этих свойствах. Подробности об этом методе можно почитать здесь. В этом материале, опубликованном в блоге React, речь идёт о том, что применение этого метода оправдано далеко не во всех тех ситуациях, в которых он, как кажется, может пригодиться. Его неправильное использование может приводить к различным ошибкам, к падению производительности приложений, поэтому пользоваться им следует с осторожностью. Не следует

пытаться решать с его помощью задачи, для решения которых он не предназначен. Вот документация к этому методу.

Теперь поговорим о методе `getSnapshotBeforeUpdate()`. Вот как выглядит его объявление в теле класса:

```
getSnapshotBeforeUpdate() {

}
```

Его можно рассматривать как метод жизненного цикла, который позволяет создавать нечто вроде резервной копии того, что имеется в компоненте перед его обновлением. Она напоминает мгновенный снимок состояния приложения. При этом нужно отметить, что разработчики React говорят о том, что сфера применения этого метода ограничена. Вот документация по нему.

Учебный курс по React, часть 20: первое занятие по условному рендерингу

Занятие 36. Условный рендеринг, часть 1

Оригинал

Технологии условного рендеринга используются в тех случаях, когда что-то нужно вывести на страницу в соответствии с неким условием. На этом занятии мы поговорим о том, как вывести особое сообщение (оно вполне может быть представлено чем-то вроде загрузочного экрана) в то время, когда приложение готовится к работе, загружая данные, и, после того, как оно будет готово, как заменить это сообщение на что-то другое.

Экспериментировать сегодня мы будем с приложением, созданным средствами create-react-app, в файле `App.js` которого содержится следующий код:

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

import Conditional from "./Conditional"


class App extends Component {

    constructor() {

        super()

        this.state = {

            isLoading: true

        }

    }


    componentDidMount() {

        setTimeout(() => {

            this.setState({
```

```
                isLoading: false
            })
        }, 1500)
    }

    render() {
        return (
            <div>
                <Conditional isLoading={this.state.isLoading}/>
            </div>
        )
    }
}

export default App
```

Кроме того, в той же папке, где находится файл `App.js`, есть файл компонента `Conditional.js` со следующим содержимым:

```
import React from "react"

function Conditional(props) {
    return (

    )
}

export default Conditional
```

На данном этапе работы это приложение пока работать не будет, в процессе разбора материала мы это исправим.

Одна из сложностей, встающих перед теми, кто осваивает React, заключается в том, что им приходится изучать множество инструментов, которые можно использовать различными способами. Программист не обязательно ограничен лишь одним способом использования некоего средства. Определённое влияние на это оказывает тот факт, что React-разработка чрезвычайно близка к разработке на обычном JavaScript. Поэтому у нас есть возможности использования разных подходов к решению одинаковых задач, поэтому одни и те же инструменты могут использоваться по-разному. Условный рендеринг —

это та область React, в которой вышеозвученные идеи проявляют себя особенно сильно. Собственно говоря, прежде чем мы начнём, хотелось бы отметить, что, хотя мы разберём несколько подходов к применению этой технологии, ими реальные варианты её использования не ограничиваются.

Поговорим о коде, с которым мы сейчас будем экспериментировать. У нас, в файле `App.js`, имеется компонент, основанный на классе. В его конструкторе инициализировано состояние, содержащее свойство `isLoading`, установленное в значение `true`. Подобная конструкция часто применяется в случаях, когда для приведения компонента в рабочее состояние нужно, например, выполнять запросы к некоему API, и, пока компонент ожидает поступления данных и разбирает их, нужно что-то показать на экране. Возможно, на выполнение обращения к API требуется 3-4 секунды, и вы не хотите, чтобы пользователь, глядя на экран, думал бы, что ваше приложение дало сбой. В результате в состоянии есть свойство, которое указывает на то, выполняет ли в настоящий момент приложение некие служебные действия. И условный рендеринг будет использоваться для вывода на экран чего-то, что сообщает пользователю о том, что приложение в настоящий момент что-то загружает в фоне.

В коде компонента `App` есть метод `componentDidMount()`, который мы уже совсем скоро обсудим. Пока же обратим внимание на метод `render()`. Здесь мы выводим компонент `Condition`, который импортирован в коде, находящемся в верхней части файла `App.js`. Этому компоненту передаётся свойство `isLoading`, представляющее собой текущее значение свойства `isLoading` из состояния компонента `App`. Код компонента `Conditional` пока не возвращает ничего такого, что можно вывести на экран, этим компонентом мы займёмся немного позже. Пока же давайте вернёмся к методу `componentDidMount()` из кода компонента `App`.

Вспомните о том, что метод `componentDidMount()` даёт нам возможность выполнять некий код сразу после того, как компонент, в нашем случае это компонент `App`, впервые будет выведен на экран. В коде этого метода мы имитируем обращение к некоему API. Здесь мы устанавливаем таймер на полторы секунды. Когда это время пройдёт — будет запущен код функции, переданной функции `setTimeout()`. В этой функции, исходя из предположения о том, что её вызов символизирует окончание загрузки данных из API, выполняется изменение состояния. А именно, его свойство `isLoading` устанавливается в значение `false`. Это говорит о том, что загрузка данных завершена, и приложение после этого может нормально работать. На будущих занятиях мы поговорим об использовании функции `fetch()` для загрузки данных, пока же ограничимся вышеописанной имитацией этого процесса.

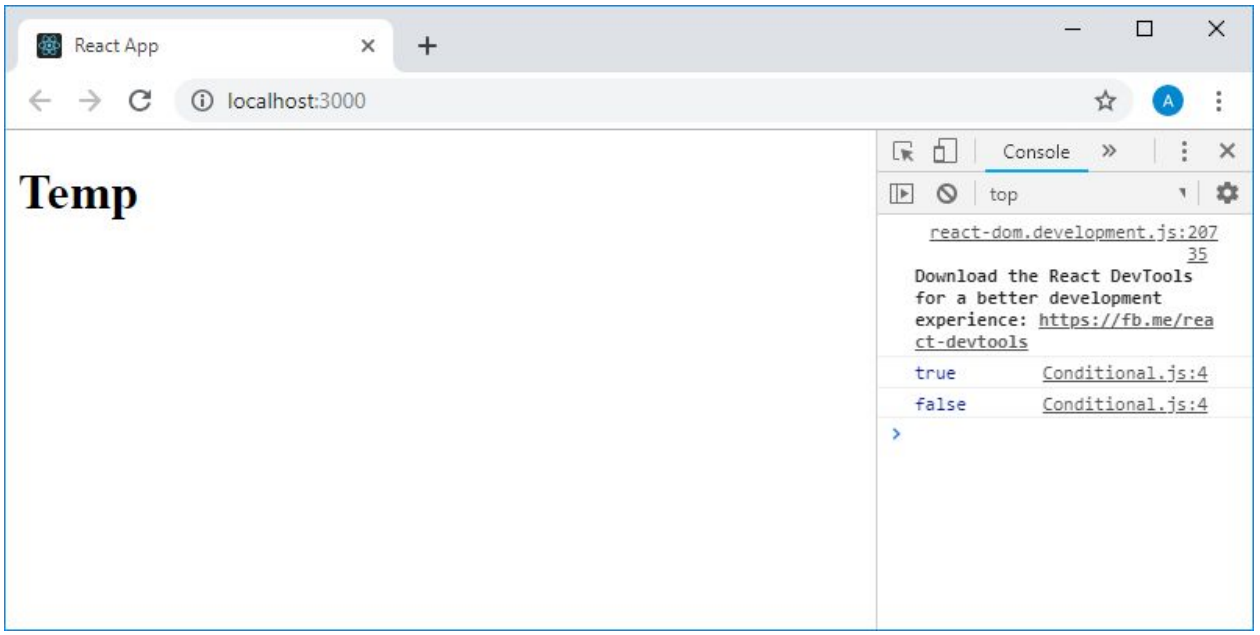
Кстати, тут будет уместно ещё раз поднять тему методов жизненного цикла компонента. Дело в том, что как только свойство состояния `isLoading` меняется с `true` на `false`, компонент `Conditional` получает новое значение свойства. Сначала, при первом выводе компонента на экран, он получает, в свойстве `isLoading`, значение `true`, а затем, после того, как состояние меняется, он получает то же свойство с новым значением. Собственно говоря, при изменении состояния повторно вызывается метод `render()`, в результате компонент `Conditional` также будет повторно выведен на экран. Напомним о том, что `Conditional` — это обычный функциональный компонент, то есть его повторный рендеринг означает повторный вызов функции, которой он представлен. Но то, что мы возвращаем из этой функции при повторном рендеринге компонента, может отличаться от того, что возвращалось ранее. Причиной такого изменения является изменение того, что мы передаём компоненту.

Итак, компонент `Conditional` принимает свойство `isLoading`. Прежде чем мы приступим к работе над кодом, проверим, работают ли те механизмы, которые в нём уже имеются. Для этого мы вернём из компонента некую разметку и выведем в консоль значение `props.isLoading`. После этого код компонента будет выглядеть так:

```
import React from "react"
```

```
function Conditional(props) {
    console.log(props.isLoading)
    return (
        <h1>Temp</h1>
    )
}

export default Conditional
```

Страница приложения после этого будет выглядеть так, как показано на следующем рисунке.



Страница приложения в браузере

Обратите внимание на то, что `true` выводится в консоль сразу же после загрузки приложения, а `false` — через 1.5 секунды. Это происходит благодаря работе вышеописанного механизма в методе `componentDidMount()` компонента `App`.

Теперь поговорим об условном рендеринге. Его суть заключается в том, что мы выводим что-то на экран только в том случае, если выполняется некое условие. В данном случае, вместо вывода на страницу строки `Temp`, мы, в компоненте `Conditional`, можем проверить значение `props.isLoading`, и, если оно равно `true`, вывести на страницу текст `Loading...`. Если же это значение равно `false`, что символизирует окончание загрузки, из компонента можно вернуть какой-нибудь другой текст. В коде это будет выглядеть так:

```
import React from "react"

function Conditional(props) {
    if(props.isLoading === true) {
```

```
        return (
            <h1>Loading...</h1>
        )
    } else {
        return (
            <h1>Some cool stuff about conditional rendering</h1>
        )
    }

}

export default Conditional
```

Попробуйте запустить у себя этот код, обновите страницу и понаблюдайте за тем, как, при загрузке страницы, выводится один текст, а через некоторое время — другой.

Учитывая особенности JavaScript, мы можем упростить вышеприведённый код так:

```
import React from "react"

function Conditional(props) {
    if(props.isLoading === true) {
        return (
            <h1>Loading...</h1>
        )
    }
    return (
        <h1>Some cool stuff about conditional rendering</h1>
    )
}

export default Conditional
```

Если условие, проверяемое в блоке `if`, является истинным, то сработает выражение `return`, находящееся в этом блоке, после чего выполнение функции завершится. Если же условие является

ложным, то выражение `return` из этого блока не выполняется и осуществляется возврат из функции того, что задано во втором выражении `return`.

Сейчас давайте поговорим о том, как можно решать задачи условного рендеринга с использованием тернарного оператора. Эта конструкция существует в JavaScript уже очень давно. Её часто используют в React для решения задач условного рендеринга. Вот как она выглядит:

```
условие ? выражение1 : выражение2
```

Значение выражения 1 возвращается в том случае, если условие истинно, значение выражения 2 — в том случае, если условие ложно.

В нашем случае с использованием тернарного оператора код компонента `Conditional` можно переписать так:

```
import React from "react"


function Conditional(props) {

    return (

        props.isLoading === true ? <h1>Loading...</h1> : <h1>Some cool stuff
about conditional rendering</h1>

    )


}


export default Conditional
```

Такая конструкция, хотя и работает, выглядит непривычно. Дело в том, что обычно компоненты возвращают более сложные конструкции. Поэтому обернём всё это в элемент `<div>`:

```
import React from "react"


function Conditional(props) {

    return (

        <div>

            props.isLoading === true ? <h1>Loadinag...</h1> : <h1>вSome cool
stuff about conditional rendering</h1>

        </div>

    )


}
```

```
export default Conditional
```

Такой код тоже работает, правда уже не так, как нужно. На страницу попадает всё то, что заключено в элемент `<div>`. Для того, чтобы это исправить, вспомним о том, что JS-конструкции, используемые в разметке, возвращаемой из компонентов, нужно заключать в фигурные скобки и соответствующим образом перепишем код:

```
import React from "react"


function Conditional(props) {
    return (
        <div>

            {props.isLoading === true ? <h1>Loading...</h1> : <h1>Some cool
stuff about conditional rendering</h1>}
        </div>
    )

}

export default Conditional
```

Теперь всё снова работает так, как надо.

Надо отметить, что в реальном компоненте разметка, возвращаемая им, выглядела бы сложнее. Тут, например, в верхней части того, что выводит компонент, может присутствовать некая навигационная панель, в нижней части может быть предусмотрен «подвал» страницы, и так далее. Выглядеть это может так:

```
import React from "react"


function Conditional(props) {
    return (
        <div>
            <h1>Navbar</h1>

            {props.isLoading === true ? <h1>Loading...</h1> : <h1>Some cool
stuff about conditional rendering</h1>}
```

```
            <h1>Footer</h1>
        </div>
    )
}

export default Conditional
```

При этом наличие в разметке, возвращаемой компонентом, дополнительных элементов, не мешает механизмам условного рендеринга. Кроме того, эти элементы будут выводиться и тогда, когда `props.isLoading` равно `true`, и тогда, когда это свойство равно `false`.

Ещё одно улучшение, которое можно внести в этот код, основано на том, что, так как `props.isLoading` — это логическое свойство, принимающее значение `true` или `false`, его можно использовать непосредственно, без применения оператора строгого сравнения его с `true`. В результате получается следующее:

```
import React from "react"

function Conditional(props) {
    return (
        <div>
            <h1>Navbar</h1>

            {props.isLoading ? <h1>Loading...</h1> : <h1>Some cool stuff about
conditional rendering</h1>}

            <h1>Footer</h1>
        </div>
    )
}

export default Conditional
```

Теперь мы вышли на работающий пример использования технологии условного рендеринга, но тех же результатов можно добиться множеством способов. Например, обычно в компонентах, подобных

нашему, не выводятся навигационные панели и «подвалы» страниц. Такие элементы страниц обычно выводятся либо самим компонентом `App`, либо специальными компонентами, выводимыми компонентом `App`.

Кроме того, надо отметить, что здесь вся логика условного рендеринга расположена внутри метода `render()` функционального компонента, что сделано лишь для того, чтобы продемонстрировать компактный код, собранный в одном месте. Но, вероятно, ответственным за условный рендеринг стоило бы сделать компонент `App`, а компонент, подобный нашему компоненту `Conditional`, должен просто выводить на экран то, что ему передано. Если компонент `App` ответственен за выяснение того, выполняется ли загрузка чего-либо в некий момент времени, и того, когда эта операция завершится, тогда он, скорее всего, должен быть ответственным и за определение того, что должно быть выведено на страницу. То есть, в нашем случае код можно было бы реорганизовать, выполнив проверку свойства `isLoading` в методе `render()` компонента `App` и выведя на экран текст наподобие `Loading...` в том случае, если загрузка не завершена, либо выведя компонент, подобный компоненту `Conditional` в том случае, если загрузка завершилась. При этом компонент `Conditional` вполне может и не принимать свойств от `App`, выводя лишь то, что он, в любом случае, должен выводить.

Вот как выглядит код компонента App, преобразованный в соответствии с этими рассуждениями:

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

import Conditional from "./Conditional"


class App extends Component {

    constructor() {

        super()

        this.state = {

            isLoading: true

        }

    }


    componentDidMount() {

        setTimeout(() => {

            this.setState({

                isLoading: false

            })

        }, 1500)

    }


    render() {
```

```
        return (
            <div>
                {this.state.isLoading ?
                <h1>Loading...</h1> :
                <Conditional />}
            </div>
        )
    }
}


export default App
```

А вот — обновлённый код компонента `Conditional`, в котором теперь нет проверки каких-либо условий:

```
import React from "react"


function Conditional(props) {
    return <h1>Some cool stuff about conditional rendering</h1>
}


export default Conditional
```

Тут мы, правда, убрали навигационную панель и «подвал», но это в данном случае неважно.

Учебный курс по React, часть 21: второе занятие и практикум по условному рендерингу

Занятие 37. Условный рендеринг, часть 2

Оригинал

На сегодняшнем занятии по условному рендерингу мы поговорим об использовании логического оператора `&&` (И). Экспериментировать будем со стандартным проектом, созданным средствами create-react-app, в файле `App.js` которого находится следующий код:

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


class App extends Component {
```

```
    constructor() {
        super()
        this.state = {
            unreadMessages: [
                "Call your mom!",
                "New spam email available. All links are definitely safe to
click."
            ]
        }
    }

    render() {
        return (
            <div>
                <h2>You have {this.state.unreadMessages.length} unread
messages!</h2>
            </div>
        )
    }
}

export default App
```

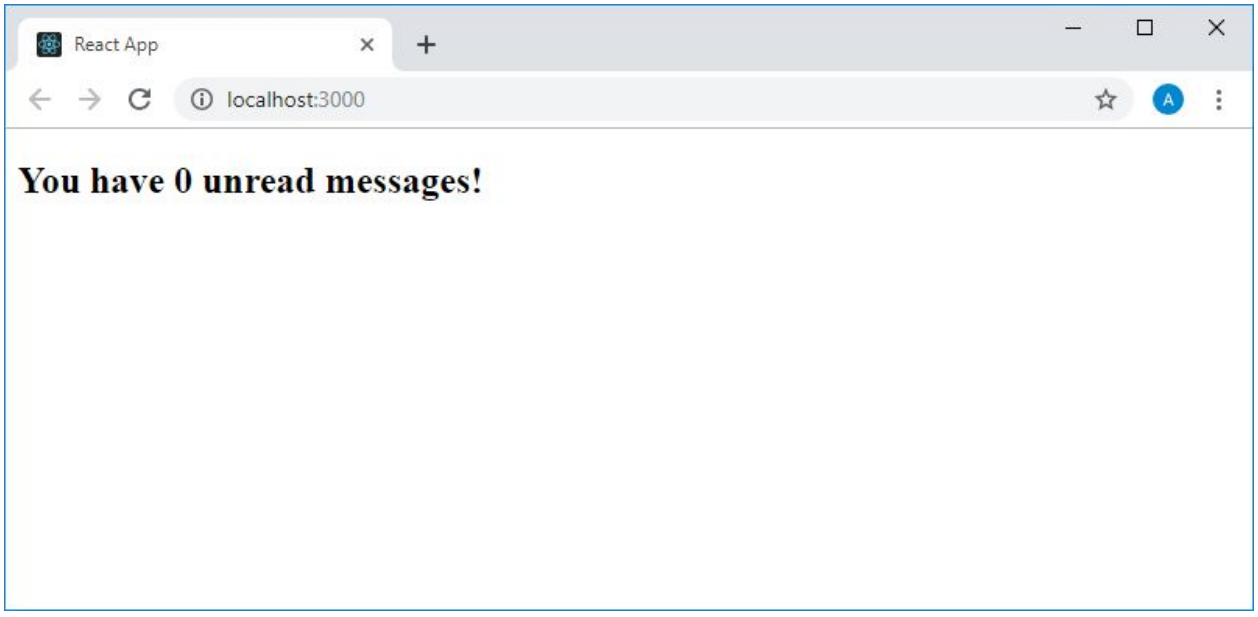
Сейчас приложение выглядит в браузере так, как показано ниже.

Страница приложения в браузере

Возможно, вы уже пользовались оператором `&&` в конструкциях наподобие `true && false` (что даёт `false`). Для того, чтобы в результате вычисления подобного выражения было бы возвращено `true`, оно должно выглядеть как `true && true`. При обработке таких выражений JavaScript определяет, является ли их левая часть истинной, и, если это так, просто возвращает то, что находится в их правой части. Если обрабатывается выражение вида `false && false`, то сразу будет возвращено `false`, без проверки правой части выражения. В результате оператор `&&` можно использовать в условном рендеринге. С его помощью можно либо вернуть что-то, что будет выведено на экран, либо не возвращать ничего.

Проанализируем код учебного приложения.

В состоянии компонента `App` хранится массив строк `unreadMessages`. Каждая строка в этом массиве представляет собой непрочитанное сообщение. На страницу выводится количество непрочитанных сообщений, определяемое на основе длины массива. Если же этот массив будет пустым, то есть в нём не будет ни одного элемента, то приложение выведет на страницу то, что показано ниже.



Приложение информирует нас о том, что непрочитанных сообщений нет

Для того чтобы добиться такого эффекта, достаточно привести массив к такому виду: `unreadMessages: []`.

Если непрочитанных сообщений нет, то вполне можно не выводить вообще никакого сообщения. Если воспользоваться для реализации такого поведения приложения тернарным оператором, о котором мы говорили в прошлый раз, метод `render()` компонента `App` можно переписать так:

```
render() {
    return (
        <div>
            {
                this.state.unreadMessages.length > 0 ?
                <h2>You have {this.state.unreadMessages.length} unread
messages!</h2> :
                null
            }
        </div>
    )
}
```

Теперь в том случае, если массив `unreadMessages` пуст, на страницу не будет выводиться ничего. Но представленный здесь код можно упростить благодаря использованию оператора `&&`. Вот как это будет выглядеть:

```
render() {
    return (
        <div>
            {
                this.state.unreadMessages.length > 0 &&
                <h2>You have {this.state.unreadMessages.length} unread
messages!</h2>
            }
        </div>
    )
}
```

Если в массиве что-то есть — на страницу выводится правая часть выражения. Если массив пуст — на страницу не выводится ничего.

Нельзя сказать, что использование оператора `&&` в условном рендеринге абсолютно необходимо, так как того же эффекта можно достичь с использованием тернарного оператора, возвращающего `null` в том случае, если проверяемое им условие ложно. Но представленный здесь подход упрощает код, и, кроме того, используется он довольно часто, поэтому вы можете столкнуться с ним, читая чужие программы.

Занятие 38. Практикум. Условный рендеринг

Оригинал

Задание
Вот код функционального компонента `App`, который хранится в файле `App.js` стандартного проекта, созданного с помощью create-react-app.

```
import React from "react"


function App() {
    return (
        <div>
            Code goes here
        </div>
    )
}


export default App
```

Вам нужно сделать следующее:

1. Преобразуйте код компонента так, чтобы оснастить его состоянием.
2. Сделайте так, чтобы в состоянии компонента хранились бы сведения о том, «вошёл» ли пользователь в систему или нет (в этом упражнении «вход» в систему и «выход» из неё означает лишь изменение соответствующего значения, хранящегося в состоянии).
3. Добавьте на страницу, которую формирует компонент, кнопку, которая позволяет пользователю входить в систему и выходить из неё.
    a. Это — дополнительное задание. Сделайте так, чтобы, если пользователь не вошёл в систему, на кнопке выводилась бы надпись `LOG IN`, а если вошёл — надпись `LOG OUT`.
4. Выведите на странице, формируемой компонентом, надпись `Logged in` в том случае, если пользователь вошёл в систему, и `Logged out` в том случае, если не вошёл.

Если сейчас вы чувствуете, что приступить к решению этих задач вам сложно — взгляните на подсказки, а потом приступайте к работе.

Подсказки
Для выполнения этого задания нужно вспомнить многое из того, о чём мы говорили на предыдущих занятиях. Начнём с того, что компонент, который может иметь состояние, должен быть компонентом, который основан на классе. У этого компонента должен быть конструктор. В состоянии компонента можно хранить логическое свойство, например, его можно назвать `isLoggedIn`, значение которого, `true` или `false`, указывает на то, вошёл пользователь в систему или нет. Для того чтобы кнопка,

которую нужно добавить на страницу, генерируемую приложением, могла бы выполнять свои функции, ей понадобится обработчик события `onClick`. Для того чтобы выводить разные тексты, опираясь на изменяющееся значение состояния, нужно будет прибегнуть к технологии условного рендеринга.

Решение
Преобразуем имеющийся в коде функциональный компонент в компонент, основанный на классе. Нам это нужно по нескольким причинам. Во-первых, нам нужно работать с состоянием приложения. Во-вторых, нам нужен обработчик события, вызываемый при щелчке по кнопке. В принципе, можно написать самостоятельную функцию и использовать её для обработки событий кнопки, но я предпочитаю описывать обработчики в пределах классов компонентов.

Вот как будет выглядеть код функционального компонента, преобразованного в компонент, основанный на классе. Здесь же мы, в конструкторе компонента, описываем его первоначальное состояние, которое содержит свойство `isLoggedIn`, установленное в значение `false`.

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            isLoggedIn: false
        }
    }
    render() {
        return (
            <div>
                Code goes here
            </div>
        )
    }
}


export default App
```

Вышеприведённый код представляет собой решение первой и второй частей задания. Теперь поработаем над добавлением кнопки на страницу, выводимую компонентом. Пока эта кнопка будет выводить одну и ту же надпись независимо от того, что хранится в состоянии приложения. Мы оснастим её обработчиком события, поместив в него, для проверки работоспособности нашего кода, простую команду вывода сообщения в консоль. Кроме того, мы, в конструкторе компонента, привяжем

этот обработчик к `this`, что пригодится нам тогда, когда мы будем, в коде этого обработчика, обращаться к механизмам, предназначенным для работы с состоянием компонента. Сейчас код выглядит так, как показано ниже.

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            isLoggedIn: false
        }
        this.handleClick = this.handleClick.bind(this)
    }


    handleClick() {
        console.log("I'm working!")
    }


    render() {
        return (
            <div>
                <button onClick={this.handleClick}>LOG IN</button>
            </div>
        )
    }
}


export default App
```

При нажатии на кнопку `LOG IN` в консоль попадает текст `I'm working!`.

Теперь вспомним о том, что нам надо, чтобы при щелчке по кнопке свойство `isLoggedIn`, хранящееся в состоянии, менялось бы с `true` на `false` и наоборот. Для этого в обработчике щелчка по кнопке нужно будет вызвать функцию `this.setState()`, которую можно использовать двумя способами. А именно, ей можно предоставить, в виде объекта, новое представление того, что должно содержаться в

состоянии. Второй вариант её использования предусматривает передачу ей функции, которая принимает предыдущее состояние компонента и формирует новое, возвращая, опять же, объект. Мы поступим именно так. Вот что у нас получилось на данном этапе работы.

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            isLoggedIn: false
        }
        this.handleClick = this.handleClick.bind(this)
    }

    handleClick() {
        this.setState(prevState => {
            return {
                isLoggedIn: !prevState.isLoggedIn
            }
        })
    }

    render() {
        return (
            <div>
                <button onClick={this.handleClick}>LOG IN</button>
            </div>
        )
    }
}
```

```
export default App
```

Тут можно было бы воспользоваться конструкцией if-else, но мы просто преобразовываем значение `true` в значение `false`, а значение `false` в значение `true` с использованием логического оператора `!` (НЕ).

Пока у нас нет механизма, который, основываясь на том, что хранится в состоянии, позволял бы влиять на внешний вид приложения. Поэтому сейчас мы решим дополнительную задачу задания №3. А именно, сделаем так, чтобы надпись на кнопке менялась бы в зависимости от состояния компонента. Для того чтобы этого добиться, можно объявить в методе `render()` переменную, значение которой, `LOG IN` или `LOG OUT`, зависит от того, что хранится в состоянии. Это значение потом можно использовать в качестве текста кнопки. Вот как это выглядит.

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            isLoggedIn: false
        }
        this.handleClick = this.handleClick.bind(this)
    }

    handleClick() {
        this.setState(prevState => {
            return {
                isLoggedIn: !prevState.isLoggedIn
            }
        })
    }

    render() {
        let buttonText = this.state.isLoggedIn ? "LOG OUT" : "LOG IN"
        return (
            <div>
```

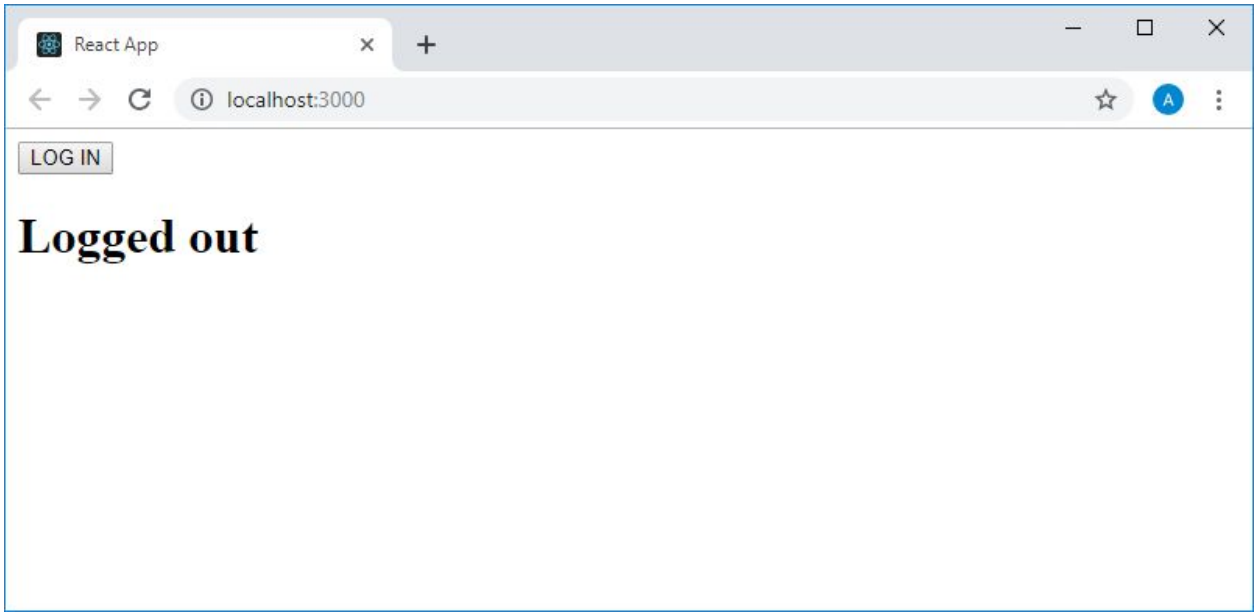
```
                <button onClick={this.handleClick}>{buttonText}</button>
            </div>
        )
    }
}

export default App
```

Теперь займёмся четвёртой частью задания. Будем выводить на страницу текст, зависящий от того, вошёл пользователь в систему или нет. Собственно говоря, учитывая всё то, что уже присутствует в коде компонента, решить эту задачу очень просто. Ниже представлен готовый код файла `App.js`.

```
import React from "react"


class App extends React.Component {
    constructor() {
        super()
        this.state = {
            isLoggedIn: false
        }
        this.handleClick = this.handleClick.bind(this)
    }


    handleClick() {
        this.setState(prevState => {
            return {
                isLoggedIn: !prevState.isLoggedIn
            }
        })
    }


    render() {
        let buttonText = this.state.isLoggedIn ? "LOG OUT" : "LOG IN"
```

```
        let displayText = this.state.isLoggedIn ? "Logged in" : "Logged out"
        return (
            <div>
                <button onClick={this.handleClick}>{buttonText}</button>
                <h1>{displayText}</h1>
            </div>
        )
    }
}

export default App
```

Вот как выглядит приложение в браузере.



Страница приложения в браузере

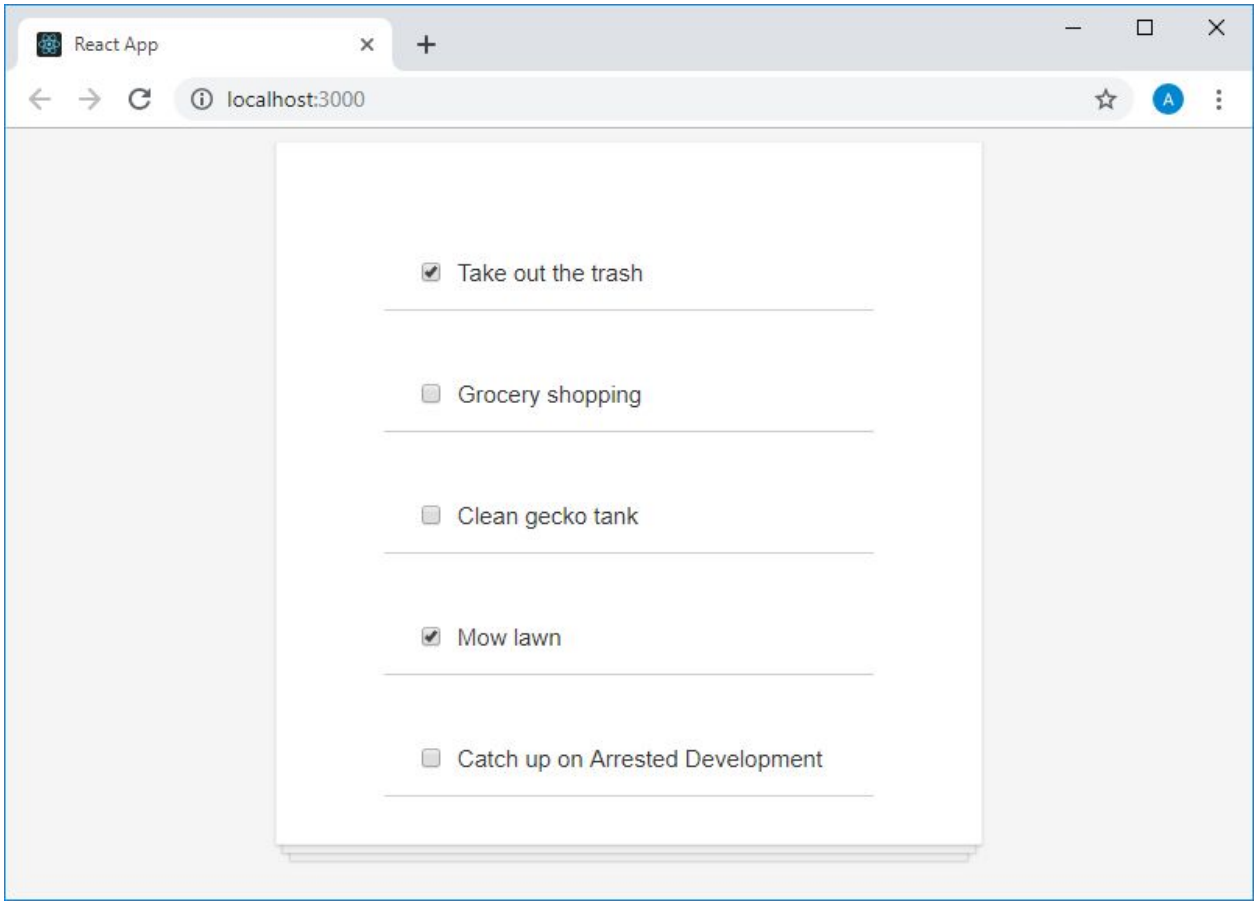
Щелчок по кнопке `LOG IN`, изображённой на предыдущем рисунке, меняет состояние приложения, после чего на кнопке выводится надпись `LOG OUT`, а на странице выводится текст, информирующий пользователя о том, что он вошёл в систему.

Учебный курс по React, часть 22: седьмой этап работы над TODO-приложением, загрузка данных из внешних источников

Занятие 39. Практикум. TODO-приложение. Этап №7

Оригинал

Задание
Сейчас Todo-приложение выглядит так, как показано на следующем рисунке.



Страница приложения в браузере

Код компонента `TodoItem` выглядит так:

```
import React from "react"


function TodoItem(props) {
    return (
        <div className="todo-item">
            <input
                type="checkbox"
                checked={props.item.completed}
                onChange={() => props.handleChange(props.item.id)}
            />
            <p>{props.item.text}</p>
        </div>
    )
}
```

```
export default TodoItem
```

Ваша задача заключается в том, чтобы стилизовать пункты списка в зависимости от их состояния. Внешний вид завершённых дел должен отличаться от незавершённых. При форматировании пунктов списка, представляющих завершённые дела, их текст можно сделать серым, его можно перечеркнуть, представить курсивом или выполнить другие его модификации.

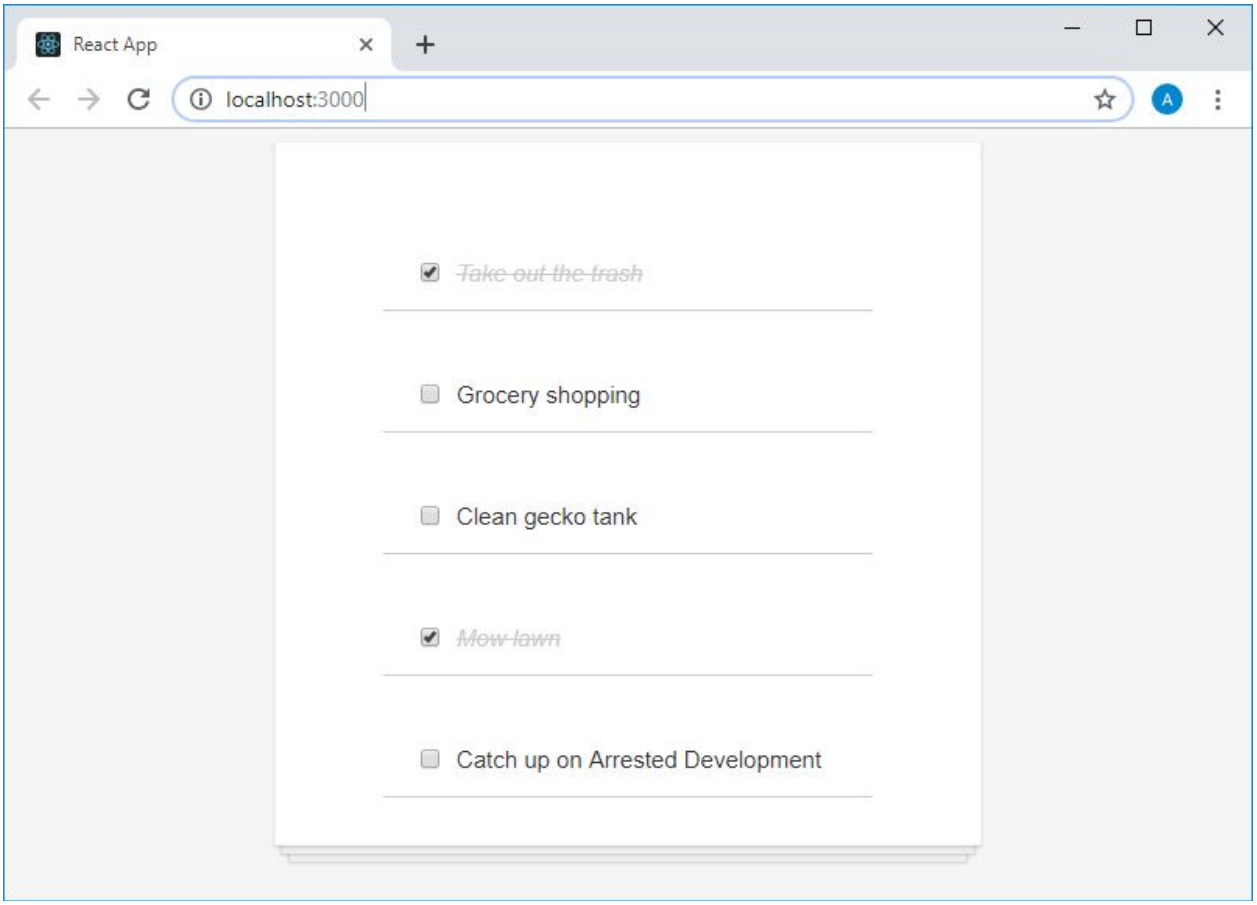
Решение

Представленную здесь задачу можно решить разными способами. Мы воспользуемся встроенным стилем, который опишем в виде константы `completedStyle` в коде функционального компонента `TodoItem`. Тут мы настроим свойства текста `fontStyle`, `color` и `textDecoration`. После этого, пользуясь методикой условного рендеринга, назначим этот стиль элементу `<p>` в том случае, если выводимое им дело отмечено как завершённое. Определять это будем, ориентируясь на переданное экземпляру компонента свойство, которое доступно в нём как `props.item.completed`.
Преобразованный код компонента будет выглядеть так:

```
import React from "react"


function TodoItem(props) {

    const completedStyle = {

        fontStyle: "italic",

        color: "#cdcdcd",

        textDecoration: "line-through"

    }


    return (

        <div className="todo-item">

            <input

                type="checkbox"

                checked={props.item.completed}

                onChange={() => props.handleChange(props.item.id)}

            />

            <p style={props.item.completed ? completedStyle:
null}>{props.item.text}</p>

        </div>

    )

}
```

```
export default TodoItem
```

Вот как изменится внешний вид страницы приложения.



Изменённая страница приложения в браузере

При этом стили применяются при установке и снятии флажков, указывающих на состояние пунктов списка дел.

На этом мы завершаем работу над Todo-приложением.

Занятие 40. Загрузка данных из внешних источников

Оригинал

На занятии, посвящённом методам жизненного цикла компонентов, мы говорили о методе `componentDidMount()`. Попытайтесь вспомнить о том, как именно он работает. Этот метод позволяет вмешиваться в работу компонента, выполняя некий код сразу после того, как компонент был добавлен в дерево DOM. Когда мы говорили о методах жизненного цикла компонентов, я упоминал о том, что метод `componentDidMount()` чаще всего используется для загрузки данных из неких внешних источников. Эти данные используются компонентом для реализации его предназначения.

Начнём наши сегодняшние эксперименты с нового проекта, созданного средствами create-react-app, файл `App.js` которого содержит следующий код:

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


class App extends Component {
```

```
    constructor() {
        super()
        this.state = {}
    }


    render() {
        return (
            <div>
                Code goes here
            </div>
        )
    }
}

export default App
```

Опишем в коде компонента `App`, основанного на классе, метод `componentDidMount()` и проверим работоспособность полученной конструкции, выведя из этого метода что-нибудь в консоль.

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

class App extends Component {
    constructor() {
        super()
        this.state = {}
    }

    componentDidMount() {
        console.log("Hi!")
    }
```

```
    render() {
        return (
            <div>
                Code goes here
            </div>
        )
    }
}


export default App
```

Вывод в консоль строки `Hi!` доказывает работоспособность кода, поэтому мы можем продолжать работу. Как уже было сказано, в этом методе обычно загружают данные, необходимые для работы компонента.

На этом занятии мы будем пользоваться несколькими вспомогательными средствами, которые пригодятся нам при загрузке данных в компонент.

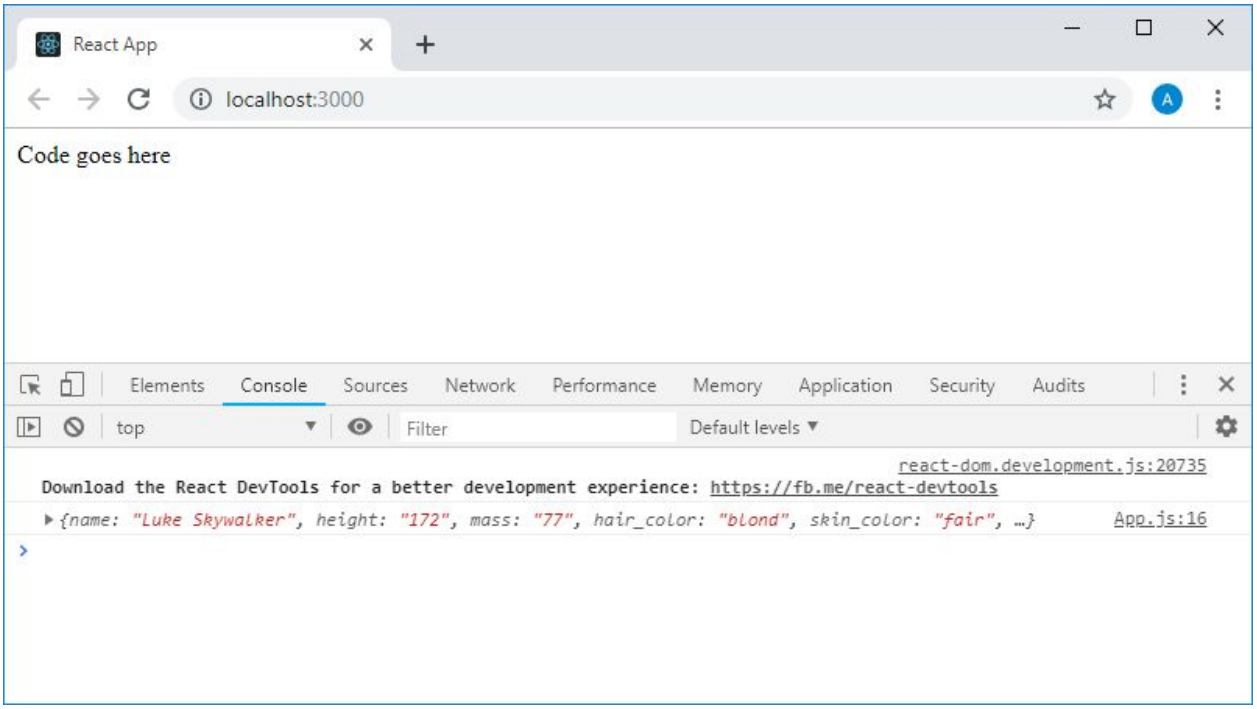
Первое из них представляет собой встроенную возможность JavaScript. Речь идёт об API Fetch, представляющем собой удобный интерфейс для получения ресурсов, основанный на промисах, позволяющий выполнять HTTP-запросы, посредством которых и загружаются данные.

Второе средство, которым мы будем пользоваться — это API Star Wars. Этот проект хорош тем, что им, без особых сложностей, можно пользоваться во фронтенд-приложениях (речь, в частности, идёт об особенностях настройки CORS).

В методе `componentDidMount()` мы собираемся воспользоваться функцией `fetch()`, передав ей адрес для загрузки данных, преобразовать эти данные к нужному нам виду, и, для того, чтобы проверить правильность работы системы, вывести эти данные в консоль. Преобразуем код метода к следующему виду:

```
componentDidMount() {
    fetch("https://swapi.co/api/people/1")
        .then(response => response.json())
        .then(data => console.log(data))
}
```

Здесь мы загружаем данные о некоем герое фильма, обращаясь к API, после этого преобразуем то, что пришло от сервера, в формат JSON, а потом выводим эти данные в консоль. То, что попало в консоль, представлено на следующем рисунке.



Данные, загруженные из API Star Wars, выведены в консоль

Как видно, в консоль попал объект с данными о Люке Скайуокере. Теперь, после того, как у нас есть данные, нам надо подумать о том, как вывести их на страницу приложения. Для того чтобы решить эту задачу, сначала стоит учитывать то, что загруженные извне данные, если их никуда не сохранить, невозможно будет вывести на страницу приложения в браузере. Местом, которое служит для хранения подобных данных, является состояние компонента. Добавим в состояние компонента новое свойство, `character`, представленное пустым объектом:

```
this.state = {

    character: {}

}
```

Мы собираемся хранить в этом свойстве объект с описанием персонажа, данные о котором загружены из внешнего источника. Они существуют в виде объекта, поэтому мы, инициализируя состояния, делаем свойство `character` пустым объектом.

После этого в том месте кода метода `componentDidMount()`, где мы получаем данные, мы запишем их в состояние, воспользовавшись методом `setState()`. При этом в данном случае то, что хранилось в состоянии до этого, нас не интересует, поэтому мы можем просто передать этому методу объект, содержащий новое представление состояния. В итоге мы приходим к такому коду метода `componentDidMount()`:

```
componentDidMount() {

    fetch("https://swapi.co/api/people/1")

        .then(response => response.json())

        .then(data => {

            this.setState({

                character: data
```

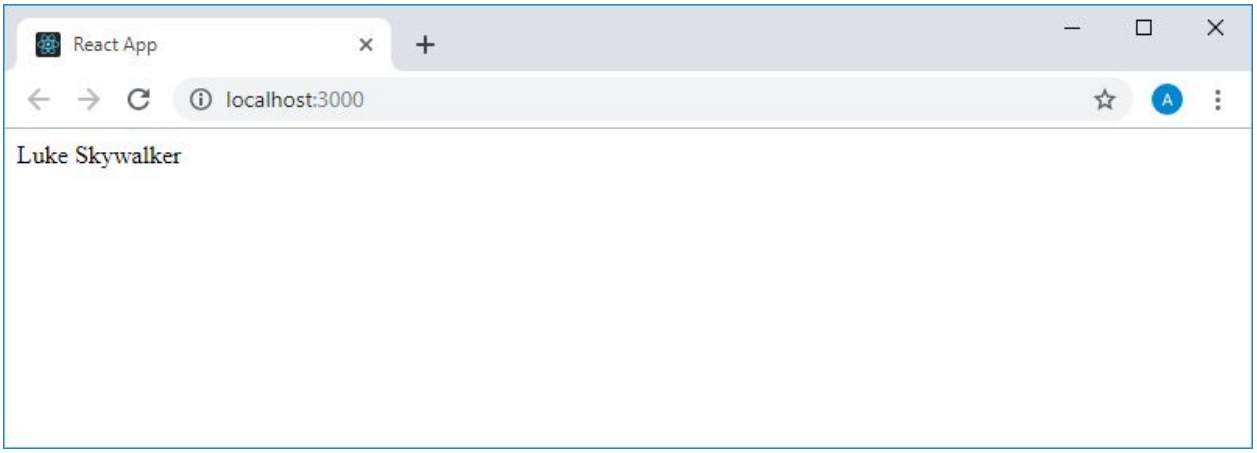
```
            })
        })
}
```

Для того чтобы проверить правильность работы тех механизмов, которые теперь существуют в коде, выведем в методе `render()` что-нибудь, что должно присутствовать в состоянии после записи в него загруженных данных. Теперь код файла `App.js` будет выглядеть так:

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


class App extends Component {
    constructor() {
        super()
        this.state = {
            character: {}
        }
    }


    componentDidMount() {
        fetch("https://swapi.co/api/people/1")
            .then(response => response.json())
            .then(data => {
                this.setState({
                    character: data
                })
            })
    }


    render() {
        return (
            <div>
                {this.state.character.name}
            </div>
```

```
        )
    }
}

export default App
```

А вот как будет выглядеть страница приложения в браузере.



Страница приложения в браузере

Вывод на страницу текста `Luke Skywalker` демонстрирует правильную работу механизмов загрузки данных.

В нашем приложении используется простой запрос, в ответ на который приложение получает небольшой объём данных, которые быстро обрабатываются и выводятся на страницу. На выполнение всех этих действий уходит очень мало времени. Поэтому данные на экран выводятся так быстро, что у нас возникает такое впечатление, будто компонент, сразу после вывода его на экран, уже их в себе содержит. Но если бы обращение к удалённому источнику данных проводилось бы с использованием очень медленной линии связи, или API, из которого загружаются данные, медленно отвечало бы на запросы, до того момента, как приложение могло бы вывести эти данные на экран, могло бы пройти немало времени. Всё это время экран оставался бы пустым. Если подобное встречается в реальных приложениях, это сбивает с толку их пользователей, которые могут решить, что такие приложения работают неправильно. Для того чтобы предусмотреть подобную ситуацию, нужно, во время загрузки и обработки данных, показывать пользователю соответствующее сообщение. Это не относится к нашей сегодняшней теме, но именно здесь уместно будет это обсудить.

В реальных приложениях, для оповещения пользователя о том, что ему нужно дождаться выполнения некоего действия, вроде загрузки данных, пользуются чем-то вроде индикатора загрузки. В нашем же случае мы, до того момента, как данные будут загружены и готовы к выводу на страницу, будем просто показывать текст `loading...`. Делая это, мы сможем оценить возможности, которые даёт нам хранение данных в состоянии приложения.

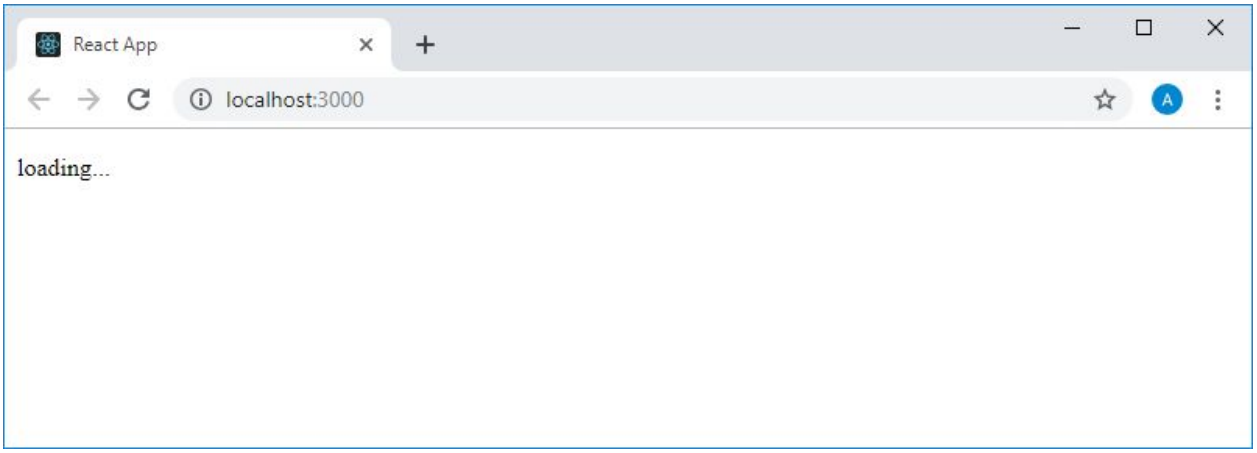
Добавим в состояние новое свойство, указывающее на то, выполняется ли в некий момент времени загрузка данных. Назовём его `loading` и инициализируем его значением `false`. После этого, сразу перед тем, как выполнять загрузку данных с использованием `fetch()`, запишем в это свойство `true`.

Далее, в методе `render()`, опираясь на свойство состояния `loading`, настроим текст, выводимый на страницу. Вот как будет выглядеть код `App.js` после этих преобразований.

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


class App extends Component {
    constructor() {
        super()
        this.state = {
            loading: false,
            character: {}
        }
    }


    componentDidMount() {
        this.setState({loading: true})
        fetch("https://swapi.co/api/people/1")
            .then(response => response.json())
            .then(data => {
                this.setState({
                    character: data
                })
            })
    }


    render() {
        const text = this.state.loading ? "loading..." :
this.state.character.name
        return (
            <div>
                <p>{text}</p>
            </div>
        )
    }
```

```
}


export default App
```

Этот код, правда, работает неправильно. А именно, вот как теперь выглядит страница приложения.



Страница приложения в браузере

Предполагается, что надпись `loading...` должна выводиться на ней только во время загрузки данных из внешнего источника, но она, судя по всему, выводится теперь на странице постоянно. Попытайтесь, прежде чем читать дальше, найти и исправить ошибки в коде.

Собственно говоря, проблема тут заключается в том, что мы, перед началом загрузки данных, установили `loading` в `true`, а после завершения загрузки не записали в `loading` `false`. В результате на странице всегда выводится текст `loading...`. Исправить эту ошибку несложно. Достаточно, там же, где мы записываем в состояние загруженные данные, установить `loading` в значение `false`. В результате код `App.js` приобретёт следующий вид:

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


class App extends Component {

    constructor() {

        super()

        this.state = {

            loading: false,

            character: {}

        }

    }


    componentDidMount() {

        this.setState({loading: true})
```

```
        fetch("https://swapi.co/api/people/1")
            .then(response => response.json())
            .then(data => {
                this.setState({
                    loading: false,
                    character: data
                })
            })
    }

    render() {
        const text = this.state.loading ? "loading..." :
this.state.character.name
        return (
            <div>
                <p>{text}</p>
            </div>
        )
    }
}

export default App
```

Теперь при загрузке данных ненадолго появляется надпись `loading...`, а после этого на страницу выводится имя персонажа.

Учебный курс по React, часть 23: первое занятие по работе с формами

Занятие 41. Работа с формами, часть 1

Оригинал

Формы являются достаточно важной частью веб-приложений. Но, как оказалось, у тех, кто занимается освоением React, работа с формами обычно вызывает определённые сложности. Дело в том, что в React с формами работают по-особенному. На этом занятии мы будем пользоваться стандартным проектом, создаваемым create-react-app, исходный вид файла `App.js` которого представлен ниже.

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

```
class App extends Component {

    constructor() {

        super()

        this.state = {}

    }


    render() {

        return (

            <div>

                Code goes here

            </div>

        )

    }

}


export default App
```

Обратите внимание на то, что для того, чтобы освоить материал этого занятия, вы должны быть знакомы с понятием состояния приложения. Если вы проработали все предыдущие занятия курса и самостоятельно выполняли практикумы, это значит, что вы обладаете знаниями, которые вам здесь понадобятся. Вот документация React, посвящённая формам. Рекомендуется, прежде чем продолжать, взглянуть на неё.

Итак, в React с формами работают немного не так, как при использовании обычного JavaScript. А именно, при обычном подходе формы описывают средствами HTML на веб страницах, после чего, пользуясь API DOM, взаимодействуют с ними из JavaScript. В частности, по нажатию на кнопку отправки формы, собирают данные из полей, заполненных пользователем, и готовят их к отправке на сервер, проверяя их, и, при необходимости, сообщая пользователю о том, что он заполнил какие-то поля неверно. В React же, вместо того, чтобы ожидать ввода всех материалов в поля формы перед тем, как приступить к их программной обработке, за данными наблюдают постоянно, пользуясь состоянием приложения. Это, например, сводится к тому, что каждый символ, введённый пользователем с клавиатуры, сразу же попадает в состояние. В результате в React-приложении мы можем оперативно работать с самой свежей версией того, что пользователь вводит в поля формы. Для того, чтобы продемонстрировать эту идею в действии, начнём с описания формы, содержащей обычное текстовое поле.

Для этого, в коде, который возвращает метод `render()`, опишем форму. При обычном подходе такая форма имела бы кнопку, по нажатию на которую программные механизмы приложения приступают к обработке данных, введённых в эту форму. В нашем же случае данные, введённые пользователем в поле, будут поступать в приложение по мере их ввода. Для этого нам понадобится обрабатывать событие поля `onChange`. В обработчике этого события (назовём его `handleChange()`) мы будем

обновлять состояние, записывая в него то, что введено в поле. Для этого нам понадобится, во-первых, узнать, что содержится в поле, и во-вторых — поместить эти данные в состояние. В состоянии создадим свойство, хранящее содержимое поля. Это поле мы собираемся использовать для хранения имени (first name) пользователя, поэтому назовём соответствующее свойство `firstName` и инициализируем его пустой строкой.

После этого, в конструкторе, привяжем обработчик события `handleChange()` к `this` и в коде обработчика воспользуемся функцией `setState()`. Так как предыдущее значение, которое хранилось в свойстве состояния `firstName`, нас не интересует, мы можем просто передать этой функции объект с новым значением `firstName`. Что должно быть записано в это свойство?

Если вспомнить то, как работают обработчики событий в JavaScript, то окажется, что при их вызове им передаются некие предопределённые параметры. В нашем случае обработчику передаётся объект события (`event`). Он и содержит интересующие нас данные. Текстовое поле, событие `onChange` которого мы обрабатываем, представлен в этом объекте в виде `event.target`. А к содержимому этого поля можно обратиться, воспользовавшись конструкцией `event.target.value`.

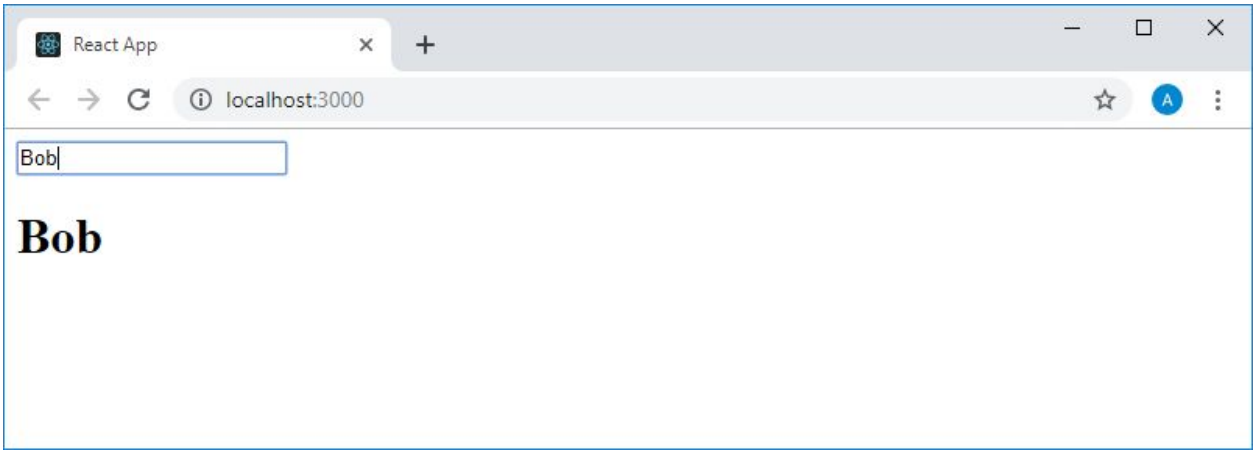
Теперь, в методе `render()`, выведем то, что будет храниться в состоянии и посмотрим на то, что у нас получилось.

Вот код, реализующий вышеописанные идеи.

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


class App extends Component {

    constructor() {

        super()

        this.state = {

            firstName: ""

        }

        this.handleChange = this.handleChange.bind(this)

    }


    handleChange(event) {

        this.setState({

            firstName: event.target.value

        })

    }


    render() {
```

```
        return (
            <form>
                <input type="text" placeholder="First Name"
onChange={this.handleChange} />
                <h1>{this.state.firstName}</h1>
            </form>
        )
    }
}

export default App
```

Вот как всё это выглядит в браузере.



Страница приложения в браузере

Каждый символ, введённый в поле, тут же появляется в элементе `<h1>`, присутствующем на странице.

Подумаем теперь о том, как добавить на форму ещё одно поле, для фамилии (last name) пользователя. Очевидно, что для того, чтобы наладить правильную обработку данных, вводимых в это поле, нам понадобится добавить в состояние ещё одно свойство и поработать над механизмами, обновляющими состояние при вводе данных в поле.

Один из подходов к решению этой задачи заключается в создании отдельного обработчика событий для нового поля. Для маленькой формы с несколькими полями ввода это — вполне нормальный подход, но если речь идёт о форме с десятками полей, создавать для каждого из них отдельный обработчик события `onChange` — не лучшая идея.

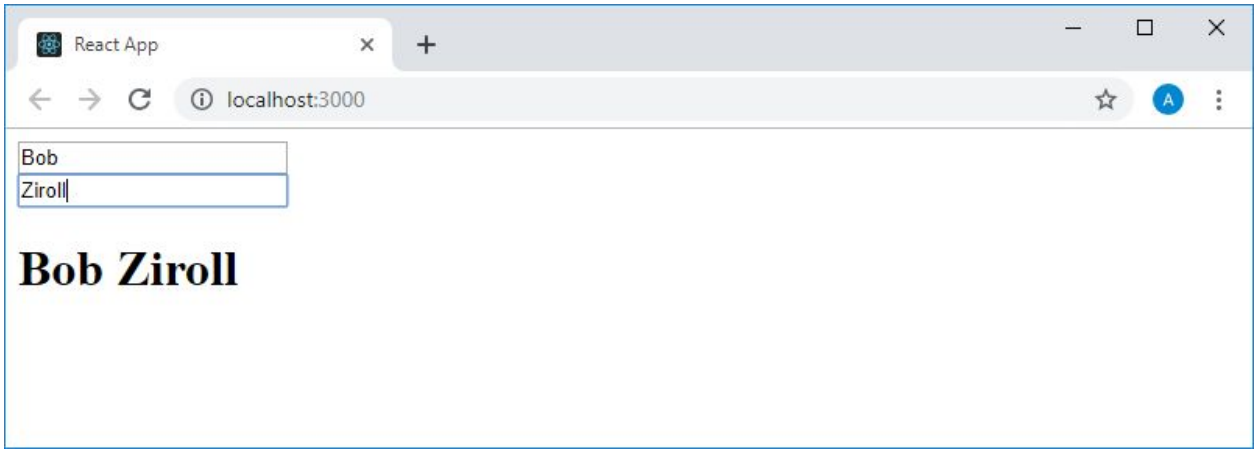
Для того чтобы в одном и том же обработчике событий различать поля, при изменении которых он вызывается, мы назначим полям свойства `name`, которые сделаем точно такими же, какими сделаны имена свойств, используемых для хранения данных полей в состоянии (`firstName` и `lastName`). После этого мы сможем, работая с объектом события, который передаётся обработчику, узнать имя поля, изменения в котором привели к его вызову, и воспользоваться этим именем. Мы будем

пользоваться им, задавая имя свойства состояния, в которое хотим внести обновлённые данные. Вот код, который реализует эту возможность:

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


class App extends Component {
    constructor() {
        super()
        this.state = {
            firstName: "",
            lastName: ""
        }
        this.handleChange = this.handleChange.bind(this)
    }


    handleChange(event) {
        this.setState({
            [event.target.name]: event.target.value
        })
    }


    render() {
        return (
            <form>
                <input type="text" name="firstName" placeholder="First Name"
onChange={this.handleChange} />
                <br />
                <input type="text" name="lastName" placeholder="Last Name"
onChange={this.handleChange} />
                <h1>{this.state.firstName} {this.state.lastName}</h1>
            </form>
        )
    }
```

```
}


export default App
```

Обратите внимание на то, что, задавая имя свойства объекта, передаваемого `setState()`, мы заключаем конструкцию `event.target.name` в прямоугольные скобки.



Страница приложения в браузере

На страницу теперь выводится то, что вводится в первое поле, и то, что вводится во второе поле.

Принципы работы с текстовыми полями, которые мы только что рассмотрели, справедливы и для других полей, основанных на них. Например, это могут быть поля для ввода адресов электронной почты, телефонов, чисел. Данные, вводимые в такие поля, можно обрабатывать, используя вышерассмотренные механизмы, для работы которых важно, чтобы имена полей соответствовали бы именам свойств в состоянии компонента, хранящих данные этих полей.

О работе с другими элементами форм мы поговорим на следующем занятии. Здесь же мы затронем ещё одну тему, которая имеет отношение к так называемым «управляемым компонентам» (controlled component), о которых вы, если посмотрели документацию React по формам, уже кое-что читали.

Если нам нужно чтобы то, что выводится в поле, в точности соответствовало бы тому, что хранится в состоянии приложения, мы можем воспользоваться описанным выше подходом, при применении которого состояние обновляется по мере ввода данных в поле. Состояние является реактивным. А при использовании элементов формы, являющихся управляемыми компонентами, тем, что выводится в этих элементах, управляет состояние. Именно оно при таком подходе является единственным источником истинных данных компонента. Для того чтобы этого добиться, достаточно добавить в код описания элемента формы атрибут `value`, указывающий на соответствующее полю свойство состояния. Вот как будет теперь выглядеть код приложения.

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


class App extends Component {

    constructor() {

        super()

        this.state = {
```

```
        firstName: "",
        lastName: ""
    }
    this.handleChange = this.handleChange.bind(this)
}

handleChange(event) {
    this.setState({
        [event.target.name]: event.target.value
    })
}

render() {
    return (
        <form>
            <input
                type="text"
                value={this.state.firstName}
                name="firstName"
                placeholder="First Name"
                onChange={this.handleChange}
            />
            <br />
            <input
                type="text"
                value={this.state.lastName}
                name="lastName"
                placeholder="Last Name"
                onChange={this.handleChange}
            />
```

```
                <h1>{this.state.firstName} {this.state.lastName}</h1>
            </form>
        )
    }
}

export default App
```

После этих изменений приложение работает точно так же, как и раньше. Главное отличие от его предыдущей версии заключается в том, что в поле выводится то, что хранится в состоянии.

Хочу дать один совет, который избавит вас в будущем от ошибок, которые очень тяжело отлаживать. Вот как выглядит код обработчика событий `onChange` сейчас:

```
handleChange(event) {
    this.setState({
        [event.target.name]: event.target.value
    })
}
```

Рекомендуется, вместо прямого обращения к свойствам объекта `event` при конструировании объекта, передаваемого `setState()`, заранее извлечь из него то, что нужно:

```
handleChange(event) {
    const {name, value} = event.target
    this.setState({
        [name]: value
    })
}
```

Здесь мы не будем вдаваться в подробности, касающиеся ошибок, которых можно избежать, конструируя обработчики событий именно так. Если вам это интересно — почитайте о SyntheticEvent в документации React.

Учебный курс по React, часть 24: второе занятие по работе с формами

Занятие 42. Работа с формами, часть 2

Оригинал

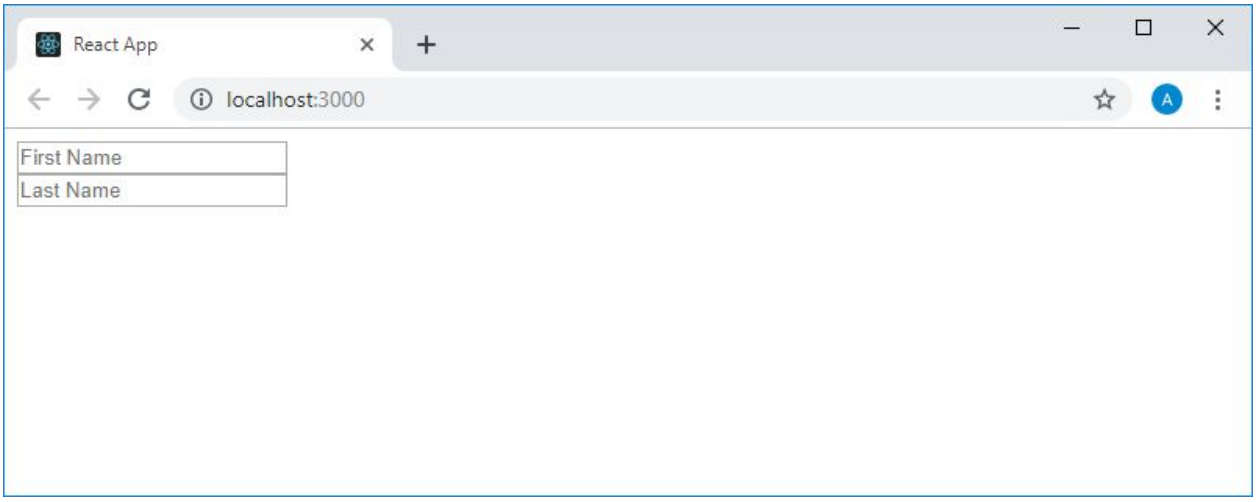
На этом занятии мы поговорим о полях для ввода многострочного текста, о флажках, о переключателях (их ещё называют «радиокнопками») и о полях со списками. К настоящему моменту мы рассмотрели лишь работу с обычными текстовыми полями ввода.

Вот код компонента `App`, с которого мы начнём сегодняшние эксперименты:

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


class App extends Component {
    constructor() {
        super()
        this.state = {
            firstName: "",
            lastName: ""
        }
        this.handleChange = this.handleChange.bind(this)
    }

    handleChange(event) {
        const {name, value} = event.target
        this.setState({
            [name]: value
        })
    }

    render() {
        return (
            <form>
                <input
                    type="text"
                    value={this.state.firstName}
                    name="firstName"
                    placeholder="First Name"
                    onChange={this.handleChange}
                />
```

```
                <br />
                <input
                    type="text"
                    value={this.state.lastName}
                    name="lastName"
                    placeholder="Last Name"
                    onChange={this.handleChange}
                />

                {
                    /**
                     * Другие полезные элементы форм:
                     *
                     *  <textarea />
                     *  <input type="checkbox" />
                     *  <input type="radio" />
                     *  <select> и <option>
                     */
                }

                <h1>{this.state.firstName} {this.state.lastName}</h1>
            </form>
        )
    }
}

export default App
```

Вот как выглядит страница приложения в браузере на данном этапе работы.



Страница приложения в браузере

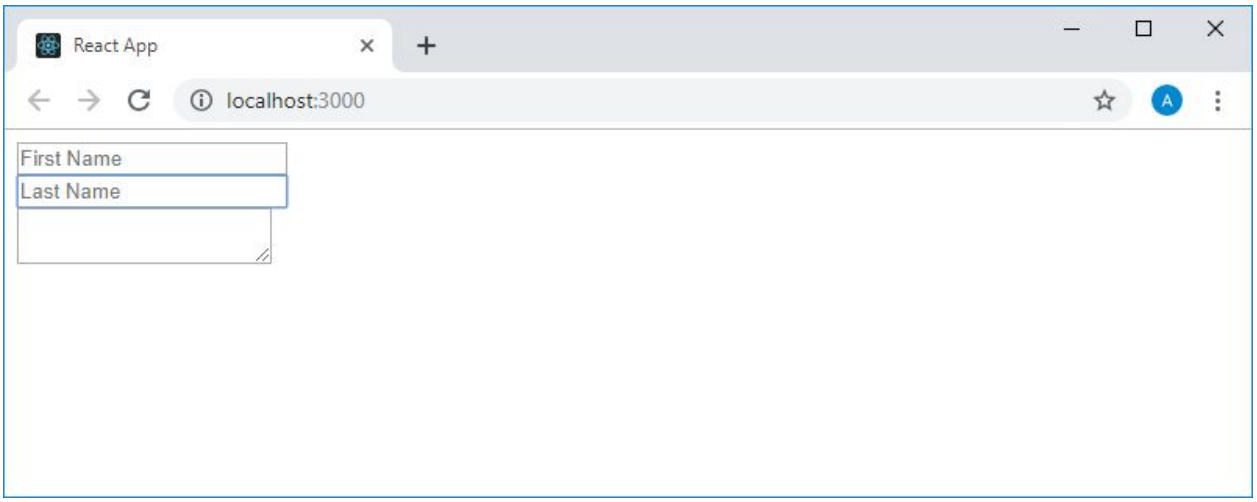
Формы обычно содержат в себе не только поля, в которые вводят короткие строки. При оснащении форм другими элементами работа с ними в React немного усложняется, хотя ничего особенного в этом нет.

В вышеприведённом коде есть закомментированный фрагмент, в котором перечислены элементы, о которых мы будем говорить. Начнём с поля для ввода многострочного текста — элемента `textarea`. Вероятно, понять то, как с ним работать, легче всего. Если вы раньше, строя обычные HTML-формы, уже пользовались этим элементом, вы знаете, что это — не самозакрывающийся тег, как было в случае с элементом `input`. У него есть открывающая и закрывающая части.

Добавим на форму этот элемент, вставив, сразу после комментария, следующий код:

```
<br />

<textarea></textarea>
```

Если теперь взглянуть на страницу приложения, то можно видеть, как на ней появилось поле для ввода многострочного текста.



Поле для ввода текста на странице

Как видно, это поле немного выше обычных полей, пользователь может менять его размеры, пользуясь маркером в его правой нижней части. Благодаря атрибутам `rows` и `cols` можно, при описании этого элемента, указывать его размеры. В обычном HTML, если нужно, чтобы после вывода этого поля в нём уже был какой-то текст, делается это путём ввода нужного текста между открывающим и закрывающим тегами элемента. В React работа с такими элементами сделана максимально похожей на работу с

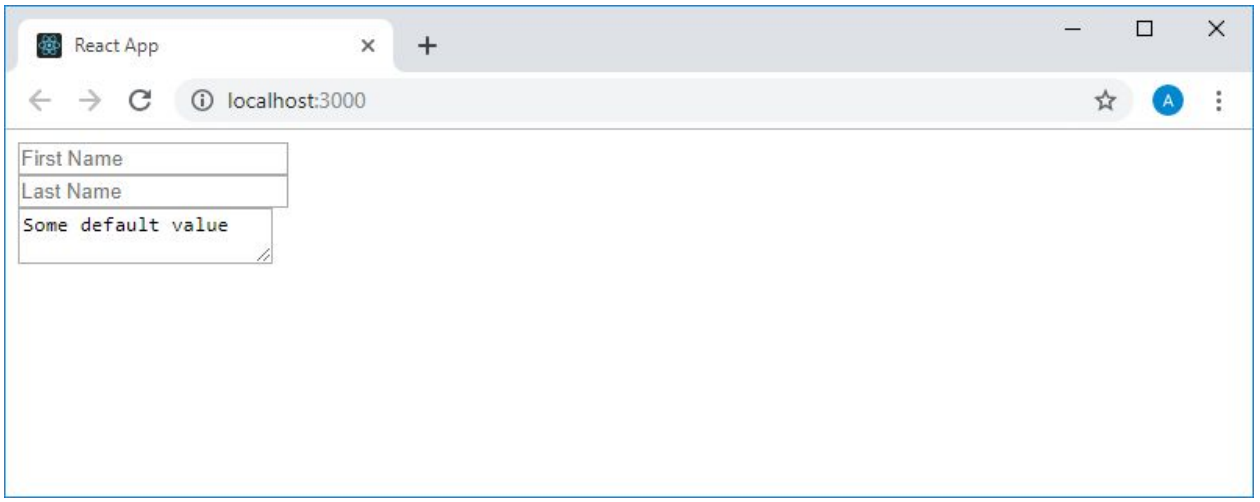
элементами `input`, о которых мы говорили в прошлый раз. А именно, в React тег `textarea` является самозакрывающимся. То есть код для вывода поля на страницу можно изменить следующим образом:

```
<textarea />
```

В этом теге можно использовать атрибут `value`, причём, работа с ним осуществляется точно так же, как и с таким же атрибутом обычных текстовых полей. Благодаря этому достигается единообразие в работе с разными элементами, и, кроме того, облегчается обновление содержимого полей путём обновления свойств состояния, привязанных к таким полям. Приведём состояние кода поля к такому виду:

```
<textarea value={"Some default value"}/>
```

Это приведёт к тому, что указанный текст появится в поле при выводе его на страницу.

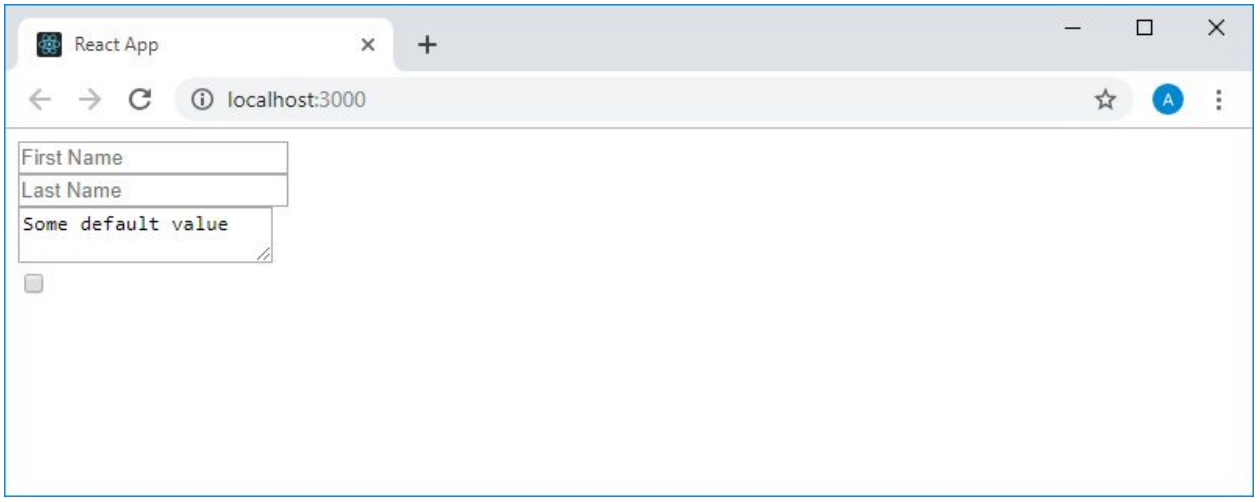


Текст, появившийся в поле

К работе с полем для ввода многострочного текста мы ещё вернёмся, а пока поговорим о флажках. Флажок — это элемент управления `input`, в качестве типа которого указан `checkbox`. Вот как выглядит его описание:

```
<input type="checkbox" />
```

Вот как выглядит флажок, описанный этой разметкой, на странице.



Флажок

Основной особенностью этого элемента управления является тот факт, что при работе с ним не используется атрибут `value`. Его применяют для того, чтобы предоставить пользователю выбор их неких двух вариантов, один из которых соответствует установленному флажку, а другой — снятому. Для отслеживания того, установлен флажок или снят, используется атрибут `checked`, который описывается логическим значением. В результате флажкам обычно соответствуют логические свойства, хранящиеся в состоянии.

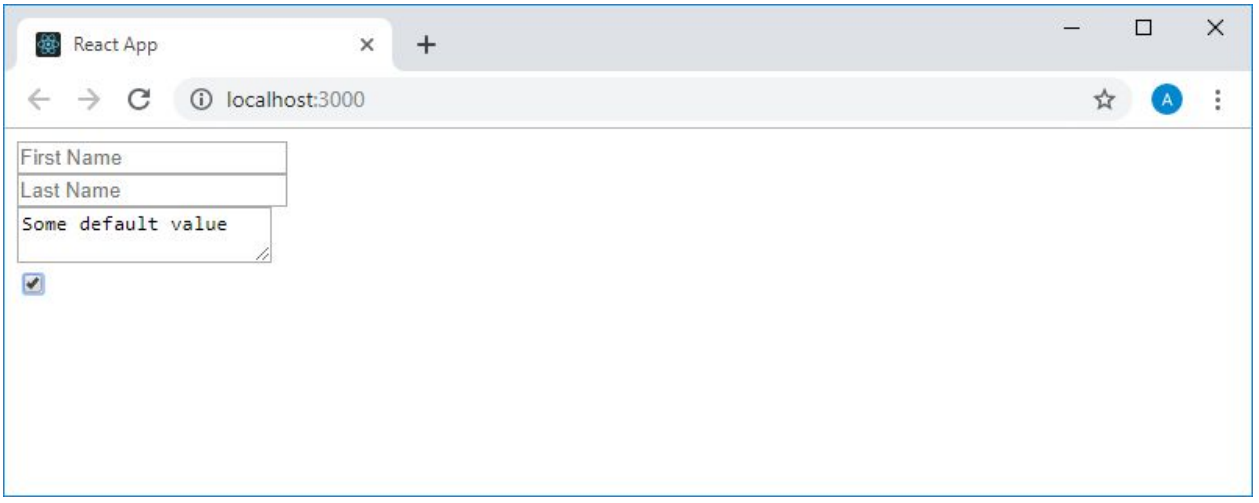
Приведём состояние компонента к такому виду:

```
this.state = {

    firstName: "",

    lastName: "",

    isFriendly: true

}
```

Код описания флажка изменим следующим образом:

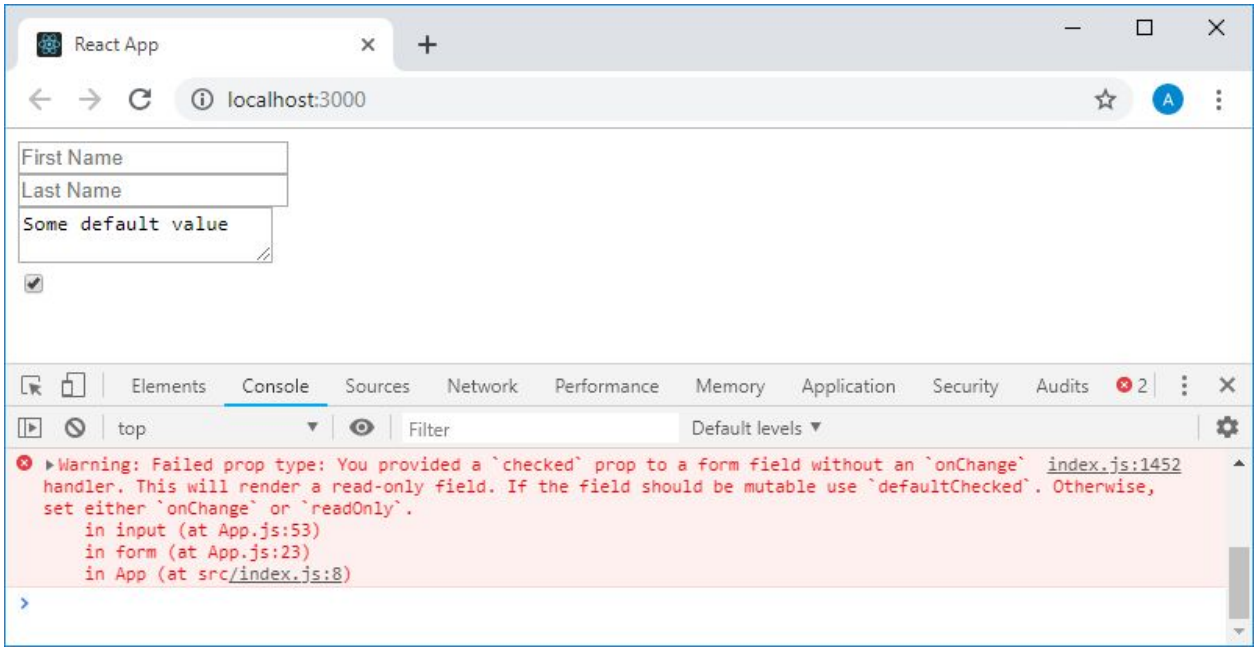
```
<input

    type="checkbox"

    checked={this.state.isFriendly}

/>
```

После этого на страницу будет выведен установленный флажок.



Установленный флажок

Правда, сейчас он не будет реагировать на щелчки по нему. Дело в том, что флажок привязан к соответствующей переменной, хранящейся в состоянии, в результате при попытке, в нашем случае, снять его, React, проверяя состояние, и обнаруживая, что свойство `isFriendly` установлено в `true`, не даёт этого сделать. При этом в консоль будет выводиться предупреждение о том, что мы не предусмотрели механизм изменения поля (обработчик события `onChange`) и оно выведено в состоянии «только для чтения».



Предупреждение в консоли

Мы вполне можем написать особый метод для работы с флажком, но в коде нашего компонента уже есть метод `handleChange()`. Сейчас он используется для работы с текстовыми полями. Подумаем о том, как воспользоваться им и для работы с флажком. Для этого сначала назначим вышеуказанный метод в качестве обработчика события `onChange` флажка и назначим флажку имя, соответствующее имени свойства состояния, относящегося к флажку. Кроме того, подпишем флажок, воспользовавшись тегом `label`:

```
<label>
     <input
        type="checkbox"
        name="isFriendly"
        checked={this.state.isFriendly}
        onChange={this.handleChange}
    /> Is friendly?
</label>
```

В методе `handleChange()`, код которого показан ниже, мы, при работе с текстовыми полями, выясняли имя элемента (`name`) и его содержимое (`value`), после чего обновляли состояние, записывая в него то, что было у поля с определённым именем в его атрибуте `value`:

```
handleChange(event) {
    const {name, value} = event.target
    this.setState({
        [name]: value
    })
```

```
}
```

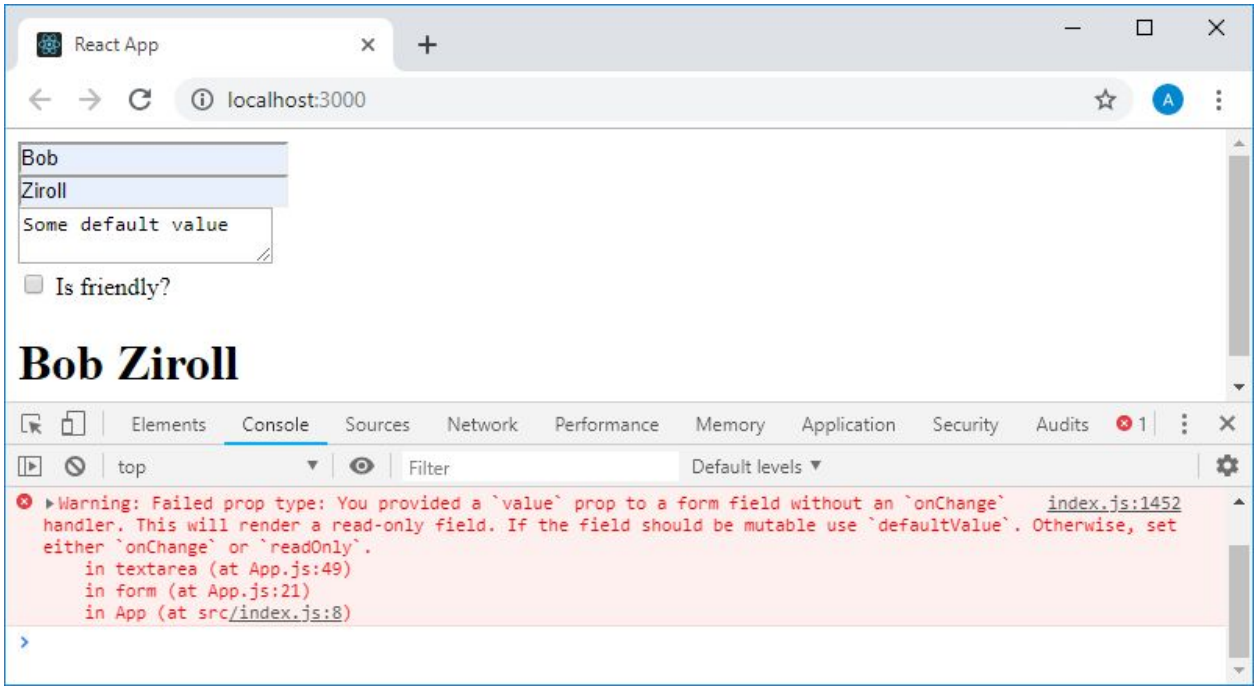
Теперь нам нужно разобраться с тем, как быть с флажком, атрибута `value` у которого нет. У него есть лишь атрибут `checked`, который может принимать только значения `true` или `false`. В результате нам, для того чтобы использовать метод `handleChange()` для работы с флажком, нужно проверить, является ли элемент, для которого вызван этот обработчик, флажком. Для того чтобы выполнить эту проверку — вспомним, что в качестве типа (`type`) элемента `input`, представляющего флажок, задано значение `checkbox`. Для того чтобы проверить это значение, можно обратиться к свойству `type` элемента `event.target`. Извлечём это свойство из `event.target`, а также — свойство `checked`, воспользовавшись следующей конструкцией:

```
const {name, value, type, checked} = event.target
```

Теперь мы можем проверить значение константы `type` и выяснить, является ли элемент, для которого вызван обработчик события, флажком. Если это так — мы запишем в состояние то, что оказалось в константе `checked`. Не забудем при этом сохранить код, ответственный за работу с текстовыми полями. В результате код `handleChange()` приобретёт следующий вид:

```
handleChange(event) {

    const {name, value, type, checked} = event.target

    type === "checkbox" ? this.setState({ [name]: checked }) : this.setState({
[name]: value })

}
```

После этого проверим работу флажка.



Проверка работы флажка

Как видно, теперь его можно снимать и устанавливать. При этом работа текстовых полей не нарушена. Из консоли исчезло уведомление, касающееся флажка, но там выводится уведомление, касающееся поля для ввода многострочного текста. Изменим код, описывающий это поле, следующим образом:

```
<textarea
```

```
    value={"Some default value"}
    onChange={this.handleChange}
/>
```

Это приведёт к исчезновению уведомления, хотя другие механизмы, позволяющие работать с этим полем средствами компонента, мы не реализовали (не указали имя для поля, не добавили в состояние соответствующее свойство). Вы можете реализовать эти возможности самостоятельно. Теперь поговорим о переключателях.

Их можно представить в виде комбинации элементов `input` типов `text` и `checkbox`. Здесь имеется в виду то, что у переключателей есть и атрибут `value`, и атрибут `checked`. Добавим в нашу форму пару переключателей, создав их код на основе кода описания флажка. Вот как это выглядит:

```
<label>
    <input
        type="radio"
        name="gender"
        value="male"
        checked={this.state.isFriendly}
        onChange={this.handleChange}
    /> Male
</label>
<br />
<label>
    <input
        type="radio"
        name="gender"
        value="female"
        checked={this.state.isFriendly}
        onChange={this.handleChange}
    /> Female
</label>
```

Мы создали этот код на основе кода описания флажка и кое-что ещё не отредактировали. Поэтому переключатели странно себя ведут. В частности, если флажок снят, то и тот и другой переключатели находятся в «выключенном» состоянии, а если установить флажок — один из них оказывается «включенным». Подобные ошибки можно предупредить, внимательно относясь к коду элементов в том случае, если он создаётся на основе кода уже существующих элементов. Сейчас мы это исправим.

Обратите внимание на то, что у этих двух элементов одно и то же имя — `gender`. Переключатели с одним и тем же именем формируют группу. Выбранным может быть лишь один переключатель, входящий в такую группу.

При настройке переключателей нельзя просто указать на то, что их значение `checked` устанавливается, скажем, в `true`, если некое свойство состояния равняется `true`. Переключатели должны поддерживать синхронизированное, в пределах группы, изменение собственного состояния. Вместо этого значение `checked` переключателей устанавливается по условию. Это условие в нашем случае будет представлено сравнением свойства состояния `this.state.gender` со строкой `male` или `female`. В коде описания переключателей это выглядит так:

```
<label>

    <input

        type="radio"

        name="gender"

        value="male"

        checked={this.state.gender === "male"}

        onChange={this.handleChange}

    /> Male

</label>

<br />

<label>

    <input

        type="radio"

        name="gender"

        value="female"

        checked={this.state.gender === "female"}

        onChange={this.handleChange}

    /> Female

</label>
```

Теперь добавим в состояние новое свойство, `gender`, инициализировав его пустой строкой:

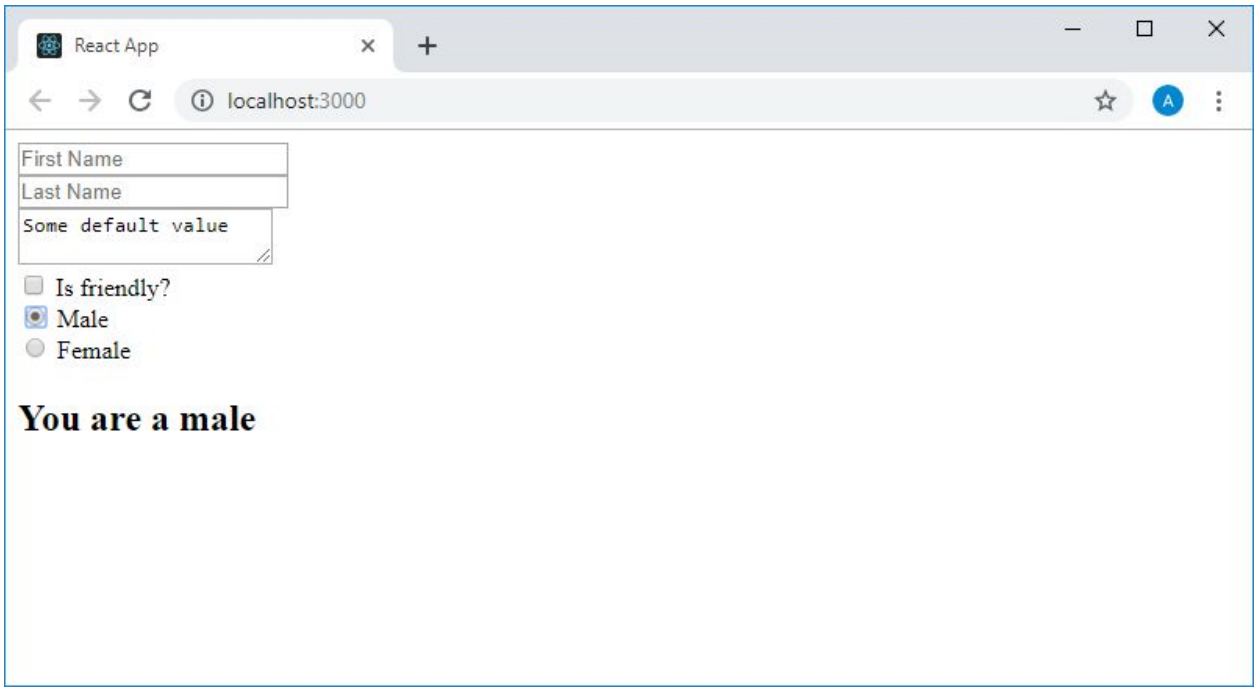
```
this.state = {

    firstName: "",

    lastName: "",

    isFriendly: false,
```

```
    gender: ""
}
```

После этого переключатели будут работать независимо от флажка. Добавим в код, выводимый компонентом, заголовок второго уровня, выводящий сведения о том, какой именно переключатель выбран:

```
<h2>You are a {this.state.gender}</h2>
```

Тут, вероятно, стоит внедрить некий механизм условного рендеринга. Это позволит, при открытии страницы, когда ни один из переключателей не выбран, сделать так, чтобы на ней не выводился бы текст `You are a`, но мы этого делать не будем, хотя вы вполне можете реализовать это самостоятельно. Взглянем теперь на то, что у нас получилось.



Переключатели на странице приложения

Всё то, о чём мы тут говорили, может показаться достаточно сложным. В частности, это касается запоминания особенностей разных элементов управления. Для того чтобы упростить работу с формами, можно применять специализированные библиотеки. Например — библиотеку formik. Эта библиотека значительно упрощает процесс разработки форм в React-приложениях.

Теперь поговорим о полях со списками.

В обычном HTML при описании полей со списком используются такие конструкции:

```
<select>
    <option></option>
    <option></option>
    <option></option>
<select/>
```

В React применяется похожий подход, хотя, как и в случае с другими элементами, используется атрибут `value`. Это позволяет легко выяснить то, какой именно элемент списка выбран, и, кроме того, это облегчает работу с состоянием компонента.

Предположим, мы хотим создать поле со списком, позволяющее пользователю выбирать его любимый цвет. Для этого в атрибут `value` элемента `select` можно поместить следующую конструкцию: `value={this.state.favColor}`. Сюда будут попадать те значения, которые будет выбирать пользователь. Теперь добавим `favColor` в состояние:

```
this.state = {

    firstName: "",

    lastName: "",

    isFriendly: false,

    gender: "",

    favColor: "blue"

}
```

Далее, оснастим поле со списком обработчиком события `onChange` и дадим ему имя. Также назначим значения `value` элементам `options` поля со списком и введём текст, который будет выводиться в поле.

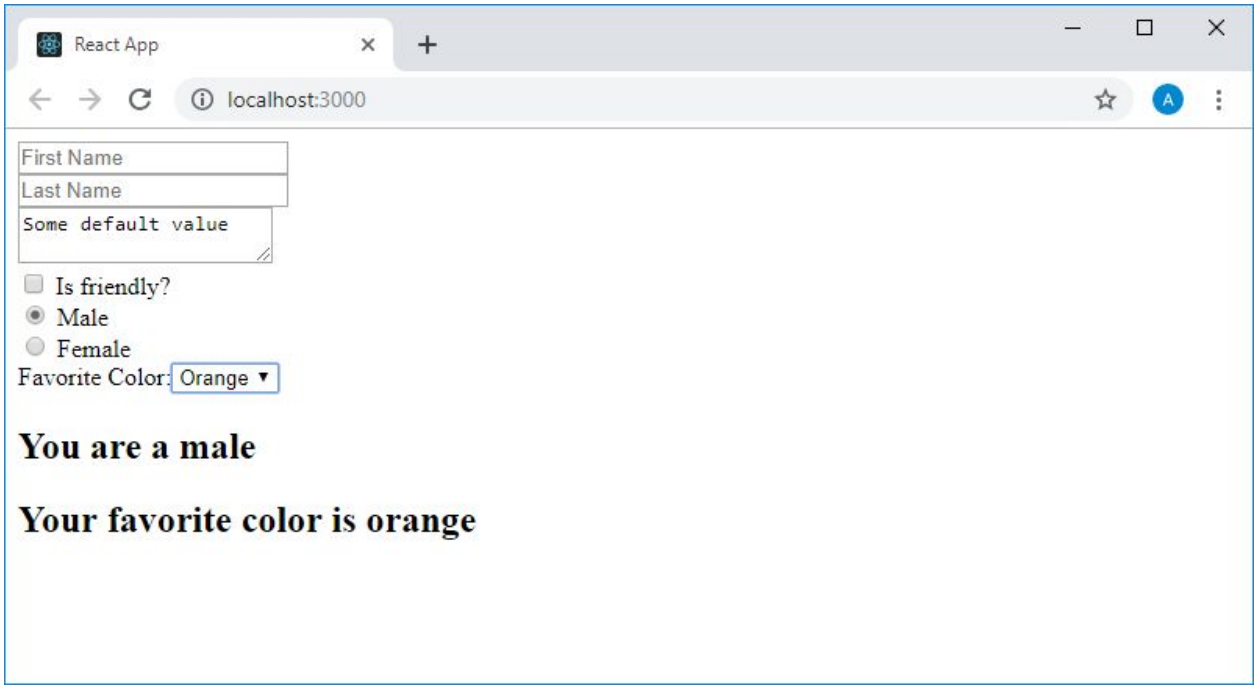
Вот как выглядит настроенный элемент `select` с подписью:

```
<label>Favorite Color:</label>

<select

    value={this.state.favColor}

    onChange={this.handleChange}

    name="favColor"

>

    <option value="blue">Blue</option>

    <option value="green">Green</option>

    <option value="red">Red</option>

    <option value="orange">Orange</option>

    <option value="yellow">Yellow</option>

</select>
```

Теперь добавим в форму ещё одну надпись, выводящую любимый цвет пользователя:

```
<h2>Your favorite color is {this.state.favColor}</h2>
```

Пришло время испытать поле со списком.



Поле со списком

Как видно, хотя наша форма и не блещет изысками дизайна, элементы управления, размещённые на ней, работают так, как ожидается.

Благодаря тому, как в React организованы API элементов управления, несложно сделать так, чтобы для обработки их событий применялся бы один и тот же обработчик. Именно такая схема работы используется и в нашем случае. Единственная особенность нашего обработчика `handleChange()` заключается в том, что нам приходится по-особенному обрабатывать события флажка.

Теперь поговорим об отправке формы или об обработке введённых в неё значений после завершения её заполнения. Можно выделить два подхода к выполнению подобных действий. При применении любого из них форму стоит оснастить кнопкой:

```
<button>Submit</button>
```

В HTML5, если в форме будет найден элемент `button`, он будет действовать как старый элемент `input` с типом `submit`. Если по этой кнопке щёлкнуть — будет вызвано событие самой формы `onSubmit`. Если нужно что-то сделать после завершения заполнения формы, можно добавить обработчик события `onClick` к кнопке, но, например, лично я предпочитаю обрабатывать подобные события на уровне формы, назначая ей обработчик события `onSubmit`:

```
<form onSubmit={this.handleSubmit}>
```

Метод, используемый в качестве обработчика этого события, ещё не написан. Это — обычный обработчик события, который, например, обращаясь к некоему API, передаёт ему данные формы.

Учебный курс по React, часть 25: практикум по работе с формами

Занятие 43. Практикум. Работа с формами

Оригинал

Задание

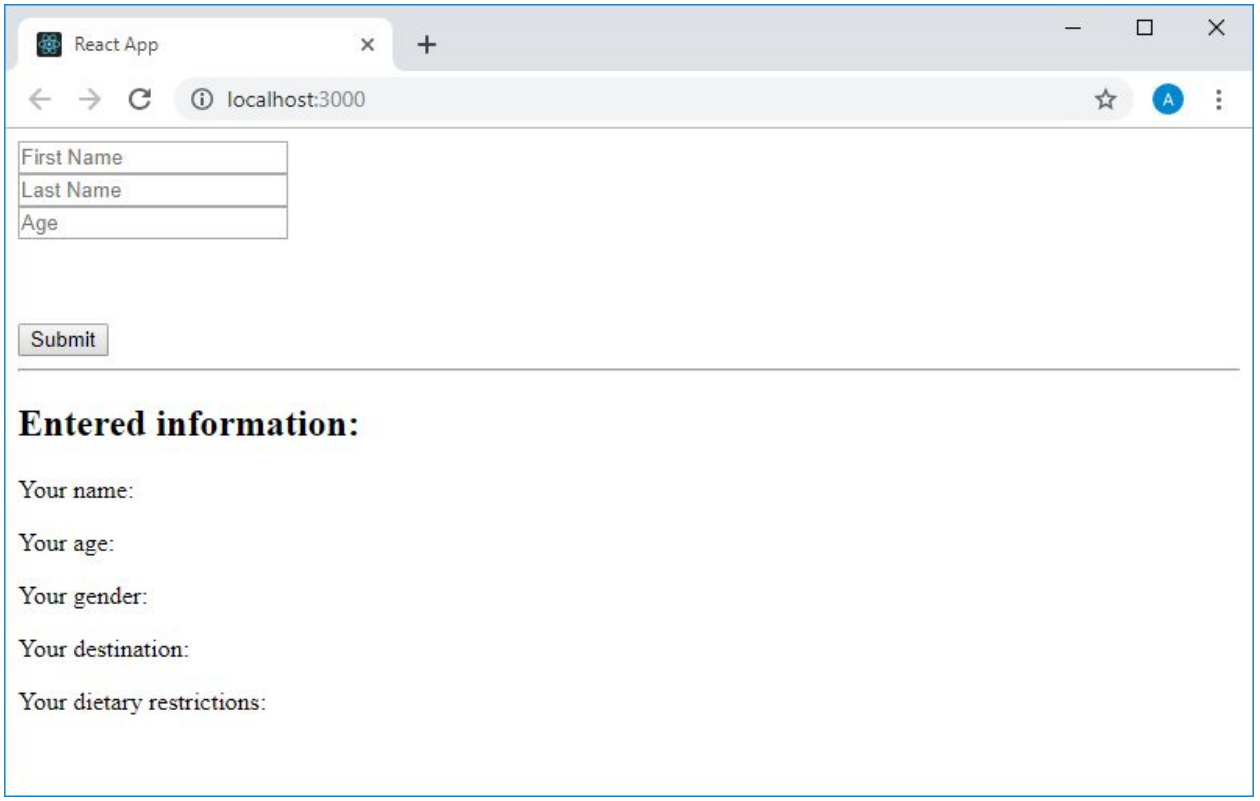
На этом практическом занятии вам предлагается привести в работоспособное состояние код компонента `App`, который находится в файле `App.js` стандартного проекта, создаваемого средствами create-react-app. Вот этот код:

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


class App extends Component {

    constructor() {

        super()

        this.state = {}

    }


    render() {

        return (

            <main>

                <form>

                    <input placeholder="First Name" /><br />

                    <input placeholder="Last Name" /><br />

                    <input placeholder="Age" /><br />


                    {/* Здесь создайте переключатели для выбора пола */}

                    <br />


                    {/* Здесь создайте поле со списком для выбора пункта
назначения */}

                    <br />


                    {/* Здесь создайте флажки для указания диетологических
ограничений */}

                    <br />


                    <button>Submit</button>
```

```
                </form>

                <hr />

                <h2>Entered information:</h2>

                <p>Your name: {/* Имя и фамилия */}</p>

                <p>Your age: {/* Возраст */}</p>

                <p>Your gender: {/* Пол */}</p>

                <p>Your destination: {/* Пункт назначения */}</p>

                <p>

                    Your dietary restrictions:

                    {/* Список диетологических ограничений */}

                </p>

            </main>

        )

    }

}


export default App
```

В целом ваша задача сводится к тому, чтобы данные, которые пользователь вводит, работая с элементами управления формы, тут же появлялись бы тексте, расположенном ниже этой формы. В ходе выполнения задания воспользуйтесь технологией управляемых компонентов. Надо отметить, что предлагаемая вам задача является адаптированной версией этого задания, поэтому вы вполне можете взглянуть на него для того, чтобы лучше понять особенности элементов управления, которые вам предлагается создать и настроить.

Вот как выглядит то, что компонент выводит сейчас на экран.



Приложение в браузере

Решение
К решению предлагаемой вам задачи можно подойти с разных сторон. Мы начнём с того, что внесём всё, что нам понадобится, в состояние, после чего займёмся настройкой элементов управления и других механизмов компонента.

В данный момент состояние компонента будет выглядеть так, как показано ниже.

```
this.state = {

    firstName: "",

    lastName: "",

    age: 0,

    gender: "",

    destination: "",

    dietaryRestrictions: []

}
```

Нужно учитывать то, что в процессе работы над программой может выясниться, что, например, состояние будет удобнее инициализировать иначе. Если мы столкнёмся с чем-то подобным — мы изменим код инициализации состояния. В частности, сейчас некоторые сомнения может вызывать число 0, записанное в свойство `age`, в котором предполагается хранить возраст, введённый пользователем. Возможно, иначе нужно будет поступить и с системой хранения данных флажков, которая сейчас представлена свойством `dietaryRestrictions`, инициализированным пустым массивом.

Теперь, после инициализации состояния, займёмся настройкой элементов управления. Так как в коде уже есть описание полей ввода — приступим к работе с ними.

Этим элементам управления понадобится дать имена, настроив их атрибуты `name` таким образом, чтобы они совпадали с именами свойств состояния, в которых будут храниться данные, введённые в эти поля. У них должен быть атрибут `value`, значение которого определяется на основе данных, хранящихся в состоянии. При вводе данных в каждое из этих полей нужно передавать введённые данные компоненту, что приводит к необходимости наличия у них обработчика события `onChange`. Все эти рассуждения приводят к тому, что описание полей выглядит теперь так:

```
<input

    name="firstName"

    value={this.state.firstName}

    onChange={this.handleChange}

    placeholder="First Name"

/>

<br />


<input

    name="lastName"

    value={this.state.lastName}

    onChange={this.handleChange}

    placeholder="Last Name"

/>

<br />


<input

    name="age"

    value={this.state.age}

    onChange={this.handleChange}

    placeholder="Age"

/>
```

В качестве метода, используемого для обработки событий `onChange` этих полей, указан несуществующий пока `this.handleChange`. Создадим этот метод:

```
handleChange(event) {
```

```
    const {name, value} = event.target

    this.setState({

        [name]: value

    })

}
```

Здесь мы извлекаем из объекта `event.target` свойства `name` и `value`, после чего используем их для установки соответствующего свойства состояния. В данный момент нас такой код универсального обработчика событий устроит, но позже, когда мы доберёмся до работы с флажками, мы внесём в него изменения.

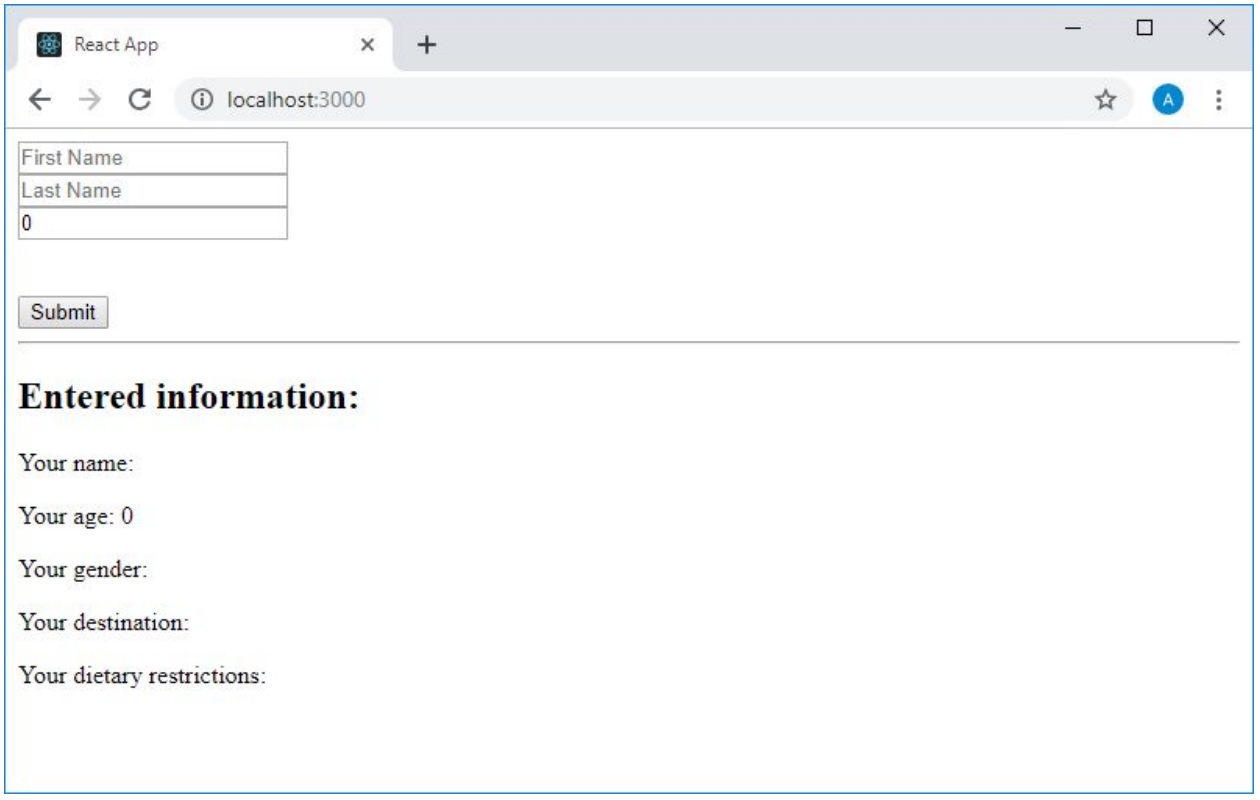
Не забудем о привязке `this`, выполняемой в конструкторе компонента:

```
this.handleChange = this.handleChange.bind(this)
```

Для того чтобы добиться вывода в нижней части страницы данных, вводимых в поля `firstName`, `secondName` и `age`, поработаем с соответствующими элементами `<p>`, приведя их к следующему виду:

```
<p>Your name: {this.state.firstName} {this.state.lastName}</p>

<p>Your age: {this.state.age}</p>
```

Теперь взглянем на то, что у нас получилось.



Приложение в браузере

Как видно, в поле, предназначенном для ввода возраста, не выводится подсказка. Вместо этого выводится то, что задано в свойстве состояния `age`, то есть — 0. Нам же нужно, чтобы в незаполненном поле выводилась бы подсказка. Попробуем заменить значение `age` в состоянии на

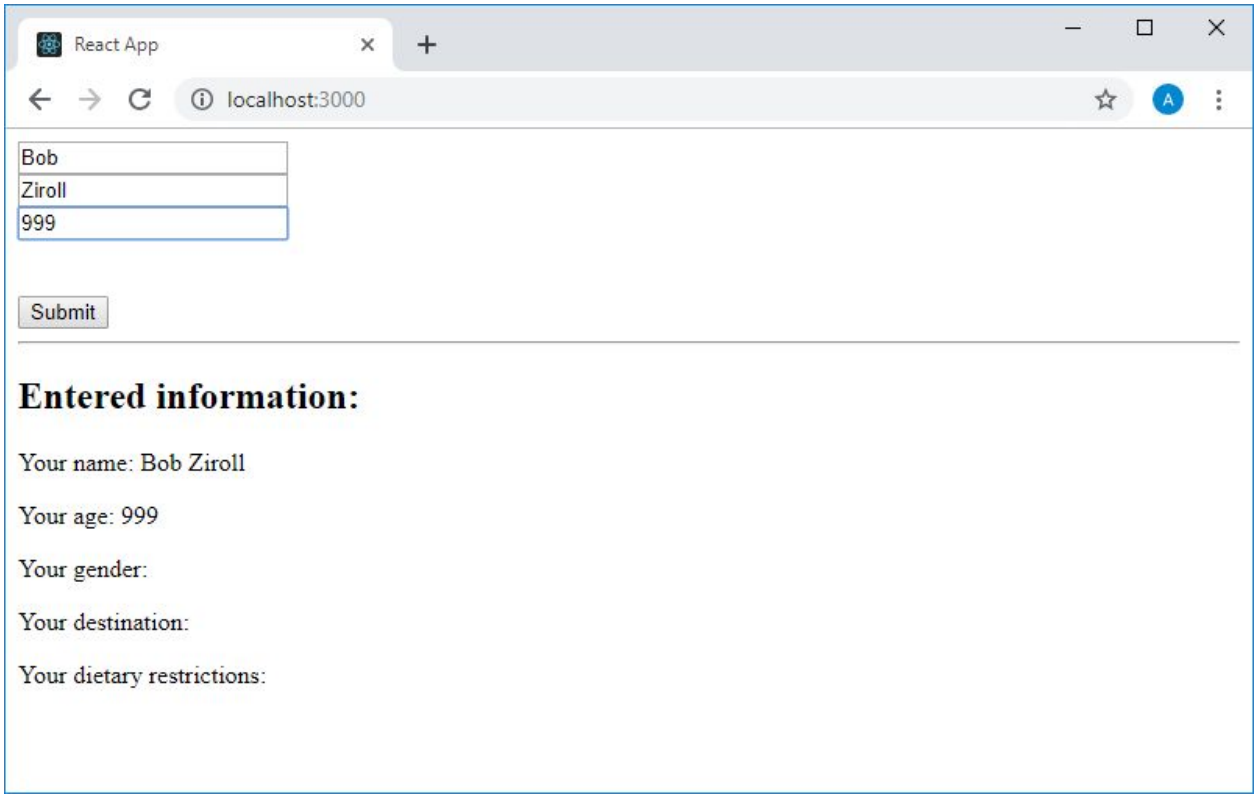
`null`. После этого оказывается, что форма выглядит так, как нужно, но в консоли выводится следующее предупреждение, касающееся поля `age`:

```
Warning: `value` prop on `input` should not be null. Consider using an empty string to clear the component or `undefined` for uncontrolled components
```

В результате нам нужно будет заменить значение свойства состояния `age` на пустую строку, приведя код инициализации состояния к следующему виду:

```
this.state = {

    firstName: "",

    lastName: "",

    age: "",

    gender: "",

    destination: "",

    dietaryRestrictions: []

}
```

Теперь испытаем форму. Сразу после открытия она будет выглядеть так же, как и в самом начале работы, то есть — в поле `age` вернётся подсказка. При заполнении полей введённые данные будут выводиться в нижней части страницы.



Приложение в браузере

Как видно, на данном этапе работы всё функционирует так, как ожидается.

Теперь займёмся новыми элементами. Следующим шагом работы над формой будет добавление на неё переключателей.

Обернём переключатели в тег `<label>`, что позволит не только подписать переключатель, но и сделать так, чтобы щелчок по этой подписи, то есть — по его родительскому элементу, приводил бы к его выбору.

Работая с переключателями стоит помнить о том, что они представляют собой нечто вроде комбинации флажков, имеющих атрибут `checked`, и текстовых полей, у которых есть атрибут `value`. Переключатели формируют группу, в которой каждому из переключателей назначают одно и то же имя, при этом свойство переключателей `checked` устанавливается по условию, настроенному так, чтобы нельзя было бы включить больше одного переключателя, входящего в одну и ту же группу. В качестве обработчика событий `onChange` переключателей назначим `this.handleChange`.

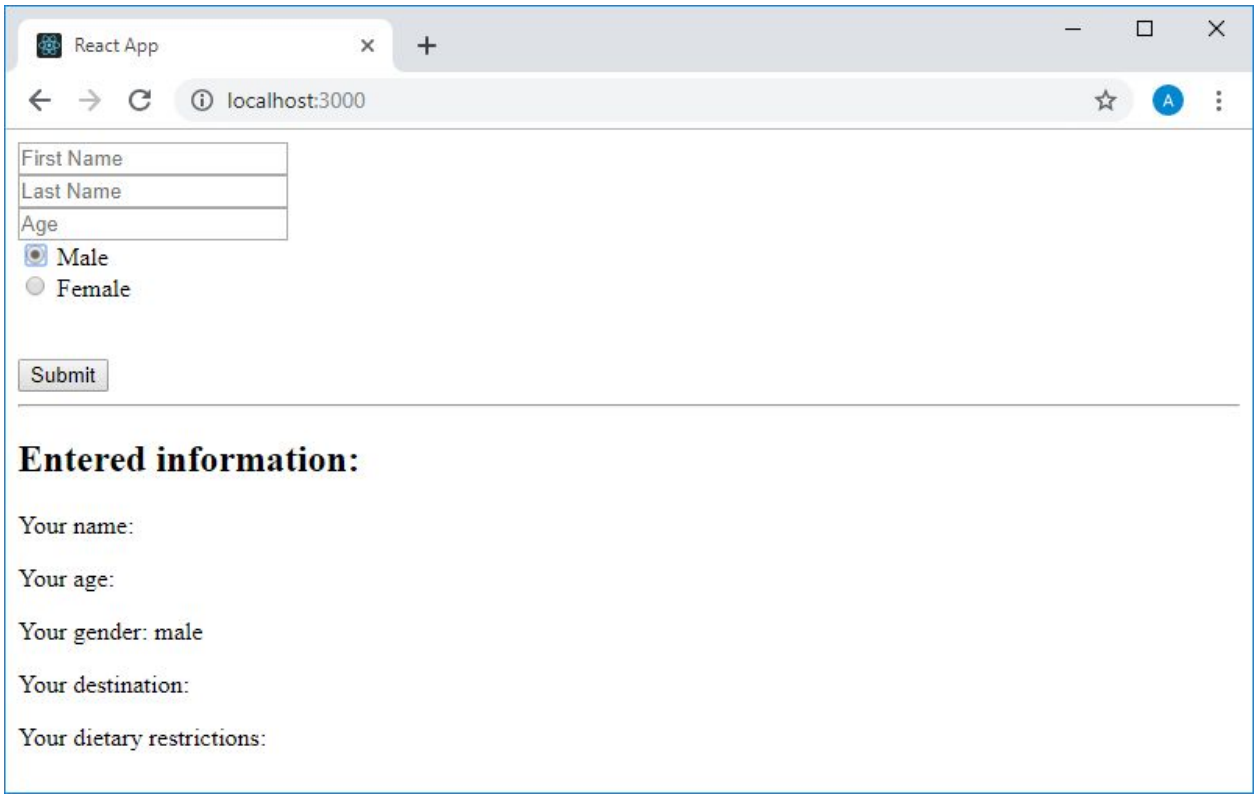
В результате код описания переключателей будет выглядеть так:

```
<label>

    <input

        type="radio"

        name="gender"

        value="male"

        checked={this.state.gender === "male"}

        onChange={this.handleChange}

    /> Male

</label>


<br />


<label>

    <input

        type="radio"

        name="gender"

        value="female"

        checked={this.state.gender === "female"}

        onChange={this.handleChange}

    /> Female

</label>
```

Теперь переработаем соответствующий элемент `<p>`, расположенный в нижней части страницы, следующим образом:

```
<p>Your gender: {this.state.gender}</p>
```

После этого форму можно испытать. Сразу после запуска оба переключателя оказываются невыбранными, так как в состоянии хранится значение, которое не позволяет ни одной из проверок, выполняемых при установке их свойства `checked`, выдать `true`. После щелчка по одному из них в состояние попадает соответствующее ему значение (хранящееся в атрибуте `value` переключателя), переключатель оказывается выбранным, соответствующий текст выводится в нижней части формы.



Приложение в браузере

Теперь займёмся работой над полем со списком. Его заготовка выглядит так:

```
<select>

    <option></option>

    <option></option>

    <option></option>

    <option></option>

</select>
```

Из этого кода видно, что мы планируем описать поле со списком, содержащим четыре элемента.

У тега `<select>` и у находящихся в нём тегов `<option>` есть атрибут `value`. Однако эти атрибуты несут различный смысл. То значение `value`, которое назначают элементу `<option>`, указывает на то, каким должно быть соответствующее свойство состояния при выборе данного элемента. Сюда попадают те строки, которые должны находиться в выпадающем списке. В нашем случае — это некие пункты назначения, например — страны. Запишем их названия с маленькой буквы для того, чтобы их внешний вид соответствовал бы значениям свойств `value` других имеющихся в коде элементов. После этого код поля со списком будет выглядеть так

```
<select value=>
```

```
    <option value="germany">Germany</option>
    <option value="norway">Norway</option>
    <option value="north pole">North Pole</option>
    <option value="south pole">South Pole</option>
</select>
```

Если же говорить об атрибуте `value` тега `<select>`, то тут будет указано не некое жёстко заданное значение, а ссылка на соответствующее свойство состояния:

```
<select value={this.state.destination}>
    <option value="germany">Germany</option>
    <option value="norway">Norway</option>
    <option value="north pole">North Pole</option>
    <option value="south pole">South Pole</option>
</select>
```

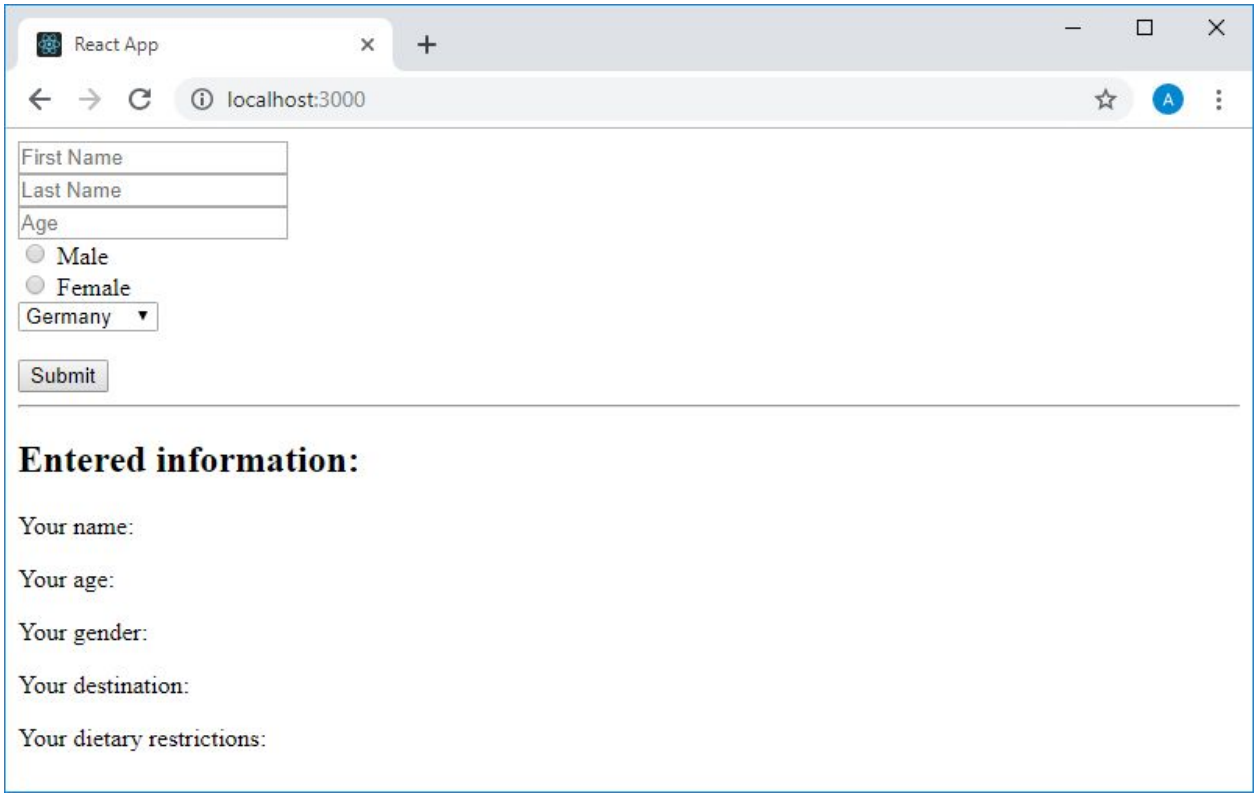
Назначим полю другие атрибуты. В частности — имя, соответствующее имени свойства в состоянии, и обработчик события `onChange` — `this.handleChange`.

```
<select
    value={this.state.destination}
    name="destination"
    onChange={this.handleChange}
>
    <option value="germany">Germany</option>
    <option value="norway">Norway</option>
    <option value="north pole">North Pole</option>
    <option value="south pole">South Pole</option>
</select>
```

Теперь настроим описание элемента `<p>`, в который будет выводиться то, что выбрано в поле `destination`:

```
<p>Your destination: {this.state.destination}</p>
```

Если прямо сейчас взглянуть на страницу в браузере, можно увидеть, что в поле автоматически выбран первый элемент списка, но это, очевидно, не приводит к обновлению состояния, так как в строке `Your destination:` после двоеточия ничего не выводится.



Приложение в браузере

Для того чтобы в состояние попало бы значение `germany`, нужно, открыв поле со списком, сначала выбрать что-нибудь другое, а потом уже — пункт `Germany`.

Часто для того чтобы учесть эту особенность полей со списками, в их списках, в качестве первого элемента, помещают нечто вроде элемента с пустым значением и с текстом наподобие `-- Please Choose a destination --`. В нашем случае это может выглядеть так:

```
<select

    value={this.state.destination}

    name="destination"

    onChange={this.handleChange}

>

    <option value="">-- Please Choose a destination --</option>

    <option value="germany">Germany</option>

    <option value="norway">Norway</option>

    <option value="north pole">North Pole</option>

    <option value="south pole">South Pole</option>

</select>
```

Мы остановимся именно на этом варианте настройки поля со списком.

Теперь займёмся, возможно, самой сложной частью нашей задачи, которая связана с флажками. Тут стоит пояснить то, что свойство состояния `dietaryRestrictions`, которое планируется использовать для работы с флажками, было инициализировано пустым массивом. Теперь, когда дело дошло до

работы с элементами управления, возникает такое ощущение, что лучше будет представить это поле в виде объекта. Так будет удобнее работать с сущностями, представляющими отдельные флажки в виде свойств этого объекта с понятными именами, а не в виде элементов массива. Свойства объекта, который теперь будет представлен свойством состояния `dietaryRestrictions`, будут содержать логические значения, указывающие на то, сброшен ли соответствующий флажок (`false`) или установлен (`true`). Теперь код инициализации состояния будет выглядеть так:

```
this.state = {

    firstName: "",

    lastName: "",

    age: "",

    gender: "",

    destination: "",

    dietaryRestrictions: {

        isVegan: false,

        isKosher: false,

        isLactoseFree: false

    }

}
```

Как видно, мы планируем создать три флажка. Все они, сразу после загрузки страницы, будут сброшены.

Опишем флажки в коде, возвращаемом компонентом, обернув их в теги `<label>` и настроив их атрибуты. Вот как будет выглядеть их код:

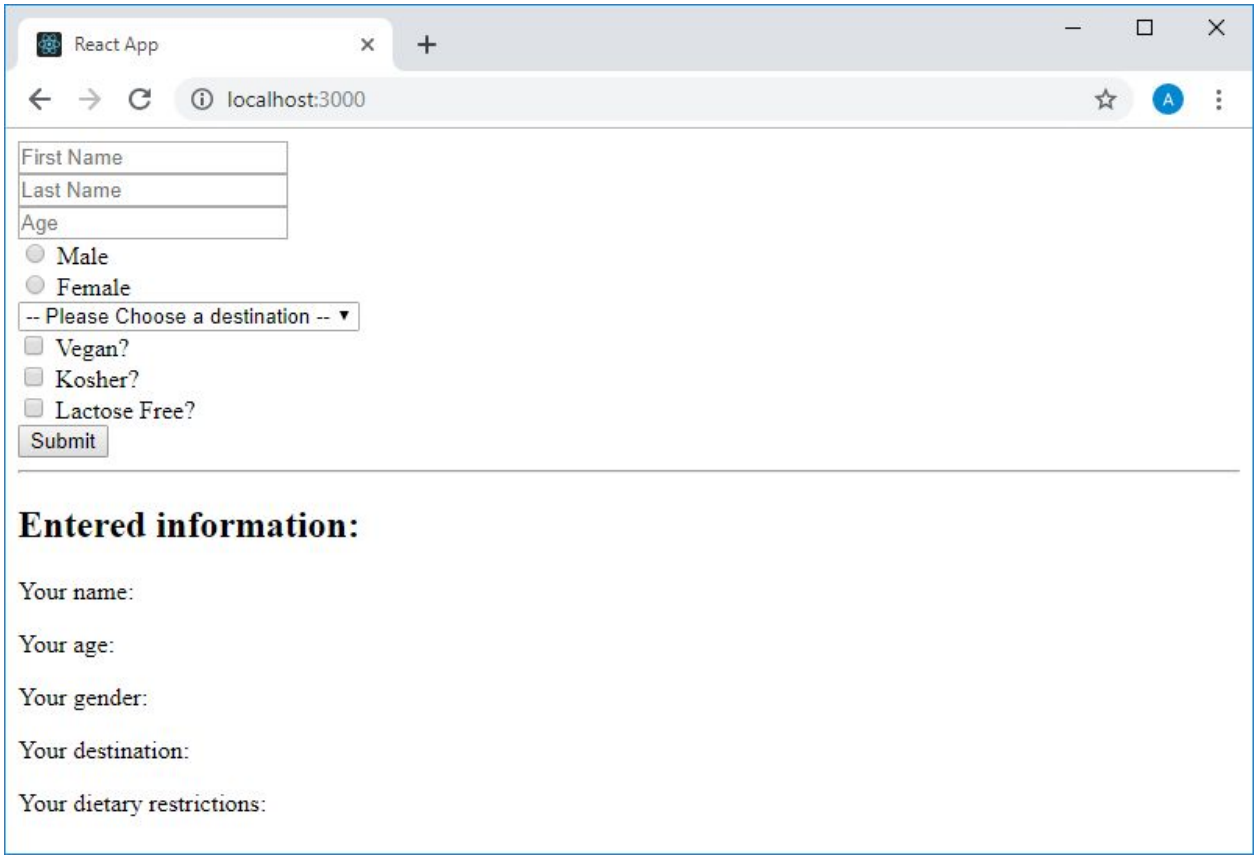
```
<label>

    <input

        type="checkbox"

        name="isVegan"

        onChange={this.handleChange}

        checked={this.state.dietaryRestrictions.isVegan}

    /> Vegan?

</label>

<br />



<label>
```

```
    <input
        type="checkbox"
        name="isKosher"
        onChange={this.handleChange}
        checked={this.state.dietaryRestrictions.isKosher}
    /> Kosher?
</label>
<br />

<label>
    <input
        type="checkbox"
        name="isLactoseFree"
        onChange={this.handleChange}
        checked={this.state.dietaryRestrictions.isLactoseFree}
    /> Lactose Free?
</label>
```

В качестве имён флажков использованы имена свойств объекта `dietaryRestrictions`, а в качестве значений их атрибутов `checked` — конструкции вида `this.state.dietaryRestrictions.isSomething`.

Обратите внимание на то, что, хотя в качестве обработчика событий флажков `onChange` указан уже имеющийся у нас обработчик `this.handleChange`, мы должны, для обеспечения правильной работы программы, внести в него некоторые изменения.

Взглянем на приложение.



Приложение в браузере

Как видно, флажки на страницу выведены, но компонент пока не содержит всех механизмов, необходимых для обеспечения их правильной работы. Займёмся обработчиком событий.

Здесь нам, для работы с флажками, понадобится извлечь из объекта `event.target`, в дополнение к уже извлечённым, свойства `type` и `checked`. Первое нужно для проверки типа элемента (флажки имеют тип, представленный строкой `checkbox`), второе — для того чтобы выяснить, установлен флажок или снят. Если оказывается, что обработчик был вызван после взаимодействия пользователя с флажком, используем особую процедуру установки состояния. События других элементов управления будем обрабатывать так же, как и прежде.

При обновлении состояния, нужно учитывать то, что React — система достаточно интеллектуальная, которая, если обновляют лишь часть состояния, автоматически скомбинирует в новом состоянии то, что осталось неизменным, с тем, что изменилось. Но при этом нельзя быть уверенным в том, что так же будет осуществляться и работа со свойствами объектов, которые представляют собой значения свойств состояния. Проверим это, приведя код `handleChange` к следующему виду. Тут мы исходим из предположения о том, что свойства объекта `dietaryRestrictions` вполне можно менять по одному:

```
handleChange(event) {

    const {name, value, type, checked} = event.target

    type === "checkbox" ?

        this.setState({

            dietaryRestrictions: {

                [name]: checked

            }
```

```
        })
    :
    this.setState({
        [name]: value
    })
}
```

Если открыть страницу приложения в браузере, то, сразу после её загрузки, всё будет выглядеть нормально, при попытке, например, ввести имя в поле `First Name`, всё тоже будет работать как и прежде, но при попытке установки одного из флажков будет выдано следующее предупреждение:

```
Warning: A component is changing a controlled input of type checkbox to be uncontrolled. Input elements should not switch from controlled to uncontrolled (or vice versa). Decide between using a controlled or uncontrolled input element for the lifetime of the component. More info: https://fb.me/react-controlled-components
```

Для того чтобы правильно обновлять содержимое объекта `dietaryRestrictions`, можно воспользоваться функциональной формой `setState`, самостоятельно сформировав новую версию состояния. Если бы нам надо было бы управлять большим количеством флажков, то, вероятно, мы так бы и сделали. Но тут мы поступим иначе. А именно, сделаем свойства объекта `dietaryRestrictions` свойствами состояния, избавившись от этого объекта:

```
this.state = {
    firstName: "",
    lastName: "",
    age: "",
    gender: "",
    destination: "",
    isVegan: false,
    isKosher: false,
    isLactoseFree: false
}
```

Теперь поменяем настройки атрибутов флажков, избавившись от `dietaryRestrictions`:

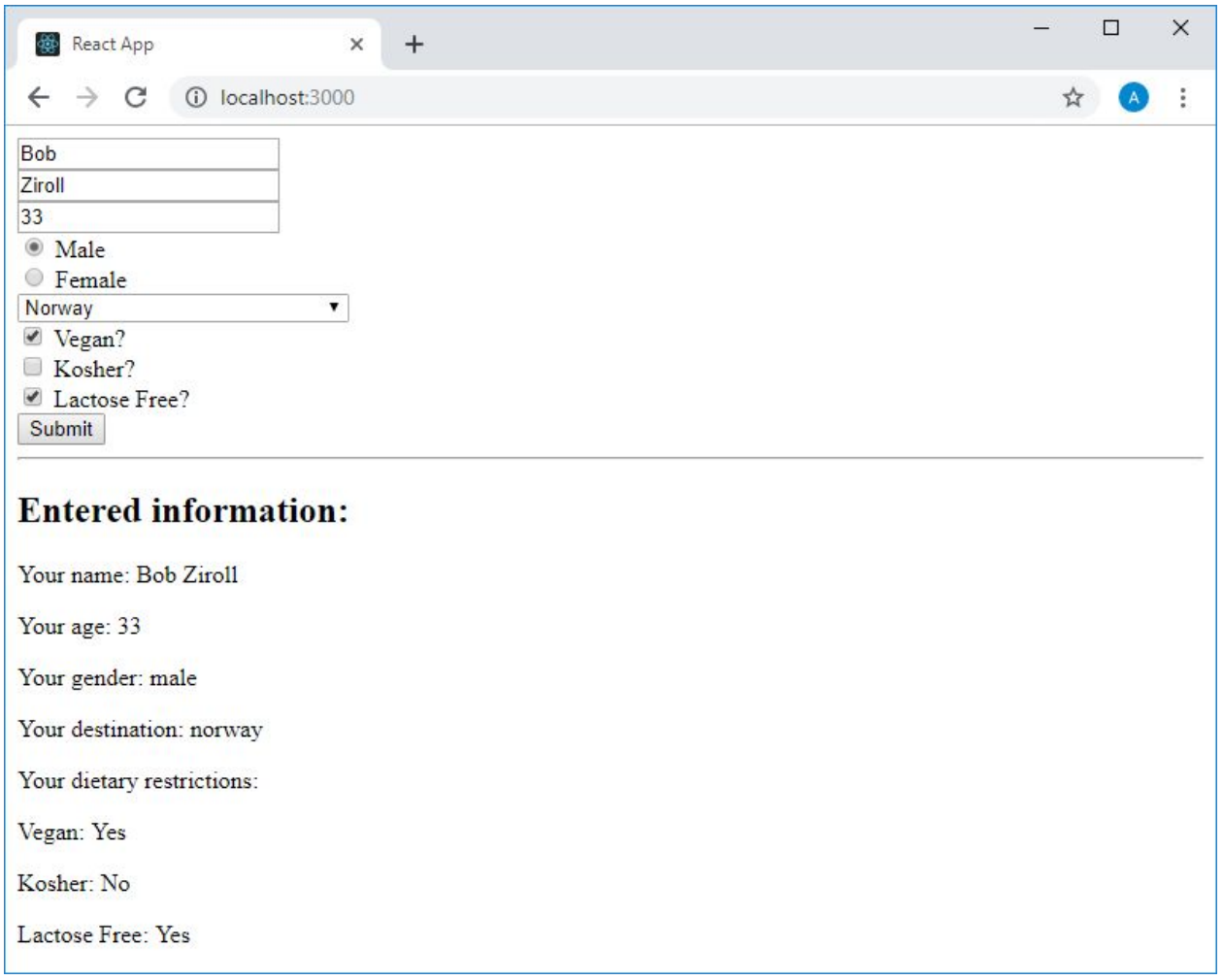
```
<label>
    <input
        type="checkbox"
        name="isVegan"
```

```
        onChange={this.handleChange}
        checked={this.state.isVegan}
    /> Vegan?
</label>
<br />

<label>
    <input
        type="checkbox"
        name="isKosher"
        onChange={this.handleChange}
        checked={this.state.isKosher}
    /> Kosher?
</label>
<br />

<label>
    <input
        type="checkbox"
        name="isLactoseFree"
        onChange={this.handleChange}
        checked={this.state.isLactoseFree}
    /> Lactose Free?
</label>
```

И, наконец, отредактируем код элемента, выводящего сведения о диетологических ограничениях, указанных пользователем:

```
<p>Your dietary restrictions:</p>

<p>Vegan: {this.state.isVegan ? "Yes" : "No"}</p>
<p>Kosher: {this.state.isKosher ? "Yes" : "No"}</p>
<p>Lactose Free: {this.state.isLactoseFree ? "Yes" : "No"}</p>
```

После этого проверим работоспособность приложения.



Приложение в браузере

Как видно, всё работает так, как ожидается.

Вот полный код компонента `App`:

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


class App extends Component {

    constructor() {

        super()

        this.state = {

            firstName: "",

            lastName: "",

            age: "",

            gender: "",

            destination: "",
```

```
        isVegan: false,
        isKosher: false,
        isLactoseFree: false
    }
    this.handleChange = this.handleChange.bind(this)
}

handleChange(event) {
    const {name, value, type, checked} = event.target
    type === "checkbox" ?
        this.setState({
            [name]: checked
        })
    :
    this.setState({
        [name]: value
    })
}

render() {
    return (
        <main>
            <form>
                <input
                    name="firstName"
                    value={this.state.firstName}
                    onChange={this.handleChange}
                    placeholder="First Name"
                />
                <br />
```

```
<input
    name="lastName"
    value={this.state.lastName}
    onChange={this.handleChange}
    placeholder="Last Name"
/>
<br />

<input
    name="age"
    value={this.state.age}
    onChange={this.handleChange}
    placeholder="Age"
/>
<br />

<label>
    <input
        type="radio"
        name="gender"
        value="male"
        checked={this.state.gender === "male"}
        onChange={this.handleChange}
    /> Male
</label>

<br />

<label>
```

```
                    <input
                        type="radio"
                        name="gender"
                        value="female"
                        checked={this.state.gender === "female"}
                        onChange={this.handleChange}
                    /> Female
                </label>

                <br />

                <select
                    value={this.state.destination}
                    name="destination"
                    onChange={this.handleChange}
                >
                    <option value="">-- Please Choose a destination
--</option>
                    <option value="germany">Germany</option>
                    <option value="norway">Norway</option>
                    <option value="north pole">North Pole</option>
                    <option value="south pole">South Pole</option>
                </select>

                <br />

                <label>
                    <input
                        type="checkbox"
                        name="isVegan"
                        onChange={this.handleChange}
```

```
                checked={this.state.isVegan}
            /> Vegan?
        </label>
        <br />

        <label>
            <input
                type="checkbox"
                name="isKosher"
                onChange={this.handleChange}
                checked={this.state.isKosher}
            /> Kosher?
        </label>
        <br />

        <label>
            <input
                type="checkbox"
                name="isLactoseFree"
                onChange={this.handleChange}
                checked={this.state.isLactoseFree}
            /> Lactose Free?
        </label>
        <br />

        <button>Submit</button>
    </form>
    <hr />
    <h2>Entered information:</h2>
    <p>Your name: {this.state.firstName} {this.state.lastName}</p>
```

```
                <p>Your age: {this.state.age}</p>

                <p>Your gender: {this.state.gender}</p>

                <p>Your destination: {this.state.destination}</p>

                <p>Your dietary restrictions:</p>


                <p>Vegan: {this.state.isVegan ? "Yes" : "No"}</p>

                <p>Kosher: {this.state.isKosher ? "Yes" : "No"}</p>

                <p>Lactose Free: {this.state.isLactoseFree ? "Yes" : "No"}</p>


            </main>

        )

    }

}


export default App
```

Учебный курс по React, часть 26: архитектура приложений, паттерн Container/Component

Занятие 44. Архитектура приложений, паттерн Container/Component

Оригинал

Иногда объём работы, за выполнение которой отвечает отдельный компонент, оказывается слишком большим, компоненту приходится решать слишком много задач. Использование паттерна Container/Component позволяет отделить логику функционирования приложения от логики формирования его визуального представления. Это позволяет улучшить структуру приложения, разделить ответственность за выполнение различных задач между разными компонентами.

На предыдущем практическом занятии мы создали огромный компонент, длина кода которого приближается к 150 строкам. Вот код, который у нас тогда получился:

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


class App extends Component {

    constructor() {

        super()

        this.state = {
```

```
        firstName: "",
        lastName: "",
        age: "",
        gender: "",
        destination: "",
        isVegan: false,
        isKosher: false,
        isLactoseFree: false
    }
    this.handleChange = this.handleChange.bind(this)
}

handleChange(event) {
    const {name, value, type, checked} = event.target
    type === "checkbox" ?
        this.setState({
            [name]: checked
        })
    :
    this.setState({
        [name]: value
    })
}

render() {
    return (
        <main>
            <form>
                <input
                    name="firstName"
```

```
    value={this.state.firstName}
    onChange={this.handleChange}
    placeholder="First Name"
/>
<br />

<input
    name="lastName"
    value={this.state.lastName}
    onChange={this.handleChange}
    placeholder="Last Name"
/>
<br />

<input
    name="age"
    value={this.state.age}
    onChange={this.handleChange}
    placeholder="Age"
/>
<br />

<label>
    <input
        type="radio"
        name="gender"
        value="male"
        checked={this.state.gender === "male"}
        onChange={this.handleChange}
    /> Male
```

```
                </label>

                <br />

                <label>
                    <input
                        type="radio"
                        name="gender"
                        value="female"
                        checked={this.state.gender === "female"}
                        onChange={this.handleChange}
                    /> Female
                </label>

                <br />

                <select
                    value={this.state.destination}
                    name="destination"
                    onChange={this.handleChange}
                >
                    <option value="">-- Please Choose a destination
--</option>
                    <option value="germany">Germany</option>
                    <option value="norway">Norway</option>
                    <option value="north pole">North Pole</option>
                    <option value="south pole">South Pole</option>
                </select>

                <br />
```

```
<label>
    <input
        type="checkbox"
        name="isVegan"
        onChange={this.handleChange}
        checked={this.state.isVegan}
    /> Vegan?
</label>
<br />

<label>
    <input
        type="checkbox"
        name="isKosher"
        onChange={this.handleChange}
        checked={this.state.isKosher}
    /> Kosher?
</label>
<br />

<label>
    <input
        type="checkbox"
        name="isLactoseFree"
        onChange={this.handleChange}
        checked={this.state.isLactoseFree}
    /> Lactose Free?
</label>
<br />
```

```
                    <button>Submit</button>
                </form>
                <hr />
                <h2>Entered information:</h2>
                <p>Your name: {this.state.firstName} {this.state.lastName}</p>
                <p>Your age: {this.state.age}</p>
                <p>Your gender: {this.state.gender}</p>
                <p>Your destination: {this.state.destination}</p>
                <p>Your dietary restrictions:</p>

                <p>Vegan: {this.state.isVegan ? "Yes" : "No"}</p>
                <p>Kosher: {this.state.isKosher ? "Yes" : "No"}</p>
                <p>Lactose Free: {this.state.isLactoseFree ? "Yes" : "No"}</p>

            </main>
        )
    }
}

export default App
```

Первый недостаток этого кода, который сразу же бросается в глаза, заключается в том, что при работе с ним его постоянно приходится прокручивать в окне редактора. Можно заметить, что основной объём этого кода составляет логика формирования интерфейса приложения, содержимое метода `render()`. Кроме того, некоторый объём кода отвечает за инициализацию состояния компонента. В компоненте есть и то, что называется «business logic» (то есть то, что реализует логику функционирования приложения). Это — код метода `handleChange()`.

По результатам некоторых исследований известно, что возможность программиста воспринимать код, на который он смотрит, сильно ухудшается в том случае, если код это достаточно длинный, и программисту приходится, чтобы просмотреть его целиком, пользоваться прокруткой. Я заметил это и в ходе проведения занятий. Когда код, о котором я рассказываю, оказывается достаточно длинным, и мне постоянно приходится его прокручивать, учащимся становится сложнее его воспринимать.

Хорошо было бы, если бы мы переработали наш код, разделив между разными компонентами ответственность за формирование интерфейса приложения (то, что сейчас описано в методе `render()`) и за реализацию логики функционирования приложения, то есть, по определению того, как должен выглядеть его интерфейс (соответствующий код сейчас представлен конструктором компонента, в котором инициализируется состояние, и обработчиком событий элементов управления

`handleChange()`). При использовании подобного подхода к проектированию приложений мы, фактически, работаем с двумя видами компонентов, при этом надо отметить, что можно столкнуться с разными названиями подобных компонентов.

Мы будем пользоваться здесь паттерном Container/Component. При его использовании приложения строят, разделяя компоненты на два вида — на компоненты-контейнеры (к ним относится слово Container в названии паттерна), и на презентационные компоненты (это — Component в названии паттерна). Иногда компоненты-контейнеры называют «умными» (smart) компонентами, или просто «контейнерами», а презентационные — «глупыми» (dumb) компонентами, или просто «компонентами». Есть и другие наименования этих видов компонентов, и, надо отметить, смысл, который вкладывается в эти наименования, может, от случая к случаю, отличаться определёнными особенностями. В целом же, общая идея рассматриваемого подхода заключается в том, что у нас есть компонент-контейнер, ответственный за хранение состояния и содержащий методы для управления состоянием, а логика формирования интерфейса передаётся другому — презентационному компоненту. Этот компонент отвечает лишь за получение от компонента-контейнера свойств и за правильное формирование интерфейса. Вот материал Дэна Абрамова, в котором он исследует эту идею.

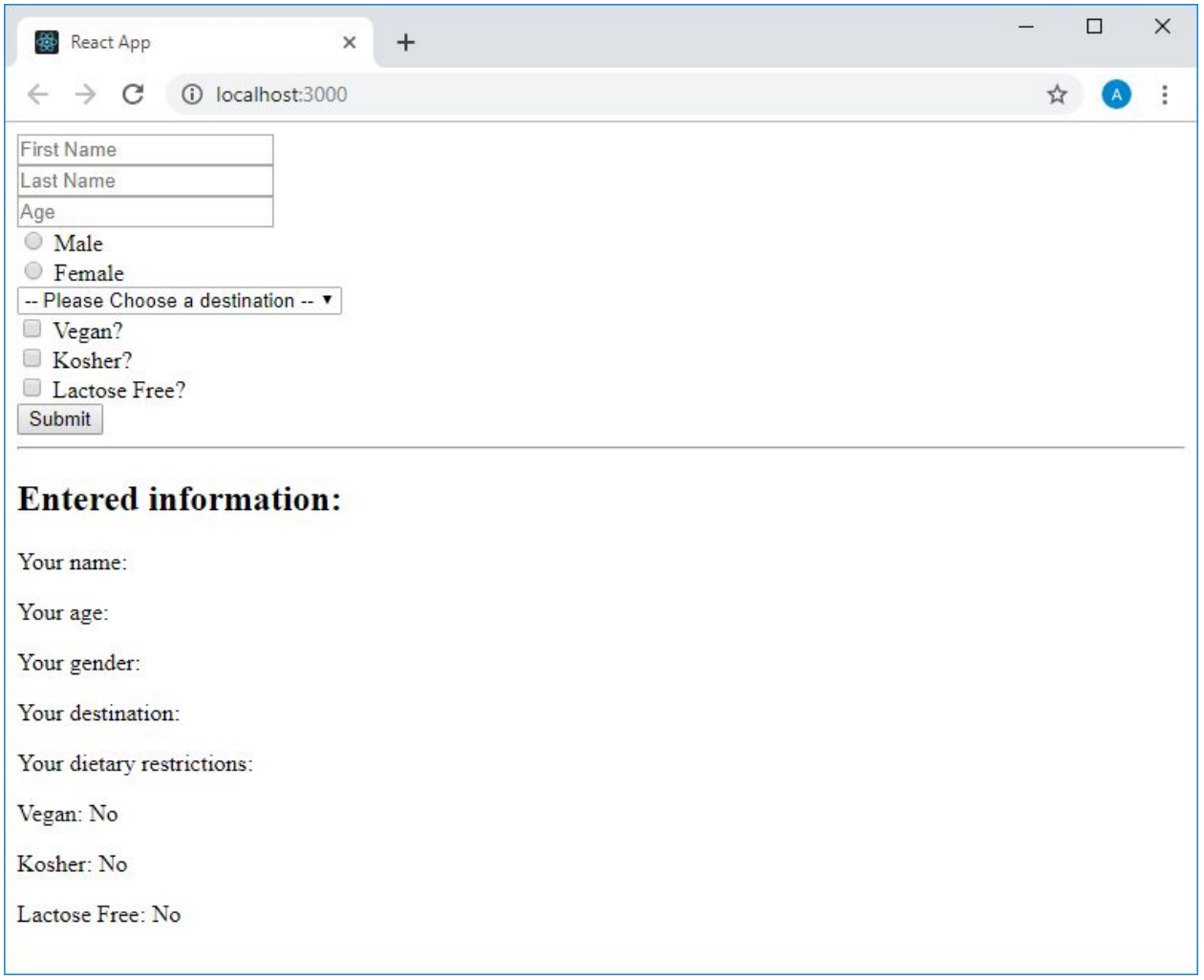
Преобразуем код нашего приложения в соответствии с паттерном Container/Component.

Для начала обратим внимание на то, что сейчас всё в приложении собрано в единственном компоненте `App`. Это приложение устроено так ради максимального упрощения его структуры, но в реальных проектах компоненту `App` вряд ли имеет смысл передавать задачу рендеринга формы и включать в него код, предназначенный для организации работы внутренних механизмов этой формы.

Добавим, в ту же папку, в которой находится файл `App.js`, файл `Form.js`, в котором будет находиться код нового компонента. Перенесём в этот файл весь код из компонента `App`, а компонент `App`, представленный сейчас компонентом, который основан на классе, преобразуем в функциональный компонент, основной задачей которого будет вывод компонента `Form`. Не забудем импортировать компонент `Form` в компонент `App`. В результате код компонента `App` будет выглядеть так:

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

import Form from "./Form"


function App() {

    return (

        <Form />

    )

}


export default App
```

Вот как выглядит на данном этапе работы то, что приложение выводит на экран.



Приложение в браузере

На предыдущих занятиях я говорил вам о том, что предпочитаю, чтобы компонент `App` представлял собой нечто вроде «оглавления» приложения, в котором указано, в каком порядке на страницу выводятся её разделы, представленные другими компонентами, которым делегированы задачи по рендерингу крупных фрагментов приложения.

Мы немного улучшили структуру приложения, но основную проблему, выражающуюся в том, что на один компонент возлагается слишком большая ответственность, пока не решили. Мы просто перенесли всё, что раньше было в компоненте `App`, в компонент `Form`. Поэтому теперь займёмся решением этой проблемы. Для этого создадим, в той же папке, в которой находятся файлы `Form.js` и `App.js`, ещё один файл — `FormComponent.js`. Этот файл будет представлять презентационный компонент, ответственный за визуализацию формы. На самом деле, назвать его можно и по-другому, можно и иначе структурировать файлы компонентов, всё зависит от нужд и масштабов конкретного проекта. Файл `Form.js` будет содержать логику функционирования формы, то есть — код компонента-контейнера. Поэтому переименуем его в `FormContainer.js` и поменяем команду импорта в коде компонента `App`, приведя её к такому виду:

```
import Form from "./FormContainer"
```

Можно ещё и переименовать компонент `Form` в `FormContainer`, но мы этого делать не будем. Теперь перенесём код, ответственный за рендеринг формы, из файла `FormContainer.js` в файл `FormComponent.js`.

Компонент `FormComponent` будет функциональным. Вот как будет выглядеть его код на данном этапе работы:

```
function FormComponent(props) {
    return (
        <main>
            <form>
                <input
                    name="firstName"
                    value={this.state.firstName}
                    onChange={this.handleChange}
                    placeholder="First Name"
                />
                <br />

                <input
                    name="lastName"
                    value={this.state.lastName}
                    onChange={this.handleChange}
                    placeholder="Last Name"
                />
                <br />

                <input
                    name="age"
                    value={this.state.age}
                    onChange={this.handleChange}
                    placeholder="Age"
                />
                <br />

                <label>
                    <input
```

```
            type="radio"
            name="gender"
            value="male"
            checked={this.state.gender === "male"}
            onChange={this.handleChange}
        /> Male
    </label>

    <br />

    <label>
        <input
            type="radio"
            name="gender"
            value="female"
            checked={this.state.gender === "female"}
            onChange={this.handleChange}
        /> Female
    </label>

    <br />

    <select
        value={this.state.destination}
        name="destination"
        onChange={this.handleChange}
    >
        <option value="">-- Please Choose a destination --</option>
        <option value="germany">Germany</option>
        <option value="norway">Norway</option>
```

```
    <option value="north pole">North Pole</option>
    <option value="south pole">South Pole</option>
</select>

<br />

<label>
    <input
        type="checkbox"
        name="isVegan"
        onChange={this.handleChange}
        checked={this.state.isVegan}
    /> Vegan?
</label>
<br />

<label>
    <input
        type="checkbox"
        name="isKosher"
        onChange={this.handleChange}
        checked={this.state.isKosher}
    /> Kosher?
</label>
<br />

<label>
    <input
        type="checkbox"
        name="isLactoseFree"
```

```
                        onChange={this.handleChange}
                        checked={this.state.isLactoseFree}
                    /> Lactose Free?
                </label>
                <br />

                <button>Submit</button>
            </form>
            <hr />
            <h2>Entered information:</h2>
            <p>Your name: {this.state.firstName} {this.state.lastName}</p>
            <p>Your age: {this.state.age}</p>
            <p>Your gender: {this.state.gender}</p>
            <p>Your destination: {this.state.destination}</p>
            <p>Your dietary restrictions:</p>

            <p>Vegan: {this.state.isVegan ? "Yes" : "No"}</p>
            <p>Kosher: {this.state.isKosher ? "Yes" : "No"}</p>
            <p>Lactose Free: {this.state.isLactoseFree ? "Yes" : "No"}</p>

        </main>
    )
}
```

Если взглянуть на этот код, то становится понятным, что простым его переносом из файла в файл мы ограничиться не можем, так как сейчас здесь присутствуют ссылки на состояние (например — `this.state.firstName`) и на обработчик событий (`this.handleChange`), которые раньше находились в том же компоненте, основанном на классе, в котором находился и этот код рендеринга. Теперь же всё то, что раньше бралось из того же класса, в котором находился код рендеринга, будет браться из свойств, передаваемых компоненту. Тут есть и некоторые другие проблемы. Сейчас мы исправим этот код, но сначала вернёмся к коду компонента `Form`, который сейчас находится в файле `FormContainer.js`.

Его метод `render()` теперь пуст. Нам нужно, чтобы в этом методе выводился бы компонент `FormComponent` и нужно организовать передачу ему необходимых свойств. Импортируем

`FormComponent` в файл `Form` и выведем `FormComponent` в методе `render()`, передав ему обработчик событий, и, в виде объекта, состояние. Теперь код компонента `Form` будет выглядеть так:

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

import FormComponent from "./FormComponent"


class Form extends Component {

    constructor() {

        super()

        this.state = {

            firstName: "",

            lastName: "",

            age: "",

            gender: "",

            destination: "",

            isVegan: false,

            isKosher: false,

            isLactoseFree: false

        }

        this.handleChange = this.handleChange.bind(this)

    }


    handleChange(event) {

        const {name, value, type, checked} = event.target

        type === "checkbox" ?

            this.setState({

                [name]: checked

            })

        :

        this.setState({

            [name]: value

        })
```

```
    }

    render() {
        return(
            <FormComponent
                handleChange={this.handleChange}
                data={this.state}
            />
        )
    }
}

export default Form
```

Исправим код компонента `FormComponent`, приведя его к следующему виду:

```
import React from "react"

function FormComponent(props) {
    return (
        <main>
            <form>
                <input
                    name="firstName"
                    value={props.data.firstName}
                    onChange={props.handleChange}
                    placeholder="First Name"
                />
                <br />

                <input
                    name="lastName"
```

```
    value={props.data.lastName}
    onChange={props.handleChange}
    placeholder="Last Name"
/>
<br />

<input
    name="age"
    value={props.data.age}
    onChange={props.handleChange}
    placeholder="Age"
/>
<br />

<label>
    <input
        type="radio"
        name="gender"
        value="male"
        checked={props.data.gender === "male"}
        onChange={props.handleChange}
    /> Male
</label>

<br />

<label>
    <input
        type="radio"
        name="gender"
```

```
            value="female"
            checked={props.data.gender === "female"}
            onChange={props.handleChange}
        /> Female
    </label>

    <br />

    <select
        value={props.data.destination}
        name="destination"
        onChange={props.handleChange}
    >
        <option value="">-- Please Choose a destination --</option>
        <option value="germany">Germany</option>
        <option value="norway">Norway</option>
        <option value="north pole">North Pole</option>
        <option value="south pole">South Pole</option>
    </select>

    <br />

    <label>
        <input
            type="checkbox"
            name="isVegan"
            onChange={props.handleChange}
            checked={props.data.isVegan}
        /> Vegan?
    </label>
```

```
        <br />

        <label>
            <input
                type="checkbox"
                name="isKosher"
                onChange={props.handleChange}
                checked={props.data.isKosher}
            /> Kosher?
        </label>
        <br />

        <label>
            <input
                type="checkbox"
                name="isLactoseFree"
                onChange={props.handleChange}
                checked={props.data.isLactoseFree}
            /> Lactose Free?
        </label>
        <br />

        <button>Submit</button>
    </form>
    <hr />
    <h2>Entered information:</h2>
    <p>Your name: {props.data.firstName} {props.data.lastName}</p>
    <p>Your age: {props.data.age}</p>
    <p>Your gender: {props.data.gender}</p>
    <p>Your destination: {props.data.destination}</p>
```

```
            <p>Your dietary restrictions:</p>


            <p>Vegan: {props.data.isVegan ? "Yes" : "No"}</p>

            <p>Kosher: {props.data.isKosher ? "Yes" : "No"}</p>

            <p>Lactose Free: {props.data.isLactoseFree ? "Yes" : "No"}</p>


        </main>

    )

}


export default FormComponent
```

Здесь мы исправили код с учётом того, что компонент теперь получает данные и ссылку на обработчик событий через свойства.

После всех этих преобразований не изменится ни внешний вид формы, ни то, как она работает, но структуру кода проекта мы улучшили, хотя размер кода компонента `FormComponent` всё ещё оказывается довольно большим. Однако теперь этот код решает лишь одну задачу, он отвечает только за визуализацию формы. Поэтому работать с ним теперь гораздо легче.

В результате мы добились разделения ответственности между компонентами. Компонент `Form` из файла `FormContainer.js` теперь занят исключительно логикой функционирования приложения, а компонент `FormComponent` из файла `FormComponent.js` содержит только код, формирующий интерфейс приложения. Компонент `App` теперь отвечает лишь за сборку страницы из крупных блоков.

Тут стоит отметить, что с учётом существования библиотек наподобие `Redux` и недавно вышедшего API `Context`, рассмотренный здесь паттерн Container/Component уже не так актуален, как прежде. Например, средствами Redux можно поддерживать глобальное состояние приложения, которым могут пользоваться компоненты.

Учебный курс по React, часть 27: курсовой проект

Занятие 45. Курсовой проект. Генератор мемов

Оригинал

Вот мы и добрались до курсового проекта. Займёмся созданием приложения, которое будет генерировать мемы. Начнём работу со стандартного проекта create-react-app, созданного с помощью такой команды:

```
npx create-react-app meme-generator
```

Здесь можно найти сведения об особенностях её использования.

В ходе работы над этим проектом вам будет предложено реализовывать некоторые его части самостоятельно, а потом уже читать пояснения о них. В стандартном проекте уже есть шаблонный код, находящийся, в частности, в файлах `index.js` и `App.js`. Вы вполне можете этот код удалить и

попытаться написать его самостоятельно для того чтобы проверить себя в реализации стандартных механизмов React-приложений.

В этом проекте вам предлагается использовать следующие стили:

```
* {
    box-sizing: border-box;
}

body {
    margin: 0;
    background-color: whitesmoke;
}

header {
    height: 100px;
    display: flex;
    align-items: center;
    background: #6441A5;  /* fallback for old browsers */
    background: -webkit-linear-gradient(to right, #2a0845, #6441A5);  /* Chrome 10-25, Safari 5.1-6 */
    background: linear-gradient(to right, #2a0845, #6441A5); /* W3C, IE 10+/ Edge, Firefox 16+, Chrome 26+, Opera 12+, Safari 7+ */
}

header > img {
    height: 80%;
    margin-left: 10%;
}

header > p {
    font-family: VT323, monospace;
    color: whitesmoke;
    font-size: 50px;
```

```
    margin-left: 60px;
}

.meme {
    position: relative;
    width: 90%;
    margin: auto;
}

.meme > img {
    width: 100%;
}

.meme > h2 {
    position: absolute;
    width: 80%;
    text-align: center;
    left: 50%;
    transform: translateX(-50%);
    margin: 15px 0;
    padding: 0 5px;
    font-family: impact, sans-serif;
    font-size: 2em;
    text-transform: uppercase;
    color: white;
    letter-spacing: 1px;
    text-shadow:
        2px 2px 0 #000,
        -2px -2px 0 #000,
        2px -2px 0 #000,
```

```
        -2px 2px 0 #000,
        0 2px 0 #000,
        2px 0 0 #000,
        0 -2px 0 #000,
        -2px 0 0 #000,
        2px 2px 5px #000;
}

.meme > .bottom {
    bottom: 0;
}

.meme > .top {
    top: 0;
}

.meme-form {
    width: 90%;
    margin: 20px auto;
    display: flex;
    justify-content: space-between;
}

.meme-form > input {
    width: 45%;
    height: 40px;
}

.meme-form > button {
    border: none;
```

```
    font-family: VT323, monospace;
    font-size: 25px;
    letter-spacing: 1.5px;
    color: white;
    background: #6441A5;
}

.meme-form > input::-webkit-input-placeholder { /* Chrome/Opera/Safari */
  font-family: VT323, monospace;
  font-size: 25px;
  text-align: cen
}
.meme-form > input::-moz-placeholder { /* Firefox 19+ */
  font-family: VT323, monospace;
  font-size: 25px;
  text-align: cen
}
.meme-form > input:-ms-input-placeholder { /* IE 10+ */
  font-family: VT323, monospace;
  font-size: 25px;
  text-align: cen
}
.meme-form > input:-moz-placeholder { /* Firefox 18- */
  font-family: VT323, monospace;
  font-size: 25px;
  text-align: cen
}
```

Эти стили можно включить в уже имеющийся в проекте файл `index.css` и подключить в файле `index.js`.

Итак, исходя из предположения о том, что файлы `index.js` и `App.js` сейчас пусты, вам, в качестве первого задания, предлагается самостоятельно написать код `index.js`, создать простейший компонент в `App.js` и вывести его в `index.js`.

Вот что должно оказаться в `index.js`:

```
import React from "react"

import ReactDOM from "react-dom"

import './index.css'

import App from "./App"


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

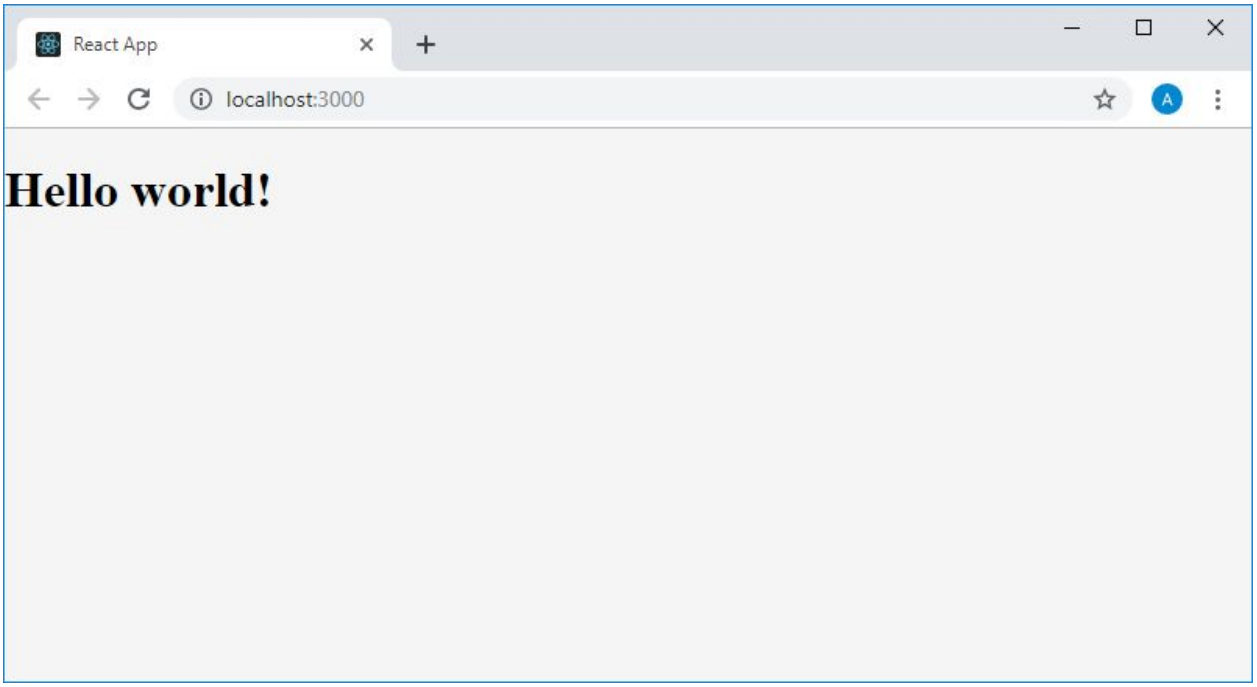
Здесь мы импортируем `React` и `ReactDOM`, импортируем стили из `index.css` и компонент `App`. После этого, с помощью метода `ReactDOM.render()`, выводим то, что формирует компонент `App`, в элемент страницы `index.html` с идентификатором `root (<div id="root"></div>)`.

Вот как может выглядеть файл `App.js`:

```
import React from "react"


function App() {

    return (

        <h1>Hello world!</h1>

    )

}


export default App
```

Тут сейчас представлен простейший функциональный компонент.

На данном этапе работы проект выглядит так, как показано ниже.



Приложение в браузере

Теперь создайте два новых компонента, в двух файлах, имена которых соответствуют именам компонентов:

- Компонент `Header`, который будет использоваться для вывода заголовка приложения.
- Компонент `MemeGenerator`, в котором будут решаться основные задачи, возлагаемые на приложение. А именно, здесь будут выполняться обращения к API. Здесь же будут храниться данные приложения.

Учитывая то, какие функции возлагаются на эти компоненты, подумайте о том, какими они должны быть.

Вот содержимое файла `Header.js`:

```
import React from "react"


function Header() {

    return (

        <h1>HEADER</h1>

    )

}


export default Header
```

Так как этот компонент будет использоваться только для вывода заголовка приложения, его мы оформили в виде функционального компонента.

Вот код файла `MemeGenerator.js`:

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


class MemeGenerator extends Component {
    constructor() {
        super()
        this.state ={}
    }

    render() {
        return (
            <h1>MEME GENERATOR SECTION</h1>
        )
    }
}

export default MemeGenerator
```

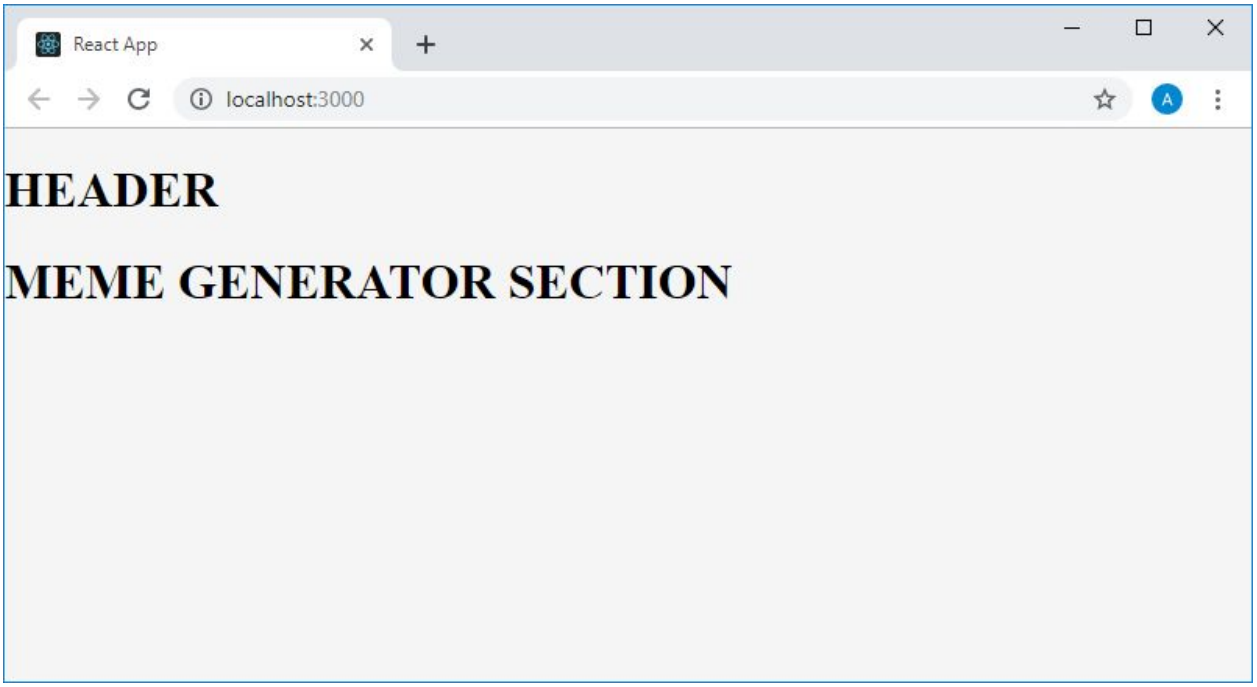
Тут мы, учитывая задачи, которые предполагается решать средствами компонента `MemeGenerator`, будем использовать компонент, основанный на классе. Здесь имеется конструктор, в котором мы инициализируем состояние пустым объектом.

Создав эти файлы, импортируем их в `App.js` и возвратим из функционального компонента `App` разметку, в которой используются экземпляры этих компонентов, не забывая о том, что, если функциональный компонент возвращает несколько элементов, их нужно во что-то обернуть. В нашем случае это — тег `<div>`. Вот обновлённый код `App.js`:

```
import React from "react"
import Header from "./Header"
import MemeGenerator from "./MemeGenerator"


function App() {
    return (
        <div>
            <Header />
            <MemeGenerator />
```

```
        </div>
    )
}


export default App
```

Проверим внешний вид приложения.



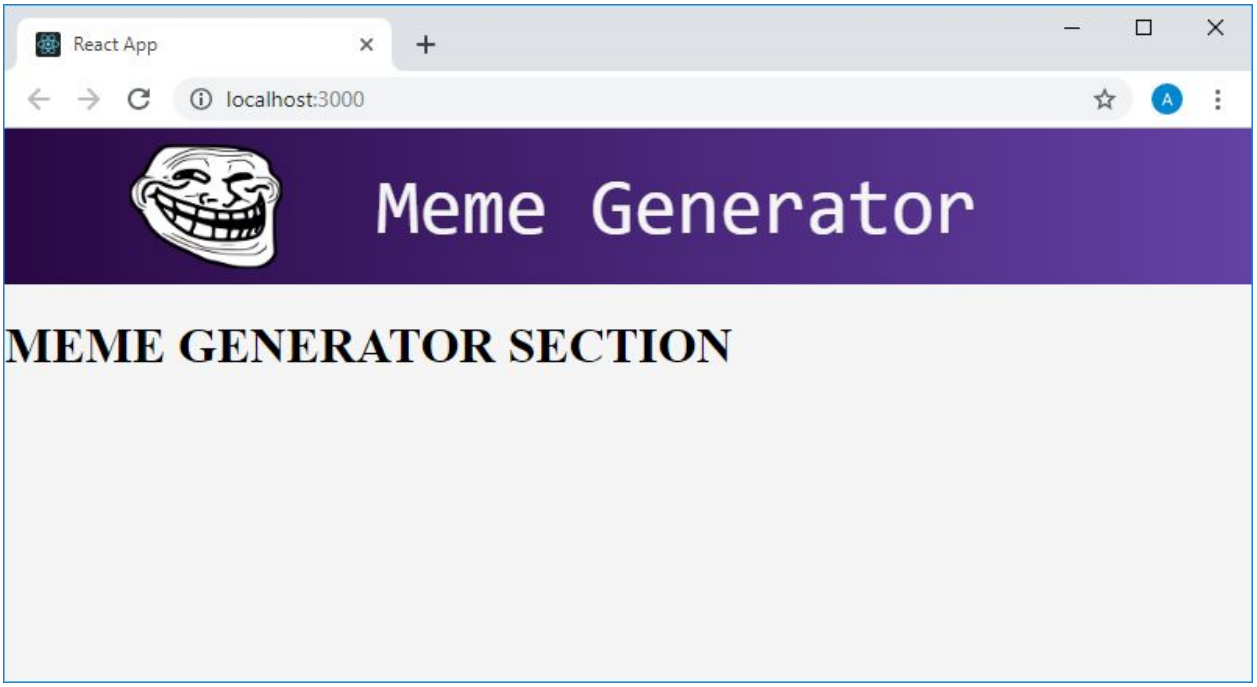
Приложение в браузере

Теперь поработаем над компонентом `Header`. Здесь мы воспользуемся семантическим элементом HTML5 `<header>`. В этом теге будет размещено изображение и текст. Теперь код файла `Header.js` будет выглядеть так:

```
import React from "react"


function Header() {
    return (
        <header>
            <img

src="http://www.pngall.com/wp-content/uploads/2016/05/Trollface.png"
                alt="Problem?"
            />
            <p>Meme Generator</p>
```

```
        </header>
    )
}


export default Header
```

Вот как изменится внешний вид приложения.



Приложение в браузере

Заголовок приложения оформлен в соответствии с ранее подключёнными в `index.js` стилями. Работа над компонентом `Header` на этом завершена.

Продолжим заниматься компонентом `MemeGenerator`. Сейчас вам предлагается самостоятельно инициализировать состояние этого компонента, записав в него следующие данные:

- Текст, выводимый в верхней части мема (свойство `topText`).
- Текст, выводимый в нижней части мема (свойство `bottomText`).
- Случайное изображение (свойство `randomImage`, которое нужно инициализировать ссылкой http://i.imgflip.com/1bij.jpg).

Вот каким будет код `MemeGenerator.js` после инициализации состояния:

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


class MemeGenerator extends Component {
    constructor() {
        super()
        this.state = {
```

```
            topText: "",
            bottomText: "",
            randomImg: "http://i.imgflip.com/1bij.jpg"
        }
    }

    render() {
        return (
            <h1>MEME GENERATOR SECTION</h1>
        )
    }
}

export default MemeGenerator
```

Сейчас на внешний вид приложения это не повлияет.

Мы будем использовать обращения к API, которое возвращает массив объектов, содержащих ссылки на изображения, на основе которых можно создавать мемы. На данном этапе работы над проектом вам предлагается реализовать в компоненте `MemeGenerator` следующий функционал:

- Выполните обращение к API https://api.imgflip.com/get_memes/.
- Сохраните данные, доступные в ответе в виде массива `response.data.memes`, в новом свойстве состояния (`allMemeImgs`).

Вот, чтобы было понятнее, фрагмент JSON-данных, возвращаемых при обращении к этому API:

```
{
   "success":true,
   "data":{
      "memes":[
         {
            "id":"112126428",
            "name":"Distracted Boyfriend",
            "url":"https:\/\/i.imgflip.com\/1ur9b0.jpg",
            "width":1200,
            "height":800,
```

```
            "box_count":3
         },
         {
            "id":"87743020",
            "name":"Two Buttons",
            "url":"https:\/\/i.imgflip.com\/1g8my4.jpg",
            "width":600,
            "height":908,
            "box_count":2
         },
         {
            "id":"129242436",
            "name":"Change My Mind",
            "url":"https:\/\/i.imgflip.com\/24y43o.jpg",
            "width":482,
            "height":361,
            "box_count":2
         },
         ….
   ]
   }
}
```

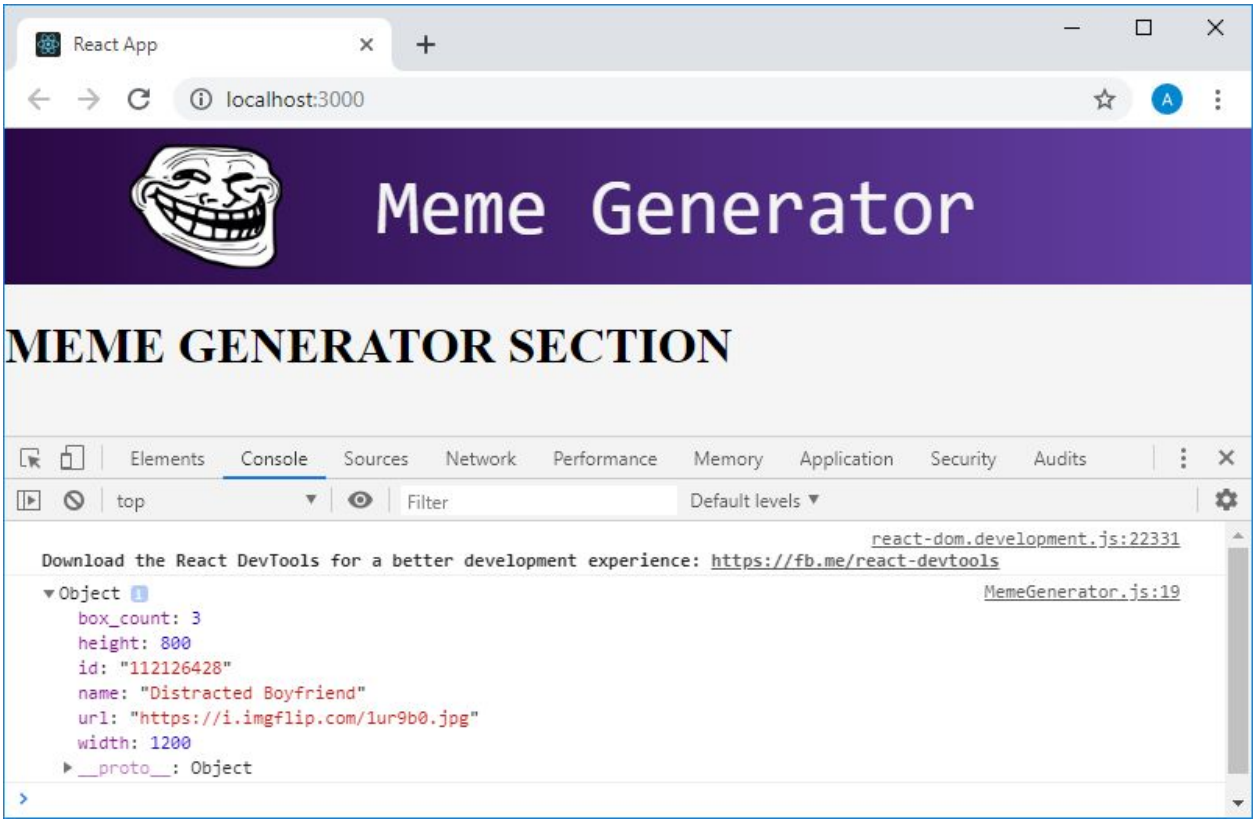

Решая задачу, поставленную выше, нужно учитывать то, что речь идёт о данных, которые нужны компоненту в самом начале работы приложения. Поэтому для их загрузки мы прибегнем к методу жизненного цикла компонента `componentDidMount()`. Здесь мы, воспользовавшись стандартным методом `fetch()`, выполним обращение к API. Оно возвращает промис. После загрузки данных нам будет доступен объект ответа, из него мы извлекаем массив `memes` и помещаем его в новое свойство состояния `allMemeImgs`, инициализированное пустым массивом. Так как эти данные пока не используются для формирования чего-то такого, что выводится на экран, мы, для проверки правильности работы механизма загрузки данных, выведем первый элемент массива в консоль.

Вот как выглядит код компонента `MemeGenerator` на данном этапе работы:

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

```
class MemeGenerator extends Component {
    constructor() {
        super()
        this.state = {
            topText: "",
            bottomText: "",
            randomImg: "http://i.imgflip.com/1bij.jpg",
            allMemeImgs: []
        }
    }

    componentDidMount() {
        fetch("https://api.imgflip.com/get_memes")
            .then(response => response.json())
            .then(response => {
                const {memes} = response.data
                console.log(memes[0])
                this.setState({ allMemeImgs: memes })
            })
    }

    render() {
        return (
            <h1>MEME GENERATOR SECTION</h1>
        )
    }
}

export default MemeGenerator
```

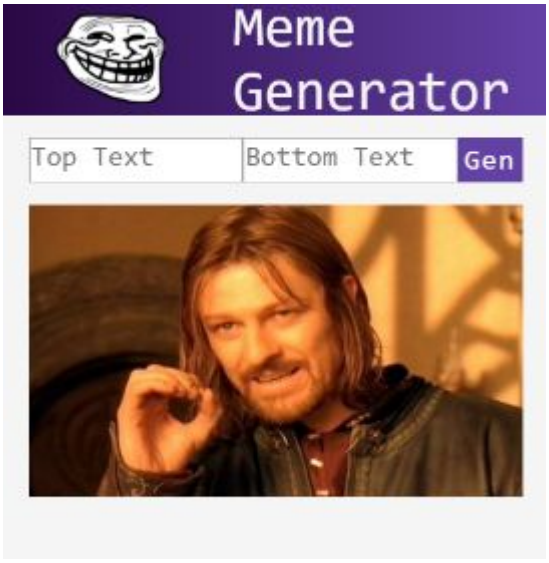
Вот что попадает в консоль после успешной загрузки данных.



Приложение в браузере, вывод в консоль первого элемента загруженного массива

Обратите внимание на то, что изображение описано с использованием множества свойств. Мы будем использовать лишь свойство `url`, дающее доступ к ссылке для загрузки изображения.

В начале курса мы говорили о том, как будет выглядеть это приложение.



Генератор мемов

В частности, в его интерфейсе имеется пара полей для ввода текста, который будет выводиться в верхней и нижней частях изображения. Сейчас вам предлагается, взяв за основу показанный ниже обновлённый код компонента `MemeGenerator`, который отличается от вышеприведённого кода этого компонента тем, что сюда добавлена заготовка формы, самостоятельно создать пару текстовых полей, `topText` и `bottomText`. Учитывайте то, что это должны быть управляемые компоненты. Добавьте к ним необходимые атрибуты. Создайте обработчик событий `onChange` этих полей, в котором нужно, по мере ввода текста в них, обновлять соответствующие свойства состояния.

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

class MemeGenerator extends Component {
    constructor() {
        super()
        this.state = {
            topText: "",
            bottomText: "",
            randomImg: "http://i.imgflip.com/1bij.jpg",
            allMemeImgs: []
        }
    }

    componentDidMount() {
        fetch("https://api.imgflip.com/get_memes")
            .then(response => response.json())
            .then(response => {
                const {memes} = response.data
                this.setState({ allMemeImgs: memes })
            })
    }

    render() {
        return (
            <div>
                <form className="meme-form">
                    {
                        // Здесь должны быть текстовые поля
                    }
```

```
                    <button>Gen</button>
                </form>
            </div>
        )
    }
}


export default MemeGenerator
```

Кстати, обратите внимание на то, что для того чтобы включить комментарий в код, возвращаемый методом `render()`, мы заключили его в фигурные скобки для того чтобы указать системе на то, что данный фрагмент она должна воспринимать как JavaScript-код.

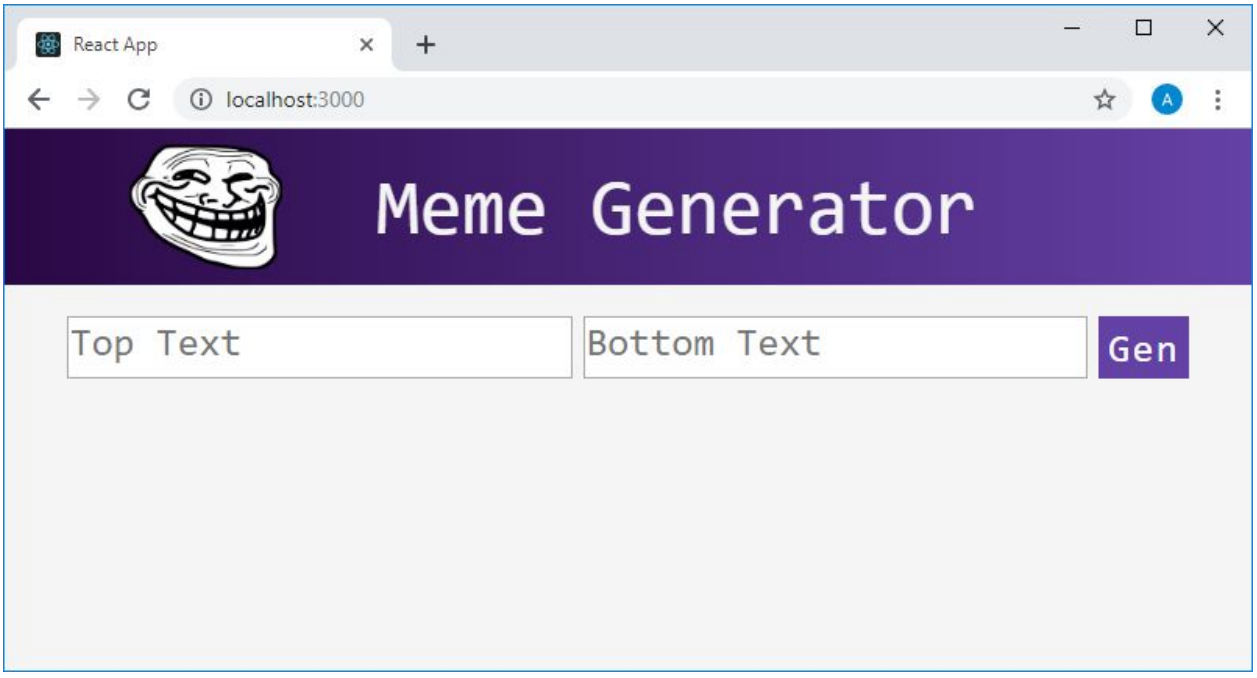
Вот что у вас должно получиться на данном этапе работы над приложением:

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


class MemeGenerator extends Component {
    constructor() {
        super()
        this.state = {
            topText: "",
            bottomText: "",
            randomImg: "http://i.imgflip.com/1bij.jpg",
            allMemeImgs: []
        }
        this.handleChange = this.handleChange.bind(this)
    }


    componentDidMount() {
        fetch("https://api.imgflip.com/get_memes")
            .then(response => response.json())
            .then(response => {
                const {memes} = response.data
```

```
            this.setState({ allMemeImgs: memes })
        })
}

handleChange(event) {
    const {name, value} = event.target
    this.setState({ [name]: value })
}

render() {
    return (
        <div>
            <form className="meme-form">
                <input
                    type="text"
                    name="topText"
                    placeholder="Top Text"
                    value={this.state.topText}
                    onChange={this.handleChange}
                />
                <input
                    type="text"
                    name="bottomText"
                    placeholder="Bottom Text"
                    value={this.state.bottomText}
                    onChange={this.handleChange}
                />

                <button>Gen</button>
            </form>
```

```
            </div>
        )
    }
}

export default MemeGenerator
```

Теперь страница приложения будет выглядеть так, как показано ниже.



Приложение в браузере

Пока на экран выводятся лишь поля с текстом подсказок, ввод данных в них не приводит к изменениям интерфейса. Для того чтобы проверить правильность работы реализованных здесь механизмов, вы можете воспользоваться командой `console.log()`.

Теперь поработаем над той частью приложения, которая ответственна за вывод на экран изображения-мема. Вспомним о том, что сейчас у нас имеется массив, содержащий сведения об изображениях, которые планируется использовать в качестве основы мемов. Приложение должно, по нажатию на кнопку `Gen`, случайным образом выбирать из этого массива изображение и формировать мем.

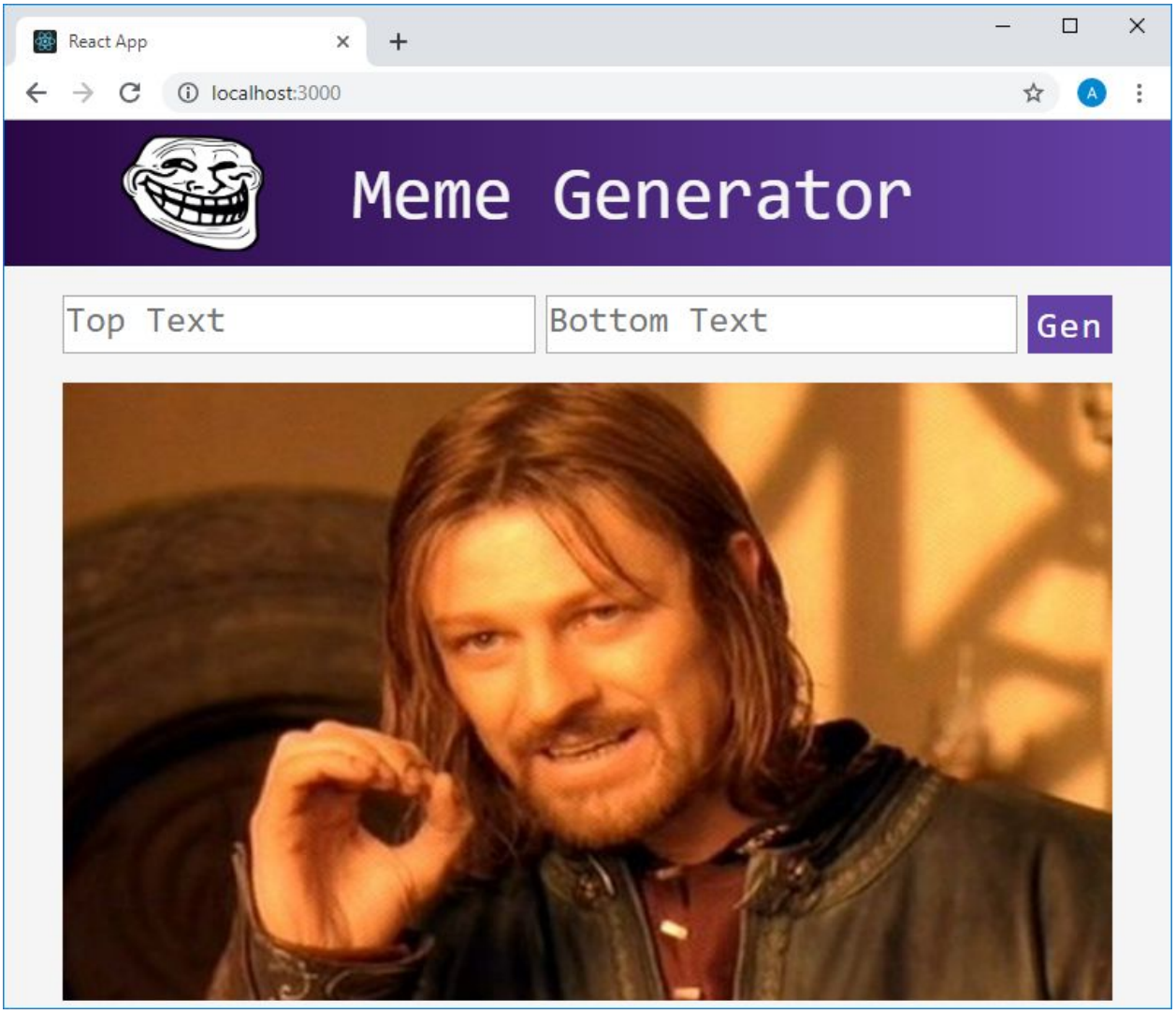
Вот обновлённый код компонента `MemeGenerator`. Здесь, в методе `render()`, ниже кода описания формы, имеется элемент `<div>`, включающий в себя элемент `<img>`, выводящий изображение, и пару элементов `<h2>`, которые выводят надписи. Элементы `<div>` и `<h2>` оформлены с использованием стилей, которые мы добавляли в проект в самом начале работы над ним.

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


class MemeGenerator extends Component {

    constructor() {
```

```
    super()
    this.state = {
        topText: "",
        bottomText: "",
        randomImg: "http://i.imgflip.com/1bij.jpg",
        allMemeImgs: []
    }
    this.handleChange = this.handleChange.bind(this)
}

componentDidMount() {
    fetch("https://api.imgflip.com/get_memes")
        .then(response => response.json())
        .then(response => {
            const {memes} = response.data
            this.setState({ allMemeImgs: memes })
        })
}

handleChange(event) {
    const {name, value} = event.target
    this.setState({ [name]: value })
}

render() {
    return (
        <div>
            <form className="meme-form">
                <input
                    type="text"
```

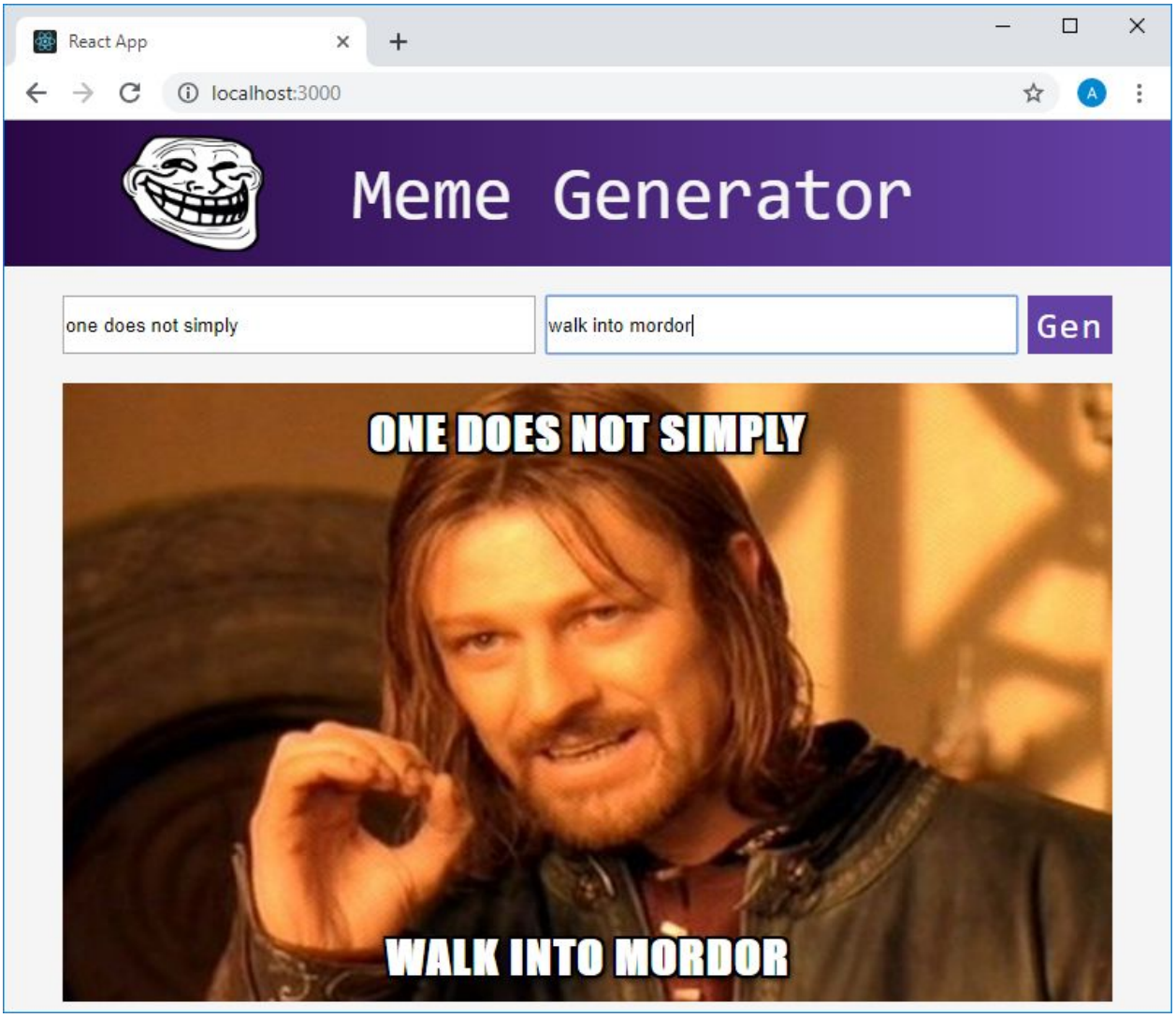
```
                        name="topText"
                        placeholder="Top Text"
                        value={this.state.topText}
                        onChange={this.handleChange}
                    />
                    <input
                        type="text"
                        name="bottomText"
                        placeholder="Bottom Text"
                        value={this.state.bottomText}
                        onChange={this.handleChange}
                    />

                    <button>Gen</button>
                </form>
                <div className="meme">
                    <img src={this.state.randomImg} alt="" />
                    <h2 className="top">{this.state.topText}</h2>
                    <h2 className="bottom">{this.state.bottomText}</h2>
                </div>
            </div>
        )
    }
}

export default MemeGenerator
```

Вот как приложение выглядит теперь.



Приложение в браузере

Обратите внимание на то, что здесь выводится то изображение, которым инициализировано состояние. Мы пока не пользуемся изображениями, которые хранятся в свойстве состояния `allMemeImgs`. Попробуем ввести что-нибудь в текстовые поля.



Приложение в браузере

Как видно, подсистемы приложения, ответственные за работу с текстом, функционируют так, как ожидается. Теперь осталось лишь сделать так, чтобы по нажатию на кнопку `Gen` из массива с данными изображений выбиралось бы случайное изображение и загружалось бы в элемент `<img>`, присутствующий на странице ниже полей для ввода текста.

Для того чтобы оснастить приложение этой возможностью — выполните следующее задание. Создайте метод, который срабатывает при нажатии на кнопку `Gen`. Этот метод должен выбирать одно из изображений, сведения о которых хранятся в свойстве состояния `allMemeImgs`, после чего выполнять действия, которые позволяют вывести это изображение в элементе `<img>`, расположенном под полями ввода текста. Учитывайте то, что в `allMemeImgs` хранится массив объектов, описывающих изображения, и то, что у каждого объекта из этого массива есть свойство `url`.

Вот код, в котором приведено решение этой задачи:

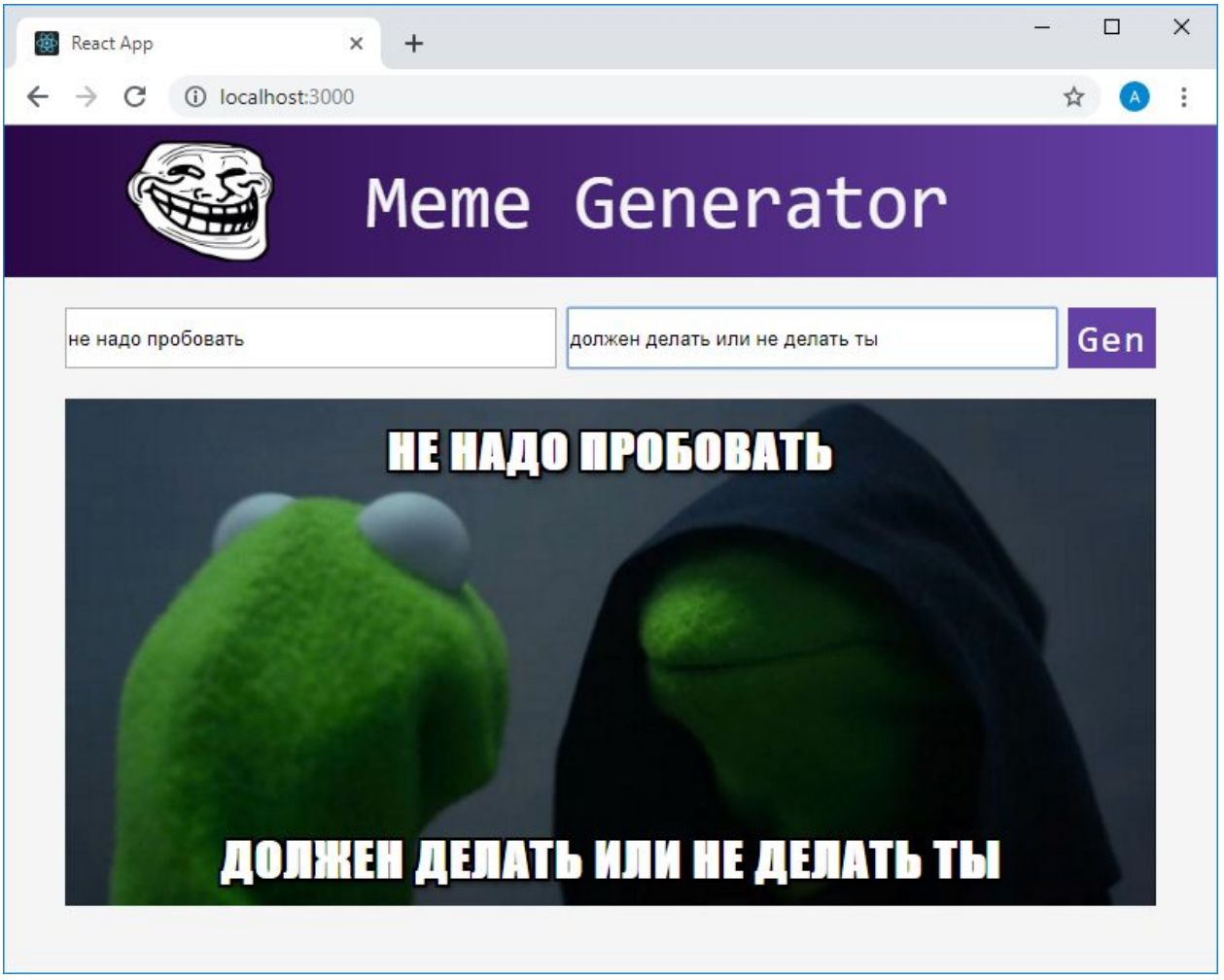
```
import React, {Component} from "react"


class MemeGenerator extends Component {

    constructor() {

        super()
```

```
        this.state = {
            topText: "",
            bottomText: "",
            randomImg: "http://i.imgflip.com/1bij.jpg",
            allMemeImgs: []
        }
        this.handleChange = this.handleChange.bind(this)
        this.handleSubmit = this.handleSubmit.bind(this)
    }

    componentDidMount() {
        fetch("https://api.imgflip.com/get_memes")
            .then(response => response.json())
            .then(response => {
                const {memes} = response.data
                this.setState({ allMemeImgs: memes })
            })
    }

    handleChange(event) {
        const {name, value} = event.target
        this.setState({ [name]: value })
    }

    handleSubmit(event) {
        event.preventDefault()
        const randNum = Math.floor(Math.random() *
this.state.allMemeImgs.length)
        const randMemeImg = this.state.allMemeImgs[randNum].url
        this.setState({ randomImg: randMemeImg })
    }
```

```
render() {
    return (
        <div>
            <form className="meme-form" onSubmit={this.handleSubmit}>
                <input
                    type="text"
                    name="topText"
                    placeholder="Top Text"
                    value={this.state.topText}
                    onChange={this.handleChange}
                />
                <input
                    type="text"
                    name="bottomText"
                    placeholder="Bottom Text"
                    value={this.state.bottomText}
                    onChange={this.handleChange}
                />

                <button>Gen</button>
            </form>
            <div className="meme">
                <img src={this.state.randomImg} alt="" />
                <h2 className="top">{this.state.topText}</h2>
                <h2 className="bottom">{this.state.bottomText}</h2>
            </div>
        </div>
    )
}
```

```
}


export default MemeGenerator
```

Кнопке `Gen` можно назначить обработчик события, возникающего при щелчке по ней, как это делается при работе с любыми другими кнопками. Однако, учитывая то, что эта кнопка используется для отправки формы, лучше будет воспользоваться обработчиком события `onSubmit` формы. В этом обработчике, `handleSubmit()`, мы вызываем метод поступающего в него события `event.preventDefault()` для того, чтобы отменить стандартную процедуру отправки формы, в ходе которой выполняется перезагрузка страницы. Далее, мы получаем случайное число в диапазоне от 0 до значения, соответствующего индексу последнего элемента массива `allMemeImgs` и используем это число для обращения к элементу с соответствующим индексом. Обратившись к элементу, являющемуся объектом, мы получаем свойство этого объекта `url` и записываем его в свойство состояния `randomImg`. После этого выполняется повторный рендеринг компонента и внешний вид страницы меняется.



Страница приложения в браузере

Курсовой проект завершён.

Учебный курс по React, часть 28: современные возможности React, идеи проектов, заключение

Занятие 46. Разработка современных React-приложений

Оригинал

Работой над библиотекой React занимается немало программистов в Facebook, вклад в проект делают и члены многочисленного сообщества, сложившегося вокруг React. Всё это ведёт к тому, что React очень быстро развивается. Например, если вы, изучая React в начале 2019 года, смотрели материалы по этой библиотеке, скажем, двухгодичной давности, вы не могли не заметить изменений, произошедших в React с момента выхода тех материалов. Например, в React 16.3 появились некоторые новые методы жизненного цикла компонентов, а некоторые методы были признаны устаревшими. А, скажем, в React 16.6 появилось ещё больше новых возможностей. Огромное количество новшеств ожидается в React 17.0 и в следующих версиях этой библиотеки.

Сейчас мы поговорим о некоторых современных возможностях React.

Многие из этих возможностей зависят от того, какая версия спецификации ECMAScript поддерживается инструментами, используемыми при разработке React-проекта. Скажем, если вы пользуетесь танспилятором Babel — это означает, что вам доступны самые свежие возможности JavaScript. При этом надо отметить, что при использовании в проектах неких возможностей JavaScript, ещё не включённых в стандарт, вы можете столкнуться с тем, что они, если будут включены в стандарт, могут измениться.

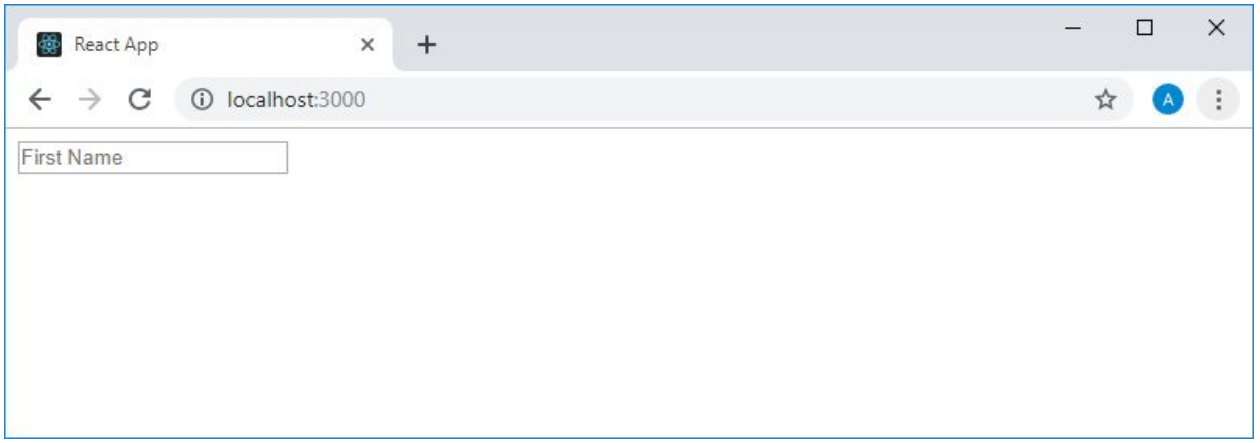
Одной из современных возможностей JavaScript, которой можно пользоваться при разработке React-приложений, является возможность объявления методов классов с использованием синтаксиса стрелочных функций.

Вот код компонента `App`, который выводит на экран текстовое поле:

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


class App extends Component {

    // Перепишем с использованием свойств класса

    constructor() {

        super()

        this.state = {

            firstName: ""

        }

        this.handleChange = this.handleChange.bind(this)

    }


    // Перепишем в виде стрелочной функции
```

```
    handleChange(event) {
        const { name, value } = event.target
        this.setState({
            [name]: value
        })
    }

    render() {
        return (
            <main>
                <form>
                    <input
                        type="text"
                        name="firstName"
                        value={this.state.firstName}
                        onChange={this.handleChange}
                        placeholder="First Name"
                    />
                </form>
                <h1>{this.state.firstName}</h1>
            </main>
        )
    }
}

export default App
```

Вот как выглядит страница этого приложения в браузере.



Приложение в браузере

Перепишем метод `handleChange()` в виде стрелочной функции, приведя код компонента к следующему виду:

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


class App extends Component {
    // Перепишем с использованием свойств класса
    constructor() {
        super()
        this.state = {
            firstName: ""
        }
    }

    // Переписано в виде стрелочной функции
    handleChange = (event) => {
        const { name, value } = event.target
        this.setState({
            [name]: value
        })
    }

    render() {
        return (
```

```
            <main>
                <form>
                    <input
                        type="text"
                        name="firstName"
                        value={this.state.firstName}
                        onChange={this.handleChange}
                        placeholder="First Name"
                    />
                </form>
                <h1>{this.state.firstName}</h1>
            </main>
        )
    }
}


export default App
```

В ходе такого преобразования в код внесены небольшие изменения, но эти изменения оказывают серьёзное влияние на то, как будет работать метод. Ключевое слово `this` в стрелочных функциях указывает на лексическую область видимости, в которой они находятся. Эти функции не поддерживают привязку `this`. Эта особенность стрелочных функций ведёт к тому, что методы, объявленных с их использованием, не нужно привязывать к `this` в конструкторе класса.

Ещё одна возможность, которую мы тут рассмотрим, заключается в использовании свойств классов. Сейчас мы, при инициализации состояния в конструкторе, пользуемся инструкцией `this.state`. Так мы создаём свойство экземпляра класса. Теперь же свойства можно создавать за пределами конструктора. В результате можно преобразовать код следующим образом:

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


class App extends Component {
    // Переписано с использованием свойств класса
    state = { firstName: "" }


    // Переписано в виде стрелочной функции
```

```
    handleChange = (event) => {
        const { name, value } = event.target
        this.setState({
            [name]: value
        })
    }

    render() {
        return (
            <main>
                <form>
                    <input
                        type="text"
                        name="firstName"
                        value={this.state.firstName}
                        onChange={this.handleChange}
                        placeholder="First Name"
                    />
                </form>
                <h1>{this.state.firstName}</h1>
            </main>
        )
    }
}

export default App
```

Обратите внимание на то, что тут мы избавились от конструктора, инициализировав состояние при объявлении соответствующего свойства. Всё указывает на то, что эта возможность JS будет, в обозримом будущем, включена в стандарт.

Вот список материалов, посвящённых современным возможностям React.

- API Context. Его можно использовать вместо Redux, хотя это не говорит о том, что библиотека Redux потеряла актуальность.

- Границы ошибок.
- Шаблон Render props.
- Компоненты высшего порядка.
- Маршрутизатор React.
- Хуки React.
- Новые возможности React 16.6.

В целом можно отметить, что, так как React развивается очень быстро, всем, кто занимается React-разработкой, рекомендуется постоянно наблюдать за новшествами этой библиотеки.

Занятие 47. Идеи React-проектов

Оригинал

В ходе освоения React мы с вами создали пару проектов — Todo-приложение и генератор мемов. Вполне возможно, что вы уже знаете — что хотите создать с использованием React. Может быть, вы уже разрабатываете собственное приложение. Если же вы пока не определились с выбором, и учитывая то, что практика — это лучший способ осваивать компьютерные технологии — вот, вот и вот — материалы, в которых вы найдёте целую кучу идей веб-приложений, которые можно создать с помощью React.

Занятие 48. Заключение

Оригинал

Примите поздравления! Вы только что завершили изучение курса, посвящённого библиотеке React. Вы ознакомились с базовыми строительными блоками React-приложений, которые вы уже можете использовать для создания собственных проектов. Правда, если вы хотите создавать что-то с использованием React, будьте готовы к тому, что вам предстоит узнать ещё очень много нового.

Пройдёмся по основным концепциям, которые вы изучили в ходе освоения этого курса.

- JSX. JSX позволяет описывать пользовательские интерфейсы с применением синтаксиса, очень похожего на обычный HTML-код.
- Два подхода к разработке компонентов. Компоненты, основанные на классах и функциональные компоненты.
- Разные способы стилизации React-приложений.
- Передача свойств от родительских компонентов дочерним компонентам.
- Использование состояния компонентов для хранения данных и для работы с ними.
- Условный рендеринг.
- Работа с формами в React.

Благодарим за внимание!