# Введение. История создания React

**До JavaScript**

В эти давние времена веб-сайты целиком генерировались на сервере, вся разметка в готовом виде отдавалась браузеру и не изменялась на стороне клиента. Если пользователь нажимал на кнопку, браузер требовал от сервера новую разметку, и страница перезагружалась. Так происходило каждый раз при любом действии пользователя на сайте.

**После JavaScript и AJAX**

После появления этих инструментов сайты стали интерактивными. JS позволял производить манипуляции с DOM-деревом, а технология AJAX — загружать данные на страницу по частям, без полной перезагрузки.

Вычислительная мощность компьютеров и телефонов росла, средствами JS клиент стал выполнять всё более сложные операции в браузере. Программы начали постепенно мигрировать из десктопных версий в веб-версии. Но для написания сложных приложений всё ещё требовалось много усилий.

**Библиотеки и фреймворки**

Тогда стали появляться библиотеки и фреймворки. Одна из первых — jQuery — мощный и изящный по тем меркам инструмент. Но даже с ним писать приложения было довольно трудоёмко.

А дело было в императивности такой разработки. Добавление новых возможностей в приложение тех времён могло легко сломать существующую функциональность. Связанность одних фич с другими была колоссальной, и на разработку уходило много сил и времени. Например, при разработке интернет-магазина нужно было проверять, не сломалась ли функциональность личного кабинета после доделок в корзине. Это была изматывающая и кропотливая работа.

Чтобы уменьшить боль разработчиков при написании сложных сайтов и приложений, создали первые фреймворки: Backbone.js, Knockout.js и Ember.js. Но поскольку это были фреймворки, а не библиотеки, они навязывали определённую структуру приложения и архитектурные решения. Разработчикам приходилось подолгу привыкать к новым способам написания кода. Фреймворки хоть и решали часть проблем, но всё же накладывали ограничения на свободомыслие разработчиков при проектировании сайтов и приложений. А ещё из-за императивного подхода возникала проблема масштабирования приложений.

**И вот появилась она — библиотека React**

Многие компании искали решение проблемы масштабирования, и одной из них была команда Facebook. Сначала библиотека React была внутренним инструментом Facebook и в 2011‒2012 годах использовалась в лентах новостей и Instagram. А уже в 2013 году Facebook представил исходный код этой библиотеки всему миру — это было публичное рождение библиотеки React.

Библиотека, в отличие от фреймворка, не накладывает ограничений на архитектуру приложения и предоставляет свободу выбора: разработчик сам решает, какую библиотеку использовать вместе с React для роутинга, управления состоянием, взаимодействия с сервером и так далее. React можно применять практически в любых условиях и усложнять стек, только если это нужно для функциональности приложения.

У React множество преимуществ, которые выделяют его на фоне других библиотек. Вы убедитесь в этом сами, когда начнёте работать с библиотекой в рамках курса. А пока взгляните, как изящно выглядит код на React:

Скопировать кодJSX

*//React.js*

import React, { useState } from 'react';

function Counter() {

const [count, setCount] = useState(0);

return (

<div>

<p>Вы кликнули {count} раз(а)</p>

<button onClick={() => setCount(count + 1)}>

Нажми на меня

</button>

</div>

);

}

Сравните его с кодом на Ember.js:

Скопировать кодHTML

//Ember.js

<div>

<p>Вы кликнули {{this.count}} раз(а)</p>

<button type="button" {{on "click" (fn this.change)}}>

Нажми на меня

</button>

</div>

Скопировать кодJSX

import Component from '@glimmer/component';

import { tracked } from '@glimmer/tracking';

import { action } from '@ember/object';

export default class CounterComponent extends Component {

@tracked count = 0;

@action

change() {

this.count = this.count + 1;

}

}

Или, например, с тем же кодом на Backbone.js:

Скопировать кодJSX

*//Backbone.js:*

var Counter = Backbone.Model.extend({

defaults: { value: 0 },

add: function() {

this.set({value: this.get('value')+1});

}

});

var cnt = new Counter();

var CounterView = Backbone.View.extend({

initialize: function() {

this.model.on('change', this.render, this);

},

events:{

'click .plus': 'add'

},

render: function() {

var html='<div><p>Вы кликнули '+ this.model.get('value')+' раз(а)</p>';

html += '<button class="plus">Нажми на меня</button><div>';

this.$el.html(html);

},

add: function() {

this.model.add();

}

});

var view = new CounterView({ model: cnt });

view.render();

Поддержка кода на React выглядит посильной задачей даже для новичка, тогда как для масштабирования приложения на других фреймворках потребуется провести минимум полдня наедине с документацией. Именно декларативный подход к разработке пользовательских интерфейсов сделал библиотеку React таким удобным и популярным инструментом.

В следующих уроках подробнее расскажем об особенностях React. В конце каждого урока вас ждут практические задания, которые помогут закрепить новые знания. Приступим!

# Подключение React

В этом уроке расскажем, как подключить React и написать с его использованием первый код. Вы уже знаете, что React — библиотека, созданная на JavaScript, а любую библиотеку можно добавить на HTML-страницу и воспользоваться её функционалом.

Мы подключим две библиотеки: react и react-dom. Основная библиотека react универсальна и предназначена не только для веб-проектов, а react-dom отвечает исключительно за взаимодействие с DOM. При работе с браузерным приложением нам потребуются обе эти библиотеки.

Подключим react и react-dom на HTML-страницу:

Скопировать кодHTML

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<script

src="https://unpkg.com/react@17/umd/react.development.js"

crossorigin

></script>

<script

src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"

crossorigin

></script>

</head>

<body>

</body>

</html>

Мы использовали версии библиотек с расширением .development.js — они содержат дополнительные инструменты для отладки кода на React.

Для начала работы с библиотекой React нам нужно добавить «точку входа», или «корневой узел». Так называют DOM-узел, в котором React управляет содержимым. Не удивляйтесь, что в HTML-документе этот узел пустой. Дело в том, что React использует концепцию Virtual DOM, то есть определяет структуру HTML в недрах JavaScript, а не напрямую в HTML-документе. Мы расскажем об этом подробнее в следующих уроках, а пока рассмотрим корневой узел:

Скопировать кодHTML

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<script

src="https://unpkg.com/react@17/umd/react.development.js"

crossorigin

></script>

<script

src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"

crossorigin

></script>

</head>

<body>

<div id="root"></div>

</body>

</html>

В примере используется <div id="root"></div>, но корневым может быть любой уникальный элемент. Обычно в приложениях, которые целиком написаны на React, существует только один корневой узел. Но бывает, что React встраивают в приложение частично, по кусочкам, в этом случае корневых узлов может быть несколько.

Вы уже готовы написать первое приложение на React. Для этого создайте .html файл и скопируйте в него код:

Скопировать кодHTML

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<script

src="https://unpkg.com/react@17/umd/react.development.js"

crossorigin

></script>

<script

src="https://unpkg.com/react-dom@17/umd/react-dom.development.js"

crossorigin

></script>

</head>

<body>

<div id="root"></div>

<script>

ReactDOM.render(

React.createElement("h1", null, "Привет, Мир!"),

document.getElementById('root')

);

</script>

</body>

</html>

Откройте этот файл в браузере, вы увидите заголовок h1 с текстом «Привет, Мир!». Этот код — один из самых простых примеров использования React.

Мы добавили тег <script></script> на страницу и вызвали внутри функцию ReactDOM.render с двумя аргументами. Первый её аргумент выглядит громоздко, но в следующих уроках вы узнаете, как описывать разметку способом, больше похожим на привычный HTML. Функция ReactDOM.render отвечает за рендеринг нашей разметки.

Процесс появления HTML-разметки на странице называется рендеринг (от англ. render — «изображать»). В следующих уроках подробнее расскажем о рендеринге и возможности React управлять разметкой, которая попадает в корневой узел.

# Рендеринг в React

Из предыдущего урока вы узнали, как подключить React и начать его использовать. Теперь расскажем, как осуществляется рендеринг, а разметка из React-кода попадает в HTML.

## React в браузере

[Мы уже вызывали функцию ReactDOM.render](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/c8988959-d7f2-4146-bdca-869a4f016911/). Именно с неё начинается любой веб-проект на React. Напомним, как она выглядит:

Скопировать кодJSX

ReactDOM.render(

React.createElement("h1", null, "Привет, React!"),

document.getElementById('root')

);

Функция принимает два аргумента. Первый — разметка, которую мы хотим поместить в корневой узел. Второй — указатель на DOM-узел, который будем использовать в качестве корневого.

Детальнее разберём первый аргумент функции ReactDOM.render. Это вызов ещё одной функции React.createElement, но уже из библиотеки react. Она вызывается с тремя аргументами:

1. Названием результирующего HTML-тега.
2. Дополнительными свойствами, которые прикреплены к этому тегу. Это может быть CSS-идентификатор class или id либо другие свойства.
3. Содержимым, которое попадёт в HTML-тег. В данном случае это строка "Привет, React!". Содержимым может быть не только текст, но и другие элементы.

После третьего аргумента могут идти и другие. Все они будут добавлены друг за другом внутрь создаваемого элемента, как если бы мы использовали метод appendChild. В следующих уроках подробнее расскажем об аргументах и их особенностях, а также наследовании и вложении элементов.

А пока рассмотрим пример:

Скопировать кодJSX

ReactDOM.render(

React.createElement(

"div",

null,

React.createElement("h1", null, "Текущее время:"),

React.createElement("h2", null, new Date().toLocaleTimeString())

),

document.getElementById('root')

);

Этот код выведет в корневой узел такую разметку:

Скопировать кодHTML

<div>

<h1>Текущее время:</h1>

<h2>11:54:14</h2>

</div>

Если обновлять страницу, таймер будет каждый раз показывать новое время. Правда сайт, который требует от пользователя постоянно нажимать F5, едва ли станет популярным.

Реальные приложения порой состоят из десятков тысяч элементов. В случае изменения элемента на верхнем уровне все его потомки будут перерисованы. К сожалению, браузеры устроены так, что манипуляции с DOM — трудозатратные процессы для пользовательских устройств. При этом нагрузка на процессор, оперативную память и аккумулятор значительно вырастет.

Как же React остаётся таким быстрым в этих условиях? Всё дело в хитрых механизмах оптимизации внутри библиотеки и VirtualDOM. React стремится изменять только тот кусок разметки, который нужно, и не трогать при этом остальной DOM.

Допишем предыдущий пример, чтобы обновление счётчика происходило без перезагрузки страницы:

Скопировать кодJSX

function timer() {

const element = React.createElement(

"div",

null,

React.createElement("h1", null, "Текущее время:"),

React.createElement("h2", null, new Date().toLocaleTimeString())

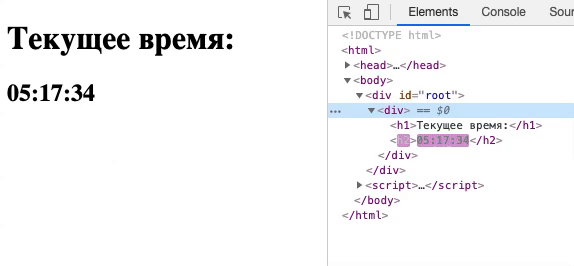
);

ReactDOM.render(element, document.getElementById('root'));

}

setInterval(timer, 1000);

Теперь раз в секунду будет вызываться функция timer, задача которой — рендерить DOM-дерево из нескольких узлов. Кажется, будто все элементы дерева будут отрендерены заново, раз в секунду, но это не так. Вот что происходит на самом деле:



Обновляется только элемент h2, который содержит в себе new Date().toLocaleTimeString(), а остальные элементы дерева не перерисовываются. ReactDOM сравнивает элемент и его дочернее дерево с предыдущей версией и вносит в DOM только необходимые изменения.

Элементы React иммутабельны: после создания элемента невозможно изменить его потомков или атрибуты. При иммутабельности можно только пересоздать элементы, что позволяет избавиться от множества ошибок, которые возникают при создании пользовательских интерфейсов. В React элемент можно представить как кадр киноплёнки: он отражает состояние интерфейса в конкретный момент времени. В рамках курса вы ещё не раз столкнётесь с иммутабельностью.

Вы уже знакомы с одним способом обновления интерфейса — созданием нового элемента через ReactDOM.render. На практике большинство приложений, написанных на React, вызывают ReactDOM.render только один раз — при инициализации. В этом курсе вы изучите и другие способы влияния на интерфейс. В следующем уроке расскажем о механизме, который позволяет обновить только изменившийся HTML-элемент — VirtualDOM.

# Virtual DOM

В предыдущем уроке мы разобрали особенности рендеринга разметки и преобразования ReactDOM в DOM. Настало время узнать, что же стоит за этим колдовством.

Как вы помните, элемент в React — простой объект. Рассмотрим код элемента:

Скопировать кодJSX

React.createElement(

"section",

null,

[

React.createElement(

"div",

null,

[

React.createElement("h1", null, "Заголовок"),

React.createElement("h2", null, "Подзаголовок")

]

),

React.createElement(

"div",

null,

React.createElement("p", null, "Параграф")

)

]

)

Так выглядит дерево элементов в React.

Этот код можно представить в упрощённом виде:

Скопировать кодJSX

{

element: "section",

children: [

{

element: "div",

children: [

{

element: "h1",

children: "Заголовок"

},

{

element: "h2",

children: "Подзаголовок"

}

]

},

{

element: "div",

children: [

{

element: "p",

children: "Параграф"

}

]

}

]

}

Здесь представлена древовидная структура объектов. В ней один корневой объект "section" содержит несколько дочерних — они указаны у него в children. А дочерние элементы тоже содержат свои children. У дерева объектов может быть огромное количество уровней вложенности. Именно такая структура передаётся в функцию ReactDOM.render, где и происходит магия Virtual DOM.

React использует эту структуру данных для двух целей:

1. На её основе строятся и вставляются на страницу реальные DOM-элементы (с помощью document.createElement и appendChild), а также вызывается рендер компонентов. В результате рендера компонентов это «дерево», а с ним и реальный DOM, «обрастают» дополнительными ветвями, структура которых была описана в компонентах.
2. React хранит в ней виртуальный DOM (англ. Virtual DOM) — виртуальную копию текущей структуры DOM. При последующих рендерах обновлённая структура поэлементно сравнивается с последней сохранённой копией. Это позволяет React определять нужное количество изменений и применять их в реальном DOM.

Такой подход к созданию HTML-разметки сложнее, чем написание HTML в .html файлах, но он необходим, потому что большинство операций с реальным DOM трудозатратны для браузера и пользовательских устройств. Алгоритм React, который называется “reconciliation” (англ. «согласование»), сводит количество таких операций к минимуму. Сравнение двух даже сложных JavaScript-объектов во много раз быстрее, чем аналогичные операции над DOM-деревом. Такой алгоритм позволяет React сохранить высокую скорость работы в приложениях с большим количеством меняющихся данных.

Вы можете прочитать [об особенностях рендеринга HTML-разметки движком браузера](https://developer.mozilla.org/ru/docs/Web/Performance/Critical_rendering_path#%D0%BF%D0%BE%D0%BD%D0%B8%D0%BC%D0%B0%D0%BD%D0%B8%D0%B5_%D1%8D%D1%82%D0%B0%D0%BF%D0%BE%D0%B2_crp), чтобы глубже разобраться в особенностях этого процесса. Эти знания будут полезны не только в курсе по React, но и в целом при любой веб-разработке.

# Знакомство c JSX

В предыдущих уроках вы подробно познакомились с функцией React.createElement. Но когда в коде много вложенных компонентов, его становится сложно читать. Да и в целом такое описание интерфейса больше похоже на императивный подход. А как вы помните, React славится своей декларативностью. В этом уроке рассмотрим общепринятый способ описывать интерфейсы декларативно с помощью JSX.

JSX — расширение языка JavaScript. С помощью babel оно преобразуется в стандартный JavaScript. [В онлайн-версии babel](https://babeljs.io/repl#?browsers=defaults%2C%20not%20ie%2011%2C%20not%20ie_mob%2011&build=&builtIns=false&spec=false&loose=false&code_lz=DwCwjAfI-CCAIggcIIJhBCsIIIRAA0ACASgUwIYDGALgITAD04EQA&debug=false&forceAllTransforms=false&shippedProposals=false&circleciRepo=&evaluate=false&fileSize=false&timeTravel=false&sourceType=script&lineWrap=true&presets=env%2Creact&prettier=false&targets=&version=7.13.9&externalPlugins=) можно наглядно увидеть, как происходят такие изменения. Вот так это можно представить:

Скопировать кодJSX

*// Создание элемента без использования JSX*

const element = React.createElement("h1", null, "Привет, React!");

*// Эквивалентная запись с использованием JSX*

const element = <h1>Привет, React!</h1>

Синтаксис JSX походит на язык шаблонов, по типу twig или handlebars, но наделён всеми возможностями JavaScript. JSX позволяет создавать гибкие шаблоны и описывать все возможные состояния интерфейса. В этом уроке рассмотрим основные конструкции, которые используются в JSX.

### Скобки

HTML-подобный код перемешивается с «настоящим» JavaScript. Это затрудняет чтение:

Скопировать кодJSX

ReactDOM.render(

<p>

Съешь же ещё этих мягких французских булок, да выпей чаю

</p>,

document.querySelector('#root')

);

Чтобы разделить код и вёрстку, рекомендуется обрамлять JSX в круглые скобки:

Скопировать кодJSX

ReactDOM.render((

<h1>

I Want To Believe 🛸

</h1>

), document.querySelector('#root'));

### Встраивание выражений

В примере мы объявляем переменную folderName и используем её внутри JSX, обрамляя фигурными скобками:

Скопировать кодJSX

const folderName = "Папка всем папкам папка";

ReactDOM.render(

<h1>

{folderName}

</h1>,

document.querySelector('#root')

);

Выражение может дополнять уже cуществующий контент:

Скопировать кодJSX

const person = "Арамишка";

ReactDOM.render((

<span>

Поздоровайся, это моя подружка {person}

</span>

), document.querySelector('#root'));

Часто вы будете видеть подобные конструкции:

Скопировать кодJSX

const onlyGoodNewsClassName = "good-news post only-good pet-news"

ReactDOM.render((

<section className={onlyGoodNewsClassName}>

Блок хороших новостей

</section>

), document.querySelector('#root'));

Свойство элемента в этом примере — константа onlyGoodNewsClassName. Использование этих возможностей ограничено, по большому счёту, только фантазией разработчика.

JSX позволяет и вызывать функции внутри разметки. Такие функции должны возвращать разметку или null и быть синхронными:

Скопировать кодJSX

const renderAddress = (street, house) => {

return (

<div>

<p>Улица {street}</p>

<p>Дом {house}</p>

</div>

);

}

ReactDOM.render((

<section>

<p>Я проживаю по адресу:</p>

{renderAddress("Пушкина", "Колотушкина")}

</section>,

), document.querySelector('#root'));

### Фрагменты

Иногда в JSX используется тег <React.Fragment>...</React.Fragment> или его более ёмкий вариант <>...</>.

Это так называемый «фрагмент» в React:

Скопировать кодJSX

ReactDOM.render((

<>

<div id="myElement">Click me!</div>

<div id="myAnotherElement">It was clicked!</div>

</>

), document.querySelector('#root'));

Дело в том, что блок JSX-кода по правилам должен содержать только один элемент верхнего уровня. В примере используются сразу два элемента, которые находятся рядом: #myElement и #myAnotherElement. Чтобы JSX-код работал правильно, мы «оборачиваем» эти два элемента во фрагмент.

Фрагменты не обладают дополнительной функциональностью и могут восприниматься как своеобразные «обрамляющие скобки» в JSX-коде. Вместо них можно было использовать обычный <div>, но это создаст лишние элементы в DOM — напрасная трата ресурсов, которая может вызывать дополнительные сложности с вёрсткой.

### Условная логика

Одна из самых частых задач в построении динамического интерфейса (да и вообще в программировании) — реализация условной логики. В JSX внутри фигурных скобок можно писать любые JavaScript-выражения, поэтому мы можем использовать операторы ?: и &&.

**Тернарный оператор** ?: — сокращённый аналог if/else:

Скопировать кодJSX

ReactDOM.render((

<div>

{isDaylight ? (

<h2>Добрый день!</h2>

) : (

<h2>Спокойной ночи!</h2>

)}

</div>

), document.querySelector('#root'));

Бывает, что в таких конструкциях возвращают null, в этом случае в DOM ничего не попадёт:

Скопировать кодJSX

ReactDOM.render((

<React.Fragment>

{userName ? (

<h2>Добро пожаловать, {userName}!</h2>

) : null}

</React.Fragment>

), document.querySelector('#root'));

**Оператор** && (логическое И) — аналог if, но без else — используется, если нужно отобразить часть JSX разметки только при определённом условии, а в противном случае вообще ничего не отображать:

Скопировать кодJSX

ReactDOM.render((

<div>

{isLunchTime && <h2>Время обеда!</h2>}

</div>

), document.querySelector('#root'));

Оба оператора могут содержать составные условия:

Скопировать кодJSX

ReactDOM.render((

<div>

{isThursday && wasRaining && <h2>Пора вернуть долг!</h2>}

</div>

), document.querySelector('#root'));

Скопировать кодJSX

ReactDOM.render((

<div>

{(isFrost && isSun) ? (

<h2>День чудесный</h2>

) : (

<h2>День обычный</h2>

)}

</div>

), document.querySelector('#root'));

### Свойства элементов

JSX очень похож на HTML, но есть некоторые важные отличия, которые придётся запомнить. Начнём с того, что для задания CSS-класса вместо class используется атрибут className:

Скопировать кодJSX

ReactDOM.render((

<div>

<div className="proletariat">

Шаг держи революционный!

Близок враг неугомонный!

Вперёд, вперёд, вперёд,

Рабочий народ!

</div>

<div className="bourgeoisie">

Ешь ананасы, рябчиков жуй,

День твой последний приходит, буржуй.

</div>

</div>

), document.querySelector('#root'));

Ещё в JSX отличается атрибут for у label. В JSX варианте требуется писать htmlFor вместо for:

Скопировать кодJSX

ReactDOM.render((

<div>

<label htmlFor="title">

<input id="title" type="text" />

</label>

</div>

), document.querySelector('#root'));

Для всех остальных свойств в JSX используется camelCase стиль вместо обычных имён HTML-атрибутов:

Скопировать кодJSX

onclick -> onClick

tabindex -> tabIndex

fill-rule -> fillRule

overline-thickness -> overlineThickness

stroke-width -> strokeWidth

onmouseover -> onMouseOver

### Стили

Одна из фич JSX — возможность задавать значения style в виде JavaScript-объектов. При этом в названиях CSS-свойств вместо kebab-case используется camelCase:

Скопировать кодJSX

const styles = {

width: 6792,

height: 6752,

borderRadius: '50%',

background: '#934838',

color: 'black',

};

ReactDOM.render((

<div style={styles}>Какая я планета?</div>

), document.querySelector('#root'));

Значение можно задать сразу без промежуточной переменной. Для этого используются двойные фигурные скобки {{...}}: внешние означают подстановку значения, а внутренние относятся к объекту, который описывает набор стилей:

Скопировать кодJSX

ReactDOM.render((

<div style={{

width: 3475,

height: 3472,

borderRadius: '50%',

background: '#d0d5d2',

color: '#444444',

}}>

Я тоже хочу быть планетой!

</div>

), document.querySelector('#root'));

Ещё один плюс: можно не указывать размерность для значений в пикселях — React сделает это сам.

### Самозакрывающиеся теги

Любой открывающий тег должен иметь свою пару — закрывающий тег. Если у элемента нет внутреннего содержимого, то тег должен быть самозакрывающимся. Для этого ставят слеш / перед его закрывающей скобкой:

Скопировать кодJSX

<img src="logo.png" />

### Комментарии в JSX

Как и в обычном JavaScript, в JSX можно оставлять комментарии. Для этого используется такая конструкция: {/\* Это элемент \*/}. Фигурные скобки помогают babel понять, что это не обычная строка, которую нужно поместить в DOM, а именно комментарий. В коде это может выглядеть так:

Скопировать кодJSX

ReactDOM.render((

<div>

{/\* Это комментарий к элементу label \*/}

<label htmlFor="title">

{/\*

Этот комментарий многострочный,

в нём можно написать подсказку для коллег.

\*/}

<input id="title" type="text" />

</label>

</div>

), document.querySelector('#root'));

В этом уроке вы познакомились с JSX. Это расширение языка JavaScript позволяет декларативно описывать пользовательский интерфейс. React основан на принципе, что логика рендеринга неразрывно связана с прочей логикой пользовательского интерфейса: обработкой событий, изменением состояния и подготовкой данных к отображению. JSX призван во всём соответствовать этому принципу. На этом способы применения JSX не заканчиваются — ещё больше вы узнаете по мере прохождения курса.

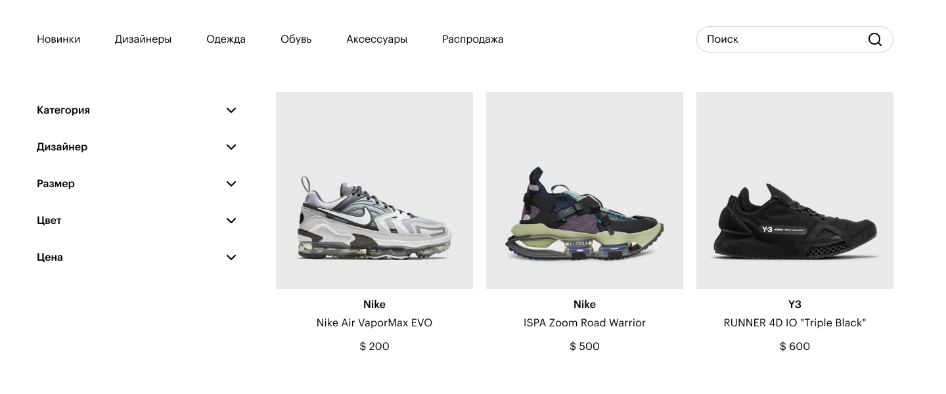
# Декомпозиция UI

Суть любой декомпозиции — в разделении большого и сложного на маленькое и простое. В этом уроке расскажем, как декомпозиция помогает при разработке пользовательских интерфейсов.

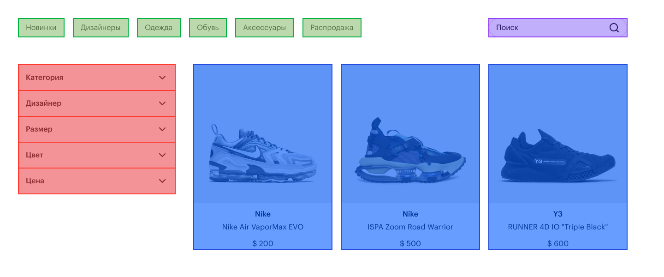
## Декомпозиция интерфейса

Декомпозиция интерфейса — процесс разделения приложения на отдельные блоки. Это позволяет добавлять функциональность, исправлять баги, реорганизовывать UI и управлять составными частями приложения независимо друг от друга. Разделение ответственности — одна из самых важных задач в программировании, и от её успешного решения зависит надёжность и масштабируемость системы. Почти как у Макиавелли: «Разделяй и властвуй».

Так можно визуализировать декомпозицию интерфейса на компоненты:



Так видит сайт обычный пользователь



А вот так его видит React-разработчик

Блоки одного цвета имеют общую структуру, но разное содержимое. Мы можем описать для них общий HTML, CSS и JavaScript в одном компоненте, присвоить ему имя и потом просто «призывать» его в нужном месте, меняя лишь содержимое.

Но как понять, должен ли элемент интерфейса стать компонентом? Когда стоит разбивать один компонент на несколько более мелких? Обычно это решение принимает разработчик. Но есть и несколько общих правил декомпозиции интерфейса:

1. Если приложение усложняется и код компонентов в редакторе разрастается на несколько экранов — время задуматься о декомпозиции.
2. Следуйте принципу DRY — Don't repeat yourself (англ. «не повторяйтесь»). Например, если код кнопки повторяется, стоит выделить его в самостоятельный компонент и использовать внутри других компонентов. Если позже захотите доработать такую кнопку, изменения применятся во всех местах её использования — в этом и смысл компонентов.
3. Мысленно разделите приложение на функциональные блоки. Например, в интернет-магазине это могут быть корзина, карточка товара, фильтр, личный кабинет, подвал сайта, поиск. Когда разобьёте приложение на части, сможете быстрее находить повторяющиеся элементы и декомпозировать пользовательский интерфейс.

class NavBar extends React.Component {

render() {

return (

<header className='header'>

{this.props.children}

</header>

);

}

}

class Logo extends React.Component {

render() {

return (

<img className='logo' src='./images/logo.svg' alt='логотип.' />

);

}

}

class Menu extends React.Component {

render() {

return (

<nav className='menu'>

{this.props.children}

</nav>

);

}

}

class MenuItem extends React.Component {

render() {

return (

<button className='item-btn' type='button'>{this.props.text}</button>

);

}

}

class Button extends React.Component {

render() {

return (

<button className='log-in-btn' type='button'>{this.props.text}</button>

);

}

}

class App extends React.Component {

render() {

return (

<div className='page'>

<NavBar>

<Logo/>

<Menu>

<MenuItem text='О НАС'/>

<MenuItem text='ЦЕНЫ'/>

<MenuItem text='БЛОГ'/>

</Menu>

<Button text='ВОЙТИ'/>

</NavBar>

</div>

);

}

}

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

# Компоненты в React

В предыдущем уроке вы познакомились с декомпозицией UI. Настало время поговорить о том, как работать с компонентами в React.

Суть разработки на React — создание компонентов пользовательского интерфейса, которые можно переиспользовать и комбинировать. Компоненты могут состоять из компонентов поменьше, а те — из ещё более мелких. С их помощью можно разрабатывать пользовательский интерфейс и не опасаться за уже готовые «запчасти». В этом уроке поговорим о разновидностях компонентов и особенностях их рендеринга.

## Функциональные и классовые компоненты

Компоненты в React во многом ведут себя как функции, которые может создавать разработчик. Как и функции, они принимают на вход аргументы и возвращают React-элементы, о которых мы рассказывали [в уроке про JSX](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/4a263618-0f56-4a01-9050-1d18d6f19403/). Аргументы компонентов позволяют переиспользовать компоненты в приложении и называются пропсы (подробнее о них расскажем чуть позже).

Вот самый простой способ создать React-компонент:

Скопировать кодJSX

function UserGreeting(props) {

return <h1>Здравствуйте, {props.fullName}</h1>;

}

*// или в виде стрелочной функции*

const UserGreeting = (props) => {

return <h1>Здравствуйте, {props.fullName}</h1>;

}

*// ещё компактнее*

const UserGreeting = props => <h1>Здравствуйте, {props.fullName}</h1>;

Функция в этом примере — React-компонент. Компонент получает на вход пропсы и возвращает React-элемент h1. Такие компоненты называются «функциональными», так как они и есть функции.

Компоненты можно объявить и иным способом. Для этого воспользуемся классами ES6:

Скопировать кодJSX

class UserGreeting extends React.Component {

render() {

return <h1>Здравствуйте, {this.props.fullName}</h1>;

}

}

Так какой же способ правильнее? Оба метода объявления компонентов активно применяются в реальных приложениях. Всё зависит от задач, которые должен решать компонент. На курсе мы не раз вернёмся к различиям и особенностям классовых и функциональных компонентов. А пока поговорим про их рендеринг.

## Как рендерятся компоненты в React

Разметка из JSX в функциональных компонентах попадает в DOM из выражения return:

Скопировать кодJSX

function Heading(props) {

return <h1>Заголовок новой статьи</h1>

}

Разметка из классовых компонентов попадает в DOM из выражения return, которое находится внутри метода render:

Скопировать кодJSX

class Heading extends React.Component {

render(){

return <h1>Заголовок новой статьи</h1>

}

}

Рендеринг всего дерева элементов в большинстве приложений на React начинается с вызова ReactDOM.render. Первым аргументом передаётся элемент, который включает все остальные компоненты. В итоге получается древовидная структура из вложенных друг в друга компонентов.

Чтобы не запутаться в понятиях и понимать разницу между элементом и компонентом, рассмотрим их определения:

Компонент — функция, строительный блок пользовательского интерфейса.

Элемент — результат выполнения функции, то есть разметка, возвращаемая из компонента.

Важно отметить, что компонент может быть элементом, но не наоборот. Вот наглядный пример:

Скопировать кодJSX

// Это компонент

function UserPage(props) {

return (

{/\* Это элемент \*/}

<div>

{/\* Это элемент \*/}

<h1>Страница пользователя</h1>

{/\* Avatar — это компонент, который является элементом внутри UserPage \*/}

<Avatar src={props.user.photo}>

</div>

);

}

[В уроке про JSX](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/4a263618-0f56-4a01-9050-1d18d6f19403/) мы показывали, как рендерятся React-элементы, которые представляют собой HTML-теги:

Скопировать кодJSX

const element = (

<div>

<h1>I Want To Believe 🛸</h1>

</div>

);

const anotherElement = <p>Лорем ипсум</p>;

Но элементами могут быть наши собственные компоненты:

Скопировать кодJSX

const element = <RegistrationForm />

const anotherElement = <UserGreeting fullName="Павел Валерьевич" />;

Когда React встречает подобный элемент, он собирает все JSX-атрибуты в один объект и передаёт их компоненту. Именно этот объект называется «пропсы» (props). Объект props для компонента UserGreeting выглядит так:

Скопировать кодJSX

{

fullName: 'Павел Валерьевич'

}

Рассмотрим пример:

Скопировать кодJSX

function UserGreeting(props) {

return <h1>Здравствуйте, {props.fullName}</h1>;

}

const element = <UserGreeting fullName="Павел Валерьевич" />;

ReactDOM.render(

element,

document.getElementById('root')

);

Этот код выведет на страницу фразу «Здравствуйте, Павел Валерьевич».

Вот что происходит в коде:

1. Мы передаём React-элемент <UserGreeting fullName="Павел Валерьевич" /> в ReactDOM.render().
2. React вызывает компонент UserGreeting с пропсами {fullName: 'Павел Валерьевич'}.
3. Компонент UserGreeting возвращает элемент <h1>Здравствуйте, Павел Валерьевич</h1> в качестве результата.
4. React DOM делает свою работу и по итогу помещает <h1>Здравствуйте, Павел Валерьевич</h1> в DOM.

Название компонентов всегда должно начинаться с прописной буквы. Так React понимает, когда имеет дело с HTML-тегом, а когда — с компонентом:

Скопировать кодJSX

*// Неправильно! Это компонент, и он должен быть записан с заглавной буквы:*

function userGreeting(props) {

*// Правильно! Использование <h1> разрешено, так как это валидный HTML-тег:*

return <h1>Здравствуйте, {props.fullName}</h1>;

}

*// Неправильно! React думает, что <userGreeting /> — это HTML-тег,*

*// потому что он записан с маленькой буквы:*

const element = <userGreeting fullName="Антонина Николаевна" />

*// Произойдёт ошибка*

ReactDOM.render(

element,

document.getElementById('root')

);

Чтобы React корректно распознал компонент, мы должны переименовать userGreeting в UserGreeting:

Скопировать кодJSX

*// Правильно! Это компонент, и он должен быть записан с заглавной буквы:*

function UserGreeting(props) {

*// Правильно! Использование <h1> разрешено, так как это валидный HTML-тег:*

return <h1>Здравствуйте, {props.fullName}</h1>;

}

*// Правильно! React знает, что <UserGreeting /> — это компонент,*

*// потому что он написан с заглавной буквы:*

const element = <UserGreeting fullName="Антонина Николаевна" />

ReactDOM.render(

element,

document.getElementById('root')

);

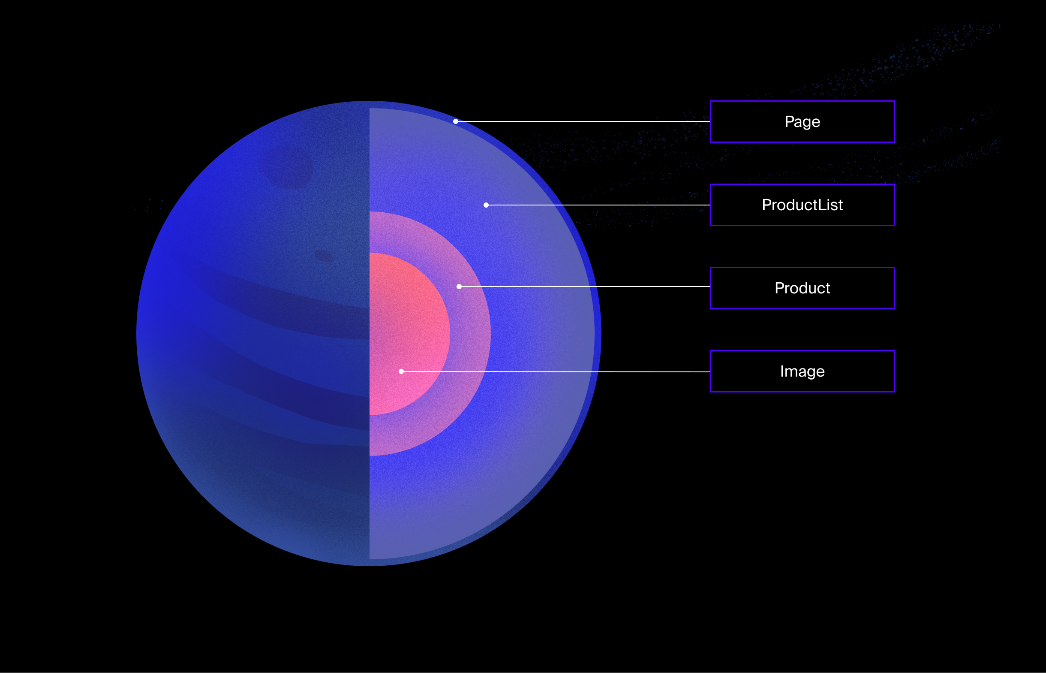
В этом уроке вы освоили главную концепцию React — компоненты. Мы подготовили несколько практических заданий для её закрепления. Впереди вас ждёт не менее важная тема — извлечение компонентов. Эта техника сделает разработку приятной и предсказуемой.

# Композиция и извлечение компонентов в React

В предыдущем уроке мы рассказали про основную идею React — компоненты. Благодаря им на React разрабатывают большие и сложные приложения, которые легко поддерживать. В этом уроке разберём, как использовать компоненты друг с другом и выделять элементы интерфейса в компоненты.

## Композиция компонентов

Сила React — в возможности вкладывать одни компоненты в другие, которые, в свою очередь, могут состоять из ещё более мелких компонентов, и так до самых примитивных HTML-тегов. Структура React-приложения напоминает луковицу или планету:



Эта особенность React позволяет использовать одну и ту же абстракцию на любом уровне приложения. Следите за руками: компонентом может быть изображение товара, карточка товара, список товаров или даже страница категории товаров целиком. Всё это — компоненты, которые содержат другие компоненты.

Например, страницу интернет-магазина можно упрощённо представить так:

Скопировать кодJSX

function Cart(props) {

return <div>Корзина товаров пуста</div>;

}

function Header(props) {

return (

<header>

<h1>Магазин глазиrawанных сырков</h1>

<Cart />

</header>

);

}

function Products(props) {

*// в этом компоненте будет описан рендеринг списка товаров*

}

function Filters(props) {

*// в этом компоненте будет описан рендеринг фильтров для товаров*

}

function Footer(props) {

return <div>Компания существует с 1999 года</div>;

}

function Shop() {

return (

<>

<Header/>

<div style={{

display: "flex",

alignItems: "flex-start"

}}>

<Filters />

<Products />

</div>

<Footer/>

</>

);

}

ReactDOM.render(

<Shop />,

document.getElementById('root')

);

Эта возможность библиотеки React делает разработку пользовательских интерфейсов быстрой и увлекательной.

Компоненты можно использовать многократно:

Скопировать кодJSX

function UserGreeting(props) {

return <h1>Здравствуйте, {props.fullName}</h1>;

}

function ProducerParty(props) {

return (

<div>

<UserGreeting fullName="Паоло Соррентино" />

<UserGreeting fullName="Дэвид Линч" />

<UserGreeting fullName="Ридли Скотт" />

<UserGreeting fullName="Лана Вачовски" />

<UserGreeting fullName="Гай Ричи" />

<UserGreeting fullName="Гаспар Ноэ" />

</div>

);

}

ReactDOM.render(

<ProducerParty />,

document.getElementById('root')

);

В этом примере мы персонально поприветствовали режиссёров кино.

## Извлечение компонентов

Старайтесь разбивать приложение на части. Так его проще поддерживать и развивать.

Представим, что мы работаем над неким компонентом Order:

Скопировать кодJSX

function Order(props) {

return (

<div className="Order">

<div className="Customer">

<img

className="Avatar"

src={props.customer.picture}

alt={props.customer.fullName}

/>

<div className="Customer-fullName">{props.customer.fullName}</div>

</div>

<div className="Order-total">{formatPrice(props.total)}</div>

<div className="Order-date">{props.date}</div>

</div>

);

}

Этот компонент используется для отображения карточки заказа в панели управления интернет-магазином. В качестве пропсов он принимает объект такого вида:

Скопировать кодJAVASCRIPT

{

customer: {

fullName: "Elon Musk",

picture: "https://static01.nyt.com/images/2019/11/01/multimedia/01xp-elonmusk/merlin\_162496989\_edfc836d-1bcf-45b1-8c42-68eaaac5d1de-superJumbo.jpg"

}

total: 23190,

date: "11.10.2021 15:44"

}

Из-за вложенности объектов в пропсах с этим компонентом может быть не очень удобно работать. Такой компонент можно разбить на более мелкие. Попробуем выделить аватар пользователя в отдельный компонент:

Скопировать кодJSX

function CustomerPicture(props) {

return (

<img

className="Avatar" src={props.customer.picture}

alt={props.customer.fullName}

/>

)

}

После этой доработки код компонента Order выглядит так:

Скопировать кодJSX

function Order(props) {

return (

<div className="Order">

<div className="Customer">

<CustomerPicture customer={props.customer} />

<div className="Customer-fullName">

{props.customer.fullName}

</div>

</div>

<div className="Order-total">

{formatPrice(props.total)}

</div>

<div className="Order-date">

{props.date}

</div>

</div>

);

}

Работа с компонентом Order стала удобнее, но имя пользователя всё равно берётся из вложенного объекта props. Попробуем ещё упростить Order. Для этого объединим информацию о покупателе в одном компоненте:

Скопировать кодJSX

function CustomerInfo(props) {

return (

<div className="Customer">

{/\* Используем компонент для отображения аватара \*/}

<CustomerPicture сustomer={props.сustomer} />

<div className="Customer-username">

{props.customer.fullName}

</div>

</div>

);

}

Получилось отлично! Теперь компонент Order выглядит так:

Скопировать кодJSX

function Order(props) {

return (

<div className="Order">

<CustomerInfo customer={props.customer} />

<div className="Order-total">

{formatPrice(props.total)}

</div>

<div className="Order-date">

{props.date}

</div>

</div>

);

}

В этом примере мы выделили самостоятельную единицу данных из пропсов и сделали для неё самодостаточный компонент CustomerInfo.

Разделять компоненты на более мелкие стоит, если:

* одна и та же функциональность используется многократно: например, Modal, Button, Avatar;
* функциональность компонента сама по себе сложная: корзина и фильтры в интернет-магазине, лента новостей в социальных сетях.

В больших приложениях нужно иметь набор компонентов, которые используются многократно. Это позволяет сократить время на разработку, поддержку и обновление приложения. Кроме того, при обновлении компонента, который используется многократно, например Button, вы можете быть уверенными, что Button поменяется во всех местах использования.

В этом уроке вы научились комбинировать React-компоненты друг с другом и объединять в компоненты связанные по смыслу блоки пользовательского интерфейса. В рамках этого курса вам предстоит создать множество разных компонентов и узнать про их возможности взаимодействия. В следующем уроке мы детальнее разберём работу с пропсами.

# Продвинутое использование пропсов

В предыдущем уроке мы рассказали о компонентах в React и компонентном подходе к созданию пользовательских интерфейсов. В этом уроке изучим пропсы (props) и их особенности.

Пропсы — по сути аргументы функции, которые позволяют влиять на рендеринг элементов внутри компонентов и логику их работы. Как и аргумент, пропс может быть чем угодно: функцией, объектом, массивом, строкой, числом.

Вы уже знаете, что React при работе с компонентами объединяет JSX-атрибуты в один объект — props. В классовом компоненте доступ к этому объекту можно получить через this.props, а в функциональном props доступен в качестве первого аргумента функции:

Скопировать кодJSX

function Heading(props) {

return <h1>{props.text}</h1>

}

ReactDOM.render(

<Heading text="Заголовок интересной статьи" />,

document.querySelector('#root');

)

В качестве имени аргумента принято использовать “props”. Рекомендуем называть аргумент именно так: это упростит восприятие вашего кода другими разработчиками, ведь многие из них тоже следуют этому принципу.

В уроке [про композицию и извлечение компонентов](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/12b222c8-3b89-434a-9aa9-5ebba7dde0b2/) вы уже видели, что пропсов может быть несколько и их структура допускает вложенность:

Скопировать кодJSX

const Product = (props) => (

<div>

<p>{props.productData.name}</p> {/\* Гидрокостюм для дайвинга \*/}

<span>{props.productData.price}</span> {/\* 14299 \*/}

</div>

);

const ShoppingCart = (props) => (

<>

<h1>Корзина товаров</h1>

<Product productData={{ name: "Гидрокостюм для дайвинга", price: 14299 }} />

</>

);

В этом примере компонент ShoppingCart «рендерит» другой компонент — Product. В качестве аргументов (пропсов) ему передаётся объект с двумя значениями: name и price.

Чтобы сделать код более читаемым, стоит упростить структуру передаваемых в компонент пропсов и использовать в качестве аргументов примитивы. Едва ли это отразится на производительности, ведь React очень быстрый, но работать с примитивами проще.

Взгляните на пример, в котором props — сложный объект:

Скопировать кодJSX

*// Компонент*

const Product = (props) => (

<div>

<p>{props.productData.name}</p> {/\* Гидрокостюм для дайвинга \*/}

<span>{props.productData.price}</span> {/\* 14299 \*/}

</div>

);

*// Использование компонента*

<Product productData={{ name: "Гидрокостюм для дайвинга", price: 14299 }} />

Код, использующий этот компонент, выглядит сложно, а внутренняя реализация требует большей внимательности от разработчика. Давайте упростим этот компонент:

Скопировать кодJSX

*// Компонент*

const Product = (props) => (

<div>

<p>{props.name}</p> {/\* Гидрокостюм для дайвинга \*/}

<span>{props.price}</span> {/\* 14299 \*/}

</div>

);

*// Использование компонента*

<Product name="Гидрокостюм для дайвинга" price={14299} />

Из примера видно, что реализация стала проще, а использование компонента — более декларативным.

## Пропсы можно только читать

React-компоненты обязаны вести себя как «чистые» функции по отношению к своим пропсам вне зависимости от того, функциональные они или классовые. Это означает, что компоненты не должны что-то записывать в свои пропсы. Разберём это на примере «чистой» и «нечистой» функций.

Функция formatPrice — «чистая», потому что не изменяет свои входные данные и предсказуемо возвращает одинаковый результат для одних и тех же аргументов:

Скопировать кодJSX

function formatPrice(price) {

return price + '₽';

}

А вот функция notificationSettings — «нечистая», так как записывает данные в свой же аргумент — объект user:

Скопировать кодJSX

function notificationSettings(user, notificationValue) {

user.notifications = notificationValue;

}

Чтобы разработка на React оставалась предсказуемой и удобной, пропсы должны быть доступны только для чтения. В следующих уроках мы расскажем, как повлиять на props компонента изнутри этого компонента.

## JavaScript-выражения как пропсы

В качестве пропсов можно передавать любые JavaScript-выражения. Для этого нужно обернуть их в {}. К примеру, в этом JSX:

Скопировать кодJSX

<MusicGenreItem genre={'rock' + '&' + 'roll'} />

В компоненте MusicGenreItem значение props.genre равно rock&roll, потому что выражение 'rock' + '&' + 'roll' будет вычислено. То же самое справедливо для функций:

Скопировать кодJSX

function sayHi(name) {

return `Привет, ${name}!`

}

<WelcomeComponent textToRender={sayHi('Варвара')} />

В этом примере в пропс textToRender попадёт результат выполнения функции sayHi('Варвара'), то есть строка "Привет, Варвара!".

Операторы if, switch и цикл for  не являются выражениями в JavaScript. Из-за этого их невозможно напрямую применять в JSX. Чтобы их использовать, необходимо сначала получить результат работы этих операторов, а затем этот результат поместить в JSX-код:

Скопировать кодJSX

function TrafficLightComponent(props) {

let action;

switch (props.color) {

case "red":

action = <b>Стоять</b>

case "yellow":

action = <i>Завершайте манёвр</i>

case "green":

action = <strong>Проезжайте</strong>;

default:

action = <span>Будьте внимательны, светофор не работает</span>

}

return <div>Светофор рекомендует: {action}</div>;

}

## Строковые литералы как пропсы

Строковые литералы также могут быть использованы в качестве пропсов. К примеру, эти компоненты эквивалентны:

Скопировать кодJSX

<WelcomeComponent name="Екатерина" />

<WelcomeComponent name={'Екатерина'} />

<WelcomeComponent name={`Екатерина`} />

<WelcomeComponent name='Екатерина' />

Если передавать в компонент строковый литерал, то все его возможные символы будут преобразованы в соответствующие HTML-сущности. Оба этих компонента эквивалентны:

Скопировать кодJSX

<FavoritePet pet="&#128008;" />

<FavoritePet pet={'🐈'} />

## Значение пропсов по умолчанию равно true

Если не передать значение в пропс, по умолчанию будет true. Эти два JSX выражения эквивалентны:

Скопировать кодJSX

<TextInput disabled />

<TextInput disabled={true} />

Такое поведение пропсов в React-компонентах существует, чтобы соответствовать поведению обычного HTML. Лучше не полагаться на такую особенность и указывать значения props в явном виде, ведь иначе такая запись может быть интерпретирована React как [короткая нотация свойств объекта](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Object_initializer#%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D1%81%D0%B2%D0%BE%D0%B9%D1%81%D1%82%D0%B2):

Скопировать кодJAVASCRIPT

const title = "Автостопом по галактике"

{ title }

Такая запись эквивалентна объекту { title: "Автостопом по галактике" }, но точно не объекту { title: true }.

## Атрибуты расширения

Вложенная структура пропсов — частое явление, особенно при работе с сервером, который возвращает многоуровневый JSON. Чтобы сделать код более компактным и читаемым, можно использовать оператор расширения. В этом примере оба компонента идентичны:

Скопировать кодJSX

function CustomerPage(props) {

return (

<ProfileInfo

firstName={props.profileData.firstName}

lastName={props.profileData.lastName}

/>

);

}

function CustomerPage(props) {

return <ProfileInfo {...props.profileData} />;

}

Оператор расширения помогает отсеивать нужные пропсы и передавать в целевой компонент только необходимые данные:

Скопировать кодJSX

const Input = props => {

const { size, userId, ...otherProps } = props;

const className = size === "default" ? "DefaultInput" : "SmallInput";

return <input className={className} {...otherProps} />;

};

const LandingPage = () => {

return (

<>

<b>Введите промокод:</b>

<Input size="default" type="text" disabled={false} userId={1112983} />

</>

);

};

В этом примере в компонент Input передаются такие пропсы:

Скопировать кодJSX

{

size: "default",

type:"text",

disabled: false,

userId: 1112983

}

При этом пропсы size и userId используются безопасно и не передаются в DOM-элемент <input/>. Пропс size используется внутри для вычисления класса, а userId не задействован вовсе. Поэтому в объекте otherProps остаются только нужные DOM-элементу пропсы — type и disabled.

Атрибуты расширения могут быть полезны и сокращают код, но есть риск передать лишние пропсы в компоненты или невалидные HTML-атрибуты в DOM. Используйте этот инструмент с осторожностью.

## Пропсы: коротко о главном

В этом уроке мы погрузились в возможности пропсов React-компонентов. Напомним самое главное:

1. Пропсы можно только читать.
2. Пропсами могут быть JavaScript выражения.
3. По умолчанию значения пропсов выставляется в true.
4. Пропсы можно передать пачкой с помощью оператора расширения.

React-разработчик применяет эти навыки регулярно, а умение правильно «приготовить» пропсы сильно влияет на качество и надёжность кода. В следующем уроке расскажем о внутреннем состоянии и разберём, как применять его вместе с пропсами.

# Декомпозиция пропсов

В предыдущих уроках мы рассказывали про компоненты, их извлечение и композицию, а также о продвинутых техниках использования пропсов. В этом уроке разберём, как не допустить возврата в эпоху императивной разработки при использовании пропсов в сложных компонентах.

По мере развития приложения возникает необходимость добавлять в компонент дополнительные пропсы. Сначала один, потом другой, не успеете оглянуться, как окажется, что компонент ждёт десяток другой аргументов.

Взгляните на пример сложного компонента OrderContent:

Скопировать кодJSX

// Плохой пример

<OrderContent

orderId={"ADF-21244324-11"}

addressCoords={[56.033661, 37.859848]}

addressName="ул. 26 Бакинских Комиссаров, дом 11"

mapProvider="yandex"

products={

[

{ id: 12, name: "Ватные диски", count: 2 },

{ id: 45, name: "Корм для рыбок", count: 1 }

{ id: 78, name: "Гранатовый сок", count: 4 }

]

}

time="25.09.2021 14:33:24"

customerName="Евгения Лучинова"

customerEmail="email.pochta@example.com"

customerPhone={"+7 (495) 777-77-77"}

productsView="list"

totalPrice={1293.30}

favoriteProduct={{ id: 78, name: "Гранатовый сок" }}

onCancelOrder={

showNotification(

"Вы уверены, что хотите отменить заказ?",

{ onConfirm: cancelOrder, onCancel: Modal.active.close }

)

}

onSubmitOrder={proceedToPayement}

onChangeFavoriteProduct={setFavorite}

onChangeProductView={toggleView}

onApplyCode={

showPrompt(

"Введите промокод:",

{ onSubmit: applyCode, onCancel: Modal.active.close }

)

}

shouldSaveAddress={true}

/>

Поддерживать такой код трудно, а порой и опасно: одно неверное движение, и весь компонент перестанет работать, а внутренняя реализация такого компонента будет занимать не одну сотню строк кода.

React не ограничивает ни тип пропсов, ни их количество, но злоупотреблять ими не стоит, гораздо лучше прибегнуть к декомпозиции:

Скопировать кодJSX

// Хороший пример

<OrderContent>

<OrderInfo time="25.09.2021 14:33:24" orderId={"ADF-21244324-11"} />

<Customer {...customer} />

<Products data={products} />

<Address

addressCoords={[56.033661, 37.859848]}

addressName="ул. 26 Бакинских Комиссаров, дом 11"

mapProvider="yandex"

/>

<PromoCodeComponent />

<span className="total-price">1293.30</span>

<OrderControls />

</OrderContent>

В этом примере много компонентов, а пропсы разделены по смыслу: всё, что связано с адресом, мы передаём компоненту Address, у компонента Customer тоже своя узкая задача и так далее. Такой код легко поддерживать, а добавление новых функциональных возможностей не подвергает опасности работоспособность всего приложения.

Когда пропсов становится слишком много — это знак, что стоит применить декомпозицию. Старайтесь не допускать их «разрастания», когда пишете код. Это сэкономит время и нервы в будущем, и коллеги будут вам благодарны. В следующем уроке расскажем про ещё один приём, который может быть полезен при декомпозиции.

# Продвинутое использование JSX. React.Children

Вы уже знаете, что JSX очень похож на HTML: большинство элементов могут иметь открывающий и закрывающий теги. В случае с JSX всё, что написано между открывающим и закрывающими тегами, попадает в пропсы элемента. React использует специальный ключ в объекте props — children.

Разберём на примере:

Скопировать кодJSX

const FancyParagraph = (props) => (

<p className={"text-beautiful"}>

{props.children}

</p>

);

const CoolShinySwagText = (props) => (

<>

<h1>Вау! Какой красивый текст:</h1>

<FancyParagraph>Этот параграф выглядит очень красиво.</FancyParagraph>

<FancyParagraph>А этот параграф ничуть не хуже предыдущего.</FancyParagraph>

</>

);

React помещает текст, который написан между <FancyParagraph> и </FancyParagraph>, в children. А обратиться к значению этого пропса можно уже в дочернем компоненте через props.children.

Пытливый ум задаётся вопросом: что будет, если children указать как props у компонента, а не описывать внутри тега? Если так сделать, ничего страшного не произойдёт, но children, описанный между тегами элемента, будет приоритетнее чем children, который задан как props:

Скопировать кодJSX

*// Будет выведено: Пицца с ананасами прекрасна!*

<FancyParagraph children={"Пицца с ананасами ужасна!"}>

Пицца с ананасами прекрасна!

</FancyParagraph>

*// Будет выведено: Пицца с ананасами ужасна!*

<FancyParagraph children={"Пицца с ананасами ужасна!"} />

В некоторых случаях props.children помогает упростить код и сделать его более читаемым. Посмотрим на пример, в котором используется notificationText в пропсах для отображения предупреждающего сообщения в компоненте Notification:

Скопировать кодJSX

*// Компонент*

const Notification = (props) => (

<div className="Notification">

<p>{props.notificationText}</p> {/\* Внимание! Обнаружено повышение радиации в 4-м блоке \*/}

</div>

);

*// Использование компонента*

<Notification

notificationText="Внимание! Обнаружено повышение радиации в 4-м блоке"

/>

В этом примере нет ничего плохого (кроме радиации), но нужно помнить, что компонент ожидает на вход notificationText. Это создаёт дополнительную когнитивную нагрузку и отнимает время.

Вместо этого мы можем воспользоваться props.children и сделать код более читаемым:

Скопировать кодJSX

*// Компонент*

const Notification = (props) => (

<div className="Notification">

<p>{props.children}</p> {/\* Внимание! Обнаружено повышение радиации в 4-м блоке \*/}

</div>

);

*// Использование компонента*

<Notification>

Внимание! Обнаружено повышение радиации в 4-м блоке

</Notification>

Этот вариант выглядит естественнее, так как больше походит на HTML. В каких случаях использовать такой подход — решать вам.

## Ещё немного о props.children

Обратите внимание, что в children попадает всё, что расположено между открывающим и закрывающим тегами элемента. Это предоставляет дополнительные возможности для создания UI:

Скопировать кодJSX

const DropdownMenu = (props) => (

<div className="DropdownMenu">

<span className="DropdownMenu-heading">{props.heading}</span>

<div className="DropdownMenu-items">

{props.children}

</div>

</div>

);

const NavBar = () => (

<nav>

<Logo />

<DropdownMenu>

<ul>

<li className="DropdownMenu-item">О магазине</li>

<li className="DropdownMenu-item">Доставка</li>

<li className="DropdownMenu-item">Оплата</li>

</ul>

<hr />

<ul>

<li className="DropdownMenu-item">Оставить отзыв</li>

<li className="DropdownMenu-item">Политика конфиденциальности</li>

</ul>

</DropdownMenu>

</nav>

)

В этом примере меню на сайте формируется в компоненте NavBar. Затем с соблюдением HTML-разметки рендерится в компоненте DropdownMenu с помощью props.children. Такое меню можно использовать в разных местах сайта с разным содержимым и быть уверенным, что оно будет вести себя одинаково.

Умение пользоваться props.children позволяет создавать более податливый к кастомизации пользовательский интерфейс. В следующем уроке вас ждёт увлекательное путешествие в мир состояния в веб-приложениях, написанных с использованием библиотеки React. А пока перейдём к заданиям.

const Quote = (props) => {

return (

<div className="fancy-quote">

<p className='text'>{props.children}</p> {/\* Переписать на props.children \*/}

<span className='author'>{props.author}</span>

</div>

);

}

const DuneQuote = () => {

return (

<Quote

author="Принцесса Ирулан «Избранные&nbsp;изречения Муад'Диба»">

"Арракис учит «пониманию ножа», учит относиться к жизни так: отрезать всё несовершенное и незавершённое, говоря: «Вот теперь это совершенно и завершено — ибо кончается здесь»."

</Quote>

);

}

ReactDOM.render(<DuneQuote />, document.querySelector('#root'))

В этой задаче поработаем над блоком меню, которое отображается в подвале сайта. Компонент MenuBlock рендерит пункты меню из пропса menuItems, а заголовок берёт из children. Но гораздо удобнее было бы указывать пункты меню как children, а заголовок пунктов меню брать из пропса menuHeading. Перепишите компонент, чтобы упростить его применение.

const MenuBlock = props => {

return (

<div className="menu">

<div className="content">

<p className="menu\_\_title">{props.menuHeading}</p>

{props.children}

</div>

</div>

);

};

const Footer = () => {

return (

<div className="footer">

<div className="container">

<img className="menu\_\_img" src="./images/bg.svg" alt="Подвал" />

<MenuBlock menuHeading="О компании">

<ul className="nav">

<li className="nav\_\_link">Наша миссия</li>

<li className="nav\_\_link">Вакансии</li>

<li className="nav\_\_link">Блог</li>

</ul>

</MenuBlock>

</div>

</div>

);

};

ReactDOM.render(<Footer />, document.querySelector('#root'));

# Внутреннее состояние компонентов

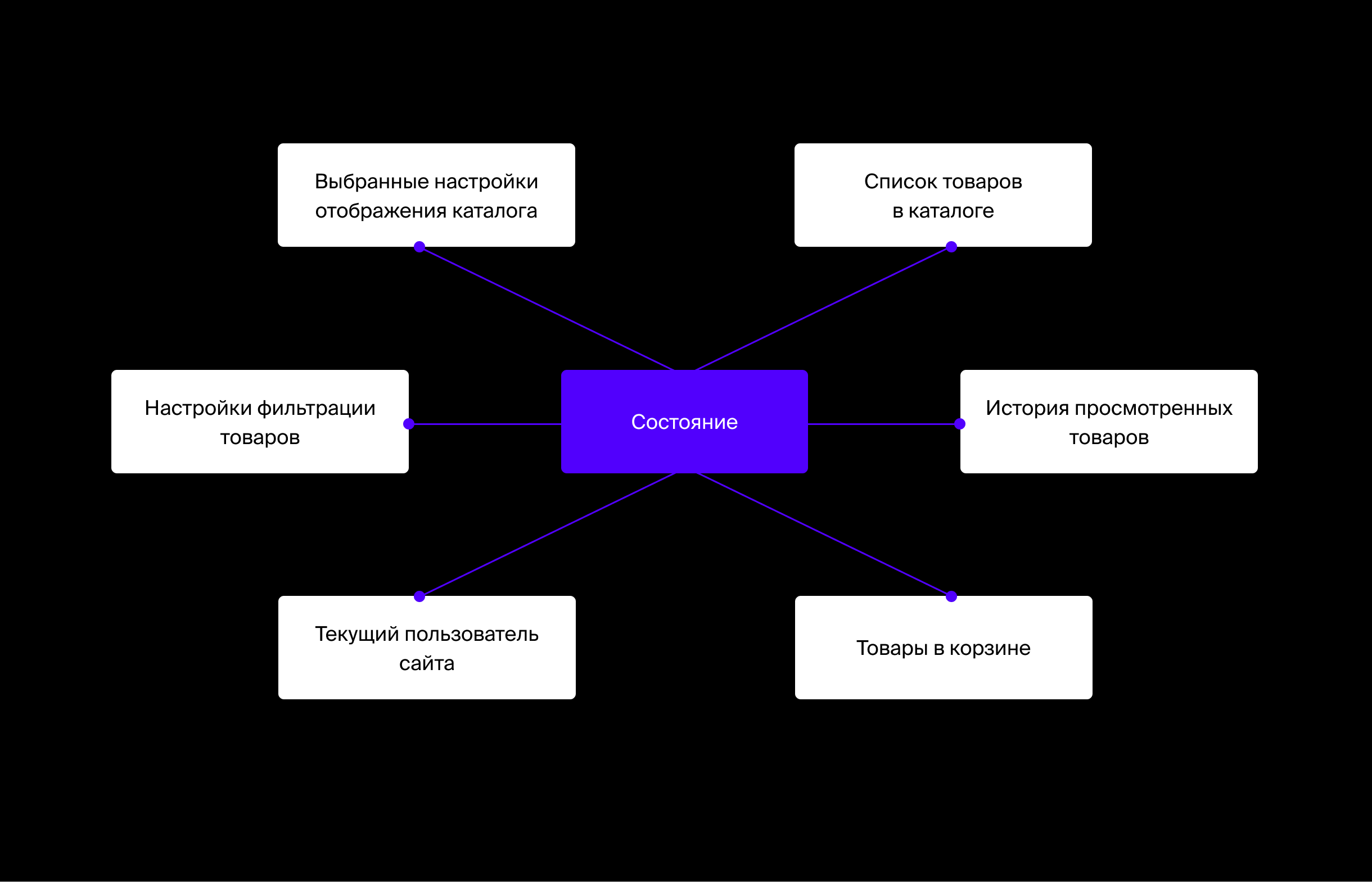
В предыдущих уроках вы разобрались с компонентами и их пропсами в React. Вы уже знаете, что влиять на логику работы и отображение компонентов можно через пропсы. В React есть ещё один инструмент для влияния на компонент — внутреннее состояние компонента.

Благодаря состоянию компоненты могут реагировать на действия пользователя, ответы сервера и другие события и при этом не нарушать принцип чистоты компонентов. Но начнём с основ — расскажем, что такое внутреннее состояние компонентов в React.

## Что такое внутреннее состояние

Состояние окружает нас в повседневной жизни: компьютер **включён**, время суток — **вечер**, музыка — **играет**. В пользовательских интерфейсах, у компонентов и созданных с их помощью приложений тоже может быть состояние: меню **раскрыто**, тёмная тема оформления **включена**, пользователь **авторизован**, данные **загружаются.**

В интернет-магазине список товаров, состав корзины, наличие активных заказов, применённые фильтры и промокод — всё это состояние.



Оно может меняться от действий пользователя и внутренних процессов в приложении, а эти изменения влияют на логику работы и отображение пользовательского интерфейса. Библиотека React предоставляет мощный инструментарий для управления внутренним состоянием компонентов и построения динамических пользовательских интерфейсов.

## Состояние в React-компонентах

В React внутреннее состояние классового компонента задаётся с помощью свойства state. Компонент перерисовывается, когда state изменяется. Состояние очень похоже на пропсы, за исключением того, что оно задаётся внутри компонента, а не приходит снаружи. Чуть позже покажем, как state и props работают вместе.

Вот пример компонента с внутренним состоянием:

Скопировать кодJSX

class ThemeToggle extends React.Component {

state = { theme: "светлая" };

render() {

return (

<div>

<p>Включена {this.state.theme} тема</p>

</div>

);

}

}

Компонент содержит в себе состояние: { theme: "светлая" }. Опираясь на это состояние, можно создавать потрясающие вещи, например добавить тёмную тему в приложение:

Скопировать кодJSX

class ThemeToggle extends React.Component {

state = { theme: "светлая" };

render() {

return (

<div className={this.state.theme === "светлая" ? "th-light" : "th-dark"}>

<p>Включена {this.state.theme} тема</p>

</div>

);

}

}

В этом примере применяется разный класс к элементу div в зависимости от значения this.state.theme. А этот класс, в свою очередь, влияет на цветовую схему.

## Иерархия и форма состояния в React-компонентах

Как и props, state может быть иерархичным и принимать любую форму:

Скопировать кодJSX

state = {

theme: "светлая",

posts: [

{ id: 1, title: "Новость 1" },

{ id: 2, title: "Новость 2" }

],

commentsEnabled: true,

user: {

name: "Гекльберри Финн",

uuid: "123e4567-e89b-12d3-a456-426655440000",

lastActive: 1614498769824

}

};

В этом примере в state хранится информация о теме, применённой в приложении, и текущем пользователе — его имя, идентификатор, время последней активности, настройка доступности комментариев, список новостей. Объект state можно использовать для описания любой «модели данных», на которую хватит фантазии.

Мы подготовили несколько практических задач, которые помогут вам увереннее использовать внутреннее состояние React-компонентов. В следующем уроке расскажем, как на это состояние влиять, и рассмотрим, как state и props работают вместе.

# Как изменять состояние в Reaсt-компонентах

Вы уже знаете, что внутреннее состояние позволяет влиять на логику работы и рендеринг элементов. Но как влиять на эти состояния из самого компонента, скажем, при нажатии на кнопку или движении курсора? Разберём в этом уроке.

У state есть особенность — он иммутабельный. Это означает, что изменить его напрямую (как если бы это была переменная) не получится и перерисовки DOM не произойдёт.

Изменять значение состояния можно методом setState в React.Component. Когда мы вызываем этот метод, React «под капотом» сравнивает состояние компонента до и после изменений. Если React обнаружит различия, ReactDOM перерисует элемент, который обновился. Рассмотрим это на примере:

Скопировать кодJSX

class ThemeToggle extends React.Component {

state = { theme: "светлая" };

toggleTheme = () => {

this.setState({

theme: this.state.theme === "светлая" ? "тёмная" : "светлая"

});

};

render() {

return (

<div className={this.state.theme === "светлая" ? "th-light" : "th-dark"}>

<p>Включена {this.state.theme} тема</p>

<button onClick={this.toggleTheme}>

{this.state.theme === "светлая" ? "🌚" : "🌞"}

</button>

</div>

);

}

}

Нажатие на кнопку вызывает метод toggleTheme. Внутри него вызывается setState, где мы проверяем, какая тема включена, и меняем её на противоположную.

## Особенности метода setState

У метода setState есть несколько особенностей, которые важно знать:

1. В качестве аргумента this.setState принимает либо функцию, либо обновлённое состояние:

Скопировать кодJSX

this.setState(function(prevState, props){

return { showForm: !prevState.showForm }

});

*// либо*

this.setState({ showForm: !this.state.showForm });

Если использовать функцию, внутри будут доступны состояния компонента до обновления и текущие свойства компонента — props. В некоторых сложных компонентах пропсы могут быть использованы для реализации вариативной логики применения изменений. Но на практике вы редко будете сталкиваться с таким применением метода setState.

1. У setState есть второй аргумент, но он необязательный. В качестве второго аргумента можно передать колбэк-функцию. Эта функция будет вызвана, когда изменения применятся к состоянию компонента. Второй аргумент метода setState следует использовать, когда нужно реализовать логирование или сложную последовательную логику изменения состояний в компонентах:

Скопировать кодJSX

this.setState(

{ showForm: !this.state.showForm },

() => {

this.myCustomLogger("Состояние компонента изменилось!")

}

);

Но будьте осторожны: не допускайте бесконечного цикла обновлений и колбэков — это приведёт к зависанию браузера.

1. Вызов setState не применяет изменения моментально. Это асинхронный метод. Такое поведение называется batch update: оно позволяет накопить грядущие изменения, а потом разом их применить. Этот механизм помогает улучшить производительность — сократить количество перерисовок контента в браузере.

## Использование state вместе c props

Представьте, что вам нужно создать компонент с заголовком и текстом, которые можно задавать через пропсы. Текст по умолчанию скрыт и должен раскрываться/скрываться по нажатию на заголовок. Спроектируем такой компонент на React с использованием state и props:

Скопировать кодJSX

class CollapsableTextBlock extends React.Component {

state = { toggled: false };

onHeadingClick = () => {

this.setState({toggled: !this.state.toggled})

}

render() {

return (

<div className="CollapsableTextBlock">

<h1 onClick={this.onHeadingClick}>

{this.props.header}

</h1>

{this.state.toggled && <p>{this.props.children}</p>}

</div>

);

}

}

В этом примере мы использовали state компонента для реализации функциональности скрытия и раскрытия текста, а props — для наполнения компонента текстом. Код этого компонента выглядит так:

Скопировать кодJSX

<CollapsableTextBlock header="Заголовок">

Этот текст будет скрыт до нажатия на заголовок

</CollapsableTextBlock>

Внутреннее состояние компонента может использоваться как пропс для дочернего компонента. Слегка изменим наш компонент:

Скопировать кодJSX

function CollapsableTextContent(props) {

if (!props.toggled) {

return null;

}

return <p>{props.children}</p>

}

class CollapsableTextBlock extends React.Component {

state = { toggled: false };

onHeadingClick = () => {

this.setState({toggled: !this.state.toggled})

}

render() {

return (

<div className="CollapsableTextBlock">

<h1 onClick={this.onHeadingClick}>

{this.props.header}

</h1>

<CollapsableTextContent toggled={this.state.toggled}>

{this.props.children}

</CollapsableTextContent>

</div>

);

}

}

В этом примере мы выделили разметку текста и логику её появления в самостоятельный компонент CollapsableTextContent. Этот компонент принимает от родительского CollapsableTextBlock пропcы children и toggled.

Проброс пропсов через несколько уровней компонентов называется props drilling. Однако увлекаться таким пробросом пропсов через иерархию компонентов не стоит. При таком подходе появляется большая связанность компонентов, и от этого падает качество кода и скорость разработки. Если пропс проходит три и больше уровней компонентов — стоит задуматься о декомпозиции состояния и разметки. Мы ещё вернёмся к этой проблеме и способам её решения.

Комбинируя state и props, можно создавать гибкие и удобные компоненты.

## Использование setState для иерархического состояния

Иногда нужно изменить вложенное свойство или элемент массива, не трогая остальные. Для таких случаев удобно применять spread оператор (...):

Скопировать кодJSX

class NewsPanel extends React.Component {

*// Состояние компонента*

state = {

theme: "светлая",

posts: [

{ id: 1, title: "Новость 1" },

{ id: 2, title: "Новость 2" }

],

commentsEnabled: true,

user: {

name: "Гекльберри Финн",

uuid: "123e4567-e89b-12d3-a456-426655440000",

lastActive: 1614498769824

}

};

*// Метод, который отвечает за добавление*

addNewPost = () => {

*// Получаем количество новостей в state на данный момент*

const postCount = this.state.posts.length;

*// Вычисляем номер добавляемой новости на основании текущего количества*

const newPostNumber += postCount;

*// Вызываем метод setState для обновления состояния*

this.setState(prevState => ({

*// Сохраняем текущие значения, над которыми не производим действий*

...prevState,

posts: [

*// Сохраняем текущий список новостей*

...prevState.posts,

*// Добавляем в список новостей новую запись с вычисленным номером*

{ id: newPostNumber, title: `Новость ${newPostNumber}` }

]

}));

};

render() {

return (

{*/\* При нажитии на кнопку будет вызван метод addNewPost \*/*}

<button onClick={this.addNewPost}>Добавить новость</button>

);

}

}

При нажатии на кнопку в массив posts будет добавлен ещё один элемент. Существующие элементы не будут изменены, как и остальные ключи состояния. Не стоит чрезмерно усложнять состояние, старайтесь чаще применять декомпозицию. Это сделает разработку удобной, а приложение производительным.

В этом уроке вы узнали, как изменять состояние компонентов и использовать его вместе с props. Умение работать с состоянием — важный навык для React-разработчика. Для закрепления мы подготовили несколько практических задач.

class Door extends React.Component {

state = { isOpen: false };

handleClickOpen = () => {

this.setState({ isOpen: !this.state.isOpen })

}

render() {

return (

<div className={this.state.isOpen ? 'content bg\_open' : 'content bg\_close'}>

<h1 className="content\_\_title" >{this.state.isOpen ? `Замок открыт 🔓` : `Замок закрыт 🔒`}</h1>

<button className="content\_\_btn" onClick={this.handleClickOpen}>{this.state.isOpen ? 'Закрыть' : 'Открыть'}</button>

</div>

);

}

}

ReactDOM.render(<Door />, document.querySelector('#root'));

В этой задаче нужно точечно изменить значение state, как бы глубоко оно ни лежало. Допишите функцию toggleSidebar так, чтобы её вызов переключал enabled в ключе state.userSettings.sidebar с true на false и наоборот. Важно сохранить прочие значения в state неизменными.

class SettingsMenu extends React.Component {

state = {

theme: 'dark',

userSettings: {

notificationsEnabled: true,

sidebar: {

title: 'Боковая панель',

enabled: false

}

}

};

toggleSidebar = () => {

this.setState(prevState => ({

...prevState,

userSettings: {

...prevState.userSettings,

sidebar: {

...prevState.userSettings.sidebar,

enabled: !this.state.userSettings.sidebar.enabled

}

}

}));

};

render() {

const { enabled } = this.state.userSettings.sidebar;

return (

<>

{enabled && <aside className="aside" />}

<section className="content">

<button className="content\_\_button" onClick={this.toggleSidebar}>

{enabled ? 'Выключить' : 'Включить'}

</button>

</section>

</>

);

}

}

ReactDOM.render(<SettingsMenu />, document.querySelector('#root'));

}

Перед вами упрощённый вариант корпоративного календаря. Вас позвали на встречу, и нужно подтвердить ваше присутствие.

В state компонента есть поле invited и сurrentUser. Среди invited находится пользователь, который авторизован в системе, и его id соответствует currentUser. Применять изменения нужно к этому пользователю.

Допишите три функции: getCurrentUserConfirmationStatus, confirm и cancel.

Функция confirm должна изменять состояние status у текущего пользователя на "confirmed", а cancel — на "canceled". Функция getCurrentUserConfirmationStatus должна возвращать статус текущего пользователя. Использовать эту функцию необходимо в методах confirm и cancel .

Решение от автора:

getCurrentUserConfirmationStatus() {

return this.state.invited.find(user => user.id === this.state.currentUser).status;

}

confirm = () => {

this.getCurrentUserConfirmationStatus !== STATUS\_CONFIRMED &&

this.setState(prevState => ({

...prevState,

invited: this.state.invited.map(user => {

if (user.id === this.state.currentUser) {

user.status = STATUS\_CONFIRMED;

return user;

}

return user;

})

}));

};

cancel = () => {

this.getCurrentUserConfirmationStatus !== STATUS\_CANCELED &&

this.setState(prevState => ({

...prevState,

invited: this.state.invited.map(user => {

if (user.id === this.state.currentUser) {

user.status = STATUS\_CANCELED;

return user;

}

return user;

})

}));

};

Мое решение:

const STATUS\_CONFIRMED = 'confirmed';

const STATUS\_PENDING = 'pending';

const STATUS\_CANCELED = 'canceled';

const renderStatus = {

confirmed: 'Подтверждён',

pending: 'Не подтверждён',

canceled: 'Отменён'

};

class User extends React.Component {

render() {

return (

<div className="user">

<img className="user\_\_avatar" src={this.props.avatar} alt="фото." />

<div className="user\_\_info">

<p className="user\_\_text">{`${this.props.name}, ${this.props.role}`}</p>

{this.props.status && (

<p className={`user\_\_status ${this.props.status}`}>{renderStatus[this.props.status]}</p>

)}

</div>

</div>

);

}

}

class CalendarEvent extends React.Component {

state = {

currentUser: 34047044,

owner: { id: 34049221, name: 'Павел', role: 'Технический директор', avatar: './images/1.png' },

subject: 'Обсуждение редизайна административной панели сайта',

invited: [

{

id: 34049119,

name: 'Татьяна',

role: 'Дизайнер',

status: STATUS\_CONFIRMED,

avatar: './images/2.png'

},

{

id: 34047044,

name: 'Кирилл',

role: 'Разработчик',

status: STATUS\_PENDING,

avatar: './images/3.png'

},

{

id: 34048196,

name: 'Константин',

role: 'Менеджер',

status: STATUS\_CANCELED,

avatar: './images/4.png'

}

],

durationdate: '10.11.2021',

timeStart: '14:30',

duration: 40,

location: 'Переговорная №4'

};

getCurrentUserConfirmationStatus() {

const currentUserId = this.state.currentUser;

const currentUserInfo = this.state.invited.filter(i => i.id === currentUserId)

return currentUserInfo[0].status

}

confirm = () => {

const currentUserId = this.state.currentUser;

const currentUserFind = this.state.invited.find(i => i.id === currentUserId)

currentUserFind.status = STATUS\_CONFIRMED

this.setState(prevState => ({

...prevState,

invited: [

...prevState.invited,

currentUserFind

]

}

))

}

cancel = () => {

const currentUserId = this.state.currentUser;

const currentUserFind = this.state.invited.find(i => i.id === currentUserId)

currentUserFind.status = STATUS\_CANCELED

this.setState(prevState => ({

...prevState,

invited: [

...prevState.invited,

currentUserFind

]

}

))

}

render() {

const confirmed = this.getCurrentUserConfirmationStatus() === STATUS\_CONFIRMED;

return (

<section className="main">

<div className="calendar">

<p className="calendar\_\_menu">Тема:</p>

<h1 className="calendar\_\_title">{this.state.subject}</h1>

<p className="calendar\_\_menu">Организатор:</p>

<User {...this.state.owner} />

<p className="calendar\_\_menu">Приглашены:</p>

<div className="calendar\_\_invited">

{this.state.invited.map((user, index) => (

<User key={index} {...user} />

))}

</div>

<p className="calendar\_\_menu">Дата:</p>

<p className="calendar\_\_text">{this.state.durationdate}</p>

<p className="calendar\_\_menu">Начало:</p>

<p className="calendar\_\_text">{this.state.timeStart}</p>

<p className="calendar\_\_menu">Продолжительность:</p>

<p className="calendar\_\_text">{this.state.duration}</p>

<p className="calendar\_\_menu">Место:</p>

<p className="calendar\_\_text">{this.state.location}</p>

</div>

<div className="buttons">

<button className="button cancel" type="button" onClick={this.cancel}>

Отменить

</button>

<button className="button confirm" type="button" onClick={this.confirm}>

Подтвердить

</button>

</div>

</section>

);

}

}

ReactDOM.render(<CalendarEvent />, document.querySelector('#root'));

# Списки в React. Свойство key

Если вы читаете это, значит, уже проделали колоссальный путь. Вы уже многое знаете о компонентах и пропсах в React, но кое-что осталось за скобками. [В предыдущих уроках](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/c401de6c-42d0-4b87-aef4-1806624f11c2/) мы показывали, как массивы хранятся во внутреннем состоянии или передаются как пропсы, но не объяснили, как рендерить элементы массива в JSX. Чтобы это понять, нужно изучить списки.

Список — набор однородных и соседствующих элементов. То есть это любые однотипные данные, которые повторяются более одного раза: меню сайта, список пользователей, галерея изображений.

Допустим, мы работаем над компонентом для чата и у нас есть такие данные:

Скопировать кодJSX

[{

id: 1,

user: 'Ольга',

text: 'Привет! Можешь помочь со списками в React?',

}, {

id: 2,

user: 'Николай',

text: 'Здравствуй! Конечно, это проще простого! Какой у тебя вопрос?',

}, {

id: 3,

user: 'Ольга',

text: 'Спасибо! Как они попадают в JSX?',

}]

Каждый элемент массива мы хотим отображать как сообщение в чате в виде такой разметки:

Скопировать кодJSX

<div className="Message">

<span className="Message-user">{message.user}</span>

<span className="Message-text">{message.text}</span>

</div>

В JSX можно использовать метод массивов map. Применим его и создадим компонент, который будет отображать список сообщений так, как нужно:

Скопировать кодJSX

class ChatRoom extends React.Component {

state = {

messages: [{

id: 1,

user: 'Ольга',

text: 'Привет! Можешь помочь со списками в React?',

}, {

id: 2,

user: 'Николай',

text: 'Здравствуй! Конечно, это проще простого! Какой у тебя вопрос?',

}, {

id: 3,

user: 'Ольга',

text: 'Спасибо! Как они попадают в JSX?',

}];

}

render(){

return (

<div className="ChatRoom">

{this.state.messages.map((message, index)=>(

<div className="Message">

<span className="Message-user">{message.user}</span>

<span className="Message-text">{message.text}</span>

</div>

))}

</div>

);

}

}

Вот и всё! Код будет работать, но в консоли отобразится предупреждение о необходимости использовать пропс key для каждого элемента списка. Разберём, что это за пропс и как им пользоваться.

## Свойство key

Как вы знаете, при изменении props, state, компонентов и их потомков React рекурсивно сравнивает дерево компонентов «До» изменений с деревом компонентов «После» них. Там, где находятся несоответствия, React вызывает перерисовку всех потомков обновившегося компонента. По логике, при изменении содержимого списка React должен перерисовать его весь, даже если изменения коснулись только одного элемента.

В примере с чатом пользователь мог отредактировать сообщение после его отправки. Это действие повлияет только на одно сообщение из чата, но может привести к ухудшению производительности или даже полному зависанию вкладки браузера, ведь React запустит повторный рендер для всего чата. К счастью, есть способ это предотвратить: все элементы в React поддерживают пропс key. Указывать это свойство нужно у первого элемента разметки внутри цикла. Если задать это свойство элементу, то при изменении его соседей в списке сам элемент не перерисуется:

Скопировать кодJSX

...

render(){

return (

<div className="ChatRoom">

{this.state.messages.map((message, index)=>(

{/\* Добавили проп key \*/}

<div className="Message" key={message.id}>

<span className="Message-user">{message.user}</span>

<span className="Message-text">{message.text}</span>

</div>

))}

</div>

);

}

...

Теперь, если пользователи напишут новое сообщение или редактируют старое, остальные сообщения в списке не обновятся. Приложение не будет тормозить, а пользователи — нервничать.

Свойство key всегда обязательно применять для рендеринга списков. Даже фрагменты поддерживают это свойство (и оно единственное у фрагментов):

Скопировать кодJSX

...

{this.state.messages.map((message, index)=>(

<React.Fragment key={message.id}>

<img className="Avatar" src={message.user.avatar} alt="avatar" />

<div className="Message">

<span className="Message-user">{message.user}</span>

<span className="Message-text">{message.text}</span>

</div>

</React.Fragment>

))}

...

В этом примере мы добавили в разметку аватар отправителя, но обернули всё с помощью React.Fragment.

Свойство key должно быть уникальным для каждого элемента в списке. Повторение ключей элементов списка может привести к непредсказуемой работе и ошибкам рендеринга. Ключи могут повторяться только в том случае, если используются двумя разными списками.

Если у элементов списка отсутствуют уникальные идентификаторы, например id, можно воспользоваться вторым аргументом из метода map — индексом элемента массива. Индекс уникален для каждого из элементов, расположенных в одном массиве. Это решение подходит только в том случае, если вы уверены, что порядок элементов не поменяется.

Скопировать кодJSX

...

render(){

return (

<div className="ChatRoom">

{this.state.messages.map((message, index)=>(

{/\* Используется индекс элемента в качестве key \*/}

<div className="Message" key={index}>

<span className="Message-user">{message.user}</span>

<span className="Message-text">{message.text}</span>

</div>

))}

</div>

);

}

...

В этом примере мы использовали второй аргумент метода map для свойства key. Такой приём часто будет встречаться в реальных приложениях. В этом курсе вам предстоит ещё не раз поработать с массивами, поэтому вас ждёт пара задач для закрепления навыков.

В этом задании займёмся разработкой чата для соседей по подъезду. Компоненты RepliedMessage и Message уже готовы, осталось только дописать компонент Chat. Он должен рендерить сообщения из props.thread с помощью метода массива map.

Некоторые сообщения могут быть ответом на другое сообщение. В этом случае присутствует поле replyTo, которое содержит id сообщения, на которое был ответ. Чтобы реализовать эту функциональность, нужно в компонент Message передать пропс repliedMessage, он должен содержать сообщение, на которое был ответ. Не забудьте, что каждый новый элемент списка должен иметь пропс key для корректной работы.

);

import React from 'react';

import './styles.css';

const Message = ({ message, repliedMessage, className = 'message' }) => (

<div className={className}>

{repliedMessage && <RepliedMessage message={repliedMessage} />}

<h3>{message.user}</h3>

<p>{message.text}</p>

</div>

);

const RepliedMessage = ({ message }) => <Message message={message} className={'replied-message'} />;

const Chat = ({ thread }) => (

<div className="tread">

{thread.map((message) => (

message.replyTo ?

<Message key={message.id} message={message} repliedMessage={thread.find((m) => m.id === message.replyTo)}/>

:

<Message message={message} key={message.id}/>

)

)}

</div>

);

export default class App extends React.Component {

state = {

thread: [

{

id: 1,

user: 'Тамара',

text: 'Всем привет! Кто в курсе, когда в нашем доме отключат горячую воду?'

},

{

id: 2,

user: 'Алексей',

replyTo: 1,

text: 'В подъезде висит объявление, скоро буду там, сфотографирую и пришлю сюда'

},

{

id: 3,

user: 'Катя',

replyTo: 2,

text: 'О! Спасибо! Ждём! :)'

}

]

};

render() {

return (

<div className="App">

<Chat thread={this.state.thread} />

</div>

);

}

}

# Классовые компоненты в деталях

В предыдущем уроке мы разобрали рендеринг списков в React и пропс key у элементов в JSX. В этом уроке подробнее рассмотрим классовые компоненты, а также изучим дополнительные возможности React.Component.

В этом курсе мы будем использовать React версии 17 и выше. Эта версия обладает обширным инструментарием, но так было не всегда. До версии React 16.8 компоненты делили на:

* Компоненты с внутренним состоянием. Их ещё называют stateful, или «умные», компоненты. Их описывают с помощью классов.
* Компоненты без внутреннего состояния — stateless, или «глупые», компоненты. Этот вид описывают с помощью функций.

Начиная с версии 16.8 такое разделение потеряло значимость: у «глупых» компонентов, как и у «умных», появилась возможность иметь внутреннее состояние и реализовывать в себе сложную логику работы. Но в реальном мире на классах написано много кода, который тоже нужно поддерживать и развивать. Для начала научимся создавать и инициализировать компоненты — изучим конструктор.

## Конструктор компонента: constructor(props)

Конструктор — специальный метод классов, который служит для создания и инициализации объектов. В React такими объектами являются компоненты. Рассмотрим пример использования конструктора в React-компоненте:

Скопировать кодJSX

class SayHi extends React.Component {

constructor(props) {

super(props);

this.state = { theme: 'dark' }

this.showGreeting = this.showGreeting.bind(this);

}

showGreeting() {

alert(this.props.name);

}

render() {

const className = this.state.theme === 'dark' ? 'Btn-dark' : 'Btn-light';

return (

<button

className={className}

onClick={this.showGreeting}>Нажми на меня

</button>

)

}

}

Разберём этот пример подробнее:

Первым делом при использовании конструктора нужно вызывать super(props). Если этого не сделать, в коде могут появиться баги, ведь this.props не будет определён. В этом примере компонент SayHi в своём конструкторе с помощью ключевого слова super вызывает конструктор родительского класса — React.Component.

Далее в конструкторе следует присвоить значение this.state. Конструктор — единственное место, где можно явно указать значение this.state. В других частях компонента потребуется уже известный вам метод this.setState.

Ну и напоследок внутри конструктора компонента SayHi нужно привязать обработчик события — this.showGreeting. Чтобы не использовать конструкцию bind(this), вы можете описывать методы в виде стрелочных функций — тогда контекст выполнения будет автоматически взят из класса.

Чего не стоит делать в конструкторе:

1. Подписываться на события и обрабатывать сайд-эффекты:

Скопировать кодJSX

constructor(props) {

super(props);

*// Не делайте так*

document.addEventListener('scroll', () => сonsole.log(window.pageYOffset));

}

О том, как правильно это делать, расскажем в следующем уроке.

1. Назначать this.state из props. Компонент перестанет реагировать на изменения props и не будет рендерить обновлённую разметку:

Скопировать кодJSX

constructor(props) {

super(props);

*// Не делайте так*

this.state = { name: props.name }

}

Используйте эту возможность, только если хотите отказаться от обновлений компонента на основании его пропсов. Если props требуется вам в качестве внутреннего состояния компонента, воспользуйтесь методом getDerivedStateFromProps. Ознакомиться с ним можно [по ссылке](https://ru.reactjs.org/docs/react-component.html#static-getderivedstatefromprops).

Вы можете не использовать конструктор в React-компоненте, если не привязываете методы. Помните об этом, ведь в продакшене очень многое написано на классах.

## Пропсы по умолчанию: defaultProps

В случае, когда у компонента должны быть пропсы по умолчанию, можно воспользоваться свойством defaultProps. Значения из этого свойства будут браться для неопределённых (undefined) пропсов, но не для пропсов со значением null. Например:

Скопировать кодJSX

class ImageContainer extends React.Component {

// ...

}

ImageContainer.defaultProps = {

background: 'black'

};

В случае, когда props.background не передаётся, по умолчанию будет установлено значение из defaultProps, то есть 'black':

Скопировать кодJSX

render() {

return <ImageContainer /> ; *// props.background = 'black'*

}

Если props.background имеет значение null, оно останется null:

Скопировать кодJSX

render() {

return <ImageContainer background={null} /> ; *// props.background = null*

}

## Ручное управление рендерингом: forceUpdate()

По умолчанию при изменении состояния компонента или пропса происходит повторный рендер. Если отрисовка компонента зависит от других данных, вы можете вызвать повторный рендер методом forceUpdate:

Скопировать кодJSX

class ChartContainer extends React.Component {

*// ...*

const onMyCustomEvent = () => {

this.forceUpdate(callbackFn)

}

*// ...*

}

В вызов метода forceUpdate можно передать колбэк-функцию.

В следующем уроке изучим жизненные циклы компонента и разберёмся, как получать данные с сервера в React-компонентах.

# Жизненный цикл компонентов

В предыдущем уроке мы рассказывали о дополнительных свойствах и методах, которые доступны в классовых компонентах. Но это не всё — у классовых компонентов есть несколько методов, которые позволяют более «хирургически» контролировать логику работы и рендеринг.

В этом уроке вы научитесь контролировать метод render() компонента и вызывать его только при наступлении определённых условий. Кроме этого, вы получите навык работы с обработчиками событий в React и узнаете, как не допустить утечек памяти.

## Что такое жизненный цикл

У каждого классового компонента есть жизненный цикл. Чтобы понять, что это такое, представьте искусственную новогоднюю ёлку: в декабре её достают с антресоли, собирают, наряжают игрушками, а после праздника (в марте) разбирают до следующего Нового года. Похожим образом ведут себя и компоненты. Каждый компонент в течение своего жизненного цикла может быть монтирован (англ. mount), обновлён (англ. update) и размонтирован (англ. unmount). Библиотека React позволяет выполнять код на конкретном этапе этого процесса.

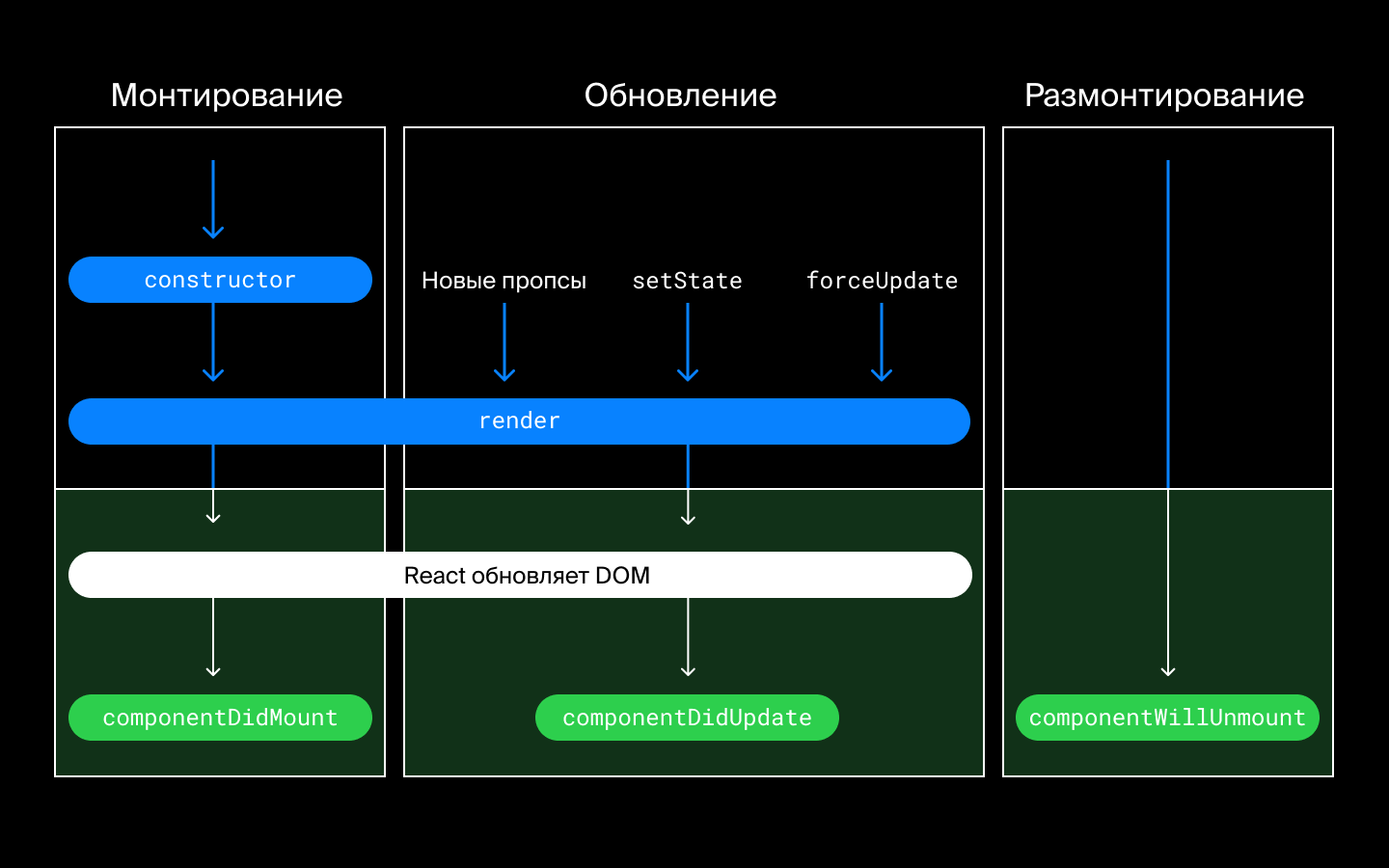
**Монтирование** происходит, когда HTML-дерево компонента вставляется движком React в DOM.

**Обновление** осуществляется в одном из трёх случаев:

* если произошёл рендер родительского компонента,
* если изменилось внутреннее состояние в результате вызова this.setState,
* если обновление инициировано вызовом встроенного метода this.forceUpdate.

**Размонтирование** происходит, когда HTML-дерево компонента удаляется из DOM.

Каждому из этих событий соответствует один из трёх методов жизненного цикла: componentDidMount, componentDidUpdate и componentWillUnmount:



Можно ознакомиться [*с интерактивным вариантом схемы по ссылке*](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/)

Как и render, это методы обратного вызова. В отличие от setState их не нужно вызывать вручную, а достаточно описать внутри компонентов, чтобы в нужный момент они были вызваны изнутри движка React — как колбэки.

Чаще всего методы жизненного цикла применяют для описания сайд-эффектов (англ. side effects). Этим понятием принято называть всю деятельность компонента, связанную с «внешним миром»: взаимодействие с различными API, подписка на события, обращение к серверу за данными и т. д.

## Монтирование: componentDidMount

Этот метод вызывается сразу после монтирования. Внутри componentDidMount можно подписываться на события:

Скопировать кодJSX

class VirtualList extends React.Component {

state = { offsetY: 0 }

componentDidMount() {

document.addEventListener(

'scroll',

()=>this.setState({ offsetY: window.pageYOffset })

)

}

*// ...*

}

В этом примере компонент при монтировании добавляет на документ обработчик скролла и начинает записывать значение pageYOffset в своё внутреннее состояние.

Кроме этого, метод жизненного цикла componentDidMount отлично подходит для взаимодействия с сервером, например для запроса данных:

Скопировать кодJSX

class Stories extends React.Component {

state = {

stories: [],

loading: true,

hasError: false

}

componentDidMount() {

fetch('/stories/all')

.then(res => res.json())

.then(data => this.setState({ stories: data.stories, loading: false }))

.catch(e => this.setState({ ...this.state, loading: false, hasError: true }))

}

*// ...*

}

В этом примере при монтировании компонента будет выполнен запрос на сервер. Результат запроса будет записан во внутреннее состояние компонента. Из состояния он впоследствии сможет быть использован для создания JSX разметки. Обратите внимание, что кроме самих данных, в состоянии содержатся ключи hasError и loading. Используя эти ключи, мы сможем показывать пользователям разную разметку с помощью [условного рендеринга](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/4a263618-0f56-4a01-9050-1d18d6f19403/): компонент индикации загрузки или ошибку в случае, если ответ с сервера не пришёл в ожидаемом виде.

## Обновление: componentDidUpdate

Метод componentDidUpdate() вызывается сразу после обновления внутреннего состояния или пропсов. Но его нельзя вызвать при монтировании.

В этом методе можно работать с реальным DOM-деревом при обновлении компонента. Также он подходит для взаимодействия с сервером. Если сетевой запрос зависит от пропсов компонента, например от идентификатора товара, то в componentDidUpdate можно сравнить предыдущие пропсы (prevProps) и обновившиеся (props). Если идентификаторы различаются, нужно запросить новые данные. Если же пропсы не изменились, новый запрос может и не потребоваться. Взгляните на такой пример:

Скопировать кодJSX

class Product extends React.Component {

state = {

productData: null,

loading: true,

hasError: false

}

getProductData = () => {

fetch(`/api/v1/products/${this.props.productId}`)

.then(res => res.json())

.then(data => this.setState({ productData: data.productData, loading: false }))

.catch(e => this.setState({ ...this.state, loading: false, hasError: true }))

}

componentDidMount() {

this.getProductData()

}

componentDidUpdate(prevProps, prevState) {

*// Сравниваем предыдущие пропсы с обновившимися.*

*// Если они отличаются, то делаем запрос:*

if (this.props.productId !== prevProps.productId) {

this.getProductData();

}

}

*// ...*

}

Кроме предыдущих пропсов, в замыкании метода доступно предыдущее состояние компонента (prevState). Сравнение состояний может быть полезным для реализации вариативной логики работы компонента.

В методе componentDidUpdate() можно воспользоваться методом setState(), но важно не забыть обернуть его в условие, чтобы не возник бесконечный цикл:

Скопировать кодJSX

*// ...*

componentDidUpdate(prevProps, prevState) {

*// Сравниваем предыдущее состояние с новыми*

*// Если нужные нам ключи отличаются, то делаем запрос:*

if (this.props.productId !== prevProps.productId) {

this.setState({ ...this.state, loading: true });

this.getProductData();

}

}

*// ...*

Вызов setState() влечёт за собой дополнительный рендер, который незаметен для пользователя, но может повлиять на производительность компонента.

## Проверка на необходимость повторного рендера: shouldComponentUpdate

Метод shouldComponentUpdate позволяет повысить производительность кода компонента. Он сравнивает текущие значения props и/или state и решает, нужен ли повторный рендеринг. Если метод возвращает true, то при изменении props или state произойдет повторный рендеринг. Если же он вернет false, то, несмотря на изменения state и props, повторного рендеринга не произойдёт. Метод shouldComponentUpdate позволяет оптимизировать производительность в тех местах приложения, где рендеринг происходит слишком часто.

Чтобы разобраться с этим методом, посмотрим на его аргументы и пример использования. В замыкании shouldComponentUpdate доступны два аргумента: nextProps и nextState — обновлённые объекты пропсов и состояния соотвественно. Вот, как выглядит применение этого метода:

Скопировать кодJSX

class TodoItem extends React.Component {

constructor(props) {

super(props);

this.state = { done: false };

}

shouldComponentUpdate(nextProps, nextState) {

if (this.props.tagColor !== nextProps.tagColor) {

*// Если изменилось значение props.tagColor, то будет вызван повторный рендер*

return true;

}

if (this.state.done !== nextState.done) {

*// Если изменилось значение state.done, то будет вызван повторный рендер*

return true;

}

*// Во всех остальных случаях повторного рендеринга не будет*

return false;

}

toggleTodo = () => {

this.setState({ done: !this.state.done });

};

render() {

const btnText = this.state.done ? "Вернуть в работу" : "Выполнено";

return (

<>

<TodoTag tagColor={this.props.tagColor} />

<button onClick={this.toggleTodo}>{btnText}</button>

</>

);

}

}

В этом примере компонент TodoItem будет повторно рендериться только если изменились значения props.tagColor или state.done. Компонент не будет повторно перерисован, даже если в него переданы другие пропсы, которые изменились. shouldComponentUpdate особенно эффективен в сложных компонентах, где рендеринг ощутимо бьет по производительности.

Помимо этого метода жизненного цикла, в библиотеке React существуют дополнительные возможности для оптимизации производительности, к ним мы ещё вернёмся в рамках курса.

## Размонтирование: componentWillUnmount

В предыдущих методах мы могли подписываться на событие или устанавливать соединение с сервером. Теперь расскажем, как отписываться от события, закрывать соединения и не допускать утечки памяти.

Для этого существует метод componentWillUnmount. Его нужно вызывать перед удалением компонента из DOM. Чтобы избежать утечек памяти, при размонтировании компонентов следует «прибраться»: удалить таймеры, отписаться от событий, закрыть соединения с сервером. Ведь если этого не сделать, то, когда произойдёт размонтирование компонента, JavaScript продолжит свою работу и память засорится. В этом примере мы подписываемся на событие прокрутки в методе componentDidMount, а отписываемся от этого события в методе componentWillUnmount:

Скопировать кодJSX

class VirtualList extends React.Component {

state = { offsetY: 0 };

*// Создадим отдельную функцию для того чтобы слушатель событий можно было удалить*

setOffset = () => {

this.setState({ offsetY: window.pageYOffset });

};

componentDidMount() {

document.addEventListener("scroll", this.setOffset);

}

componentWillUnmount() {

*// Не забывайте отписываться от событий, чтобы не допустить утечек памяти*

document.removeEventListener("scroll", this.setOffset);

}

*// ...*

}

Удаление таймеров, очистка обработчиков событий, прерывание сетевых запросов и web-socket соединений — всё это следует делать именно в методе componentWillUnmount.

Но не стоит вызывать setState в этом методе. Изменения не применятся: компонент будет отмонтирован и отрисуется повторно. Когда компонент снова появится, его жизненный цикл начнётся сначала.

Мы написали два компонента: App и MouseTrack. Компонент App позволяет включить или выключить отображение компонента MouseTrack. У компонента MouseTrack есть метод trackMousePos, который записывает актуальное положение курсора на странице в state.mousePosition.

Сделайте так, чтобы при монтировании компонента этот метод задавался в качестве обработчика события mousemove объекта document.

class App extends React.Component {

state = { trackMouse: false };

render() {

const btnClassName = this.state.trackMouse ? 'btn-on' : 'btn-off';

return (

<div className={'container d-flex gap20'}>

<div>

<h3 className="app\_\_title">Отслеживать движение мыши</h3>

<button

className={`btn ${btnClassName}`}

onClick={() => this.setState({ trackMouse: !this.state.trackMouse })}

>

{this.state.trackMouse ? 'Остановить' : 'Начать'}

</button>

</div>

<MouseTrack trackMouse={this.state.trackMouse} />

</div>

);

}

}

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

class MouseTrack extends React.Component {

state = {

mousePosition: []

};

trackMousePos = e => {

this.setState({

mousePosition: [e.clientX, e.clientY]

});

};

componentDidMount() {

document.addEventListener("mousemove", this.trackMousePos);

}

componentWillUnmount() {

document.removeEventListener("mousemove", this.trackMousePos);

}

render() {

return (

<div>

<h3 className="app\_\_title">Позиция курсора:</h3>

<h2>

X: {this.props.trackMouse ? this.state.mousePosition[0] : ' -'}

<br />

Y: {this.props.trackMouse ? this.state.mousePosition[1] : ' -'}

</h2>

</div>

);

}

}

# Подъём состояния

В предыдущем уроке мы рассказывали о жизненном цикле компонентов. Теперь разберёмся, как в React поднимаются данные от дочерних компонентов к родительским.

Часто бывает, что несколько компонентов отображают одни и те же изменяющиеся данные. Некоторые компоненты могут влиять на эти данные, а остальные должны моментально реагировать на изменения, перерисовывая разметку. Так во всех частях интерфейса всегда отображаются актуальные данные.

Разберёмся с подъёмом состояния в React на примере: напишем корзину для интернет-магазина.

Состояние корзины выглядит так:

Скопировать кодJSX

state = {

cart: [

{

id: 1,

name: "Скафандр",

count: 1,

price: 19345

},

{

id: 2,

name: "Сельдереевый смузи",

count: 3,

price: 46

},

{

id: 3,

name: "Запасной парашют",

count: 1,

price: 1870

}

]

}

Упростим задачу и сосредоточимся на работе с данными. Наша корзина состоит из трёх компонентов:

1. Главный компонент корзины. Он хранит состояние о товарах и передаёт его дочерним компонентам.
2. Карточка товара. Этот компонент содержит информацию о товаре: его название, количество и цену. Из карточки товара мы сможем влиять на количество товара, нажимая на плюс или минус.
3. Компонент, который отображает итоговую стоимость покупок и позволяет очистить корзину целиком, если нужно. В этом компоненте мы будем вычислять налог на добавленную стоимость (НДС) для товаров в корзине.

Начнём с конца списка и напишем функциональный компонент с итоговой стоимостью и вычислением НДС:

Скопировать кодJSX

*// Мы воспользовались возможностями деструктуризации объекта пропсов*

*// для удобства и краткости и сразу извлекли total из props:*

const CartTotal = ({ total }) => {

return (

<div>

<h1>{total}</h1>

<p>В том числе НДС: {total \* 0.2}</p>

<button>Очистить корзину</button>

</div>

);

};

Компонент CartTotal получает итоговую стоимость в виде пропса — total.

Теперь пойдём вверх по списку компонентов и сделаем карточку товара:

Скопировать кодJSX

const Product = ({ product }) => {

return (

<div>

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

<p>{product.price}</p>

<button>-</button>

<span>{product.count}</span>

<button>+</button>

</div>

);

};

Компонент Product получает из пропсов данные о товаре, которые затем превращает в разметку.

Закончим со списком компонентов и напишем родительский компонент — корзину:

Скопировать кодJSX

export default class Cart extends React.Component {

state = {

cart: [

{

id: 1,

name: "Скафандр",

count: 1,

price: 19345

},

{

id: 2,

name: "Сельдереевый смузи",

count: 3,

price: 46

},

{

id: 3,

name: "Запасной парашют",

count: 1,

price: 1870

}

]

};

render() {

const total = this.state.cart.reduce((acc, p) => acc + p.price \* p.count, 0);

return (

<>

{this.state.cart.map((product) => (

<Product

product={product}

key={product.id}

/>

))}

<CartTotal total={total} />

</>

);

}

}

Этот компонент чуть сложнее предыдущих, так как у него есть внутреннее состояние и он использует другие компоненты в разметке. Мы рекомендуем поднимать общее состояние до ближайшего общего предка, в данном случае Cart. Для рендеринга списка товаров мы используем ранее написанный компонент Product и с помощью цикла превращаем состояние в разметку. Для отображения общей цены и НДС мы также воспользовались компонентом — CartTotal.

Обратите внимание, что НДС считается внутри компонента CartTotal, а общая сумма заказа передаётся в компонент уже вычисленной.

С одной стороны, мы могли бы вычислить всё в компоненте Cart, но расчёт НДС — зона ответственности компонента CartTotal.

С другой стороны, мы могли бы передать в CartTotal состояние корзины целиком, применяя this.state.cart, а уже внутри вычислить total и НДС. Но и так мы не делаем, потому что компоненту CartTotal не нужна вся информация о товарах: их количество, названия и другие сущности.

Старайтесь передавать в компонент только необходимые ему пропсы. Такие компоненты проще поддерживать и переиспользовать.

Сейчас корзина отображает товары, общую сумму заказа и все необходимые кнопки. Но нажатие на эти кнопки ни к чему не приводит — отсутствуют обработчики onClick. Мы добавим их позже, а пока разберёмся, как изменять количество товаров или очищать корзину целиком из дочерних компонентов в родительском компоненте Cart.

Из урока про [продвинутое использование пропсов](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/ab654c36-9a7d-445b-b9ab-07abb51ccff1/) вы узнали, что в качестве пропсов можно передавать JavaScript выражения. Функции тоже являются JavaScript выражениями, и мы можем передать их из родительского компонента дочерним. Именно так дочерние компоненты смогут общаться со своим родителем.

Напишем несколько методов в компоненте Cart, их мы будем передавать в дочерние компоненты. Начнём с простого — метода сброса корзины товаров:

Скопировать кодJSX

*// <Cart />*

...

clearCart = () => {

this.setState({ cart: [] });

};

...

Вызов этого метода очищает корзину товаров, присваивая пустой массив значению cart в состоянии компонента.

Теперь реализуем метод посложнее. Он должен принимать на вход товар из списка и проверять по id товара его наличие в состоянии внутри Cart. Если товар в наличии, нужно проверить его количество, переданное методу на вход. Так мы поймём, что делать с этим товаром. Например, если количество подаваемого на вход товара равно нулю, его нужно удалить из внутреннего состояния, а если оно больше нуля, то стоит записать это количество в state:

Скопировать кодJSX

*// <Cart />*

...

*//Передаём на вход товар*

updateProduct = (product) => {

*// Находим индекс товара в массиве this.state.cart*

const currentProduct = this.state.cart.findIndex(

(p) => p.id === product.id

);

*// Если индекс товара найден, выполняем следующие шаги*

if (currentProduct !== -1) {

*// Создаём копию корзины,*

*// так как оригинал состояния иммутабельный*

const newCart = [...this.state.cart];

*// Проверяем, что количество товара, переданного на вход, не равно нулю:*

if (product.count) {

*// Заменяем товар в копии на новый*

newCart[currentProduct] = product;

*// Записываем в состояние актуализированную копию*

this.setState({ cart: newCart });

} else {

*// Если количество товара, переданного на вход, равно нулю,*

*// мы запрашиваем подтверждение у пользователя*

*// и записываем его результат в переменную res*

const res = window.confirm("Удалить " + product.name + "?");

if (res) {

*// Если пользователь подтвердил удаление товара,*

*// удаляем его из копии*

newCart.splice(currentProduct, 1);

*// Записываем в состояние актуализированную копию*

this.setState({ cart: newCart });

}

}

}

};

...

Методы готовы, теперь нужно передать их в качестве пропсов дочерним компонентам:

Скопировать кодJSX

*// <Cart />*

...

render() {

const total = this.state.cart.reduce((acc, p) => acc + p.price \* p.count, 0);

return (

<>

{this.state.cart.map((product) => (

<Product

product={product}

key={product.id}

updateProduct={this.updateProduct}

/>

))}

<CartTotal total={total} clearCart={this.clearCart} />

</>

);

}

...

Мы передали методы компонента Cart дочерним компонентам: updateProduct в компонент Product и clearCart в компонент CartTotal.

Осталось только подготовить компоненты к использованию новых пропсов. Внутри компонента Product расположены две кнопки: для увеличения и уменьшения количества товара. Добавим обработчики onClick для этих кнопок:

Скопировать кодJSX

const Product = ({ product, updateProduct }) => {

const plus = () => {

updateProduct({

...product,

count: product.count + 1

});

};

const minus = () => {

updateProduct({

...product,

count: product.count - 1

});

};

return (

<div>

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

<p>{product.price}</p>

<button onClick={minus}>-</button>

<span>{product.count}</span>

<button onClick={plus}>+</button>

</div>

);

};

Добавим функцию updateProduct в пропсы компонента. Вызовем эту функцию внутри функций plus и minus и передадим в неё товар, изменяя только его количество: добавим единицу товара в plus и сократим количество товара на единицу в minus. Внутри компонента Product всегда будет актуальная информация о количестве товара, так как она приходит из пропсов и не мутируется напрямую в этом компоненте.

Теперь допишем компонент CartTotal. В него мы передаём функцию clearCart — это функция без аргументов, поэтому применим её прямо в обработчике onClick на кнопке «Очистить корзину»:

Скопировать кодJSX

const CartTotal = ({ total, clearCart }) => {

return (

<div>

<h1>{total}</h1>

<p>В том числе НДС: {total \* 0.2}</p>

<button onClick={clearCart}>Очистить корзину</button>

</div>

);

};

Как и в компоненте Product, в CartTotal добавился новый пропс — clearCart. Нажатие на кнопку «Очистить корзину» будет вызывать метод родительского компонента Cart и очищать содержимое корзины.

Взгляните, что у нас получилось:

Скопировать кодJSX

import React from "react";

const Product = ({ product, updateProduct }) => {

const plus = () => {

updateProduct({

...product,

count: product.count + 1

});

};

const minus = () => {

updateProduct({

...product,

count: product.count - 1

});

};

return (

<div>

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

<p>{product.price}</p>

<button onClick={minus}>-</button>

<span>{product.count}</span>

<button onClick={plus}>+</button>

</div>

);

};

const CartTotal = ({ total, clearCart }) => {

return (

<div>

<h1>{total}</h1>

<p>В том числе НДС: {total \* 0.2}</p>

<button onClick={clearCart}>Очистить корзину</button>

</div>

);

};

export default class Cart extends React.Component {

state = {

cart: [

{

id: 1,

name: "Скафандр",

count: 1,

price: 19345

},

{

id: 2,

name: "Сельдереевый смузи",

count: 3,

price: 46

},

{

id: 3,

name: "Запасной парашют",

count: 1,

price: 1870

}

]

};

updateProduct = (product) => {

const currentProduct = this.state.cart.findIndex(

(p) => p.id === product.id

);

if (currentProduct !== -1) {

const newCart = [...this.state.cart];

if (product.count) {

newCart[currentProduct] = product;

this.setState({ cart: newCart });

} else {

const res = window.confirm("Удалить " + product.name + "?");

if (res) {

newCart.splice(currentProduct, 1);

this.setState({ cart: newCart });

}

}

}

};

clearCart = () => {

this.setState({ cart: [] });

};

render() {

const total = this.state.cart.reduce((acc, p) => acc + p.price \* p.count, 0);

return (

<>

{this.state.cart.map((product) => (

<Product

product={product}

key={product.id}

updateProduct={this.updateProduct}

/>

))}

<CartTotal total={total} clearCart={this.clearCart} />

</>

);

}

}

Вот и всё! Наша корзина готова. Состояние от родительского компонента «погружается» в дочерние, а затем с помощью пропсов всплывает обратно к родительскому компоненту. Он, в свою очередь, решает, как ему поступать с новым состоянием, и только после этого все остальные компоненты получают обновлённые пропсы.

Для компонентов Product и CartTotal единственным источником истины является компонент Cart. То есть изменения, которые вызываются в одном компоненте, влияют на общее состояние приложения, другие же потребители данных сразу отзываются на эти изменения. Ещё это называется однонаправленный поток данных.

В этом уроке мы рассказали, как поднимать состояние вверх по дереву компонентов. Вы проделали большой путь! Впереди вас ждёт заключительный урок темы, в нём мы подведём промежуточные итоги и двинемся дальше.

# PropTypes: проверка типов

В предыдущем уроке мы рассказали про подъём состояния. Теперь поговорим о способах повысить надёжность React-приложений.

По мере того, как приложение развивается, а структура его компонентов усложняется, возникает необходимость проверять пропсы на соответствие ожидаемым типам. То есть проверять типы аргументов, которые передаются в компоненты.

Такую возможность предоставляет библиотека prop-types, которая до недавнего времени входила в состав библиотеки React. PropTypes представляет собой гибкую надстройку для использования в компонентах.

В режиме разработки вы сможете проверять props на наличие ошибок и несоответствий и устранять их до того, как с ними столкнутся пользователи.

## Проверка типов с помощью PropTypes

Для запуска этой проверки на пропсах компонента нужно использовать специальное свойство propTypes:

Скопировать кодJSX

import React from 'react';

import PropTypes from 'prop-types';

class OrderNumber extends React.Component {

render() {

return (

<span>Номер заказа: {this.props.orderId}</span>

);

}

}

OrderNumber.propTypes = {

orderId: PropTypes.number

};

PropTypes можно применять и к функциональным компонентам:

Скопировать кодJSX

function OrderNumber(props) {

return (

<span>Номер заказа: {props.orderId}</span>

);

}

OrderNumber.propTypes = {

orderId: PropTypes.number

};

Для этого необходимо добавить PropTypes напрямую к компоненту OrderNumber, как и в случае с классовыми компонентами.

PropTypes предоставляет набор валидаторов, которые могут использоваться для проверки типов пропсов, передаваемых в компоненты. В примере мы воспользовались PropTypes.number. Если в коде будет компонент OrderNumber c некорректным типом props.orderId, то в консоль выведется ошибка:

Скопировать кодJSX

<OrderNumber orderId="1138" /> *// Передаётся строка. Это приведёт к ошибке*

<OrderNumber orderId={1138} /> *// Передаётся число. Ошибки не будет*

Проверка типов с использованием propTypes выполняется только в режиме разработки, продакшн-сборка приложения будет игнорировать ошибки. Это сделано так из-за производительности: проверка типов порой бывает ресурсозатратной задачей.

## Валидаторы PropTypes

Рассмотрим примеры использования возможных валидаторов:

Скопировать кодJSX

import PropTypes from 'prop-types';

ProductPage.propTypes = {

*// Проверка пропса на соответствие определённому JS-типу.*

optionalString: PropTypes.string,

optionalNumber: PropTypes.number,

optionalBool: PropTypes.bool,

optionalArray: PropTypes.array,

optionalObject: PropTypes.object,

optionalSymbol: PropTypes.symbol,

optionalFunc: PropTypes.func,

*// Значение любого типа*

optionalAny: PropTypes.any

*// Всё, что может быть отрендерено:*

*// числа, строки, элементы или массивы*

*// (или фрагменты), содержащие эти типы*

optionalNode: PropTypes.node,

*// React-элемент*

optionalElement: PropTypes.element,

*// Тип React-элемента (например, MyComponent).*

optionalElementType: PropTypes.elementType,

*// Можно указать, что пропс должен быть экземпляром класса*

*// Для этого используется JS-оператор instanceof.*

optionalMessage: PropTypes.instanceOf(Review),

*// Вы можете задать ограничение конкретными значениями*

*// при помощи перечисления*

optionalEnum: PropTypes.oneOf(['Size', 'Color', 'Brand']),

*// Объект одного из нескольких типов*

optionalUnion: PropTypes.oneOfType([

PropTypes.string,

PropTypes.number,

PropTypes.instanceOf(Review)

]),

*// Массив объектов конкретного типа*

optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

*// Объект со свойствами конкретного типа*

optionalObjectOf: PropTypes.objectOf(PropTypes.number),

*// Объект с определённой структурой*

optionalObjectWithShape: PropTypes.shape({

color: PropTypes.string,

fontSize: PropTypes.number

}),

*// При наличии необъявленных свойств в объекте будут вызваны предупреждения*

optionalObjectWithStrictShape: PropTypes.exact({

name: PropTypes.string,

quantity: PropTypes.number

}),

*// Можно добавить `isRequired` к любому приведённому выше типу,*

*// чтобы показывать предупреждение,*

*// если пропс не передан*

requiredFunc: PropTypes.func.isRequired,

*// Можно добавить собственный валидатор.*

*// Он должен возвращать объект `Error` при ошибке валидации.*

*// Не используйте `console.warn` или `throw`:*

*// это не будет работать внутри `oneOfType`*

customProp: function(props, propName, componentName) {

if (!/matchme/.test(props[propName])) {

return new Error(

'Проп `' + propName + '` компонента' +

' `' + componentName + '` имеет неправильное значение'

);

}

},

*// Можно задать свой валидатор для `arrayOf` и `objectOf`.*

*// Он должен возвращать объект Error при ошибке валидации.*

*// Валидатор будет вызван для каждого элемента в массиве*

*// или для каждого свойства объекта.*

*// Первые два параметра валидатора*

*// — это массив или объект и ключ текущего элемента*

customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName, location, propFullName) {

if (!/matchme/.test(propValue[key])) {

return new Error(

'Проп `' + propFullName + '` компонента' +

' `' + componentName + '` имеет неправильное значение'

);

}

})

};

Как вы могли заметить, PropTypes позволяет комбинировать валидаторы в сложные структуры.

## Ограничение на один дочерний компонент

С помощью PropTypes.element вы можете указать, что в качестве дочернего может быть передан только один элемент:

Скопировать кодJSX

import PropTypes from 'prop-types';

class ContainerBlock extends React.Component {

render() {

*// Это должен быть ровно один элемент.*

*// Иначе вы увидите предупреждение*

const children = this.props.children;

return (

<div>

{children}

</div>);

}

}

ContainerBlock.propTypes = {

children: PropTypes.element.isRequired

};

Использование PropTypes поможет вам сэкономить время на написание кода, а другим разработчикам — избегать ошибок при использовании ваших компонентов.

Пример.

import React from 'react';

import PropTypes from 'prop-types';

import './styles.css';

const messagePropTypes = PropTypes.shape({

id: PropTypes.number.isRequired,

user: PropTypes.string.isRequired,

replyTo: PropTypes.number,

text: PropTypes.string.isRequired

});

const Message = ({ message, repliedMessage, className = 'message' }) => (

<div className={className}>

{repliedMessage && <RepliedMessage message={repliedMessage} />}

<h3>{message.user}</h3>

<p>{message.text}</p>

</div>

);

Message.propTypes = {

message: messagePropTypes.isRequired,

repliedMessage: messagePropTypes,

className: PropTypes.string

};

const RepliedMessage = ({ message }) => <Message message={message} className={'replied-message'} />;

RepliedMessage.propTypes = {

message: messagePropTypes

};

const Chat = ({ thread }) => (

<div className="tread">

{thread.map(message => {

const repliedMessage = thread.find(m => m.id === message.replyTo);

return <Message key={message.id} repliedMessage={repliedMessage} message={message} />;

})}

</div>

);

Chat.propTypes = {

thread: PropTypes.arrayOf(messagePropTypes.isRequired).isRequired,

};

export default class App extends React.Component {

state = {

thread: [

{

id: 1,

user: 'Тамара',

text: 'Всем привет! Кто в курсе, когда в нашем доме отключат горячую воду?'

},

{

id: 2,

user: 'Алексей',

replyTo: 1,

text: 'В подъезде висит объявление, скоро буду там, сфотографирую и пришлю сюда'

},

{

id: 3,

user: 'Катя',

replyTo: 2,

text: 'О! Спасибо! Ждём! :)'

}

]

};

render() {

return (

<div className="App">

<Chat thread={this.state.thread} />

</div>

);

}

}