**Введение**

Пора чуть глубже погрузиться в тонкости внутреннего устройства «Реакта».

После изучения этой темы вы будете лучше понимать принципы его работы и сможете писать более эффективный и производительный код.

Вы изучите тему оптимизации — одну из любимых на собеседованиях.

Также познакомитесь с некоторыми дополнительными методиками и паттернами, которые часто используются в проектах на «Реакте». Будет интересно.

# Списки и ключи

В предыдущем спринте мы рассказывали [об алгоритме «Реакта», который называется “reconciliation”.](https://praktikum.yandex.ru/trainer/web/lesson/b3814563-4d75-427e-9fa9-92f7f6d3b4a7) Он неплохо справляется со своей задачей, но иногда ему требуется наша помощь.

Пробегаясь по ветвям дерева виртуального DOM, алгоритм сравнивает узлы в новом и старом дереве по двум признакам:

* по типу — это может быть элемент, компонент, строка или пустой узел;
* по названию тега или имени компонента.

Если хотя бы один из этих признаков не совпадает, значит, узел и все его дочерние узлы в дереве должны быть полностью заменены. Старый узел будет удалён из DOM, а компонент — размонтивирован. На их место будет вставлена новая ветвь элементов. Или не будет, если в обновлённом виртуальном дереве на их месте ничего нет.

В противном случае, когда оба условия выполняются, значит, перед нами тот же самый узел. Если это DOM-элемент, «Реакт» при необходимости обновит его атрибуты и отправится проверять его «детей» — вглубь кроны дерева. Если это компонент, «Реакт» обновит его пропсы и повторно вызовет рендер, чтобы сравнить изменения в ветви, за которую отвечает этот компонент.

Этот простой алгоритм работает отлично, за исключением некоторых случаев.

# Списки в «Реакте»

Представьте, что вы пишете приложение-мессенджер, в котором есть список чатов, отсортированных по времени последнего сообщения. Каждый элемент этого списка — экземпляр компонента Chat. Массив с данными чатов приходит из API в таком виде:

Скопировать кодJSX

const chats = [{

id: 10,

name: 'Gregory',

lastMessageAt: '20:45',

}, {

id: 5,

name: 'Allison',

lastMessageAt: '12:31',

}, {

id: 3,

name: 'James',

lastMessageAt: '07:40',

}];

В JSX работа со списками осуществляется с помощью метода map:

Скопировать кодJSX

ReactDOM.render((

<>

<h2>Чаты</h2>

{chats.map((chat) => (

<Chat id={chat.id} name={chat.name} lastMessageAt={chat.lastMessageAt} />

))}

</>

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

При этом сам компонент Chat может выглядеть предельно просто:

Скопировать кодJSX

function Chat(props) {

return (

<div className="chat">

<img src={`img/${props.id}.png`} width="75" />

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

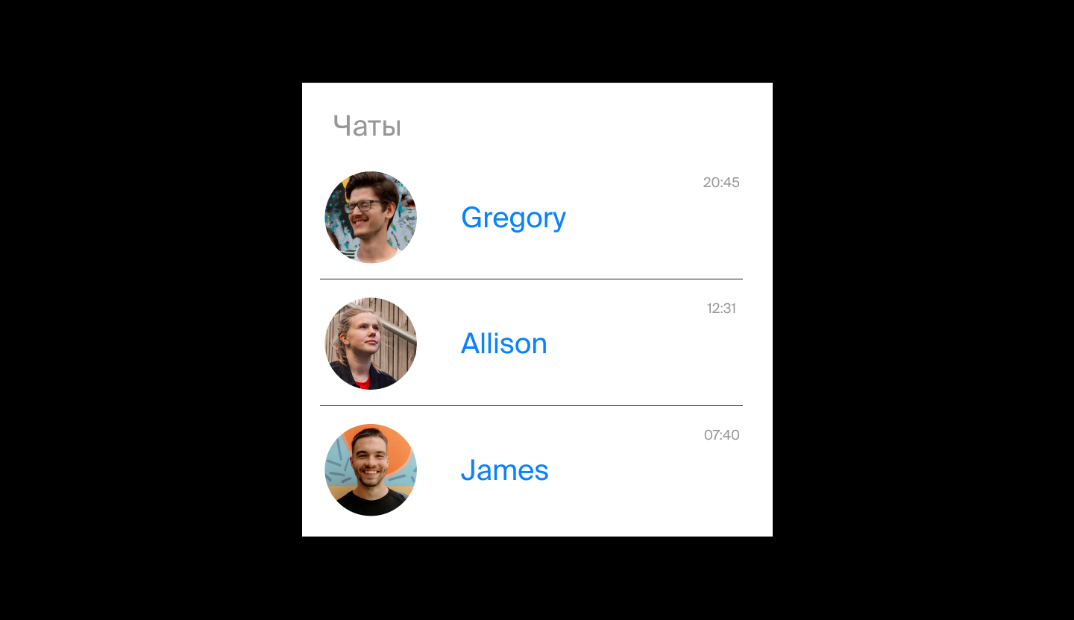
<div className="date">{props.lastMessageAt}</div>

</div>

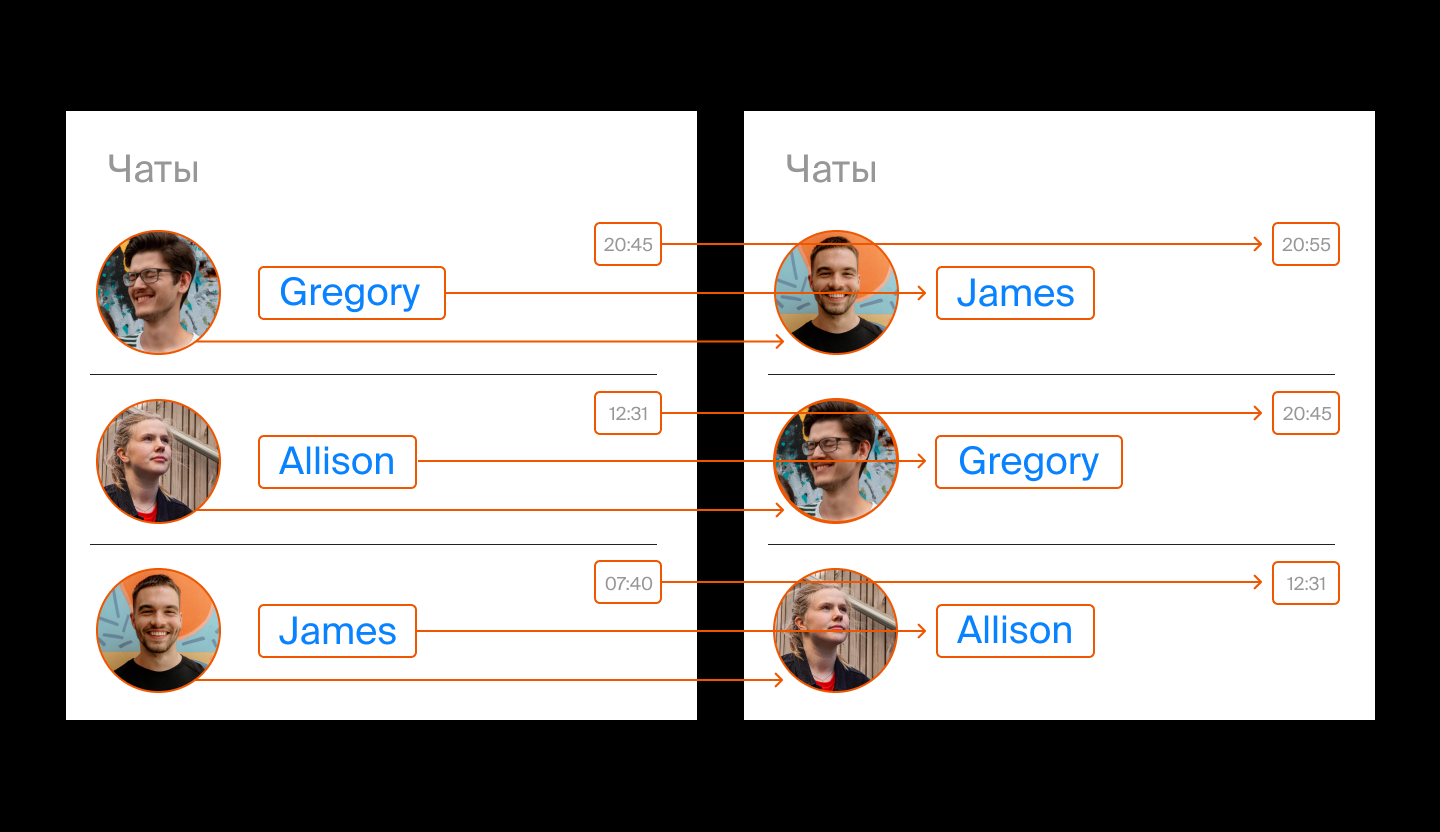
);

}

А такое получится приложение:

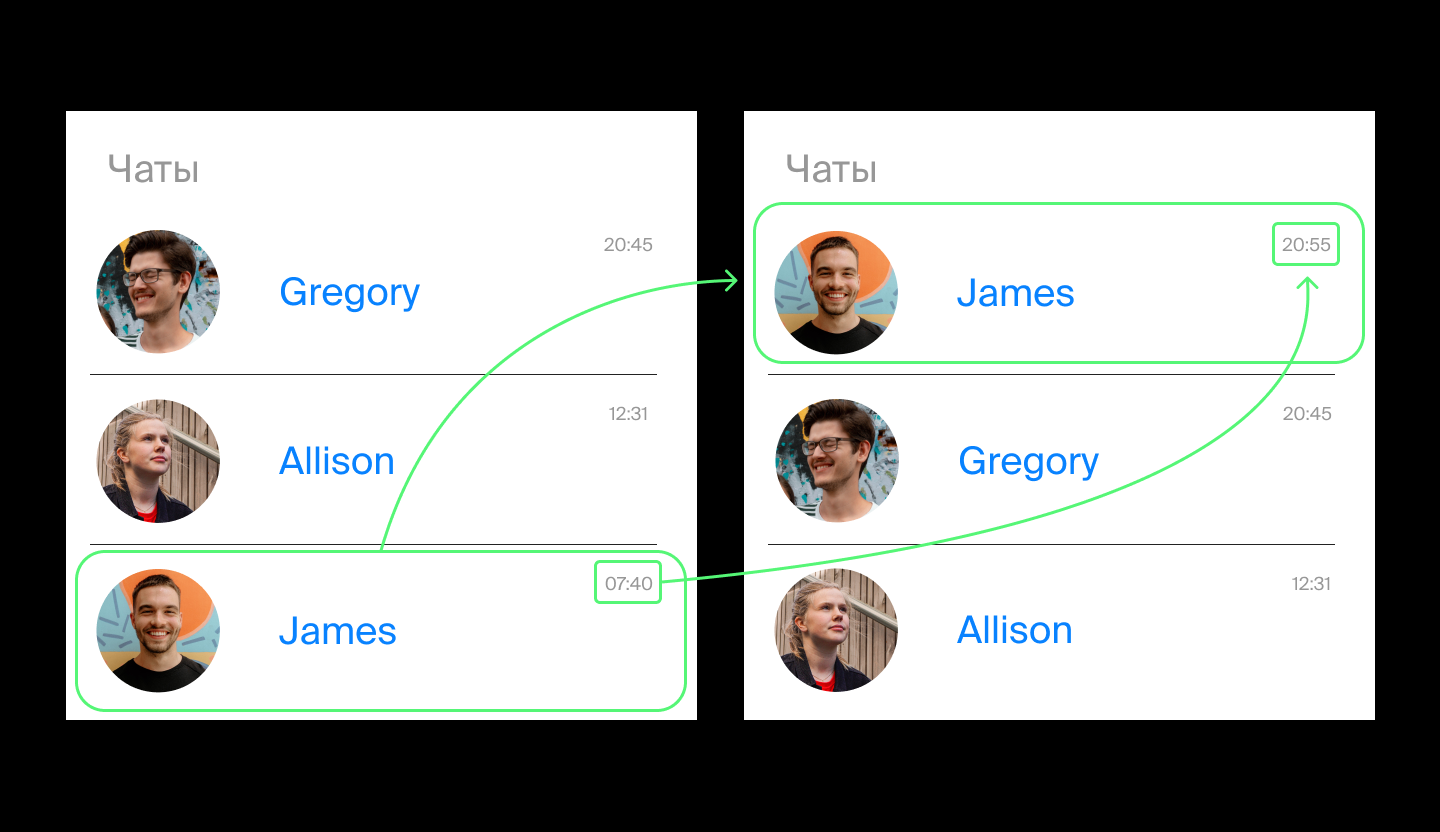


Когда в один из чатов придёт новое сообщение, он должен переместиться в верх списка, а все остальные чаты должны сдвинуться на одну позицию вниз. Но когда это произойдёт, алгоритм «Реакта» начнёт по порядку сравнивать старое и новое дерево элементов и не сможет понять, что узлы поменялись местами. Тип узла и имя компонента останутся прежними для каждого элемента списка, поэтому он начнёт искать изменения в их внутреннем содержимом (аватарках, именах пользователей, времени) и сделает много лишних действий в DOM.



Есть и более серьёзная проблема: если в этих компонентах хранится внутреннее состояние, например счётчики непрочитанных сообщений, то их значения перемешаются, потому что «Реакт» всего лишь заменит текстовое содержимое у всех чатов, оставив сами экземпляры компонентов в предыдущем порядке.

На самом же деле ему достаточно было бы обновить содержимое только одного чата и переместить его DOM-элемент в верх списка:



Чтобы помочь алгоритму понять, что элементы просто поменялись местами, нужно применить специальный атрибут key (англ. «ключ») — уникальный идентификатор узла, который используется как третий отличительный признак при сравнении. Теперь, когда «Реакт» будет сравнивать новые списки чатов, он сможет опираться на этот атрибут, чтобы понять, что элементы изменили свой порядок внутри списка.

Скопировать кодJSX

ReactDOM.render(

<>

<h2>Чаты</h2>

{chats.map((chat) => (

<Chat

key={chat.id}

id={chat.id}

name={chat.name}

lastMessageAt={chat.lastMessageAt}

/>

))}

</>,

document.querySelector("#root")

);

# Ключи вне списков

У key есть и другие применения, кроме списков. Например, после выбора чата может понадобиться заменить весь список сообщений, за который отвечает компонент MessageList. Иногда самое простое решение — дать команду «Реакту» полностью демонтировать старый компонент MessageList и смонтировать на его месте новый. Чтобы это сделать, компоненту можно задать новый ключ, соответствующий ID выбранного чата:

Скопировать кодJSX

ReactDOM.render(

<>

<h2>Чаты</h2>

{chats.map((chat) => (

<Chat

key={chat.id}

id={chat.id}

name={chat.name}

lastMessageAt={chat.lastMessageAt}

/>

))}

<h2>Сообщения</h2>

<MessageList key={selectedChatId} />

</>,

document.querySelector("#root")

);

Таким образом с помощью смены ключа мы сообщаем «Реакту», что текущий экземпляр компонента MessageList нужно полностью заменить новым экземпляром.

Пример

import React, { useState, useEffect } from 'react';

import './Tips.css';

function Tips() {

useEffect(() => {

fetch('https://api.nomoreparties.co/todays-tips-rus').then((res) => {

return res.json();

}).then((res) => {

setList(Object.values(res));

})

}, []);

const [list, setList] = useState({});

return (

<div className="tips">

<ul className="tips\_\_list">

{list &&

list.map((item, i) => {

return (

<li key={i} className="tips\_\_item">{item.tip}</li>

);

})

}

</ul>

</div>

);

}

export default Tips;

# Работа с формами

В «Реакте» принято использовать «управляемые компоненты». Это элементы управления в формах, значения которых синхронизируются со стейтом компонента, где они находятся. Это удобно, потому что позволяет напрямую управлять содержимым форм из JavaScript.

Не запутайтесь в терминологии: управляемые компоненты не являются «Реакт-компонентами» сами по себе. Это простые элементы формы, значение которых связано с одной из переменных состояния «Реакт-компонента», в составе которого они находятся.

Вот пример такого управляемого компонента:

Скопировать кодJSX

function Input() {

*// Стейт, в котором содержится значение инпута*

const [value, setValue] = React.useState('');

*// Обработчик изменения инпута обновляет стейт*

function handleChange(e) {

setValue(e.target.value);

}

return (

*// Значение элемента «привязывается» к значению стейта*

<input type="text" value={value} onChange={handleChange} />

);

}

ReactDOM.render((

<Input />

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

Раньше мы уже приводили подобные примеры. Это удобный подход, потому что с ним можно легко получить текущее значение элемента управления (например, чтобы отправить на бэкенд) и так же легко изменить его.

Теперь добавим две кнопки: одна из них выводит текущее значение инпута в alert, а вторая очищает его. В обоих случаях мы изменяем стейт без необходимости работать с инпутом напрямую.

Скопировать кодJSX

function Input() {

const [value, setValue] = React.useState('');

function handleChange(e) {

setValue(e.target.value);

}

function handleShow() {

alert(value);

}

function handleClear() {

setValue('');

}

return (

<>

<input type="text" value={value} onChange={handleChange} />

<button onClick={handleShow}>Показать</button>

<button onClick={handleClear}>Очистить</button>

</>

);

}

Это короткий урок, а вот тема объёмная. Чтобы лучше разбираться в ней, изучите подробное описание работы с веб-формами в официальном руководстве React. Из неё вы узнаете про элементы textarea и select и «неуправляемые компоненты».

<https://ru.reactjs.org/docs/forms.html>

Пример

Мы уже добавили в Calorie-Zen форму для записи продуктов и калоража. Осталось её настроить. Для этого в компоненте FoodAdder:

1. Добавьте атрибуты onSubmit и onChange соответствующим элементам внутри JSX, в качестве значений передав обработчики. Заготовки обработчиков уже есть в коде.
2. Внутри handleChange напишите код, который будет обновлять состояние, если пользователь внесёт изменения. Для этого вызовите функцию this.setState, передав ей обновлённый объект состояния. Маленькая подсказка: чтобы получить имя инпута, можно использовать e.target.name, а чтобы получить значение — e.target.value.
3. После отправки формы нужно сбросить состояние на значения по умолчанию.

import React from 'react';

import './FoodAdder.css';

class FoodAdder extends React.Component {

constructor(props) {

super(props);

this.state = {

food: '',

calories: ''

};

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

this.handleSubmit = this.handleSubmit.bind(this);

}

handleChange = (e) => {

this.setState({

[e.target.name]: e.target.value

})

// установите нужное состояние

// используйте e.target.name и e.target.value

}

handleSubmit = (e) => {

e.preventDefault();

this.setState(

{

food: '',

calories: ''

}

)

this.props.handleSubmit(this.state.food, this.state.calories);

}

render() {

// добавьте элементами обработчики handleSubmit и handleChange

return (

<form className="food-adder" onSubmit={this.handleSubmit}>

<input onChange={this.handleChange} name="food" value={this.state.food} type="text" required placeholder="Добавьте продукты" />

<input onChange={this.handleChange} name="calories" value={this.state.calories} type="number" required placeholder="Калории" />

<button className="food-adder\_\_submit"></button>

</form>

);

}

}

export default FoodAdder;

# Рефы

Рефы (англ. ref от reference, «указатель») — ещё один способ хранения данных внутри функциональных компонентов. Они используются сразу для двух целей.

# Прямой доступ к DOM-элементам

«Реакт» реализует декларативный подход к описанию веб-интерфейса. Это означает, что обычно разработчику не нужно взаимодействовать с реальными DOM-элементами и обращаться к ним напрямую. Всё необходимое можно делать в JSX: устанавливать CSS-классы, задавать стили или создавать подписку на события.

Из этого правила есть исключения, и иногда наличие прямого указателя на реальный DOM-элемент необходимо. «Реакт» предоставляет такой механизм — рефы. В [официальном руководстве React](https://ru.reactjs.org/docs/refs-and-the-dom.html) приводятся ситуации, в которых использование рефов для этой цели оправдано:

* управление фокусом, выделение текста или воспроизведение медиа;
* императивный вызов анимаций;
* интеграция со сторонними DOM-библиотеками.

**Для функциональных компонентов** существует специальный хук useRef. Он возвращает объект, который с помощью JSX-атрибута ref можно присвоить любому элементу, чтобы получить доступ к нему. Этот объект содержит одно единственное поле current. Именно в это поле React запишет указатель на DOM-элемент, когда будет формировать DOM-дерево.

Так это выглядит:

Скопировать кодJSX

function VideoPlayer() {

const videoRef = React.useRef(); *// записываем объект, возвращаемый хуком, в переменную*

function handleClick() {

videoRef.current.play(); *// вызываем нужный метод на поле current объекта*

}

return (

<>

<video ref={videoRef} src="./clip.mp4" /> // указали элементу атрибут ref => получили прямой доступ к DOM-элементу

<button onClick={handleClick}>▶️</button> /\* привязали обработчик к кнопке \*/

</>

);

}

**Для классовых компонентов** всё почти так же, только для создания рефа используется функция React.createRef, а сам реф обычно записывается в this:

Скопировать кодJSX

class VideoPlayer extends React.Component {

constructor() {

super();

this.videoRef = React.createRef(); *// создали реф и записали его в переменную –– она будет свойством у this*

}

handleClick = () => {

this.videoRef.current.play(); *// аналогично вызываем нужный метод на поле current объекта*

};

render() {

return (

<>

<video ref={this.videoRef} src="./clip.mp4" /> // всё так же, как у функциональных компонентов, но нужно добавить this

<button onClick={this.handleClick}>▶️</button> // всё так же, как у функциональных компонентов, но нужно добавить this

</>

);

}

}

# Рефы как локальные переменные

У рефов есть дополнительное назначение в функциональных компонентах. Иногда нужно записать какие-то данные внутри компонента, но использовать стейт при этом не получается.

Предположим, мы хотим вести счётчик рендеров в функциональном компоненте. Если используем стейт с помощью хука useState, то каждый раз при его обновлении будет снова вызываться рендер и обновлять счётчик — получится замкнутый круг, и приложение зависнет.

В классовых компонентах для этого используют свойства экземпляра класса: просто сохраняют всё нужное в this и при необходимости считывают.

В функциональных компонентах вместо этого используют рефы. Вот как должен выглядеть этот пример:

Скопировать кодJSX

function MessageComposer() {

const [value, setValue] = React.useState('');

const counterRef = React.useRef(0);

function handleChange(e) {

setValue(e.target.value);

}

return (

<>

<input type="text" value={value} onChange={handleChange} />

<h4>Рендеров: {++counterRef.current}</h4>

</>

);

}

Здесь мы используем паттерн «управляемый инпут», который был описан в предыдущем уроке. У этого компонента происходит рендер каждый раз, когда пользователь меняет текст в поле ввода.

**Старайтесь не злоупотреблять рефами и использовать их только в случае необходимости.**

# Чистые компоненты

Благодаря технологии Virtual DOM «тяжёлые операции» в реальном DOM сведены к минимуму. Но в виртуальном DOM, наоборот, зачастую производится избыток «лёгких операций», что может сказываться на производительности кода. Важно знать о внутреннем устройстве «Реакта», чтобы писать оптимальный код.

# Проблема оптимизации: цепная реакция рендеров

Как вы знаете из урока про [«Жизненный цикл классовых компонентов»](https://praktikum.yandex.ru/trainer/web/lesson/3c792d50-412d-49f6-9f03-318e0bfac201/task/37bff174-ce2a-4f40-9451-4e69e7a6fedb/), если в каком-то месте виртуального дерева произошли изменения и был вызван рендер соответствующего компонента, это вызывает цепную реакцию рендеров для всех его дочерних компонентов. Часто это бывает не обязательно и даже избыточно. Особенно, если в исходном компоненте, к примеру, всего лишь поменялся CSS-класс.

Рассмотрим тот же пример со списком чатов. Допустим, при наведении курсора вы хотите, чтобы отображалась панель действий с чатом: «Не беспокоить» и «Удалить». Конечно, лучше это сделать на чистом CSS, но это не всегда возможно.

Добавим внешний компонент ChatWithTools, который содержит Chat внутри себя и реагирует на события мыши, чтобы отображать дополнительные элементы.

Скопировать кодJSX

const ChatWithTools = (props) => {

const [shouldShowTools, setShouldShowTools] = React.useState(false);

function handleMouseEnter() {

setShouldShowTools(true);

}

function handleMouseLeave() {

setShouldShowTools(false);

}

return (

<div className="chat-with-tools" onMouseEnter={handleMouseEnter} onMouseLeave={handleMouseLeave}>

<Chat {...props} />

{shouldShowTools && (

<div className="tools">

<button>Не беспокоить</button>

<button>Удалить</button>

</div>

)}

</div>

);

};

const chats = */\* ... \*/*

ReactDOM.render((

<>

<h2>Чаты</h2>

{chats.map((chat) => (

<ChatWithTools id={chat.id} name={chat.name} lastMessageAt={chat.lastMessageAt} />

))}

</>

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

Обратите внимание на «пробрасывание» пропсов: <Chat {...props} />. С помощью такой конструкции можно копировать в дочерний компонент сразу все собственные пропсы. Так в этом примере мы смогли разом пробросить все пропсы ChatWithTools в дочерний Chat.

В таком коде есть потенциальная проблема. Как мы говорили в одном из предыдущих уроков, отрисовка компонента всегда вызывается при отрисовке его родительского компонента. Убедиться в этом можно, добавив в структуру дочернего компонента генератор случайных чисел — при каждом вызове рендера случайное число будет меняться, и мы заметим это в пользовательском интерфейсе:

Скопировать кодJSX

const Chat = (props) => {

return (

<div className="chat">

<img src={`img/${props.id}.png`} width="75" />

<h2>{Math.random()}</h2>

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

</div>

);

};

Если бы не случайное число, никаких изменений бы не произошло, а рендер вызывался бы напрасно. С другой стороны, структура Chat могла быть сложнее и содержала бы в себе другие дочерние компоненты, которые в свою очередь содержали бы ещё компоненты. Для них рендер был бы вызван напрасно.

Чтобы этого не происходило, нужно использовать специальный тип компонентов, рендер которых происходит не при любом рендере их родителей, а только тогда, когда меняются их собственные пропсы. Такие компоненты называются «чистыми».

В классовом и функциональном подходе чистые компоненты создают по-разному.

# Создание чистого функционального компонента

Чтобы сделать функциональный компонент чистым, достаточно обернуть его в специальную функцию React.memo:

Скопировать кодJSX

const Chat = React.memo((props) => {

return (

<div className="chat">

<img src={`img/${props.id}.png`} width="75" />

<h2>{Math.random()}</h2>

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

</div>

);

});

Теперь при наведении мыши не происходит цепной реакции — она сразу «обрывается» на компоненте Chat, так как его пропсы не меняются:

Так мы используем ресурсы более разумно.

# Создание чистого классового компонента

Сделать чистым классовый компонент ещё проще: надо лишь наследоваться не от React.Component, а от React.PureComponent:

Скопировать кодJSX

class Chat extends React.PureComponent {

render() {

return (

<div className="chat">

<img src={`img/${this.props.id}.png`} width="75" />

<h2>{Math.random()}</h2>

<div className="date">{this.props.lastMessageAt}</div>

</div>

);

};

}

И React.memo, и React.PureComponent работают по схожему принципу: они запоминают последние переданные пропсы и последний результат рендера своего дочернего компонента. При следующей попытке рендера, если пропсы не изменились, они просто возвращают закэшированный результат. Если же пропсы изменились, вызывается рендер, после чего сохраняется новый кэш и последние пропсы.

Старайтесь всегда использовать чистые компоненты, кроме случаев, когда компоненты должны обновляться вместе со своими родителями.

# Поверхностное сравнение

React.memo и React.PureComponent определяют изменения в пропсах по алгоритму «поверхностного сравнения» (англ. shallow equal): предыдущее значение каждого пропса сравнивается с новым оператором ===.

Поэтому нужно избегать ситуаций, когда значения пропсов со сложными типами создаются в момент передачи, например:

Скопировать кодJSX

<MyPureComponent

userNames={['Gregory', 'James', 'Allison']}

onClick={() => console.log(1)}

/>

При каждом рендере значения этих пропсов по сути не меняются, но в действительности они создаются заново, значит, фильтр чистого компонента просто не сработает. Это происходит потому, что в JavaScript сложные типы данных не эквивалентны друг другу:

Скопировать кодJSX

['Gregory', 'James', 'Allison'] === ['Gregory', 'James', 'Allison'] *// => false!*

(() => console.log(1)) === (() => console.log(1)) *// => false!*

Чистые компоненты, которым задаются пропсы таким образом, работать не будут. В будущем мы научимся избегать подобных ситуаций.

# Компоненты более высокого порядка

В этой теме мы уже рассмотрели функцию React.memo. Эта функция — пример паттерна проектирования, который называется HOC (от англ. higher-order component, «компонент более высокого порядка»).

НОС-компоненты позволяют добавить функциональность одному или нескольким имеющимся компонентам. Если какие-то компоненты испытывают потребность в общей функциональности, можно обернуть их в HOC-компонент и описать в нём всё, что нужно.

Функция React.memo запоминает последние пропсы и кэширует результат рендера своего дочернего компонента. При следующем рендере, если пропсы не изменились, она возвращает закэшированный результат. Такое поведение универсально и применимо почти к любому компоненту.

HOC-компонент может менять не только логику, но и внешний вид компонента. Другой пример, который мы рассматривали ранее — ChatWithTools. Это компонент более высокого порядка по отношению к компоненту Chat. HOC-компоненты часто имеют слово "with" в названии, указывая на то, какая именно функциональность добавляется.

Вы можете прочитать о НОС-компонентах подробнее в официальном руководстве React:

[Компоненты высшего порядка - React](https://ru.reactjs.org/docs/higher-order-components.html)

# Заключение

В этой теме вы ближе познакомились с внутренним устройством библиотеки React.js.

Вы узнали, как работает движок «Реакта», что представляет из себя виртуальный DOM, а также зачем нужны ключи. Научились правильно работать с формами и получать прямой доступ к DOM-элементам с помощью рефов. Вы познакомились с идеями оптимизации и научились сокращать число отрисовок с помощью чистых компонентов.

Впереди вас ждёт проектная работа и множество новых интересных тем.