**Введение. Продвинутые возможности React**

Всё меняется и улучшается, а когда речь про фронтенд, то скорость изменений тут часто превышает скорость усвоения информации. Но не расстраивайтесь, что-то остаётся вечным: например, борьба объектно-ориентированного и функционального подходов к созданию программного обеспечения. Эффект от этой борьбы заметен и в библиотеке React.

В предыдущих уроках мы упоминали «глупые компоненты». Они не имеют собственного состояния, а создаются конструкцией function Component(){...}. «Глупыми» или stateless их называли, потому что до недавнего времени использование жизненного цикла компонента и конструкций this.state и this.setState в React было возможно только в компонентах, созданных с помощью классов: class Component extends React.Component{...}. Но разработчики React придумали механизм хуков, и начиная с версии 16.8 API хуков стало доступно в стабильной версии библиотеки. Компоненты, созданные через function или const, смогли быть как stateless, так и stateful.

Это нововведение изменило многое: процесс создания компонентов, хранение их логики, удобство декомпозиции. При этом осталась преемственность и поддержка классовых компонентов, а также сохранилась возможность реагировать на разные этапы жизненного цикла компонента (для это создали специальный встроенный хук). Разработка стала ещё более декларативной и быстрой. Подробнее обо всём этом расскажем в следующих уроках спринта.

Кроме этого, разберём некоторые особенности и способы работы с классовыми компонентами. Вы узнаете про компоненты более высокого порядка, императивную работу с DOM-элементами в React и чистые компоненты. На этом изучение классовых компонентов завершится, и мы перейдём к stateful функциональным компонентам.

# HOC: Компоненты высшего порядка

В этом уроке расскажем, как реорганизовывать приложение по мере его роста. В приложениях часто появляются компоненты, которые незначительно различаются, но обладают схожей функциональностью. React предлагает подходы для работы с такими компонентами. Чтобы лучше понять ценность этих подходов, придётся покодить.

Для этого отправимся в веб-студию вместе с Галей, начинающей веб-разработчицей. Когда будете готовы к погружению — жмите кнопку.

Ладно, поняла. Постараюсь быстренько всё сделать.

Приступим к работе. У нас как раз есть похожий код такого компонента. Выведем на страницу заголовок и описание с кнопкой, при нажатии на которую откроется галерея изображений:

Скопировать кодJSX

import React from 'react';

class Gallery extends React.Component {

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

this.state = {

isToggled: false,

}

}

onButtonClick () {

this.setState({

isToggled: !this.state.isToggled,

})

}

render() {

return (

<section>

<h2>{this.props.title}</h2>

<p>{this.props.description}</p>

<button onClick={this.onButtonClick}>

{this.state.isToggled ? 'Скройте это немедленно!' : 'Хочу посмотреть!'}

</button>

{this.state.isToggled && (

<ul>

{

this.props.list.map((item) => (

<li key={item.id}>

<img src={item.image} alt={item.description} />

</li>

))

}

</ul>

)}

</section>

)

}

}

export default Gallery;

В state хранится булевое значение isToggled, которое показывает или скрывает галерею изображений. В props передадим описание отеля и массив объектов с фотографиями и описанием. Вызовем компонент в App:

Скопировать кодJSX

import React from 'react';

import styles from './app.module.css';

import Gallery from '../gallery/gallery';

const galleryData = {

*// Здесь хранятся данные для изображений, заголовка и описания.*

}

class App extends React.Component {

render() {

return (

<div className={styles.app}>

{/\* Другие компоненты в App \*/}

<Gallery {...galleryData} />

</div>

)

}

}

Основная часть написана, остаётся только поработать с CSS, и базовый вариант компонента будет готов.

Василий

Галя, тут небольшой апдейт по задаче. На созвоне продакт сказал, что нужно начать прорабатывать детальную страницу каждого номера в отеле. Можешь заодно и для детальной страницы сделать простое MVP?

Галя

Кажется, что сегодня я уже не успею его закончить. Там много?

Василий

Нет, можно сделать что-то похожее на галерею. По нажатию на кнопку будем выводить полное описание номера, а потом прикрутим туда бронирование и оплату. Ну и 3D-тур по номеру тоже. Это всё у нас готово.

Галя

Ааа, тогда всё достаточно просто. За час соберу.

Василий

Круто, спасибо!

Сроки поджимают. Значит, воспользуемся функциональностью из компонента Gallery. Возьмём оттуда работу со стейтом isToggled и чуть-чуть поменяем JSX:

Скопировать кодJSX

*// card/card.js*

import React from "react";

class Card extends React.Component {

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

this.state = {

isToggled: false,

};

}

onButtonClick() {

this.setState({

isToggled: !this.state.isToggled,

});

}

render() {

return (

<section>

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

<img src={this.props.image} alt={this.props.description} />

<span>{this.props.caption}</span>

<button onClick={this.onButtonClick}>

{this.state.isToggled ? "Скрыть" : "Показать полное описание"}

</button>

{this.state.isToggled && <div>{this.props.text}</div>}

</section>

);

}

}

export default Card;

Теперь можно подключить компонент Card:

Скопировать кодJSX

*// app/app.js*

import React from "react";

import styles from "./app.module.css";

import Gallery from "../gallery/gallery";

import Card from "../card/card";

const galleryData = {

*// Здесь хранятся данные для изображений, заголовка и описания для галереи.*

};

const cardData = {

*// Здесь хранятся данные о карточке номера отеля.*

};

class App extends React.Component {

render() {

return (

<div className={styles.app}>

{/\* Другие компоненты в App \*/}

<Gallery {...galleryData} />

<Card {...cardData} />

</div>

);

}

}

Card работает так же, как и компонент Gallery. Это те самые схожие компоненты, о которых мы говорили в начале урока. Отличие только в возвращаемой разметке и пропсах. Сейчас мы решили проблему схожих компонентов путём копирования: взяли Gallery и создали почти такой же компонент Card. Если появится ещё несколько схожих компонентов, придётся снова копировать поведение из onButtonClick и объявление стейта. Это утомительно, а в будущем ещё и увеличит время на поддержание и доработку кода.

Для решения таких задач существует способ объединения логики компонентов — создание HOC (от англ. higher-order component, «компонент более высокого порядка»). В качестве аргумента HOC получает оборачиваемый компонент и необходимые для работы свойства, а возвращает насыщенный свойствами и функциональностью оригинальный компонент. При этом HOC не изменяет и не наследует поведение компонента, а только оборачивает его посредством композиции.

НОС можно сравнить с удобрениями для растений. С помощью правильных удобрений можно вырастить из вялого фикуса целое дерево (а потом лежать под ним и представлять, что вы в тёплых краях). При этом фикус так и останется фикусом, а не поменяет вид.

Напишем простой HOC, который будет отрисовывать оборачиваемый компонент. HOC принято хранить в отдельной директории hocs на уровне директории components, а файлы называть с with — так другие разработчики смогут сразу понять их назначение. Назовём наш компонент withToggle — в соответствии с основной функциональностью. Порядок действий по созданию HOC простой — нужно поэтапно перенести общую функциональность компонентов Card и Gallery в withToggle:

1. Создадим HOC и обернём компонент Card.
2. Наполним компонент Card свойствами HOC.
3. Перенесём стейт isToggled и хендлер onButtonClick в HOC.
4. Передадим оборачиваемому компоненту все необходимые свойства.

## Создадим withToggle и обернём Card в HOC:

Скопировать кодJSX

*// hocs/with-toggle.js*

import React from 'react';

*// Представим withToggle в виде обычного функционального компонента.*

const withToggle = (WrappedComponent) => {

return (

<WrappedComponent />

*// Вернём переданный компонент.*

)

};

export default withToggle;

*// components/app/app.js*

import React from 'react';

import styles from './app.module.css';

import withToggle from '../hoc/with-toggle';

import Card from '../card/card';

const WithToggleCard = withToggle(Card);

*// Присвоим переменной withToggle обёрнутый компонент Card в HOC.*

class App extends React.Component {

render() {

return (

<div className={styles.app}>

<WithToggleCard />

{/\* Другие компоненты в App \*/}

</div>

)

}

};

## Добавим пропсы

Текущая реализация компонента нас не устраивает: как минимум потерялись пропсы, а кроме того — описание состояния и setState остались в компонентах Gallery и Card. Исправим это, добавив props:

Скопировать кодJSX

*// hocs/with-toggle.js*

import React from 'react';

*// Представим withToggle в виде обычного функционального компонента.*

const withToggle = WrappedComponent => props => {

return (

<WrappedComponent {...props} />

*// Вернём переданный компонент.*

)

};

export default withToggle;

*// components/app/app.js*

import React from 'react';

import styles from './app.module.css';

import withToggle from '../hoc/with-toggle';

import Card from '../card/card';

const cardData = {

*// Здесь хранятся данные для изображений, заголовка и описания для галереи.*

}

const WithToggleCard = withToggle(Card);

*// Присвоим переменной WithToggleCard обёрнутый компонент Card в HOC.*

class App extends React.Component {

render() {

return (

<div className={styles.app}>

<WithToggleCard {...cardData}/>

{/\* Передадим cardData в HOC \*/}

{/\* Другие компоненты в App \*/}

</div>

)

}

};

Простой тип HOC, который не содержит в себе состояния, а отрисовывает переданный в него компонент с пропсами, можно встретить достаточно часто. Например, когда нужно проверить свойство и отрисовать разные компоненты, используют условный рендеринг.

Код в HOC выглядит красиво, но это только промежуточная реализация. Также необходимо объединить функциональность компонентов Gallery и Card. Для этого перейдём к расширению компонента withToggle. Потребуется переместить стейт isToggled в HOC — для этой задачи воспользуемся классом внутри withToggle:

Скопировать кодJSX

*// hocs/with-toggle.js*

import React from 'react';

*// Представим withToggle в виде обычного функционального компонента.*

const withToggle = WrappedComponent => {

return class extends React.Component {

*// Возвращаем экземпляр класса с переданными пропсами.*

constructor(props) {

super(props);

this.state = {

isToggled: false

}

*// Объявляем стейт isToggled.*

}

render() {

return (

<WrappedComponent {...this.props} />

)

*// Вернём переданный компонент.*

}

}

};

export default withToggle;

Постепенно функциональность из компонентов Gallery и Card мигрирует в withToggle компонент. Мы указали state и не забыли про props. Следующий шаг — перенести хендлер onButtonClick, который вызывает изменение isToggled:

Скопировать кодJSX

*// hocs/with-toggle.js*

import React from "react";

*// Представим withToggle в виде обычного функционального компонента.*

const withToggle = (WrappedComponent) => {

return class extends React.Component {

*// Возвращаем экземпляр класса с переданными пропсами.*

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

this.state = {

isToggled: false,

};

*// Объявляем стейт isToggled.*

}

onButtonClick() {

this.setState({

isToggled: !this.state.isToggled,

});

}

render() {

return <WrappedComponent {...this.props} />;

*// Вернём переданный компонент.*

}

};

};

export default withToggle;

Мы перенесли основную функциональность в HOC, а оборачиваемый компонент Card насыщен пропсами. Теперь добавим оборачиваемому компоненту логику, описанную в HOC. Передадим текущее состояние isToggled и хендлер onButtonClick в оборачиваемый компонент:

Скопировать кодJSX

*// hocs/with-toggle.js*

import React from 'react';

*// Представим withToggle в виде обычного функционального компонента.*

const withToggle = WrappedComponent => {

return class extends React.Component {

*// Возвращаем экземпляр класса с переданными пропсами.*

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

this.state = {

isToggled: false

}

*// Объявляем стейт isToggled.*

}

onButtonClick() {

this.setState({

isToggled: !this.state.isToggled,

})

}

render() {

return (

<WrappedComponent

{...this.props}

onButtonClick={this.onButtonClick}

/\* Передадим хендлер onButtonClick \*/

isToggled={this.state.isToggled}

/\* Передадим состояние isToggled \*/

/>

)

*// Вернём переданный компонент.*

}

}

};

export default withToggle;

Теперь HOC withToggle готов: состояние и пропсы передаются оригинальному компоненту. Финальный штрих — редактирование компонента Card. Удалим из него описание состояния, а всё необходимое будем доставать из свойств, переданных HOC. Компонент Card расцветёт и станет намного проще — в нём теперь не определяется состояние и изменение состояния:

Скопировать кодJSX

*// card/card.js*

import React from "react";

class Card extends React.Component {

render() {

return (

<section>

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

<img src={this.props.image} alt={this.props.description} />

<span>{this.props.caption}</span>

<button onClick={this.props.onButtonClick}>

{/\* Обработчик onButtonClick теперь берётся из пропсов. \*/}

{this.props.isToggled ? "Скрыть" : "Показать полное описание"}

</button>

{this.props.isToggled &&

({

/\* Значение isToggled теперь берётся из пропсов. \*/

},

(<div>{this.props.text}</div>))}

</section>

);

}

}

export default Card;

Такие же изменения произойдут и в компоненте Gallery. Его также подключим в App:

Скопировать кодJSX

*// components/app/app.js*

import React from "react";

import styles from "./app.module.css";

import withToggle from "../hoc/with-toggle";

import Card from "../card/card";

import Gallery from "../gallery/gallery";

const cardData = {

*// Здесь хранятся данные для изображений, заголовка и описания для галереи.*

};

const galleryData = {

*// Здесь хранятся данные о карточке номера отеля.*

};

const WithToggleCard = withToggle(Card);

const WithToggleGallery = withToggle(Gallery);

*// Присвоим обёрнутые компоненты Card и Gallery в переменные.*

class App extends React.Component {

render() {

return (

<div className={styles.app}>

<WithToggleGallery {...galleryData} />

<WithToggleCard {...cardData} />

{/\* Передадим cardData и galleryData в HOC \*/}

{/\* Другие компоненты в App \*/}

</div>

);

}

}

HOC полностью готов. Компонент withToggle хорошо подошёл для решения этой задачи, но применение НОС не всегда уместно. Мы реализовали его, чтобы переиспользовать несколько компонентов и облегчить код. Но есть:

## Ситуации, в которых НОС не понадобится:

* Если функциональность одного компонента больше нигде не встречается. HOC станет надстройкой, в которой нет необходимости.
* Если необходимо отдельно настраивать функциональность каждого оборачиваемого компонента. HOC наполнится очень ветвистой логикой, которая не сочетается с декларативным подходом такого компонента.
* Если потребуется передавать много пропсов. Вы познакомились со сложными пропсами в уроке [«Декомпозиция пропсов»](https://praktikum.yandex.ru/trainer/react/lesson/446a3cea-83e2-4281-bae2-3c5634a48adb/task/c3d928b8-94fa-4c0c-b1b3-8fba1297ea90/). Из-за того, что HOC работает с похожими компонентами, количество пропсов в HOC-компоненте может быть очень большим. Это сильно затруднит поддержание кода.

## А вот ситуации, в которых HOC пригодится:

* Если нужно описать поведение для множества компонентов. В коде выше мы сначала нашли схожую функциональность, а после — вынесли её в HOC.
* Если к оборачиваемому компоненту не требуется добавлять собственную логику. Компоненты Card и Gallery с более индивидуальным поведением внутри метода onButtonClick не очень хорошо сочетались бы с HOC.

Эти правила помогут быстро сориентироваться, когда следует создать HOC. Но перед тем, как перейти к итогам урока, ещё чуть-чуть отредактируем код. В withToggle мы объявляем изначальное состояние isToggled как false. Практика, когда мы определяем состояние сразу в компоненте, достаточно редка. Представим, что начальное состояние isToggled будет приходить из компонента выше. Передадим HOC каждого обёрнутого компонента в App начальное состояние initialToggleState, равное false:

Скопировать кодJSX

*// components/app/app.js*

*// Остальной код в компоненте App*

class App extends React.Component {

render() {

return (

<div className={styles.app}>

<WithToggleGallery initialToggleState={false} {...galleryData} />

<WithToggleCard initialToggleState={false} {...cardData} />

{/\* Передадим initialToggleState в HOC \*/}

</div>

);

}

}

*// hocs/with-toggle.js*

const withToggle = (WrappedComponent) => {

return class extends React.Component {

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

this.state = {

isToggled: this.props.initialToggleState,

};

}

*// Остальной код в компоненте withToggle*

};

};

export default withToggle;

С появлением нового свойства HOC образовалась и новая проблема. Свойство initialToggleState будет передано в оборачиваемый компонент Card, но использоваться не будет. Это лишняя логика, которая при высокой вложенности может накапливаться в небольших компонентах. Чтобы не оставлять данные, которые относятся к функциональности самого HOC, разделим пропсы с помощью деструктуризации ... перед тем, как передать оборачиваемому компоненту:

Скопировать кодJSX

*// hocs/with-toggle.js*

const withToggle = WrappedComponent => {

return class extends React.Component {

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

this.state = {

isToggled: this.props.initialToggleState

};

}

onButtonClick() {

this.setState({

isToggled: !this.state.isToggled,

});

}

render() {

const {initialToggleState, ...props} = this.props;

*// Извлечём все необходимые пропсы.*

return (

<WrappedComponent

{...props}

{/\* Передадим оборачиваемому компоненту только его пропсы \*/}

onButtonClick={this.onButtonClick}

isToggled={this.state.isToggled}

/>

);

}

}

};

export default withToggle;

Теперь withToggle контролирует свойства, которые передаются оригинальным компонентам.

Сейчас HOC считается устаревшим способом взаимодействия с приложением. Но это не значит, что в реальной работе вы с ним никогда не столкнётесь и использовать его не нужно. Чуть позже мы расскажем про более современные подходы. А пока заканчиваем работу в веб-студии и переходим к заданиям.

Небольшая подсказка: воспользуйтесь каррированием внутри компонента withFetch.

## Простой пример каррирования выглядит так:

Скопировать кодJSX

function carried(props) {

return function carrying(callback) {

return callback(props);

}

}

*// С использованием стрелочных функций:*

const carried = props => callback => callback(props);

// with-fetch.js

import React from 'react';

import styles from '../../styles.module.css'

const withFetch = props => WrappedComponent => class extends React.Component {

state = {

isLoading: false,

hasError: false,

data: []

};

componentDidMount() {

this.getData();

}

getData() {

this.setState({ ...this.state, hasError: false, isLoading: true });

fetch(props)

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

.then((data) =>

this.setState({ ...this.state, data, isLoading: false })

)

.catch((e) => {

this.setState({ ...this.state, hasError: true, isLoading: false });

});

}

render() {

const { data, isLoading, hasError } = this.state;

return (

<section className={styles.grid}>

{isLoading && "Загрузка..."}

{hasError && "Произошла ошибка"}

{!isLoading &&

!hasError &&

data.length &&

data.map((film, index) => <WrappedComponent key={index} data={film} />)}

</section>

);

}

};

export default withFetch;

// app.js

import React from 'react';

import withFetch from './hocs/with-fetch';

import Film from './film';

import styles from '../styles.module.css'

const WithFetchFilm = withFetch('<https://api.nomoreparties.co/beatfilm-movies>')(Film);

class App extends React.Component {

// eslint-disable-next-line class-methods-use-this

render() {

return (

<div className={styles.app}>

<WithFetchFilm />

</div>

)

}

}

export default App;

# Ref: Доступ к DOM из React

В предыдущих темах вы узнали, что React использует декларативный подход разработки интерфейсов. Другими словами, вы не взаимодействуете с DOM-элементами напрямую, а описываете инструкции по работе с DOM в JSX.

Из этого правила есть исключения, и иногда необходимо наличие императивного мостика к реальному DOM-элементу рендер-метода. React предоставляет такой мостик — механизм рефов (англ. ref от reference, «указатель»). [В официальном руководстве React](https://ru.reactjs.org/docs/refs-and-the-dom.html) приводятся ситуации, в которых использование рефов для работы с DOM-элементами оправдано:

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

## Как работает атрибут ref. Колбэк-рефы

Атрибут ref принимает колбэк-функцию. Функция будет вызвана после монтирования компонента. Создадим простой компонент с атрибутом ref и посмотрим, с каким аргументом будет вызвана колбэк-функция:

Скопировать кодJSX

import React from "react";

import videoItem from "../images/sky.mp4";

import playerStyles from "./video-player.module.css";

class VideoPlayer extends React.Component {

render() {

return (

<div className={playerStyles.player}>

<video

className={playerStyles.video}

src={videoItem}

ref={(element) => console.log(element)}

/>

{/\* Используем атрибут ref и выведем аргумент element \*/}

<button className={playerStyles.button}> ▶️ </button>

</div>

);

}

}

export default VideoPlayer;

*// console.log(element)*

*// <video class="video-player\_video\_\_1fULr" src="/static/media/sky.59e75226.mp4"></video>*

Из результата вызова console.log видно: для HTML-элементов атрибут ref передаёт в качестве аргумента HTML-элемент.

Попробуем указать атрибут ref классовому компоненту в рендер-методе. Для этого вынесем элемент <video /> в отдельный класс:

Скопировать кодJSX

*// video-player/video-player.js*

import React from "react";

import Video from "../video/video";

*// Импортируем компонент Video.*

import videoItem from "../images/sky.mp4";

import playerStyles from "./video-player.module.css";

class VideoPlayer extends React.Component {

render() {

return (

<div className={playerStyles.player}>

<Video ref={(element) => console.log(element)} />

{/\* Используем атрибут ref и выведем аргумент element компонента Video \*/}

<button className={playerStyles.button}> ▶️ </button>

</div>

);

}

}

export default VideoPlayer;

*// console.log(element)*

*// Video {props: {…}, context: {…}, refs: {…}, updater: {…}, \_reactInternals: FiberNode, …}*

Здесь другая ситуация — атрибут ref указывает на экземпляр компонента Video. Это может быть полезно для работы с рефами дочернего компонента внутри родительского. Запомним и вернёмся сюда чуть позже.

А пока возьмём предыдущий пример, где элемент <video /> был в рендер-методе компонента VideoPlayer. Мы не сохраняем указатель, а только выводим его значение в консоль. Сохранить реф, как и любую локальную переменную, можно в обычное поле классового компонента:

Скопировать кодJSX

import React from "react";

import videoItem from "../images/sky.mp4";

import playerStyles from "./video-player.module.css";

class VideoPlayer extends React.Component {

render() {

return (

<div className={playerStyles.player}>

<video

className={playerStyles.video}

src={videoItem}

ref={(element) => (this.videoPlayerRef = element)}

/>

{/\* Используем атрибут ref и присвоим указатель полю videoPlayerRef \*/}

<button className={playerStyles.button}> ▶️ </button>

</div>

);

}

}

export default VideoPlayer;

Теперь в коде есть реф this.videoPlayerRef, который указывает на элемент <video />. Мы можем обратиться к рефу в классовом компоненте. Добавим возможность воспроизводить видео по клику на кнопку:

Скопировать кодJSX

import React from "react";

import videoItem from "../images/sky.mp4";

import playerStyles from "./video-player.module.css";

class VideoPlayer extends React.Component {

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

}

onButtonClick() {

this.videoPlayerRef.play();

}

*// Создадим обработчик onButtonClick, при вызове которого будет воспроизводиться*

*// видео.*

render() {

return (

<div className={playerStyles.player}>

<video

className={playerStyles.video}

src={videoItem}

ref={(element) => (this.videoPlayerRef = element)}

/>

{/\* Используем атрибут ref и присвоим указатель полю videoPlayerRef \*/}

<button className={playerStyles.button} onClick={this.onButtonClick}>

{" "}

▶️{" "}

</button>

</div>

);

}

}

export default VideoPlayer;

Мы научились создавать рефы внутри классовых компонентов. Атрибут ref можно использовать с HTML-элементами и классовыми компонентами в рендер-методе. Мы убедились в том, что ref работает с обычной колбэк-функцией. Эта функция возвращает значение в зависимости от узла, которому присваивается атрибут.

Способ указывать функцию в качестве значения ref, о котором мы рассказали выше, называется колбэк-рефом. Он помогает лучше понять работу указателей. В практической разработке на классовых компонентах можно использовать более короткую запись — метод React.createRef().

## Метод React.createRef()

Этот метод работает достаточно просто: создаёт объект с полем current: null. После монтирования компонента атрибут ref присвоит полю current узел. Перепишем код из примера выше с использованием React.createRef:

Скопировать кодJSX

import React from "react";

import videoItem from "../images/sky.mp4";

import playerStyles from "./video-player.module.css";

class VideoPlayer extends React.Component {

constructor(props) {

super(props);

this.videoPlayerRef = React.createRef();

*// Присвоим полю videoPlayerRef объект с полем current.*

*// Изначально значение current: null.*

this.onButtonClick = this.onButtonClick.bind(this);

}

onButtonClick() {

this.videoPlayerRef.current.play();

*// Обратимся к полю current поля videoPlayerRef - там хранится HTML-элемент.*

}

render() {

return (

<div className={playerStyles.player}>

<video

className={playerStyles.video}

src={videoItem}

ref={this.videoPlayerRef}

/>

{/\* Передадим атрибуту ref наш объект this.videoPlayerRef \*/}

<button className={playerStyles.button} onClick={this.onButtonClick}>

{" "}

▶️{" "}

</button>

</div>

);

}

}

export default VideoPlayer;

Теперь код выглядит проще — в нём больше нет колбэк-функции. Атрибут ref «под капотом» использует простое условие — если передана функция, то в качестве аргумента в неё будет передан узел. В другом случае узел будет присвоен полю current.

Теперь сделаем небольшую паузу и посмотрим на место вызова метода React.createRef:

Скопировать кодJSX

this.videoPlayerRef = React.createRef();

Запись React.createRef() равнозначна { current: null }. Более того, мы можем создать в таком объекте любое поле: { current: null, name: 'RefName' }. Оно не исчезнет после монтирования компонента, а вот поле current после монтирования изменится. Рефы можно мутировать, а их изменение не приведёт к повторной отрисовке компонента. Это может привести к коллизиям, поэтому рефы стоит использовать только при необходимости.

## Добавление ref классовому компоненту

Последнее, что важно попробовать в работе с рефом, — добавить его классовому компоненту. Мы уже делали это выше: выделяли элемент <video /> в отдельный компонент. Сейчас попробуем изменить код — при монтировании компонента VideoPlayer будем автоматически проигрывать видео.

Создадим компонент AutoPlay:

Скопировать кодJSX

import React from "react";

import VideoPlayer from "../video-player/video-player";

class AutoPlay extends React.Component {

constructor(props) {

super(props);

this.videoPlayerRef = React.createRef();

*// Создаём реф*

}

componentDidMount() {

this.videoPlayerRef.current.onButtonClick();

*// Вызываем метод onButtonClick компонента VideoPlayer*

}

render() {

return (

<VideoPlayer ref={this.videoPlayerRef} />

*// Получаем экземпляр компонента*

);

}

}

export default AutoPlay;

В компоненте AutoPlay мы также создаём реф. В рефе компонента AutoPlay будет экземпляр компонента VideoPlayer. После монтирования компонента при обращении к полю current мы получим доступ к методу onButtonClick компонента VideoPlayer. Сам компонент VideoPlayer останется почти без изменений. Поменяется только атрибут autoPlay={true} HTML-элемента <video />. Без него автоматического воспроизведения не получится.

Есть альтернативный способ — передать вложенному компоненту пропс с рефом. Тогда появится возможность получить доступ не к экземпляру компонента, а сразу к HTML-элементу. Это удобно — метод onButtonClick в компоненте VideoPlayer содержит вызов метода play() элемента <video />. Поэтому код выше можно представить так:

Скопировать кодJSX

*// video-player.js*

import React from "react";

import videoItem from "../images/sky.mp4";

import playerStyles from "./video-player.module.css";

class VideoPlayer extends React.Component {

constructor(props) {

super(props);

this.onButtonClick = this.onButtonClick.bind(this);

}

onButtonClick() {

this.props.videoRef.current.play();

}

render() {

return (

<div className={playerStyles.player}>

<video

className={playerStyles.video}

autoplay={true}

src={videoItem}

ref={this.props.videoRef}

/>

{/\* Передадим атрибуту ref пропс videoRef из компонента AutoPlay \*/}

<button className={playerStyles.button} onClick={this.onButtonClick}>

{" "}

▶️{" "}

</button>

</div>

);

}

}

export default VideoPlayer;

*// auto-play.js*

import React from "react";

import VideoPlayer from "../video-player/video-player";

class AutoPlay extends React.Component {

constructor(props) {

super(props);

this.videoPlayerRef = React.createRef();

*// Создаём реф*

}

componentDidMount() {

this.videoPlayerRef.current.play();

*// Вызовем метод play() на элементе <video />*

}

render() {

return (

<VideoPlayer videoRef={this.videoPlayerRef} />

*// Передадим реф videoPlayerRef в качестве пропса videoRef*

);

}

}

export default AutoPlay;

Рефы — не очень сложный механизм, но научиться работать с ним важно. В следующих уроках мы будем неоднократно возвращаться к рефам. А пока — перейдём к практике.

Пример с фокусом инпута и сообщением о подписке

**import React from 'react';**

**import formStyles from './form.module.css';**

**class Form extends React.Component {**

**constructor(props) {**

**super(props);**

**this.state = {**

**isSubmit: false,**

**value: ''**

**};**

**this.emailInput = React.createRef();**

**this.handleFormSubmit = this.handleFormSubmit.bind(this);**

**}**

**handleFormSubmit(e) {**

**e.preventDefault();**

**this.setState = {**

**isSubmit: true,**

**value: this.emailInput.value**

**};**

**}**

**componentDidMount() {**

**this.emailInput.focus();**

**}**

**render() {**

**return (**

**<div className={formStyles.root}>**

**<form**

**className={formStyles.form}**

**onSubmit={this.handleFormSubmit}**

**>**

**<input**

**className={formStyles.input}**

**type='email'**

**placeholder={'Введите свой e-mail'}**

**ref={this.emailInput}**

**/>**

**<button**

**className={formStyles.button}**

**type={'submit'}**

**>**

**Подписаться**

**</button>**

**</form>**

**<p className={formStyles.message}>**

**{**

**this.state.isSubmit && this.emailInput.value &&**

**<>**

**<span>Почта {this.state.value} успешно подписана на рассылку</span>**

**<span>Но это не точно</span>**

**</>**

**}**

**</p>**

**</div>**

**);**

**}**

**}**

**export default Form;**

# Рендеринг вне корневого элемента. React.createPortal

В предыдущем уроке вы узнали о доступе в DOM через рефы. В этом опять поговорим про DOM: разберём, как рендерить дочерние элементы вне структуры DOM родительского компонента.

## Для чего это нужно

До этого урока мы уже показывали, как рендерить дочерние элементы:

Скопировать кодJSX

render() {

*// Монтируем div и рендерим в него дочерние элементы*

return

<div>

{this.props.children}

</div>;

}

Но иногда нужно отрендерить дочерние элементы вне DOM-иерархии родительского компонента. Типовой пример — рендеринг элементов, которые выходят за рамки родительского элемента с заданным стилем overflow: hidden или z-index. Это могут быть всплывающие подсказки, диалоговые окна, уведомления. Тут-то и приходит на помощь React.createPortal, или «портал».

## Использование порталов

Функция ReactDOM.createPortal очень похожа на ReactDOM.render: её первый аргумент — любой React-компонент, который может быть отрендерен; второй аргумент — любой доступный на странице HTML-элемент, контейнер, в который будут отрендерены дочерние элементы. Взгляните на код:

Скопировать кодJSX

const modalRoot = document.getElementById("react-modals");

class Modal extends React.Component {

render() {

const { children, header, onClose } = this.props;

*// Возвращаем ReactDOM.createPortal,*

*// который поместит дочерние элементы в modalRoot*

return ReactDOM.createPortal(

<>

<div className="Modal">

<ModalHeader onClose={onClose}>{header}</ModalHeader>

{children}

</div>

<ModalBackDrop onClose={onClose} />

</>,

modalRoot

);

}

}

В этом примере мы написали простую реализацию модального окна. Контент, который мы обернём в компонент <Modal></Modal>, будет отрендерен внутри элемента с id="react-modals".

Вот так может выглядеть HTML-код, в котором есть несколько «точек входа» для нашего приложения:

Скопировать кодHTML

...

<body>

...

*<!-- Корневой элемент для всего приложения -->*

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

*<!-- Корневой элемент для модальных окон -->*

<div id="react-modals"></div>

*<!-- Корневой элемент для уведомлений на сайте -->*

<div id="react-notifications"></div>

...

</body>

...

Использование модального окна из примера может выглядеть так:

Скопировать кодJSX

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

visible: false,

};

this.handleOpenModal = this.handleOpenModal.bind(this);

this.handleCloseModal = this.handleCloseModal.bind(this);

}

handleOpenModal() {

this.setState({ visible: true });

}

handleCloseModal() {

this.setState({ visible: false });

}

render() {

const modal = (

<Modal header="Внимание!" onClose={this.handleCloseModal}>

<p>Спасибо за внимание!<p>

<p>Открывай меня, если станет скучно :)</p>

</Modal>

);

return (

<div style={{overflow: 'hidden'}}>

<button onClick={this.handleOpenModal}>Открыть модальное окно</button>

{this.state.visible && modal}

</div>

);

}

}

При нажатии на кнопку открывается модальное окно, при этом оно рендерится внутри элемента с id="react-modals", а элемент div cо стилем overflow: hidden в компоненте ему не помеха.

При разработке интерфейсов зачастую требуется реализовать всплывающие окна или подсказки к элементам интерфейса — порталы призваны решать эти задачи.

# События и хендлеры. SyntheticEvent

Часто в приложении требуется реагировать на пользовательский ввод или другие события в браузере. Наверняка вы уже умеете добавлять обработчики событий в JS. В этом уроке расскажем о событиях и их обработке в React — эти процессы несколько различаются.

Вы уже знакомы с интерфейсом Event. Это объект, который состоит из описания события, произошедшего в DOM:

Скопировать кодJSX

const button = document.querySelector('.button');

function handleClick(e) {

console.log('Не дави на меня!');

console.log(e);

}

button.addEventListener('click', handleClick);

*// console.log*

*// Не дави на меня!*

*// MouseEvent {...}*

Но у стандартного интерфейса Event есть проблемы. Одна из них — поведение, которое различается в зависимости от браузера. React по умолчанию предоставляет обёртку над интерфейсом, которая называется синтетическим событием, или SyntheticEvent. Поэтому когда речь идёт про объект событий в React, то мы говорим о синтетическом событии.

## Синтетическое событие: SyntheticEvent

Синтетическое событие кроссбраузерно, так что об особенностях конкретного браузера беспокоиться не придётся. При этом SyntheticEvent наследует все важные методы обычного события, например stopPropagation() и preventDefault(). Важно понимать, что React сохраняет стандартное событие в одно из полей объекта SynthetiсEvent.

Напишем простой компонент и посмотрим на сам объект SyntheticEvent:

Скопировать кодJSX

import React from "react";

class DissatisfiedButton extends React.Component {

handleAgressiveButtonClick(e) {

console.log("Не дави на меня!");

console.log(e);

}

render() {

return (

<>

<button onClick={this.handleAgressiveButtonClick}>

Поиграй со мной!

</button>

</>

);

}

}

export default DissatisfiedButton;

После клика по недовольной (но игривой) кнопке в консоль выведется два сообщения:

* реакция кнопки на клик — «Не дави на меня!»,
* объект SyntheticBaseEvent.

Нас интересует реакция кнопки на клик, поэтому до конца урока будем выяснять, чем же кнопка недовольна. Шутка. Разберёмся с объектом SyntheticBaseEvent.

В отличие от стандартного Event, синтетическое событие не содержит свойств, специфических для определённого браузера. Не будем рассматривать каждое свойство под микроскопом, а остановимся на свойстве nativeEvent.

### Свойство nativeEvent

Это стандартное событие Event. К нему можно обратиться, используя запись e.nativeEvent.

Стандартный Event применяют крайне редко, но знать про его существование важно. Например, один из обработчиков событий может быть добавлен на DOM-элемент императивно с помощью метода .addEventListener и всплытие стандартного события можно остановить с помощью nativeEvent.stopPropagation():

Скопировать кодJSX

import React from "react";

class DissatisfiedButton extends React.Component {

handleAgressiveButtonClick(e) {

console.log("Не дави на меня!");

console.log(e);

e.nativeEvent.stopPropagation();

*// Всплытие обычного события будет остановлено.*

}

handleElementClick(e) {

console.log("Помогите, на кнопку давят!");

*// Это событие сработает, т. к. мы остановили всплытие только .nativeEvent*

}

componentDidMount() {

document.addEventListener("click", (e) => console.log("Событие всплыло"));

}

render() {

return (

<div onClick={this.handleElementClick}>

<button onClick={this.handleAgressiveButtonClick}>

Поиграй со мной!

</button>

</div>

);

}

}

export default DissatisfiedButton;

*// После клика по кнопке в консоле будут два сообщения:*

*// > Не дави на меня!*

*// > Помогите, на кнопку давят!*

Однако остановку всплытия именно стандартного события применяют очень редко.

Теперь сделаем небольшую паузу и подведём итоги:

* React использует собственную обёртку для событий — SyntheticEvent.
* Синтетическое событие похоже на обычное, но кроссбраузерное.
* SyntheticEvent содержит и обычное событие — nativeEvent, которое может пригодиться (но это не точно).

## Прекрасный мир событий в React

Перед погружением в события React важно узнать, что все они всплывают. И даже те события, которые не всплывают в JS, например mouseenter. Это удобно для реализации паттернов делегирования событий.

Есть лишь одно исключение — событие onScroll. В новых версиях React оно не всплывает из-за возможных коллизий в коде.

Все слушатели событий в React начинаются с ключевого слова on и записываются в нотации camelCase. Рассмотрим основные события, с которыми можно столкнуться в повседневной работе.

### События мыши

Это большая группа событий, из которых выделим три наиболее используемых:

* onMouseEnter — мышка наводится на элемент;
* onMouseLeave — мышка перестала наводиться на элемент;
* onClick — на элемент кликнули.

Можно добавлять несколько атрибутов одному элементу, поэтому все три события укажем для одной недовольной кнопки:

Скопировать кодJSX

import React from 'react';

class DissatisfiedButton extends React.Component {

handleAgressiveButtonClick(e) {

console.log('Не дави на меня!');

}

handleAgressiveButtonMouseEnter(e) {

*// Функция будет вызвана при наведении мышки*

console.log('Вы мне солнце заслонили!');

}

handleAgressiveButtonMouseLeave(e) {

*// Функция будет вызвана после того, как пропадёт наведение на элемент*

console.log('Ну вот, теперь слишком жарко!');

}

render() {

return (

<>

<button

onClick={this.handleAgressiveButtonClick}

onMouseEnter={this.handleAgressiveButtonMouseEnter}

onMouseLeave={this.handleAgressiveButtonMouseLeave}

>

{/\* Добавляем сразу несколько обработчиков \*/}

Поиграй со мной!

</button>

</>

);

}

}

export default DissatisfiedButton;

### События формы

В группу событий формы входят отправка, сброс и изменение формы:

* onChange (или onInput) — в поле ввода происходит изменение;
* onSubmit — отправка формы;
* onInvalid — при отправке формы есть некорректно заполненные поля;
* onReset — сброс формы.

События этой группы знакомы вам из JS. Но повторим важный момент: событие onChange всплывает.

### События фокуса

Состоят из:

* onFocus — перемещение фокуса на элемент;
* onBlur — пропадание фокуса с элемента.

Аналогичны событиям focusin и focusout.

### События клавиатуры

Включают в себя:

* onKeyDown — срабатывает при нажатии клавиши;
* onKeyPress — срабатывает при нажатии клавиши, показывает введённый символ;
* onKeyUp — срабатывает, когда клавишу отпустили.

Событие onKeyPress устарело, поэтому мы рекомендуем использовать событие onKeyDown.

Позже вам потребуются и другие события. Полный список можно посмотреть [в официальной документации React](https://ru.reactjs.org/docs/events.html#supported-events).

## Обработка событий

Добавлять обработчики событий в классовых компонентах можно несколькими способами:

* объявить обработчик как метод класса,
* использовать синтаксис общедоступных полей класса,
* применить стрелочную функцию в колбэке слушателя события.

В коде выше мы уже использовали первый способ с указанием в качестве обработчика событий один из методов класса:

Скопировать кодJSX

import React from "react";

class DissatisfiedButton extends React.Component {

handleAgressiveButtonClick(e) {

console.log("Не дави на меня!");

}

*// Объявляем метод класса handleAgressiveButtonClick*

handleAgressiveButtonMouseEnter(e) {

console.log("Вы мне солнце заслонили!");

}

handleAgressiveButtonMouseLeave(e) {

console.log("Ну вот, теперь слишком жарко!");

}

render() {

return (

<>

<button

onClick={this.handleAgressiveButtonClick}

onMouseEnter={this.handleAgressiveButtonMouseEnter}

onMouseLeave={this.handleAgressiveButtonMouseLeave}

>

{/\* Ссылаемся на метод handleAgressiveButtonClick \*/}

Поиграй со мной!

</button>

</>

);

}

}

export default DissatisfiedButton;

В данном случае мы ссылаемся на обработчик handleAgressiveButtonClick. Другими словами — не указываем () в слушателе onClick. Поэтому при использовании контекста this внутри самого обработчика handleAgressiveButtonClick он будет не определён:

Скопировать кодJSX

import React from "react";

class DissatisfiedButton extends React.Component {

handleAgressiveButtonClick(e) {

console.log(this);

console.log("Не дави на меня!");

*// При вызове этого обработчика this равен undefined;*

}

handleAgressiveButtonMouseEnter(e) {

console.log("Вы мне солнце заслонили!");

}

handleAgressiveButtonMouseLeave(e) {

console.log("Ну вот, теперь слишком жарко!");

}

render() {

return (

<>

<button

onClick={this.handleAgressiveButtonClick}

onMouseEnter={this.handleAgressiveButtonMouseEnter}

onMouseLeave={this.handleAgressiveButtonMouseLeave}

>

Поиграй со мной!

</button>

</>

);

}

}

export default DissatisfiedButton;

Поведение, при котором значение контекста равно undefined, — нормальное для методов класса. По умолчанию методы класса не привязаны к контексту. Для решения этой задачи обычно используют явную привязку контекста — метод .bind(). Он позволяет вызвать функцию с переданным в качестве параметра контекстом. Привязывают контекст, как правило, в теле конструктора:

Скопировать кодJSX

import React from "react";

class DissatisfiedButton extends React.Component {

constructor(props) {

super(props);

this.handleAgressiveButtonClick =

this.handleAgressiveButtonClick.bind(this);

*// Говорим методу handleAgressiveButtonClick о том,*

*// какой контекст использовать в момент вызова.*

}

handleAgressiveButtonClick(e) {

console.log(this);

console.log("Не дави на меня!");

*// При вызове этого обработчика this равен экземпляру компонента.*

}

handleAgressiveButtonMouseEnter(e) {

console.log("Вы мне солнце заслонили!");

}

handleAgressiveButtonMouseLeave(e) {

console.log("Ну вот, теперь слишком жарко!");

}

render() {

return (

<>

<button

onClick={this.handleAgressiveButtonClick}

onMouseEnter={this.handleAgressiveButtonMouseEnter}

onMouseLeave={this.handleAgressiveButtonMouseLeave}

>

Поиграй со мной!

</button>

</>

);

}

}

export default DissatisfiedButton;

Объявление хендлеров в качестве методов класса — самый распространённый способ обработки событий. Но в нём есть единственный недостаток — дополнительный код с явным указанием контекста.

Существует более удобный способ — синтаксис общедоступных полей класса. Несмотря на то, что это относительно новый синтаксис, Babel его успешно транспилирует. В общедоступных полях класса контекст определён сразу и использовать .bind() не потребуется:

Скопировать кодJSX

import React from "react";

class DissatisfiedButton extends React.Component {

constructor(props) {

super(props);

this.handleAgressiveButtonClick =

this.handleAgressiveButtonClick.bind(this);

}

handleAgressiveButtonClick(e) {

console.log(this);

console.log("Не дави на меня!");

}

handleAgressiveButtonMouseEnter = (e) => {

*// Присвоим handleAgressiveButtonMouseEnter значение в виде стрелочной функции.*

console.log(this);

*// При вызове этого обработчика this равен экземпляру компонента.*

console.log("Вы мне солнце заслонили!");

};

handleAgressiveButtonMouseLeave(e) {

console.log("Ну вот, теперь слишком жарко!");

}

render() {

return (

<>

<button

onClick={this.handleAgressiveButtonClick}

onMouseEnter={this.handleAgressiveButtonMouseEnter}

onMouseLeave={this.handleAgressiveButtonMouseLeave}

>

Поиграй со мной!

</button>

</>

);

}

}

export default DissatisfiedButton;

В примере с хендлером handleAgressiveButtonMouseEnter мы создаём поле класса с именем handleAgressiveButtonMouseEnter и значением в виде хендлера событий. Хендлер записан через стрелочную функцию, поэтому контекст в функции будет определён.

Последний способ работы с хендлерами событий — использование стрелочной функции в колбэке слушателя события. В этом случае контекст будет привязан к текущему экземпляру компонента:

Скопировать кодJSX

import React from 'react';

class DissatisfiedButton extends React.Component {

constructor(props) {

super(props);

this.handleAgressiveButtonClick = this.handleAgressiveButtonClick.bind(this)

}

handleAgressiveButtonClick(e) {

console.log(this);

console.log('Не дави на меня!');

}

handleAgressiveButtonMouseEnter = (e) => {

console.log(this);

console.log('Вы мне солнце заслонили!');

}

handleAgressiveButtonMouseLeave(e) {

console.log(this);

*// Несмотря на использование метода класса,*

*// this будет равен текущему экземпляру компонента*

console.log('Ну вот, теперь слишком жарко!');

}

render() {

return (

<div>

<button

onClick={this.handleAgressiveButtonClick}

onMouseEnter={this.handleAgressiveButtonMouseEnter}

onMouseLeave={e => this.handleAgressiveButtonMouseLeave(e)}

>

{/\* В качестве значения onMouseLeave указываем стрелочную функцию \*/}

Поиграй со мной!

</button>

</>

);

}

}

export default DissatisfiedButton;

Но при использовании колбэков возникает проблема: с каждым рендером создаётся новый экземпляр функции. При передаче такого колбэка дочернему компоненту может произойти повторная отрисовка компонента — поведение, которое разработчик умышленно не закладывает в компонент. Поэтому мы рекомендуем использовать первые два способа обработки событий: метод .bind() для явной привязки контекста либо общедоступные поля класса.

С классовыми компонентами почти разобрались. Рассмотрим,

### как добавлять слушатели событий на рефы.

Это можно сделать с помощью метода .addEventListener, знакомого вам из JS.

Сделаем рядом с недовольной кнопкой соседа с атрибутом ref. А при монтировании компонента добавим слушатель событий:

Скопировать кодJSX

import React from "react";

class DissatisfiedButton extends React.Component {

constructor(props) {

super(props);

this.buttonElement = React.createRef();

*// Создадим реф buttonElement*

this.handleAgressiveButtonClick =

this.handleAgressiveButtonClick.bind(this);

}

handleAgressiveButtonClick(e) {

console.log(this);

console.log("Не дави на меня!");

}

handleAgressiveButtonMouseEnter = (e) => {

console.log(this);

console.log("Вы мне солнце заслонили!");

};

handleAgressiveButtonMouseLeave(e) {

console.log(this);

console.log("Ну вот, теперь слишком жарко!");

}

componentDidMount() {

this.buttonElement.current.addEventListener(

"click",

this.handleAgressiveButtonClick

);

*// После монтирования компонента добавим слушатель на реф.*

}

render() {

return (

<>

<button

onClick={this.handleAgressiveButtonClick}

onMouseEnter={this.handleAgressiveButtonMouseEnter}

onMouseLeave={(e) => this.handleAgressiveButtonMouseLeave(e)}

>

Поиграй со мной!

</button>

<button ref={this.buttonElement}>

{/\* Укажем поле buttonElement в качестве значения ref \*/}И со мной

поиграй!

</button>

</>

);

}

}

export default DissatisfiedButton;

К сожалению, обе кнопки оказались недовольными. Зато мы разобрались с добавлением слушателей на рефы.

Теперь разберём,

### как работать с хендлерами в функциональных компонентах.

Это проще, потому что в большинстве случаев о потере контекста беспокоиться не приходится — синтаксис класса в функциональном компоненте отсутствует. При работе с функциональными компонентами мы опираемся не на контекст, а на области видимости внутри компонента. Поэтому код выше мы можем представить обычным функциональным компонентом:

Скопировать кодJSX

import React from "react";

function DissatisfiedButton() {

function handleAgressiveButtonClick() {

console.log("Не дави на меня!");

}

function handleAgressiveButtonMouseEnter() {

console.log("Вы мне солнце заслонили!");

}

function handleAgressiveButtonMouseLeave() {

console.log("Ну вот, теперь слишком жарко!");

}

return (

<button

onClick={handleAgressiveButtonClick}

onMouseEnter={handleAgressiveButtonMouseEnter}

onMouseLeave={handleAgressiveButtonMouseLeave}

>

Поиграй со мной!

</button>

);

}

export default DissatisfiedButton;

Мы разобрались с тем, как устанавливать обработчики событий и узнали, что такое синтетическое событие в React. Теперь перейдём к заданиям и закрепим на практике обработку событий. В следующем уроке погрузимся в JS-механизмы — всплытие и захват.

import React from 'react';

import lootboxerStyles from './lootboxer.module.css';

import lockImage from '../../images/lock.svg';

import checkImage from '../../images/check.svg';

import retryImage from '../../images/retry.svg';

import crumbsImage from '../../images/crumbs.png';

import drinkImage from '../../images/drink.png';

import monitorImage from '../../images/monitor.png';

import mouseImage from '../../images/mouse.png';

import videocardImage from '../../images/videocard.png';

import wheelImage from '../../images/wheel.png';

const itemsData = [

{

name: 'GeekForce RTX 9000',

description: 'Да-да, та самая видеокарта. Теперь вам не нужно продавать почку, чтобы поиграть в самые новые игры.',

image: videocardImage

},

{

name: 'Aimbot MX420',

description: 'Топовая игровая мышь с гиперчувствительным сенсором и эргономичным дизайном. Главное, чтобы не забанили за допинг.',

image: mouseImage

},

{

name: 'SpeedyMonster',

description: 'Энергетический напиток, после которого можно играть в режиме нон-стоп всю неделю. Не злоупотребляйте.',

image: drinkImage

},

];

const getRandomElement = arr => arr[Math.floor(Math.random() \* arr.length)];

class Lootboxer extends React.Component {

state = {

currentItem: null,

isHovered: false

};

handleNewTryClick = () => {

this.setState({

...this.state,

currentItem: getRandomElement(itemsData)

});

};

handleItemHover = () => {

this.setState({

...this.state,

isHovered: !this.state.isHovered

});

};

handleTakeClick = () => {

this.setState({

...this.state,

currentItem: null

});

};

render() {

return (

<section className={lootboxerStyles.root}>

<p className={lootboxerStyles.description}>

{

this.state.currentItem && this.state.isHovered &&

<span>{this.state.currentItem.description}</span>

}

</p>

<div

className={lootboxerStyles.card}

onMouseEnter={this.handleItemHover}

onMouseLeave={this.handleItemHover}

>

{

!this.state.currentItem

? <img src={lockImage} alt='Lock' />

: <img src={this.state.currentItem.image} alt='Item' />

}

</div>

<h2

className={lootboxerStyles.title}

onMouseEnter={this.handleItemHover}

onMouseLeave={this.handleItemHover}

>

{

this.state.currentItem &&

this.state.currentItem.name

}

</h2>

<div className={lootboxerStyles.controls}>

{

!this.state.currentItem ?

<button

className={lootboxerStyles.button}

onClick={this.handleNewTryClick}

>

Поехали

</button>

:

<>

<button

className={lootboxerStyles.button}

onClick={this.handleTakeClick}

>

<img src={checkImage} alt='Check' />

Забираю

</button>

<button

className={lootboxerStyles.button}

onClick={this.handleNewTryClick}

>

<img src={retryImage} alt='Retry' />

Ещё разок

</button>

</>

}

</div>

</section>

);

}

}

export default Lootboxer;

# События и их обработка. Регистрация событий на фазе захвата. event.currentTarget

В предыдущем уроке мы рассказали о том, что почти все события в React всплывают. Также на фазе всплытия происходит и регистрация событий в React. Убедимся в этом с помощью очередного эксперимента над недовольной кнопкой:

Скопировать кодJSX

import React from "react";

function DissatisfiedButton() {

function handleClickBubble() {

console.log("Поймаю событие после handleAgressiveButtonClick!");

}

function handleAgressiveButtonClick() {

console.log("Не дави на меня!");

}

return (

<div onClick={handleClickBubble}>

<button onClick={handleAgressiveButtonClick}>Поиграй со мной!</button>

</div>

);

}

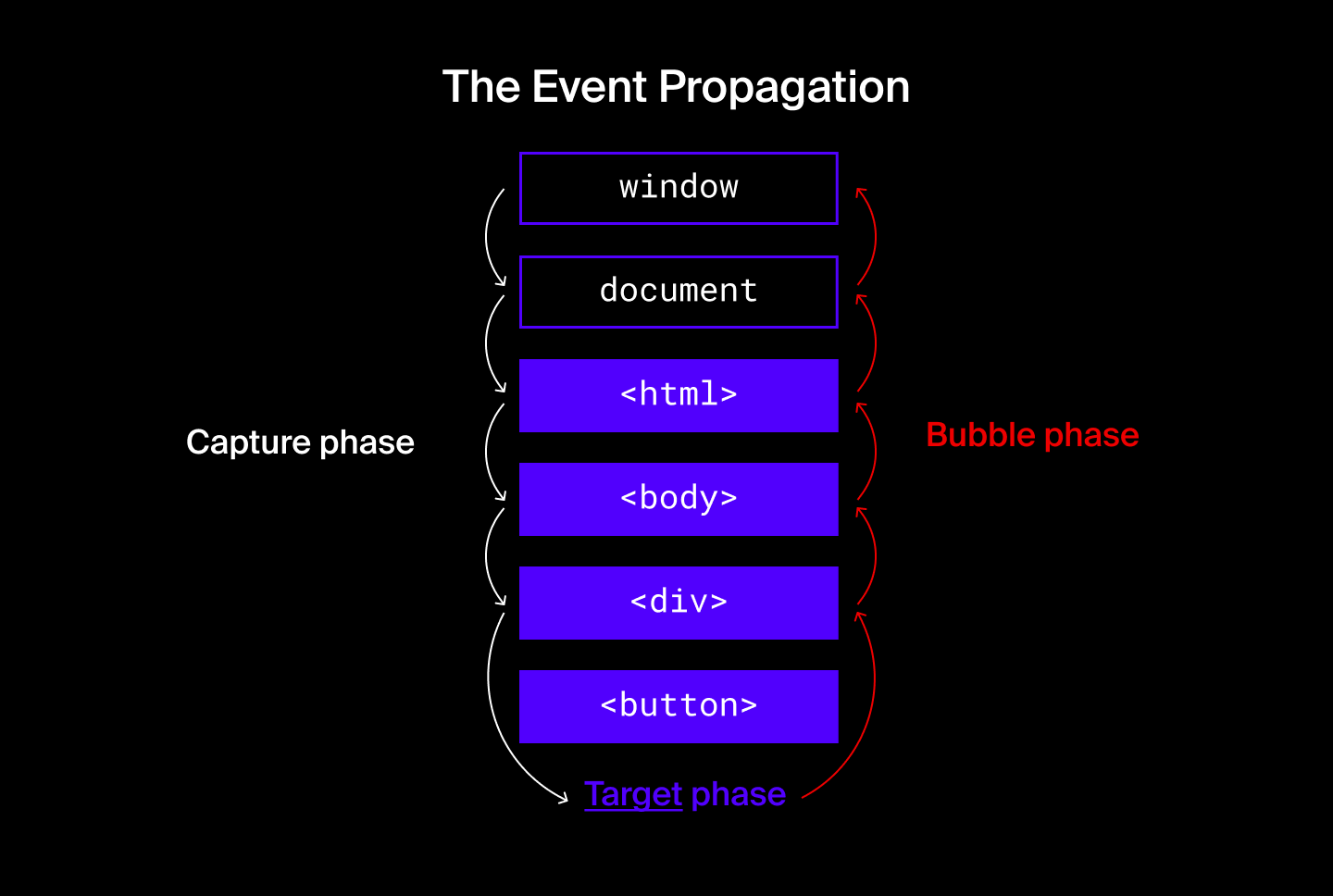
export default DissatisfiedButton;

*// После клика по кнопке в консоли будут два сообщения:*

*// > Не дави на меня!*

*// > Поймаю событие после handleAgressiveButtonClick!*

Такая последовательность появления сообщений в консоли возникла из-за того, что событие onClick на элементе div было зарегистрировано на фазе всплытия. Всплытие — заключительный этап прохождения события:



Но ещё в React возможно регистрировать события не на фазе всплытия, а на фазе захвата. Для этого нужно просто на целевом элементе к названию нужного события добавить слово Capture:

Скопировать кодJSX

import React from "react";

function DissatisfiedButton() {

function handleCaptureClick() {

console.log("Что-то мне подсказывает, что на кнопку сейчас надавят...");

}

*// Добавим хендлер для события, зарегистрированного на фазе захвата.*

function handleClickBubble() {

console.log("Поймаю событие после handleAgressiveButtonClick!");

}

function handleAgressiveButtonClick() {

console.log("Не дави на меня!");

}

return (

<div onClick={handleClickBubble} onClickCapture={handleCaptureClick}>

{/\* Добавим обработчик onClickCapture \*/}

<button onClick={handleAgressiveButtonClick}>Поиграй со мной!</button>

</div>

);

}

export default DissatisfiedButton;

*// После клика по кнопке в консоли будет три сообщения:*

*// > Что-то мне подсказывает, что на кнопку сейчас надавят...*

*// > Не дави на меня!*

*// > Поймаю событие после handleAgressiveButtonClick!*

В этом случае хендлер handleCaptureClick сработает раньше остальных, потому что обработка происходит на фазе захвата в обработчике onClickCapture у элемента div.

## Метод stopPropagation() и preventDefault()

Стандартное поведение всплытия и захвата способно приводить к коллизиям: в элемент с обработчиком событий может быть вложен другой элемент с таким же обработчиком. Для предотвращения повторной обработки события на фазе захвата или всплытия существует знакомый вам из JS метод — stopPropagation(). Этот метод останавливает процессы всплытия на элементе:

Скопировать кодJSX

import React from "react";

function DissatisfiedButton() {

function handleCaptureClick() {

console.log("Что-то мне подсказывает, что на кнопку сейчас надавят...");

}

function handleClickBubble() {

console.log("Поймаю событие после handleAgressiveButtonClick!");

}

function handleAgressiveButtonClick(e) {

e.stopPropagation();

*// Останавливаем фазы всплытия и захвата на данном элементе;*

console.log("Не дави на меня!");

}

return (

<div onClick={handleClickBubble} onClickCapture={handleCaptureClick}>

<button onClick={handleAgressiveButtonClick}>Поиграй со мной!</button>

</div>

);

}

export default DissatisfiedButton;

*// После клика по кнопке в консоли будет два сообщения:*

*// > Что-то мне подсказывает, что на кнопку сейчас надавят...*

*// > Не дави на меня!*

Иногда метода stopPropagation() недостаточно — за некоторыми элементами закреплено событие по умолчанию. Например, при отправке формы или клике по активному чекбоксу происходят стандартные браузерные события. Для их отмены используют метод preventDefault().

В React оба метода, stopPropagation() и preventDefault(), выполняют такую же задачу, как и в обычном JS.

## Свойства Event.target и Event.currentTarget

В завершение разберём два свойства событий в React — currentTarget и target. Свойство target объекта Event указывает на элемент, который вызывает обработку события. Проверим это и вложим в недовольную кнопку элемент span, убрав с неё все слушатели событий:

Скопировать кодJSX

import React from "react";

function DissatisfiedButton() {

function handleAgressiveButtonClick(e) {

console.log(e.target);

console.log("Не дави на меня!");

}

return (

<div onClick={handleAgressiveButtonClick}>

<button>

<span>Поиграй со мной!</span>

</button>

</div>

);

}

export default DissatisfiedButton;

Значение свойства target будет изменяться:

* при клике на span с текстом e.target будет равен span,
* при клике на button значение e.target будет равно button,
* при клике на div значение e.target будет равно div.

Свойство currentTarget, наоборот, всегда указывает на элемент с добавленным слушателем событий. Вне зависимости от того, кликнули мы на button, span или сам div, значением e.currentTarget всегда будет элемент div.

В этом и предыдущем уроках мы так и не выяснили, чем недовольна кнопка. Зато разобрались с обработкой событий в React и освежили базовые знания событий в JS. А в следующем уроке разберёмся с чистыми компонентами и мемоизацией.

# Чистые компоненты. React.PureComponent и React.memo

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 React, чтобы писать оптимальный код.

## Проблема оптимизации: цепная реакция рендеров

Как вы знаете из урока про [«Жизненный цикл классовых компонентов»](https://practicum.yandex.ru/trainer/react/lesson/f40cbd35-fdfb-468e-a517-2495c4e039e4/task/7a8db21e-fd46-4da8-aa91-a85fcef8152b/), 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 пример со списком чатов. Допустим, вы хотите, чтобы при наведении курсора 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 и реагирует на события мыши, чтобы отображать дополнительные элементы:

Скопировать кодJSX

class ChatWithTools extends React.Component {

state = {

shouldShowTools: false,

};

handleMouseEnter = () => {

this.setState({ shouldShowTools: true });

};

handleMouseLeave = () => {

this.setState({ shouldShowTools: false });

};

render() {

return (

<div

className="chat-with-tools"

onMouseEnter={this.handleMouseEnter}

onMouseLeave={this.handleMouseLeave}

>

<Chat {...this.props} />

{this.state.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 {...this.props}/>. С помощью такой конструкции можно копировать в дочерний компонент сразу все его 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>

);

};

Никаких изменений не произошло бы и рендер не вызвался бы напрасно, если бы не случайное число. 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 также был бы вызван напрасно.

Чтобы этого не происходило, нужно использовать специальный тип компонентов, отрисовка 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: React.memo

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: React.PureComponent

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>

);

};

}

React.PureComponent очень похож на React.Component. Но есть небольшое отличие: React.Component не реализует метод жизненного цикла shouldComponentUpdate() по умолчанию, а React.PureComponent реализует его поверхностным сравнением пропсов и состояния.

Для решения большинства задач, связанных с производительностью, подойдёт React.PureComponent. А для сложных сценариев, например когда пропс — сложный тип данных и требуется глубокое сравнение, мы рекомендуем использовать метод жизненного цикла shouldComponentUpdate() внутри React.Component.

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 закешированный результат. Если же пропсы изменились, вызывается рендер, после чего сохраняется новый кеш 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 значением при помощи оператора ===.

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.

**Введение в хуки**

Если вы не в восторге от объектно-ориентированного программирования, рекомендуем присмотреться к «хукам». Хуки появились в React сравнительно недавно, и главная их цель — привнести всю мощь классовых компонентов в мир функциональных.

Функциональный и классовый подходы сильно отличаются с точки зрения внутреннего устройства, но в целом призваны решать одни и те же задачи. С появлением хуков они стали во многом равноценными и взаимозаменяемыми. С хуками вы сможете использовать тот подход, который вам больше нравится.

Хук — специальная функция, которую можно вызывать в теле функционального компонента, чтобы осуществить доступ к его внутренним ресурсам. Библиотека React обладает готовым набором хуков, а также поддерживает сторонние, которые вы можете написать самостоятельно.

В предыдущих уроках вы изучили два основных преимущества классовых компонентов: наличие [внутреннего состояния](https://practicum.yandex.ru/trainer/react/lesson/c401de6c-42d0-4b87-aef4-1806624f11c2/task/64a387ac-4bf2-410b-92a5-e28f91c0d9cb) и управление эффектами в рамках [жизненного цикла](https://practicum.yandex.ru/trainer/react/lesson/f40cbd35-fdfb-468e-a517-2495c4e039e4/task/d351639a-7297-42a4-80f0-4f5fb482d9e8). Для решения обеих задач в функциональных компонентах можно использовать соответствующие типы хуков. Их и разберём в этой теме.

# Хуки. useState

Вы уже знакомы с внутренним состоянием в классовых компонентах и способами его применения. В этом уроке изучим реализацию состояния в функциональных компонентах. В версии React 16.8 разработчики добавили API хуков. Один из таких хуков — useState().

Вспомним пример с переключением темы из светлой в тёмную и наоборот. Сравним две реализации: с классовым компонентом и с функциональным компонентом и хуком состояния React.useState.

Скопировать кодJSX

*// Классовый компонент*

class ThemeToggle extends React.Component {

*// Начальное состояние компонента*

state = { theme: "светлая" };

*// Обработчик клика на кнопку*

toggleTheme = () => {

*// Вызов метода для изменения состояния*

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>

);

}

}

Скопировать кодJSX

*// Функциональный компонент*

function ThemeToggle() {

*// Начальное состояние компонента и функция для изменения состояния*

const [theme, setTheme] = React.useState("светлая");

*// Обработчик клика на кнопку*

const toggleTheme = () => {

*// Вызов функции для изменения состояния*

setTheme(theme === "светлая" ? "тёмная" : "светлая");

};

return (

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

<p>Включена {theme} тема</p>

<button onClick={toggleTheme}>

{theme === "светлая" ? "🌚" : "🌞"}

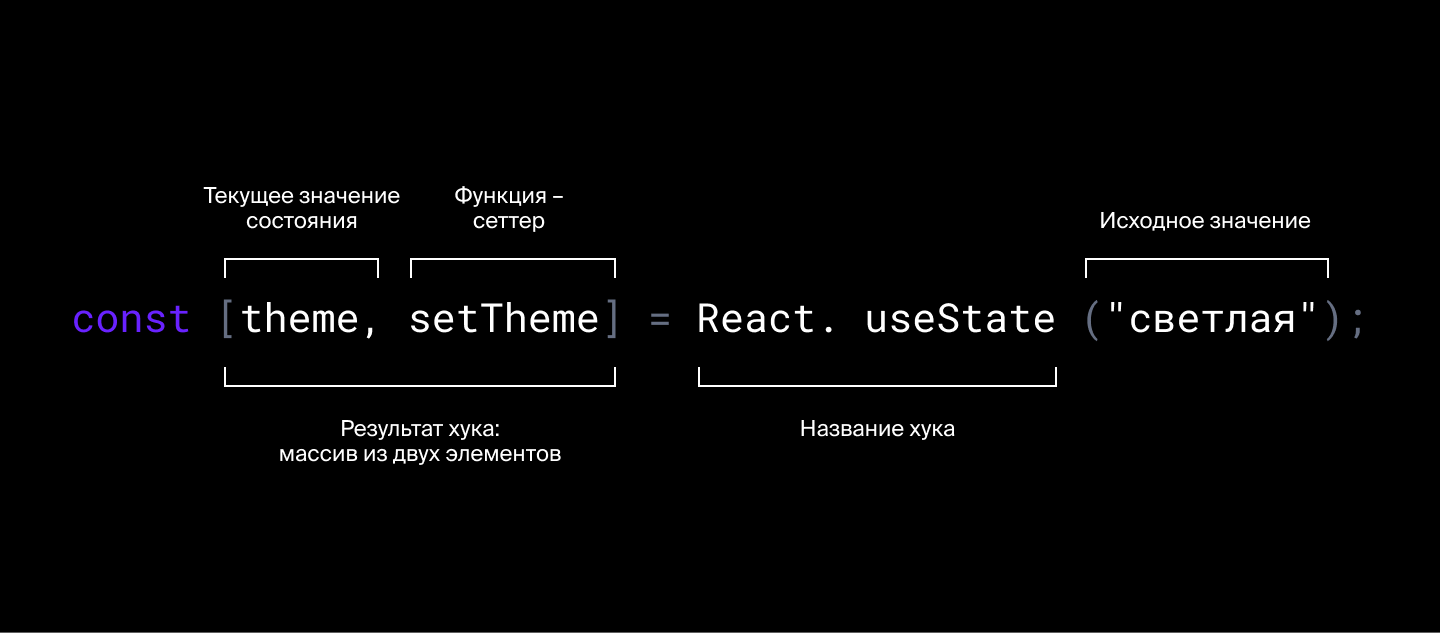
</button>

</div>

);

}

Код стал лаконичнее: например, this больше не используется, а встроенное поле this.state и метод this.setState заменились одной строкой с вызовом хука.



Магия хуков: объявление двух локальных переменных с помощью деструктурирующего присваивания из возвращаемого массива

Хук React.useState принимает единственный аргумент — начальное значение какой-либо переменной состояния, и сохраняет его в своём внутреннем хранилище. Взамен он возвращает пару сущностей в виде массива: текущее значение (которое при первом вызове совпадает с начальным) и функцию-сеттер, с помощью которой его можно изменять.

Эта функция действует аналогично this.setState, но в отличие от последней она управляет не всем набором переменных состояния компонента, а лишь одной из них — в данном случае theme.

Когда произойдёт вызов функции setTheme с новым значением, React обновит это значение в своём хранилище и инициирует перерисовку компонента, то есть вызовет саму функцию-компонент (в нашем случае ThemeToggle). При этом повторный вызов хука React.useState вернёт уже другой массив: первый элемент в нём будет содержать новое значение, а функция-сеттер останется прежней.

## Деструктурирующее присваивание

Обратите внимание, что хук «не знает», какой именно переменной состояния он управляет. Он лишь принимает на вход некое значение и возвращает пару: то же значение и функцию, которая может его менять. Сами по себе элементы массива не имеют имён, но с помощью деструктуризации мы создаём на их основе две переменные и присваиваем им удобные имена. Работает это так:

Скопировать кодJSX

const nameArr = ['Юрий', 'Гагарин'];

console.log(nameArr[0]); *// Юрий*

console.log(nameArr[1]); *// Гагарин*

const [name, surname] = nameArr;

console.log(name) *// Юрий*

console.log(surname) *// Гагарин*

Таким образом, хук React.useState можно использовать в компонентах несколько раз для создания независящих друг от друга переменных внутреннего состояния. При этом значения могут иметь любой тип данных, включая массивы и объекты:

Скопировать кодJSX

const [rating, setRating] = React.useState(0); // Число

const [title, setTitle] = React.useState('Рассказ о том, как я щуку поймал руками'); // Строка

const [isBlocked, setIsBlocked] = React.useState(false); // true или false

const [notes, setNotes] = React.useState(['Пока нет заметок', 'Хотя... Их уже целых две']); // Массив строк

const [user, setUser] = React.useState({ id: 1, name: 'Полина' }); // Объект

## Неизменный порядок хуков между вызовами компонента

Так как React не знает, какой именно переменной состояния управляет каждый из хуков, внутри себя он использует порядковый номер вызова каждого хука, чтобы ассоциировать с ним данные в хранилище.

Это указывает на важное правило: порядок хуков не должен меняться между вызовами компонента. То есть хуки не могут находиться внутри условных блоков, циклов и других подобных конструкций:

Скопировать кодJSX

const [rating, setRating] = React.useState(0);

// Это условие может выполниться, а может и не выполниться:

if (isRaining) {

const [isBlocked, setIsBlocked] = React.useState(false);

}

// Из-за if этот хук может оказаться либо вторым, либо третьим — так нельзя!

const [notes, setNotes] = React.useState(['Пока нет заметок']);

Из этого правила вытекает и то, что хуки нельзя использовать в других функциях, обработчиках или модулях — вызывайте их только из основной функции компонента. Есть одно исключение: ваши собственные хуки, но о них мы поговорим чуть позже.

## Мутации сложных объектов

Если для стейта вы используете массивы или объекты с полями, избегайте их изменений напрямую (мутаций) — всегда передавайте в сеттер новый объект. Для этого удобно использовать оператор spread ..., который создаёт изменённую копию исходного объекта:

Скопировать кодJSX

const [array, setArray] = React.useState(['Раз', 'Два', 'Три']);

// Так нельзя!

array.push('Четыре');

setArray(array);

// Нужно делать так:

setArray([...array, 'Четыре']);

Скопировать кодJSX

const [object, setObject] = React.useState({ name: 'James', surname: 'Wilson' });

// Так нельзя!

object.name = 'Gregory';

setObject(object);

// Нужно делать так:

setObject({

...object,

name: 'Gregory',

});

Дело в том, что React сравнивает старое и новое значение с помощью оператора ===, а значит, мы не можем передавать исходный объект, иначе проверка всегда будет возвращать true.

function GoodDeeds() {

const [deeds, setDeeds] = React.useState([])

function handleAddTask(e) {

const input = e.target.previousSibling;

setDeeds([...deeds, input.value])

input.value = '';

}

return (

<>

<h3>Мои хорошие поступки</h3>

<input type="text" placeholder="Поступок" />

<button onClick={handleAddTask}>Добавить!</button>

<ol>

{deeds.map((deed, i) => (

<li key={i}>{deed}</li>

))}

</ol>

</>

);

}

ReactDOM.render((

<GoodDeeds />

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

# Хуки. Побочные эффекты с помощью React.useEffect

В предыдущем уроке вы узнали, как работать с внутренним состоянием в функциональных компонентах. В этом — ещё больше углубимся в возможности хуков и расскажем, как выполнять побочные эффекты в функциональных компонентах.

## Для чего нужен хук useEffect

Из урока про [жизненный цикл](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/f40cbd35-fdfb-468e-a517-2495c4e039e4/) вы уже знаете, как выполнять побочные эффекты в классовых компонентах. Хук useEffect позволяет выполнять побочные эффекты (действия) в функциональных React-компонентах.

Есть два распространённых вида побочных эффектов в React-компонентах:

1. Эффекты, которые требуют сброса. Например, отписка от слушателя событий или закрытие соединений с сервером.
2. Эффекты, которые не требуют сброса. Например, запрос данных с сервера, логирование, изменение DOM напрямую.

Мы ещё рассмотрим их подробнее, а пока разберём, как используют хук useEffect.

## Первый аргумент хука React.useEffect

Первый аргумент хука useEffect — функция, код которой выполняется при монтировании компонента, изменении его пропсов или состояния.

Сценарий использования хука зависит от задачи, которую мы хотим решить. Вот так выглядит использование хука useEffect в самом простом виде:

Скопировать кодJSX

import React, { useEffect } from 'react';

function Hi = ({name}) => {

useEffect(() => {

console.log(`Привет, ${name}!`);

})

return <h1>Я хук <code>useEffect</code>, посмотри в консоль :)</h1>;

}

function App = () => {

return <Hi name={'Рудольф'} />

}

В этом примере компонент Hi при монтировании выведет в консоль сообщение "Привет, Рудольф!". Если у компонента изменится props, хук будет вызван повторно. В примере хук useEffect ведёт себя как совокупность двух методов жизненного цикла: componentDidMount и componentDidUpdate.

Как и любой хук из библиотеки React, useEffect можно использовать многократно. Взгляните на пример:

Скопировать кодJSX

const MultiEffectComponent = () => {

useEffect(()=>{

console.log('Эффект 1')

})

useEffect(()=>{

console.log('Эффект 2')

})

useEffect(()=>{

console.log('Эффект 3')

})

return (

*//...*

);

}

Код этого компонента выведет в консоль три сообщения в том порядке, в котором они были написаны. А вот пример посложнее:

Скопировать кодJSX

const BrowserTabRating = () => {

const [rating, setRating] = useState(0);

const plus = () => {

setRating(rating + 1);

}

const minus = () => {

setRating(rating - 1);

}

useEffect(() => {

document.title = `Рейтинг: ${rating}`;

});

return (

<>

<button onClick={minus}>-1</button>

<button onClick={plus}>+1</button>

</>

)

}

Компонент BrowserTabRating рендерит две кнопки: нажатие изменяет значение rating, а благодаря хуку useEffect меняется document.title. Это пример побочного эффекта — мы изменяем DOM напрямую. Каждый раз, когда состояние или пропс изменяются, хук вызывает код, переданный ему в первом аргументе. Постарайтесь избегать такого использования хука, это может вызвать проблемы с производительностью: если на повторный рендеринг компонента влияет множество факторов, например большое количество пропсов и сложное внутреннее состояние, то хук будет вызывать побочный эффект на каждый рендер.

Но как быть, если в компоненте нужно выполнить побочный эффект только при монтировании и/или обновлении определённых свойств? Тут потребуется второй аргумент хука useEffect. Поговорим о нём подробнее.

## Второй аргумент хука React.useEffect

Второй аргумент хука useEffect называется dependencies (deps, или зависимости). В этом уроке вы впервые столкнётесь с концепцией зависимостей в хуках.

Чтобы выполнить побочный эффект только при монтировании, вторым аргументом в хук useEffect нужно передать пустой массив:

Скопировать кодJSX

function NavBar = (props) => {

const [state, setState] = useState({

isVisible: true

});

useEffect(() => {

*// Код выполнится только при первичном монтировании*

console.log('Привет! Я примонтировался');

}, []) *// Передали пустой массив вторым аргументом в хук*

return (

<div>

{/\* ... \*/}

</div>

);

}

Теперь вне зависимости от того, изменится ли внутреннее состояние или пропсы компонента NavBar, код из useEffect выполнится только при монтировании. Такой компонент будет идентичен классовому аналогу:

Скопировать кодJSX

class NavBar extends React.Component {

constructor(props){

super(props);

this.state = {

isVisible: true

}

}

componentDidMount() {

console.log('Привет! Я примонтировался');

}

render() {

return (

<div>

{/\* ... \*/}

</div>

);

}

}

Некоторые хуки из библиотеки React тоже поддерживают зависимости — с ними познакомимся в следующих уроках.

### Зависимости в хуках

В одном из уроков вы изучили [метод жизненного цикла](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/7e2a703f-0df6-4ca7-9d20-a3e67e1ed501/lessons/f40cbd35-fdfb-468e-a517-2495c4e039e4/) componentDidUpdate. В этом методе для того, чтобы реализовывать вариативную логику, можно сравнивать предыдущие значения пропсов (prevProps) и состояния (prevState) с текущими. Зависимости (deps) в чём-то похожи на prevProps и prevState, только они более автоматизированные и гибкие.

Если зависимости передать вторым аргументом в хук useEffect в виде массива, то этот побочный эффект будет вызываться каждый раз, когда они изменятся. Звучит сложно, но на деле всё довольно понятно. Посмотрите сами:

Скопировать кодJSX

function NavBar = (props) => {

const [state, setState] = useState({

isVisible: true

});

useEffect(() => {

*// Код выполнится при монтировании,*

*// а также при изменении isVisible*

console.log(`Состояние isVisible: ${state.isVisible}`);

}, [state.isVisible]) *// Передали «зависимость» в хук*

return (

<div>

{/\* ... \*/}

</div>

);

}

Сообщение выведется в консоль при монтировании и в случае, если состояние isVisible изменится. Прочие изменения (например, пропсов) не приведут к повторному выполнению побочного эффекта. В зависимости можно передать любое количество аргументов, все они должны быть помещены в массив:

Скопировать кодJSX

useEffect(() => {

// Код побочного эффекта

}, [

props,

state.visible,

someOtherVariable,

state.active,

title,

fetchData

])

Вы можете передать массив или объект в зависимости, но мы рекомендуем использовать в качестве зависимостей примитивы и только нужные ключи объектов, а для массивов использовать JSON.stringify() или деструктуризацию:

Скопировать кодJSX

useEffect(() => {

*// Код побочного эффекта*

}, [

...props.products,

someObject.nestedValue,

JSON.stringify(filters.values)

])

Дело в том, что зависимости в хуке сравниваются с предыдущими значениями, и чем меньше в зависимостях будет сложных и вложенных объектов, тем быстрее будет работать код. С помощью зависимостей вы можете управлять побочными эффектами в компонентах и вызвать их, когда изменились значения, действительно важные для побочного эффекта.

# Хуки. Асинхронные эффекты в React.useEffect

В этом уроке продолжим разбираться с хуком useEffect. На этот раз посмотрим, что делать, если в компоненте нужно выполнить асинхронный эффект.

Первый аргумент useEffect всегда должен быть синхронной функцией. Это позволит избежать отложенного изменения условий и связанных с этим багов. Если в функциональном компоненте требуется выполнить асинхронный побочный эффект, например запросить данные с сервера, то делать это необходимо внутри хука useEffect:

Скопировать кодJSX

const Product = ({productId}) => {

const [state, setState] = useState({

productData: null,

loading: true

})

useEffect(() => {

const getProductData = async () => {

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

const res = await fetch(`/api/v1/products/${productId}`);

const data = await res.json();

setState({ productData: data.productData, loading: false });

}

getProductData();

}, [productId])

return (

*// ...*

)

}

При использовании useEffect не нужно дополнительно сравнивать пропс productId c предыдущим ему значением, как это делалось в методе жизненного цикла componentDidUpdate классового компонента. Хук useEffect сам сравнит зависимости, и если productId изменится, то новые данные будут запрошены с сервера.

import React from 'react';

import styles from './styles.module.css';

const Film = ({ data }) => {

const image = (

<img

src={

data.image

? `https://api.nomoreparties.co${data.image.formats.thumbnail.url}`

: 'https://via.placeholder.com/250x150'

}

alt={data.nameRU}

/>

);

return (

<div>

<div className={styles.img}>{image}</div>

<p className={styles.name}>{data.nameRU}</p>

<p className={styles.description}>{`${data.year}, ${data.country}`}</p>

<p className={styles.description}>{`${data.duration} мин.`}</p>

</div>

);

};

const App = () => {

const [state, setState] = React.useState(

{

isLoading: false,

hasError: false,

data: []

})

React.useEffect(() => {

const getFilms = async () => {

setState({ ...state, hasError: false, isLoading: true });

try {

const res = await fetch('https://api.nomoreparties.co/beatfilm-movies');

const data = await res.json();

setState({ ...state, data, isLoading: false })

}

catch (e) {

setState({ ...state, hasError: true, isLoading: false });

};

};

getFilms();

}, [])

const { data, isLoading, hasError } = state;

return (

<div className={`${styles.app} ${styles.grid}`}>

{isLoading && 'Загрузка...'}

{hasError && 'Произошла ошибка'}

{!isLoading &&

!hasError &&

data.length &&

data.map((film, index) => <Film key={index} data={film} />)}

</div>

);

}

export default App;

# Хуки. Использование React.useEffect со сбросом

В этом уроке мы завершаем изучение хука useEffect. Вы уже познакомились с побочными эффектами, которые не требуют сброса. Но бывают эффекты, в которых нужно выполнить сброс. Например, при отслеживании курсора на странице: в этом случае важно сбросить слушатель события при размонтировании, чтобы не случилось утечки памяти. Это можно сделать двумя способами: с помощью методов жизненного цикла в классовых компонентах и с помощью хука useEffect со сбросом. Реализуем оба этих способа и сравним.

Рассмотрим пример с классовым компонентом, который использует два метода жизненного цикла:

Скопировать кодJSX

class MouseSpy extends React.Component {

state = {

mousePosition: []

};

*// Устанавливаем слушатель события при монтировании*

componentDidMount() {

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

}

*// Сбрасываем слушатель события при удалении компонента из DOM*

componentWillUnmount() {

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

}

trackMousePos = (e) => {

this.setState({

mousePosition: [e.clientX, e.clientY]

});

};

render(){

return (

*//...*

)

}

}

В этом примере мы используем два метода жизненного цикла: componentDidMount и componentWillUnmount. Теперь посмотрим на такой же компонент, только написанный с помощью функции и хука useEffect:

Скопировать кодJSX

const MouseSpy = () => {

const [mousePosition, setMousePosition] = useState([]);

useEffect(()=>{

*// Устанавливаем слушатель события при монтировании*

document.addEventListener("mousemove", trackMousePos);

*// Сбрасываем слушатель события при удалении компонента из DOM*

return () => {

document.removeEventListener("mousemove", trackMousePos);

}

}, [])

const trackMousePos = (e) => {

setMousePosition([e.clientX, e.clientY])

}

return (

*//...*

)

}

Обратите внимание, что внутри функции, которая выполняется в useEffect, мы возвращаем ещё одну функцию — это и есть код сброса. Именно эта функция и будет вызвана при размонтировании компонента или при изменении зависимостей у эффекта.

Посмотрите ещё раз на код сброса:

Скопировать кодJSX

useEffect(()=>{

*// Код эффекта*

*// Код сброса*

return () => {

*// отписка от событий, закрытие соединений*

}

}, [])

Эффект ведёт себя как совокупность двух методов жизненного цикла: componentDidMount и componentWillUnmount, но есть одно исключение. Если вовсе не передать второй аргумент в эффект, то сброс будет выполняться не только при размонтировании, но и при каждом рендере компонента. Также сброс будет вызываться при изменении зависимостей, переданных в хук useEffect вторым аргументом.

После классовых компонентов это поведение может показаться непривычным, но ему есть вполне логичное объяснение. Проще всего разобраться с поведением на примере.

Представьте, что у вас есть список чатов и вы выбрали один из них. Компонент, который отображает список сообщений, должен произвести такие действия (побочные эффекты):

1. Запросить у сервера историю сообщений чата.
2. Подписаться на получение сообщений из чата, чтобы они появлялись в интерфейсе.

При выборе другого чата важно отписаться от сообщений из предыдущего.

Взгляните, как выглядел бы код при использовании классового компонента:

Скопировать кодJSX

class Chat extends React.Component {

componentDidMount() {

*// Получаем историю сообщений*

ChatAPI.getHistory(this.props.chatId);

*// Подписываемся на получение новых сообщений*

ChatAPI.subscribe(chatId);

}

componentDidUpdate(prevProps) {

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

*// Отписываемся от получения новых сообщений,*

*// если изменился chatId или при размонтировании*

ChatAPI.unsubscribe();

*// Получаем историю сообщений*

ChatAPI.getHistory(this.props.chatId);

*// Подписываемся на получение новых сообщений*

ChatAPI.subscribe(chatId);

}

}

componentWillUnmount() {

*// Отписываемся от получения новых сообщений*

*// при размонтировании*

ChatAPI.unsubscribe();

}

render() {

return (

*//...*

)

}

}

Код, который имеет отношение к одному эффекту в классовом компоненте, разнесён на части. Кроме того, код частично дублируется в методах componentDidUpdate и componentDidMount.

А теперь посмотрим на реализацию такого компонента с использованием хуков:

Скопировать кодJSX

const Chat = ({chatId}) => {

useEffect(() => {

*// Получаем историю сообщений*

ChatAPI.getHistory(chatId);

*// Подписываемся на получение новых сообщений*

ChatAPI.subscribe(chatId);

return () => {

*// Отписываемся от получения новых сообщений,*

*// если изменился chatId или при размонтировании*

ChatAPI.unsubscribe();

}

}, [chatId])

return (

*//...*

)

}

Сброс будет вызван при изменении chatId или размонтировании. Хук useEffect позволяет написать всю логику побочного эффекта в одном месте. Кроме того, удалось избежать дублирования кода, так как сброс в useEffect работает как совокупность методов componentDidUpdate и componentWillUnmount. Хотя useEffect и не обладает всеми возможностями методов жизненного цикла в классовых компонентах, он способен решить большинство повседневных задач, которые возникают у React-разработчика.

## Повторение: побочные эффекты в функциональных компонентах с применением хука useEffect

Вот что важно помнить:

* Существует шесть режимов работы хука:
  1. Без зависимостей:

Скопировать кодJSX

useEffect(() => {

console.log('Я вызываюсь на каждый рендер')

});

* 1. С пустым массивом зависимостей:

Скопировать кодJSX

useEffect(() => {

console.log('Я вызываюсь только на первый рендер')

}, []);

* 1. С зависимостями:

Скопировать кодJSX

useEffect(() => {

console.log(`

Я вызываюсь только если изменились значения,

указанные в зависимостях

`)

}, [state.someValue, props.someOtherValue]);

* 1. Без зависимостей и со сбросом:

Скопировать кодJSX

useEffect(() => {

console.log('Я вызываюсь на каждый рендер')

return () => {

console.log('Сброс выполняется на каждый рендер')

}

});

* 1. С пустым массивом зависимостей и со сбросом:

Скопировать кодJSX

useEffect(() => {

console.log('Я вызываюсь только на первый рендер')

return () => {

console.log('Сброс выполняется только при размонтировании')

}

}, []);

* 1. С зависимостями и со сбросом:

Скопировать кодJSX

useEffect(() => {

console.log(`

Я вызываюсь только если изменились значения,

указанные в зависимостях

`)

return () => {

console.log(`

Сброс выполняется каждый раз, когда изменились значения,

указанные в зависимостях

`)

}

}, [state.someValue, props.someOtherValue]);

* Первый аргумент хука useEffect — всегда синхронная функция. Асинхронный код можно написать внутри неё, но вызывать его без await, либо использовать then.

Скопировать кодJSX

useEffect(() => {

const getProductData = async () => {

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

const res = await fetch(`/api/v1/products/${productId}`);

const data = await res.json();

setState({ productData: data.productData, loading: false });

}

getProductData();

}, [productId])

* Сброс выполняется при каждом вызове хука useEffect, например при изменении зависимостей.
* В компоненте возможно любое количество эффектов, можно использовать useEffect многократно в рамках одного компонента.

Последние несколько уроков были непростыми, но вы справились и познакомились с очень важным хуком из библиотеки React, который позволяет работать с побочными эффектами в функциональных компонентах. В следующем уроке расскажем про использование уже знакомых вам рефов в функциональных компонентах. А пока перейдём к задачам.

const MouseTrack = (props) => {

const [mousePosition, setMousePosition] = React.useState([]);

React.useEffect(() => {

document.addEventListener('mousemove', trackMousePos);

return () => document.removeEventListener('mousemove', trackMousePos);

}, []);

const trackMousePos = e => {

setMousePosition(...mousePosition, [e.clientX, e.clientY]);

};

return (

<div>

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

<h2>

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

<br />

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

</h2>

</div>

);

}

# Хуки. useRef

Вы уже знаете, как работать с побочными эффектами в функциональных компонентах. Теперь нужно разобраться, как в них обстоят дела с рефами. В этом уроке расскажем о хуке из библиотеки React — useRef.

Из урока [про рефы в классовых компонентах](https://practicum.yandex.ru/learn/react/courses/8bb9f1d2-104b-4854-a4d5-d5d8766421ec/sprints/9582/topics/6673a3cf-f58b-4824-a5f0-32616f61ff80/lessons/7a35fc6f-ad83-414e-b546-6e85029b442c/) вы помните, что рефы могут использоваться для императивного доступа к DOM из React-компонента. В функциональных компонентах тоже есть такая возможность. Для этого в библиотеке React есть хук useRef. Этот хук возвращает изменяемый ref-объект, который содержит свойство .current. Оно инициализируется аргументом, переданным в хук. Возвращаемый объект сохраняется в течение всей жизни компонента, пока мы не изменим его напрямую. В отличие от state, мы можем изменить значение current обычным присваиванием. Мы ещё расскажем, зачем это бывает нужно, но сперва разберём, как использовать рефы для прямого доступа к DOM.

Запишем пример с применением хука useRef:

Скопировать кодJSX

import { useRef, useState } from 'react';

function InputWithEditButton() {

const inputRef = useRef(null);

const [disabled, setDisabled] = useState(true);

const onEditClick = () => {

inputRef.current.focus();

};

return (

<>

<input type="text" ref={inputRef} />

<button onClick={onEditClick}>Edit field</button>

</>

);

}

В качестве аргумента мы передали null. Можно вовсе не передавать аргумент, в этом примере это не критично, но мы рекомендуем использовать null вместо пустого (undefined) аргумента.

Дело в том, что значение current не определено до первого рендера: ссылка на DOM-элемент присваивается ему только в блоке return. Значение null примечательно тем, что выражение typeof null вернёт "object", тогда как typeof undefined вернёт "undefined". Вот наглядный пример, когда это можно использовать:

Скопировать кодJSX

function SomeComponent() {

const undefinedRef = useRef();

const nullRef = useRef(null);

if (typeof nullRef.current === "object") {

*// Этот блок кода выполнится, так как typeof null === "object"*

}

if (typeof undefinedRef.current === "object") {

*// Этот блок кода не выполнится, так как typeof undefined === "undefined"*

}

return (

*//...*

)

}

Этот компонент использует два рефа, одному из которых в качестве начального значения передали null.

## Альтернативное использование useRef

Вы уже знаете, что хук useRef возвращает объект с ключом current, который остаётся неизменным при повторном рендеринге. Значение current можно менять присваиванием напрямую, а аргумент хука позволяет задать начальное значение. Эту особенность хука можно использовать для решения специфических задач. Например, когда вам нужно иметь доступ к предыдущим значениям state или props в компоненте. Хук useRef отлично подходит для этой задачи:

Скопировать кодJSX

function GameResult({points}) {

const prevPointsRef = useRef();

useEffect(() => {

prevPointsRef.current = points;

});

const prevPoints = prevPointsRef.current;

return <h1>Новый рекорд: {points}, Предыдущий рекорд: {prevPoints}</h1>;

}

В этом примере мы использовали хук useEffect, который вызывается после каждого рендера. Эта особенность позволяет записать в prevPointsRef.current значение пропса points, которое соответствует предыдущему рендеру. Чтобы лучше понять этот момент, представим, что мы сделали очень простую игру, где при нажатии на кнопку добавляются баллы:

Скопировать кодJSX

const Game = () => {

const [points, setPoints] = useState(0);

const addPoint = () => setPoints(points + 1);

return (

<>

<button onClick={addPoint}>Добавить очко Гриффиндору</button>

<GameResult points={points} />

</>

)

}

ReactDOM.render(<Game />, document.getElementById("root"));

Так выглядит последовательность присваиваний и рендеров:

1. Рендерим компонент Game с помощью ReactDOM.render.
2. В компоненте Game инициализируется state и функция addPoint.
3. Компонент Game рендерит кнопку и компонент GameResult со значением из state в качестве пропса points.
4. Компонент GameResult инициализирует константу prevPointsRef с помощью хука useRef. Начальное значение prevPointsRef.current равно undefined.
5. Компонент GameResult рендерит такую разметку: <h1>Новый рекорд: 0, Предыдущий рекорд:</h1>. Значение предыдущего рекорда берётся из prevPointsRef.current и всё ещё равно undefined, поэтому в разметке нет значения предыдущего рекорда на данный момент.
6. В компоненте GameResult после рендера вызывается useEffect, в нём мы присваиваем значение пропса points рефу prevPointsRef.current. Помните, что изменение current не ведёт к повторному рендерингу.
7. В компоненте Game нажимаем на кнопку и увеличиваем число баллов с нуля до единицы.
8. Компонент Game рендерится повторно, так как изменился state, и обновлённое значение попадает в компонент GameResult.
9. Компонент GameResult рендерит обновившуюся разметку, значение prevPointsRef.current не сбрасывается на undefined, так как useRef уже был вызван при первом рендере.
10. Компонент GameResult рендерит такую разметку: <h1>Новый рекорд: 1, Предыдущий рекорд: 0</h1>. Теперь предыдущий рекорд отображается, так как был записан на шестом шаге.
11. В компоненте GameResult после рендера вызывается useEffect, в нём мы снова присваиваем значение пропса points нашему рефу prevPointsRef.current.

Мы рекомендуем использовать комбинацию хуков useEffect и useRef для реализации сложных компонентов, например в случае, когда логика вашего компонента связана с предыдущими значениями props и/или state.

В следующем уроке мы продолжим изучать хуки из библиотеки React. Но прежде чем переходить дальше, нужно закрепить навыки использования рефов в функциональных компонентах.

Задача

Вы уже выполняли похожее задание в уроке про рефы в классовых компонентах. Но форм мало не бывает. Поэтому есть ещё одна — с восстановлением пароля. В форме два компонента — PasswordResetForm и TextInput.

TextInput — компонент поля ввода. Не содержит состояния.

PasswordResetForm — компонент формы. Есть состояния email и isSubmited.

Сейчас по клику на кнопку «Восстановить пароль» ничего не происходит. Доработайте эту функциональность:

* При сабмите формы обновите состояние email. В состояние должно попасть значение поля ввода компонента TextInput. Для этого воспользуйтесь рефами. Создайте реф emailInput и передайте его как обычный пропс inputElement в компонент TextInput.
* При монтировании компонента PasswordResetForm фокус должен устанавливаться на поле ввода компонента TextInput. Воспользуйтесь уже созданным хуком useEffect.

import React from 'react';

import inputStyles from './text-input.module.css';

export default function TextInput({inputElement}) {

return (

<input type='email' placeholder="Введите свой e-mail" required

className={inputStyles.input}

ref={inputElement}

/>

)

}

import React from 'react';

import TextInput from '../text-input/text-input';

import { ReactComponent as PasswordResetLogo } from '../../images/protect.svg';

import formStyles from './password-reset-form.module.css'

export default function PasswordResetForm() {

const [email, setEmail] = React.useState('');

const [isSubmited, setSubmited] = React.useState(false);

const emailInput = React.useRef();

React.useEffect(()=> {

emailInput.current.focus();

}, []);

const handleFormSubmit = (e) => {

e.preventDefault();

setEmail(emailInput.current.value);

setSubmited(true);

};

return (

<form onSubmit={handleFormSubmit} className={formStyles.form}>

<PasswordResetLogo className={formStyles.logo} />

<TextInput inputElement={emailInput}/>

<button type='submit' className={formStyles.submit}>

Восстановить пароль

</button>

{isSubmited &&

<p className={formStyles.description}>

Ссылка на восстановление пароля отправлена на {email}

</p>

}

</form>

)

}

Задача курс валют

Нужно сохранять предыдущее значение курса в реф, а по разнице между предыдущим курсом и текущим показывать тренд валютной пары. Если актуальное значение курса выше предыдущего — тренд положительный. В противном случае — отрицательный.

Весь код приложения записан в одном компоненте — CoursePredictor. Мы расскажем в общих чертах о текущей функциональности компонента:

* Набор утилитарных функций getRandomNum и getRatio исправлять не требуется. Первая функция генерирует случайное число, а вторая — занимается вычислением тренда. Мы ещё не написали отдельный интенсив по бэкенду, поэтому в каждом уроке предлагаем работать с захардкоженными данными.
* Состояние currentCourse содержит актуальный курс валюты.
* isDown — утилитарная функция внутри компонента. Она влияет на визуализацию тренда — отображает отрицательный или положительный тренд.

В компоненте есть три константы с JSX:

* ratioElement отвечает за отображения блока трендов,
* previousElement — за отображение блока с предыдущим курсом,
* currentElement — за отображение блока с текущим курсом.

С функциональностью приложения немного разобрались. Перейдём к самому заданию.

Для работы с предыдущим значением валюты потребуется реф. Создайте пустой реф previousCourseRef. После монтирования или обновления компонента рефу previousCourseRef присваивается текущее состояние currentCourse.

Для корректной работы с рефами присвойте переменной previousCourse значение previousCourseRef.current. Именно эту переменную используйте для всех расчётов и отрисовки данных.

Теперь значением переменной previousCourse нужно поделиться с функциями для расчётов. Передайте переменную вторым параметром в функции getRatio. Про расчёты поговорили — обсудим отрисовку данных.

Содержимое ratioElement отрисовывается только при наличии и currentCourse, и previousCourse.

Элемент output в previousElement должен отрисовываться при наличии previousCourse:

<output className={coursePredictorStyles.value}>

{/\* Предыдущее значение валюты \*/} ₽

</output>

import React from 'react';

import {ReactComponent as TriangleImage} from '../../images/triangle.svg'

import coursePredictorStyles from './course-predictor.module.css';

const MIN\_VALUE = 50;

const MAX\_VALUE = 250;

const getRandomNum = () => (Math.random() \* (MAX\_VALUE - MIN\_VALUE) + MIN\_VALUE).toFixed(2);

const getRatio = (current, previous=0) => ((current - previous) / (current < previous ? previous : current) \* 100).toFixed(2);

export default function CoursePredictor() {

const [currentCourse, setCurrentCourse] = React.useState();

const previousCourseRef = React.useRef();

React.useEffect(() => {

previousCourseRef.current = currentCourse;

});

const previousCourse = previousCourseRef.current;

const handleGenerateCourse = () => {

setCurrentCourse(getRandomNum());

};

const isDown = getRatio(currentCourse, previousCourse) < 0 ? coursePredictorStyles.ratioDown : coursePredictorStyles.ratioUp;

const ratioElement = (

previousCourse && currentCourse &&

<output className={isDown}>

<TriangleImage className={coursePredictorStyles.ratioIcon} />

{getRatio(currentCourse, previousCourse)}%

</output>

);

const previousElement = (

<p className={coursePredictorStyles.previousElement}>

Предыдущее значение:

{previousCourse &&

<output className={coursePredictorStyles.value}>

{previousCourse} ₽

</output>

}

</p>

);

const currentElement = (

currentCourse &&

<output className={coursePredictorStyles.currentElement}>

{currentCourse} ₽

</output>

);

return (

<section className={coursePredictorStyles.card}>

{previousElement}

<div className={coursePredictorStyles.cardRow}>

{currentElement}

{ratioElement}

</div>

<p className={coursePredictorStyles.caption}>

За 1 $

</p>

<button onClick={handleGenerateCourse} className={coursePredictorStyles.button}>

Заглянуть в будущее

</button>

</section>

)

}

# Повышение производительности. Мемоизация. Хуки useMemo и useCallback

В этом уроке поговорим про важный аспект разработки — мемоизацию. Это состояние, когда вместо нормальных ответов разработчик начинает общаться мемами и гифками.

На самом деле мы поговорим про оптимизацию вычислений в функциональных компонентах. И совсем чуть-чуть про мемы. Но чтобы оптимизировать код, сначала его надо написать. Так что познакомьтесь с новым простым приложением — «Мемопедией».



Разберём компоненты, которые уже написаны. Все данные — тестовые и лежат в компоненте App. В App определено состояние для чекбоксов и поискового запроса:

Скопировать кодJSX

const App = () => {

const [search, setSearch] = React.useState({});

// Состояние search - обновляется при нажатии на кнопку поиска

const [checkboxes, setChecked] = React.useState({});

// Состояние каждого чекбокса

// Остальное содержимое компонента App

};

В этом же компоненте описаны все обработчики событий и функция filteredItems. Она возвращает массив мемов, которые удовлетворяют текущему состоянию search:

Скопировать кодJSX

const App = () => {

const [search, setSearch] = React.useState({});

*// Состояние search - обновляется при нажатии на кнопку поиска*

const [checkboxes, setChecked] = React.useState({});

*// Состояние каждого чекбокса*

const handleCheck = (event) => {

const checkboxName = event.target.name;

const checkboxValue = event.target.checked;

*// Получаем имя и значение checked чекбокса, по которому кликнул пользователь*

setChecked({

...checkboxes,

[checkboxName]: checkboxValue

});

*// Обновляем состояние*

};

const handleSearch = () => {

setSearch(checkboxes);

};

*// При нажатии на кнопку поиска сохраняем в состояние search текущее состояние*

*// чекбоксов*

const filteredItems = memesList.filter(meme => search[meme.year]);

*// Функция фильтрации массива memesList*

*// Остальное содержимое компонента App*

};

Теперь разберёмся, что отрисовывает App, и посмотрим на каждый из дочерних компонентов:

Скопировать кодJSX

const App = () => {

*// Состояние в компоненте App, обработчики событий handleCheck и handleSearch,*

*// функция filteredItems*

return (

<div>

{

checkboxesData.map(item => (

<label key={item.key}>

{item.name}

<Checkbox name={item.name} onChange={handleCheck} />

</label>

))

}

{/\* По нажатию на чекбокс мы вызываем обновление состояния checkboxes \*/}

<button type="button" onClick={handleSearch}>

Поозорничать

</button>

{/\* По нажатию на кнопку мы вызываем обновление состояния search \*/}

<List list={filteredItems} />

{/\* Компонент List отображает результаты поиска \*/}

</div>

);

};

const List = ({ list }) => {

return (

<ul>

{

list.map((item) => (

<ListItem key={item.id} item={item} />

))

}

</ul>

);

};

*// Компонент List получает отфильтрованный массив мемов*

const ListItem = ({ item }) => (

<li>

<span>{item.name}</span>

<img src={item.image} alt={item.name} />

</li>

);

*// Компонент ListItem отрисовывает карточку мема*

const Checkbox = ({ type='checkbox', name, onChange }) => (

<input type={type} name={name} onChange={onChange} />

);

*// Компонент чекбокса*

### Пример функции с вычислением

Вы познакомились со структурой «Мемопедии» и, скорее всего, задались вопросом — зачем мы про это рассказываем? Всё дело в небольшой вычислительной функции filteredItems. Если вызвать console.log внутри функции и попробовать понажимать чекбоксы, то в консоли появятся сообщения о вызове функции:

Скопировать кодJSX

const App = () => {

const filteredItems = memesList.filter((meme) => {

console.log('Меня вызвали!');

return search[meme.year]

});

}

*// console.log*

*// Меня вызвали!*

Сообщения будут появляться при каждом клике по чекбоксу, а их количество будет равно длине массива memesList. С точки зрения React всё логично — мы изменили состояние checkboxes, а значит, нужен повторный рендеринг и, соответственно, повторные вычисления. Но для разработчика этот процесс не логичен, ведь мы не обновляли состояние search, которое используется в вычислениях.

React очень быстрый, поэтому для небольшого приложения повторные вычисления не страшны: они никак не скажутся на скорости работы приложения. Но и мемов в мире немало — даже в наших тестовых данных их несколько. А на самом деле данных может быть в тысячи раз больше.

При работе с большими объёмами данных стоимость вычислений возрастает: они могут замедлить скорость работы приложения. Чтобы этого избежать, существует мемоизация — способ оптимизации вычислений. Мемоизация позволяет сохранить данные выполнения функции и предотвратить повторные вычисления, если входные данные не изменились.

Мемоизация — это как списывание на контрольной по алгебре. Отличник (первое мемоизированное вычисление) самостоятельно подсчитал, чему равно расстояние между пунктами A и B, из которых одновременно навстречу друг другу с разной скоростью выехали автомобили. А двоечник (повторный рендеринг без изменения deps) просто списал ответ у отличника и получил те же 320 км.

## Хук useMemo

Один из способов мемоизации вычислений в React — хук useMemo. Он возвращает мемоизированное значение. Хук принимает два аргумента — функцию и массив deps. В первый аргумент передадим функцию filteredItems. В качестве массива зависимостей передадим состояние search — повторные вычисления будут происходить только при его изменении:

Скопировать кодJSX

const App = () => {

const filteredItems = React.useMemo(

() =>

memesList.filter((meme) => {

console.log('Меня вызвали!')

return search[meme.year]

}),

[search]

);

}

*// console.log*

*// Меня вызвали!*

Теперь изменение состояния checkboxes не приведёт к повторному вычислению filteredItems — сообщения в консоли появятся только при монтировании компонента App и изменении состояния search.

## Хук useCallback

Хук useCallback похож на useMemo, но он мемоизирует не вычисления, а экземпляр функции. Хук useCallback возвращает не значение, а мемоизированный колбэк. При повторном рендере компонента происходит повторное объявление функций. При этом, если такие функции (например, обработчики событий) передаются дочерним компонентам в качестве пропса, дочерние компоненты также заново отрисуются.

Разберёмся на практике, как возникает повторный рендер. Для этого добавим простой обработчик в компонент List и передадим его дочерним компонентам ListItem. Чтобы визуализировать каждый повторный рендер компонентов — добавим console.log в List и ListItem:

Скопировать кодJSX

const List = ({ list }) => {

console.log('Компонент List заново родился!')

const handleListItemClick = (item) => {

console.log(item, 'По этому элементу кликнули')

}

*// Обработчик, выводящий объект того мема, по которому кликнули*

return (

<ul>

{

list.map((item) => (

<ListItem key={item.id} item={item} onCardClick={handleListItemClick} />

))

}

</ul>

);

};

*// Компонент List получает отфильтрованный массив мемов*

const ListItem = ({ item, onCardClick }) => {

console.log('Детишки компонента List тоже заново родились!')

const handleClick = () => onCardClick(item);

*// Передаём в обработчик handleListItemClick проп item*

return (

<li onClick={handleClick}>

{/\* Добавляем слушатель onClick \*/}

<span>{item.name}</span>

<img src={item.image} alt={item.name} />

</li>

);

}

*// Компонент ListItem отрисовывает карточку мема*

*// console.log*

*// > Компонент List заново родился!*

*// > Детишки компонента List тоже заново родились!*

Теперь при повторном рендере компонента List создаётся новый экземпляр функции handleListItemClick. Эта функция передаётся дочерним компонентам ListItem в качестве пропса и вызывает их повторный рендер, поэтому при каждом клике по чекбоксу оба сообщения выведутся в консоль.

Если мы обернём компонент ListItem в НОС React.memo, то результат останется таким же — пропс handleListItemClick по-прежнему будет изменяться после повторного рендера:

Скопировать кодJSX

const List = ({ list }) => {

console.log('Компонент List заново родился!')

const handleListItemClick = (item) => {

console.log(item, 'По этому элементу кликнули')

}

*// Обработчик, выводящий объект мема, по которому кликнули*

return (

<ul>

{

list.map((item) => (

<ListItem key={item.id} item={item} onCardClick={handleListItemClick} />

))

}

</ul>

);

};

*// Компонент List получает отфильтрованный массив мемов*

const ListItem = React.memo(({ item, onCardClick }) => {

console.log('Детишки компонента List тоже заново родились!')

const handleClick = () => onCardClick(item);

*// Передаём в обработчик handleListItemClick проп item*

return (

<li onClick={handleClick}>

{/\* Добавляем слушатель onClick \*/}

<span>{item.name}</span>

<img src={item.image} alt={item.name} />

</li>

);

}

);

*// Компонент ListItem отрисовывает карточку мема*

*// console.log*

*// > Компонент List заново родился!*

*// > Детишки компонента List тоже заново родились!*

Мы увидим, что при клике на любой элемент интерфейса в консоль попадают оба сообщения. Причина, как мы уже говорили выше, — повторное декларирование функции handleListItemClick.

Оптимизация таких случаев — одна из задач хука useCallback. В качестве первого аргумента в функцию передаётся колбэк, в качестве второго — массив deps. Если мы обернём слушатель handleListItemClick в хук useCallback, дочерний компонент ListItem будет получать мемоизированный колбэк handleListItemClick:

Скопировать кодJSX

const List = ({ list }) => {

console.log('Компонент List заново родился!')

const handleListItemClick = React.useCallback(

(item) => {

console.log(item, 'По этому элементу кликнули')

},

[]

);

*// Обработчик, выводящий объект мема, по которому кликнули*

return (

<ul>

{

list.map((item) => (

<ListItem key={item.id} item={item} onCardClick={handleListItemClick} />

))

}

</ul>

);

};

const ListItem = React.memo(({ item, onCardClick }) => {

console.log('Детишки компонента List тоже заново родились!')

const handleClick = () => onCardClick(item);

*// Передаём в обработчик handleListItemClick проп item*

return (

<li onClick={handleClick}>

{/\* Добавляем слушатель onClick \*/}

<span>{item.name}</span>

<img src={item.image} alt={item.name} />

</li>

);

}

);

*// Компонент ListItem отрисовывает карточку мема*

*// console.log*

*// > Компонент List заново родился!*

Новый экземпляр функции handleListItemClick будет создан только при изменении пропсов, которые были переданы в компонент List. Это значит, что при работе с интерфейсом, компоненты ListItem будут повторно отрендерены только при необходимости. В идеальном мире компонент List также можно обернуть в НОС React.memo: это уменьшит количество ненужных рендеров компонента.

В хук useCallback можно передавать и аргументы, поэтому хук используется для мемоизации вычислений, где нужны не только deps, но и передаваемые параметры.

На этом мемоизация заканчивается (не бойтесь, мы про хуки, а не про мемы). Оптимизация React-приложения важна, но в рамках проектной работы пока не понадобится. React — быстрая библиотека, а хуки useMemo и useCallback применяются, когда очень большие объёмы данных действительно сказываются на скорости работы приложения. Но не переживайте, вы не зря прочитали всю эту теорию: useMemo и useCallback могут быть очень полезны в комбинации с другими хуками. Об этом поговорим в следующих уроках, а пока — перейдём к заданиям.

Задача калькулятор

//calculator.js

import calculatorStyles from './calculator.module.css';

import React from 'react';

function isFloat(n) {

return Number(n) === n && n % 1 !== 0;

}

export function Calculator() {

const [value, setValue] = React.useState(0);

const [decimalPending, setDecimalPending] = React.useState(false);

const { pad, display, btn, btn\_equal, container } = calculatorStyles;

const [currentOperation, setCurrentOperation] = React.useState('');

const memoizedValue = React.useRef(0);

const finishDecimal = React.useCallback(() => {

setDecimalPending(false);

}, []);

const clear = React.useCallback(() => {

memoizedValue.current = 0;

setCurrentOperation('');

}, []);

const onNumberClick = React.useCallback(

e => {

if (decimalPending) {

finishDecimal();

}

setValue(Number(value + e.target.value));

},

[value, decimalPending, finishDecimal]

);

const onResetClick = React.useCallback(

() => {

setValue(0);

clear();

if (decimalPending) {

setDecimalPending(false);

}

},

[decimalPending, clear]

);

const onDecimalClick = React.useCallback(

() => {

if (!isFloat(value)) {

setValue(value + '.');

setDecimalPending(true);

}

},

[value]

);

const onEqualClick = React.useCallback(

() => {

switch (currentOperation) {

case 'plus': {

const res = memoizedValue.current + value;

setValue(res);

clear();

return res;

}

case 'minus': {

const res = memoizedValue.current - value;

setValue(res);

clear();

return res;

}

case 'multiply': {

const res = memoizedValue.current \* value;

setValue(res);

clear();

return res;

}

case 'divide': {

const res = memoizedValue.current / value;

setValue(res);

clear();

return res;

}

default: {

return value;

}

}

},

[clear, value, currentOperation]

);

const onOperationClick = React.useCallback(

operation => () => {

let currentValue = value;

if (memoizedValue.current) {

currentValue = onEqualClick();

}

setCurrentOperation(operation);

memoizedValue.current = currentValue;

setValue(0);

},

[value, onEqualClick]

);

return (

<div className={container}>

<div className={pad}>

<div className={display}>{value}</div>

<button onClick={onResetClick} className={btn}>

AC

</button>

<button onClick={onNumberClick} className={btn} value={1}>

1

</button>

<button onClick={onNumberClick} className={btn} value={2}>

2

</button>

<button onClick={onNumberClick} className={btn} value={3}>

3

</button>

<button onClick={onOperationClick('divide')} className={btn} disabled={decimalPending}>

÷

</button>

<button onClick={onNumberClick} className={btn} value={4}>

4

</button>

<button onClick={onNumberClick} className={btn} value={5}>

5

</button>

<button onClick={onNumberClick} className={btn} value={6}>

6

</button>

<button onClick={onOperationClick('multiply')} className={btn} disabled={decimalPending}>

×

</button>

<button onClick={onNumberClick} className={btn} value={7}>

7

</button>

<button onClick={onNumberClick} className={btn} value={8}>

8

</button>

<button onClick={onNumberClick} className={btn} value={9}>

9

</button>

<button onClick={onOperationClick('minus')} className={btn} disabled={decimalPending}>

-

</button>

<button onClick={onNumberClick} className={btn} value={0}>

0

</button>

<button onClick={onDecimalClick} className={btn} disabled={decimalPending}>

.

</button>

<button onClick={onEqualClick} className={`${btn} ${btn\_equal}`} disabled={decimalPending}>

=

</button>

<button onClick={onOperationClick('plus')} className={btn} disabled={decimalPending}>

+

</button>

</div>

</div>

);

}

Задача загруженность дорог

import React from 'react';

import suezTrafficStyles from './suez-traffic.module.css';

import { ReactComponent as NavigationIconImage } from '../../images/arrow.svg';

const GAP = 60;

const ALLOWABLE\_DIFFERENCE = 30;

const MAX\_TRAFFIC\_LOAD = 100;

const INITIAL\_TRAFFIC\_LOAD = [

{

color: '#BDFF00',

value: 0

},

{

color: '#FA290C',

value: 36.46

},

{

color: '#000000',

value: 42.71

},

{

color: '#FA290C',

value: 51.04

},

{

color: '#DBFF00',

value: 74.48

},

{

color: '#00E717',

value: 100

}

];

const getCoordinates = currentCoordinate =>

Math.floor(

Math.random() \* (currentCoordinate + GAP - (currentCoordinate - GAP)) +

(currentCoordinate + GAP)

);

const getTrafficLoad = currentTrafficValue => {

const newTrafficValue = Math.floor(

Math.random() \* (MAX\_TRAFFIC\_LOAD - currentTrafficValue) + currentTrafficValue

);

return newTrafficValue - currentTrafficValue > ALLOWABLE\_DIFFERENCE

? getTrafficLoad(currentTrafficValue)

: newTrafficValue;

};

export default function SuezTraffic() {

const [coordinate, setCoordinate] = React.useState(0);

const [load, setLoad] = React.useState([]);

React.useEffect(() => {

const interval = setInterval(() => {

setCoordinate(getCoordinates(coordinate));

}, 4000);

setLoad(INITIAL\_TRAFFIC\_LOAD);

return () => clearInterval(interval);

}, []);

const generateLoad = React.useMemo(() => {

return load.reduce(

(acc, reducer, position) => [

...acc,

{

color: reducer.color,

value: getTrafficLoad(acc[position - 1] ? acc[position - 1].value : 0)

}

],

[]

)}, [load]);

const renderTrafficBar = generateLoad

.map(element => `${element.color} ${element.value}%`)

.join(', ');

const trafficBarStyles = {

background: `linear-gradient(90deg, ${renderTrafficBar})`

};

return (

<div className={suezTrafficStyles.root}>

<NavigationIconImage

style={{ transform: `rotate(${coordinate}deg)` }}

className={suezTrafficStyles.arrow}

/>

<div className={suezTrafficStyles.trafficBarContainer}>

<span className={suezTrafficStyles.trafficBar} style={trafficBarStyles} />

</div>

</div>

);

}

}

# Правила хуков

В предыдущих уроках вы познакомились с основными хуками из библиотеки React. Как вы уже знаете, хуки — функции, которые позволяют использовать возможности классовых компонентов внутри функциональных. Но у этих функций есть правила, которые важно соблюдать, чтобы хуки работали как надо:

**1. Хуки нужно использовать только на верхнем уровне функционального компонента**

React полагается на порядок вызова хуков и их количество. Если количество или порядок меняются из-за вложенности хуков в условия или другие языковые конструкции, то React не гарантирует надёжность кода и консистентность данных в таком компоненте. Разберём это на примере:

Скопировать кодJSX

function ReciepeForm() {

*// 1. Используем переменную состояния title*

const [title, setTitle] = useState('Рецепт пасты с трюфелем');

*// 2. Используем эффект для сохранения данных формы в localStorage*

useEffect(function saveFormToLocalStorage() {

localStorage.setItem('reciepe', name);

});

*// 3. Используем переменную состояния personCount*

const [personCount, setPersonCount] = useState(4);

*// 4. Используем эффект для обновления заголовка страницы*

useEffect(function updateDocumentTitle() {

document.title = `${title} | Количество персон: ${personCount}`;

});

return (

*// ...*

)

}

При первом рендере хуки в этом компоненте работают так:

Скопировать кодJSX

// 1. Инициализируем переменную title значением 'Рецепт пасты с трюфелем'

useState('Рецепт пасты с трюфелем')

// 2. Добавляем эффект для сохранения данных формы

useEffect(saveFormToLocalStorage})

// 3. Инициализируем переменную personCount значением 4

useState(4)

// 4. Добавляем эффект для обновления заголовка страницы

useEffect(updateDocumentTitle)

Когда пользователь что-то меняет в форме, происходят повторные рендеры. При повторном рендеринге хуки работают так:

Скопировать кодJSX

// 1. Читаем переменную состояния title

useState('Рецепт пасты с креветками')

// 2. Заменяем эффект для сохранения данных формы

useEffect(saveFormToLocalStorage})

// 3. Читаем переменную personCount

useState(4)

// 4. Заменяем эффект для обновления заголовка страницы

useEffect(updateDocumentTitle)

Пока порядок остаётся таким, внутренние механизмы React могут гарантировать предсказуемость поведения компонента, ведь работа одних хуков может быть завязана на работу других. Если мы изменяем порядок хуков или их количество, это приведёт к багам:

Скопировать кодJSX

if (title !== '') {

useEffect(function saveFormToLocalStorage() {

localStorage.setItem('reciepe', title);

});

}

Теперь, если пользователь очистит поле с названием рецепта, вызов эффекта проигнорируется, а последовательность хуков нарушится. Из-за особенностей внутреннего устройства библиотеки React порядок вызовов хуков критически важен. Вот как станут вести себя хуки в случае нарушения этого правила:

Скопировать кодJSX

// 1. Читаем переменную состояния title

useState('')

// 2. Заменяем эффект для сохранения данных формы

// useEffect(saveFormToLocalStorage})

// Хук игнорируется, так как title === ''.

// 3. Читаем переменную personCount

useState(4)

// Ошибка при чтении переменной personCount, так как изменился порядок

// Теперь этот хук второй, а не третий, как было раньше

// 4. Заменяем эффект для обновления заголовка страницы

useEffect(updateDocumentTitle)

// Ошибка при замене эффекта, так как изменился порядок

// Теперь этот хук третий, а не четвёртый, как было раньше

В этом примере React попытался достать значение personCount из состояния, а наткнулся на эффект saveFormToLocalStorage — раньше тот был на втором месте, и аналогичная проблема произошла дальше, так как все следующие хуки сместились.

Чтобы работать с условиями в эффектах, нужно помещать эти условия внутрь хука useEffect:

Скопировать кодJSX

useEffect(function saveFormToLocalStorage() {

if (title !== '') {

localStorage.setItem('reciepe', title);

}

});

Так мы не нарушим правила порядка и количества хуков.

**2. Хуки можно использовать только в React-функциях**

Вызов хуков в обычных JS-функциях невозможен — для них React не запускает внутренние механизмы работы с хуками. Рассмотрим пример:

Скопировать кодJSX

import { useState } from "React";

function addTitle(autoGeneration = false) {

const [title, setTitle] = useState("");

if (autoGeneration) {

setTitle("Новая папка 11111");

return title;

}

return title + " (Проект1011)";

}

Такой код не будет работать, так как не проходит через «внутреннюю кухню» библиотеки React.

Чтобы обойти это ограничение, вы можете создавать собственные хуки на основе тех, которые уже доступны в библиотеке, а затем вызывать их в функциональных компонентах. К этой потрясающей возможности мы ещё вернёмся в рамках курса.

Следуя этим правилам, можете быть уверены: хуки будут работать предсказуемо, а в коде будет меньше багов. В следующем уроке расскажем про неотъемлемую часть любого приложения — работу с формами и пользовательским вводом.

# Ref как аргумент компонента. React.forwardRef

Бывают случаи, когда вам может понадобиться передать реф из родительского компонента в качестве параметра. Например, если вы создаёте компонент-обёртку над стандартным DOM-элементом. Это можно сделать с помощью специального метода forwardRef. Разберём, как его применять.

Сперва создадим компонент кнопки:

Скопировать кодJSX

function UIButton(props) {

return (

<button onClick={props.onClick}>

{props.content}

</button>

);

}

Пока это обычный элемент кнопки, который обёрнут в компонент. Чтобы добавить его существованию немного смысла, используем компонент UIButton в форме пользовательского соглашения:

Скопировать кодJSX

function AgreementForm() {

function handleNo() {

console.log("Нет");

}

function handleYes() {

console.log("Да");

}

return (

<div>

<p>Добавить существованию немного смысла.</p>

<UIButton content={"No"} onClick={handleNo} />

<UIButton content={"Yes"} onClick={handleYes} />

</div>

);

}

Обычно при рендере таких форм один из вариантов ответа выделяется с помощью фокуса.

Вызвать событие фокуса у элемента можно с помощью рефа:

Скопировать кодJSX

function AgreementForm() {

const yesButtonRef = useRef(null);

useEffect(() => {

if (yesButtonRef.current) {

yesButtonRef.current.focus();

}

}, []);

// ...

}

Но если вы попробуете просто создать реф в компоненте AgreementForm, передать его компоненту UIButton и вызвать у него событие focus(), то ничего не получится. Причина в том, что для корректной работы событие фокуса должно вызываться у элемента button, а не у его компонента-обёртки UIButton.

Чтобы решить эту проблему, можно воспользоваться специальным методом forwardRef, который позволяет «пробросить» реф через пропсы компонента до нужного элемента:

Скопировать кодJSX

const UIButton = React.forwardRef((props, ref) => {

return (

*// ref прямо из параметра вкладывается в аргумент кнопки*

<button ref={ref} onClick={props.onClick}>

{props.content}

</button>

);

});

Как вы могли заметить, способ создания компонента немного изменился. Теперь UIButton — это не функция, которая возвращает React-компонент, а результат выполнения метода forwardRef. Но если разобраться, концептуально ничего не поменялось: UIButton — всё такой же компонент-обёртка над элементом button, просто создаётся немного по-другому.

В качестве аргументов метод forwardRef получает не только пропсы, но и ref, который ему передают. Затем этот реф можно передать необходимому DOM-элементу.

После применения forwardRef нужная кнопка автоматически подсвечивается при монтировании компонента. Форма стала удобнее, пользователям нужно совершать меньше действий, а вы можете добавить ещё один полезный метод в свой инструментарий и перейти к следующему уроку.

# Работа с формами в React

В этом уроке разберём, как работать с формами. В React HTML-элементы формы ведут себя не как остальные DOM-элементы. Причина кроется в их собственном внутреннем состоянии. Мы расскажем, как работать с каждым элементом формы и как обрабатывать множество полей одним обработчиком.

## Управляемые компоненты

Поля ввода, такие как input, textarea и select, обычно обладают собственным состоянием и обновляют его, когда пользователь вводит данные. React позволяет связать события изменения и значение поля ввода в компоненте c кодом компонента, который содержит это поле ввода. Такие поля ввода называются «управляемые компоненты». Их значения синхронизируются с состоянием компонента. Это удобно, потому что позволяет управлять содержимым форм напрямую из JavaScript.

Вот пример управляемого компонента:

Скопировать кодJSX

function Input() {

*// Состояние, в котором содержится значение поля ввода*

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

*// Обработчик изменения поля ввода обновляет состояние*

function handleChange(e) {

setValue(e.target.value);

}

return (

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

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

);

}

ReactDOM.render((

<Input />

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

А вот похожий пример, только с textarea и кнопкой сброса введённого текста:

Скопировать кодJSX

function NewMessage() {

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

function handleChange(e) {

setValue(e.target.value);

}

function resetValue() {

setValue('');

}

return (

<>

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

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

</>

);

}

В этом примере нажатие на кнопку «Очистить» сбросит значение value.

## Неуправляемые компоненты

Чтобы не создавать обработчики для каждого события, в React есть возможность использовать неуправляемые компоненты. Такие компоненты используют DOM как источник данных, доступ к которому можно получить через ref:

Скопировать кодJSX

class BlogpostTitleInput extends React.Component {

constructor(props) {

super(props);

*// Создаём ссылку на DOM-элемент*

this.input = React.createRef();

}

handleSubmit = (event) => {

*// Получаем доступ к значению поля ввода через DOM*

alert('Заголовок поста в блоге: ' + this.input.current.value);

event.preventDefault();

}

render() {

return (

<form onSubmit={this.handleSubmit}>

<label>

Заголовок:

<textarea ref={this.input} />

</label>

<input type="submit" value="Отправить" />

</form>

);

}

}

Как вы могли заметить, изменения в поле ввода не соединены с компонентом и не влияют на его внутреннее состояние. Это делает код более производительным, но в большинстве случаев вы не заметите разницы.

Неуправляемые компоненты можно использовать при интеграции React с другим кодом. У неконтролируемых полей ввода есть дополнительная возможность — значение по умолчанию. Это специальный пропс, который будет использован при первичном монтировании. Особенность значения по умолчанию в том, что дальнейшие его изменения не повлекут за собой дополнительного рендеринга:

Скопировать кодJSX

<textarea ref={this.input} defaultValue={"Привет, мир!"} />

Строка «Привет, мир!» уже будет записана в textarea.

## Загрузка файлов

Поле ввода <input type="file"> позволяет выбрать один или несколько файлов для загрузки с устройства на сервер:

Скопировать кодJSX

class AvatarInput extends React.Component {

constructor(props) {

super(props);

this.fileInput = React.createRef();

}

handleSubmit = (event) => {

event.preventDefault();

alert(

`Новый аватар - ${this.fileInput.current.files[0].name}`

);

}

render() {

return (

<form onSubmit={this.handleSubmit}>

<label>

Выберите фото:

<input type="file" ref={this.fileInput} />

</label>

<button type="submit">Сохранить</button>

</form>

);

}

}

В React <input type="file"> всегда неуправляемый компонент — его значение нельзя установить средствами JS, это может сделать только пользователь.

## Работа с select

В HTML выбранный пункт списка в <select> отмечается с помощью атрибута selected у тега <option>. В React для удобства используется атрибут value у тега <select>. Это позволяет обновлять значение только в одном месте, а не во всех <option>:

Скопировать кодJSX

const EmployeeRole = () => {

const [role, setRole] = useState('designer');

const handleChange = (event) => {

setRole(event.target.value);

}

return (

<label>

Ваша роль в проекте:

<select value={role} onChange={handleChange}>

<option value="designer">Дизайнер</option>

<option value="developer">Разработчик</option>

<option value="teamlead">Тимлид</option>

<option value="project-manager">Руководитель проекта</option>

</select>

</label>

);

}

<select> можно использовать и для множественного выбора:

Скопировать кодJSX

<select multiple={true} value={['cheddar', 'mozzarella']}>

...

</select>

Поля ввода <input type="text">, <textarea>, и <select> работают очень похоже. Они принимают атрибут value и позволяют получить доступ к значению в событии onChange через event.target.value.

## Работа с checkbox и radio

Для <input type="radio" /> необходимо проверять установленное значение на соответствие value и использовать результат этой проверки в атрибуте checked:

Скопировать кодJSX

const SelectAppearance = () => {

const [mode, setMode] = useState('dark');

const onValueChange = (event) => {

setMode(event.target.value);

}

const formSubmit = (event) => {

event.preventDefault();

console.log(mode)

}

return (

<form onSubmit={formSubmit}>

<div>

<label>

<input

type="radio"

value="light"

checked={mode === "light"}

onChange={onValueChange}

/>

Светлая тема

</label>

</div>

<div className="radio">

<label>

<input

type="radio"

value="dark"

checked={mode === "dark"}

onChange={onValueChange}

/>

Тёмная тема

</label>

</div>

<button type="submit">

Submit

</button>

</form>

);

}

Этот тип поля также принимает атрибут value и позволяет получить доступ к значению в событии onChange через event.target.value. При этом атрибут checked должен быть булевого типа.

Для чекбоксов применяются те же правила, только в event.target будет не value, а checked:

Скопировать кодJSX

const RememberOnLogIn = () => {

const [checked, setChecked] = useState(true);

const onChange (event) => {

*// Тут у event.target используем свойство checked*

setState(event.target.checked);

}

return (

<label>

Запомнить меня и больше не разлогинивать

<input

name="rememberMe"

type="checkbox"

checked={checked}

onChange={onChange}

/>

</label>

);

}

Как и у radio, у чекбоксов в checked атрибут должен быть булевым.

## Обработка нескольких полей одним обработчиком

Для обработки нескольких полей ввода одним обработчиком нужно назначить каждому полю атрибут name и подготовить структуру состояния так, чтобы ключи состояния совпадали с этими атрибутами полей. В самом обработчике у нас есть доступ к событию, а в нём — к event.target.name. С использованием вычисляемых имён свойств и name мы можем адресно записывать изменения в состояние:

Скопировать кодJSX

const SubscribeSettings = () => {

const [state, setState] = useState({

*// Соответствует name полю ввода*

email: "",

*// Соответствует subscribed полю ввода*

subscribed: false,

});

const handleInputChange = (event) => {

const target = event.target;

*// Определяем, откуда пришло событие: из чекбокса или текстового поля ввода*

const value = target.type === "checkbox" ? target.checked : target.value;

const name = target.name;

*// Применяем вычисляемые имена свойств*

setState({

...state,

[name]: value,

});

};

return (

<form>

<label>

Получать уведомления

<input

name="subscribed"

type="checkbox"

checked={state.subscribed}

onChange={handleInputChange}

/>

</label>

<label>

E-mail:

<input

disabled={state.subscribed}

name="email"

type="text"

value={state.email}

onChange={handleInputChange}

/>

</label>

</form>

);

};

Такой подход здорово экономит время на написании обработчиков для каждого поля.

Умение работать с формами и пользовательским вводом — важный навык, который пригодится вам при разработке почти любого сайта или приложения на React. В этом уроке вы разобрались в управляемых и неуправляемых компонентах и узнали, как управлять формами одним обработчиком и задавать значения по умолчанию.

Вы прошли важную тему. Впереди вас ждёт проект, но перед этим — немного практики.

Задача форма

В этом задании вам предстоит поработать с формой для ввода адреса доставки в интернет-магазине. В форме есть следующие поля:

1. Тип адреса: дом, работа, другой
2. Город
3. Улица
4. Индекс
5. Дом
6. Корпус
7. Доставка до двери: да/нет

Если выбрана доставка до двери, появляются дополнительные поля:

1. Подъезд
2. Этаж
3. Квартира
4. Дополнительная информация

Напишите один обработчик для всех полей формы и пробросьте значения из state в каждое поле в соответствии с атрибутом name.

import React, { useState } from 'react';

import './styles.css';

export default function App() {

const [state, setState] = useState({

addressType: 'home',

city: '',

street: '',

zipCode: '',

houseNumber: '',

building: '',

toDoor: true,

entrance: '',

floor: '',

apartment: '',

info: ''

});

const onChange = e => {

if (e.target.name === 'toDoor') {

setState({ ...state, toDoor: e.target.checked });

} else {

setState({ ...state, [e.target.name]: e.target.value });

}

};

const submit = e => {

e.preventDefault();

console.log(state);

};

return (

<div className="App">

<h2 className="mb25">Укажите адрес доставки</h2>

<form onSubmit={submit}>

<div className="field mb25">

<select value={state.addressType} onChange={onChange} name="addressType">

<option value="home">Дом</option>

<option value="work">Работа</option>

<option value="other">Другое</option>

</select>

<label className="label">Тип адреса</label>

</div>

<div className="field mb25">

<input

type="text"

name="city"

placeholder="Введите название города"

value={state.city}

onChange={onChange}

/>

<label className="label">Город</label>

</div>

<div className="field mb25">

<input

type="text"

name="street"

placeholder="Введите название улицы"

value={state.street}

onChange={onChange}

/>

<label className="label">Улица</label>

</div>

<div className="field\_\_group mb25">

<div className="field field\_min">

<input type="number" name="zipCode" value={state.zipCode} onChange={onChange} />

<label className="label">Индекс</label>

</div>

<div className="field field\_min">

<input type="number" name="houseNumber" value={state.houseNumber} onChange={onChange} />

<label className="label">Дом</label>

</div>

<div className="field field\_min">

<input type="number" name="building" value={state.building} onChange={onChange} />

<label className="label">Корпус</label>

</div>

</div>

<div className="checkbox-container mb25">

<input

type="checkbox"

name="toDoor"

id="toDoor"

checked={state.toDoor}

onChange={onChange}

/>

<label className="to-door-label" htmlFor="toDoor">

Требуется доставка до двери

</label>

</div>

{state.toDoor && (

<>

<div className="field\_\_group mb25">

<div className="field field\_min">

<input type="number" name="entrance" value={state.entrance} onChange={onChange} />

<label className="label">Подъезд</label>

</div>

<div className="field field\_min">

<input type="number" name="floor" value={state.floor} onChange={onChange} />

<label className="label">Этаж</label>

</div>

<div className="field field\_min">

<input type="number" name="apartment" value={state.apartment} onChange={onChange} />

<label className="label">Квартира</label>

</div>

</div>

<div className="field mb25">

<textarea

name="info"

rows="4"

placeholder="Код домофона, как пройти"

value={state.info}

onChange={onChange}

/>

<label className="label">Дополнительная информация</label>

</div>

</>

)}

<button type="submit">Сохранить адрес</button>

</form>

</div>

);

}

# Обработка ошибок

Итак, вы написали приложение, в котором нет ошибок. Вообще ни одной. Приложение быстро работает и обладает безупречной отказоустойчивостью. Вы показали его другим разработчикам. Они в шоке. Мы — тоже. Джордан Валке жмёт вам руку и просит присоединиться к команде React.

А потом вы просыпаетесь.

Реальность такова, что создать приложение, в котором не произойдёт ни одной ошибки, практически невозможно: в системе слишком много участников и всегда есть вероятность, что что-то пойдёт не так.

В этом уроке разберём, как «ловить» и обрабатывать ошибки, которые могут возникнуть во время работы React-приложения. Сначала рассмотрим подход Error boundary, а затем — уже знакомую из JS конструкцию try...catch.

## Подход Error boundary

Error boundary — подход, который позволяет «поймать» ошибки, которые возникают в приложении на этапе рендеринга или в методах жизненного цикла компонентов, и отреагировать на них. Если в приложении возникает ошибка, вы можете отправить её детальное описание в сервис логирования ошибок. Одновременно с этим пользователь увидит страницу с указанием на проблему и инструкцией по дальнейшим действиям.

Разберём это на примере одноимённого компонента — ErrorBoundary:

Скопировать кодJSX

class ErrorBoundary extends React.Component {

constructor(props) {

super(props);

this.state = { hasError: false };

}

*// с помощью этого метода меняем стейт компонента при возникновении ошибки:*

static getDerivedStateFromError(error) {

return { hasError: true };

}

*// с помощью этого метода логируем информацию об ошибке:*

componentDidCatch(error, info) {

console.log("Возникла ошибка!", error, info);

}

render() {

if (this.state.hasError) {

*// если возникла ошибка, сообщаем об этом пользователю в специальном компоненте:*

return (

<section>

<h1>Что-то пошло не так :(</h1>

<p>

В приложении произошла ошибка. Пожалуйста, перезагрузите страницу.

</p>

</section>

);

}

*// если всё работает штатно, рендерим дочерние компоненты*

return this.props.children;

}

}

Компонент ErrorBoundary состоит из трёх методов:

* getDerivedStateFromError отвечает за изменение стейта компонента при появлении ошибки;
* componentDidCatch нужен, чтобы удобно логировать ошибки;
* render — стандартный рендер-метод. Используется в ErrorBoundary для отображения компонента с сообщением об ошибке.

Метод жизненного цикла getDerivedStateFromError вызывается сразу после того, как в одном из дочерних компонентов возникает ошибка. Этот метод в параметрах получает информацию об ошибке — error. А благодаря тому, что getDerivedStateFromError возвращает значение обновлённого стейта, его удобно применять для управления условным рендерингом: сразу после возникновения ошибки вы можете показать пользователю компонент, который сообщит об этом.

componentDidCatch — ещё один метод жизненного цикла, который также вызывается после возникновения ошибки. Получает в параметрах информацию об ошибке — error, и информацию о компоненте, в котором возникла ошибка — info. Эти параметры отлично подходят для логирования.

Метод render применяется точно так же, как и в любом другом классовом компоненте. Как правило, ErrorBoundary рендерит либо страницу с сообщением об ошибке, либо children, если ошибок нет.

Сейчас не существует способа вызвать методы жизненного цикла getDerivedStateFromError и componentDidCatch в функциональном компоненте, поэтому компонент ErrorBoundary возможно создать только с применением классового подхода.

ErrorBoundary применяется как компонент-обёртка: методы getDerivedStateFromError и componentDidCatch срабатывают при возникновении ошибки в дочерних компонентах на любом уровне вложенности. Например, вы можете использовать ErrorBoundary прямо в корневом компоненте App — в таком случае будут отслеживаться все возможные ошибки:

Скопировать кодJSX

export default function App() {

return (

<div className="App">

<ErrorBoundary>

{/\* если в компоненте RestOfTheApp или любом его дочернем компоненте \*/}

{/\* возникнет ошибка, она будет обработана в ErrorBoundary \*/}

<RestOfTheApp />

</ErrorBoundary>

</div>

);

}

Error boundary отлично подходит для реагирования на проблемы с рендерингом компонентов, но методы getDerivedStateFromError и componentDidCatch никак не отвечают на ошибки, которые возникают в обработчиках событий. Это обусловлено спецификой React: обработчики вызываются вне этапов рендеринга компонента, и даже если в обработчике возникнет ошибка, это не помешает React, ведь ему уже известно, что показывать на экране.

В случае, если вам нужно следить за ошибками в обработчиках и реагировать на них, можно воспользоваться стандартной конструкцией JavaScript — try...catch.

## Конструкция try...catch

Для начала вспомним, как работает эта конструкция. В блок try помещается код, во время выполнения которого может возникнуть ошибка. А в блоке catch описывается логика обработки этой ошибки. [Подробнее с особенностями работы try...catch можно ознакомиться на MDN.](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch)

А вот пример, как try...catch можно применять в обработчиках событий:

Скопировать кодJSX

function ComponentWithError() {

const [error, setError] = useState(null);

const handleClick = () => {

try {

*// В этом случае в хэндлере точно возникнет ошибка*

throw new Error("Simple error");

} catch (err) {

*// Обычно ошибки логируются или обрабатываются дополнительной логикой,*

*// но для наглядности мы просто кладём ошибку в стейт*

setError(err.message);

}

};

return (

<div className="App">

<h1>{error}</h1>

<button onClick={handleClick}>Make a mistake</button>

</div>

);

}

Если в обработчике handleClick возникнет ошибка, ErrorBoundary её проигнорирует. Но благодаря try...catch эту ошибку можно поймать и обработать в блоке catch.

В утопическом мире с идеальными приложениями ошибок не бывает. Но пока такого мира и таких приложений создать не удалось, с ошибками нужно что-то делать.

Обычно ошибки просто логируются в специальных документах или сервисах, например [в Sentry](https://sentry.io/), после чего можно изучить эти ошибки и исправить причины их возникновения.

В некоторых случаях ошибки обрабатываются прямо на клиенте, но тут всё индивидуально. Например, если «упал» запрос, можно попробовать отправить его ещё раз.

Но самое важное — не забывать о пользователях. Если какая-то функциональность приложения нарушилась, нужно сообщить об этом пользователю с помощью всплывающего окна или иным способом, здесь вам поможет try...catch. А если возникшая ошибка может привести к поломке всего приложения, важно не допускать полного падения. В этом случае следует применить подход ErrorBoundary и показать пользователю информацией об ошибке, а так же подсказать, что делать дальше.

Эти два способа удобно использовать вместе: Error boundary применить для отлова ошибок рендеринга, а в try...catch обернуть любые другие уязвимые места в коде.