**Введение**

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

Функциональный и классовый подходы довольно сильно отличаются с точки зрения внутреннего устройства, но в целом были призваны решать одни и те же задачи, а с появлением хуков во многом стали равноценными и взаимозаменяемыми. Изучив хуки, вы сможете использовать в своих проектах тот подход, который вам больше понравится.

Что же такое хуки? Хук — это специальная функция (обычно предоставляемая движком «Реакта»), которую можно вызывать в теле функционального компонента, чтобы осуществить доступ к его внутренним ресурсам.

В прошлых уроках вы изучили два основных преимущества классовых компонентов: наличие внутреннего состояния и управление эффектами в рамках жизненного цикла. Для решения обеих задач в функциональных компонентах можно использовать соответствующие типы хуков. Их и разберём в этой теме.

# Внутреннее состояние: хук useState

Вспомним пример, где можно было голосовать за рейтинг пользователей. Сравним предыдущую реализацию и вариант с функциональным компонентом и хуком состояния React.useState.

Скопировать кодJSX

*// Классовый компонент User*

class User extends React.Component {

constructor(props) {

super(props);

*// Начальные значения внутреннего состояния*

this.state = { rating: 0 };

}

*/\**

*\* Обработчики событий: изменяют внутреннее состояние*

*\*/*

handleLike = () => {

this.setState({ rating: 1 });

};

handleDislike = () => {

this.setState({ rating: -1 });

};

render() {

return (

<div className="user">

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

{this.props.name}

<div className="rating">

<button onClick={this.handleLike} disabled={this.state.rating > 0}>👍</button>

{this.state.rating}

<button onClick={this.handleDislike} disabled={this.state.rating < 0}>👎</button>

</div>

</div>

);

}

}

Скопировать кодJSX

*// Функциональный компонент User*

function User(props) {

*// Хук, управляющий внутренним состоянием.*

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

*/\**

*\* Обработчики событий: изменяют внутреннее состояние*

*\*/*

function handleLike() {

setRating(1);

}

function handleDislike() {

setRating(-1);

}

return (

<div className="user">

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

{props.name}

<div className="rating">

<button onClick={handleLike} disabled={rating > 0}>👍</button>

{rating}

<button onClick={handleDislike} disabled={rating < 0}>👎</button>

</div>

</div>

);

}

Код стал короче и 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 деструктуризации 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 (при первом вызове совпадает с начальным) и функцию, с помощью которой его можно изменять — «сеттер». Эта функция действует аналогично this.setState, однако, в отличие от последней, она управляет не всем набором переменных состояния компонента, а лишь какой-то одной из них — в данном случае rating.

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

# Деструктуризация

Важно обратить внимание на то, 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 каких-либо имён, однако, с помощью деструктуризации мы создаём на основе них две переменные и присваиваем им удобные нам имена.

Таким образом, хук React.useState можно использовать в компонентах несколько раз для создания любого числа подобных пар, управляющих отдельными переменными внутреннего состояния. При этом значения могут иметь любой тип данных, включая массивы и объекты:

Скопировать кодJSX

const [rating, setRating] = React.useState(0); // -1, 0 или 1

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

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

# Главное правило хуков

Так как «Реакт» не знает о том, какой именно переменной состояния управляет каждый из хуков, внутри себя он использует порядковый номер вызова каждого хука для того, чтобы ассоциировать с ним данные в хранилище.

Это указывает на важное правило: порядок хуков не должен меняться между вызовами компонента. Следовательно, 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, то избегайте их изменений напрямую (мутаций) — всегда передавайте в сеттер новый объект. Для этого удобно использовать оператор spread ..., создающий 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 движок «Реакта» сравнивает старое и новое значение с помощью оператора ===, а значит, мы не можем передавать исходный объект, иначе проверка всегда будет возвращать true.

Кстати, это правило применимо и к классовым компонентам.

Пример «хорошие дела»

function GoodDeeds() {

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

function handleAddTask(e) {

const input = e.target.previousSibling;

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

/\* А здесь передавайте в сеттер массив с новым элементом из 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'));

**Эффекты: хук useEffect**

Второе важное преимущество классовых компонентов — методы обратного вызова для создания «побочных эффектов». Хук useEffect позволяет устанавливать аналогичные колбэки и в функциональных компонентах.

Снова сравним пример из предыдущего урока и его аналог с использованием хуков.

Скопировать кодJSX

*// Пример из предыдущего урока*

class NeonCursor extends React.Component {

constructor(props) {

super(props);

this.state = { top: 0, left: 0 };

}

*// Метод будет вызван сразу после монтирования: создаём эффекты*

componentDidMount() {

document.addEventListener('mousemove', this.handleMouseMove);

document.body.classList.add('no-cursor');

}

*// Метод будет вызван непосредственно перед размонтированием: удаляем эффекты*

componentWillUnmount() {

document.body.classList.remove('no-cursor');

document.removeEventListener('mousemove', this.handleMouseMove);

}

handleMouseMove = (event) => {

this.setState({

top: event.pageY,

left: event.pageX,

});

};

render() {

return (

<img

src="./cursor.png"

width={30}

style={{

position: 'absolute',

top: this.state.top,

left: this.state.left,

pointerEvents: 'none',

}}

/>

);

}

}

Скопировать кодJSX

*// Перепишем код с использованием хуков*

function NeonCursor() {

const [position, setPosition] = React.useState({ top: 0, left: 0 });

React.useEffect(() => {

function handleMouseMove(event) {

setPosition({

top: event.pageY,

left: event.pageX,

});

}

*// Список действий внутри одного хука*

document.addEventListener('mousemove', handleMouseMove);

document.body.classList.add('no-cursor');

*// Возвращаем функцию, которая удаляет эффекты*

return () => {

document.body.classList.remove('no-cursor');

document.removeEventListener('mousemove', handleMouseMove);

};

});

return (

<img

src="./cursor.png"

width={30}

style={{

position: 'absolute',

top: position.top,

left: position.left,

pointerEvents: 'none',

}}

/>

);

}

Кроме того, что с помощью React.useState мы заменили this.state на единый объект position, произошло ещё одно изменение: управление эффектами переместилось внутрь колбэка, передаваемого в хук React.useEffect.

«Реакт» вызовет этот колбэк после того, как компонент будет смонтирован или обновлён — то есть этот колбэк является пересечением функциональности componentDidMount и componentDidUpdate. На практике такое объединение оказывается более удобным, однако существует способ и вернуть разделение. Его мы рассмотрим позже.

Чтобы «подчистить» результаты эффекта (когда компонент будет размонтирован), колбэк эффекта может вернуть ещё один колбэк — он будет использован движком «Реакта» по аналогии с componentWillUnmount. Такой колбэк обычно называется “cleanup” (от англ. «очистка»).

Обратите внимание, что сам код эффекта, его очистка и даже обработчик события handleMouseMove описаны рядом: внутри одного вызова React.useEffect — такой подход позволяет лучше организовать код. Кроме того, разработчик может создавать сколько угодно эффектов, если он хочет логически их разделить. Например, в нашем варианте мы могли бы добавить отдельные эффекты для подписки на событие mousemove и для установки класса no-cursor:

Скопировать кодJSX

React.useEffect(() => {

function handleMouseMove(e) {

setPosition({

top: e.pageY,

left: e.pageX,

});

}

document.addEventListener('mousemove', handleMouseMove);

return () => {

document.removeEventListener('mousemove', handleMouseMove);

};

});

React.useEffect(() => {

document.body.classList.add('no-cursor');

return () => {

document.body.classList.remove('no-cursor');

};

});

В случае с классовыми компонентами все инструкции должны быть расположены внутри методов жизненного цикла, что зачастую приводит к их плохой читаемости. Код с использованием функциональных компонентов и хуков лишён этого недостатка и в целом выглядит лаконичнее.

Пример с background

function toggleBackground(shouldShow) {

document.body.classList.toggle('with-bg', shouldShow);

}

function Beautifier() {

const [isBeautiful, setIsBeautiful] = React.useState(false);

React.useEffect(() => {

function handleChange() {

setIsBeautiful(!isBeautiful);

}

toggleBackground(isBeautiful)

}

);

return (

<label>

<input type="checkbox" onChange={handleChange} />

Включить красивый фон

</label>

);

}

ReactDOM.render((

<Beautifier />

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

Пример с курсором

function App() {

const [isCustomCursor, setIsCustomCursor] = React.useState();

function handleChange() {

setIsCustomCursor(!isCustomCursor);

}

return (

<>

<label>

<input type="checkbox" onChange={handleChange}/>

Включить неоновый курсор

</label>

{isCustomCursor && <NeonCursor/>}

</>

);

}

function NeonCursor () {

const [position, setPosition] = React.useState({ top: 0, left: 0 });

React.useEffect(() => {

function handleMouseMove(event) {

setPosition({

top: event.pageY,

left: event.pageX,

});

}

document.addEventListener('mousemove', handleMouseMove);

document.body.classList.add('no-cursor');

return () => {

document.body.classList.remove('no-cursor');

document.removeEventListener('mousemove', handleMouseMove);

};

});

return (

<img

src="https://code.s3.yandex.net/web-code/react/cursor.svg"

width={30}

style={{

position: 'absolute',

top: position.top,

left: position.left,

pointerEvents: 'none',

}}

/>

);

}

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

# Зависимости

Пожалуй, наиболее распространённый вид эффекта — обращение к API за данными.

Предположим, мы разрабатываем стандартное приложение — электронный магазин волшебных палочек. В центре экрана находится основной компонент, отображающий все тактико-технические характеристики выбранной палочки, а снизу находится список альтернативных вариантов — палочек других производителей, которые могут также заинтересовать покупателя.

При клике на одном из таких вариантов меняется переменная состояния приложения selectedId — ID выбранной в данный момент палочки. Эта переменная хранится в стейте компонента App и передаётся в виде пропса основному компоненту MagicWand, который при её изменении должен выполнить запрос в API, чтобы загрузить информацию о выбранной палочке.

Скопировать кодJAVASCRIPT

*// Хук состояния внутри App*

const [selectedId, setSelectedId] = React.useState();

*// return*

Внутри этого компонента есть уже знакомые вам кнопки 👍 и 👎, с помощью которых посетитель может выразить своё отношение к предложенному товару. Их нажатие приводит к изменению внутреннего состояния компонента MagicWand, который создаётся с помощью такого хука:

Скопировать кодJAVASCRIPT

*// Хук состояния внутри WandInfo*

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

*// function handleLike*

*// function handleDislike*

Также внутри компонента MagicWand находится хук эффекта, который как раз выполняет запрос в API за данными после каждой отрисовки:

Скопировать кодJAVASCRIPT

React.useEffect(() => {

fetchMagicWandById(selectedId);

});

Здесь есть маленькая проблема: компонент MagicWand будет перерисован и при изменении выбранной палочки, и при установке лайка или дизлайка, а значит во втором случае у нас будут лишние обращения к API. Для того, чтобы этот эффект вызывался только в первом случае, для него нужно указать зависимость: переменную, изменение которой должно провоцировать выполнение хука.

Если у хука не указаны зависимости, он будет вызван после каждого рендера. Если зависимости указаны — только если хотя бы одна из них изменилась с момента прошлого вызова.

Зависимости передаются вторым аргументом в функцию React.useEffect — в виде массива с переменными. Это могут быть как пропсы, так и переменные внутреннего состояния текущего компонента. В нашем случае нас интересует пропс selectedId, поэтому нужный нам код будет выглядеть так:

Скопировать кодJAVASCRIPT

React.useEffect(() => {

fetchMagicWandById(selectedId);

}, [selectedId]);

Теперь это эффект с зависимостью, который запоминает последнее значение selectedId и вызывает переданный в него колбэк, только когда это значение поменяется. Теперь пользователь может сколько угодно раз подряд ставить лайки и дизлайки одной и той же волшебной палочке, вызывая перерисовку компонента, — лишних запросов к API не будет.

Бывает, что зависимостей больше одной — тогда в массиве будет несколько элементов.

Скопировать кодJAVASCRIPT

React.useEffect(() => {

if (!isFetched) {

fetchMagicWandById(selectedId);

}

}, [isFetched, selectedId]);

Когда к эффекту добавляется хотя бы одна зависимость, появляется важное правило: теперь все используемые в его колбэке пропсы и стейт-переменные должны быть перечислены в массиве его зависимостей. Объяснение этого правила довольно сложное и основывается на понятии замыкания, поэтому для начала его лучше просто запомнить.

# Эффект при монтировании

Из концепции массива зависимостей вытекает интересное свойство: если передать пустой массив, то такой эффект будет вызван всего один раз. Если задуматься, это довольно логично: ведь если ни одной зависимости нет, то нечему и измениться, чтобы этот эффект спровоцировался повторно.

Таким образом, мы можем задать колбэк, который будет вызван всего один раз при монтировании компонента — прямой аналог componentDidMount из классовых компонетов.

Более того, если из такого эффекта мы вернём колбэк-очистку, она будет также вызвана всего один раз, при размонтировании компонента — аналог componentWillUnmount.

Хук с пустым массивом зависимостей не стоит путать с обычным хуком, которым мы пользовались ранее — совсем без второго аргумента — такой хук всегда будет вызываться после каждого рендера.

Зависимости поначалу могут показаться довольно непростой темой, но к ним быстро привыкаешь, и пользоваться ими становится комфортно.

Пример

function playSound(fileName) {

const audio = new Audio();

audio.src = `https://code.s3.yandex.net/web-code/react/${fileName}`;

audio.play();

}

function App() {

const [isTimeShown, setIsTimeShown] = React.useState();

function handleChange() {

setIsTimeShown(!isTimeShown);

}

return (

<>

<label>

<input type="checkbox" onChange={handleChange}/>

Включить время (осторожно, звук!)

</label>

{isTimeShown && <Time/>}

</>

);

}

function Time() {

const [hours, setHours] = React.useState(0);

const [minutes, setMinutes] = React.useState(0);

const [seconds, setSeconds] = React.useState(0);

const [milliseconds, setMilliseconds] = React.useState(0);

React.useEffect(() => {

requestAnimationFrame(() => {

const date = new Date();

setMilliseconds(date.getMilliseconds());

setSeconds(date.getSeconds());

setMinutes(date.getMinutes());

setHours(date.getHours());

});

});

React.useEffect(() => {

playSound('tick.mp3');

}, [seconds]);

React.useEffect(() => {

playSound('gong.mp3');

}, []);

React.useEffect(() => {

return () => {playSound('gong.mp3')};

}, []);

return (

<h2>{hours}:{minutes}:{seconds}:{milliseconds}</h2>

);

}

ReactDOM.render((

<App/>

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

**Заключение**

В этой теме вы изучили сравнительно новую концепцию «Реакта» — хуки функциональных компонентов.

Два основных хука — useState и useEffect — привносят в функциональные компоненты мощь классовых: внутреннее состояние и побочные эффекты.

Разработчики «Реакта» считают, что использование хуков более естественно и близко идеологии «Реакта», чем использование классов. Несмотря на то, что оба подхода остаются равносильными по возможностям, по их мнению, за хуками будущее.

Однако это совсем не означает, что все имеющиеся классовые компоненты должны быть переписаны на хуки. Просто теперь при создании новых проектов у разработчика появляется выбор.

А что бы вам пришлось по душе? Впереди проектная работа, и у вас тоже будет возможность этого выбора.

Привет. При реализации в этой статье дана рекомендация использовать пустой массив. Но тогда появляется ворнинг о том, что функции должны быть указаны в массиве зависимостей. Если же их там указываю, то теряется эффект пустого массива и выводит рекомендацию использовать useEffect callback. Помоги пожалуйста разобраться, как этого избежать, и нужно ли функции вносить в зависимости, или оставить пустой массив?

2 files

https://ca.slack-edge.com/TPV9DP0N4-U01EVLJ5XJM-7d9fb5e82d6d-48

[**Михаил Зятьков**](https://app.slack.com/team/U01EVLJ5XJM)  [7 minutes ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1621935087088200?thread_ts=1621748186.082700&cid=G01ENBXA9BR)

[@Александр Катрюхин](https://yandex-students.slack.com/team/U01F0MCBYVB) Привет, давай сначала разберемся с последним вопросом: нужно ли функции вносить в зависимости, т.к. он здесь самый важный.  
В useEffect в зависимости мы вносим то, что будет изменяться. В данном случае насколько я понимаю не предполагается изменение наших функций handleEscClose и handleOverlayClose.Теперь разберемся, что делать с варнингами.

1. мы можем просто игнорировать его, если понимаем почему и для чего мы это делаем. Для этого просто ставим комментарий // eslint-disable-next-line react-hooks/exhaustive-deps
2. Мы можем перенести тело наших функций в сам эффект, если они нигде больше в компоненте не используются
3. Мы можем вынести эти две функции за рамки нашего компонента, если они не используют никакие данные из самого компонента
4. Если перечисленное выше не наш случай, то добавляем их в массив зависимостей в useEffect, но при этом сами функции оборачиваем в хук [useCallback](https://ru.reactjs.org/docs/hooks-reference.html" \l "usecallback" \t "_blank)

Привет. [@Михаил Зятьков](https://yandex-students.slack.com/team/U01EVLJ5XJM) При редактировании профиля все отрабатывает, но в консоли такой warning. Это ошибка? Запутался немного с управляемыми компонентами...

[**Михаил Зятьков**](https://app.slack.com/team/U01EVLJ5XJM)  [26 days ago](https://yandex-students.slack.com/archives/G01ENBXA9BR/p1621396112015600?thread_ts=1621342918.007800&cid=G01ENBXA9BR)

[@Александр Катрюхин](https://yandex-students.slack.com/team/U01F0MCBYVB) чтобы избежать этой ошибки, тебе нужно в компоненте EditProfile в эффекте добавить небольшую проверку и в зависимости от результатов вызывать  setName и setDescription.  
Смотри, что у тебя сейчас происходит.

1. currentUser по-умолчанию равен пустой строке
2. currentUser мы получаем асинхронно, следовательно, компонент EditProfile создается раньше, чем мы получаем значение с сервера
3. т.к. currentUser - пустая строка, то ‘’.name и ‘’.about равны undefined
4. поэтому мы видим ошибку в консоли

 React.useEffect((currentUser) => {

    if (currentUser) {

      setName(currentUser.name);

      setDescription(currentUser.about);

  }}, [currentUser]);