05. Условный рендеринг

" ИЛИ - || " запинается на правде , "

И - && " запинается на лжи...

[ИЛИ](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie) **[«||»](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie)** [находит](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie) **[первое истинное](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie)**[или](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie) **[последнее ложное (если все ложные)](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie)** [значение](https://learn.javascript.ru/logical-operators" \l "ili-nahodit-pervoe-istinnoeznachenie) и возвращает его

[И](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie) **[«&&»](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie)** [находит](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie) **[первое ложное](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie)** [или](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie) **[последнее истинное (если все истинные)](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie)**[значение](https://learn.javascript.ru/logical-operators" \l "i-nahodit-pervoe-lozhnoeznachenie) и возвращает его

**В JS есть 5 псевдоложных значений:**

* undefined
* null
* 0
* пустая строка
* NaN

**Псевдоистинные значения, кроме исключающих псевдоложные:**

-Любой массив, даже пустой

-любая строка не пустая

Так не пишем, только для примера. Например, есть компонент, и в зависимости от true/false нам нужно отобразить компонент либо нет. Если мы запишем таким образом, используя логические операторы:

export function Accordion(props: AccordionPropsType) {  
 return (  
 <div>  
 <AccordionTitle titleValue={props.title}/>  
 {true && <AccordionBody/>}  
 </div>  
 )  
}

То сработает последний true, компонент <AccordionBody/> отобразится.

Если напишем так:

export function Accordion(props: AccordionPropsType) {  
 return (  
 <div>  
 <AccordionTitle titleValue={props.title}/>  
 {false && <AccordionBody/>}  
 </div>  
 )  
}

То сработает первый false и компонент <AccordionBody/> **не** отобразится. Используя это, запишем условную конструкцию так (первое условие **даст true**):

export function Accordion(props: AccordionPropsType) {  
 return (  
 <div>  
 <AccordionTitle titleValue={props.title}/>  
 {props.collapsed === false && <AccordionBody/>}  
 </div>  
 )  
}

или сделаем simplify

export function Accordion(props: AccordionPropsType) {  
 return (  
 <div>  
 <AccordionTitle titleValue={props.title}/>  
 {!props.collapsed && <AccordionBody/>}  
 </div>  
 )  
}

Таким же образом можно отрефакторить **<Star/>** отображение пяти звездочек:

function App() {  
 return (  
 <div className="App">  
 <PageTitle title={"This is app component"}/>  
 <PageTitle title={"My friends"}/>  
 Article 1  
 <Rating value={2}/>  
 <Accordion title={"Menu"} collapsed={true}/>  
 <Accordion title={"Users"} collapsed={false}/>  
 Article 2  
 <Rating value={1}/>  
 <Rating value={2}/>  
 <Rating value={3}/>  
 <Rating value={4}/>  
 <Rating value={5}/>  
  
 </div>  
 );  
}

export function Rating(props: RatingPropsType) {  
  
 return (  
 <div>  
 <Star selected={props.value > 0}/>  
 <Star selected={props.value > 1}/>  
 <Star selected={props.value > 2}/>  
 <Star selected={props.value > 3}/>  
 <Star selected={props.value > 4}/>  
 </div>)  
}  
  
  
function Star(props: StarPropsType) {  
 if (props.selected === true) {  
 return (<span><b>star </b></span>)  
 } else {  
 return (  
 <span>star </span>  
 )  
 }  
}

06. Inline стили

в Ract не является плохим тоном использовать inline-стили, этим можем воспользоваться.

- Создается объект в самой компоненте, стили с ДЕФИСОМ прописываются camelCase.

- Свойства прописываются через ‘,’ как в объекте,

- Значения свойств берутся в “кавычки”

import classes from './OnOf.module.css';  
  
type OnOffPropsType = {  
 on: boolean  
}  
  
export function OnOff(props: OnOffPropsType) {  
  
 const onStyle = {  
 width: '30px',  
 height: '30px',  
 border: '1px solid black'  
 }  
 const offStyle = {  
 width: '30px',  
 height: '30px',  
 border: '1px solid black'  
 }  
 const indicatorStyle = {  
 width: '30px',  
 height: '30px',  
 borderRadius: '30px',  
 border: '1px solid black'  
 }  
  
 return (  
 <div className={classes.wrapper}>  
 <div style={onStyle}>On</div>  
 <div style={offStyle}>Off</div>  
 <div style={indicatorStyle}>+</div>  
 </div>  
 )  
  
}

В объекте стилей мы можем прописать условия, в данном случае тернарное выражение:

backgroundColor: props.on === true? 'green' : 'red'

Итого получим:

import classes from './OnOf.module.css';  
  
type OnOffPropsType = {  
 on: boolean  
}  
  
export function OnOff(props: OnOffPropsType) {  
  
 const onStyle = {  
 width: '30px',  
 height: '30px',  
 border: '1px solid black',  
 **backgroundColor: props.on === true? 'green' : 'white'**  
 }  
 const offStyle = {  
 width: '30px',  
 height: '30px',  
 border: '1px solid black',  
 **backgroundColor: props.on === false? 'red' : 'white'**  
 }  
 const indicatorStyle = {  
 width: '30px',  
 height: '30px',  
 borderRadius: '30px',  
 border: '1px solid black',  
 **backgroundColor: props.on === false? 'red' : 'green'**  
 }  
  
 return (  
 <div className={classes.wrapper}>  
 <div style={onStyle}>On</div>  
 <div style={offStyle}>Off</div>  
 <div style={indicatorStyle}>+</div>  
 </div>  
 )  
  
}

В строках

backgroundColor: props.on === true? 'green' : 'white'

можем сократить запись

backgroundColor: props.on? 'green' : 'white'

backgroundColor: !props.on? 'red' : 'white'

Можно вынести часть стилей в CSS, часть оставить в inline: (т.е совместно использовать style={} и className={})

import classes from './OnOf.module.css';  
  
type OnOffPropsType = {  
 on: boolean  
}  
  
export function OnOff(props: OnOffPropsType) {  
  
 const onStyle = {  
 backgroundColor: props.on ? 'green' : 'white'  
 }  
 const offStyle = {  
 backgroundColor: !props.on ? 'red' : 'white'  
 }  
 const indicatorStyle = {  
 backgroundColor: !props.on ? 'red' : 'green'  
 }  
  
 return (  
 <div className={classes.wrapper}>  
 <div style={onStyle} className={classes.button}>On  
 </div>  
 <div style={offStyle} className={classes.button}>Off  
 </div>  
 <div style={indicatorStyle} className={classes.circle}>+</div>  
 </div>  
 )  
}

06. Uncontrolled Components **без PROPS**

Это компоненты, которые меняют свое состояние **без PROPS** (в нее они не передаются). **Сделать это можно при помощи событий и использования useState**

export function UnControlledRating() {  
  
 let [value, setValue] = useState(0)  
  
 return (  
  
 <div>  
 <Star selected={value > 0}/>  
 <button onClick={() => setValue(1)}>1</button>  
 <Star selected={value > 1}/>  
 <button onClick={() => setValue(2)}>1</button>  
 <Star selected={value > 2}/>  
 <button onClick={() => setValue(3)}>1</button>  
 <Star selected={value > 3}/>  
 <button onClick={() => setValue(4)}>1</button>  
 <Star selected={value > 4}/>  
 <button onClick={() => setValue(5)}>1</button>  
 </div>  
 )  
}

**Как поменять значение (состояние) у КОНТРОЛИРУЕМОЙ компоненты?**

- Хранить значение за пределами реакта и изменять ее там, потом передаем данные через все дерево(изменения фиксируем с помощью)

- Хранить данные в самой компоненте (как в TODOlist)

**Компонента может быть смешанного типа**

08. CallBacks\_1. Передача callback через props в компоненты.

Кроме данных в PROPS мы можем также отправить ФУНКЦИИ, которую потом можно будет вызывать внутри компоненты.

**НЕКОНТРОЛИРУЕМЫЕ КОМПОНЕНТЫ (НЕТ PROPS)**

Модифицируем компоненты STARS, чтобы при клике вызывался callback и отрисовывалось необходимое количество звездочек

type StarPropsType = {  
 selected: boolean  
 value: 1 | 2 | 3 | 4 | 5  
 setValue: (value: 1 | 2 | 3 | 4 | 5)=>void  
}  
  
export function UnControlledRating() {  
  
 let [value, setValue] = useState(0)  
  
 return (  
  
 <div><h4>Stars choose(uncontrolled component)</h4>  
 <Star value={1} selected={value > 0} setValue={setValue}/>  
 <Star value={2} selected={value > 1} setValue={setValue}/>  
 <Star value={3} selected={value > 2} setValue={setValue}/>  
 <Star value={4} selected={value > 3} setValue={setValue}/>  
 <Star value={5} selected={value > 4} setValue={setValue}/>  
  
 </div>  
 )  
}  
  
  
function Star(props: StarPropsType) {  
 return <span onClick={()=>props.setValue(props.value)}>  
 {props.selected ? <b>star </b> : 'star '}  
 </span>  
}

**Callback позволяет общаться дочернему компоненту с родительским.**

либо второй вариант этого компонента

type StarPropsType = {  
 selected: boolean  
 setValue: ()=>void  
}  
export function UnControlledRating() {  
  
 let [value, setValue] = useState(0)  
  
 return (  
  
 <div><h4>Stars choose(uncontrolled component)</h4>  
 <Star selected={value > 0} **setValue={()=>setValue(1)}/>**  
 <Star selected={value > 1} **setValue={()=>setValue(2)}/>**  
 <Star selected={value > 2} **setValue={()=>setValue(3)}/>**  
 <Star selected={value > 3} **setValue={()=>setValue(4)}/>**  
 <Star selected={value > 4} **setValue={()=>setValue(5)}**/>  
  
 </div>  
 )  
}  
  
function Star(props: StarPropsType) {  
 **return <span onClick={()=>props.setValue()}>**  
 {props.selected ? <b>star </b> : 'star '}  
 </span>  
}

**Следующий пример:**

Задание: нужно повесить обработчик событий на заголовок меню, при нажатии меню будет сворачиваться/разворачиваться. Реализация ниже:

type AccordionPropsType = {  
 title: string  
  
}  
  
export function UnControlledAccordion(props: AccordionPropsType) {  
  
 let [collapsed, setCollapsed] = useState(true)  
  
 return (  
 <div>  
 <AccordionTitle titleValue={props.title}  
 onClick={() => {  
 setCollapsed(!collapsed)  
 }}/>  
 {!collapsed && <AccordionBody/>}  
 </div>)  
}  
  
type AccordionTitlePropsType = {  
 titleValue: string  
 onClick: () => void  
}  
  
function AccordionTitle(props: AccordionTitlePropsType) {  
 return (  
 <h3  
 onClick={() => props.onClick()}>{props.titleValue}</h3>  
 )  
}  
  
function AccordionBody() {  
 return (  
 <ul>  
 <li>1</li>  
 <li>2</li>  
 <li>3</li>  
 </ul>  
 )  
}

**КОНТРОЛИРУЕМЫЕ КОМПОНЕНТЫ (STATE зависит от PROPS)**

В данном случае state прописываем в родительском компоненте, из которого приходят PROPS.

function App() {  
  
 const [ratingValue, setRatingValue] = useState<RatingValueType>(0)  
  
 return (  
 <div className="App">  
  
 …  
 <Rating value={ratingValue} onClick={setRatingValue}/>  
 </div>  
 );  
}

Как в данном случае поменять STATE?

Мы отправили данные в Rating, затем из него в Star

export type RatingValueType = 0 | 1 | 2 | 3 | 4 | 5  
  
type RatingPropsType = {  
 value: number  
 onClick: (value:RatingValueType) => void  
}  
  
type StarPropsType = {  
 value: RatingValueType  
 selected: boolean  
 onClick: (value:RatingValueType) => void  
}  
  
export function Rating(props: RatingPropsType) {  
  
 return (  
 <div><h4>Stars(controlled components with props)</h4>  
 <Star selected={props.value > 0} onClick={props.onClick} value = {1}/>  
 <Star selected={props.value > 1} onClick={props.onClick} value = {2}/>  
 <Star selected={props.value > 2} onClick={props.onClick} value = {3}/>  
 <Star selected={props.value > 3} onClick={props.onClick} value = {4}/>  
 <Star selected={props.value > 4} onClick={props.onClick} value = {5}/>  
 </div>)  
}  
  
  
function Star(props: StarPropsType) {  
 return (<span onClick={() => props.onClick(props.value)}>  
 {props.selected ? <b>Star </b> : 'Star '}  
 </span>  
 )  
}

Все работает!

**БОЛЬШИНСТВО КОМПОНЕНТ НА ПРАКТИКЕ - КОТРОЛИРУЕМЫЕ!**

10. STORY BOOK (библиотека)

<https://storybook.js.org/>

Устанавливаем Storybook

**npx sb init**

**Запускаем**

NPM:

**npm run storybook**

YARN:

**yarn storybook**

В папке с компонентой создаем файл вида имя\_компонента.stories.tsx

В содержимое прокидываем например следующее:

import { Rating } from './Rating' // импорт компоненты

export default {

  title: 'Rating stories', // название сторибука для группы кейсов

  component: Rating, // имя компоненты

}

//здесь прописываем компоненты и передаем в них пропсы

export *const* EmptyRating = () *=>* <*Rating* value={0} onClick={*x=>x*}/>

export *const* Rating1 = () *=>* <*Rating* value={1} onClick={*x=>x*}/>

export *const* Rating2 = () *=>* <*Rating* value={2} onClick={*x=>x*}/>

export *const* Rating3 = () *=>* <*Rating* value={3} onClick={*x=>x*}/>

export *const* Rating4 = () *=>* <*Rating* value={4} onClick={*x=>x*}/>

export *const* Rating5 = () *=>* <*Rating* value={5} onClick={*x=>x*}/>

Можно написать что-то вроде тестов, т.е. добавить функцию событие onChange

import { useState } from 'react'

import { Rating, RatingValueType } from './Rating'

export default {

  title: 'Rating stories',

  component: Rating,

}

export *const* EmptyRating = () *=>* <*Rating* value={0} onClick={*x=>x*}/>

export *const* Rating1 = () *=>* <*Rating* value={1} onClick={*x=>x*}/>

export *const* Rating2 = () *=>* <*Rating* value={2} onClick={*x=>x*}/>

export *const* Rating3 = () *=>* <*Rating* value={3} onClick={*x=>x*}/>

export *const* Rating4 = () *=>* <*Rating* value={4} onClick={*x=>x*}/>

export *const* Rating5 = () *=>* <*Rating* value={5} onClick={*x=>x*}/>

export *const* RatingChanging = () *=>* {

*const* [rating, setRating] = useState<RatingValueType>(5)

  return <*Rating* value={rating} onClick={setRating}/>

}

Сделаем такую же Сторибук для компоненты OnOff:

* состояние, когда включен On
* состояние, когда включен Off
* кликабельное состояние для демонстрации поведения

import { useState } from 'react'

import { OnOff, OnOffPropsType } from './OnOff'

export default {

  title: 'OnOff stories',

  component: OnOff,

}

export *const* OnMode = () *=>* <*OnOff* on={true} onClick={*x=>x*}/> // когда ВКЛ

export *const* OffMode = () *=>* <*OnOff* on={false} onClick={*x=>x*}/> // когда ВЫКЛ

export *const* ModeChanging = () *=>* { // демонстрация поведения

*const* [value, setValue] = useState<*boolean*>(true)

  return <*OnOff* on={value} onClick={setValue}/>

}

Если компонента НЕКОНТРОЛИРУЕМАЯ, то ПРОС для состояний для СториБук создаем искусственно. В этом случае надо задавать defautValue и прокидывать в ПРОПС(см файл)

ИСПОЛЬЗОВАНИЕ TEMPLATES

Если мы хотим сделать истории нескольких компонент, можно использовать templates. Например, у нас есть такая компонента:

export *const* CollapsedMode = () *=>* {

  return (

    <*Accordion*

      title={'Accordion collapsed title'}

      onClick={*x* *=>* *x*}

      accordionCollapsed={true}

    />

  )

}

можно ее преобразовать в такую:

*const* Template: Story<AccordionPropsType> = *args* *=>* <*Accordion* {...args} />

*const* AccordionTemplateProps = { //вынесем отдельно повторияющиеся Props

  title: 'Accordion title',

  onClick: (*x*: *boolean*) *=>* *x*,

}

export *const* CollapsedMode2 = Template.bind({})

CollapsedMode2.args = {

  ...AccordionTemplateProps, // используем спред плюс добавляем неповторяющиеся props

  accordionCollapsed: true

}

КОММЕНТАРИИ К ИСТОРИЯМ (ДОКУМЕНТИРОВАНИЕ ПРОЕКТА)

Чтобы пользователь/дизайнер видел, какие пропсы за что отвечают, им нужно давать описание:

export *type* AccordionPropsType = {

  /\*\*

   \* Title, that will be shown in a new component instance

  \*/

  title: *string*

  /\*\*

   \* Value determines is Accordion collapsed or not

  \*/

  accordionCollapsed: *boolean*

  /\*\*

   \* Callback called when any item is clicked

  \*/

  onClick: (*value*: *boolean*) *=>* *void*

}

Тогда они будут видны в СториБуке как описание параметров, а также при наведении в IDE

СОЗЗДАЕМ СВОИ ГРУППЫ В ИСТОРИЯХ

Если мы хотим создать свою группу в сториз, можно прописать название следующим образом, тогда появится папа Components

export default {

  title: 'MyComponents/Accordion stories',

  component: Accordion

}

ЧИТАЙ ДОКУМНТАЦИЮ для TYPESCRIPT!

<https://storybook.js.org/>

12. CONTROLLED INPUT

Если мы просто создаем компоненту **<input/>** то она будет **неконтролируемой,** вводи, что хочешь.

export *const* UnControlledInput = () *=>* <input/>

Но даже в этом случае мы хотим знать, что введено в поле. Как мы можем трекать это значение?

Вариант 1) **использование useState()**

export *const* UnControlledInputWithTrackingValue = () *=>* {

*const* [value, setValue] = useState('')

  return (

    <>

      <input

        onChange={*event* *=>* {

*const* actualValue = *event*.currentTarget.value

          setValue(actualValue)

        }}/>{' '} - {value}

    </>

  )

}

Вариант 2) **с** **помощью хука useRef() - позволяет напрямую работать с DOM элементами.** Хук useRef не приводит к повторному перерендериванию компонента даже когда происходит изменение данных в нем. C другой стороны злоупотребление работы со ссылками в React компонентах будет вести к нарушению работы в виртуальном DOM, о чем всегда необходимо помнить.

export *const* UnControlledInputWithTrackingValueVar2 = () *=>* {

*const* [value, setValue] = useState('')

*const* inputRef = useRef<*HTMLInputElement*>(null)

*const* save = () *=>* {

*const* el = inputRef.current as *HTMLInputElement*

    setValue(el.value)

  }

  return (

    <>

      <input ref={inputRef}/> //указываем, из какого инпута получаем значение

      <button onClick={save}>save</button>- actual value: {value}

    </>

  )

}

Если мы зададим атрибут **<input value=”it incubator”/>,** то она станет **контролируемой** с фиксированным значением

export *const* ControlledInputWithFixedValues = () *=>* <input value={'it-incubator.by'}/>

13. Круговорот данных, FLUX

UI не может поменяться, если не поменяется state! Перерисовка UI может проходить только после изменения state! Даже если изменяется текст в инпуте, когда мы печатаем, ее надо приводить туда из стейта по событию OnChange, как в ToDolist

Например, создадим инпут, который будем контролировать через useState. Логика такая: когда происходит изменение состояния в поле по событию onChange, данные из ивента отправляются в Стейт, и уже стейт перерисовывает содержимое Инпута:

export *const* ControlledInput = () *=>* {

*const* [parentValue, setParentValue] = useState('')

*const* onChange = (*e*: ChangeEvent<*HTMLInputElement*>) *=>* {

    setParentValue(*e*.currentTarget.value)

  }

  return <input value={parentValue} onChange={onChange} />

}

Таким же образом сделаем для <INPUT type=”CHECKBOX”/>

export *const* ControlledCheckbox = () *=>* {

*const* [parentValue, setParentValue] = useState(false)

*const* onChange = (*e*: ChangeEvent<*HTMLInputElement*>) *=>* {

    setParentValue(*e*.currentTarget.checked)

  }

  return <input type='checkbox' checked={parentValue} onChange={onChange} />

}

Таким же образом сделаем для <SELECT>

export *const* ControlledSelect = () *=>* {

*const* [parentValue, setParentValue] = useState<*string*|*undefined*>('1')

*const* onChange = (*e*: ChangeEvent<*HTMLSelectElement*>) *=>* {

    setParentValue(*e*.currentTarget.value)

  }

  return (

    <select value={parentValue} onChange={onChange}> //вешаем onChange именно на select!!!

      <option value={'0'}>0</option>

      <option value={'1'}>1</option>

      <option value={'2'}>2</option>

    </select>

  )

}

Так мы сделали три КОНТРОЛИРУЕМЫЕ компоненты

14. Делаем Select component

Скрыть ненужную область данных можно:

* силами CSS (через display: none)
* путем изменения стейта с использованием условного рендеринга.

Выбирать нужно по месту, что удобнее, что быстрее и пр.

Для того, чтобы работал onKeyPress на div , нужно прописать TabIndex = 0 в атрибутах этого div. СТРЕЛКИ на onKeyPress не работают!!! Они относятся к скроллу!

15. useReducer()

**Если в локальном стейте один useState - используем useState**

**Если в локальном стейте два и более useState - используем useReducer**

**Типы action лучше выносить в отдельные константы:**

*const TOGGLE = ‘TOGGLE’*

Reducer - это функция преобразователь. В ней содержатся инструкция, как преобразовать стейт. Она говорит: дайте мне action(тип инструкции для преобразования стейта), и я преобразую стейт.

Если useState много, ими сложно управлять! Для этого нужен useReducer()

action:

type: ‘купить хлеб’

type: ‘уволить сотрудника’

Reducer: дайте мне action и я преобразую стейт

REDUCER при возврате должен создавать копию стейта!!! Это правило иммутабельности! Работать иначе не будет!

*type* ActionType = {

  type: *string*

}

*const* reducer = (*state*: *boolean*, *action*:ActionType) *=>* {

  //инструкции, как преобразовать стейт

  return *state* //возвращаем измененный стейт

}

Заменим используемый useState() на useReducer()

export *function* UnControlledAccordion (*props*: AccordionPropsType) {

  // let [collapsed, setCollapsed] = useState(

  //   props.defaultOn ? props.defaultOn : false

  // )

*let* [collapsed, dispatch] = useReducer(reducer, false)

**Первый аргумент** useReducer()- это функция редьюсер

**Второй аргумент** use Reducer() - исходное состояние стейта.

Хук возвращает массив пар значений:

**collapsed** - значение измененного стейта.

**dispatch()** - функция диспатч, которая позволяет отправить в реакт и инструкции **action:**

**dispatch (action)**

**Теперь в JSX мы вызываем dispatch при клике, в нем передаем тип action:**

 <*AccordionTitle*

        titleValue={*props*.title}

        onClick={() *=>* {

          dispatch({ type: 'TOGGLE-COLLAPSED' })

        }}

      />

      {!collapsed && <*AccordionBody* />}

16. HOC React.memo

(для увеличения скорости работы приложения)

**HOC** - high order component (компоненты высшего порядка). Принимает в себя одну компоненту, возвращает другую.

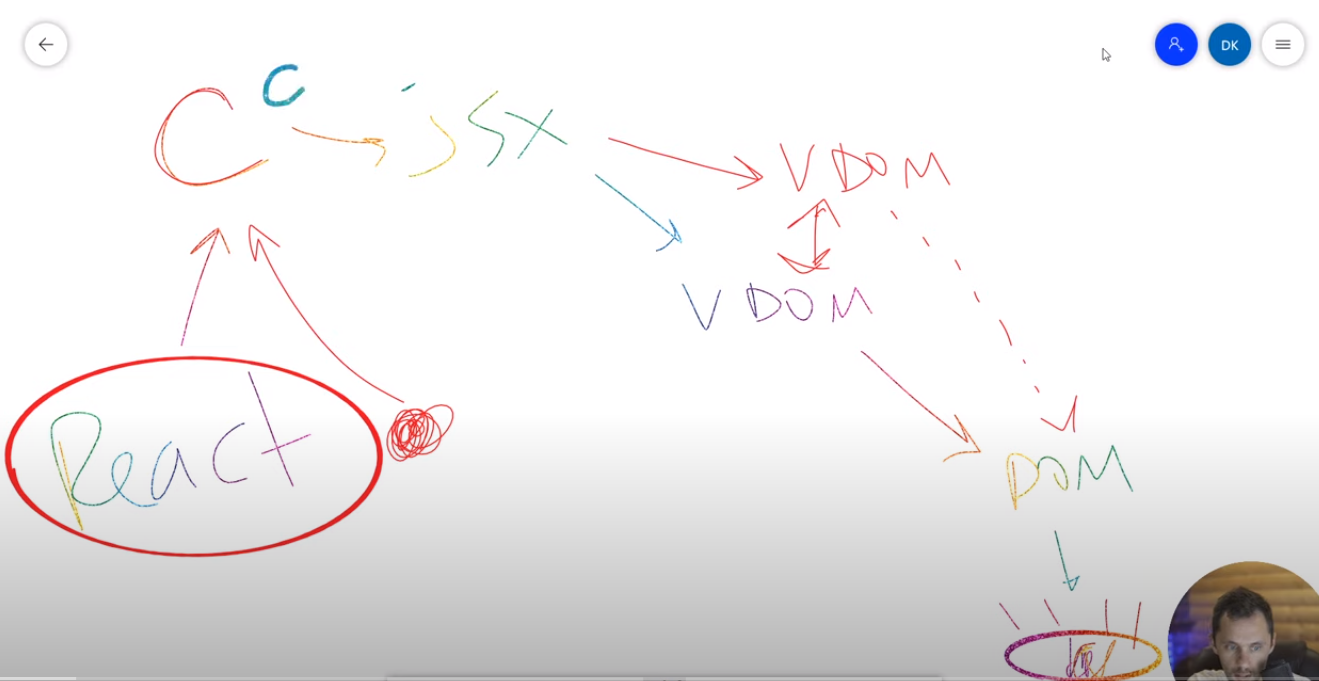
**Мемоизация** - это как кеширование. Запомни, а потом верни, что запомнил

Если ваш компонент всегда рендерит одно и то же при неменяющихся пропсах, вы можете обернуть его в вызов React.memo для повышения производительности

React.memo затрагивает только изменения пропсов. Если функциональный компонент обёрнут в React.memo и использует [useState](https://ru.reactjs.org/docs/hooks-state.html), [useReducer](https://ru.reactjs.org/docs/hooks-reference.html#usereducer) или [useContext](https://ru.reactjs.org/docs/hooks-reference.html" \l "usecontext), он будет повторно рендериться при изменении состояния или контекста.

Реакт при запуске приложения создает VirtualDOM, затем, при изменении стейта, он создает новый VDOM, сравнивает с предыдущим, и после этого отрисовывает обычный DOM, старую версию VDOM удаляет.

В больших приложениях сравнение VDOM между собой может требовать больших ресурсов:



**Суть работы React.memo.** Допустим, в родительском компоненте хранится стейт. В родительском компоненте есть два дочерних. Изменение состояние одного из дочерних компонентов приведет к сравнению VDOM большой компоненты. Чтобы не происходило перерисовки общей компоненты, сделан react.memo.

Допустим локальный стейт хранится в этой родительской компоненте

в данном случае при перерисовке setCounter будет заново отрисовываться (сравниваться) вся компонента

*const* NewMessagesCounter = (*props*: { counter: *number* }) *=>* {

  return <div>{*props*.counter}</div>

}

*const* Users = (*props*: { users: *Array*<*string*> }) *=>* {

  return (

    <>

      {*props*.users.map((*u*, *i*) *=>* (

        <div key={*i*}>{*u*}</div>

      ))}

    </>

  )

}

export *const* Example1 = () *=>* {

  //допустим локальный стейт хранится в этой родительской компоненте

  // в данном случае при перерисоке setCounter будет заново отрисовываться(сравниваться) вся компонента

*let* [counter, setCounter] = useState(0)

*let* [users, setUsers] = useState(['Alex', 'Stan', 'John'])

  return (

    <>

      <button

        onClick={() *=>* {

          setCounter(counter + 1)

        }}

      >

        Increment

      </button>

      <*NewMessagesCounter* counter={counter} />

      <*Users* users={users} />

    </>

  )

}

Обернем компоненту SecretUsers (она же users) в React.memo(). Она будет запоминать стейт(или props), и если он не будет меняться, компонента users не будет перерисовываться.

*const* NewMessagesCounter = (*props*: { counter: *number* }) *=>* {

  return <div>{*props*.counter}</div>

}

*const* SecretUsers = (*props*: { users: *Array*<*string*> }) *=>* {

  console.log('Users rendering')

  return (

    <>

      {*props*.users.map((*u*, *i*) *=>* (

        <div key={*i*}>{*u*}</div>

      ))}

    </>

  )

}

*const* Users = React.memo(SecretUsers)

export *const* Example1 = () *=>* {

  //допустим локальный стейт хранится в этой родительской компоненте

  // в данном случае при перерисоке setCounter будет заново отрисовываться(сравниваться) вся компонента

*let* [counter, setCounter] = useState(0)

*let* [users, setUsers] = useState(['Alex', 'Stan', 'John'])

  return (

    <>

      <button

        onClick={() *=>* {

          setCounter(counter + 1)

        }}

      >

        Increment

      </button>

      <*NewMessagesCounter* counter={counter} />

      <*Users* users={users} />

    </>

  )

}

**при этом будет работать обновление стейта Users при добавлении пользователя. В стейт нужно добавлять КОПИЮ ОБЪЕКТА, правило ИММУТАБЕЛЬНОСТИ**

export *const* Example1 = () *=>* {

  //допустим локальный стейт хранится в этой родительской компоненте

  // в данном случае при перерисоке setCounter будет заново отрисовываться(сравниваться) вся компонента

*let* [counter, setCounter] = useState(0)

*let* [users, setUsers] = useState(['Alex', 'Stan', 'John'])

// при этом будет работать обновление стейта Users при добавлении пользвателя.

// в стейт нужно добавлять КОПИЮ ОБЪЕКТА, правило ИММУТАБЕЛЬНОСТИ

*const* addUser = () *=>* {

*const* newUsers: *Array*<*string*> = [...users, 'Sveta']

    setUsers(newUsers)

  }

  return (

    <>

      <button

        onClick={() *=>* {

          setCounter(counter + 1)

        }}

      >

        Increment

      </button>

      <button onClick={addUser}>Add user</button>

      <*NewMessagesCounter* counter={counter} />

      <*Users* users={users} />

    </>

  )

}

17. useMemo() хук

(для увеличения скорости работы приложения)

1. **Используем для увеличения производительности для предотвращения пересчета данных**

При постоянно повторяющемся объеме информации, полученного, например, с сервера, лучше информацию **закешировать**. Но здесь важно понимать, что кеш может со временем стать **невалидным. В этом случае надо заново пересчитать информацию и закешировать заново. Т.е. если переменные поменялись => надо обновить кеш.**

**ContainerComponent = React.memo(Component)**. В реакт. мемо() мы передаем компоненту, он возвращает нам контейнерную компоненту.

**useMemo возвращает переменную!!!**

В данном примере при обновлении одной переменной, идет перерисовка всей компоненты. Это очень долго. Но зачем перерисовывать всю компоненту, если меняется только один параметр?

export *const* Example1 = () *=>* {

  // BLL

*const* [a, setA] = useState<*number*>(5)

*const* [b, setB] = useState<*number*>(5)

*let* resultA = 1

*let* resultB = 1

  // алгоритм расчета факториала с искусственно заданным сложным расчетом while

  for (*let* i = 1; i <= a; i++) {

*let* fake = 0

    while (fake < 10000000) {

      fake++

*let* fakeValue = Math.random()

    }

    resultA = resultA \* i

  }

  for (*let* i = 1; i <= b; i++) {

*let* fake = 0

    while (fake < 10000000) {

      fake++

*let* fakeValue = Math.random()

    }

    resultB = resultB \* i

  }

  // UI

  return (

    <div>

      <input value={a} onChange={*e* *=>* setA(*Number*(*e*.currentTarget.value))} />

      <input value={b} onChange={*e* *=>* setB(*Number*(*e*.currentTarget.value))} />

      <hr />

      <div>Result for a: {resultA}</div>

      <div>Result for b: {resultB}</div>

    </div>

  )

}

**Будем использовать хук useMemo**

**useMemo( () => {} , [a] )**

**Читается так: если поменялся а(или хотя бы одно из зависимостей зависимости), вызови функцию колл-бек(создающую функцию). Возвращает мемоизированное значение.**

функция, переданная useMemo, запускается во время рендеринга, в отличие от useEffect()!Функция, переданная в useEffect, будет запущена после того, как рендер будет зафиксирован на экране.

resultA = useMemo(() *=>* {

*let* tempResultA = 1

    for (*let* i = 1; i <= a; i++) {

*let* fake = 0

      while (fake < 10000000) {

        fake++

*let* fakeValue = Math.random()

      }

      tempResultA = tempResultA \* i

    }

    return tempResultA

  }, [a])

**Теперь при пересчете B будет возвращаться закешированное(мемоизированное) ранее значение resultA из useMemo. Это значительно ускорит работу компонента**

1. **Используем для корректной работы HOC React.memo**

**UseMemo() помогает HOC React.memo не ошибиться.**

**Например, мы хотим оградить users от перерисовок, как и примере с HOC React.memo**

**В данном случае просто использование HOC React.memo работать не будет, т.к. метод filter каждый раз возвращает НОВЫЙ МАССИВ. Будем использовать useMemo()!**

*const* newArray=useMemo(()*=>*{return users.filter(*item* *=>* *item*.indexOf('a') > -1)}, [users])

**Обязательно писать внутри функции RETURN!!**

18. useCallBack() хук

(возвращает функцию в соответствии с тем, изменилась ли зависимость)

**useCallback( f(), […]) говорит:**

запомни функцию f( ) и если зависимости не поменялись, верни эту функцию, которую я, например, передам другому ребенку и он не будет перерисовываться

**useCallback можно заменить использованием хука useMemo, разницу смотри в коде ниже**

Суть в том, что функция это тоже объект, и две функции с одинаковым кодом не равны друг другу.

Например, есть функция, описанная в Parent как const function() {…}. При перерисовке компоненты Parent будет пересоздаваться функция const function() {…}, а это вызовет перерисовку Child (т.е child будет получать один и тот же код, но будет также перерисовываться каждый раз, т.к. объекты не будут равны)

**Например, в родительской компоненте объявлена функция, ее мы передаем дочерней в пропсах. Дочерняя функция вызывает через колл-бек переданную ей от родителя функцию. Если родитель будет обновляться (useState(data)), то и дочерняя компонента тоже, т.к. в родительской компоненте пересоздается функция, которая передается в дочернюю компоненту! И это несмотря на использование HOC.React.memo!**

  // в случае с функцией можно использовать useMemo(),

  // будет сложный синтаксис

*const* memoisedAddBook = useMemo(() *=>* {

    return () *=>* {

      console.log('books rendered')

*const* newBooks = [...books, 'Angular' + new *Date*().getTime()]

      setBooks(newBooks)

    }

  }, [books])

  // но лучше использовать useCallBack()

*const* memoisedAddBook2 = useCallback(() *=>* {

    console.log('books rendered')

*const* newBooks = [...books, 'Angular' + new *Date*().getTime()]

    setBooks(newBooks)

  }, [books])

**!!! Если зависимость оставить пустой, по функция выполнится с исходным значением и больше его не поменяет (см. лексическое окружение)**

**КОГДА ИСПОЛЬЗОВАТЬ USECALLBACK?**

Всегда, когда нужно оптимизировать рендеринг, также и с HOC React.memo.

**В ЧЕМ РАЗНИЦА МЕЖДУ USECALLBACK() и USEMEMO()?**

Основная разница в том, что useCallback() возвращает мемоизированный колл-бек, а useMemo() возвращает мемоизированное значение, которое является результатом выполнения функцию Т.е. если работаешь с большим объемом данных - используй useMemo(). Если есть функция в родительской компоненте, которая передается в дочернюю через пропс, рендеринг родителя вызовет пересоздание функции и как следствие, рендериг дочерней компоненты. Чтобы это избежать, нужно использовать useCallback()

19. Навороченный useState()

Как только мы вызываем useState, компонета перерисовывается.

1. **Передача функции в useState(function) при сложных вычислениях initialState**

Если при передаче initial state в useState идут данные с большими вычислениями, то скорость перерисовки компоненты будет зависеть от скорости расчета Initial state. Чтобы это предотвратить, нужно передать в initial state функцию, которая вернет значение. Причем вернет оно его только один раз и перерисовываться не будет.

Передать можно только СИНХРОННУЮ ОПЕРАЦИЮ (функцию)

*function* generateData() {

  console.log('generateData');

  return 1513215113

}

export *const* Example1 = () *=>* {

*let* [counter, setCounter] = useState(generateData)

  return (

    <>

      <button onClick={() *=>* { setCounter(counter + 1) }}>Increment</button>

      {counter}

    </>

  )

}

**Это можно было решить с помощью useMemo, но не надо, т.к. в данном случае можно использовать useState!**

1. **Передача функции в useState(function) для изменения initialState (что-то вроде редьюсера)**

export *const* Example2 = () *=>* {

  console.log('render Example2');

*let* [counter, setCounter] = useState(0)

*const* changer = (*state*: *number*) *=>* {

    return *state* + 1

  }

  return (

    <>

      <button onClick={() *=>* { setCounter(changer) }}>Increment</button>

      {counter}

    </>

  )

}

20. UseEffect() основы

Мы не можем начать запрашивать данные с сервера, пока не отобразим компоненту (хоть что-нибудь пользователю, пустую разметку).

В классовой компоненте метод setState() заменяется на хук useState() в функциональной компоненте. Методы жизненного цикла в классовых компонентах (componentWillMount, componentDidMount и пр) заменяются на хук **useEffect()**

**В хук useEffect передается call-back? который называется эффектом или сайд-эффектом**

**useEffect( ()=>{} , [зависимость] )**

**Сайд-эффектом может быть:**

**- запрос на сервер (api.getUsers().then()) - асинхронный код**

**- setInterval - асинхронный код**

**- indexedDb (работа с локальной базой данных)**

**- обращение к DOM (document.getElementById)**

**- изменение titile документа и прочее.**

**Функция в useEffect запускается только после рендеринга компоненты!!!**

**ЗАВИСИМОСТИ:**

**Если вообще не передать зависимость useEffect(()=>{} ) в useEffect? то она будет перезапускаться каждый раз при изменении состояния. Если зависимость передать, то useEffect будет вызывать функцию только тогда, когда произойдут изменения в зависимости**

**Если передать ПУСТОЙ МАССИВ зависимостей useEffect( ()=>{}, [ ] ), то useEffect сработает только один раз при первой отрисовке компоненты!**

export *const* Example = () *=>* {

*let* [fake, setFake] = useState(1)

*let* [counter, setCounter] = useState(1)

  console.log('render Example');

  useEffect(() *=>* {

    console.log('use useEffect');

    document.title = counter.toString()

  }, [fake])

  return (

    <>

      <button onClick={() *=>* { setCounter(counter + 1) }}>Increment</button>

      <button onClick={() *=>* { setFake(fake + 1) }}>Increment</button>

      Counter: {counter}

      Fake: {fake}

    </>

  )

}

20. UseEffect() - setTimeout - setInterval.

Асинхронная логика

**Все асинхронные операции лучше закидывать в useEffect**, т.к. при каждом рендеринге компоненты будет перезапускаться асинхронный код. Например

export *const* SetTimeoutExample = () *=>* {

*let* [fake, setFake] = useState(1)

*let* [counter, setCounter] = useState(1)

  console.log('render setTimeoutExample');

  useEffect(() *=>* {

    console.log('use useEffect');

    setInterval(() *=>* {

      setCounter((*state*) *=>* *state* + 1)

    }, 1000)

  }, []) // пустой массив зависимостей - отрисуется только один раз

  return (

    <>

      {/\* <button onClick={() => { setCounter(counter + 1) }}>Increment</button>

      <button onClick={() => { setFake(fake + 1) }}>Increment</button> \*/}

      Counter: {counter}

      Fake: {fake}

    </>

  )

}

**При этом в функцию setTimeout в SetСounter передаем функцию-изменитель! Иначе произойдет замыкание на одном значении counter.**

**Для примера, напишем функцию, показывающую часы в реальном времени:**

export *const* ClockExample = () *=>* {

*let* date = new *Date*

*let* [clock, setClock] = useState('')

  useEffect(() *=>* {

    setInterval(() *=>* {

      setClock((*state*) *=>* {

        date = new *Date*

        return *state* = `Current time is...${date.getHours()}:${date.getMinutes()}:${date.getSeconds()}`

      })

    }, 1000)

  }, []) // пустой массив зависимостей - отрисуется только один раз

  return (

    <>{clock}</>

  )

}

22 - useEffect - компонента часы

**Создадим компоненту ЧАСЫ с use Effect**

*type* PropsType = {

}

*const* get2digitsString = (*num*: *number*) *=>* {

  return *num* < 10 ? "0" + *num* : *num*

}

export *const* Clock: React.FC<PropsType> = (*props*) *=>* {

*const* [date, setDate] = useState<*Date*>(new *Date*())

  useEffect(() *=>* {

*const* intervalId = setInterval(() *=>* { setDate(new *Date*()) }, 1000)

// убьем интервал перед закрытием компоненты

 return () *=>* {

      clearInterval(intervalId)

    }

  }, [])

  return <div>

    <span>{get2digitsString(date.getHours())}</span>

    :

    <span>{get2digitsString(date.getMinutes())}</span>

    :

    <span>{get2digitsString(date.getSeconds())}</span>

  </div>

}

setInteval() будет **работать вечно**, даже если прибить компоненту. У setInteval() есть свой ID, по этому id можно его прибить с помощью функции **clearInterval(id)** То же работает и с **setTimeout()**

**В классовых компонентах есть методы жизненного цикла componentWillUnmount(), и там мы можем убить интервал перед тем, как умрет компонента. В функциональной компоненте нужно использовать колл-бек в useEffect()**

24 - React кабзда как подробно: useEffect, cleanup (сброс, зачистка)

Запуск useEffect в зависимости от ЗАВИСИМОСТИ

**- Выполнится один раз при отрисовке компоненты:**

useEffect(() *=>* {

    console.log('useEffect occured');

  }, [])

**- Выполняется каждый раз при перерисовке компоненты (например, срабатывании useState)**

 useEffect(() *=>* {

    console.log('useEffect occured');

  }) // НЕТ МАССИВА ЗАВИСИМОСТЕЙ

**- Выполнится каждый раз при изменении зависимости a:**

useEffect(() *=>* {

    console.log('useEffect occured');

  }, [a])

Иногда нужно сбросить useEffect. Для этого внутрь useEffect пишем **return**

   useEffect(() *=>* {

   console.log('useEffect occured');

    return () *=>* {

      console.log('RESET EFFECT');

    }

  }, [])

В примере выше запустится RESET EFFECT, когда компонента будет демонтирована! а также, когда будет перерисована (есть массив зависимостей, компонента умирает каждый раз при перерисовке )

**Для чего нужна ЗАЧИСТКА useEffect()?**

- Когда мы запускаем setInterval, который нужно остановить при демонтаже компоненты.

- Когда мы делаем запрос на сервер, и не дожидаясь ответа от сервера демонтируем компоненту. В этом случае следует оборвать запрос на сервер.

- При добавлении (подписке) на нативные элементы, такие как скороллинг. Например, подпишемся на прослушку клавиш в объекте document. Если не пропишем return (не зачистим useEffect), то компонента будет слушать бесконечно! Обязательно пишем убиваем в return прослушку события:

  useEffect(() *=>* {

*const* handler = (*e*: *KeyboardEvent*) *=>* {

      console.log(*e*.key);

      setText((*state*) *=>* *state* + *e*.key)

    }

    window.addEventListener('keypress', handler)

    return () *=>* {

      window.removeEventListener('keypress', handler)

    }

  }, [])

либо так, с зависимостью text:

  useEffect(() *=>* {

*const* handler = (*e*: *KeyboardEvent*) *=>* {

      console.log(*e*.key);

      setText(text + *e*.key)

    }

    window.addEventListener('keypress', handler)

    return () *=>* {

      window.removeEventListener('keypress', handler)

    }

  }, [text])