Eslint

import js from '@eslint/js'

import globals from 'globals'

import reactHooks from 'eslint-plugin-react-hooks'

import reactRefresh from 'eslint-plugin-react-refresh'

export default [

  { ignores: ['dist'] },

  {

    files: ['\*\*/\*.{js,jsx}'],

    languageOptions: {

      ecmaVersion: 2020,

      globals: globals.browser,

      parserOptions: {

        ecmaVersion: 'latest',

        ecmaFeatures: { jsx: true },

        sourceType: 'module',

      },

    },

    plugins: {

      'react-hooks': reactHooks,

      'react-refresh': reactRefresh,

    },

    rules: {

      ...js.configs.recommended.rules,

      ...reactHooks.configs.recommended.rules,

      'no-unused-vars': ['error', { varsIgnorePattern: '^[A-Z\_]' }],

      'react-refresh/only-export-components': [

        'warn',

        { allowConstantExport: true },

      ],

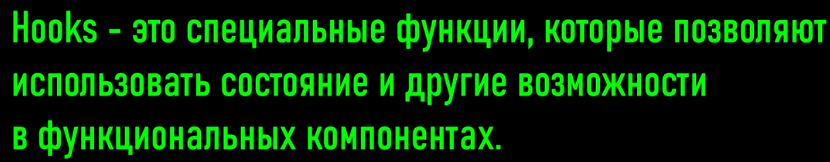
      'react/prop-types': 'off',

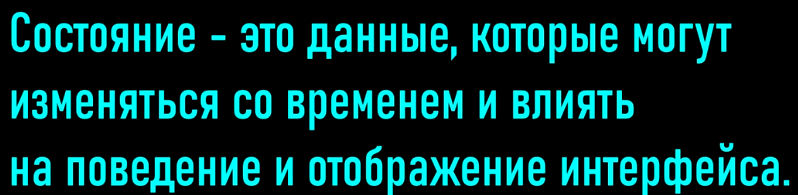
    },

  },

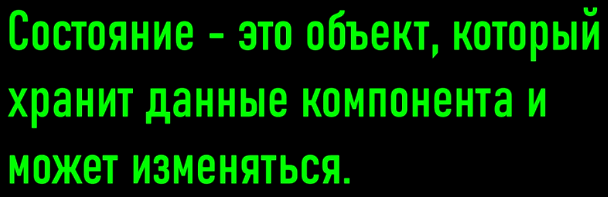
];

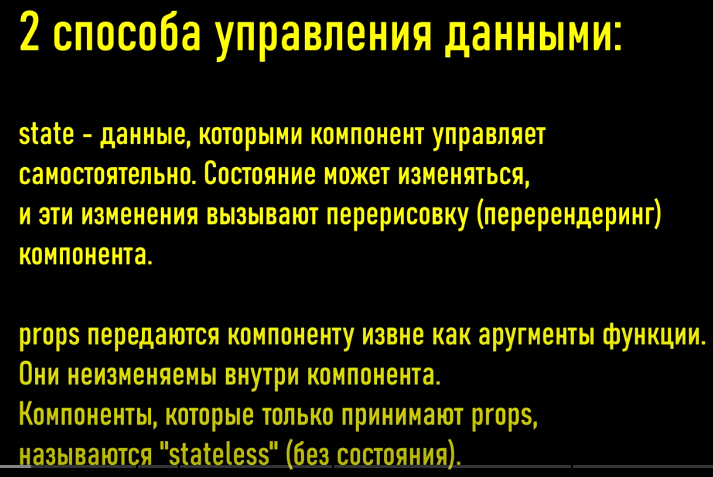
Выключаем поддержку ошибки типов.





В реакт состояние:





Функция обновления состояния работает асинхронно.

Состояние можно передавать только через общего родителя.

Функции обработчики событий должны начинаться через handle

**function HomePage({ clicks, setClicks }) {**

**const handleClick = () => {**

**alert('Click');**

**};**

**return (**

**<div className="home-page">**

**<Button label="Counter" callback={handleClick} />**

**<p>Количество кликов: {clicks}</p>**

**</div>**

**);**

**}**

**Условный рендеринг**

**if**

**function Greeting({ isLoggedIn }) {**

**if (isLoggedIn) {**

**return <h1>Добро пожаловать</h1>;**

**} else {**

**return <h1>Пожалуйста, войдите</h1>;**

**}**

**}**

**export default Greeting;**

**<Greeting isLoggedIn={true} />**

**<Greeting isLoggedIn={false} />**

**Тернарник**

**function Greeting({ isLoggedIn }) {**

**return (**

**<div>**

**{isLoggedIn ? (**

**<h1>Добро пожаловать</h1>**

**) : (**

**<h1>Пожалуйста, войдите</h1>**

**)}**

**</div>**

**);**

**}**

**Оператор двойного аперсанта**

**function Notification({ messages }) {**

**return (**

**<div>**

**{messages.length > 0 && <h1>У вас {messages.length} новых сообщений</h1>}**

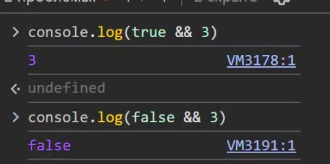
**</div>**

**);**

**}**

**export default Notification;**

**Используется для отображения элемнетов, если условие истинно.**

****

**Рендеринг списков react**

Ключ в элементе позволяет реакту идентифицировать какие элементы были добавлены, изменены, удалены.

Ключи помогают реакту оптимизировать процес рендеринга.

Ключи дожны быть строковыми литералами.

**const numbers = [1, 2, 3, 4, 5];**

**const listItems = numbers.map((number) => (**

**<li key={number.toString()}>{number}</li>**

**));**

**return <ul>{listItems}</ul>;**

**}**

Второй способ

**import { v4 as uuidv4 } from 'uuid';**

**function HomePage() {**

**const numbers = [1, 2, 3, 4, 5];**

**const listItems = numbers.map((number) => (**

// <li key={number.toString()}>{number}</li>

**<li key={uuidv4()}>**

**{number} - {uuidv4()}**

**</li>**

**));**

**return <ul>{listItems}</ul>;**

**}**

**Но лучше использовать id в сущностях.**

**function HomePage() {**

**const users = [**

**{ id: 1, name: 'Ivan' },**

**{ id: 2, name: 'Alice' },**

**{ id: 3, name: 'IvJoban' },**

**];**

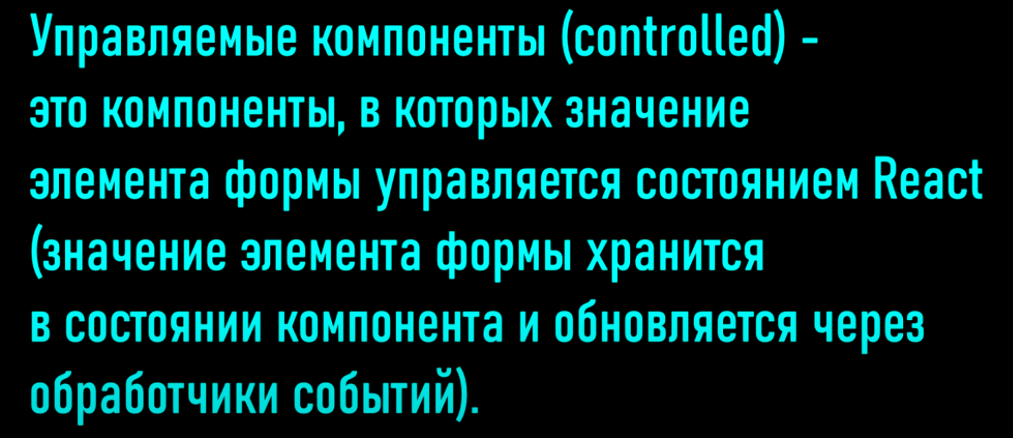
**const userList = users.map((user) => <li key={user.id}>{user.name}</li>);**

**return <ul>{userList}</ul>;**

**}**

**export default HomePage;**

У**правляемые и не управляемые компоненты в React**

****

**Пример управляемых  
import { useState } from 'react';**

**function ControlledForm() {**

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

**function handleSubmit(event) {**

**alert('Форма отправлена: ' + value);**

**}**

**function handleChange(event) {**

**setValue(event.target.value);**

**}**

**return (**

**<form action={handleSubmit}>**

**<label>**

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

**</label>**

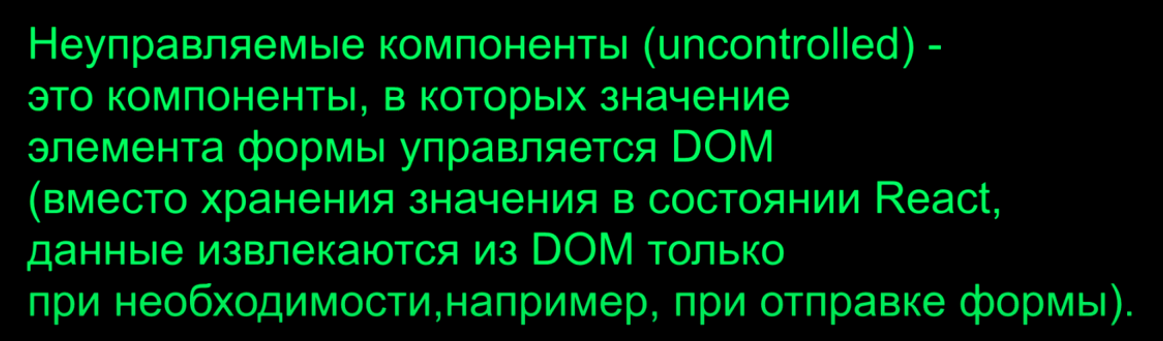
**<button type="submit">Отправить</button>**

**<p>{value}</p>**

**</form>**

**);**

**}**

**export default ControlledForm;**

**import { useState } from 'react';**

**function UnControlledForm() {**

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

**function handleSubmit(event) {**

**event.preventDefault();**

**const inputValue = event.target.elements.name.value;**

**alert('Форма отправлена: ' + inputValue);**

**}**

**function handleChange(event) {**

**setValue(event.target.value);**

**}**

**return (**

**<form onSubmit={handleSubmit}>**

**<label>**

**<input type="text" name="name" />**

**</label>**

**<button type="submit">Отправить</button>**

**</form>**

**);**

**}**

**export default UnControlledForm;**

**React form**

https://react-hook-form.com/

**npm install react-hook-form**

npm i react-scripts

[https://react-hook-form.com/get-started#Applyvalidation](https://react-hook-form.com/get-started" \l "Applyvalidation)

**Компоненты высшего порядка в React (HOC)**

Это функуции, которые принимают компонент и возвращают новый компонент с доп. логикой или пропсами.

Лучше в коде не использовать.

Замена: useContext, Redux.

**Children**

**props**

Один компонент может в себя содержать другие компоненты. И чтобы управлять содержимым внутри компонента — реакт предоставляет спец. props — children.

**function Container({children}) {**

**return <div>{children}</div>;**

**}**

**export default Container**

**function App() {**

**return (**

**<main className="app">**

**<Header />**

**<Container>**

**<h2>Это дочерний компонент</h2>**

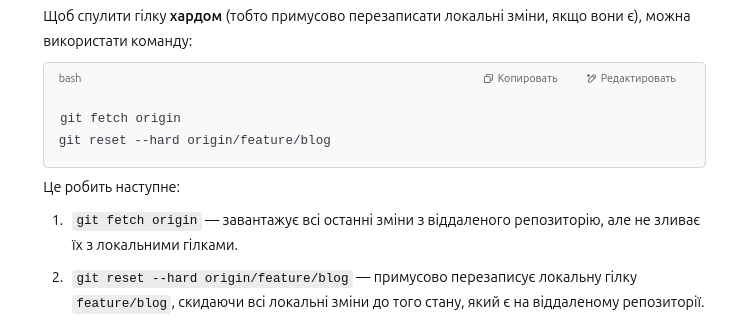
**</Container>**

**<Footer />**

**</main>**

**);**

**}**

**CSS Modules**

Все имена класов имеют локальную область видимости.

Имена класов преобразуются в уникальные id, чтобы избежать конфликтов.

**Используется для стилизации отдельные компонентов.**

composes — это специальный синтаксис, используемый в **CSS Modules**, который позволяет переиспользовать стили из других классов, даже если они находятся в отдельных файлах.

В данном случае:

css

КопироватьРедактировать

.button {

composes: base from './Base.module.css';

}

Этот код означает, что класс .button наследует (или **композирует**) стили из класса .base, который определён в файле Base.module.css.

**Зачем это нужно?**

* Позволяет **избежать дублирования кода**.
* Улучшает **структуру и поддержку стилей**.
* Можно **комбинировать** стили из нескольких классов.

**Styled components**

Также реализуется потход — css in js.

https://www.npmjs.com/package/styled-components

**npm i styled-components**

**Button.styled.js**

import styled from 'styled-components';

// создадим стилизованную кнопку

export const Button = styled.button`

padding: 10px 20px;

font-size: 16px;

border: none;

cursor: pointer;

width: 200px;

margin: 0 auto;

background-color: yellow;

`;

**HomePage.jsx**

import { Button } from '../components/StyledComponent/Button.styled';

const HomePage = () => {

return <Button>Кнопка 1</Button>;

};

**Выводы**: стили созданные с помощью styledComponents автоматически ограничиваются компонентом, которому они применяются. Это предотвращает конфликт стилей.

Также библиотека автоматически добавляет вендорные префиксы к css свойствам.

# Tailwind CSS + React.

Утилитарный css фреймворк, который позволяет создавать пользовательские интерфесы, используя готовые классы.

https://tailwindcss.com/

npm install tailwindcss @tailwindcss/vite

Нужно произвести изменения в конфигурационном файле.

import { defineConfig } from 'vite';

**import tailwindcss from '@tailwindcss/vite';**

import react from '@vitejs/plugin-react';

// https://vite.dev/config/

export default defineConfig({

plugins: [react(), **tailwindcss()**],

});

**main.css**

@import "tailwindcss";

**main.jsx**

import './main.css';

Tailwnd установлен.

Конвертор

https://tailwind-converter.netlify.app/

Импорт шрифтов с гугла:

https://tailwindcss.com/docs/font-family

@import url("https://fonts.googleapis.com/css2?family=Roboto&display=swap");

@import "tailwindcss";

@theme {

--font-display: 'Poppins', sans-serif;

}

Или так:

@theme {

--font-display: 'Poppins', sans-serif;

}

/\* body {

font-family: var(--font-display);

} \*/

<div className="bg-gray-900 mx-auto my-10 py-10 font-display container">

Вариант через font-face:

**@font-face {**

**font-family: Oswald;**

**font-style: normal;**

**font-weight: 200 700;**

**font-display: swap;**

**src: url("/fonts/Oswald.woff2") format("woff2");**

**}**

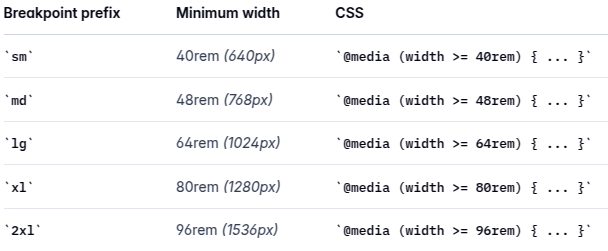
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Базовые стили Tailwind

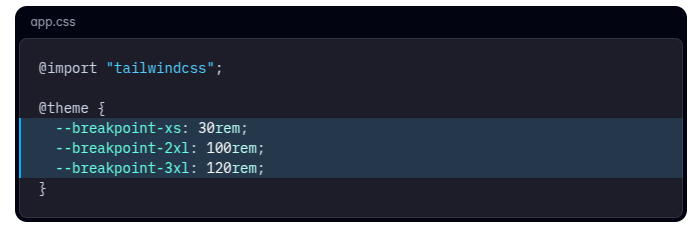
https://tailwindcss.com/docs/preflight

С коробки базовые стили уже установлены.

Breakpoints.



Можем переопределить breakpoints:



Можем указать кастомный брейкпоинт:  
--breakpoint-ddd: 666px;

Можем передать свои переменные:

@theme {

--breakpoint-xs: 30rem;

--breakpoint-2xl: 100rem;

--breakpoint-3xl: 120rem;

--breakpoint-ddd: 666px;

--color-vitos: red;

}

<h1 className="text-vitos">Hello</h1>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Переключение темы

@custom-variant dark (&:where([data-theme=dark], [data-theme=dark] \*));

**App.jsx**

const [theme, setTheme] = useState('light');

const toggleTheme = () => {

setTheme((prevTheme) => (prevTheme === 'light' ? 'dark' : 'light'));

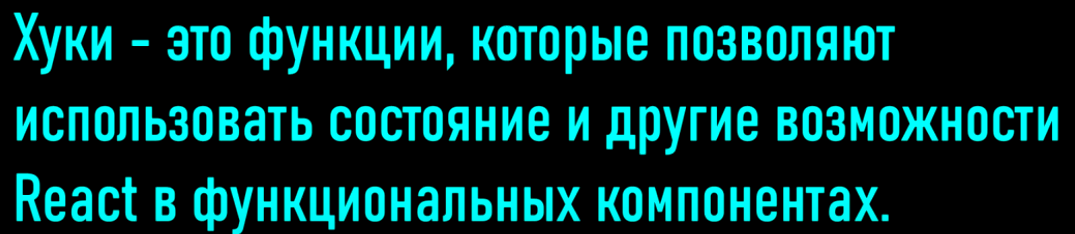
};

И в нужном месте:

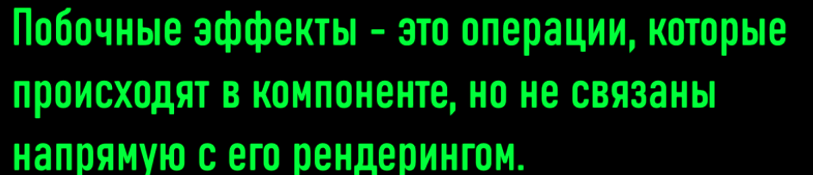
<h1 className="text-vitos **dark:text-white**">Hello</h1>

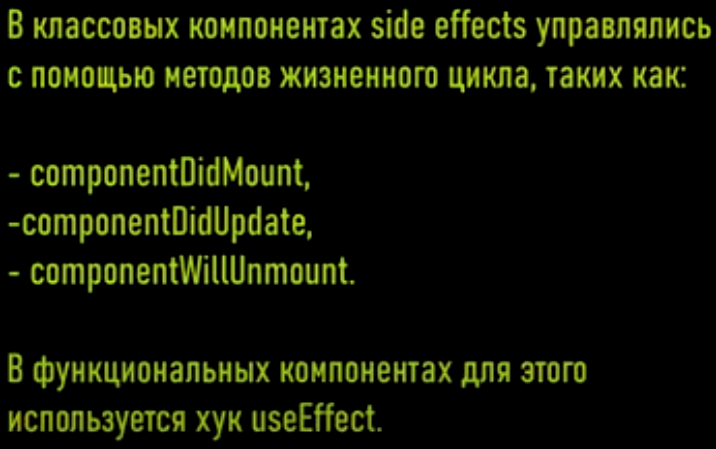
<h2 className="text-emerald-700 **dark:text-white**">h2</h2>

**Хуки**



**useEffexct** — хук, которые позволяет выполнять побочные эффекты в функциональных компонентах.





Side effect — запросы к API, подписки на события, изменение DOM вручную, таймеры, интервалы.

**Жизненный цикл компонента на примере с useEffect**

**1) Присходило монтирование, инициализировалось состояние useState:**

function DataFertchApi() {

const [data, setData] = useState([]);

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

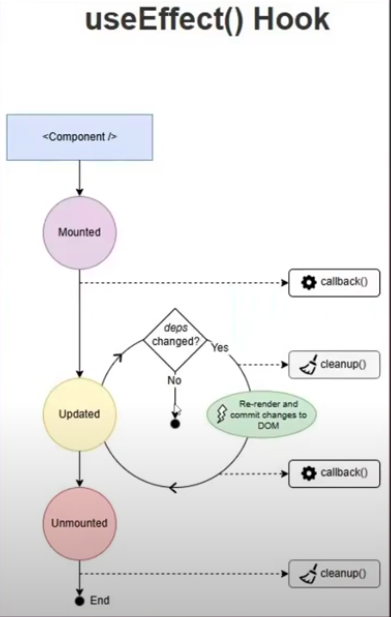
const [loading, setIsloading] = useState(true);

**2) Запускался useEffect. Тогда компонент обновлялся.**

**3) 3 этап. Размонитование компонента.**

Это процес при котором компонент удаляется из дерева компонентов. Тоесть из виртуального DOM. И его состояние и эффекты могут очищаться.

Это нужно когда компонент не нужен, тоесть пользователь переходит на другую страницу, или родит. Компонент обновился и не включает данный компонент.



**Размонтирование может поизойти в таких случаях:**

Например используем маршрутизацию, и когда переходит на новую страницу, то предыдущие компоненты будут размонтированы.

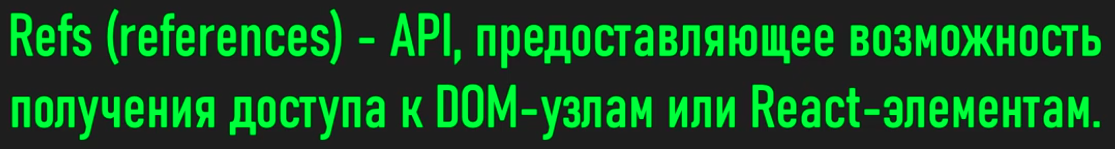
Или компонент будет размонтирован при ложном срабатывании тернарного оператора.

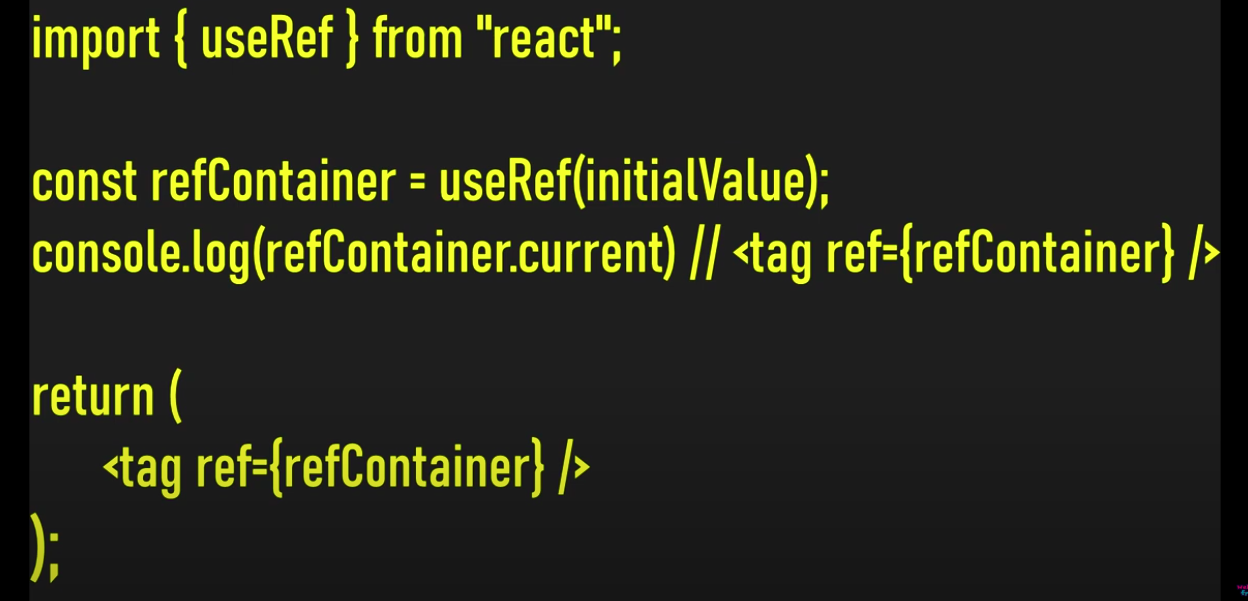
Размонтированиен может произойти в результате обновления состояния родительского компонента.

**Scrictmode** — режим, при котором реакт два раза монтирует и размонтирует компонент в режиме разработки, которые могут помочь обнаружить побочные эффектам, которые могут привести к утечки памяти или не предсказуемому поведению.

Следовательно компонент монитруется дважды и реает два раза создает таймер.

**useRef**

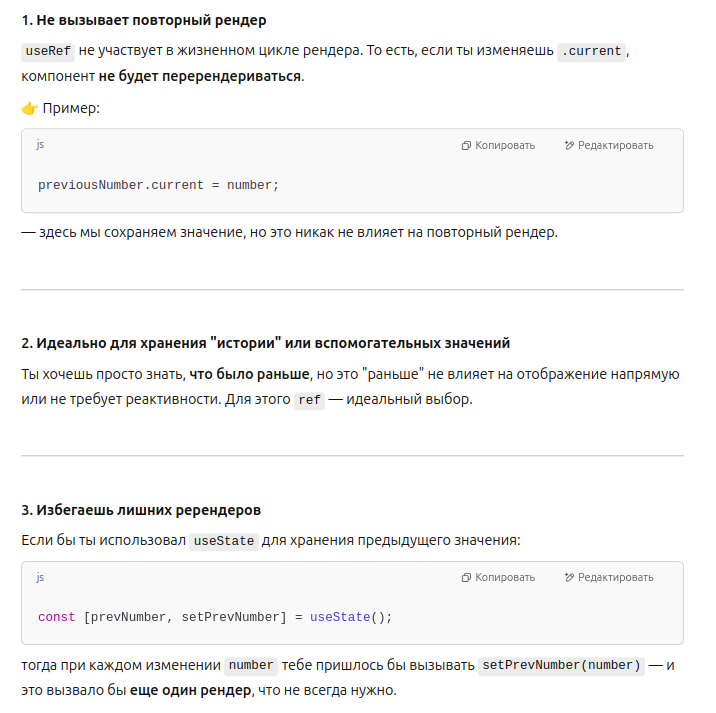


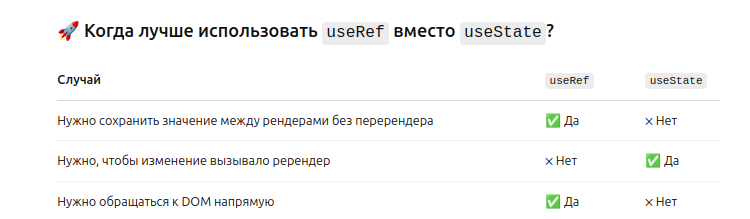


**Хранение предыдущего значения**

useRef в React позволяет **хранить данные между рендерами**, **не вызывая повторный рендер**.

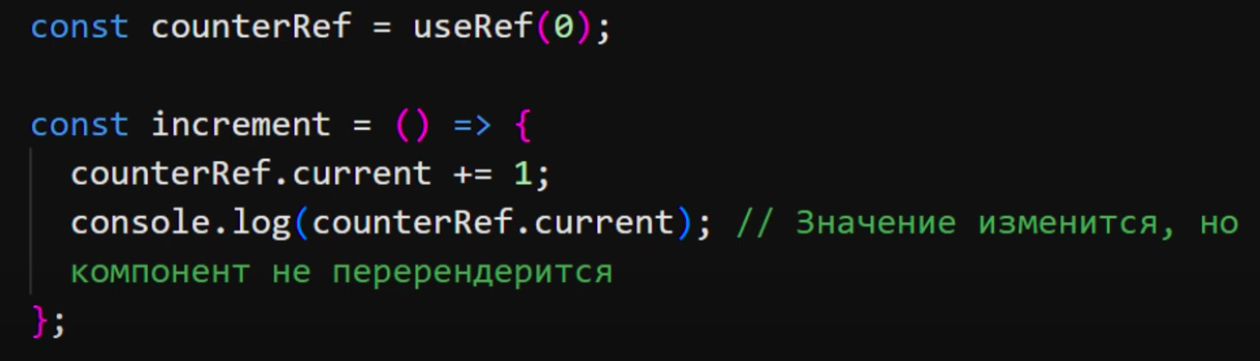
**Основные преимущества хранения предыдущего состояния через useRef:**

****

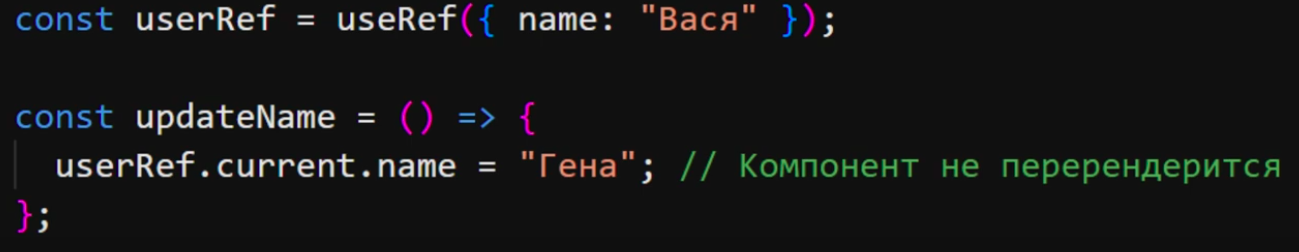
****

**Недостатки useRef**

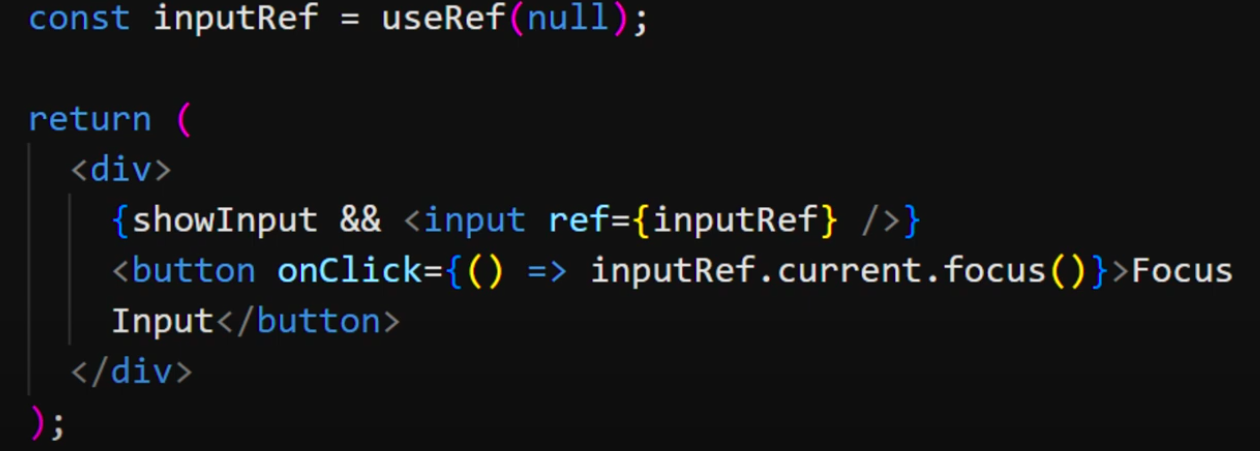
**Такой код не приведет к перерендыру компонента**

****

**useRef не потходит для хранения данных, которые должны быть реактивными**

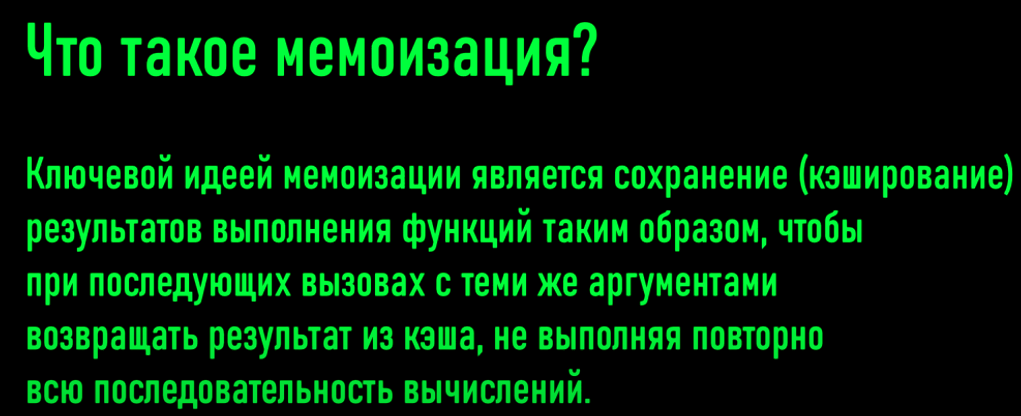
****

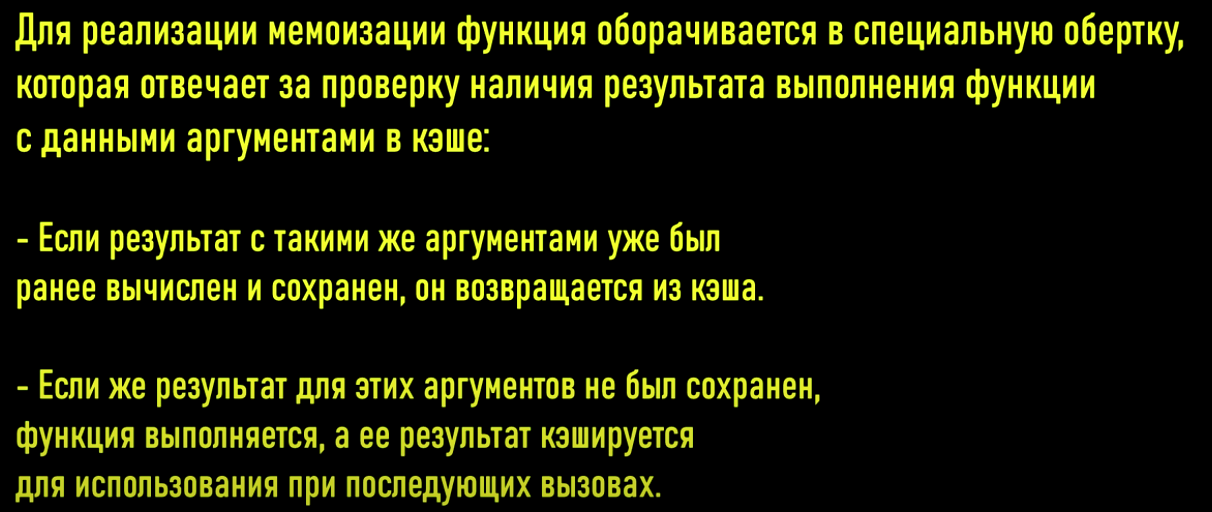
**useRef не потходит для условного рендеринга**

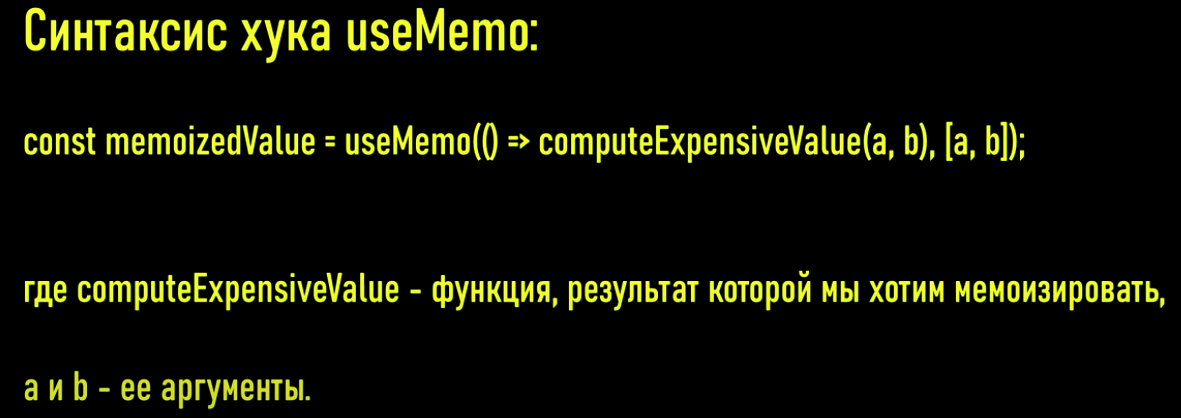
****

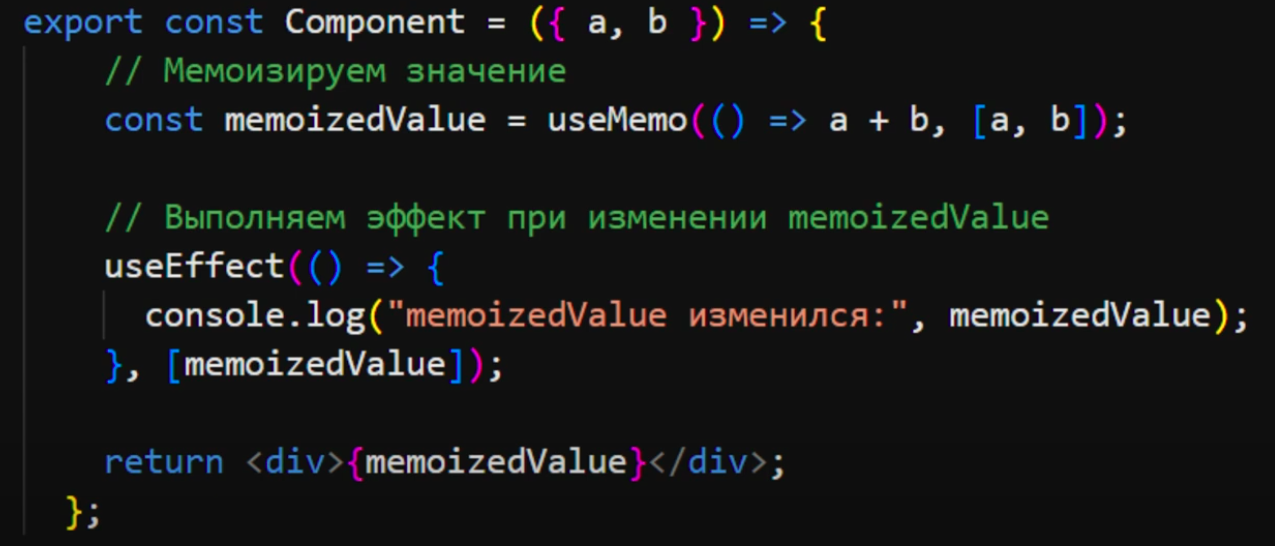
**useMemo**

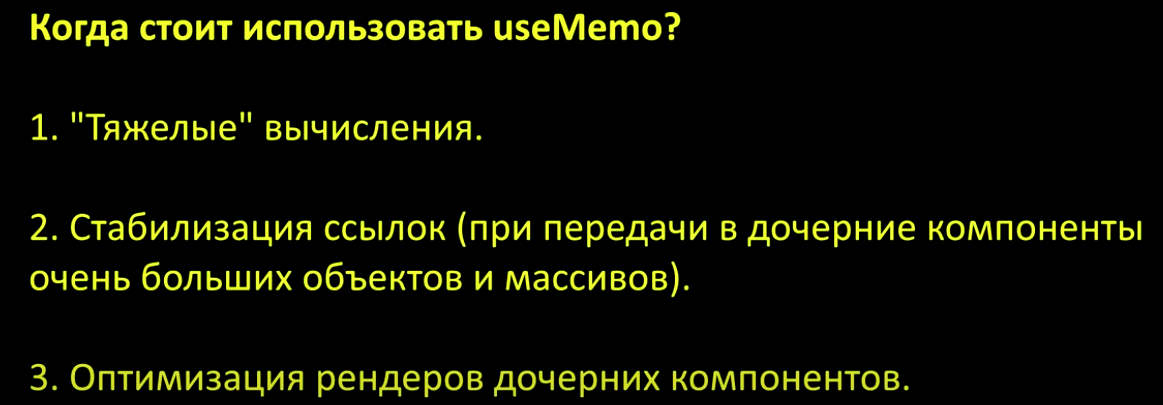
Для оптимизации вычислений и рендеринга компонента

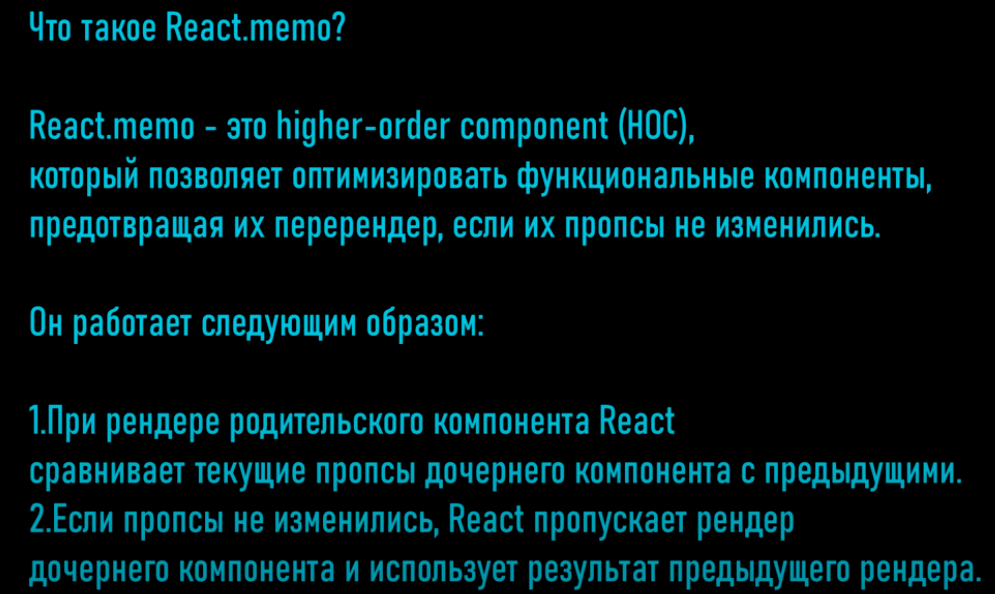










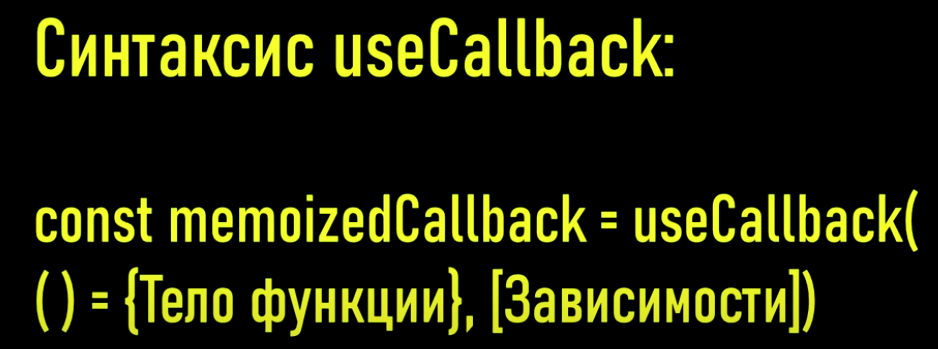


**useCallback**

**Хук useCallback в React используется для мемоизации функций, то есть чтобы не создавать одну и ту же функцию заново при каждом рендере компонента.**

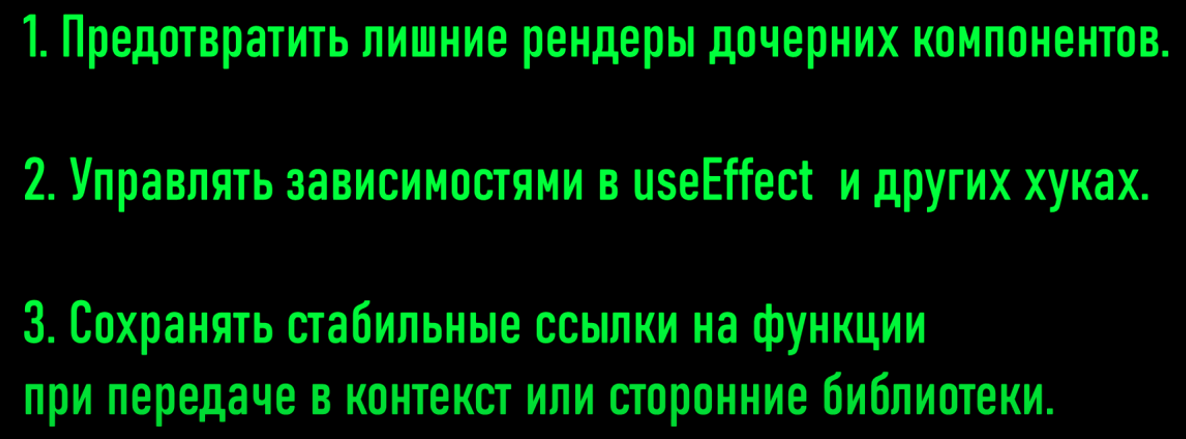
Он нужен тогда, когда:

* ты передаёшь функцию в дочерний компонент,
* или используешь её в зависимости каких-то useEffect, useMemo, и не хочешь, чтобы они срабатывали каждый раз без причины.

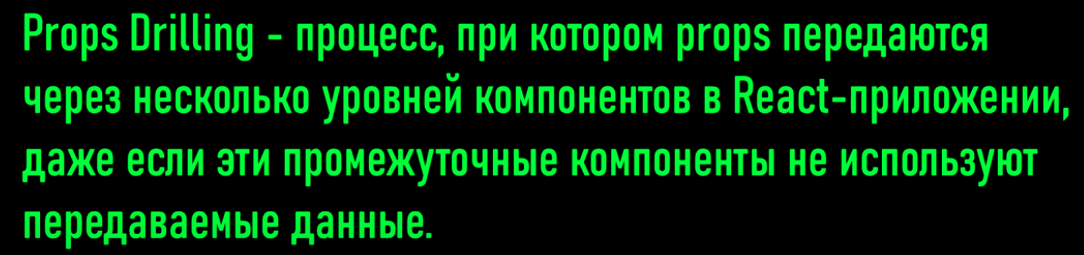
****

**Возвращаемая мемоизированная функция будет сохраняться до тех пор, пока не изменится какая либо переменная из списка зависимостей.**

**Таким образом функция не будет создаваться заново при каждом рендере.**

****

**Хук useState**

****