**Frontend questions**

**React**

https://metanit.com/web/react/1.1.php

https://my-js.org/docs/other/react-questions2/

<https://www.jscamp.app/ru/docs/interview/interview01/>

https://www.w3schools.com/react/

[Медиум вопросы](https://kanby.medium.com/вопросы-про-react-на-собеседовании-3bc1f4e021f4)

**Что такое React?**

React- это открытая (с открытым исходным кодом) JavaScript-библиотека для фронтенда, предназначенная для создания пользовательских интерфейсов, особенно, если речь идет о создании одностраничных приложений (Single Page Applications, SPA). Она отвечает за слой представления (view) в веб и мобильных приложениях. Reactбыл разработан [Jordan Walke](https://github.com/jordwalke)- инженером Facebook. Reactбыл представлен на Facebook News Feedв 2011 году, а для Instagram- в 2012 году.

React - это JavaScript-библиотека, предназначенная для создания быстрых и интерактивных пользовательских интерфейсов (user interfaces, UI) для веб- и мобильных приложений. Это открытая (с открытым исходным кодом), основанная на компонентах, библиотека для фронтенда, отвечающая только за слой представления (view) приложения.

Основная задача React - разработка быстрых пользовательских интерфейсов. В нем используется виртуальная объектная модель документа (virtual document object model (DOM) - программный интерфейс приложения (application programming interface (API)), для HTML и XML-документов. Он определяет логическую структуру документа, способы доступа к документу и управления им. Виртуальный DOM - это JavaScript-объект, что повышает производительность приложения. Виртуальный DOM быстрее обычного (браузерного, реального или настоящего). Мы можем использовать React как на стороне клиента, так и на стороне сервера, а также вместе с другими фреймворками. В нем используются компоненты и различные паттерны проектирования для работы с данными, что улучшает читаемость кода и облегчает поддержку больших приложений.

[Читать подробнее](https://reactjs.org/tutorial/tutorial.html)

React - это библиотека JavaScript, которая используется для создания пользовательского интерфейса. Первый релиз библиотеки увидел свет в марте 2013 года. Текущей версий на данный момент (март 2022 года) является версия React v18.0.

Первоначально React предназначался для веба, для создания веб-сайтов, однако позже появилась платформа React Native, которая уже предназначалась для мобильных устройств.

React представляется идеальный инструмент для создания масштабируемых веб-приложений (в данном случае речь идет о фронтенде), особенно в тех ситуациях, когда приложение представляет SPA (одностраничное приложение).

React относительно прост в освоении, имеет понятный и лаконичный синтаксис.

[**Перечислите особенности React?**](https://youtu.be/7TvS0iKR3_c?t=671)

**Виртуальный DOM**

Вся структура веб-страницы может быть представлена с помощью DOM (Document Object Model)- организация элементов html, которыми мы можем манипулировать, изменять, удалять или добавлять новые. Для взаимодействия с DOM применяется язык JavaScript. Однако когда мы пытаемся манипулировать html-элементами с помощью JavaScript, то мы можем столкнуться со снижением производительности, особенно при изменении большого количества элементов. А операции над элементами могут занять некоторое время, что неизбежно скажется на пользовательском опыте. Однако если бы мы работали из кода js с объектами JavaScript, то операции производились бы быстрее.

Для решения проблемы производительности как раз и появилась концепция виртуального DOM.

Виртуальный DOM представляет легковесную копию обычного DOM. И отличительной особенностью React является то, что данная библиотека работает именно с виртуальным DOM, а не обычным.

Если приложению нужно узнать информацию о состоянии элементов, то происходит обращение к виртуальному DOM.

Если необходимо изменить элементы веб-страницы, то изменения вначале вносятся в виртуальный DOM. Потом новое состояние виртуального DOM сравнивается с текущим состоянием. И если эти состояния различаются, то React находит минимальное количество манипуляций, которые необходимы до обновления реального DOM до нового состояния и производит их.

В итоге такая схема взаимодействия с элементами веб-страницы работает гораздо быстрее и эффективнее, чем если бы мы работали из JavaScript с DOM напрямую.

**Другие особенности React**

Другой отличительной чертой библиотеки является концентрация на компонентах - мы можем создать отдельные компоненты и затем их легко переносить из проекта в проект.

Еще одна особенность React - использование JSX. JSX представляет комбинацию кода JavaScript и XML и предоставляет простой и интуитивно понятный способ для определения кода визуального интерфейса.

React развивается как открытый проект, и все сайты библиотеки доступны на репозитории на github. Кроме того, при изучении Reacta также будет полезен официальный сайт с документацией - https://reactjs.org/, где можно найти всю информацию по библиотеке.

Основными особенностями React является следующее:

1. Использование VirtualDOM (виртуальной объектной модели документа) вместо RealDOM (настоящий или реальный DOM), поскольку манипуляции с RealDOM являются дорогостоящими с точки зрения производительности.
2. Поддержка рендеринга на стороне сервера (Server Side Rendering, SSR).
3. Следование принципу однонаправленного потока или связывания данных (one-directional data flow).
4. Использование переиспользуемых (reusable) компонентов пользовательского интерфейса (User Interface, UI) для формирования слоя представления.

### Опишите поток данных в React[​](https://my-js.org/docs/other/react-questions" \l "опишите-поток-данных-в-react)

React реализует однонаправленный реактивный поток данных (one-directional reactive data flow) с помощью пропов, что ускоряет рендеринг и легче в изучении, чем традиционное двустороннее связывание данных.

В реакте поток данных — однонаправленный. Это значит, что данные передаются как водопад, сверху вниз, от родителя к ребенку, через props. При этом props — неизменяемый объект, предназначенный только для чтения. Проще понять, если думать о компонентах, как о функциях (а они по сути ими и являются).

**Ключевые особенности React: декларативность, универсальность, компонентный подход, виртуальный DOM, JSX**

Одна из ключевых особенностей React — универсальность. Эту библиотеку можно использовать на сервере и на мобильных платформах с помощью React Native. Это принцип Learn Once, Write Anywhere или «Научитесь один раз, пишите где угодно».

Ещё одна важная особенность библиотеки — декларативность. С помощью React разработчик описывает, как компоненты интерфейса выглядят в разных состояниях. Декларативный подход сокращает код и делает его понятным.

React основан на компонентах, это ещё одна ключевая особенность библиотеки. Каждый компонент возвращает часть пользовательского интерфейса со своим состоянием. Объединяя компоненты, программист создаёт завершённый интерфейс веб-приложения.

Важная особенность React — использование JSX. Это расширение синтаксиса JavaScript, которое удобно использовать для описания интерфейса. JSX похож на HTML, тем не менее это всё-таки JavaScript. Пример JSX можно увидеть ниже:

const header = text ? <h1>{text}</h1> : null;

const vdom = (

<div>

{header}

<Hello />

</div>

);

JSX позволяет писать JavaScript-код с помощью готовых компонентов, которые практически полностью повторяют HTML. Это упрощает разработку.

К важным особенностям React относится использование виртуального DOM (Virtual DOM). Виртуальный DOM — объект, в котором хранится информация о состоянии интерфейса. При изменении состояния, например, после отправки формы или нажатия кнопки, React рассчитывает разницу между старым и новым состоянием. После этого библиотека отрисовывает новое состояние. Использование виртуального DOM позволяет библиотеке эффективно обновлять реальный DOM.

Компоненты (components) - это строительные блоки приложения. Обычно, приложение состоит из множества компонентов. Простыми словами, компонент - это класс или функция, принимающая свойства (props) и возвращающая React-элемент, описывающий, как должна выглядеть определенная часть пользовательского интерфейса.

Раньше прослеживалась четкая градация между компонентами с состоянием и без. В роли компонентов с состоянием выступали классовые компоненты, а в роли компонентов без состояния - функциональные компоненты. С появлением в React хуков, такой подход к дифференциации компонентов во многом утратил свою актуальность.

Компонент без состояния

*import* React *from* 'react'

*const* ExampleComponent = (props) => {

*return* <h1>Добро пожаловать в мир React!</h1>

}

*export* *default* *class* App *extends* React.Component {

render() {

*return* (

<div>

<ExampleComponent/>

</div>

)

}

}

В приведенном примере компонент без состояния ExampleComponent, возвращающий элемент h1, встраивается в компонент App.

Компонент с состоянием

*import* React *from* 'react'

*class* ExampleComponent *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = {

heading: "Заголовок компонента"

}

}

render() {

*return* (

<div>

<h1>{ *this*.props.welcomeMsg }</h1>

<h2>{ *this*.state.heading }</h2>

</div>

)

}

}

*export* *default* *class* App *extends* React.Component {

render() {

*const* welcomeMsg = "Добро пожаловать в мир React!"

*return* (

<div>

<ExampleComponent welcomeMsg={welcomeMsg}/>

</div>

)

}

}

В приведенном примере компонент с состоянием ExampleComponent, содержащий элементы h1 и h2, обернутые в div, встраивается в компонент App. Элемент h1 берет данные из пропов, а элемент h2 - из внутреннего состояния ExampleComponent.

Пропы

Пропы (props) - это опциональные входные данные, передаваемые из родительского компонента в дочерний. Для дочернего компонента они являются иммутабельными, т.е. доступными только для чтения. Они также могут использоваться для отображения статических значений.

Состояние

Состояние (state) - это некоторая информация о компоненте. Данная информация хранится и контролируется самим компонентом. При любом изменении состояния происходит повторный рендеринг компонента.

Жизненный цикл

Каждый компонент имеет методы жизненного цикла (lifecycle methods). Эти методы определяют поведение компонента на разных стадиях жизненного цикла (монтирование - встраивание в DOM, обновление, размонтирование - удаление из DOM).

**Что такое Virtual DOM? Как он работает с React?**

Разрабатывая клиентские приложения, команда разработчиков Facebook осознала, что DOM является медленным. Для того, чтобы сделать его быстрее, React использует виртуальный DOM, который, по сути, является представлением DOM-дерева в JavaScript. Когда возникает необходимость чтения или записи в DOM, используется данное представление. Затем виртуальный DOM пытается определить наиболее эффективный способ обновления реального DOM.

В отличие от DOM-элементов браузера, создание элементов в React обходится гораздо дешевле (с точки зрения производительности). Виртуальный DOM заботится об обновлении настоящего для полного совпадения с React-элементами. Это объясняется тем, что JavaScript очень быстрый, и хранение DOM-дерева в виде объекта ускоряет модификацию последнего.

**О виртуальном DOM см. вопрос об особенностях реакта.**

В React при каждом обновлении DOM или изменении данных, используемых страницей, создается новое объектное представление пользовательского интерфейса. Это всего лишь легковесная копия DOM.

Виртуальный DOM имеет почти такие же свойства, что и настоящий DOM, но он не может напрямую модифицировать содержимое (контент) страницы. Манипуляции с виртуальным DOM быстрее, поскольку он ничего не обновляет на экране. Простыми словами, работа с виртуальным DOM - это работа с копией реального DOM, не более того.

Обновление виртуального DOM быстрее благодаря следующему:

1. Использованию эффективного алгоритма определения различий (DOM diffing).
2. Совмещению (группировке, объединению) операций по обновлению.
3. Оптимизированному обновлению поддеревьев.
4. Использованию наблюдаемых (observable) объектов для определения изменений вместо "грязной" проверки (dirty check).

Как работает виртуальный DOM

При рендеринге JSX-элемента обновляется виртуальный DOM. Это происходит очень быстро. После обновления новая версия виртуального DOM сравнивается со старой (снимком (snapshot), выполненным перед обновлением). После этого, React определяет отличия между двумя объектами. Данный процесс называется diffing (определение различий).

Затем React обновляет только те части настоящего DOM, которые подверглись изменениям. За счет этого React очень сильно выигрывает в производительности.

Если кратко, то вот что происходит при обновлении DOM в React:

1. Обновляется виртуальный DOM.
2. Обновленный виртуальный DOM сравнивается с предыдущим.
3. Определяются различия между версиями виртуального DOM.
4. Обновляются только изменившиеся элементы реального DOM.
5. Изменения отображаются на экране.

[Подробнее](https://habr.com/ru/companies/macloud/articles/558682/)

<https://ru.legacy.reactjs.org/docs/faq-internals.html>

**Для чего нужен атрибут key при рендере списков?**

key - это специальный строковый атрибут, который следует использовать при создании списков элементов. Проп key помогает React определять, какие элементы подверглись изменениям, были добавлены или удалены.

Чаще всего в качестве ключа (key) мы используем идентификаторы: id

При отсутствии id в качестве ключа можно использовать индекс элемента.

Обратите внимание:

1. Использовать индексы в качестве ключей не рекомендуется, если порядок расположения элементов может измениться. Это может негативно сказаться на производительности, а также привести к проблемам с состоянием компонента.
2. При извлечении элемента списка в качестве самостоятельного компонента применяйте ключи к этим компонентам, а не к тегу li.
3. При отсутствии пропа key в консоль будет выведено соответствующее предупреждение.

Ключи (keys) помогают React определять, какие элементы подверглись изменениям, были добавлены или удалены. Проп key должен быть присвоен каждому элементу массива для обеспечения его стабильности.

Случаи безопасного использование индексов элементов в качестве ключей

* Список является статичным, т.е. никогда не меняется.
* Порядок расположения элементов также не меняется.
* Список не будет фильтроваться (добавление/удаление элементов из списка).
* Элементы не имеют идентификаторов или других уникальных значений.

Обратите внимание: использование индексов в качестве ключей может привести к непредсказуемому поведению компонента.

### Что такое списки?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-списки)

В JSX мы можем рендерить списки с помощью встроенного метода map. Данный метод часто используется для преобразования данных (он возвращает массив измененных каким-либо образом элементов).

Внутри map() следует использовать уникальные идентификаторы - ключи (keys), помогающие React обрабатывать обновление, добавление или удаление элементов. Это объясняет производительность React при работе с большими списками.

[https://ru.react.js.org/docs/reconciliation.html#recursing-on-children](https://ru.react.js.org/docs/reconciliation.html" \l "recursing-on-children)

**Что такое PureComponent?**

Чистые компоненты (pure components) - это компоненты, которые не рендерятся повторно при обновлении их состояния или пропов одними и теми же значениями. Если значение предыдущего и нового состояния и пропов равны, компонент не повторно отрисовывается. Чистые компоненты ограничивают повторный рендеринг, обеспечивая повышение производительности приложения.

Особенности чистых компонентов

* Предотвращают повторный рендеринг компонента, если его состояние и пропы остались прежними.
* Неявно реализуют метод shouldComponentUpdate.
* state и props сравниваются поверхностно (shallow).
* В ряде случаев такие компоненты являются более производительными, чем обычные.

По аналогии с чистыми функциями в JavaScript, React-компонент считается чистым, если он возвращает (рендерит) одинаковый результат для одних и тех же значений пропов и состояния. Для создания таких компонентов React предоставляет базовый класс PureComponent. Классовый компонент, расширяющий React.PureComponent, обрабатывается как чистый компонент.

Чистые компоненты похожи на обычные, за исключением того, что они неявно реализуют метод shouldComponentUpdate, проводя поверхностное сравнение состояния и пропов. Если текущие и следующие состояние и пропы являются одинаковыми, повторный рендеринг компонента не выполняется.

React-компоненты перерисовываются в следующих случаях:

1. В компоненте вызывается setState().
2. Обновляются значения props.
3. Вызывается forceUpdate().

Чистые компоненты не перерисовываются вслепую, без оценки значений state и props. Если обновленные значения аналогичны предыдущим, повторный рендеринг не запускается.

Компонент без состояния

*import* { pure } *from* 'recompose'

*export* *default* pure ((props) => <p>Компонент без состояния</p>)

Компонент с состоянием

*import* React, { PureComponent } *from* 'react'

*export* *default* *class* Test *extends* PureComponent{

render() {

*return* <p>Компонент с состоянием</p>

}

}

Пример

*class* Test *extends* React.PureComponent {

constructor(props) {

*super*(props)

*this*.state = {

taskList: [

{ title: 'Изучить React'},

{ title: 'Изучить TypeScript'},

{ title: 'Изучить Node.js'},

]

}

}

componentDidMount() {

setInterval(() => {

*this*.setState((oldState) =>

({ taskList: [...oldState.taskList] })

)

}, 1000)

}

render() {

console.log("Запущен рендеринг списка задач")

*return* (

<div>

{*this*.state.taskList.map((task, i) =>

<Task

key={i}

title={task.title}

/>

)}

</div>

)

}

}

*class* Task *extends* React.Component {

render() {

console.log("Задача добавлена в список")

*return* (

<h2>

{*this*.props.title}

</h2>

)

}

}

ReactDOM.render(<Test />, document.getElementById('app'))

### Зачем использовать "чистые" компоненты? Когда следует использовать "чистые" компоненты вместо обычных?[​](https://my-js.org/docs/other/react-questions2" \l "зачем-использовать-чистые-компоненты-когда-следует-использовать-чистые-компоненты-вместо-обычных)

Компоненты имеют один недостаток: они всегда повторно рендерятся вслед за родительским компонентом, даже если их пропы остались прежними.

Компоненты также всегда перерисовываются при обновлении их состояния, даже если новое значение состояние идентично предыдущему. При обновлении родительского компонента перерисовываются все его потомки, потомки потомков и т.д.

Такое поведение может привести к большому количеству лишних перерисовок. Действительно, если компонент зависит только от пропов и состояния, тогда он должен обновляться только при их изменении независимо от того, что происходит с его родительским компонентом.

Это как раз то, для чего предназначены "чистые" компоненты (pure components) - они останавливают "порочный круг" рендеринга. "Чистые" компоненты не перерисовываются до тех пор, пока не изменятся их пропы и состояние.

Случаи использования чистых компонентов:

* В целях предотвращения повторного рендеринга компонента, чьи состояние и пропы остались прежними.
* Состояние и пропы компонента являются иммутабельными.
* Мы не планируем реализовывать собственный метод жизненного цикла shouldComponentUpdate.

С другой стороны, мы не должны использовать PureComponent в случаях, когда:

* Состояние или пропы являются изменяемыми.
* Мы планируем реализовать собственный метод shouldComponentUpdate.

### **В чем разница между обычными и "чистыми" компонентами?[​](https://my-js.org/docs/other/react-questions2" \l "в-чем-разница-между-обычными-и-чистыми-компонентами)**

PureComponent - это то же самое, что Component, за исключением автоматической реализации метода shouldComponentUpdate. Главное отличие между "чистыми" и обычными компонентами состоит в поверхностном (shallow) сравнении состояний. Поверхностное сравнение означает, что примитивы сравниваются по значению, а объекты - по ссылке. Это повышает производительность кода.

Обычно дочерний компонент рендерится вслед за родительским при изменении состояния или пропов последнего. "Чистый" компонент не следует этому правилу и остается прежним до изменения его собственных props или state.

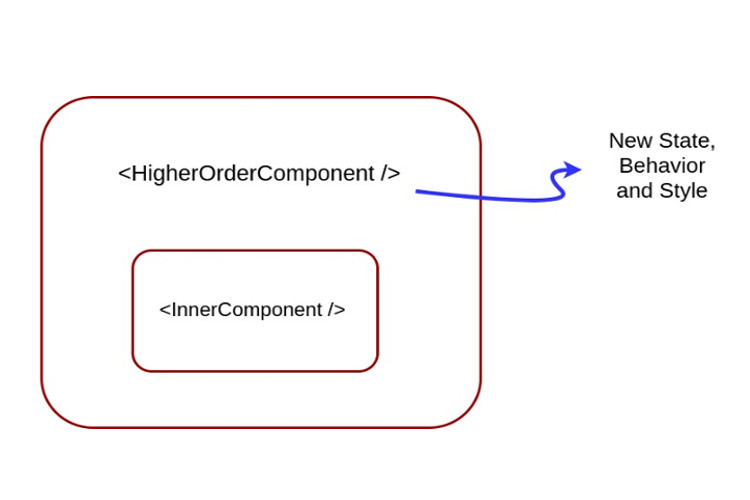
Условия использования "чистых" компонентов

* Состояние/пропы должны представлять собой иммутабельные объекты.
* Состояние/пропы не должны быть иерархическими структурами.
* При изменении данных следует вызывать forceUpdate().

**[Medium](https://medium.com/frontend-notes/purecomponent-и-components-5c15cf206ba7" \l ":~:text=Разница между Component и PureComponent заключается в методе updating lifecycle%3A shouldComponentUpdate.&text=PureComponent изначально определяет функцию%2C которая,продолжать updating lifecycle или нет.)**

**Что такое Компонент высшего порядка (Higher-Order Component/HOC)?**

### **Как создать компонент высшего порядка?[​](https://my-js.org/docs/other/react-questions2" \l "как-создать-компонент-высшего-порядка)**



Компонент высшего порядка (Higher Order Component, HOC) - это функция, принимающая компонент и возвращающая новый компонент. Это продвинутая техника, позволяющая повторно использовать логику компонента. HOC не являются частью React API. HOC является паттерном, производным от композиционной природы React. Компонент преобразует пропы в UI, а HOC трансформирует один компонент в другой. Примерами популярных HOC являются методы connect в Redux и createContainer в Relay.

*// HOC.js*

*import* React, { Component } *from* 'react'

*export* *default* *function* Hoc(WrappedComponent) {

*return* *class* extends Component {

render() {

*return* (

<div>

<WrappedComponent></WrappedComponent>

</div>

)

}

}

}

*// App.js*

*import* React, { Component } *from* 'react'

*import* Hoc *from* './HOC'

*class* App *extends* Component {

render() {

*return* (

<div>

Компонента высшего порядка

</div>

)

}

}

App = Hoc(App)

*export* *default* App

Обратите внимание:

* Мы не модифицируем компоненты, а создаем новые.
* HOC используются для композиции компонентов в целях обеспечения возможности повторного использования кода.
* HOC являются "чистыми" (pure) функциями. Они не имеют побочных эффектов (side effects) и всегда возвращают одинаковые результаты для одних и тех же аргументов.

Компонент высшего порядка (HOC) - это компонент, принимающий компонент в качестве аргумента и возвращающий новый компонент. HOC позволяют создавать композицию компонентов.

*import* React, { useEffect } *from* 'react'

*const* withLogging = (Component) => (props) => {

useEffect(() => {

fetch(`/logger?location=${window.location}`)

}, [])

*return* <Component {...props} />

}

*export* *default* withLogging

Мы можем комбинировать несколько HOC и обернуть ими каждую страницу:

*import* React *from* 'react'

*import* withAuth *from* './with-auth.js'

*import* withLogging *from* './with-logging.js'

*import* withLayout *from* './with-layout.js'

*const* page = compose(

withRedux,

withAuth,

withLogging,

withLayout('default')

)

*export* *default* page

Пример использования:

*import* page *from* '../hocs/page.js'

*import* MyPageComponent *from* './my-page-component.js'

*export* *default* page(MyPageComponent)

Higher-order component (HOC) - это функция, принимающая компонент и возвращающая новый компонент. Это паттерн, производный от композиционной природы React.

Мы называем такие компоненты чистыми, поскольку они могут принимать и динамически предоставлять дочерние компоненты, но не меняют и не копируют их поведение.

*const* EnhancedComponent = higherOrderComponent(WrappedComponent)

HOC обычно используются для:

1. Обеспечения возможности повторного использования кода, логики, а также для абстрагирования шаблонов.
2. Отложенного рендеринга.
3. Абстрагирования и манипуляции состоянием.
4. Манипуляции пропами.

### **Как в HOC создаются прокси для пропов?[​](https://my-js.org/docs/other/react-questions" \l "как-в-hoc-создаются-прокси-для-пропов)**

Пропы, передаваемые в компонент, можно добавлять/редактировать с помощью шаблона проксирования пропов следующим образом:

*function* HOC(WrappedComponent) {

*return* *class* Test *extends* Component {

render() {

*const* newProps = {

title: 'Новый заголовок',

footer: false,

showFeatureX: false,

showFeatureY: true

}

*return* <WrappedComponent {...*this*.props} {...newProps} />

}

}

}

[Documentation](https://reactdev.ru/types/054/" \l "hoc_2)

**Разница между управляемыми (controlled) и не управляемыми (uncontrolled) компонентами?**

### **Что такое управляемые компоненты?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-управляемые-компоненты)**

Компоненты, которые управляют инпутами формы для текущего пользователя, называются управляемыми. Любое изменение состояния имеет соответствующий обработчик.

Например, для того, чтобы значение было представлено прописными буквами (в верхнем регистре), мы используем такой обработчик:

handleChange(event) {

*this*.setState({ value: event.target.value.toUpperCase() })

}

### Что такое неуправляемые компоненты?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-неуправляемые-компоненты)

Неуправляемые компоненты - это компоненты, которые имеют собственное состояние. При необходимости получения значения их текущего состояния используются ссылки на DOM-элементы. Это больше похоже на обычный HTML.

В приведенном ниже примере мы получаем доступ к полю для ввода имени пользователя по ссылке:

*class* UserProfile *extends* React.Component {

constructor(props) {

*super*(props)

*this*.handleSubmit = *this*.handleSubmit.bind(*this*)

*this*.input = React.createRef()

}

handleSubmit(event) {

alert('Ваше имя: ' + *this*.input.current.value)

event.preventDefault()

}

render() {

*return* (

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

<label>

Имя:

<input type="text" ref={*this*.input} />

</label>

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

</form>

)

}

}

Для обработки форм рекомендуется использовать управляемые компоненты.

### **Как установить значение по умолчанию для неуправляемого компонента?[​](https://my-js.org/docs/other/react-questions" \l "как-установить-значение-по-умолчанию-для-неуправляемого-компонента)**

В React значение атрибута элемента формы перезаписывает соответствующее значение в DOM. В случае использования неуправляемых компонентов может потребоваться установить начальное значение, но оставить его неуправляемым при последующих обновлениях. Для решения этой задачи используется атрибут defaultValue вместо атрибута value:

render() {

*return* (

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

<label>

Имя пользователя:

<input

defaultValue="Иван"

type="text"

ref={*this*.input} />

</label>

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

</form>

)

}

То же самое справедливо в отношении полей select и textarea. Однако, для полей checkbox и radio следует использовать атрибут defaultChecked.

### Что такое управляемые и неуправляемые компоненты?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-управляемые-и-неуправляемые-компоненты)

В управляемых компонентах данные из полей формы обрабатываются React-компонентом. Альтернативой являются неуправляемые компоненты, где данные обрабатываются DOM.

Управляемые компоненты

В управляемых компонентах данные формы обрабатываются с помощью состояния компонента. Состояние компонента выступает в роли "единственного источника истины" (single source of truth) для "инпутов".

*import* React, { Component } *from* 'react'

*class* App *extends* Component {

state = {

message: ''

}

updateMessage = (newText) => {

console.log(newText)

*this*.setState(() => ({

message: newText

}))

}

render() {

*return* (

<div className="App">

<div className="container">

<input

type="text"

placeholder="Введите текст сообщения..."

value={*this*.state.message}

onChange={(event) => *this*.updateMessage(event.target.value)}

/>

<p>Сообщение: {*this*.state.message}</p>

</div>

</div>

)

}

}

*export* *default* App

Неуправляемые компоненты

Неуправляемые компоненты похожи на обычные HTML-элементы. Данные каждого инпута сохраняются в DOM, а не в компоненте. Вместо обработчиков событий для обновления состояния, в таких компонента используются refs (ссылки, рефы) для извлечения значений из узлов DOM. Ссылки позволяют получать доступ к узлам DOM или элементам React, создаваемым в методе render:

*import* React, { Component } *from* 'react'

*class* App *extends* Component {

constructor(props){

*super*(props)

*this*.handleChange = *this*.handleChange.bind(*this*)

*this*.input = React.createRef()

}

handleChange = (newText) => {

console.log(newText)

}

render() {

*return* (

<div className="App">

<div className="container">

<input type="text"

placeholder="Введите текст сообщения..."

ref={*this*.input}

onChange={(event) => *this*.handleChange(event.target.value)}

/>

</div>

</div>

)

}

}

*export* *default* App

Ключевые моменты

1. По возможности используйте контролируемые компоненты.
2. Контролируемые компоненты не требуют элемента формы, чтобы считаться контролируемым компонентом.
3. Если у компонента есть входной элемент, имеющий атрибут value, связанный с состоянием, и обработчик события для обновления указанного состояния, он является контролируемым компонентом.
4. Для страниц с большим количеством элементов ввода работа с контролируемыми компонентами может быть обременительной.
5. Поток данных является однонаправленным в контролируемых компонентах, причем состояние внутри компонента выступает в качестве единственного источника правды.
6. Все изменения состояния в контролируемом компоненте должны выполняться через функцию setState.
7. Неконтролируемые компоненты хранят свои данные в DOM как традиционный элемент ввода HTML.
8. **React.createRef()** используется для создания переменных экземпляра внутри неконтролируемых конструкторов компонентов. Эти переменные затем связываются с элементами ввода через атрибут **ref**.
9. Ссылки не могут быть созданы и использованы на функциональных компонентах.

<https://dev-gang.ru/article/kontroliruemye-i-nekontroliruemye-komponenty-v-react-lo65aat2do/>

**Методы жизненного цикла компонента в React?**

### Назовите методы жизненного цикла[​](https://my-js.org/docs/other/react-questions" \l "назовите-методы-жизненного-цикла)

До React 16.3:

* componentWillMount: выполняется перед рендерингом для настройки корневого компонента на уровне приложения;
* componentDidMount: выполняется после первого рендеринга, здесь выполняются AJAX-запросы, обновляется DOM или состояние компонента, регистрируются обработчики событий;
* componentWillReceiveProps: выполняется при обновлении любого пропа для запуска перехода состояния;
* shouldComponentUpdate: определяет, должен ли компонент обновляться. Значением по умолчанию является true. Если компонент не нуждается в повторном рендеринге при изменении состояния или пропов, можно вернуть false. Это подходящее место для улучшения производительности, позволяющее предотвратить ненужные рендеринги при получении компонентом новых пропов;
* componentWillUpdate: выполняется перед повторным рендерингом компонента при изменении состояния или пропов и при истинном значении, возвращаемом shouldComponentUpdate();
* componentDidUpdate: в основном, используется для обновления DOM в соответствии с изменениями состояния или пропов;
* componentWillUnmount: используется для отмены сетевых запросов или удаления обработчиков событий, связанных с компонентом.

React 16.3+:

* getDerivedStateFromProps: запускается перед вызовом метода render и при каждом повторном рендеринге. Он используется в редких случаях, когда требуется производное состояние (derived state). [Читать подробнее](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html);
* componentDidMount: выполняется после первого рендеринга, здесь выполняются AJAX-запросы, обновляется DOM или состояние компонента, регистрируются обработчики событий;
* shouldComponentUpdate: определяет, должен ли компонент обновляться. Значением по умолчанию является true. Если компонент не нуждается в повторном рендеринге при изменении состояния или пропов, можно вернуть false. Это подходящее место для улучшения производительности, позволяющее предотвратить ненужные рендеринги при получении компонентом новых пропов;
* getSnapshotBeforeUpdate: выполняется перед применением результатов рендеринга к DOM. Любое значение, возвращенное этим методом, передается в componentDidUpdate(). Это может быть полезным для получения информации из DOM, например, позиции курсора или величины прокрутки;
* componentDidUpdate: в основном, используется для обновления DOM в соответствии с изменением состояния или пропов. Не выполняется, если shouldComponentUpdate() возвращает false;
* componentWillUnmount используется для отмены сетевых запросов или удаления обработчиков событий, связанных с компонентом.

https://ru.react.js.org/docs/react-component.html

https://tproger.ru/translations/react-after-learning-basics/

https://massivepixel.io/blog/react-lifecycle-methods/

### Назовите методы жизненного цикла компонента, относящиеся к монтированию[​](https://my-js.org/docs/other/react-questions" \l "назовите-методы-жизненного-цикла-компонента-относящиеся-к-монтированию)

При создании и встраивании компонента в DOM методы жизненного цикла вызываются в следующем порядке:

1. constructor.
2. static getDerivedStateFromProps.
3. render.
4. componentDidMount.

### Какие методы жизненного цикла были признаны устаревшими в React 16?[​](https://my-js.org/docs/other/react-questions" \l "какие-методы-жизненного-цикла-были-признаны-устаревшими-в-react-16)

Следующие методы жизненного цикла являются небезопасными практиками кодирования и усложняют работу с асинхронной логикой:

1. componentWillMount.
2. componentWillReceiveProps.
3. componentWillUpdate.

Начиная с React 16.3, эти методы следует использовать с префиксом UNSAFE\_, а версии без префиксов были удалены в React 17.

### Для чего используется метод жизненного цикла getDerivedStateFromProps?[​](https://my-js.org/docs/other/react-questions" \l "для-чего-используется-метод-жизненного-цикла-getderivedstatefromprops)

Статический метод жизненного цикла getDerivedStateFromProps вызывается после инстанцирования элемента перед его повторным рендерингом. Он может возвращать объект для обновления состояния или null как индикатор того, что новые пропы не требуют обновления состояния.

*class* MyComponent *extends* React.Component {

*static* getDerivedStateFromProps(props, state) {

*// ...*

}

}

Этот метод вместе с componentDidUpdate() охватывает все случаи использования componentWillReceiveProps().

### Для чего используется метод жизненного цикла getSnapshotBeforeUpdate?[​](https://my-js.org/docs/other/react-questions" \l "для-чего-используется-метод-жизненного-цикла-getsnapshotbeforeupdate)

Метод жизненного цикла getSnapshotBeforeUpdate вызывается сразу после обновления DOM. Значение, возвращенное этим методом, передается в качестве третьего аргумента componentDidUpdate().

*class* MyComponent *extends* React.Component {

getSnapshotBeforeUpdate(prevProps, prevState) {

*// ...*

}

}

Этот метод вместе с componentDidUpdate() охватывает все случаи использования componentWillUpdate()

### В каком порядке вызываются методы жизненного цикла при повторном рендеринге компонента?[​](https://my-js.org/docs/other/react-questions" \l "в-каком-порядке-вызываются-методы-жизненного-цикла-при-повторном-рендеринге-компонента)

Обновление компонента может быть вызвано изменением пропов или состояния. При повторном рендеринге методы вызываются в следующим порядке:

1. static getDerivedStateFromProps.
2. shouldComponentUpdate.
3. render.
4. getSnapshotBeforeUpdate.
5. componentDidUpdate.

### Какие методы вызываются при обработке ошибок?[​](https://my-js.org/docs/other/react-questions" \l "какие-методы-вызываются-при-обработке-ошибок)

При возникновении ошибки в процессе рендеринга, в методе жизненного цикла, в конструкторе или любом потомке компонента, вызываются следующие методы:

1. static getDerivedStateFromError.
2. ComponentDidCatch.

### Как реализовать однократное выполнение компонентом некоторой операции при первоначальном рендеринге?[​](https://my-js.org/docs/other/react-questions2" \l "как-реализовать-однократное-выполнение-компонентом-некоторой-операции-при-первоначальном-рендеринге)

Для этого можно использовать метод жизненного цикла componentDidMount в классовом компоненте:

*class* Homepage *extends* React.Component {

componentDidMount() {

trackPageView('Homepage')

}

render() {

*return* <div>Домашняя страница</div>

}

}

Любые операции, определенные в componentDidMount(), будут выполнены только один раз при монтировании компонента.

Аналогичный функционал можно реализовать с помощью хука useEffect с пустым массивом зависимостей:

*const* Homepage = () => {

useEffect(() => {

trackPageView('Homepage')

}, [])

*return* <div>Домашняя страница</div>

}

useEffect() является более гибким, чем componentDidMount(). Он принимает два параметра:

* Первым параметром является функция обратного вызова, подлежащая выполнению.
* Опциональный второй параметр - массив, содержащий отслеживаемые переменные (массив зависимостей).

Второй параметр контролирует запуск эффекта:

* Если второй параметр отсутствует, эффект выполняется при каждом рендеринге.
* Если массив содержит переменные, то эффект запускается при монтировании компонента, а также при каждом изменении любой переменной.
* Если массив является пустым, то эффект будет запущен только один раз при монтировании компонента. В этом случае функционал будет аналогичен вызову componentDidMount() в классовом компоненте.

### Для чего предназначен метод жизненного цикла shouldComponentUpdate?[​](https://my-js.org/docs/other/react-questions2" \l "для-чего-предназначен-метод-жизненного-цикла-shouldcomponentupdate)

shouldComponentUpdate() позволяет остановить повторный рендеринг компонента, если в его обновлении нет необходимости. По умолчанию React не сравнивает пропы. При обновлении props или state React запускает процесс повторного рендеринга контента соответствующего компонента.

По умолчанию рассматриваемый метод возвращает true. Для того, чтобы остановить рендеринг "обновленного" компонента, необходимо вернуть false:

shouldComponentUpdate(nextProps, nextState) {

console.log(nextProps, nextState)

console.log(*this*.props, *this*.state)

*return* false

}

Предотвращение ненужного рендеринга

Метод shouldComponentUpdate - самый простой способ повысить производительность приложения. Он сравнивает текущие пропы и состояние со следующими и возвращает true, если они отличаются, и false, если они идентичны. Данный метод не вызывается при первоначальном рендеринге, а также при использовании forceUpdate().

### Для чего предназначена функция render?[​](https://my-js.org/docs/other/react-questions2" \l "для-чего-предназначена-функция-render)

Все React-приложения начинают с корневого узла DOM, который содержит ту часть DOM, которая управляется React. Когда React запускает процесс рендеринга, JSX конвертируется в обычный JavaScript. Функция render является частью жизненного цикла компонента. ReactDOM - это объект, содержащий метод render, который используется для встраивания JSX-контента в DOM.

Обычно, ReactDOM.render() используется для рендеринга компонента уровня приложения, все остальные компоненты являются его потомками. Результат рендеринга каждого компонента определяется в его методе render (в классовых компонентах). Любой JSX-код, содержащийся в этом методе, преобразуется в React.createElement(tag, props, children) перед рендерингом в DOM.

*// App.js*

*import* React *from* 'react'

*import* './App.css'

*function* App() {

*return* (

<div className="App">

Привет, народ!

</div>

)

}

*export* *default* App

*// index.js*

*import* React *from* 'react'

*import* ReactDOM *from* 'react-dom'

*import* './index.css'

*import* App *from* './App/App'

ReactDOM.render(

<React.StrictMode>

<App />

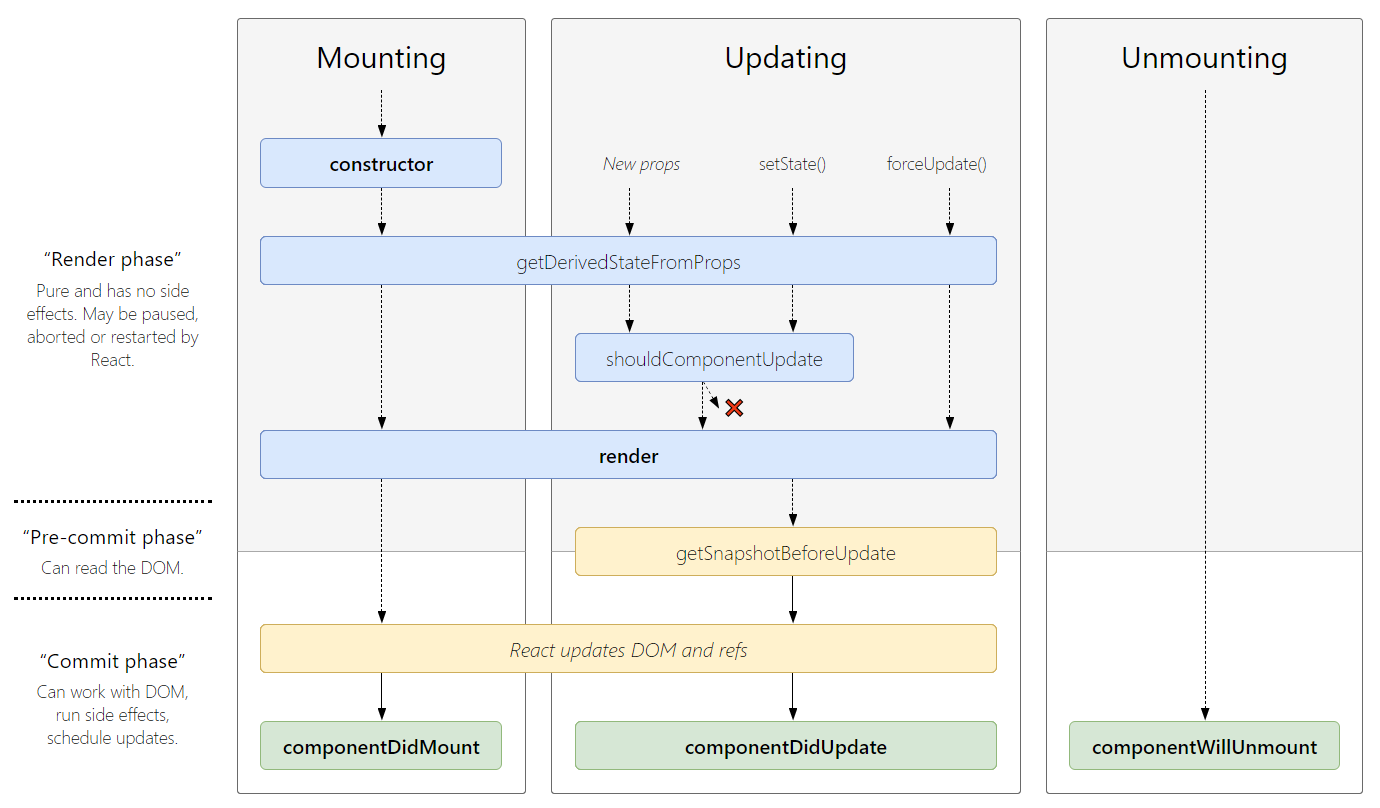
</React.StrictMode>,

document.getElementById('root')

)

**Стадии жизненного цикла компонента в React?**

### Назовите стадии жизненного цикла компонента[​](https://my-js.org/docs/other/react-questions2" \l "назовите-стадии-жизненного-цикла-компонента)



React предоставляет несколько методов, уведомляющих нас о происходящих процессах. Эти методы называются методами жизненного цикла (lifecycle methods) компонента. Они вызываются в определенном порядке. Жизненный цикл компонента делится на 4 стадии.

1. Mounting (монтирование).

При создании и встраивании компонента в DOM вызываются следующие методы:

* constructor()
* getDerivedStateFromProps()
* render()
* componentDidMount()

constructor()

constructor() вызывается при инициализации компонента. Это отличное место для присвоения начального значения объекту состояния и других настроек компонента.

constructor() вызывается с props в качестве аргумента и мы всегда должны начинать с вызова super(props), инициализирующего родительский конструктор, что позволяет потомку наследовать поля и методы предка (React.Component).

getDerivedStateFromProps()

getDerivedStateFromProps() вызывается сразу после рендеринга элемента в DOM. В качестве аргумента он принимает состояние и возвращает объект с его изменениями.

*class* Color *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { color: "red" }

}

*static* getDerivedStateFromProps(props, state) {

*return* { color: props.favourColor }

}

render() {

*return* (

<h1>Мой любимый цвет - {*this*.state.color}</h1>

)

}

}

ReactDOM.render(<Color favourColor="yellow"/>, document.getElementById('root'))

render()

Данный метод является единственным обязательным методом любого классового компонента. Как следует из его названия, он рендерит разметку для DOM.

componentDidMount()

componentDidMount() вызывается после монтирования компонента.

1. Updating (обновление).

Следующей стадией жизненного цикла компонента является его обновление. Компонент обновляется при изменении его состояния или пропов.

При обновлении компонента вызываются следующие методы:

* getDerivedStateFromProps()
* shouldComponentUpdate()
* render()
* getSnapshotBeforeUpdate()
* componentDidUpdate()

getDerivedStateFromProps()

Это первый метод, вызываемый при обновлении компонента. Это подходящее место для установки объекта состояния на основе начальных пропов.

*class* Color *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { color: "red" }

}

*static* getDerivedStateFromProps(props, state) {

*return* { color: props.favourColor }

}

changeColor = () => {

*this*.setState({ color: "blue" })

}

render() {

*return* (

<div>

<h1>Мой любимый цвет - {*this*.state.color}</h1>

<button type="button" onClick={*this*.changeColor}>Изменить цвет</button>

</div>

)

}

}

ReactDOM.render(<Color favourColor="yellow"/>, document.getElementById('root'))

shouldComponentUpdate()

shouldComponentUpdate() возвращает логическое значение, определяющее должен ли React продолжать рендеринг компонента. Значением по умолчанию является true.

*class* Color *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { color: "red" }

}

shouldComponentUpdate() {

*return* false

}

changeColor = () => {

*this*.setState({ color: "blue" })

}

render() {

*return* (

<div>

<h1>Мой любимый цвет - {*this*.state.color}</h1>

<button type="button" onClick={*this*.changeColor}>Изменить цвет</button>

</div>

)

}

}

ReactDOM.render(<Color />, document.getElementById('root'))

render()

Данный метод осуществляет повторный рендеринг компонента с учетом изменений.

getSnapshotBeforeUpdate()

В getSnapshotBeforeUpdate() мы получаем доступ к props и state компонента перед обновлением. Это означает, что даже после обновления мы можем увидеть, какими были пропы и состояние ранее.

getSnapshotBeforeUpdate() должен использоваться совместно с componentDidUpdate(). В противном случае, будет выброшено исключение.

*class* Color *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { color: "red" }

}

componentDidMount() {

setTimeout(() => {

*this*.setState({ color: "yellow" })

}, 1000)

}

getSnapshotBeforeUpdate(prevProps, prevState) {

document.getElementById("div1").innerHTML =

"Предыдущим значением значением 'color' было " + prevState.color

}

componentDidUpdate() {

document.getElementById("div2").innerHTML =

"Текущим значением 'color' является " + *this*.state.color

}

render() {

*return* (

<div>

<h1>Мой любимый цвет - {*this*.state.color}</h1>

<div id="div1"></div>

<div id="div2"></div>

</div>

)

}

}

ReactDOM.render(<Color />, document.getElementById('root'))

componentDidUpdate()

componentDidUpdate() вызывается после применения обновлений к DOM.

*class* Color *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { color: "red" }

}

componentDidMount() {

setTimeout(() => {

*this*.setState({ color: "yellow" })

}, 1000)

}

componentDidUpdate() {

document.getElementById("mydiv").innerHTML =

"Обновленным значением 'color' является " + *this*.state.color

}

render() {

*return* (

<div>

<h1>Мой любимый цвет - {*this*.state.color}</h1>

<div id="mydiv"></div>

</div>

)

}

}

ReactDOM.render(<Color />, document.getElementById('root'))

1. Unmounting (размонтирование)

Следующей стадией жизненного цикла компонента является его удаление из DOM или размонтирование в терминологии React.

componentWillUnmount()

Нажатие кнопки удаляет header:

*class* Container *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { show: true }

}

removeHeader = () => {

*this*.setState({ show: false })

}

render() {

*let* myHeader

*if* (*this*.state.show) {

myHeader = <Child />

}

*return* (

<div>

{myHeader}

<button type="button" onClick={*this*.removeHeader}>Удалить "шапку"</button>

</div>

)

}

}

*class* Child *extends* React.Component {

componentWillUnmount() {

alert("Размонтирование.")

}

render() {

*return* (

<h1>Привет, народ!</h1>

)

}

}

ReactDOM.render(<Container />, document.getElementById('root'))

<https://retool.com/blog/the-react-lifecycle-methods-and-hooks-explained/>

**Что такое React Reconciliation?**

Virtual DOM(VDOM) - это представление Real DOM, хранимое в памяти. Данное представление синхронизируется с настоящим DOM. Сравнение происходит между вызовом функции рендеринга и отображением элемента на экране. Данный внутренний процесс называется согласованием(reconciliation).

После изменения состояния или пропов компонента, React определяет, нуждается ли DOM в обновлении посредством сравнения нового элемента с предыдущим. Когда эти элементы отличаются, React обновляет DOM. Данный процесс называется согласованием (reconciliation).

### Для чего React сравнивает DOM?[​](https://my-js.org/docs/other/react-questions2" \l "для-чего-react-сравнивает-dom)

DOM (Document Object Model, объектная модель документа) - это интерфейс, представляющий HTML-документ в форме древовидной структуры (tree) с узлами (nodes). Эта структура позволяет разработчикам работать с узлами, представленными объектами, тем самым модифицируя документ. DOM формируется браузером при загрузке страницы.

Виртуальный DOM похож на браузерный (реальный или настоящий). Это древовидная структура, хранящаяся в памяти, с элементами, представленными в виде объектов. Данное дерево имеет почти все свойства браузерного DOM, за исключением возможности прямого изменения того, что отображается на экране. Это объект, содержащий компоненты приложения, предоставляющий возможность быстрого и эффективного обновления.

При рендеринге JSX-элемента или при изменении состояния элемента, создается новый виртуальный DOM. Функция, отвечающая за создание этого дерева, это функция render (в классовых компонентах). Процесс создания нового виртуального дерева является очень быстрым, поскольку это всего лишь объект и при этом UI не перерисовывается.

После создания виртуального DOM, React сравнивает это новое представление со снимком предыдущей версии для определения изменившихся элементов.

После определения различий, React обновляет только те объекты браузерного DOM, которые подверглись изменениям, после чего браузер перерисовывает эти объекты. После следующего изменения состояния или пропов компонента, создается новый виртуальный DOM и процесс повторяется.

Процесс определения различий между новым виртуальным DOM и старым называется diffing (определение различий). Для сравнения используется эвристический алгоритм со сложностью O(n). В процессе сравнения React вычисляет минимальное количество операций, необходимых для обновления браузерного DOM, игнорируя ненужные изменения. Данный процесс также именуется reconciliation (согласованием).

React реализует эвристический O(n) алгоритм, основываясь на двух допущениях:

1. Элементы разного типа приводят к формированию разных деревьев.
2. Разработчик может пометить определенные элементы как стабильные с помощью пропа key.

shouldComponentUpdate() позволяет остановить повторный рендеринг компонента, если в его обновлении нет необходимости. По умолчанию React не сравнивает пропы. При обновлении props или state React запускает процесс повторного рендеринга контента соответствующего компонента.

По умолчанию рассматриваемый метод возвращает true. Для того, чтобы остановить рендеринг "обновленного" компонента, необходимо вернуть false:

shouldComponentUpdate(nextProps, nextState) {

console.log(nextProps, nextState)

console.log(*this*.props, *this*.state)

*return* false

}

Предотвращение ненужного рендеринга

Метод shouldComponentUpdate - самый простой способ повысить производительность приложения. Он сравнивает текущие пропы и состояние со следующими и возвращает true, если они отличаются, и false, если они идентичны. Данный метод не вызывается при первоначальном рендеринге, а также при использовании forceUpdate().

### Для чего предназначена функция render?[​](https://my-js.org/docs/other/react-questions2" \l "для-чего-предназначена-функция-render)

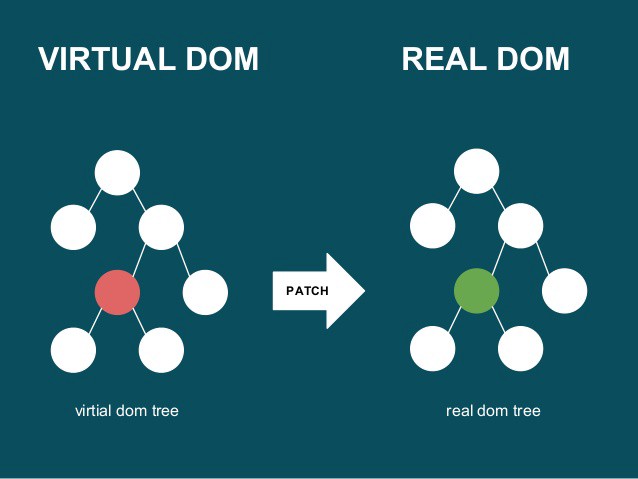
Все React-приложения начинают с корневого узла DOM, который содержит ту часть DOM, которая управляется React. Когда React запускает процесс рендеринга, JSX конвертируется в обычный JavaScript. Функция render является частью жизненного цикла компонента. ReactDOM - это объект, содержащий метод render, который используется для встраивания JSX-контента в DOM.

Обычно, ReactDOM.render() используется для рендеринга компонента уровня приложения, все остальные компоненты являются его потомками. Результат рендеринга каждого компонента определяется в его методе render (в классовых компонентах). Любой JSX-код, содержащийся в этом методе, преобразуется в React.createElement(tag, props, children) перед рендерингом в DOM.

### Что такое согласование?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-согласование)

Согласование (reconciliation) - это процесс, в ходе которого React обновляет DOM.

Разрабатывая приложение, мы создаем дерево компонентов. React берет это дерево, обрабатывает его и создает виртуальный DOM, который хранится в памяти. При обновлении приложения (т.е. при изменении state или props), React берет обновленный виртуальный DOM и сравнивает его с предыдущим. После этого, React определяет, что и каким образом изменилось. Наконец, изменения применяются к браузерному DOM в части касающейся. Это процедура повторяется снова и снова.



Синхронизация виртуального DOM с браузерным обеспечивается библиотекой ReactDOM. React должен проводить сравнение деревьев очень быстро, поэтому он использует эвристический алгоритм со сложностью O(n). Такая сложность выполнения алгоритма означает, что если у нас имеется 1000 узлов, нам потребуется выполнить всего 1000 сравнений. Данный алгоритм является улучшенной версией алгоритма со сложностью O(n\^3) => для 1000 узлов потребуется 1 млрд. сравнений.

Пример

Создадим простой компонент, в котором складываются два числа. Числа вводятся в соответствующие поля:

*class* App *extends* React.Component {

state = {

result: '',

entry1: '',

entry2: ''

}

handleEntry1 = (event) => {

*this*.setState({ entry1: event.target.value })

}

handleEntry2 = (event) => {

*this*.setState({ entry2: event.target.value })

}

handleAddition = (event) => {

*const* firstInt = parseInt(*this*.state.entry1)

*const* secondInt = parseInt(*this*.state.entry2)

*this*.setState({ result: firstInt + secondInt })

}

render() {

*const* { entry1, entry2, result } = *this*.state

*return*(

<div>

<div>

Результат: { result }

</div>

<span><input type='text' value={entry1} onChange={*this*.handleEntry1} /></span>

<br />

<br />

<span><input type='text' value={entry2} onChange={*this*.handleEntry2} /></span>

<div>

<button onClick={*this*.handleAddition} type='submit'>Сложить</button>

</div>

</div>

)

}

}

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

При вводе первого числа React создает новое дерево компонентов. Это дерево (виртуальный DOM) содержит новое значение состояния для свойства entry1. Затем React сравнивает новое дерево со старым, определяет различия и применяет их к браузерному DOM. Новое дерево создается при каждом изменении состояния - при вводе чисел в поля и при нажатии кнопки.

https://unetway.com/tutorial/reactjs-reconciliation-sverka

<https://habr.com/ru/articles/662549/>

https://reactdev.ru/handbook/reconciliation/

**Что такое портал (Portal)?**

Порталы (portals) предоставляют быстрый и удобный способ рендеринга дочернего компонента в узле DOM, находящимся за пределами иерархии DOM родительского компонента.

Обычно, компонент рендерит дерево элементов (как правило, с помощью JSX), React-элемент определяет, как должен выглядеть DOM родительского компонента.

ReactDOM.createPortal(child, container)

Возможности

* Дочерний компонент передается узлу, находящемуся в другом поддереве компонентов.
* По умолчанию таким узлом является document.body, но им может быть любой существующий DOM-элемент.
* Поддерживается рендеринг на стороне сервера.
* Массивы не требуется оборачивать в div или React.Fragment.
* Используются компоненты <Portal /> и <PortalWithState />.

Случаи использования

* Модальные окна.
* Всплывающие подсказки.
* Раскрывающиеся меню.
* Виджеты и т.д.

Пример

*// App.js*

*import* React, { Component } *from* 'react'

*import* './App.css'

*import* PortalDemo *from* './PortalDemo.js'

*class* App *extends* Component {

render () {

*return* (

<div className='App'>

<PortalDemo />

</div>

)

}

}

*export* *default* App

*// PortalDemo.js*

*import* React *from* 'react'

*import* ReactDOM *from* 'react-dom'

*function* PortalDemo(){

*return* ReactDOM.createPortal(

<h1>Это есть портал</h1>,

document.getElementById('portal-root')

)

}

*export* *default* PortalDemo

*<!-- index.html -->*

<body>

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

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

</body>

[https://reactdev.ru/handbook/portals/#event-bubbling-through-portals](https://reactdev.ru/handbook/portals/" \l "event-bubbling-through-portals)

[Medium](https://medium.com/nuances-of-programming/понятие-о-порталах-в-react-с-примерами-использования-9184c95a54d7)

**Что такое контекст (Context)?**

Context API предоставляет возможность передавать данные в дереве компонента без необходимости передачи пропов на каждом уровне вручную.

Например, статус аутентификации пользователя, языковые предпочтения или цветовая схема могут использоваться многими компонентами приложения:

*const* { Provider, Consumer } = React.createContext(defaultValue)

### **Для чего используется дефолтное значение контекста?[​](https://my-js.org/docs/other/react-questions" \l "для-чего-используется-дефолтное-значение-контекста)**

Аргумент defaultValue используется в случаях, когда в дереве компонента не найден подходящий провайдер контекста (Provider). Это может быть полезным для тестирования компонентов в изоляции без необходимости их оборачивания в Provider:

*const* MyContext = React.createContext(defaultValue)

### **Что такое потребитель?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-потребитель)**

Потребитель (consumer) - это компонент, подписанный (реагирующий) на изменения контекста. Ему требуется функция в качестве дочернего элемента, получающая текущее значение контекста и возвращающая узел React. Аргумент value равняется пропу value ближайшего провайдера данного контекста:

### **Как решать проблемы производительности при использовании контекста?[​](https://my-js.org/docs/other/react-questions" \l "как-решать-проблемы-производительности-при-использовании-контекста)**

Контекст использует идентификацию ссылок для определения необходимости в повторном рендеринге. Существуют некоторые ошибки, которые могу привести к непреднамеренному рендерингу в потребителях при повторном рендеринге родительского компонента.

Например, в представленном ниже примере все потребители будут перерисовываться при каждом рендеринге провайдера, поскольку каждый раз создается новый объект-значение пропа value:

*class* App *extends* React.Component {

render() {

*return* (

<Provider value={{ something: 'нечто' }}>

<Toolbar />

</Provider>

)

}

}

Данная проблема может быть решена путем сохранения состояния в конструкторе родительского компонента:

*class* App *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = {

value: { something: 'нечто' },

}

}

render() {

*return* (

<Provider value={*this*.state.value}>

<Toolbar />

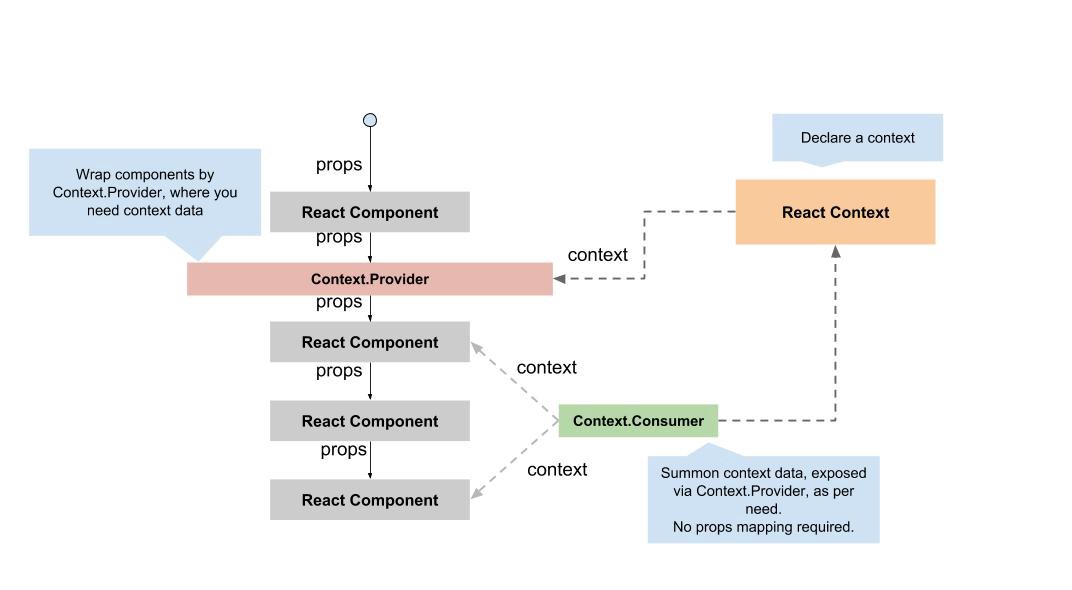
</Provider>

)

}

}

### **Что такое useContext()?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-usecontext)**



React Context API позволяет получать данные на разных уровнях дерева компонентов без их передачи через props:

*import* React *from* "react"

*import* ReactDOM *from* "react-dom"

*// создаем контекст*

*const* NumberContext = React.createContext()

*// он возвращает такой объект*

*// { Provider, Consumer }*

*function* App() {

*// используем провайдер для предоставления потомкам*

*// доступа к данным, содержащимся в контексте*

*return* (

<NumberContext.Provider value={10}>

<Display />

</NumberContext.Provider>

)

}

*function* Display() {

*const* value = useContext(NumberContext)

*return* <div>Ответ: {value}</div>

}

**https://highload.today/react-context/**

Контекст обеспечивает способ передачи данных через дерево компонентов без необходимости передавать реквизиты вручную на каждом уровне.

В типичном приложении React данные передаются сверху вниз (от родителя к дочерним) через реквизиты, но это может быть громоздким для определенных типов реквизита (например, предпочтения локали, темы пользовательского интерфейса), которые требуются для многих компонентов в приложении. Контекст предоставляет способ совместного использования таких значений между компонентами без необходимости явно передавать опору через каждый уровень дерева.

## Когда использовать контекст

Контекст предназначен для обмена данными, которые можно считать «глобальными» для дерева компонентов React, таких как текущий аутентифицированный пользователь, тема или предпочтительный язык.

## API

## React.createContext

**const** {Provider, Consumer} = React.createContext(defaultValue);

Создает { Provider, Consumer } пару. Когда React отображает контекст Consumer, он будет считывать текущее значение контекста из ближайшего соответствия Providerнад ним в дереве.

defaultValueАргумент используется , когда вы оказываете Потребитель без согласования Поставщика над ним в дереве. Это может быть полезно для тестирования компонентов отдельно, без их упаковки.

## Provider

<Provider value={/\* some value \*/}>

Компонент React, который позволяет потребителям подписываться на изменения контекста.

Принимает valueоповещение, которое будет передано потребителям, которые являются потомками этого Провайдера. Один провайдер может быть подключен ко многим потребителям. Провайдеры могут быть вложены, чтобы переопределять значения глубже в дереве.

## Consumer

<Consumer>

{value => /\* render something based on the context value \*/}

</Consumer>

Компонент React, который подписывается на изменения контекста.

Требуется функция в качестве дочернего элемента . Функция получает текущее значение контекста и возвращает узел React. value аргумент, переданные функции будет равен valueподпоркой ближайшего поставщика для данного контекста выше в дереве. Если для этого контекста нет провайдера, value аргумент будет равен defaultValue переданному createContext().

Все потребители переопределяются при изменении значения поставщика. Изменения определяются путем сравнения новых и старых значений с использованием того же алгоритма, что и Object.is.

Поскольку контекст использует ссылочную идентификацию, чтобы определить, когда нужно повторно рендерить, есть некоторые ошибки, которые могут вызвать непреднамеренное рендеринг в потребителях, когда родительский агент переопределяет. Например, приведенный ниже код будет повторно отображать всех потребителей каждый раз, когда поставщик повторно создает, поскольку новый объект всегда создается для value:

**class** App extends React.Component {

render() {

**return** (

<**Provider** value={{something: 'something'}}>

<**Toolbar** />

</**Provider**>

);

}

}

https://unetway.com/tutorial/reactjs-context-kontekst

[**Что такое React хуки (Hooks)?**](https://youtu.be/RpcB5jnJvcI?t=475)

Хуки (hooks) - это встроенные функции, которые позволяют использовать состояние и методы жизненного цикла внутри функциональных компонентов. Они не конфликтуют с классовыми компонентами, так что их можно легко внедрить в существующую кодовую базу.

Правила использования хуков

* Хуки нельзя использовать внутри циклов, условий или вложенных функций.
* Хуки можно использовать либо внутри компонентов, либо внутри других хуков (в том числе, пользовательских, "кастомных", custom).

Встроенные хуки

Основные (базовые)

* useState()
* useEffect()
* useContext()

Дополнительные (продвинутые)

* useReducer()
* useCallback()
* useMemo()
* useRef()
* useImperativeHandle()
* useLayoutEffect()
* useDebugValue()

Преимущества хуков

* С ними легче работать, их легче тестировать (как отдельные функции компонентов), они делают код чище, улучшают его читаемость - сложная логика может быть вынесена в "кастомный" хук.
* Позволяют разделять сложную логику на маленькие функции, используемые внутри компонентов.
* Повышают возможность повторного использования кода.
* Позволяют распределять логику между компонентами через пользовательские хуки.
* Являются более податливыми к перемещениям в дереве компонентов.

Пример

Классовый компонент:

*import* React, { Component } *from* 'react'

*class* App *extends* Component {

constructor(props) {

*super*(props)

*this*.state = {

isButtonClicked: false

}

*this*.handleClick = *this*.handleClick.bind(*this*)

}

handleClick() {

*this*.setState((prevState) => ({

isButtonClicked: !prevState.isButtonClicked,

}))

}

render() {

*return* (

<button

onClick={handleClick}

>

{*this*.state.isButtonClicked ? 'Кнопка нажата' : 'Нажми на меня'}

</button>

)

}

}

Функциональный компонент:

*import* React, { useState } *from* 'react'

*const* App = () => {

*const* [isButtonClicked, setIsButtonClicked] = useState(false)

*return* (

<button

onClick={() => setIsButtonClicked(!isButtonClicked)}

>

{isButtonClicked ? 'Кнопка нажата' : 'Нажми на меня'}

</button>

)

}

Заменяют ли хуки рендер-пропы и компоненты высшего порядка?[​](https://my-js.org/docs/other/react-questions2" \l "заменяют-ли-хуки-рендер-пропы-и-компоненты-высшего-порядка)

Хуки

Хуки позволяют функциональным компонентам иметь состояние и методы жизненного цикла подобно классовым компонентам.

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

Существует несколько встроенных хуков:

*import* {

useState,

useEffect,

useReducer,

useCallback,

useMemo,

...

} *from* 'react'

Компоненты высшего порядка

Компонент высшего порядка (HOC) - это компонент, принимающий компонент в качестве аргумента и возвращающий новый компонент. HOC позволяют создавать композицию компонентов.

*import* React, { useEffect } *from* 'react'

*const* withLogging = (Component) => (props) => {

useEffect(() => {

fetch(`/logger?location=${window.location}`)

}, [])

*return* <Component {...props} />

}

*export* *default* withLogging

Мы можем комбинировать несколько HOC и обернуть ими каждую страницу:

*import* React *from* 'react'

*import* withAuth *from* './with-auth.js'

*import* withLogging *from* './with-logging.js'

*import* withLayout *from* './with-layout.js'

*const* page = compose(

withRedux,

withAuth,

withLogging,

withLayout('default')

)

*export* *default* page

Пример использования:

*import* page *from* '../hocs/page.js'

*import* MyPageComponent *from* './my-page-component.js'

*export* *default* page(MyPageComponent)

### Что такое веб-хуки?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-веб-хуки)

Веб-хуки (web hooks) - это механизм, позволяющий серверу отправлять клиенту уведомления о возникновении интересующего его события. Данный механизм напоминает Server-Sent Events.

Веб-хуки иногда называют реверсивными API. В обычных API клиент обращается к серверу (потребляет его данные). В веб-хуках сервер обращается к веб-хуку (конечной точке - endpoint, предоставленной клиентом), т.е. сервер обращается к клиенту.

https://habr.com/ru/companies/simbirsoft/articles/652321/

https://metanit.com/web/react/6.1.php

https://reactdev.ru/handbook/hooks-reference/

**Что Такое JSX?**

JSX(JavaScript и XML) - это XML-подобныйсинтаксис, расширяющий возможности ECMAScript. По сути, он является синтаксическим сахаром для функции React.createElement, совмещая выразительность JavaScriptс HTML-подобнымсинтаксисом разметки.

JSX позволяет создавать HTML-элементы прямо в JavaScript и помещать их в DOM без использования таких методов, как createElement или appendChild. JSX преобразует HTML-теги в элементы React. React использует JSX для шаблонизации вместо обычного JavaScript. Использовать JSX не обязательно, однако он предоставляет несколько преимуществ:

* Он быстрее благодаря оптимизации во время компиляции кода в JavaScript.
* Он также является "типобезопасным", большинство ошибок перехватываются во время компиляции.
* Он позволяет легче и быстрее создавать шаблоны.

*import* React *from* 'react'

*class* App *extends* React.Component {

render() {

*return* (

<div>

Привет, народ!

</div>

)

}

}

*export* *default* App

JSX - это выражения JavaScript. JSX-выражения являются валидными JavaScript-выражениями. После компиляции, они становятся обычными объектами. Например, такой код:

*const* hello = <h1 className = "greet"> Привет, народ!</h1>

Компилируется в такой:

*const* hello = React.createElement {

type: "h1",

props: {

className: "greet",

children: "Привет, народ!"

}

}

Поскольку JSX компилируется в объекты, он может использоваться наравне с обычными выражениями JavaScript.

### Сколько внешних элементов может содержаться в JSX-выражении?[​](https://my-js.org/docs/other/react-questions2" \l "сколько-внешних-элементов-может-содержаться-в-jsx-выражении)

JSX-выражение должно содержать только один внешний (родительский) элемент. Например:

*const* headings = (

<div id = "outermost-element">

<h1>Заголовок</h1>

<h2>Еще один заголовок</h1>

</div>

)

Обратите внимание: для предотвращения рендеринга лишних DOM-элементов можно воспользоваться компонентом <React.Fragment></React.Fragment> или его сокращенной версией <></>. Сокращенная версия работает не во всех вспомогательных инструментах.

### Почему название компонента должно начинаться с большой буквы?[​](https://my-js.org/docs/other/react-questions2" \l "почему-название-компонента-должно-начинаться-с-большой-буквы)

В JSX имена тегов в нижнем регистре считаются названиями HTML-тегов. Существует одно исключение: имена с точкой (аксессор свойства).

Когда название элемента начинается с маленькой буквы, это указывает на встроенный элемент, такой как <div> или <span>, передаваемые в React.createElement(). Названия элементов, начинающиеся с большой буквы, указывают на созданный (пользовательский) или импортированный компонент.

* <component /> компилируется в React.createElement('component') (HTML-тег)
* <Component /> компилируется в React.createElement(Component)
* <obj.component /> компилируется в React.createElement(obj.component)

### В чем заключаются преимущества использования JSX?[​](https://my-js.org/docs/other/react-questions2" \l "в-чем-заключаются-преимущества-использования-jsx)

JSX - это расширение синтаксиса JavaScript, облегчающее создание компонентов. JSX включает в себя разметку и выражения JavaScript. В действительности, JSX является сокращением для React.createElement(), делая код чище и проще.

JSX может быть полезен для создания выразительных конструкций или пользовательских компонентов, снижая количество опечаток в больших древовидных структурах и облегчая процесс преобразования разметки в деревья React-элементов. Кроме того, при использовании JSX в консоль выводятся предупреждения и потенциальные ошибки, что позволяет устранять проблемы на ранней стадии разработки.

JSX — **расширение синтаксиса JavaScript**. Этот синтаксис выглядит как язык шаблонов, но наделён всеми языковыми возможностями JavaScript. В результате компиляции JSX и вызова React. createElement() возникают простые объекты — «React-элементы».

https://ru.react.js.org/docs/introducing-jsx.html

<https://habr.com/ru/articles/319270/>

https://metanit.com/web/react/1.3.php

**Разница между состоянием(state) и пропсами(props)?**

Состояние (state) - это данные, содержащиеся внутри компонента. Состояние является локальным (принадлежащем определенному компоненту). Компонент может обновлять состояние с помощью метода setState:

*class* Employee *extends* React.Component {

constructor() {

*this*.state = {

id: 1,

name: "Иван"

}

}

render() {

*return* (

<div>

<p>{*this*.state.id}</p>

<p>{*this*.state.name}</p>

</div>

)

}

}

*export* *default* Employee

Пропы (props) - это данные, передаваемые дочернему компоненту от родительского. props являются доступными только для чтения в получающем их потомке. Тем не менее, передаваемая функция обратного вызова, может быть использована в потомке для обновления его состояния:

*class* ParentComponent *extends* Component {

render() {

*return* (

<ChildComponent name="Потомок" />

)

}

}

*const* ChildComponent = (props) => {

*return* <p>{props.name}</p>

}

Разница между состоянием и пропами

| Пропы | Состояние |
| --- | --- |
| Доступны только для чтения | Состояние обновляется асинхронно |
| Являются иммутабельными | Является изменяемым, но не напрямую |
| Позволяют передавать данные от одного компонента другому в качестве аргумента | Содержит информацию о компоненте |
| Доступны для дочерних компонентов | Недоступно для дочерних компонентов |
| Используются для взаимодействия между компонентами | Используются для рендеринга динамических изменений компонента |
| Компонент без состояния может иметь пропы | Компонент без состояния не может иметь пропов |
| Могут сделать компонент переиспользуемым | Не может сделать компонент преиспользуемым |
| Являются внешними и управляются родительским компонентом | Является внутренним и управляется самим компонентом |

### **Почему не следует обновлять состояние напрямую?[​](https://my-js.org/docs/other/react-questions" \l "почему-не-следует-обновлять-состояние-напрямую)**

Если попытаться обновить состояние напрямую, компонент не будет подвергнут повторному рендерингу:

*// неправильно*

*this*.state.message = 'Привет, народ!'

Вместо этого, следует использовать метод setState. Он планирует (откладывает) обновление состояния компонента. Когда состояние меняется, компонент перерисовывается:

*// правильно*

*this*.setState({ message: 'Привет, народ!' })

Обратите внимание: состояние компонента можно изменять напрямую в constructor() или с помощью синтаксиса определения полей класса (данное предложение находится на 3 стадии рассмотрения).

https://keslo.gitbooks.io/reactjs-help/content/props\_vs\_state.html

[**Что такое React Fiber?**](https://youtu.be/RpcB5jnJvcI?t=689)

React Fiber (волокно) - это новый алгоритм согласования. Согласование (reconcilation) - это процесс сравнения старого и нового деревьев элементов для определения различий между ними. В старом алгоритме (который теперь называют стековым согласованием - stack reconciler) сравнение проводилось синхронно, поэтому основной поток выполнения программы был недоступен для других частей UI, таких как анимация, формирование макета страницы и обработка жестов. Fiber Reconciler преследует такие цели, как:

* Возможность разделения операций на части.
* Возможность задавать приоритеты, перемещать и повторно запускать операции в ходе их выполнения.
* Возможность переключаться между предками и потомками в интересах "макетирования".
* Возможность возвращать несколько React-элементов из ReactDOM.render().

Волокно - это JavaScript-объект, содержащий информацию о компоненте, его входных и выходных данных. Экземпляр компонента имеет два связанных с ним волокна: текущее и формирующееся. Волокно может определяться в виде единичной операции.

React Fiber выполняет согласование в два этапа: render и commit.

1. Методы жизненного цикла, вызываемые на стадии рендеринга (render):

* UNSAFE\_componentWillMount()
* UNSAFE\_componentWillReceiveProps()
* getDerivedStateFromProps()
* shouldComponentUpdate()
* UNSAFE\_componentWillUpdate()
* render()

1. Методы жизненного цикла, вызываемые на стадии фиксации (commit):

* getSnapshotBeforeUpdate()
* componentDidMount()
* componentDidUpdate()
* componentWillUnmount()

Раньше процесс согласования был синхронным (рекурсивным), в настоящее время он делится на 2 стадии. Стадия рендеринга (этап согласования) является асинхронной, 3 метода жизненного цикла, вызываемые на этой стадии, помечены как небезопасные, помещение кода с побочными эффектами в эти методы может вызвать проблемы, поскольку методы разных компонентов могут вызываться в разном порядке.

React Fiber использует requestIdleCallback() для планирования операций с низким приоритетом (так было раньше, в настоящее время вместо requestIdleCallback() используется пакет scheduler) и requestAnimationFrame() для планирования операций с высоким приоритетом.

Проблемы текущей реализации:

* Продолжительные задачи могут приводить к пропуску кадров (frame drops).
* Разные задачи имеют разный приоритет.

Преимущества React Fiber

* Делает приложения более гибкими и отзывчивыми.
* В будущем появится возможность параллельного выполнения операций.
* Ускоряет рендеринг компонентов, помещенных в Suspense.

Волокно доступно для использования, но запускается в режиме совместимости с текущей реализацией.

### Что такое React Fiber?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-react-fiber)

Fiber (волокно) - это новый движок согласования, изменение основного алгоритма, появившийся в React 16. Основной задачей React Fiber является повышения производительности в таких областях, как анимация, создание макета страницы, обработка жестов, возможность приостанавливать, прерывать или повторно запускать выполнение операций, предоставление приоритета определенным типам обновлений, а также новые примитивы параллелизма.

### Для чего предназначен React Fiber?[​](https://my-js.org/docs/other/react-questions" \l "для-чего-предназначен-react-fiber)

Цель React Fiber - повышение производительности в таких областях, как анимация, создание макета страницы и обработка жестов. Основной его особенностью является incremental rendering (инкрементальный рендеринг; используется в Angular): возможность разделения процесса рендеринга на части и их объединение через различные фреймы.

https://habr.com/ru/articles/444276/

[Medium](https://uncleseneca.medium.com/подробный-обзор-react-fiber-66485d12bd37)

https://ru.legacy.reactjs.org/docs/faq-internals.html

https://blog.logrocket.com/deep-dive-react-fiber/

**Что такое фрагмент (Fragment)? Почему фрагмент лучше, чем div?**

Фрагмент (fragment) - это распространенный паттерн в React, который используется в компонентах, возвращающих несколько элементов. Фрагменты позволяют группировать дочерние элементы без создания лишних DOM-узлов:

render() {

*return* (

<React.Fragment>

<ChildA />

<ChildB />

<ChildC />

</React.Fragment>

)

}

Также существует сокращенный синтаксис, но он не поддерживается в некоторых библиотеках (на сегодняшний день таких библиотек почти не осталось):

render() {

*return* (

<>

<ChildA />

<ChildB />

<ChildC />

</>

)

}

### Почему фрагмент лучше, чем дополнительный div?[​](https://my-js.org/docs/other/react-questions" \l "почему-фрагмент-лучше-чем-дополнительный-div)

Вот некоторые причины:

1. Фрагменты немного быстрее и используют меньше памяти. Реальная польза от этого ощущается в очень больших и глубоких деревьях элементов.
2. Некоторые механизмы CSS, например, Flexbox и Grid используют связь предок-потомок, поэтому добавление дополнительных div может поломать макет страницы.
3. Меньшее количество элементов облегчает инспектирование DOM.

Что такое фрагменты с ключами?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-фрагменты-с-ключами)

Фрагменты, определяемые с помощью React.Fragment, могут иметь ключи. Типичным примером такого использования является создание коллекции фрагментов:

*function* Glossary(props) {

*return* (

<dl>

{props.items.map((item) => (

*// без `key` React выведет в консоль предупреждение*

<React.Fragment key={item.id}>

<dt>{item.term}</dt>

<dd>{item.description}</dd>

</React.Fragment>

))}

</dl>

)

}

Обратите внимание: key - это единственный атрибут, который в настоящее время можно добавлять фрагменту. Как обещают разработчики React, в будущем появится возможность добавлять фрагментам другие атрибуты, такие как обработчики событий.

Фрагменты (fragments) позволяют рендерить несколько дочерних элементов за раз без добавления лишних узлов DOM:

*class* App *extends* React.Component {

render() {

*return* (

<React.Fragment>

<ChildA />

<ChildB />

<ChildC />

</React.Fragment>

)

}

}

Преимущества

* Это немного повышает производительности и уменьшает расход памяти. Реальная польза от этого ощущается в очень больших и/или глубоких деревьях компонентов, однако производительность приложения зависит от тысячи мелких деталей, и это одна из них.
* Некоторые механизмы CSS основаны на взаимоотношениях предок-потомок, добавление лишних контейнеров в середину может "поломать" макет страницы.
* Легче инспектировать DOM с помощью DevTools.

https://unetway.com/tutorial/reactjs-fragmenty

[Медиум](https://medium.com/nuances-of-programming/фрагменты-в-react-91d6dd5070b0)

https://kramarenko.com.ua/post/react-fragments-obzor

**Что такое синтетические события в React?**

В React обработчики событий оборачиваются в объект SyntheticEvent. Данные объекты являются общими, т.е. объекты, получаемые обработчиками, повторно используются другими обработчиками для повышения производительности. Это также означает, что получить доступ к свойствам такого объекта не получится, поскольку они сбрасываются перед повторным использованием.

В следующем примере в консоль будет выведено null, поскольку свойства события были сброшены в SyntheticEvent:

*function* handleClick(event) {

setTimeout(*function* () {

console.log(event.target.name)

}, 1000)

}

Эту проблему можно решить посредством сохранения свойства события:

*function* handleClick(event) {

*const* name = event.target.name

setTimeout(*function* () {

console.log(name)

}, 1000)

}

SyntheticEvent

*void* preventDefault()

*void* stopPropagation()

boolean isPropagationStopped()

boolean isDefaultPrevented()

*void* persist()

boolean bubbles

boolean cancelable

DOMEventTarget currentTarget

boolean defaultPrevented

number eventPhase

boolean isTrusted

DOMEvent nativeEvent

DOMEventTarget target

number timeStamp

string type

Что такое объединение событий?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-объединение-событий)

SyntheticEvent объединяются (это называется "event polling"). Объект события переиспользуется, а его свойства сбрасываются после вызова колбека обработчика. Это связано с повышением производительности. Поэтому асинхронный доступ к событию невозможен.

*function* onClick(event) {

console.log(event) *// => null*

console.log(event.type) *// => "click"*

*const* eventType = event.type *// => "click"*

setTimeout(*function*() {

console.log(event.type) *// => null*

console.log(eventType) *// => "click"*

}, 0)

*// не работает: `this.state.clickEvent` имеет значение `null`*

*this*.setState({ clickEvent: event })

*// однако мы по-прежнему можем экспортировать свойства события*

*this*.setState({ eventType: event.type })

}

Если нам требуется асинхронный доступ к свойствам события, мы можем вызвать event.persist() на событии, что удалит синтетическую событие из "пула" и позволит ссылкам на событие сохраняться в пользовательском коде.

SyntheticEvent - это кроссбраузерная обертка для нативных событий браузера. Этот API аналогичен браузерному, включая stopPropagation() и preventDefault(), но работает одинаково во всех браузерах.

### Как передать аргумент в обработчик событий?[​](https://my-js.org/docs/other/react-questions" \l "как-передать-аргумент-в-обработчик-событий)

При выполнении перебора обычной практикой является передача дополнительного аргумента обработчику событий. Это может быть реализовано с помощью стрелочной функции или метода bind.

Пример обновления таблицы пользователей:

<button onClick={(e) => *this*.updateUser(userId, e)}>Обновить данные пользователя</button>

<button onClick={*this*.updateUser.bind(*this*, userId)}>Обновить данные пользователя</button>

В обоих случаях "синтетический" (synthetic) аргумент передается на второй позиции. При использовании стрелочных функций его необходимо передавать явно, при использовании bind() он передается автоматически.

https://ru.react.js.org/docs/events.html

https://reactdev.ru/handbook/handling-events/

https://metanit.com/web/react/2.5.php

### В чем разница между обработкой событий в HTML и React?[​](https://my-js.org/docs/other/react-questions" \l "в-чем-разница-между-обработкой-событий-в-html-и-react)

Вот некоторые из основных отличий:

1. В HTML название события указывается строчными буквами (в нижнем регистре):

<button onclick="activateLasers()" />

В React для этого используется "верблюжий стиль" (camelCase):

<button onClick={activateLasers}>

1. В HTML можно вернуть false для отключения поведения по умолчанию:

<a href='#' onclick="console.log('Ссылка!'); *return* false;" />

В React необходимо явно вызывать метод preventDefault:

*function* handleClick(event) {

event.preventDefault()

console.log('Ссылка!')

}

1. В HTML необходимо вызывать функцию с помощью (). В React этого делать не нужно.

**Что такое React-ссылка (ref)? Как создать ссылку?**

### **Для чего используются ссылки или рефы?[​](https://my-js.org/docs/other/react-questions" \l "для-чего-используются-ссылки-или-рефы)**

Реф (ref) возвращает ссылку на DOM-элемент. Этого в большинстве случаев следует избегать. Тем не менее, ссылки могут быть полезны при необходимости получения прямого доступа к DOM-элементу или экземпляру компонента.

### Как создаются ссылки?[​](https://my-js.org/docs/other/react-questions" \l "как-создаются-ссылки)

Существует 2 подхода:

1. Новый: ссылки создаются с помощью React.createRef() и привязываются к элементу с помощью атрибута ref. Для того, чтобы иметь возможность использовать рефы во всем компоненте, просто присвойте ref свойству экземпляра в конструкторе:

*class* MyComponent *extends* React.Component {

constructor(props) {

*super*(props)

*this*.myRef = React.createRef()

}

render() {

*return* <div ref={*this*.myRef} />

}

}

1. Также можно использовать реф-колбеки (callback refs). Например, доступ к полю для ввода текста строки для поиска можно получить следующим образом:

*class* SearchBar *extends* Component {

constructor(props) {

*super*(props)

*this*.txtSearch = *null*

*this*.state = { term: '' }

*this*.setInputSearchRef = (e) => {

*this*.txtSearch = e

}

}

onInputChange(event) {

*this*.setState({ term: *this*.txtSearch.value })

}

render() {

*return* (

<input

value={*this*.state.term}

onChange={*this*.onInputChange.bind(*this*)}

ref={*this*.setInputSearchRef}

/>

)

}

}

Рефы также могут использоваться в функциональных компонентах с помощью замыканий или хука useRef.

Обратите внимание: использовать встроенные реф-колбеки не рекомендуется.

### Что такое перенаправление или передача ссылки?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-перенаправление-или-передача-ссылки)

Передача ссылки (ref forwarding) - это "фича" (feature - возможность, способность), которая позволяет компонентам принимать реф и передавать его потомкам (дочерним компонентам):

*const* ButtonElement = React.forwardRef((props, ref) => (

<button ref={ref} className="customButton">

{props.children}

</button>

))

*// создаем ссылку на кнопку*

*const* ref = React.createRef()

<ButtonElement ref={ref}>Передать ссылку</ButtonElement>

### **Как установить фокус на инпут при загрузке страницы?[​](https://my-js.org/docs/other/react-questions" \l "как-установить-фокус-на-инпут-при-загрузке-страницы)**

Это можно сделать, создав ссылку (ref) на элемент input и использовав метод componentDidMount():

*class* FocusableInput *extends* React.Component{

componentDidMount() {

*this*.nameInput.focus()

}

render() {

*return* (

<div>

<input defaultValue='Не в фокусе' />

<input

ref={(input) => (*this*.nameInput = input)}

defaultValue='В фокусе'

/>

</div>

)

}

}

Аналогичный функциональный компонент:

*function* FocusableInput() {

*const* inputRef = useRef()

useEffect(() => {

inputRef.current.focus()

}, [])

*return* (

<div>

<input defaultValue='Не в фокусе' />

<input

ref={inputRef}

defaultValue='В фокусе'

/>

</div>

)

}

Refs (ссылки, рефы) предоставляют доступ к узлам DOM или элементам React, созданным с помощью метода render. Рефы являются ссылками на DOM-элементы или классовые компоненты из родительского компонента.

Refs также предоставляют возможность связывания элементов дочернего компонента с родительским в форме передачи (перенаправления) ссылок (ref forwarding).

*class* App *extends* React.Component {

constructor(props) {

*super*(props)

*// создаем ссылку на `DOM-элемент`*

*this*.textInput = React.createRef()

*this*.state = {

value: ''

}

}

*// обновляем состояние с помощью ссылки*

handleSubmit = (e) => {

e.preventDefault()

*this*.setState({ value: *this*.textInput.current.value})

}

render() {

*return* (

<div>

<h1>createRef</h1>

{/\_ данное значение будет обновлено \_/}

<h3>Значение: {*this*.state.value}</h3>

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

{/\_ добавляем ссылку к `input` для обновления `h3` его значением \_/}

<input type="text" ref={*this*.textInput} />

<button>Отправить</button>

</form>

</div>

)

}

}

Случаи использования ссылок

* Установка фокуса, выделение текста или воспроизведение медиа.
* Запуск императивной анимации.
* Интеграция со сторонними библиотеками для работы с DOM.

Когда не следует использовать ссылки

* Во всех случаях, когда можно обойтись декларативным синтаксисом.

### **Что такое перенаправление или передача ссылки?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-перенаправление-или-передача-ссылки)**

Перенаправление или передача ссылки (ref forwarding) - это техника, заключающаяся в передаче ref через компонент в один из его потомков. Данная техника часто используется в библиотеках компонентов и компонентах высшего порядка (HOC).

Мы можем передать ref в компонент с помощью функции React.forwardRef(). Перенаправление ссылки позволяет компоненту передавать полученную ссылку потомку.

*// Ref.js*

*const* TextInput = React.forwardRef((props, ref) => (

<input type="text" placeholder="Привет, народ!" ref={ref} />

))

*const* inputRef = React.createRef()

*class* CustomTextInput *extends* React.Component {

handleSubmit = (e) => {

e.preventDefault()

console.log(inputRef.current.value)

}

render() {

*return* (

<div>

<form onSubmit={(e) => *this*.handleSubmit(e)}>

<TextInput ref={inputRef} />

<button>Отправить</button>

</form>

</div>

)

}

}

Здесь у нас имеется компонент TextInput с потомком, представляющим собой поле для ввода текста. Мы начинаем с создания ссылки:

*const* inputRef = React.createRef()

Затем мы передаем ссылку в <TextInput ref={inputRef}>, определяя ее как JSX-атрибут. React передает ref в функцию forwardRef() в качестве второго аргумента. Далее этот аргумент передается в <input ref={ref}>. После этого значение DOM-узла становится доступным через inputRef.current.

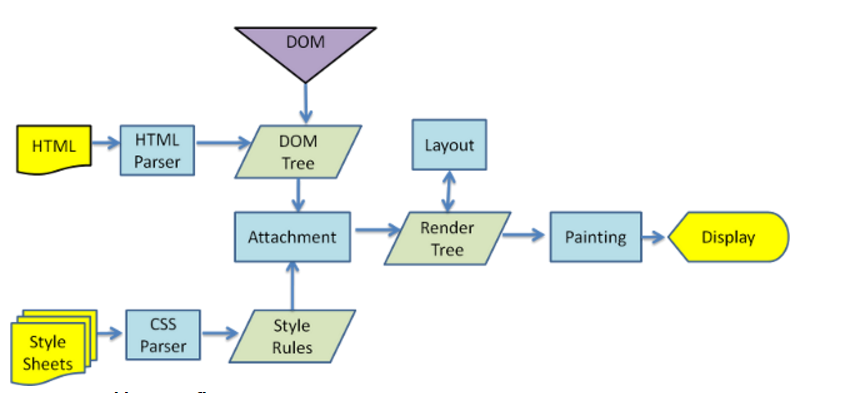
Обратите внимание: аналогичный функционал предоставляется хуком useRef.

https://ru.react.js.org/docs/refs-and-the-dom.html

[Medium](https://nickbulljs.medium.com/refs-в-react-всё-что-нужно-знать-266a979690f8)

**Разница между теневым (Shadow) и виртуальным (Virtual) DOM?**

### В чем разница между ShadowDOM и VirtualDOM?[​](https://my-js.org/docs/other/react-questions2" \l "в-чем-разница-между-shadowdom-и-virtualdom)



Document Object Model (объектная модель документа)

Это способ представления структуры документа с помощью объектов. Это кроссплатформенное и не зависящее от языка соглашение для представления и взаимодействия с данными в HTML, XML и т.д. Браузеры обрабатывают детали реализации DOM, поэтому мы можем взаимодействовать с ним с помощью JavaScript и CSS.

Virtual DOM (виртуальный DOM)

Виртуальный DOM - это абстракция над браузерным DOM. Виртуальный DOM позволяет избежать внесения ненужных изменений в DOM, которые являются "дорогими" с точки зрения производительности, поскольку изменения DOM обычно приводят к повторному рендерингу страницы. Он позволяет объединять несколько обновлений в одно, поэтому не каждое изменение приводит к повторному рендерингу. Вместо этого перерисовка браузерного DOM осуществляется только после применения всех изменений к виртуальному DOM и определения различий между ними. Таким образом, изменения к браузерному DOM применяются частично и по необходимости.

Shadow DOM (теневой DOM)

Теневой DOM, в первую очередь, связан с необходимостью инкапсуляции реализации. Пользовательские элементы (custom elements) могут реализовывать более или менее сложную логику с более или менее сложным DOM. Теневой DOM предоставляет браузеру возможность построения поддеревьев DOM-элементов в процессе рендеринга документа.

Разница

Виртуальный DOM создает дополнительный DOM, а теневой скрывает детали реализации и предоставляет изолированную область видимости для веб-компонентов.

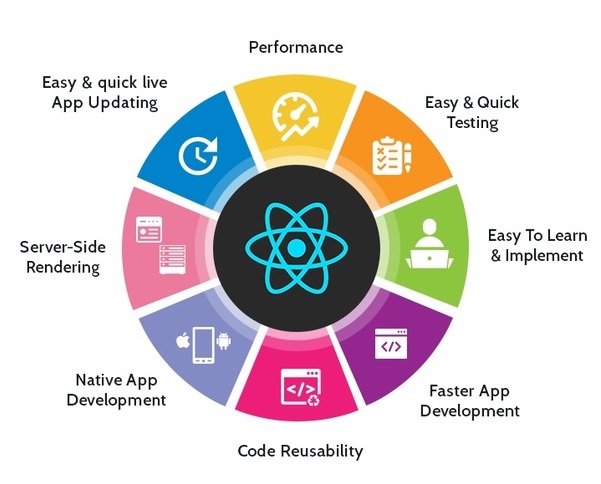
Shadow DOM - это браузерная технология, спроектированная для ограничения области видимости переменных и CSS в веб-компонентах. Virtual DOM - это концепция, реализуемая некоторыми библиотеками JavaScript поверх браузерных API.

https://learn.javascript.ru/shadow-dom

[Medium](https://medium.com/duomly-blockchain-online-courses/shadow-dom-vs-virtual-dom-what-is-the-difference-f2611da536ab)

**Назовите преимущества использования React?**

### **Назовите преимущества и ограничения React[​](https://my-js.org/docs/other/react-questions2" \l "назовите-преимущества-и-ограничения-react)**



Преимущества

* Использование виртуального DOM для определения того, какие части UI подверглись изменениям, и повторный рендеринг только этих частей в реальном DOM существенно повышают производительность приложения.
* JSX (JavaScript и XML) делает код компонентов/блоков более читаемым. Он отчетливо показывает, как компоненты связаны (скомбинированы) между собой.
* Связывание данных в React предоставляет хорошие условия для создания динамичных приложений.
* Быстрый рендеринг. Использование встроенных методов для минимизации количества операций с DOM помогает оптимизировать и ускорить процесс обновления компонентов.
* Тестируемость. React предоставляет отличные встроенные инструменты для тестирования и отладки кода.
* Дружелюбность по отношению к SEO (search engine optimization - поисковая оптимизация). React предоставляет возможность рендеринга страниц на стороне сервера и регистрации обработчиков событий на стороне клиента: React.renderToString() вызывается на сервере; React.render() вызывается на клиенте; \_ React сохраняет разметку, сгенерированную на сервере, и добавляет к ней обработчики событий.

Ограничения

* Кривая обучения. Будучи библиотекой, а не полноценным фреймворком, React требует глубоких знаний по внедрению UI во фреймворки MVC (Model-View-Controller - Модель-Представление-Контроллер).
* Одним из недостатков React также является ориентированность на слой представления. Для решения проблем "Представления" требуется поиск подходящей "Модели" и "Контроллера".
* Разработка приложения без использования изоморфного подхода приводит к проблемам с индексацией приложения поисковыми роботами (речь идет о том, что одностраничные приложения (SPA) индексируются хуже статических).

### **Какие преимущества предоставляет использование React?[​](https://my-js.org/docs/other/react-questions" \l "какие-преимущества-предоставляет-использование-react)**

Вот список основных преимуществ:

1. Повышение производительности приложения благодаря виртуальному DOM.
2. JSX облегчает написание и чтение кода.
3. Возможность рендеринга как на стороне клиента, так и на стороне сервера.
4. Возможность относительно простой интеграции с фреймворками, поскольку React - это всего лишь библиотека.
5. Возможность быстрого юнит и интеграционного тестирования с помощью таких инструментов, как Jest.

### Какие ограничения имеются в React?[​](https://my-js.org/docs/other/react-questions" \l "какие-ограничения-имеются-в-react)

Кроме преимуществ, в React существуют некоторые ограничения:

1. React - это всего лишь библиотека, отвечающая за слой представления, а не полноценный фреймворк.
2. Его трудно изучать новичкам в веб-разработке (впрочем, как и любой другой фреймворк).
3. Интеграция с традиционными MVC-фреймворками требует дополнительных настроек.
4. Код является более сложным из-за встроенных шаблонов и JSX.
5. Большое количество мелких компонентов приводит к сложности в проектировании и построении архитектуры приложения.

<https://evgenev.ru/whats-react-js/>

https://practicum.yandex.ru/blog/chto-takoe-react-i-kak-on-rabotaet/

**Что такое условный рендеринг (Conditional Rendering)?** Как его выполнить?

Для условного рендеринга (conditional rendering) можно использовать обычные if или тернарные операторы. Кроме того, в JSX можно встраивать любое выражение посредством оборачивания его в фигурные скобки, а также совместно с логическим оператором && (короткие вычисления).

<h1>Привет!</h1>

{

messages.length > 0 && !isLogin

? (<h2>

У вас {messages.length} непрочитанных сообщений.

</h2>)

: (<h2>

У вас нет непрочитанных сообщений.

</h2>)

}

### Как выполнить условный рендеринг компонента?[​](https://my-js.org/docs/other/react-questions" \l "как-выполнить-условный-рендеринг-компонента)

В некоторых случаях требуется рендерить разные компоненты в зависимости от некоторого состояния. JSX не рендерит false или undefined, поэтому для рендеринга определенной части компонента только при удовлетворении определенного условия можно использовать короткие вычисления (оператор &&):

*const* MyComponent = ({ name, address }) => (

<div>

<h2>{name}</h2>

{address && <p>{address}</p>}

</div>

)

Если нам требуется условие if else, тогда можно воспользоваться тернарным оператором:

*const* MyComponent = ({ name, address }) => (

<div>

<h2>{name}</h2>

{address

? <p>{address}</p>

: <p>Адрес отсутствует</p>

}

</div>

)

### Как реализовать условное применение CSS-класса?[​](https://my-js.org/docs/other/react-questions" \l "как-реализовать-условное-применение-css-класса)

Не следует использовать фигурные скобки внутри кавычек, поскольку в этом случае они будут оцениваться как строка:

*// не работает*

<div className="btn-panel {this.props.visible ? 'show' : 'hidden'}">

Вместо этого, фигурные скобки следует помещать снаружи (не забудьте про пробел между названиями классов):

<div className={'btn-panel ' + (*this*.props.visible ? 'show' : 'hidden')}>

Лучше использовать шаблонные строки:

<div className={`btn-panel ${*this*.props.visible ? 'show' : 'hidden'}`}>

Как показывать и скрывать элементы?[​](https://my-js.org/docs/other/react-questions2" \l "как-показывать-и-скрывать-элементы)

null

*const* AddToCart = ({ available }) => {

*if* (!available) *return* *null*

*return* (

<div className="full tr">

<button className="product--cart-button">Добавить в корзину</button>

</div>

)

}

Тернарный оператор

Тернарный оператор используется для условного рендеринга компонентов:

<div className="half">

<p>{description}</p>

{remaining === 0 ? (

<span className="product-sold-out">Товар отсутствует</span>

) : (

<span className="product-remaining">{remaining} осталось</span>

)}

</div>

В данном случае, если товаров не осталось, будет отображено "Товар отсутствует"; в противном случае, отображается количество оставшихся товаров.

Короткие вычисления

Это предполагает использование внутри JSX таких выражений, как checkIfTrue && <span>Отображается при истинном значении checkIfTrue</span>. Поскольку логический оператор "И" (&&) возвращает первый ложный операнд, выражение справа не будет оцениваться при ложном значении checkIfTrue .

<h2>

<span className="product--title\_\_large">{nameFirst}</span>

{nameRest.length > 0 && (

<span className="product--title\_\_small">{nameRest.join(" ")}</span>

)}

</h2>

Встроенные стили

<div style={{ display: showInfo ? "block" : "none" }}>Информация</div>

https://reactdev.ru/handbook/conditional-rendering/

https://www.w3schools.com/react/react\_conditional\_rendering.asp

**Что такое компонент-переключатель (Switching Component)?**

Компонент-переключатель (switching component) - это компонент, который рендерит один из нескольких компонентов. Для получения значений пропов для компонентов необходимо использовать объект.

Пример компонента-переключателя, отображающего разные страницы на основе пропа page:

*import* HomePage *from* './HomePage'

*import* AboutPage *from* './AboutPage'

*import* ServicesPage *from* './ServicesPage'

*import* ContactPage *from* './ContactPage'

*const* PAGES = {

home: HomePage,

about: AboutPage,

services: ServicesPage,

contact: ContactPage

}

*const* Page = (props) => {

*const* CurrentPage = PAGES[props.page] || HomePage

*return* <CurrentPage {...props} />

}

*// ключи объекта `PAGES` могут быть использованы в `propTypes`*

Page.propTypes = {

page: PropTypes.oneOf(Object.keys(PAGES)).isRequired

}

**Разница между React и ReactDOM?**

*import* React *from* 'react'

*import* ReactDOM *from* 'react-dom'

*class* MyComponent *extends* React.Component {

render() {

*return* <div>Привет, народ!</div>

}

})

ReactDOM.render(<MyComponent />, someDomNode)

Пакет React содержит такие методы, как React.createElement, React.createClass, React.Component, React.Children и т.д.

Пакет ReactDOM содержит такие методы, как ReactDOM.render, ReactDOM.unmountComponentAtNode, ReactDOM.findDOMNode, а также react-dom/server, включающий методы ReactDOMServer.renderToString и ReactDOMServer.renderToStaticMarkup.

Модуль ReactDOM содержит специфичные для DOM методы, в то время как React включает основные инструменты для разных платформ (например, React Native).

### **Что такое ReactDOM?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-reactdom)**

ReactDOM - это пакет (package), предоставляющий специфичные для браузера методы, которые могут быть использованы на верхнем уровне приложения для эффективного управления DOM-элементами, имеющимися на странице. ReactDOM предоставляет в распоряжение разработчиков следующие методы:

* render.
* findDOMNode.
* unmountComponentAtNode.
* hydrate.
* createPortal и др.

До версии 0.14 ReactDOM был частью React. Одной из главных причин разделения React и ReactDOM было появление React Native. React содержит функционал, используемый в веб- и мобильных приложениях. Функционал ReactDOM используется только в веб-приложениях.

ReactDOM использует наблюдаемые (observables) объекты, которые предоставляют эффективный способ работы с DOM. ReactDOM может использоваться как на стороне клиента, так и на стороне сервера.

*// index.js*

*import* React *from* 'react'

*import* ReactDOM *from* 'react-dom'

*import* App *from* './App/App'

ReactDOM.render(

<React.StrictMode>

<App />

</React.StrictMode>,

document.getElementById('root')

)

Для того, чтобы иметь возможность использовать ReactDOM в веб-приложении, написанном на React, мы, прежде всего, должны его импортировать из react-dom:

*import* ReactDOM *from* 'react-dom'

ReactDOM.render()

Эта функция используется для рендеринга отдельного компонента React или нескольких компонентов, обернутых в другой компонент, фрагмент (fragment) или контейнер div. Данная функция использует эффективные методы React для обновления DOM, имея возможность обновлять только те части DOM, которые подверглись изменениям. Функция возвращает ссылку на компонент или null, если был отрендерен компонент без состояния.

ReactDOM.render() удаляет всех потомков переданного контейнера, если таковые имеются. Он использует высокоэффективный алгоритм сравнения и способен модифицировать любое поддерево DOM:

ReactDOM.render(element, container, callback)

* element: JSX-выражение или React-элемент (компонент), который должен быть отрендерен;
* container: контейнер (HTML-элемент), в котором должен быть отрисован компонент;
* callback: опциональный параметр - функция, вызываемая после завершения рендеринга.

findDOMNode()

Эта функция, обычно, используется для получения узла DOM, в котором был отрендерен некоторый компонент. Данный метод используется редко, поскольку то же самое можно сделать с помощью атрибута ref (ссылка, реф), добавленного к компоненту.

findDOMNode() может быть реализован только в отношении смонтированных компонентов, поэтому функциональные компоненты не могут его использовать:

ReactDOM.findDOMNode(component)

Данный метод принимает один параметр - компонент, поиск которого осуществляется в DOM. Функция возвращает DOM-узел, в котором был отрисован компонент (в случае успеха) или null.

unmountComponentAtNode()

Эта функция используется для размонтирования и удаления React-компонента, который был отрендерен в определенном контейнере:

ReactDOM.unmountComponentAtNode(container)

Данный метод принимает единственный параметр - узел DOM, из которого должен быть удален компонент. Функция возвращает true в случае успеха, или false в противном случае.

hydrate()

Эта функция эквивалентна методу render, но используется при рендеринге на стороне сервера:

ReactDOM.hydrate(element, container, callback)

* element: JSX-выражение или компонент React, который должен быть отрендерен;
* container: контейнер (HTML-элемент), в котором должен быть отрисован компонент;
* callback: опциональный параметр - функция, вызываемая после завершения рендеринга.

Данная функция пытается зарегистрировать обработчики событий для существующей разметки и возвращает ссылку на компонент или null в случае рендеринга компонента без состояния.

createPortal()

Обычно, когда элемент возвращается из метода render компонента, он монтируется в DOM как потомок ближайшего родительского узла, что в некоторых случаях является нежелательным. Порталы (portals) позволяют рендерить компонент в узле DOM, который находится за пределами текущего дерева DOM родительского компонента:

ReactDOM.createPortal(child, container)

* child: JSX-выражение или React-компонент для рендеринга;
* container: контейнер (HTML-элемент), в котором должен быть отрисован компонент.

### **Что такое ReactDOMServer?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-reactdomserver)**

ReactDOMServer - это объект, позволяющий рендерить компоненты в виде статической разметки (обычно, используется на Node.js-серверах). Данный объект в основном используется при рендеринге на стороне сервера. Следующие методы могут быть использованы как на сервере, так и в браузерном окружении:

1. renderToString.
2. renderToStaticMarkup.

Например, мы запускаем основанный на Node.js веб-сервер, такой как Express, Koa или Happi, и вызываем renderToString() для рендеринга корневого элемента в виде строки, которую сервер отправляет в ответ на запрос:

*// используем `Express`*

*import* { renderToString } *from* 'react-dom/server'

*import* MyPage *from* './MyPage'

app.get('/', (req, res) => {

res.write('<!DOCTYPE html><html><head><title>My Page</title></head><body>')

res.write('<div id="content">')

res.write(renderToString(<MyPage/>))

res.write('</div></body></html>')

res.end()

})

Как в React реализовать рендеринг на стороне сервера?[​](https://my-js.org/docs/other/react-questions" \l "как-в-react-реализовать-рендеринг-на-стороне-сервера)

React поддерживает рендеринг на стороне Node.js-сервера из коробки. Для этого используется специальная версия DOM-рендерера, которая реализует такой же паттерн, что и клиентская часть приложения:

*import* ReactDOMServer *from* 'react-dom/server'

*import* App *from* './App'

ReactDOMServer.renderToString(<App />)

Метод renderToString возвращает обычный HTML в виде строки, которая затем может быть помещена в тело (body) ответа сервера. На стороне клиента React определяет предварительно отрендеренный контент и просто вставляет его в существующее дерево компонентов.

В чем разница между React и ReactDOM?[​](https://my-js.org/docs/other/react-questions" \l "в-чем-разница-между-react-и-reactdom)

Пакет react содержит React.createElement(), React.Component, React.Children и другие вспомогательные функции, связанные с элементами и компонентами. О них можно думать как об изоморфных или универсальных помощниках в создании компонентов. Пакет react-dom содержит ReactDOM.render(), а в react-dom/server у нас имеется рендеринг на стороне сервера, обеспечиваемый такими методами как ReactDOMServer.renderToString() и ReactDOMServer.renderToStaticMarkup().

### Почему ReactDOM отделен от React?[​](https://my-js.org/docs/other/react-questions" \l "почему-reactdom-отделен-от-react)

Команда React проделала большую работу по извлечению всех "фич", связанных с DOM, в отдельную библиотеку под названием react-dom. Впервые библиотеки были разделены в React 0.14. Учитывая другие библиотеки, такие как react-native, react-art, react-canvas и react-three, становится очевидным, что ядро React не должно быть тесно связано с браузером или DOM.

Для обеспечения рендеринга в разных средах выполнения кода, команда React разделила основной пакет React на две части: react и react-dom. Это позволяет легко создавать компоненты, которые могут использоваться как в веб, так и в мобильной версиях приложения.

**Разница между компонентом и контейнером?**

### **В чем разница между компонентом-представлением и компонентом-контейнером?[​](https://my-js.org/docs/other/react-questions2" \l "в-чем-разница-между-компонентом-представлением-и-компонентом-контейнером)**

1. Контейнер.

* Отвечает за работу с хранилищем.
* Как правило, содержит несколько элементов, обернутых в div.
* Обычно, имеет состояние.
* Отвечает за предоставление данных и поведение потомков (которые, чаще всего, являются компонентами-представлениями).

Контейнер - это неофициальный термин для React-компонента, подключенного (connected) к хранилищу Redux. Контейнеры получают обновления состояния и отправляют (dispatch) операции. Они, как правило, сами не рендерят элементы, а делегируют эту обязанность дочерним компонентам-представлениям.

Пример

*class* Collage *extends* Component {

constructor(props) {

*super*(props)

*this*.state = {

images: []

}

}

componentDidMount() {

fetch('/api/current\_user/image\_list')

.then((response) => response.json())

.then((images) => *this*.setState({ images }))

}

render() {

*return* (

<div className="image-list">

{*this*.state.images.map((image) => {

<div className="image">

<img src={image.url} alt="" />

</div>

})}

</div>

)

}

}

1. Представитель.

* Отвечает за внешний вид (UI).
* Содержит метод render и другую незначительную логику.
* Не знает, как получать и изменять данные для рендеринга.
* Обычно, является функциональным компонентом без состояния.

Пример

*// классовый компонент*

*class* Image *extends* Component {

render() {

*return* <img src={*this*.props.imageUrl} alt="" />

}

}

*export* *default* Image

*// функциональный компонент*

*const* Image = (props) => (

<img src={props.imageUrl} alt="" />

)

*export* *default* Image

### **В чем разница между компонентом и контейнером?[​](https://my-js.org/docs/other/react-questions" \l "в-чем-разница-между-компонентом-и-контейнером)**

Component - это классовый или функциональный компонент, описывающий визуальное представление определенной части приложения.

Container - это неофициальный термин для описания компонента, подключенного к хранилищу Redux. Контейнеры "подписываются" на обновление состояния Redux и "запускают" (dispatch) операции. Такие компоненты, как правило, не рендерят DOM-элементы: они делегируют эту обязанность дочерним компонентам, отвечающим за визуализацию.

https://reactdev.ru/libs/redux/react-redux/konteineri-i-komponenti/

**Как React обрабатывает или ограничивает использование пропсов определенного типа?**

PropTypes - хороший способ перехвата ошибок, связанных с неправильными типами props. PropTypes позволяет помечать пропы как обязательные или определять их значения по умолчанию.

Пример

*import* React *from* 'react'

*import* PropTypes *from* 'prop-types'

*const* Person = (props) => (

<div>

<h1>{props.firstName} {props.lastName}</h1>

{props.country ? <p>Страна: {props.country}</p> : *null*}

</div>

)

Person.propTypes = {

firstName: PropTypes.string,

lastName: PropTypes.string,

country: PropTypes.string

}

*export* *default* Person

PropTypes определяет тип пропа. Каждый раз, когда через проп передается какое-либо значение, оно проверяется на правильный тип. Если будет обнаружен неправильный тип, в консоль будет выведено сообщение об ошибке.

https://habr.com/ru/articles/541320/

**Что такое строгий режим в React? Его преимущества?**

StrictMode - это инструмент для определения потенциальных проблем приложения. Как и Fragment, StrictMode ничего не рендерит. Он активирует дополнительные проверки и предупреждения в отношении потомков. Проверки выполняются только в режиме для разработки, они не влияют на производственную сборку.

*import* React *from* 'react'

*export* *default* *function* App() {

*return* (

<Fragment>

<Header />

<React.StrictMode>

<div>

<ComponentOne />

<ComponentTwo />

</div>

</React.StrictMode>

<Footer />

</Fragment>

)

}

В приведенном примере проверки не выполняются по отношению к компонентам <Header> и <Footer>. Тем не менее, <ComponentOne> и <ComponentTwo>, которые являются потомками <React.StrictMode>, будут подвергнуты проверкам.

React.StrictMode для повышения эффективности и поиска потенциальных проблем запускает некоторые методы жизненного цикла и хуки по два раза.Некоторыми из таких методом являются:

* constructor()
* render()
* setState()
* getDerivedStateFromProps()
* useState()

Преимущества строгого режима

* Определение компонентов с небезопасными (unsafe) методами жизненного цикла.
* Предупреждения об использовании устаревшего API строковых ссылок (string ref).
* Предупреждения об использовании устаревшего метода findDOMNode.
* Определение неожиданных побочных эффектов.
* Определение использования устаревшего API контекста.

React.StrictMode - это полезный компонент, обеспечивающий индикацию потенциальных проблем в приложении. Как и <Fragment>, <StrictMode> не приводит к рендеренгу лишних DOM-элементов. Он активирует дополнительные проверки и предупреждения. Эти проверки применяются только в режиме для разработки.

*import* React *from* 'react'

*function* ExampleApplication() {

*return* (

<div>

<Header />

<React.StrictMode>

<div>

<ComponentOne />

<ComponentTwo />

</div>

</React.StrictMode>

<Footer />

</div>

)

}

В приведенном примере строгий режим включен только для компонентов ComponentOne и ComponentTwo.

### **В чем заключаются преимущества использования строгого режима?[​](https://my-js.org/docs/other/react-questions" \l "в-чем-заключаются-преимущества-использования-строгого-режима)**

StrictMode может быть полезен в следующих случаях:

1. Идентификация компонентов с небезопасными методами жизненного цикла.
2. Вывод предупреждений об использовании устаревших встроенных ссылок (inline refs).
3. Определение неожиданных побочных эффектов.
4. Определение использования устаревшего интерфейса контекста.
5. Вывод предупреждений об использовании устаревшего метода findDOMNode.

https://reactdev.ru/handbook/strict-mode/

https://unetway.com/tutorial/reactjs-strogij-rezim-strict-mode

**Что такое «бурение пропсов» (Prop Drilling)? Как его избежать?**

В React пропы передаются в одном направлении, сверху вниз, от родительского компонента к дочернему, и последовательно. При наличии незначительного количества пропов или потомков - это не является проблемой. Однако при росте приложения для того, чтобы передать пропы с верхнего уровня приложения компонентам, находящимся на 3 или 4 уровне вложенности, нам приходится передавать одни и те же пропы на каждом уровне дерева компонентов. Это называется бурением пропов (prop drilling).

Context API

Контекст решает некоторые проблемы, связанные с "бурением". Он позволяет компонентам получать данные на любом уровне без их передачи в виде пропов. Передаваемыми данными может быть что угодно: состояние, функция, объект и т.д. Эти данные доступны любым вложенным компонентам в пределах области видимости контекста.

*import* React *from* "react"

*import* ReactDOM *from* "react-dom"

*// создаем контекст*

*const* NumberContext = React.createContext()

*// он возвращает объект с двумя значениями*

*// { Provider, Consumer }*

*function* App() {

*// используем провайдер для предоставления потомкам*

*// доступа к данным*

*return* (

<NumberContext.Provider value={10}>

<div>

<Display />

</div>

</NumberContext.Provider>

)

}

*function* Display() {

*// извлекаем значение из контекста*

*const* value = useContext(NumberContext)

*return* <div>Ответ: {value}.</div>

}

https://devsday.ru/blog/details/78732

https://habr.com/ru/companies/ncloudtech/articles/685400/

**Что такое «опрос» (Polling)? Как его реализовать в React?**

Использование setInterval() в React-компонентах позволяет выполнять функцию или другой код через определенные промежутки времени:

useEffect(() => {

*const* interval = setInterval(() => {

console.log('Это сообщение будет выводиться в консоль каждую секунду')

}, 1000)

*return* () => clearInterval(interval)

}, [])

В приведенном примере задача выполняется каждую секунду в хуке useEffect. Задача начнет выполняться после монтирования компонента. Для отключения таймера мы возвращаем clearInterval() из useEffect().

Использование setInterval() в компонентах React

Для периодического выполнения задачи мы вызываем setInterval() внутри компонента:

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

*const* IntervalExample = () => {

*const* [seconds, setSeconds] = useState(0)

useEffect(() => {

*const* interval = setInterval(() => {

setSeconds((seconds) => seconds + 1)

}, 1000)

*return* () => clearInterval(interval)

}, [])

*return* (

<div className="App">

<header className="App-header">

{seconds} секунд прошло после монтирования компонента.

</header>

</div>

)

}

*export* *default* IntervalExample

В компоненте IntervalExample значение seconds увеличивается на 1 каждую секунду.

### **Что такое объединение событий?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-объединение-событий)**

SyntheticEvent объединяются (это называется "event polling"). Объект события переиспользуется, а его свойства сбрасываются после вызова колбека обработчика. Это связано с повышением производительности. Поэтому асинхронный доступ к событию невозможен.

*function* onClick(event) {

console.log(event) *// => null*

console.log(event.type) *// => "click"*

*const* eventType = event.type *// => "click"*

setTimeout(*function*() {

console.log(event.type) *// => null*

console.log(eventType) *// => "click"*

}, 0)

*// не работает: `this.state.clickEvent` имеет значение `null`*

*this*.setState({ clickEvent: event })

*// однако мы по-прежнему можем экспортировать свойства события*

*this*.setState({ eventType: event.type })

}

Если нам требуется асинхронный доступ к свойствам события, мы можем вызвать event.persist() на событии, что удалит синтетическую событие из "пула" и позволит ссылкам на событие сохраняться в пользовательском коде.

**Разница между элементом и компонентом?**

### **В чем разница между элементом, компонентом и экземпляром компонента?[​](https://my-js.org/docs/other/react-questions2" \l "в-чем-разница-между-элементом-компонентом-и-экземпляром-компонента)**

Компонент (component) - это шаблон. Проект или схема (blueprint). Глобальное определение. Он может быть функцией или классом (с методом рендеринга).

Элемент (element) - это то, что возвращается из компонента. Это объект, описывающий виртуальное представление определенного узла DOM, содержащегося в компоненте. В случае с функциональными компонентами, указанный объект возвращается функцией. В классовых компонентах объект возвращается методом render. Элементы React - это не то, что мы видим в браузере. Это всего лишь объекты, хранящиеся в памяти, мы не можем их изменять.

*import* React *from* 'react'

*import* ReactDOM *from* 'react-dom'

*import* './index.css'

*class* MyComponent *extends* React.Component {

constructor(props) {

*super*(props)

console.log('Это экземпляр компонента: ', *this*)

}

render() {

*const* another\_element = <div>Привет, народ!</div>

console.log('Это элемент: ', another\_element)

*return* another\_element

}

}

console.log('Это компонент: ', MyComponent)

*const* element = <MyComponent/>

console.log('Это элемент: ', element)

ReactDOM.render(

element,

document.getElementById('root')

)

Элементы React

Элемент - это обычный JavaScript-объект с определенными методами. Он имеет 4 свойства:

* type - строковое представление HTML-тега или ссылка на React-компонент;
* key - строка-идентификатор React-элемента;
* ref - ссылка на нижележащий узел DOM или экземпляр компонента React;
* props - объект со свойствами.

Элемент - это не экземпляр React-компонента. Это всего лишь упрощенное "описание" того, как должен выглядеть экземпляр компонента (или тег HTML).

Элемент, описывающий компонент, "не знает", в каком DOM-узле он будет отрендерен - эта связь абстрагирована и определяется в процессе рендеринга.

Элементы могут иметь потомков. Это предоставляет возможность формирования деревьев элементов, представляющих виртуальное дерево DOM.

React-компоненты и их экземпляры

Пользовательские компоненты создаются с помощью React.createClass() или путем расширения React.Component (ES2015). В процессе инстанцирования компонент ожидает получить объект со свойствами и возвращает экземпляр.

Компонент может иметь состояние, ему доступны методы жизненного цикла. Он обязательно должен иметь метод render, возвращающий элемент React (или несколько элементов). Обратите внимание, что мы никогда не создаем экземпляры самостоятельно, это делает React.

### В чем разница между элементами и компонентами?[​](https://my-js.org/docs/other/react-questions" \l "в-чем-разница-между-элементами-и-компонентами)

Элемент - это обычный объект, описывающий, что мы хотим увидеть на экране в терминах узлов и других частей DOM. Элементы могут содержать другие элементы в своих свойствах. Создавать элементы в React легко. Однако после создания, элемент не подлежит изменению.

Объектное представление React-элемента выглядит так:

*const* element = React.createElement(

'div',

{ id: 'login-btn' },

'Войти'

)

Функция React.createElement возвращает такой объект:

{

type: 'div',

props: {

children: 'Войти',

id: 'login-btn'

}

}

Данный объект рендерится с помощью ReactDOM.render():

<div id='login-btn'>Login</div>

В отличие от элемента, компонент может определяться по-разному. Он может быть классом с методом render (классовый компонент) или простой функцией (функциональный компонент). В любом случае компонент принимает свойства (пропы, props от properties) на вход и возвращает JSX:

*const* Button = ({ onLogin }) =>

<div id={'login-btn'} onClick={onLogin}>Войти</div>

JSX транспилируется (преобразуется) в функцию React.createElement:

*const* Button = ({ onLogin }) => React.createElement(

'div',

{ id: 'login-btn', onClick: onLogin },

'Войти'

)

Элемент — это простой объект описывающий, что вы хотите чтобы появилось на экране в терминах узлов DOM или других компонентов. Элементы могут содержать другие элементы в своих параметрах. Компонент — может быть двух видов. Он может быть классом с методом render(), который наследуется из React.

[Медиум](https://kanby.medium.com/reactjs-компоненты-элементы-и-экземпляры-3d5ce1d7b7ff)

https://ru.legacy.reactjs.org/docs/glossary.html

**Что такое ReactDOMServer?**

ReactDOMServer - это объект, позволяющий рендерить компоненты в виде статической разметки (обычно, используется на Node.js-серверах). Данный объект в основном используется при рендеринге на стороне сервера. Следующие методы могут быть использованы как на сервере, так и в браузерном окружении:

1. renderToString.
2. renderToStaticMarkup.

Например, мы запускаем основанный на Node.js веб-сервер, такой как Express, Koa или Happi, и вызываем renderToString() для рендеринга корневого элемента в виде строки, которую сервер отправляет в ответ на запрос:

*// используем `Express`*

*import* { renderToString } *from* 'react-dom/server'

*import* MyPage *from* './MyPage'

app.get('/', (req, res) => {

res.write('<!DOCTYPE html><html><head><title>My Page</title></head><body>')

res.write('<div id="content">')

res.write(renderToString(<MyPage/>))

res.write('</div></body></html>')

res.end()

})

Объект ReactDOMServer позволяет рендерить компоненты в виде статической разметки. Обычно, данный объект используется на Node.js-серверах:

*// `ESM`*

*import* ReactDOMServer *from* 'react-dom/server'

*// `CommonJS`*

*const* ReactDOMServer = require('react-dom/server')

Рендеринг на стороне сервера (server-side rendering, SSR) - это техника, позволяющая отрисовывать клиентские одностраничные приложения (single page application, SPA) на сервере и отправлять их клиенту в виде готовой разметки. Это делает динамические компоненты статическими.

* Это повышает скорость рендеринга страниц, что улучшает пользовательский опыт.
* Это улучшает поисковую оптимизацию (search engine optimization, SEO), облегчая индексацию страниц поисковыми роботами.
* Это повышает доступность метаданных (изображения, заголовок, описание и т.д.), что позволяет пользователям легко делиться приложением в социальных сетях.

Пример

Устанавливаем Express:

yarn add express

*# или*

npm i express

Файлы в директории build являются статичными, т.е. будут обрабатываться как есть:

*// server/server.js*

*import* path *from* 'path'

*import* fs *from* 'fs'

*import* express *from* 'express'

*import* React *from* 'react'

*import* ReactDOMServer *from* 'react-dom/server'

*import* App *from* '../src/App'

*const* PORT = 8080

*const* app = express()

*const* router = express.Router()

*const* serverRenderer = (req, res, next) => {

fs.readFile(path.resolve('./build/index.html'), 'utf8', (err, data) => {

*if* (err) {

console.log(err)

*return* res.status(500).send('Возникла ошибка')

}

*return* res.send(

data.replace(

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

`<div id="root">${ReactDOMServer.renderToString(<App />)}</div>`

)

)

})

}

router.use('^/$', serverRenderer)

router.use(

express.static(path.resolve(\_\_dirname, '..', 'build'), { maxAge: '30d' })

)

app.use(router)

*const* PORT = process.env.PORT || 3000

app.listen(PORT, () => {

console.log(`SSR запущен на порту ${PORT}`)

})

После этого, на клиенте, в src/index.js, вместо вызова:

ReactDOM.render(<App />, document.getElementById('root'))

Вызываем ReactDOM.hydrate(), который имеет аналогичный функционал, но также добавляет обработчики событий к существующей разметке после загрузки React:

ReactDOM.hydrate(<App />, document.getElementById('root'))

Node.js-код должен быть транспилирован с помощью Babel, поскольку Node.js не умеет работать с JSX.

Babel

yarn add @babel/register @babel/preset-env @babel/preset-react ignore-styles

*# или*

npm i @babel/register @babel/preset-env @babel/preset-react ignore-styles

Создаем точку входа (entry point) в server/index.js:

require('ignore-styles')

require('@babel/register')({

ignore: [/(node\_modules)/],

presets: ['@babel/preset-env', '@babel/preset-react']

})

require('./server')

"Собираем" React-приложение:

*# сборка приложения*

yarn build

*# или*

npm run build

*# запуск приложения на сервере*

node server/index.js

https://unetway.com/tutorial/reactjs-reactdomserver

https://habr.com/ru/articles/551948/

**Что такое предохранители (Error Boundaries)?**

Предохранители (error boundaries) - это React-компоненты, которые перехватывают любые ошибки, возникающие в дереве потомков, выводят сообщения об ошибках в консоль и отображают резервный UI вместо "сломанного". Предохранители перехватывают ошибки во время рендеринга, в методах жизненного цикла и в конструкторах дочерних компонентов.

Классовый компонент становится предохранителем, когда в нем определяются методы жизненного цикла static getDerivedStateFromError или componentDidCatch (один или оба). static getDerivedStateFromError() используется для рендеринга резервного UI после возникновения ошибки. componentDidCatch() используется для обработки ошибки.

*import* React, { Component } *from* 'react'

*class* ErrorBoundary *extends* Component {

state = {

isErrorOccured: false,

errorMessage: ''

}

componentDidCatch = (error, info) => {

*this*.setState({

isErrorOccured: true,

errorMessage: error.message

})

}

render() {

*if*(*this*.state.isErrorOccured) {

*return* <p>Ошибка: {*this*.state.errorMessage}</p>

} *else* {

*return* <div>{*this*.props.children}</div>

}

}

}

*export* *default* ErrorBoundary

Здесь у нас имеется объект состояния с двумя свойствами - isErrorOccured и errorMessage, которые будут обновлены при возникновении ошибки. Для обновления состояния мы используем метод componentDidCatch, принимающий два аргумента - error и info.

Как использовать предохранитель?

<ErrorBoundary>

<User/>

</ErrorBoundary>

Предохранители не перехватывают ошибки в:

* обработчиках событий;
* асинхронном коде (например, в setTimeout());
* при рендеринге на стороне сервера;
* при возникновении ошибки в самом предохранителе.

https://habr.com/ru/companies/2gis/articles/583894/

https://ru.react.js.org/docs/error-boundaries.html

**Что такое «ленивая» (Lazy) функция?**

### **Что такое React.lazy()?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-reactlazy)**

React.lazy() позволяет создавать компоненты, загружаемые с помощью динамического import(), которые рендерятся как обычные компоненты. Это уменьшает размер "бандла" (кусок кода, отвечающий за определенную часть приложения), поскольку загружаются только те компоненты, которые отображаются на экране в данный момент.

React.lazy() принимает функцию в качестве аргумента, которая возвращает промис в результате вызова import() для загрузки компонента. Модуль "разрешается" модулем с "дефолтным" экспортом, содержащим React-компонент.

*import* { lazy } *from* 'react'

*const* MyComponent = lazy(() => *import*('./MyComponent'))

*const* App = () => {

<Suspense fallback={<div>Загрузка...</div>}>

<MyComponent />

</Suspense>

}

Обратите внимание: лениво загружаемый компонент должен быть обернут в компонент Suspense.

Функция React.lazy позволяет рендерить результаты динамического импорта как обычные компоненты. Она автоматически загружает "бандл", содержащий OtherComponent (см. пример ниже), когда данный компонент будет отрендерен. Функция возвращает промис, который разрешается модулем с экспортом по умолчанию, содержащим React-компонент. OtherComponent обязательно должен быть обернут в компонент Suspense:

*import* React, { Suspense } *from* 'react'

*const* OtherComponent = React.lazy(() => *import*('./OtherComponent'))

*function* MyComponent() {

*return* (

<Suspense>

<OtherComponent />

</Suspense>

)

}

Обратите внимание: React.lazy() и Suspense пока не доступны для рендеринга на стороне сервера. Если нам требуется разделение кода в приложении, которое рендерится на сервере, используйте библиотеку React Loadable.

### **Что такое загружаемые компоненты?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-загружаемые-компоненты)**

Если требуется разделение кода (code splitting) в приложении с серверным рендерингом, рекомендуемым способом является использование загружаемых (loadable) компонентов, поскольку React.lazy() и Suspense недоступны на стороне сервера. Loadable позволяет рендерить результаты динамического импорта в виде обычных компонентов:

Пример:

*import* loadable *from* '@loadable/component'

*const* OtherComponent = loadable(() => *import*('./OtherComponent'))

*function* MyComponent() {

*return* (

<div>

<OtherComponent />

</div>

)

}

После этого OtherComponent будет загружаться как отдельный "бандл" (кусок кода, соответствующий определенной части приложения).

### Что такое Suspense?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-suspense)

Если модуль содержит динамический импорт (dynamic import), который не успел загрузиться к моменту рендеринга родительского компонента, требуется отобразить некоторый запасной контент, например, в виде индикатора загрузки. Это можно реализовать с помощью компонента Suspense.

Пример:

*const* OtherComponent = React.lazy(() => *import*('./OtherComponent'))

*function* MyComponent() {

*return* (

<div>

<Suspense fallback={<div>Загрузка...</div>}>

<OtherComponent />

</Suspense>

</div>

)

}

Suspense оборачивает "ленивый" (загружаемый отложенно) компонент.

https://habr.com/ru/sandbox/183594/

https://dev-gang.ru/article/lenivaja-zagruzka-v-react-d12k77qk9k/

**Разница между рендерингом и монтированием?**

Рендеринг - это функция или метод render, вызываемый компонентом, возвращающие инструкции для создания DOM. Метод render вызывается при каждом рендеринге компонента. Обновление компонента происходит при изменении его state или props.

Монтирование - это первый рендеринг компонента и построение первоначальной объектной модели документа (виртуального DOM). Монтирование компонента означает встраивание создаваемых им элементов в браузерный DOM.

Повторный рендеринг - это повторный вызов функции для получения информации об уже смонтированном компоненте.

*class* App *extends* React.Component {

state = {

showUser: false

}

render() {

*return* (

<div>

{*this*.state.showUser && <User name="Иван" />}

<button onClick={() => *this*.setState({ showUser: true })}>

Показать пользователя

</button>

<button onClick={() => *this*.setState({ showUser: false })}>

Скрыть пользователя

</button>

</div>

)

}

}

ReactDOM.render(<App />, document.getElementById('root'))

React создает экземпляр App и вызывает его метод render для получения указаний относительно того, как должен выглядеть DOM.

**Что такое сhildren?**

children ссылается на контейнер, содержимое которого неизвестно до передачи данных из родительского компонента. Этот контейнер позволяет передавать компоненты другим компонентам в качестве данных подобно любому другому пропу. Отличительной особенностью children является то, что React предоставляет его поддержку через ReactElement API и JSX

*const* Picture = (props) => {

*return* (

<div>

<img src={props.src} alt=""/>

{props.children}

</div>

)

}

Компонент Picture содержит элемент <img>, получает некоторые пропы и рендерит {props.children}. При рендеринге данного компонента отображается {props.children}, которая является ссылкой на любые компоненты между открывающим и закрывающим тегом Picture.

*// App.js*

render () {

*return* (

<div className='container'>

<Picture key={picture.id} src={picture.src}>

{/\_ указанные здесь компоненты будут переданы как `props.children` \_/}

</Picture>

</div>

)

}

### **Что такое распределенный компонент?[​](https://my-js.org/docs/other/react-questions2" \l "что-такое-распределенный-компонент)**

Распределенный компонент (shared component) - это разновидность компонента, который управляет своим внутренним состоянием, а логику рендеринга делегирует другому компоненту. Таким образом, место определения компонента отделяется от места его реализации. Это предоставляет возможность защитить специфическую логику от остального приложения, предоставляя "чистый" и выразительный API для потребления (consume) компонентом.

Распределенные компоненты конструируются таким образом, чтобы оперировать набором данных, которые передаются через дочерние компоненты вместо пропов. Под капотом они используются низкоуровневое API, такое как React.children.map() и React.cloneElement(). С помощью этих методов компонент получает возможность к "самовыражению" способом, обеспечивающим возможность применения паттернов, связанных с композицией и масштабируемостью.

*function* App() {

*return* (

<Menu>

<MenuButton>

Операции <span aria-hidden>▾</span>

</MenuButton>

<MenuList>

<MenuItem onSelect={() => alert('Download')}>Скачать</MenuItem>

<MenuItem onSelect={() => alert('Copy')}>Копировать</MenuItem>

<MenuItem onSelect={() => alert('Delete')}>Удалить</MenuItem>

</MenuList>

</Menu>

)

}

Компонент <Menu> содержит явно определенное совместное состояние. Компоненты <MenuButton>, <MenuList> и <MenuItem> имеют доступ к этому состоянию, все манипуляции осуществляются в явном виде. Это позволяет получить выразительный API.

### **Что такое children?[​](https://my-js.org/docs/other/react-questions" \l "что-такое-children)**

Потомки (дети, дочерние компоненты) - это проп (this.props.children), позволяющий передавать одни компоненты другим как любые другие пропы. Дерево компонентов, размещаемое между открывающим и закрывающим тегами, передается компоненту в качестве пропа children.

Для работы с этим пропом в React API существуют такие методы как React.Children.map, React.Children.forEach, React.Children.count, React.Children.only и React.Children.toArray.

Простой пример использования пропа children:

*const* MyDiv = React.createClass({

render: *function*() {

*return* <div>{*this*.props.children}</div>

}

})

ReactDOM.render(

<MyDiv>

<span>Привет, </span>

<span>народ</span>

</MyDiv>,

node

)

https://ru.legacy.reactjs.org/docs/composition-vs-inheritance.html

https://react.dev/reference/react/Children

**Что такое события указателя (Pointer Events)?**

События указателя (pointer events) похожи на события мыши (mousedown, mouseup и т.д.), но не зависят от устройства (мышь, стилус, прикосновения и т.д.). Это позволяет избежать реализации функционала для каждого отдельного устройства.

Рассматриваемый API работает также, как существующие обработчики событий. События указателя добавляются к React-компонентам в качестве атрибутов с колбеком, принимающим событие. Внутри колбека происходит обработка этого события.

В ReactDOM доступны следующие типы события:

* onPointerDown
* onPointerMove
* onPointerUp
* onPointerCancel
* onGotPointerCapture
* onLostPointerCapture
* onPointerEnter
* onPointerLeave
* onPointerOver
* onPointerOut

Реализация перетаскивания (Drag'n'Drop) с помощью Point Events:

*import* React, { Component } *from* 'react'

*import* logo *from* './logo.svg'

*import* './App.css'

*import* DragItem *from* './DragItem'

*class* App *extends* Component {

render() {

*return* (

<div className="App">

<header className="App-header">

<img src={logo} className="App-logo" alt="logo" />

<h1 className="App-title">Пример реализации перетаскивания с помощью событий указателя</h1>

</header>

<div className="App-intro">

<DragItem />

</div>

</div>

)

}

}

*export* *default* App

Компонент DragItem:

*import* React *from* 'react'

*const* CIRCLE\_DIAMETER = 100

*export* *default* *class* DragItem *extends* React.Component {

state = {

gotCapture: false,

circleLeft: 500,

circleTop: 100

}

isDragging = false

previousLeft = 0

previousTop = 0

onDown = (e) => {

*this*.isDragging = true

e.target.setPointerCapture(e.pointerId)

*this*.getDelta(e)

}

onMove = (e) => {

*if* (!*this*.isDragging) {

*return*

}

*const* { left, top } = *this*.getDelta(e)

*this*.setState(({ circleLeft, circleTop }) => ({

circleLeft: circleLeft + left,

circleTop: circleTop + top

}))

}

onUp = (e) => (*this*.isDragging = false)

onGotCapture = (e) => *this*.setState({ gotCapture: true })

onLostCapture = (e) => *this*.setState({ gotCapture: false })

getDelta = (e) => {

*const* left = e.pageX

*const* top = e.pageY

*const* delta = {

left: left - *this*.previousLeft,

top: top - *this*.previousTop,

}

*this*.previousLeft = left

*this*.previousTop = top

*return* delta

}

render() {

*const* { gotCapture, circleLeft, circleTop } = *this*.state

*const* boxStyle = {

border: '2px solid #ccc',

margin: '10px 0 20px',

minHeight: 400,

width: '100%',

position: 'relative'

}

*const* circleStyle = {

width: CIRCLE\_DIAMETER,

height: CIRCLE\_DIAMETER,

borderRadius: CIRCLE\_DIAMETER / 2,

position: 'absolute',

left: circleLeft,

top: circleTop,

backgroundColor: gotCapture ? 'red' : 'green',

touchAction: 'none'

}

*return* (

<div style={boxStyle}>

<div

style={circleStyle}

onPointerDown={*this*.onDown}

onPointerMove={*this*.onMove}

onPointerUp={*this*.onUp}

onPointerCancel={*this*.onUp}

onGotPointerCapture={*this*.onGotCapture}

onLostPointerCapture={*this*.onLostCapture}

/>

</div>

)

}

}

Обратите внимание: это работает только в браузерах, поддерживающих спецификацию "Pointer Events".

**Что такое инверсия наследования (Inheritance Inversion)?**

Инверсия наследования (inheritance inversion) - это HOC, который выглядит следующим образом:

*const* inheritanceInversionHOC = (WrappedComponent) => {

*return* *class* extends WrappedComponent {

render() {

*return* *super*.render()

}

}

}

Мы возвращаем класс, расширяющий WrappedComponent. Данная техника называется инверсией наследования, поскольку вместо расширения некоторого класса-усилителя (enhancer) с помощью WrappedComponent, последний сам пассивно расширяется. Отношения между ними напоминают инверсию.

Инверсия наследования предоставляет HOC доступ к экземпляру WrappedComponent. Это означает, что мы можем использовать state, props, методы жизненного цикла и даже метод render данного компонента.

Случаи использования

* Перехват рендеринга.
* Управление состоянием.

*class* Welcome *extends* React.Component {

render() {

*return* (

<div>Добро пожаловать, {*this*.props.user}!</div>

)

}

}

*const* withUser = (WrappedComponent) => {

*return* *class* extends React.Component {

render() {

*if* (*this*.props.user) *return* <WrappedComponent {...*this*.props} />

*return* <div>Добро пожаловать, гость!</div>

}

}

}

*const* withLoaded = (WrappedComponent) => {

*return* *class* extends WrappedComponent {

render() {

*const* { isLoaded } = *this*.props

*if* (!isLoaded) *return* <div>Загрузка...</div>

*return* *super*.render()

}

}

}

*export* *default* withLoaded(withUser(Welcome))

### **Как реализовать HOC-фабрику?[​](https://my-js.org/docs/other/react-questions" \l "как-реализовать-hoc-фабрику)**

Для реализации HOC (компонента высшего порядка) в React существует два способа.

1. Проксирование пропов (Props Proxy - PP).
2. Инверсия наследования (Inheritance Inversion - II).

Они используют разные подходы к управлению WrappedComponent (обернутым компонентом).

Проксирование пропов

При использовании данного подхода метод render возвращает React-элемент с типом WrappedComponent. Мы передаем ему пропы, полученные HOC, поэтому данный подход называется Props Proxy:

*function* ppHOC(WrappedComponent) {

*return* *class* PP *extends* React.Component {

render() {

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

}

}

}

Инверсия наследования

При использовании данного подхода возвращаемый HOC-класс расширяет WrappedComponent. Он называется Inheritance Inversion потому, что вместо того, чтобы WrappedComponent расширял некоторый класс Enhancer, он сам пассивно расширяется "усилителем" (enchancer - усилитель). Отношения между ними напоминают инверсию:

*function* iiHOC(WrappedComponent) {

*return* *class* Enhancer *extends* WrappedComponent {

render() {

*return* *super*.render()

}

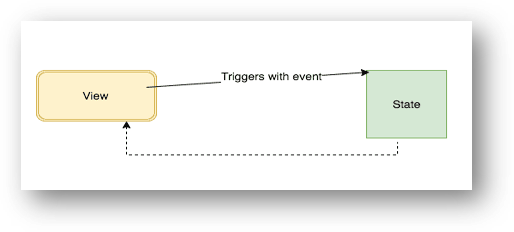
}

}

**Как в React реализовать двустороннее связывание данных?**

Двустороннее связывание данных означает следующее:

* Данные, которые мы изменяем в представлении, обновляют состояние.
* Данные в состоянии обновляют представление.



*class* UserInput *extends* React.Component{

state = {

name: "Привет, React!"

}

handleChange = (e) =>{

*this*.setState({

name: e.target.value

})

}

render(){

*return* (

<div>

<h1>{*this*.state.name}</h1>

<input type="text"

onChange={*this*.handleChange}

value={*this*.state.name}

/>

</div>

)

}

}

Здесь мы регистрируем обработчик событий onChange() на поле для ввода текста, значение атрибута подключается к this.state.name, что позволяет ему синхронизироваться с данным свойством.

Ввод текста в поле запускает обработчик onChange(), который вызывает метод this.setState, обновляющий свойство this.state.name. Обновление данного свойства приводит к повторному рендерингу компонента UserInput.

Такие компоненты называются управляемыми, поскольку значение инпута контролируется React.

Двустороннее связывание данных с помощью хуков

*import* React, { useState } *from* 'react'

*function* App(){

*const* [name, setName] = useState('')

*const* handleChange = (e) => {

setName(e.target.value)

}

*return* (

<div>

<input onChange={handleChange} value={name} />

<h1>{name}</h1>

</div>

)

}

*export* *default* App

https://habr.com/ru/articles/319134/

**Разница между классовым и функциональным компонентами?**

Функциональные компоненты

* Функциональные компоненты - это обычные функции JavaScript. Чаще всего, они представлены в форме стрелочных функций, но их вполне можно создавать и с помощью ключевого слова function.
* Их часто называют компонентами без состояния, которые просто принимают данные и отображают их в некоторой форме, поэтому они, в основном, отвечают за рендеринг UI (так было до появления хуков).
* В них нельзя использовать методы жизненного цикла, например, componentDidMount (в настоящее время хуки предоставляют альтернативы почти всем методам жизненного цикла).
* У них нет метода render.
* Как правило, они отвечают за UI и форму представления данных (например, компонент кнопки).
* Принимают и используют пропы.
* Им следует отдавать предпочтение в случаях, когда не требуется работать с состоянием (так было до появления хуков).

*const* ClockUsingHooks = props => {

*const* [time, setTime] = useState(*new* Date())

useEffect(() => {

*const* tick = setInterval(() => {

setTime(*new* Date())

}, 1000)

*return* () => { clearInterval(tick) }

}, [])

*return* (

<div className="clock">

<h1>Привет! Это часы, созданные с помощью функционального компонента</h1>

<h2>Сейчас {time.toLocaleTimeString()}</h2>

</div>

)

}

*export* *default* ClockUsingHooks

Классовые компоненты

* Для создания классовых компонентов используются классы ES6, расширяющие класс React.Component.
* Их часто называют компонентами с состоянием, поскольку в них реализуется логика поведения на основе некоторого состояния.
* Внутри классов могут использоваться методы жизненного цикла, например, componentDidMount.
* Принимают props и имеют к ним доступ через this.props.
* Могут содержать refs (ссылки, рефы) на нижележащие DOM-узлы.
* Могут использовать такие техники улучшения производительности, как shouldComponentUpdate() и PureComponent

*class* ClockUsingClass *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { date: *new* Date() }

}

componentDidMount() {

*this*.time = setInterval(() => {

*this*.changeTime()

}, 1000)

}

componentWillUnmount() {

clearInterval(*this*.time)

}

changeTime() {

*this*.setState({ date: *new* Date() })

}

render() {

*return* (

<div className="clock">

<h1>Привет! Это часы, созданные с помощью классового компонента</h1>

<h2>Сейчас {*this*.state.date.toLocaleTimeString()}</h2>

</div>

)

}

}

*export* *default* ClockUsingClass

### **Определите разницу между компонентами с состоянием и компонентами без состояния[​](https://my-js.org/docs/other/react-questions2" \l "определите-разницу-между-компонентами-с-состоянием-и-компонентами-без-состояния)**

Компоненты с состоянием и без называются по разному:

* Компоненты-контейнеры и компоненты-представители.
* Умные и глупые компоненты.

Разница между ними, как следует из названия, состоит в том, что одни компоненты имеют состояние, а другие нет. Это означает, что компоненты с состоянием следят за изменением данных, а компоненты без состояния рендерят то, что передается им с помощью пропов, т.е. они всегда рендерят одно и то же.

Stateful/Container/Smart компонент:

*class* Main *extends* Component {

constructor() {

*super*()

*this*.state = {

books: []

}

}

render() {

*return* <BooksList books={*this*.state.books} />

}

}

Stateless/Presentational/Dumb компонент:

*const* BooksList = ({ books }) =>

(<ul>

{books.map((book) => <li key={book.id}>{book.name}</li>})}

</ul>)

Функциональные компоненты или компоненты без состояния

* Похожи на "чистые" функции в JavaScript.
* Часто являются компонентами без состояния.
* Получают пропы от предков и возвращают JSX-элементы.
* Не имеют методов жизненного цикла и состояния.

Классовые компоненты или компоненты с состоянием

* Обычно, являются компонентами с состоянием.
* Имеют методы жизненного цикла и состояние.
* Могут модифицировать состояние.

Случаи использования компонентов без состояния

* Простая визуализация пропов.
* Не требуется состояние или внутренние переменные.
* Создаваемый элемент не является интерактивным.
* Требуется переиспользуемый код.

Случаи использования компонентов с состоянием

* Создаваемый элемент должен быть интерактивным (отвечать на действия пользователя и т.д.).
* Результат рендеринга зависит от состояния (запрос данных, влияющих на рендеринг, и т.п.).
* Результат рендеринга зависит от данных, которые не могут быть переданы в виде пропов.

Обратите внимание: появление хуков стерло границу между классовыми и функциональными компонентами как компонентами с состоянием и без, соответственно.

### **Как создаются компоненты?[​](https://my-js.org/docs/other/react-questions" \l "как-создаются-компоненты)**

Существует 2 способа это сделать:

1. Функциональные компоненты: это простейший способ создания компонента. Такие функции являются "чистыми" (pure), принимают объект с пропами в качестве аргумента и возвращают элемент (или несколько элементов во фрагменте):

*function* Greeting({ message }) {

*return* <h1>{`Привет, ${message}`}</h1>

}

1. Классовые компоненты: для определения компонента также можно использовать ES6-классы. Приведенный функциональный компонент может быть переписан следующим образом:

*class* Greeting *extends* React.Component {

render() {

*return* <h1>{`Привет, ${*this*.props.message}`}</h1>

}

}

### Когда лучше использовать классовый компонент, а когда функциональный?[​](https://my-js.org/docs/other/react-questions" \l "когда-лучше-использовать-классовый-компонент-а-когда-функциональный)

Если компонент нуждается в состоянии или методах жизненного цикла, тогда используйте классовый компонент, иначе, используйте функциональный компонент.

Обратите внимание: в React 16.8 были представлены хуки, позволяющие использовать состояние, методы жизненного цикла и другие возможности классовых компонентов в функциях. В настоящее время классовые компоненты в React почти не используются.

https://bestprogrammer.ru/izuchenie/rukovodstvo-po-komponentam-react-klass-protiv-funktsionala

**Разница между useEffect() и componentDidMount()?**

В React при использовании классовых компонентов мы получаем доступ к методам жизненного цикла (таким как componentDidMount, componentDidUpdate и т.д.). В функциональных компонентах альтернативой методам жизненного цикла являются хуки.

componentDidMount() и useEffect() запускаются после монтирования компонента. Тем не менее, useEffect() вызывается после отображения на экране результатов рендеринга. Это означает, что мы можем получить мерцание (flicker) в случае, когда необходимо прочитать DOM и синхронно обновить состояние для получения нового UI.

useLayoutEffect() был спроектирован специально для таких случаев. Он вызывается перед отображением на экране результатов рендеринга, т.е. синхронно. Поэтому useLayoutEffect(fn, []) ближе к componentDidMount(), чем useEffect(fn, []), по времени выполнения.

*// использование классового компонента*

*import* React, { Component } *from* 'react'

*export* *default* *class* SampleComponent *extends* Component {

componentDidMount() {

*// код, выполняемый после монтирования компонента*

}

render() {

*return* <div>foo</div>

}

}

*// использование функционального компонента*

*import* React, { useEffect } *from* 'react'

*const* SampleComponent = () => {

useEffect(() => {

*// код, выполняемый после монтирования компонента*

}, [])

*return* <div>foo</div>

}

Ограничения useEffect()

Когда useEffect() используется для получения данных от сервера:

* Первый аргумент - это колбек, вызываемый после создания макета и его отрисовки браузером. Выполнение данного колбека не блокирует процесс отрисовки макета браузером (он является асинхронным).
* Второй аргумент - массив значений или зависимостей (как правило, пропов).
* Эффект повторно запускается при изменении любого значения в массиве зависимостей.
* При отсутствии массива зависимостей эффект выполняется при каждом рендеринге.
* Если массив зависимостей является пустым, эффект выполняется только один раз при монтировании компонента по аналогии с componentDidMount().

### **Как реализовать однократное выполнение компонентом некоторой операции при первоначальном рендеринге?[​](https://my-js.org/docs/other/react-questions2" \l "как-реализовать-однократное-выполнение-компонентом-некоторой-операции-при-первоначальном-рендеринге)**

Для этого можно использовать метод жизненного цикла componentDidMount в классовом компоненте:

*class* Homepage *extends* React.Component {

componentDidMount() {

trackPageView('Homepage')

}

render() {

*return* <div>Домашняя страница</div>

}

}

Любые операции, определенные в componentDidMount(), будут выполнены только один раз при монтировании компонента.

Аналогичный функционал можно реализовать с помощью хука useEffect с пустым массивом зависимостей:

*const* Homepage = () => {

useEffect(() => {

trackPageView('Homepage')

}, [])

*return* <div>Домашняя страница</div>

}

useEffect() является более гибким, чем componentDidMount(). Он принимает два параметра:

* Первым параметром является функция обратного вызова, подлежащая выполнению.
* Опциональный второй параметр - массив, содержащий отслеживаемые переменные (массив зависимостей).

Второй параметр контролирует запуск эффекта:

* Если второй параметр отсутствует, эффект выполняется при каждом рендеринге.
* Если массив содержит переменные, то эффект запускается при монтировании компонента, а также при каждом изменении любой переменной.
* Если массив является пустым, то эффект будет запущен только один раз при монтировании компонента. В этом случае функционал будет аналогичен вызову componentDidMount() в классовом компоненте.

https://javascript.plainenglish.io/componentdidmount-and-useeffect-are-not-the-same-heres-why-cea02f474c82

**Преимущества хуков?**

Преимущества хуков

* С ними легче работать, их легче тестировать (как отдельные функции компонентов), они делают код чище, улучшают его читаемость - сложная логика может быть вынесена в "кастомный" хук.
* Позволяют разделять сложную логику на маленькие функции, используемые внутри компонентов.
* Повышают возможность повторного использования кода.
* Позволяют распределять логику между компонентами через пользовательские хуки.
* Являются более податливыми к перемещениям в дереве компонентов.

Пример

Классовый компонент:

*import* React, { Component } *from* 'react'

*class* App *extends* Component {

constructor(props) {

*super*(props)

*this*.state = {

isButtonClicked: false

}

*this*.handleClick = *this*.handleClick.bind(*this*)

}

handleClick() {

*this*.setState((prevState) => ({

isButtonClicked: !prevState.isButtonClicked,

}))

}

render() {

*return* (

<button

onClick={handleClick}

>

{*this*.state.isButtonClicked ? 'Кнопка нажата' : 'Нажми на меня'}

</button>

)

}

}

Функциональный компонент:

*import* React, { useState } *from* 'react'

*const* App = () => {

*const* [isButtonClicked, setIsButtonClicked] = useState(false)

*return* (

<button

onClick={() => setIsButtonClicked(!isButtonClicked)}

>

{isButtonClicked ? 'Кнопка нажата' : 'Нажми на меня'}

</button>

)

}

https://habr.com/ru/companies/ruvds/articles/587728/

https://habr.com/ru/articles/428317/

**Недостатки хуков?**

* Have to respect its rules, without a linter plugin it is difficult to know which rule has been broken.
* Need a considerable time practicing to use them properly (Exp: useEffect).
* Be aware of the wrong use (Exp: useCallback, useMemo). You can read this article by kentcdodds

https://nazrhan-mohcine.medium.com/react-hooks-why-what-and-pros-vs-cons-6d3fe18b8a0a

https://www.stefanjudis.com/notes/the-downsides-of-react-hooks/

**Правила (ограничения) использования хуков?**

Правила использования хуков

* Хуки нельзя использовать внутри циклов, условий или вложенных функций.
* Хуки можно использовать либо внутри компонентов, либо внутри других хуков (в том числе, пользовательских, "кастомных", custom).

### Как обеспечить соблюдение правил использования хуков?[​](https://my-js.org/docs/other/react-questions" \l "как-обеспечить-соблюдение-правил-использования-хуков)

Команда React разработала специальный ESLint-плагин, который следит за соблюдением правил использования хуков. Этот плагин можно установить следующим образом:

yarn add eslint-plugin-react-hooks

*# или*

npm i eslint-plugin-react-hooks

И добавить в настройки ESLint следующее:

*// файл с настройками линтера, например, `.eslintrc.json`*

{

"plugins": [

*// ...*

"react-hooks"

],

"rules": {

*// ...*

"react-hooks/rules-of-hooks": "error"

}

}

Обратите внимание: данный плагин применяется по умолчанию при использовании Create React App для создания проекта.

* Хуки следует вызывать только на верхнем уровне. Не вызывайте хуки внутри циклов, условий или вложенных функций.
* Хуки следует вызывать только из функциональных компонентов React. Не вызывайте хуки из обычных JavaScript-функций.
* Есть только одно исключение, откуда можно вызывать хуки — это ваши пользовательские хуки.

[Медиум](https://medium.com/nuances-of-programming/react-hooks-никакой-магии-только-массивы-59e22f31db2b)

**Что такое поднятие состояния вверх (Lifting State Up)?**

Одним из способов распределения состояния между двумя компонентами является определение состояния в их родительском компоненте. Такой подход называется подъемом состояния (state lifting). Изменения общего состояния незамедлительно отражаются на соответствующих компонентах.

Пример

Компонент App содержит компоненты PlayerContent и PlayerDetails. PlayerContent отображает кнопки с именами игроков. PlayerDetails отображает имя игрока. App содержит состояние для обоих потомков. При нажатии на кнопку, отображается соответствующее имя игрока.

App.js

*import* React *from* 'react'

*import* PlayerContent *from* './PlayerContent'

*import* PlayerDetails *from* './PlayerDetails'

*import* './App.css'

*class* App *extends* React.Component {

constructor(props) {

*super*(props)

*this*.state = { selectedPlayer: [0, 0], playerName: ''}

*this*.updateSelectedPlayer = *this*.updateSelectedPlayer.bind(*this*)

}

updateSelectedPlayer(id, name) {

*const* arr = [0, 0, 0, 0]

arr[id] = 1

*this*.setState({

playerName: name,

selectedPlayer: arr

})

}

render() {

*return* (

<div>

<PlayerContent

active={*this*.state.selectedPlayer[0]}

clickHandler={*this*.updateSelectedPlayer}

id={0}

name="Иван"

/>

<PlayerContent

active={*this*.state.selectedPlayer[1]}

clickHandler={*this*.updateSelectedPlayer}

id={1}

name="Петр"

/>

<PlayerDetails name={*this*.state.playerName}/>

</div>

)

}

}

*export* *default* App

PlayerContent.js

*import* React, { Component} *from* 'react'

*class* PlayerContent *extends* Component {

render () {

*return* (

<button

onClick={() => { *this*.props.clickHandler(*this*.props.id, *this*.props.name) }}

style={ {color: *this*.props.active ? 'red': 'blue' }}

>

{*this*.props.name}

</button>

)

}

}

*export* *default* PlayerContent

PlayerDetails.js

*import* React, { Component } *from* 'react'

*class* PlayerDetails *extends* Component {

render () {

*return* (

<div>{*this*.props.name}</div>

)

}

}

*export* *default* PlayerDetails

https://ru.react.js.org/docs/lifting-state-up.html

https://unetway.com/tutorial/reactjs-podemnoe-sostoanie-vverh-lifting-state-up

**Что делает метод shouldComponentUpdate?**

shouldComponentUpdate() позволяет остановить повторный рендеринг компонента, если в его обновлении нет необходимости. По умолчанию React не сравнивает пропы. При обновлении props или state React запускает процесс повторного рендеринга контента соответствующего компонента.

По умолчанию рассматриваемый метод возвращает true. Для того, чтобы остановить рендеринг "обновленного" компонента, необходимо вернуть false:

shouldComponentUpdate(nextProps, nextState) {

console.log(nextProps, nextState)

console.log(*this*.props, *this*.state)

*return* false

}

Предотвращение ненужного рендеринга

Метод shouldComponentUpdate - самый простой способ повысить производительность приложения. Он сравнивает текущие пропы и состояние со следующими и возвращает true, если они отличаются, и false, если они идентичны. Данный метод не вызывается при первоначальном рендеринге, а также при использовании forceUpdate().

shouldComponentUpdate(nextProps, nextState): вызывается каждый раз при обновлении объекта props или state. В качестве параметра передаются новый объект props и state. Эта функция должна возвращать true (надо делать обновление) или false (игнорировать обновление).

**Разница между createElement() и cloneElement()?**

JSX-элементы транспилируются в функции React.createElement для создания React-элементов, которые используются в качестве объектного представления UI. cloneElement(), в свою очередь, используется для копирования элемента и передачи ему новых пропов.

React.cloneElement() возвращает копию элемента. В эту функцию могут быть переданы дополнительные пропы и потомки. Ее следует использовать, когда родительскому компоненту требуется добавить или изменить props потомка.

*import* React *from* 'react'

*export* *default* *class* App *extends* React.Component {

*// рендеринг родительского и дочернего компонентов*

render() {

*return* (

<ParentComp>

<MyButton/>

<br />

<MyButton/>

</ParentComp>

)

}

}

*// родительский компонент*

*class* ParentComp *extends* React.Component {

render() {

*// добавляем новое свойство*

*const* newProp = 'red'

*// перебираем потомков,*

*// клонируем каждого, добавляем новый проп*

*return* (

<div>

{React.Children.map(*this*.props.children,

(child) => React.cloneElement(child, { newProp }, *null*)

)}

</div>

)

}

}

*// дочерний компонент*

*class* MyButton *extends* React.Component {

render() {

*return* <button style={{ color: *this*.props.newProp }}>Нажми на меня</button>

}

}

### Когда следует использовать React.cloneElement() вместо this.props.children?[​](https://my-js.org/docs/other/react-questions2" \l "когда-следует-использовать-reactcloneelement-вместо-thispropschildren-1)

React.cloneElement() работает только в том случае, когда потомком является единственный React-элемент.

<ReactCSSTransitionGroup

component="div"

transitionName="example"

transitionEnterTimeout={500}

transitionLeaveTimeout={500}

>

{React.cloneElement(*this*.props.children, {

key: *this*.props.location.pathname

})}

</ReactCSSTransitionGroup>

В остальных случаях следует использовать this.props.children. Клонирование является полезным в более продвинутых сценариях использования, когда родительский компонент содержит определенный элемент, а дочерний компонент "хочет" изменить пропы этого элемента или добавить к нему ref для доступа к DOM-элементу.

*class* Users *extends* React.Component {

render() {

*return* (

<div>

<h2>Пользователи</h2>

{*this*.props.children}

</div>

)

}

}

https://www.geeksforgeeks.org/what-is-the-difference-between-createelement-and-cloneelement/

**Что такое useReducer()?**

useReducer - это хук, принимающий функцию-редуктор и начальное состояние приложения в качестве параметров и возвращающий текущее состояние и диспетчер (dispatcher) для отправки (dispatch) операций для обновления состояния.

Несмотря на то, что useState - это базовый хук, а useReducer - продвинутый, на самом деле useState() реализован с помощью useReducer(). Это означает, что useReducer() - это примитив, который может использоваться во всех случаях использования useState(). Редуктор - мощный инструмент, который может использоваться в самых разных сценариях.

Пример

*import* React, { useReducer } *from* 'react'

*const* initialState = 0

*const* reducer = (state, action) => {

*switch* (action) {

*case* 'increment': *return* state + 1

*case* 'decrement': *return* state - 1

*case* 'reset': *return* 0

*default*: *return* state

}

}

*const* ReducerExample = () => {

*const* [count, dispatch] = useReducer(reducer, initialState)

*return* (

<div>

{count}

<button onClick={() => dispatch('increment')}>+</button>

<button onClick={() => dispatch('decrement')}>-</button>

<button onClick={() => dispatch('reset')}>0</button>

</div>

)

}

*export* *default* ReducerExample

Сначала мы определяем начальное состояние и редуктор. Затем передаем их в useReducer(). Хук возвращает текущее значение состояния и диспетчер, который используется для обновления состояния. Когда пользователь нажимает на кнопку, происходит отправка определенной операции в редуктор, который обновляет счетчик на основе операции. Мы может определять столько операций, сколько требуется нашему приложению.

useReducer() – это метод из React Hooks API, похожий на useState, но дающий больший контроль над управлением состояния. Он принимает функцию редюсер и начальное состояние в качестве аргументов, а возвращает состояние и метод dispatch: const [state, dispatch] = React.

https://build-in-saratov.com/reasons-to-use-usereducer/

https://www.w3schools.com/react/react\_usereducer.asp

**Как реализовать однократное выполнение операции при начальном рендеринге?**

Для этого можно использовать метод жизненного цикла componentDidMount в классовом компоненте:

*class* Homepage *extends* React.Component {

componentDidMount() {

trackPageView('Homepage')

}

render() {

*return* <div>Домашняя страница</div>

}

}

Любые операции, определенные в componentDidMount(), будут выполнены только один раз при монтировании компонента.

Аналогичный функционал можно реализовать с помощью хука useEffect с пустым массивом зависимостей:

*const* Homepage = () => {

useEffect(() => {

trackPageView('Homepage')

}, [])

*return* <div>Домашняя страница</div>

}

useEffect() является более гибким, чем componentDidMount(). Он принимает два параметра:

* Первым параметром является функция обратного вызова, подлежащая выполнению.
* Опциональный второй параметр - массив, содержащий отслеживаемые переменные (массив зависимостей).

Второй параметр контролирует запуск эффекта:

* Если второй параметр отсутствует, эффект выполняется при каждом рендеринге.
* Если массив содержит переменные, то эффект запускается при монтировании компонента, а также при каждом изменении любой переменной.
* Если массив является пустым, то эффект будет запущен только один раз при монтировании компонента. В этом случае функционал будет аналогичен вызову componentDidMount() в классовом компоненте.

**Что такое распределенный компонент?**

Распределенный компонент (shared component) - это разновидность компонента, который управляет своим внутренним состоянием, а логику рендеринга делегирует другому компоненту. Таким образом, место определения компонента отделяется от места его реализации. Это предоставляет возможность защитить специфическую логику от остального приложения, предоставляя "чистый" и выразительный API для потребления (consume) компонентом.

Распределенные компоненты конструируются таким образом, чтобы оперировать набором данных, которые передаются через дочерние компоненты вместо пропов. Под капотом они используются низкоуровневое API, такое как React.children.map() и React.cloneElement(). С помощью этих методов компонент получает возможность к "самовыражению" способом, обеспечивающим возможность применения паттернов, связанных с композицией и масштабируемостью.

*function* App() {

*return* (

<Menu>

<MenuButton>

Операции <span aria-hidden>▾</span>

</MenuButton>

<MenuList>

<MenuItem onSelect={() => alert('Download')}>Скачать</MenuItem>

<MenuItem onSelect={() => alert('Copy')}>Копировать</MenuItem>

<MenuItem onSelect={() => alert('Delete')}>Удалить</MenuItem>

</MenuList>

</Menu>

)

}

Компонент <Menu> содержит явно определенное совместное состояние. Компоненты <MenuButton>, <MenuList> и <MenuItem> имеют доступ к этому состоянию, все манипуляции осуществляются в явном виде. Это позволяет получить выразительный API.  
**Расскажите о хуках useCallback(), useMemo(), useImperativeHandle(), useLayoutEffect()?**

useCallback()

Хук useCallback может использоваться для оптимизации рендеринга функциональных компонентов. Он возвращает мемоизированную версию колбека. Это означает, что такой колбек обновляется только при изменении его зависимостей. Это может быть полезным при передаче колбеков оптимизированным (например, с помощью метода shouldComponentUpdate) дочерним компонентам.

*function* App() {

*const* memoizedHandleClick = useCallback(

() => console.log('Случился "клик"!'),

[]

) *// сохраняем колбек без зависимостей, т.е. делаем его статичным*

*return* <Button onClick={memoizedHandleClick}>Нажми на меня</Button>

}

useMemo()

Хук useMemo может использоваться для оптимизации вычислений, производимых в функциональных компонентах. useMemo() похож на useCallback(), за исключением того, что он принимает любые значения, а не только функции. Он принимает функцию, возвращающую значение, и массив зависимостей. Значение, возвращенное функцией, вычисляется повторно только при изменении зависимостей.

Пример приложения, которое рендерит список пользователей и позволяет фильтровать пользователей по именам (фильтрация осуществляется при нажатии соответствующей кнопки):

*import* React *from* 'react'

*const* users = [

{ id: 'a', name: 'Иван' },

{ id: 'b', name: 'Петр' },

]

*const* App = () => {

*const* [text, setText] = React.useState('')

*const* [search, setSearch] = React.useState('')

*const* handleText = (event) => {

setText(event.target.value)

}

*const* handleSearch = () => {

setSearch(text)

}

*// хук `useMemo`*

*const* filteredUsers = React.useMemo(

() =>

users.filter((user) => {

console.log('Запущена функция фильтрации...')

*return* user.name.toLowerCase().includes(search.toLowerCase())

}),

[search]

)

*return* (

<div>

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

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

Поиск

</button>

<List list={filteredUsers} />

</div>

)

}

*const* List = ({ list }) => {

*return* (

<ul>

{list.map((item) => (

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

))}

</ul>

)

}

*const* ListItem = ({ item }) => {

*return* <li>{item.name}</li>

}

*export* *default* App

filteredUsers повторно вычисляется только при изменении состояния search. Она не вызывается при изменении состояния text, поскольку данная переменная не указана в массиве зависимостей useMemo().

useImperativeHandle()

Хук useImperativeHandle позволяет кастомизировать значение, передаваемое родительскому компоненту с помощью ref. Имейте ввиду, что императивный код, в котором используются ссылки, является плохой практикой. useImperativeHandle() должен использоваться совместно с forwardRef():

*function* FancyInput(props, ref) {

*const* inputRef = useRef()

useImperativeHandle(ref, () => ({

focus: () => {

inputRef.current.focus()

}

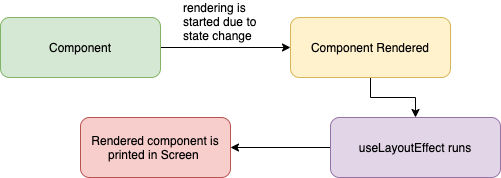
}))

*return* <input ref={inputRef} />

}

FancyInput = forwardRef(FancyInput)

useLayoutEffect()



Хук useLayoutEffect запускается после выполнения всех манипуляций с DOM, но до его отрисовки браузером. Это может быть полезным для получения дополнительной информации из DOM (например, получение величины прокрутки или стилей элемента) и использования этой информации для корректировки DOM или запуска повторного рендеринга путем обновления состояния.

Данный хук предназначен для выполнения тех же задач, которые выполняют методы componentDidMount и componentDidUpdate.

*import* React, { useState, useLayoutEffect } *from* 'react'

*import* ReactDOM *from* 'react-dom'

*const* BlinkyRender = () => {

*const* [value, setValue] = useState(0)

useLayoutEffect(() => {

*if* (value === 0) {

setValue(10 + Math.random() \_ 200)

}

}, [value])

console.log('Рендеринг ', value)

*return* (

<div onClick={() => setValue(0)}>

Значение: {value}

</div>

)

}

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

useLayoutEffect() против useEffect()

* useLayoutEffect: нам нужно внести незаметные для пользователя изменения в DOM.
* useEffect: нам не нужно взаимодействовать с DOM, или наши манипуляции с ним незаметны для пользователя (в большинстве случаев так и есть).

useDebugValue()

Хук useDebugValue может использоваться для отображения подписей к пользовательским хукам в React DevTools.

*function* useFriendStatus(friendID) {

*const* [isOnline, setIsOnline] = useState(*null*)

*// ...*

*// добавляем хуку подпись, отображаемую в инструментах разработчика*

*// это может выглядеть как `FriendStatus: В сети`*

useDebugValue(isOnline ? 'В сети' : 'Не в сети')

*return* isOnline

}

<https://reactdev.ru/handbook/hooks-reference/>

**Как отрендерить HTML код в React-компоненте?**

**???**

**Зачем в setState() нужно передавать функцию?**

### **Почему функция setState является асинхронной?[​](https://my-js.org/docs/other/react-questions2" \l "почему-функция-setstate-является-асинхронной)**

Даже если состояние компонента обновляется синхронно, его пропы всегда обновляются асинхронно. Это означает, что значение пропов является неопределенным до повторного рендеринга родительского компонента. Объекты, предоставляемые React (state, props, refs), согласованы между собой. Если мы реализуем синхронный setState(), то могут возникнуть проблемы.

setState() не изменяет state сразу, но создает запрос на изменение состояния (планирует или откладывает обновление). Поэтому после обновления state теоретически может иметь старое значение. Несколько операций обновления могут объединяться React в одну в целях повышения производительности.

Это связано с тем, что setState() изменяет состояние компонента, что приводит к его повторному рендерингу. Если операция обновления состояние будет синхронной и при этом дорогой с точки зрения производительности, то браузер пользователя лишится интерактивности (перестанет отвечать на действия пользователя до завершения операции). Поэтому такие операции являются асинхронными и группируются для обеспечения лучшего пользовательского опыта и повышения производительности.

### **Какой второй опциональный аргумент может быть передан setState() и в чем его назначение?[​](https://my-js.org/docs/other/react-questions2" \l "какой-второй-опциональный-аргумент-может-быть-передан-setstate-и-в-чем-его-назначение)**

Таким аргументом является колбек, который вызывается после обновления состояния и повторного рендеринга компонента.

Функция setState является асинхронной, поэтому она принимает колбек в качестве второго параметра. Как правило, лучше использовать какой-нибудь метод жизненного цикла, чем полагаться на этот колбек, но не лишним будет знать о такой возможности.

*this*.setState(

{ username: 'Alex' },

() => console.log('Обновление состояние завершено и компонент повторно отрисован.')

)

setState() влечет за собой повторный рендеринг до тех пор, пока shouldComponentUpdate() не вернет false. Во избежание лишних рендерингов, setState() следует вызывать только когда новое состояние действительно отличается от предыдущего. Также вызова setState() следует избегать в таких методах жизненного цикла, как componentDidUpdate, поскольку это может привести к запуску бесконечного цикла.

**Для чего предназначен метод registerServiceWorker() в React?**

Service worker фактически выполняет роль прокси-сервера, находящегося между веб-приложением и браузером, а также сетью (если доступна). Он позволяет (кроме прочего) описывать корректное поведение веб-приложения в режиме офлайн, перехватывать запросы сети и принимать соответствующие меры, основываясь на доступности сети, и обновлять данные, находящиеся на сервере при доступе к нему. Также они имеют доступ к push-уведомлениям и API для фоновой синхронизации....[https://developer.mozilla.org/ru/docs/Web/API/Serv…](https://developer.mozilla.org/ru/docs/Web/API/Service_Worker_API)

**Чем React Router отличается от обычной маршрутизации?**

React Router реализует основанный на компонентах подход к маршрутизации. Он предоставляет различные компоненты, связанные с роутингом, для нужд приложения и платформы. React Router обеспечивает синхронизацию UI с URL (адресом страницы). Он имеет простой API с мощными возможностями, такими как "ленивая" (отложенная) загрузка, динамический поиск совпадения с маршрутом, обработка разных способов переключения между страницами и т.д.

yarn add react-router-dom

*# или*

npm i react-router-dom

*import* React, { Component } *from* 'react'

*import* { BrowserRouter *as* Router, Route, Redirect, Switch } *from* 'react-router-dom'

*import* Todos *from* './components/Todos/Todos'

*import* TodosNew *from* './components/TodosNew/TodosNew'

*import* TodoShow *from* './components/TodoShow/TodoShow'

*class* Router *extends* Component {

constructor(props) {

*super*(props)

}

render() {

*return* (

<Router>

<Switch>

<Route path='/todos/new' component={ TodosNew } />

<Route path='/todos/:id' component={ TodoShow } />

<Route exact path='/' component={ Todos } />

<Redirect from='\_' to='/' />

</Switch>

</Router>

)

}

}

*export* *default* Router

<Router />

Компонент <Router /> оборачивает маршруты (routes) приложения. Внутрь этого компонента помещаются компоненты <Route />, содержащие ссылки на другие страницы.

<Switch />

Данный компонент позволяет рендерить компонент по совпавшему маршруту или резервный контент при отсутствии совпадения. <Switch> возвращает первый совпавший маршрут.

exact

Атрибут exact указывает на необходимость точного совпадения с маршрутом.

**Выше – это старая версия!!!!**

## Основные компоненты[¶](https://reactdev.ru/libs/react-router/" \l "_4)

В React Router существует 3 категории компонентов:

* роутеры (routers), например, <BrowserRouter> или <HashRouter>
* маршруты (route matchers), например, <Route> или <Switch>
* и навигация (navigation), например, <Link>, <NavLink> или <Redirect>

Все компоненты, используемые в веб-приложении, должны быть импортированы из react-router-dom.

### Роутеры[¶](https://reactdev.ru/libs/react-router/" \l "_5)

Любая маршрутизация начинается с роутера. Для веб-проектов react-router-dom предоставляет <BrowserRouter> и <HashRouter>. Основное отличие между ними состоит в способе хранения URL и взаимодействия с сервером.

* <BrowserRouter> использует обычные URL. В этом случае URL выглядят как обычно, но требуется определенная настройка сервера. В частности, сервер должен обслуживать все страницы, используемые на клиенте. Create React App поддерживает это из коробки в режиме разработки и содержит инструкции для правильной настройки сервера.
* <HashRouter> хранить текущую локацию в хэш-части URL (после символа "#"), поэтому URL выглядит примерно так: http://example.com/#/your/page. Поскольку хэш не отправляется серверу, его специальная настройка не требуется.

<https://metanit.com/web/react/4.1.php>

https://blog.webdevsimplified.com/2022-07/react-router/

## Декларативная маршрутизация

Декларативная маршрутизация — это стиль кодирования, используемый в React и React Router. Декларативные маршруты React являются компонентами и используют ту же инфраструктуру, что и любое приложение React. Эти маршруты связывают веб-адреса с определенными страницами и другими компонентами, используя мощный механизм рендеринга React и условную логику для программного включения и выключения маршрутов.

Конечно, любой маршрутизатор хорош настолько, насколько хороша его библиотека. React Router в шестой версии предоставляет множество мощных функций для упрощения задач маршрутизации и является предпочтительным решением для маршрутизации React.

Что делает React Router лучшим по сравнению с другими библиотеками маршрутизации:

* Декларативное определение маршрута (с использованием JSX внутри компонентов React)
* Отраслевой стандарт
* Библиотека предлагает множество примеров кода и обширное онлайн-руководство
* React Router предоставляет возможность использования хуков и функциональных компонентов.

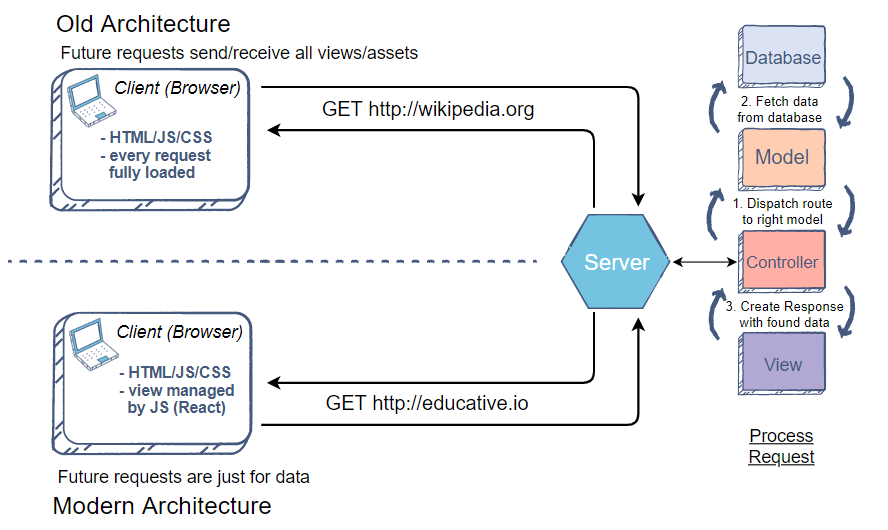
Текущая версия React Router v6 внесла ряд ключевых изменений по сравнению с предыдущей версией 5v:

* Компонент <Switch> был переименован в <Routes>
* Хук useRoutes() заменяет react-router-config для определения маршрутов как простых объектов
* Каждый дочерний компонент <Routes> должен быть <Route>, что может нарушить некоторые предыдущие методы организации и составления маршрутов.

В нашей статье все примеры будут построены c использованием React Router именно шестой версии.

[https://ru.hexlet.io/blog/posts/react-router-v6#deklarativnaya-marshrutizatsiya](https://ru.hexlet.io/blog/posts/react-router-v6" \l "deklarativnaya-marshrutizatsiya)

### Чем React Router отличается от обычной маршрутизации?[​](https://my-js.org/docs/other/react-questions2" \l "чем-react-router-отличается-от-обычной-маршрутизации)



В React существует лишь один HTML-файл (index.html). Когда пользователь переходит на новую страницу, вместо получения данных от сервера, роутер возвращает тот или иной компонент. У пользователя создается впечатление перемещения между страницами, но, в действительности, компоненты приложения являются не более чем разными представлениями одной страницы.

Как React это делает?

Router "заглядывает" в History каждого компонента и при наличии изменений в истории, компонент перерисовывается. До React Router 4 нам приходилось устанавливать значение History вручную. Однако, начиная с React Router 4, большая часть работы, связанной с маршрутизацией, автоматически выполняется компонентом <BrowserRouter> (на стороне клиента).

**Какие хуки были добавлены в React Router версии 5?**

В React Router 5 было представлено 4 новых хука для работы с маршрутизацией.

<Route path="/">

<Home />

</Route>

useHistory

* Предоставляет доступ к пропу history.
* Ссылается на пакет-зависимость history, используемую роутером.
* Основное назначение состоит в программной маршрутизации с помощью таких методов, как push, replace и др.

*import* { useHistory } *from* 'react-router-dom'

*function* Home() {

*const* history = useHistory()

*return* <button onClick={() => history.push('/profile')}>Профиль</button>

}

useLocation

* Предоставляет доступ к пропу location.
* Похож на window.location, но доступен в любом месте, поскольку является представлением состояния и текущей локации роутера.
* Основное назначение состоит в доступе к параметрам строки запроса (query params) и полному адресу страницы.

*import* { useLocation } *from* 'react-router-dom'

*function* Profile() {

*const* location = useLocation()

useEffect(() => {

*const* currentPath = location.pathname

*const* searchParams = *new* URLSearchParams(location.search)

}, [location])

}

useParams

* Предоставляет доступ к параметрам поисковой строки в URL.
* Раньше это было возможно только с помощью match.params.

*import* { useParams, Route } *from* 'react-router-dom'

*function* Profile() {

*const* { name } = useParams()

*return* <p>Профиль пользователя {`${name[0].toUpperCase()}${name.slice(1)}`}</p>

}

*function* Dashboard() {

*return* (

<>

<nav>

<Link to={`/profile/иван`}>Профиль Ивана</Link>

</nav>

<main>

<Route path="/profile/:name">

<Profile />

</Route>

</main>

</>

)

}

useRouteMatch

* Предоставляет доступ к объекту match.
* Если используется без аргументов, возвращает ближайшее совпадение в компоненте или его предках.
* Основное назначение состоит в создании вложенных маршрутов.

*import* { useRouteMatch, Route } *from* 'react-router-dom'

*function* Auth() {

*const* match = useRouteMatch()

*return* (

<>

<Route path={`${match.url}/login`}>

<Login />

</Route>

<Route path={`${match.url}/register`}>

<Register />

</Route>

</>

)

}

useRouteMatch() также может использоваться для поиска совпадения без рендеринга маршрута. Для этого ему необходимо передать аргумент location.

Компонент Redirect

Лучшим вариантом его использование является создание соответствующего свойства в состоянии компонента.

*import* { Redirect } *from* "react-router-dom"

state = { redirect: *null* }

render() {

*if* (*this*.state.redirect) {

*return* <Redirect to={*this*.state.redirect} />

}

}

Проп history

Каждый компонент, который является прямым потомком компонента <Route>, получает объект истории как проп. Это та же самая библиотека history, которая содержит историю сессии React Router. Мы можем использовать его свойства для навигации по определенным маршрутам.

*this*.props.history.push("/first")

[Про роутер 6](https://habr.com/ru/companies/kts/articles/598835/)

### Как программно выполнить перенаправление на другую страницу с помощью React Router?[​](https://my-js.org/docs/other/react-questions2" \l "как-программно-выполнить-перенаправление-на-другую-страницу-с-помощью-react-router)

useHistory()

*import* { useHistory } *from* "react-router-dom"

*function* HomeButton() {

*const* history = useHistory()

*function* handleClick() {

history.push('/home')

}

*return* (

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

Вернуться на главную страницу

</button>

)

}

withRouter()

*import* { withRouter } *from* 'react-router-dom'

*const* Button = withRouter(({ history }) => (

<button type='button' onClick={() => { history.push('/home') }}>

Вернуться на главную страницу

</button>

))

**Как передавать пропсы в React Router?**

Компонент с пропом render() принимает функцию, возвращающую React-элемент, и вызывает ее вместо реализации собственной логики рендеринга. Данная техника используется для распределения кода между компонентами с помощью пропа, значением которого является функция:

*// App.js*

*import* React *from* "react"

*import* { render } *from* "react-dom"

*import* { Greeting } *from* "./components/Greeting.js"

*import* { BrowserRouter *as* Router, Route, Link } *from* "react-router-dom"

*const* styles = {

fontFamily: "sans-serif",

textAlign: "center"

}

*const* App = () => (

<div style={styles}>

<h2>Нажмите на ссылку ниже, чтобы перейти на другую страницу</h2>

<Link to="/greeting/народ">Перейти к /greeting/народ</Link>

</div>

)

*const* RouterExample = () => (

<Router>

<div>

<ul>

<li>

<Link to="/">Главная</Link>

</li>

</ul>

<hr />

<Route exact path="/" component={App} />

<Route

path="/greeting/:name"

render={(props) => <Greeting text="Привет, " {...props} />}

/>

</div>

</Router>

)

render(<RouterExample />, document.getElementById("root"))

*// Greeting.js*

*import* React *from* "react"

*export* *class* Greeting *extends* React.Component {

render() {

*const* { text, match: { params } } = *this*.props

*const* { name } = params

*return* (

<React.Fragment>

<h1>Страница приветствия</h1>

<p>

{text}{name}!

</p>

</React.Fragment>

)

}

}

### Как извлечь параметры из строки запроса?[​](https://my-js.org/docs/other/react-questions2" \l "как-извлечь-параметры-из-строки-запроса)

Для этого можно воспользоваться хуком useParams:

*import* React *from* "react"

*import* { BrowserRouter *as* Router, Switch, Route, Link, useParams } *from* "react-router-dom"

*export* *default* *function* ParamsExample() {

*return* (

<Router>

<div>

<h2>Аккаунты</h2>

<ul>

<li>

<Link to="/netflix">Netflix</Link>

</li>

<li>

<Link to="/zillow-group">Zillow Group</Link>

</li>

<li>

<Link to="/yahoo">Yahoo</Link>

</li>

<li>

<Link to="/modus-create">Modus Create</Link>

</li>

</ul>

<Switch>

<Route path="/:id" children={<Child />} />

</Switch>

</div>

</Router>

)

}

*function* Child() {

*// мы можем использовать хук `useParams` для доступа*

*// к динамическим частям `URL`*

*const* { id } = useParams()

*return* (

<div>

<h3>Идентификатор: {id}</h3>

</div>

)

}

У компонента Routeесть параметр [render](https://reacttraining.com/react-router/web/api/Route/render-func).

<Route

render = { props => logged ? <Product /> : <Redirect to='/' /> }

exact

path='/products/:id'

/>

И важно, в доках об этом конечно сказанно, но я повторюсь.  
component имеет приоритет над render, так что использовать их вместе не стоит.

https://metanit.com/web/react/4.4.php

**Что такое Reselect и как он работает?**

Reselect - это простая библиотека для создания мемоизированных, комбинируемых функций-селекторов. Селекторы используются для эффективного вычисления производных данных из хранилища Redux.

Селекторы позволяют хранить минимально возможное состояние. Селектор вычисляется повторно только при изменении его аргументов. При изменении не связанных с селектором частей дерева компонентов, повторное вычисление селектора не производится.

Селекторы

В нашем случае, селекторы - это всего лишь функции, которые могут вычислять или извлекать данные из хранилища. Обычно, мы получаем состояние с помощью функции mapStateToProps:

*const* mapStateToProps = (state) => {

*return* {

activeData: getActiveData(state.someData, state.isActive)

}

}

Функция getActiveData возвращает из someData все записи со статусом isActive. Недостатком этой функции является то, что при любом изменении состояния она будет повторно вычислять данные.

При использовании Reselect входные аргументы кешируются в мемоизированной функции. Повторное вычисление производится только в случае изменения аргументов.

Пример

*// todo.reducer.js*

*import* { createSelector } *from* 'reselect'

*const* todoSelector = (state) => state.todo.todos

*const* searchTermSelector = (state) => state.todo.searchTerm

*export* *const* filteredTodos = createSelector(

[todoSelector, searchTermSelector],

(todos, searchTerm) =>

todos.filter(todo => todo.title.match(*new* RegExp(searchTerm, 'i')))

)

Мы можем использовать селекторы для получения всех задач (при отсутствии в состоянии searchTerm) или их отфильтрованного списка.

**Назовите основную цель React Fiber?**

Цель React Fiber — улучшить приемлемость React в таких областях как анимация, лейаут и жесты. Его главное свойство — инкрементальная отрисовка — способность разделить работу по отрисовке на части и распределить ее на несколько кадров.

https://dev.to/jennypollard/chto-takoie-react-fiber-react-fiber-architecture-2cho

**Какие типы данных может возвращать render?**

**Below are the list of following types used and return from render method,**

1. React elements: Elements that instruct React to render a DOM node. It includes html elements such as <div/> and user defined elements.
2. Arrays and fragments: Return multiple elements to render as Arrays and Fragments to wrap multiple elements
3. Portals: Render children into a different DOM subtree.
4. String and numbers: Render both Strings and Numbers as text nodes in the DOM
5. Booleans or null: Doesn't render anything but these types are used to conditionally render content.

**Разница между memo и useMemo?**

Самое главное отличие между useMemo и memo . Это то что первый является hook-ом, а второй HOC-ом, а именно Higher-Order Component или же Компонент Высшего Порядка.

https://dev-gang.ru/article/react-optimizacija-komponentov-s-pomosczu-reactmemo-usememo-i-usecallback-f7k7wo44no/

**React.memo** это компонент более высокого порядка, который запоминает результат компонента функции. Если компонент возвращает один и тот же результат с одинаковыми реквизитами, его обертывание в **memo** может привести к повышению производительности. Возьмем наш предыдущий пример **<Header/>**.

‍Скажем, это выглядит примерно так:

const Header = ({title}) => <h1>{title}</h1>

export default Header;

Мы видим, что этот компонент не будет нуждаться в визуализации без изменений **title**, поэтому его можно было бы безопасно завернуть в **React.memo**.

const Header = ({title}) => <h1>{title}</h1>

export default React.memo(Header);

Теперь, когда **Header** визуализируется, он будет делать неглубокое сравнение с его реквизитами. Если эти реквизиты одинаковы, он пропустит отрисовку и вместо этого вернет свое последнее отрисованное значение.

‍Краткое замечание об использовании **memo** и реагировании средств разработки. На момент написания этой статьи, оберните ваш компонент вот так...

const Header = React.memo(({title}) => <h1>{title}</h1>));

export default Header;

...это приведет к тому, что ваш компонент будет отображаться как **Unknown** в react dev tools. Чтобы исправить это, оберните свой компонент **memo** после его определения, как мы делали это ранее:

const Header = ({title}) => <h1>{title}</h1>;

export default React.memo(Header);

## useMemo

**useMemo** позволяет запомнить результаты функции и будет возвращать этот результат до тех пор, пока массив зависимостей не изменится.

‍Например:

const widgetList = useMemo(

() => widgets.map(

w => ({

...w,

totalPrice: someComplexFunction(w.price),

estimatedDeliveryDate: someOtherComplexFunction(w.warehouseAddress)

}),

),

[widgets],

);

В этом примере компонент получает список виджетов. Виджеты, входящие в компонент, должны быть сопоставлены, чтобы включить общую цену и предполагаемую дату доставки, которая использует какую-то сложную и дорогостоящую функцию. Если этот компонент отрисовывается и значение **widgets**равно, то нет необходимости снова запускать эти дорогостоящие функции.

‍Использование **useMemo**запомнит результат, поэтому, если **widgets**он не изменился с момента последнего рендеринга компонента, он пропустит вызов функции и вернет то, что он получил последним.

**Является ли React реактивным?**

state определяет результат рендеринга компонента. Состояние меняется в ответ на действия пользователя, ответы сервера и т.д. До появления хуков локальное состояние могли иметь только классовые компоненты.

Метод setState - единственный легитимный способ обновления состояния компонента после инициализации его начального состояния.

*import* React, { Component } *from* 'react'

*class* Search *extends* Component {

constructor(props) {

*super*(props)

*this*.state = {

searchString: ''

}

}

}

Мы передаем пустую строку в качестве начального значения объекта состояния. Для его обновления необходимо вызвать setState():

setState({ searchString: event.target.value })

Здесь мы передаем объект. Объект содержит значение, с помощью которого мы хотим обновить состояние. Обычно, это запускает процесс согласования (reconcilation). Согласование - это способ, с помощью которого React обновляет DOM, применяя изменения к компоненту на основе обновленного состояния.

При вызове setState() React создает новое дерево, содержащее реактивные элементы в компоненте (с обновленным состоянием). Это дерево используется для определения того, как должен измениться интерфейс компонента в ответ на изменение состояния посредством сравнения новых элементов с предыдущим деревом.

**Реактивность**

React назван так потому, что реагирует на изменения состояния компонентов. Все же он делает это не реактивно, а, скорее, по графику – отсюда появилась шутка, что React следовало бы назвать Schedule.

Однако, если не углубляться внутрь фреймворка, то все что мы видим – это как React реагирует на обновление компонента и автоматически отображает его изменения в дереве документа.

Помните, что входными данными для render() являются свойства (props) и внутреннее состояние, которое может быть обновлено в любое время.

Когда для render меняются входные данные, меняется и результат ее выполнения.

React.js ведет запись жизненного цикла компонента. Когда React.js видит, что один рендер отличается от другого, он переводит разницу между своим виртуальным представлением в операции с DOM API, которые будут отрисованы в документе.

https://proglib.io/p/react-js-concepts

Итак, теперь мы знаем, что React не реактивный, потому что он декларативный, и название этого фреймворка сильно вводит в заблуждение. Слово «планировщик» подходит лучше, я думаю. Но это не ошибка, поскольку он не навязывает какую-либо парадигму, которую вы хотите использовать.

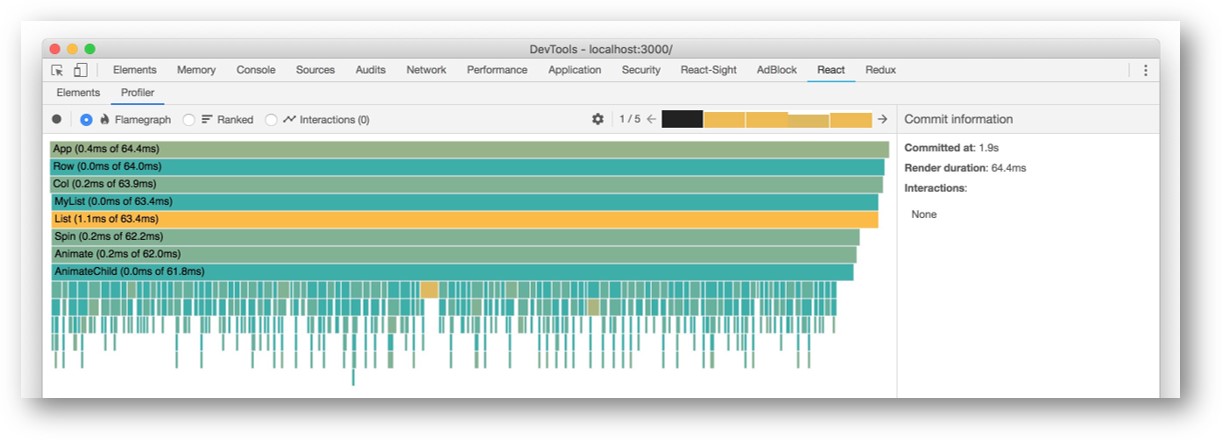
**Техники оптимизации перфоманса React?**

### **Как оптимизировать производительность?[​](https://my-js.org/docs/other/react-questions2" \l "как-оптимизировать-производительность)**

React использует несколько приемов для минимизации количества выполняемых с DOM операций. Для большинства приложений производственная сборка будет удовлетворительной с точки зрения производительности. Тем не менее, существует несколько способов повышения скорости загрузки приложения.

1. React DevTools Profiler.

При возникновении проблем с производительностью одного из компонентов, следует начать с "профилировщика" инструментов разработчика React.



1. Использование метода shouldComponentUpdate.

По умолчанию React рендерит виртуальный DOM и определяет разницу для каждого компонента в дереве при любых изменениях его состояния или пропов. Но очевидно, что это не разумно. С ростом приложения скорость повторного рендеринга и сравнения представлений виртуального DOM после каждой операции будет неуклонно снижаться.

React предоставляет простой метод жизненного цикла для определения того, нуждается ли компонент в повторном рендеринге - shouldComponentUpdate, который запускается перед повторным рендерингом. По умолчанию данный метод возвращает true.

shouldComponentUpdate(nextProps, nextState) {

*return* true

}

Указанный метод позволяет нам контролировать процесс рендеринга. Предположим, что мы хотим предотвратить повторный рендеринг определенного компонента. Для этого мы просто возвращаем false из shouldComponentUpdate(). Как видно из примера реализации метода, мы можем сравнивать текущее и следующее состояние и пропы для определения необходимости перерисовки компонента:

shouldComponentUpdate(nextProps, nextState) {

*return* nextProps.id !== *this*.props.id

}

1. Использование "чистых" компонентов.

"Чистые" (pure) компоненты - это компоненты, которые не перерисовываются при обновлении их state и props одними и теми же значениями. Если значения предыдущего state или props и нового state или props являются одинаковыми, компонент не перерисовывается. Это повышает производительность кода.

1. Использование React.memo().

React.memo() - это компонент высшего порядка. Его функционал похож на React.PureComponent, но он предназначен для функциональных компонентов, а не для классов.

*const* MyComponent = React.memo((props) => {

/\_ рендеринг с помощью пропов \_/

})

Если функциональный компонент рендерит одинаковый результат для одних и тех же пропов, мы можем обернуть его в React.memo() для повышения производительности в некоторых случаях посредством сохранения (запоминания, мемоизации) результата. Это означает, что React пропустит рендеринг компонента и воспользуется последним результатом.

React.memo() проверяет только изменение props. Если в нашем компоненте используются хуки useState или useContext, компонент будет перерисовываться при изменении state или context.

1. "Виртуализация" длинных списков.

Для решения проблем, связанных с длинной новостной лентой, команда React рекомендует использовать технику под названием windowing. Данная техника заключается в рендеринге только видимых пользователю в настоящий момент элементов списка (+/- определенный отступ), что повышает скорость рендеринга. При прокрутке страницы новые элементы извлекаются и рендерятся. react-window и react-virtualized являются двумя наиболее популярными библиотеками для "виртуализации" длинных списков.

[Читать подробнее](https://reactjs.org/docs/optimizing-performance.html)

[Медиум](https://medium.com/@vladimirmorulus/оптимизация-производительности-react-приложений-61d503c1d87e)

**Лучшие практики безопасности в React?**

https://coffee-web.ru/blog/how-to-use-react-js-in-a-secure-way/

<https://bestprogrammer.ru/izuchenie/rukovodstvo-po-bezopasnosti-react-js-ugrozy-uyazvimosti-i-sposoby-ih-ustraneniya>