**Вопросы по React**

1. Что такое React? С какой целью он создавался?

2. Что такое виртуальный DOM? Как виртуальный DOM повышает производительность React? В чем различие между реальной модель DOM?

3. Перечислите некоторые из основных преимуществ React. Приведите не менее трех примеров реализации (код).

4. Чем отличаются setState и useState? Что такое хук? Какие виды хуков существуют? Приведите пример использования.

5. Как организовать ссылку на узел DOM?

6. Для чего нужны реквизиты? Как они используются в React-компонентах?

7. Что такое JSX? В чем преимущество его использования?

8. Что такое props в компоненте? Опишите механизм его работы. Приведите примеры использования.

9. Как мы можем условно отрисовать JSX-компонент в React? Приведите примеры полного и сокращенного синтаксиса.

10. Каким способом можно создавать анимации в React? Приведите примеры.

11. Почему браузер не может читать JSX? Какие библиотеки для интерпретации вам известны?

12. Как можно объяснить фразу «В React все является компонентом».

13. Как встроить два или более компонентов в один компонент?

14. Как обновить состояния компонентов?

15. Как стрелочные функции работают в React? Как их передавать?

16. Что такое Redux? Для чего он нужен?

17. Какие команды библиотеки npm нужно использовать, чтобы создать, запустить и подготовить к размещению на сервере React-проект?

18. Для чего нужен Node.js? Какие у него особенности?

19. Опишите механизм работы метода render?

20. Опишите, каким образом можно использовать массивы в качестве состояний компонентов?

21. С помощью какой библиотеки можно реализовать многостраничность в React-приложении? Приведите примеры?

22. В чем особенность декларативного подхода по сравнению с императивным. Приведите примеры.

23. Для чего нужно экспортировать компоненты в React?

24. Каким образом представление и логика работы в React соединяются в одном месте? Приведите примеры?

25. Каким образом можно организовать работу компонент со связными состояниями?

Оглавление

[1. Что такое React? С какой целью он создавался?( Для чего нужен React) 4](#_Toc156767251)

[2. Что такое виртуальный DOM? Как виртуальный DOM повышает производительность React? В чем различие между реальной модель DOM? 5](#_Toc156767252)

[3. Перечислите некоторые из основных преимуществ React. Приведите не менее трех примеров реализации (код). 8](#_Toc156767253)

[4. Чем отличаются setState и useState? Что такое хук? Какие виды хуков существуют? Приведите пример использования. 10](#_Toc156767254)

[5. Как организовать ссылку на узел DOM? 12](#_Toc156767255)

[6. Для чего нужны реквизиты? Как они используются в React-компонентах? 15](#_Toc156767256)

[7. Что такое JSX? В чем преимущество его использования? 16](#_Toc156767257)

[8. Что такое props в компоненте? Опишите механизм его работы. Приведите примеры использования. 17](#_Toc156767258)

[9. Как мы можем условно отрисовать JSX-компонент в React? Приведите примеры полного и сокращенного синтаксиса. 19](#_Toc156767259)

[10. Каким способом можно создавать анимации в React? Приведите примеры. 20](#_Toc156767260)

[ReactTransitionGroup 22](#_Toc156767261)

[11. Почему браузер не может читать JSX? Какие библиотеки для интерпретации вам известны? 26](#_Toc156767262)

[12. Как можно объяснить фразу «В React все является компонентом». 27](#_Toc156767263)

[14. Как обновить состояния компонентов? 31](#_Toc156767264)

[15. Как стрелочные функции работают в React? Как их передавать? 32](#_Toc156767265)

[16. Что такое Redux? Для чего он нужен? 34](#_Toc156767266)

[17. Какие команды библиотеки npm нужно использовать, чтобы создать, запустить и подготовить к размещению на сервере React-проект? 38](#_Toc156767267)

[18. Для чего нужен Node.js? Какие у него особенности? 40](#_Toc156767268)

[19. Опишите механизм работы метода render? 42](#_Toc156767269)

[20. Опишите, каким образом можно использовать массивы в качестве состояний компонентов? 43](#_Toc156767270)

[22. В чем особенность декларативного подхода по сравнению с императивным. Приведите примеры. 47](#_Toc156767271)

[23. Для чего нужно экспортировать компоненты в React? 48](#_Toc156767272)

[24. Каким образом представление и логика работы в React соединяются в одном месте? Приведите примеры? 49](#_Toc156767273)

[25. Каким образом можно организовать работу компонент со связными состояниями? 51](#_Toc156767274)

# 1. Что такое React? С какой целью он создавался?( Для чего нужен React)

React.js — это библиотека для языка программирования [JavaScript](https://blog.skillfactory.ru/glossary/javascript/) с открытым исходным кодом для разработки пользовательских интерфейсов. Она помогает быстро и легко реализовать реактивность — явление, когда в ответ на изменение одного элемента меняется все остальное.

Для чего нужен React

React используют для создания одностраничных и многостраничных приложений, разработки крупных сайтов. Например, с его помощью написан стриминговый сервис Netflix и реализованы новостные ленты крупнейших социальных сетей. Библиотека предназначена:

* для создания функциональных интерактивных веб-интерфейсов, работая с которыми, не нужно постоянно обновлять страницу;
* быстрой и удобной реализации отдельных компонентов и страниц целиком — элементы в React легко использовать повторно;
* легкой разработки сложных программных структур — их просто описывать, если использовать реализованный в React подход;
* доработки новой функциональности с любым изначальным стеком технологий: библиотека не зависит от остального инструментария и будет хорошо работать, на чем бы ни был написан код;
* разработки одностраничных и многостраничных приложений (SPA и PWA). Это приложения, которые функционируют как программы и веб-сервисы и имеют соответствующий интерфейс;
* работы с серверной частью сайта или разработки мобильных приложений. В таких случаях React используют совместно с инструментами, адаптирующими веб-технологии под другие цели.

# 2. Что такое виртуальный DOM? Как виртуальный DOM повышает производительность React? В чем различие между реальной модель DOM?

Виртуальный DOM

В React для каждого объекта настоящего DOM (далее — RDOM) существует соответствующий объект VDOM. VDOM — это объектное представление RDOM, его легковесная копия. VDOM содержит те же свойства, что и RDOM, но не может напрямую влиять на то, что отображается на экране.

Виртуальный DOM (VDOM) — это концепция программирования, где идеальное или «виртуальное» представление UI хранится в памяти и синхронизируется с «реальным» DOM, используемая такими библиотеками, как ReactDOM. Данный процесс называется [согласованием (reconcilation)](https://ru.reactjs.org/docs/reconciliation.html).

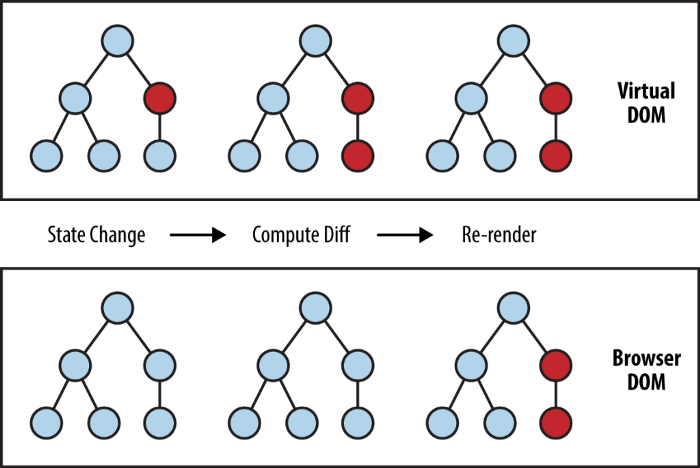
Манипуляции с RDOM являются медленными. Манипуляции с VDOM намного быстрее, поскольку они не отображаются (отрисовываются) на экране. Манипуляции с VDOM похожи на работу с проектом (или планом) здания перед началом его возведения.

Почему VDOM является более быстрым?

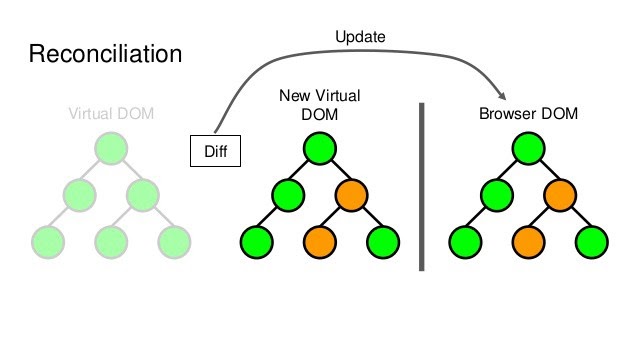
Когда в UI добавляются новые элементы, создается VDOM в виде дерева. Каждый элемент является узлом этого дерева. При изменении состояния любого элемента, создается новое дерево. Затем это новое дерево сравнивается (diffed) со старым.

После этого вычисляется наиболее эффективный метод внесения изменений в RDOM. Цель данных вычислений состоит в минимизации количества операций, совершаемых с RDOM. Тем самым, уменьшаются накладные расходы, связанные с обновлением RDOM.

На изображениях ниже представлено виртуальное DOM-дерево и процесс согласования.



Красным цветом обозначены узлы, которые были обновлены. Эти узлы представляют элементы UI, состояние которых изменилось. После этого вычисляется разница между предыдущей и текущей версиями виртуального DOM-дерева. Затем все родительское поддерево подвергается повторному рендерингу для представления обновленного UI. Наконец, это обновленное дерево используется для обновления RDOM.



Перевод от яндекса

**Различия между реальным Dom и виртуальным Dom**

| **Реальный DOM** | **Виртуальный DOM** |
| --- | --- |
| Реальный DOM представляет реальную структуру веб-страницы. | Виртуальный DOM представляет собой представление веб-страницы в виртуальной памяти. |
| Манипулирование DOM обходится очень дорого | Манипулировать DOM очень просто |
| Слишком большая потеря памяти | Отсутствие потерь памяти |
| Он обновляется медленно | Он быстро обновляется |
| Он может напрямую обновлять HTML | Он не может обновлять HTML напрямую |
| Создает новый DOM, если элемент обновляется. | Обновите JSX, если элемент обновится |
| Это позволяет нам напрямую нацеливаться на любой конкретный узел (HTML-элемент) | Он может производить около 200 000 виртуальных DOM Узлов в секунду. |
| Он представляет Ul вашего приложения | Это всего лишь виртуальное представление DOM |

# 3. Перечислите некоторые из основных преимуществ React. Приведите не менее трех примеров реализации (код).

#чисто через чатгпт

1. **Компонентный подход**: React использует компонентный подход, который помогает в построении переиспользуемого UI-кода. Это упрощает разработку и обслуживание кода.
2. **Виртуальный DOM (Virtual DOM)**: React создает легковесное представление страницы в памяти, которое называется "виртуальным DOM". Это позволяет библиотеке оптимизировать обновление DOM и повышает производительность приложения.
3. **Удобство изоляции данных и представления**: благодаря использованию состояний (state) и свойств (props), React обеспечивает четкое разделение между логикой обработки данных и их отображением, что упрощает понимание и тестирование компонентов.

**Пример 1: Создание Простого Компонента**

import React from 'react';

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

export default Welcome;

Этот пример показывает, как создать простой функциональный компонент в React, который принимает свойства (**props**) и отображает их.

**Пример 2: Использование Состояния (State)**

import React, { useState } from 'react';

function Counter() {

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

return (

<div>

<p>Вы кликнули {count} раз(а)</p>

<button onClick={() => setCount(count + 1)}>

Кликни меня

</button>

</div>

);

}

export default Counter;

Здесь мы используем хук **useState**, чтобы создать состояние компонента. Этот компонент показывает счетчик, который увеличивается при каждом нажатии на кнопку.

**Пример 3: Использование Жизненного Цикла Компонента**

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

function UserStatus(props) {

const [isOnline, setIsOnline] = useState(null);

useEffect(() => {

function handleStatusChange(status) {

setIsOnline(status.isOnline);

}

ChatAPI.subscribeToFriendStatus(props.friendID, handleStatusChange);

return () => {

ChatAPI.unsubscribeFromFriendStatus(props.friendID, handleStatusChange);

};

});

if (isOnline === null) {

return 'Загрузка...';

}

return isOnline ? 'Онлайн' : 'Оффлайн';

}

export default UserStatus;

В этом примере мы используем хук **useEffect**, чтобы выполнить подписку и отписку от сервиса, который отслеживает статус пользователя. Это демонстрирует использование жизненного цикла компонента в функциональном стиле.

# 4. Чем отличаются setState и useState? Что такое хук? Какие виды хуков существуют? Приведите пример использования.

**setState** и **useState** - это функции, используемые в React для управления состоянием компонентов. Однако они используются в разных контекстах:

1. **setState**: Это функция классовых компонентов React. Она используется для обновления состояния компонента и вызывает повторный рендеринг. **setState** принимает объект с новым состоянием или функцию, возвращающую новое состояние.

Пример использования в классовом компоненте:

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

incrementCount = () => {

this.setState({ count: this.state.count + 1 });

};

render() {

return (

<div>

<p>{this.state.count}</p>

<button onClick={this.incrementCount}>Увеличить</button>

</div>

);

}

}

**useState**: Это хук, введенный в React 16.8, позволяющий использовать состояние в функциональных компонентах. **useState** возвращает массив из двух элементов: текущее состояние и функцию для его обновления.

Пример использования в функциональном компоненте:

import React, { useState } from 'react';

function MyComponent() {

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

return (

<div>

<p>{count}</p>

<button onClick={() => setCount(count + 1)}>Увеличить</button>

</div>

);

}

**Хук** в React - это специальная функция, которая позволяет "подключаться" к функциональностям React (например, к состоянию и жизненному циклу компонентов) из функциональных компонентов. Хуки были добавлены для того, чтобы дать функциональным компонентам возможности, ранее доступные только классовым компонентам.

Виды основных хуков в React:

1. **useState**: Управление состоянием компонента.
2. **useEffect**: Выполнение побочных эффектов в компоненте (например, подписки, запросы данных, ручные изменения DOM).
3. **useContext**: Дает доступ к данным React Context без использования Consumer.
4. **useReducer**: Альтернатива useState для управления более сложным состоянием.
5. **useCallback**: Запоминает функцию, чтобы предотвратить ее пересоздание на каждый рендер.
6. **useMemo**: Запоминает результат вычислений и используется для оптимизации производительности.
7. **useRef**: Управляет ссылками на DOM-элементы и сохраняет мутабельные данные.

Пример использования **useEffect**:

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

function MyComponent() {

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

useEffect(() => {

fetchData().then(setData);

}, []); // Пустой массив зависимостей означает, что эффект выполнится один раз после монтирования компонента

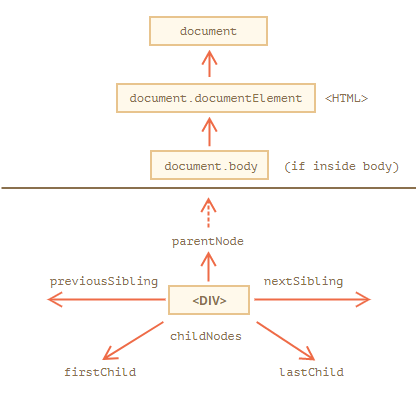
return <div>{data}</div>;

}

# 5. Как организовать ссылку на узел DOM?

Все операции с DOM начинаются с объекта document. Это главная «точка входа» в DOM. Из него мы можем получить доступ к любому узлу.

Так выглядят основные ссылки, по которым можно переходить между узлами DOM:



Создание ссылок

Ссылки создаются с использованием React.createRef() и добавляются к React-элементам с помощью атрибута ref. Ссылки обычно присваиваются свойству экземпляра, когда компонент создаётся таким образом, чтобы на них можно было ссылаться по всему компоненту.

class MyComponent extends React.Component {

constructor(props) {

super(props);

this.myRef = React.createRef();

}

render() {

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

}

}

Доступ к ссылкам

Когда ссылка передаётся элементу в render(), доступ к её узлу можно получить в атрибуте current.

const node = this.myRef.current;

Значение ссылки отличается в зависимости от типа узла:

* Когда атрибут ref используется в HTML-элементе, ref, созданный в конструкторе с помощью React.createRef(), получает базовый элемент DOM в качестве своего свойства current.
* Когда атрибут ref используется на пользовательском классовом компоненте, объектref получает примонтированный экземпляр компонента в качестве своего свойства current.
* **Вы не можете использовать атрибут ref в функциональных компонентах**, потому что у них не может быть экземпляров.

Приведённые ниже примеры демонстрируют данные различия.

Добавление ссылки на DOM-элемент

Этот код использует ref для сохранения ссылки на узел DOM:

class CustomTextInput extends React.Component {

constructor(props) {

super(props);

// Создание ссылки для сохранения DOM-элемента textInput

this.textInput = React.createRef();

this.focusTextInput = this.focusTextInput.bind(this);

}

focusTextInput() {

// Явная установка фокуса на поле ввода текста с использованием непосредственно API DOM

// Примечание: мы получаем доступ к "current" для получения DOM-узла

this.textInput.current.focus();

}

render() {

// Указать React, что мы хотим связать ссылку <input>

// со свойством `textInput`, созданного в конструкторе

return (

<div>

<input

type="text"

ref={this.textInput} />

<input

type="button"

value="Установить фокус на поле ввода текста"

onClick={this.focusTextInput}

/>

</div>

);

}

}

При монтировании компонента React присваивает свойству current элемент DOM, и назначить ему обратно значение null при размонтировании компонента. Обновления ref происходят перед выполнением хук жизненного цикла componentDidMount или componentDidUpdate.

# 6. Для чего нужны реквизиты? Как они используются в React-компонентах?

**Реквизиты**

Термин Props — это сокращение для свойств.

Они неизменяемы, это означает, что они не могут изменяться. Функциональный компонент получает их в качестве параметров. Изменение реквизитов вызывает обновление рендеринга компонента.

Здесь вы видите, как наш компонент ItemsList получает реквизит — список строк.

import { FunctionComponent, useState } from "react";

interface ItemsListProps {

list: Array<string>;

}

const ItemsList: FunctionComponent<ItemsListProps> = ({ list }) => {

console.log("ItemsList rerender");

return (

<ul>

{list.map((item) => (

<li>{item}</li>

))}

</ul>

);

};

function App() {

const [list, setList] = useState(["item1", "item2", "item3"]);

const handleClick = () => {

setList([...list, "item4"]);

};

return (

<div className="App">

<button onClick={handleClick}>Add</button>

<ItemsList list={list} />

</div>

);

}

export default App;

Если вы нажмете на кнопку, вы добавите элемент в список и обновите реквизит. Компонент выполняется, чтобы сообщить об изменении. Вы можете проверить его в консоли браузера.

# 7. Что такое JSX? В чем преимущество его использования?

JSX — это препроцессор, добавляющий синтаксис XML в JavaScript, отсюда и название JavaScript XML — JSX. Эта технология является вводной для понимания самого React и дает представление как устроены React приложения. Кто сталкивается впервые с JSX думают, что это странная смесь HTML и JavaScript, но это не совсем так, скорее совсем не так. Разработчики JSX создали синтаксис, который позволяет описывать UI в привычном для фронтенда HTML-подобном стиле, который затем с помощью транспиляции преобразуется в привычные объекты. Мы можем писать целые структуры элементов, не заботясь о том, как они будут преобразованы в DOM дерево и добавлены на страницу. За это будет отвечать сам JSX и React.

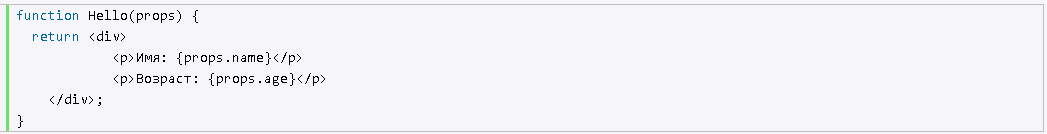
Преимущества использования JSX:

1. **Улучшенная читаемость кода**: JSX позволяет смешивать HTML и JavaScript, делая код более интуитивным и удобным для чтения, особенно когда вы создаёте сложные интерфейсы.
2. **Упрощение процесса разработки**: Поскольку JSX является похожим на HTML, разработчикам, знакомым с веб-разработкой, легче изучать и использовать React.
3. **Облегчение процесса отладки**: Ошибки в JSX легко отслеживаются, поскольку JSX транспилируется в JavaScript, и вы получаете ошибки, указывающие на конкретные строки в исходном коде JSX.
4. **Компонентный подход**: JSX идеально подходит для компонентного подхода React, позволяя описывать структуру интерфейса непосредственно в JavaScript-коде.
5. **Оптимизация производительности**: Поскольку JSX преобразуется в оптимизированный JavaScript, это обеспечивает более быстрое создание элементов DOM по сравнению с традиционным созданием HTML из JavaScript.
6. **Мощные возможности JavaScript**: JSX позволяет использовать всю мощь JavaScript, такую как логика программы и доступ к данным, непосредственно при построении интерфейса.

# 8. Что такое props в компоненте? Опишите механизм его работы. Приведите примеры использования.

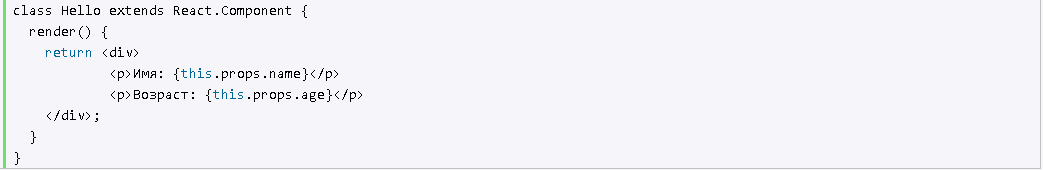
**Props** представляет коллекцию значений, которые ассоциированы с компонентом. Эти значения позволяют создавать динамические компоненты, которые не зависят от жестко закодированных статических данных.

Функциональный подход:



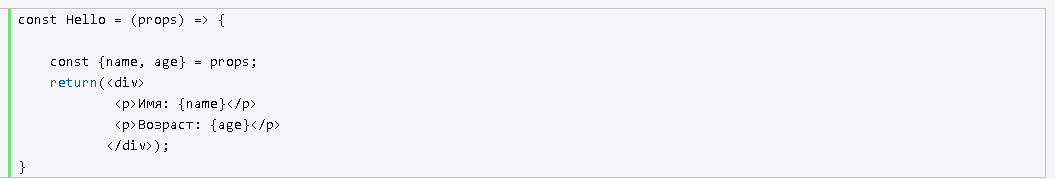
Параметр props, который передается в функцию компонента, инкапсулирует свойства объекта. В частности, свойство name и age. При рендеринге мы можем создать набор компонентов Hello, но передать в них разные данные для name и age. И таким образом, получим набор однотипной разметки html, наполненной разными данными.

Использование классов:



Класс компонента также извне получает объект свойств, который доступен через this.props.

Использование стрелочных функций (arrow functions):

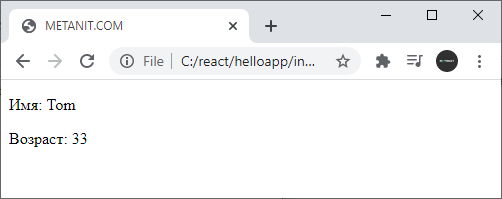


Используем компонент Hello:



При рендеринге React передает значения атрибутов в виде единого объекта "props". То есть значение из атрибута name="Tom" перейдет в свойство props.name.

В итоге будет создана следующая страница:



# 9. Как мы можем условно отрисовать JSX-компонент в React? Приведите примеры полного и сокращенного синтаксиса.

React позволяет разделить логику на независимые компоненты. Эти компоненты можно показывать или прятать в зависимости от текущего состояния.

Условный рендеринг в React работает так же, как условные выражения работают в JavaScript. Бывает нужно объяснить React, как состояние влияет на то, какие компоненты требуется скрыть, а какие — отрендерить, и как именно. В таких ситуациях используйте условный оператор JavaScript или выражения подобные if.

**Сокращенный синтаксис с использованием &&**: Этот метод полезен, когда нужно отрисовать компонент только если условие истинно. Если условие ложно, React пропустит отрисовку компонента.

{условие && <Компонент />}

Пример:

const isLoggedIn = true;

return (

<div>

{isLoggedIn && <WelcomeMessage />}

</div>

);

В этом примере компонент **WelcomeMessage** будет отрисован, только если переменная **isLoggedIn** истинна.

**Полный синтаксис с использованием тернарного оператора**: Этот метод позволяет выбрать один из двух компонентов для отрисовки в зависимости от условия.

{условие ? <Компонент1 /> : <Компонент2 />}

const isLoggedIn = true;

return (

<div>

{isLoggedIn ? <WelcomeMessage /> : <LoginButton />}

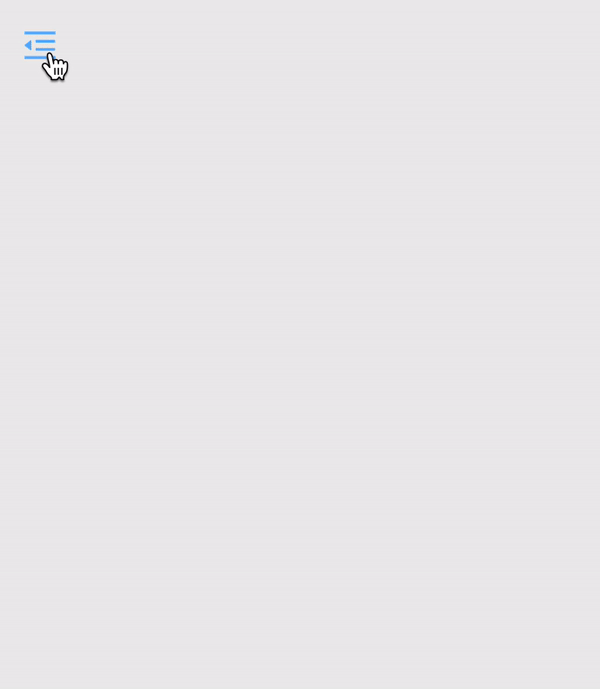
</div>

);

Здесь, если **isLoggedIn** истинно, отрисовывается **WelcomeMessage**, иначе - **LoginButton**.

# 10. Каким способом можно создавать анимации в React? Приведите примеры.

CSS

Как раз об этом методе говорилось в самом начале, и он действительно хорош. Если вместо того, чтобы импортировать библиотеки JavaScript, использовать его, сборка будет небольшой, браузеру не потребуется много ресурсов. А это, конечно же, влияет на производительность приложений. Если ваша анимация должна быть относительно простой, обратите на этот метод внимание.  
  
В качестве примера — анимированное меню:   
  
  
  
Оно относительно простое, со свойством CSS и триггером типа className = «is-nav-open» для тега HTML.   
  
Использовать этот метод можно разными способами. Например, создать над навигацией wrapper, а затем вызывать изменения полей. Поскольку навигация имеет постоянную ширину, которая равна 250px, ширина wrapper со свойством margin-left или translateX должна иметь ту же ширину. При необходимости показать навигацию нужно добавить className = «is-nav-open» для wrapper и переместить wrapper на margin-left / translateX: 0;.  
  
В конечном итоге исходник анимации будет выглядеть следующим образом:

**export** **default** **class** **ExampleCss** **extends** **Component** {

**handleClick**() {

**const** wrapper = document.getElementById('wrapper');

wrapper.classList.toggle('is-nav-open')

}

**render**() {

**return** (

<**div** id="wrapper" className="wrapper">

<**div** className="nav">

<**icon**

className="nav\_\_icon"

type="menu-fold"

onClick={() => this.handleClick()}/>

<**div** className="nav\_\_body">

Lorem ipsum dolor sit amet, consectetur adipisicing elit.

Beatae ducimus est laudantium libero nam optio repellat

sit unde voluptatum?

</**div**>

</**div**>

</**div**>

);

}

}

А вот CSS-стили:

.wrapper {

**display**: flex;

**width**: 100%;

**height**: 100%;

**transition**: margin .5s;

**margin**: 0 0 0 -250px;

}

.wrapper.is-nav-open {

**margin-left**: 0;

}

.nav {

**position**: relative;

**width**: 250px;

**height**: 20px;

**padding**: 20px;

**border-right**: 1px solid #ccc;

}

.nav\_\_icon {

**position**: absolute;

**top**: 0;

**right**: -60px;

**padding**: 20px;

**font-size**: 20px;

**cursor**: pointer;

**transition**: color .3s;

}

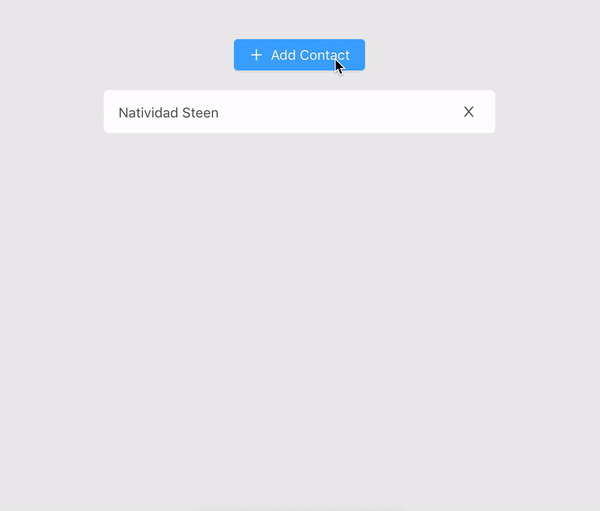
.nav\_\_icon:hover {

**color**: #5eb2ff;

}

Повторюсь, если анимация относительно проста, то этот метод — основной. Пользователей порадует быстродействие браузера.

ReactTransitionGroup

Компонент ReactTransitionGroup разработала команда сообщества ReactJS. С его помощью можно без проблем реализовать основные CSS-анимации и переходы.  
  
ReactTransitionGroup предназначен для изменения классов при изменении жизненного цикла компонента. У него небольшой размер, его нужно установить в пакете для React-приложения, что незначительно увеличит общий размер сборки. Кроме того, можно использовать и CDN.  
  
У ReactTransitionGroup есть три элемента, это Transition, CSSTransition и TransitionGroup. Для запуска анимации в них нужно обернуть компонент. Стиль, в свою очередь, нужно прописывать в классах CSS.  
  
Вот анимация, а дальше — способ ее реализации.  
  
  
  
Первым делом нужно импортировать CSSTransitionGroup из react-transition-group. После этого требуется обернуть список и установить свойство transitionName. Каждый раз при добавлении или удалении дочернего элемента в CSSTransitionGroup он получает анимационные стили.

<CSSTransitionGroup

transitionName="example">

{items}

</CSSTransitionGroup>

При установке свойства transitionName = «example» классы в таблицах стилей должны начинаться с имени примера.

.example-enter {

**opacity**: 0.01;

}

.example-enter.example-enter-active {

**opacity**: 1;

**transition**: opacity 300ms ease-in;

}

.example-leave {

**opacity**: 1;

}

.example-leave.example-leave-active {

**opacity**: 0.01;

**transition**: opacity 300ms ease-in;

Выше показан пример использования ReactTransitionGroup.  
  
Нужна еще и логика, причем два метода для реализации примера добавления списка контактов.  
  
Первый метод handleAdd — он добавляет новые контакты, получает случайное имя, которое затем помещает в массив state.items.  
  
Для удаления контакта по индексу в массиве state.items используется handleRemove.

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

**import** { CSSTransitionGroup } **from** 'react-transition-group'

**import** random **from** 'random-name'

**import** Button **from** './button'

**import** Item **from** './item'

**import** './style.css';

**export** **default** **class** **ReactTransitionGroup** **extends** **Component** {

**constructor**(props) {

super(props);

this.state = { items: ['Natividad Steen']};

this.handleAdd = this.handleAdd.bind(this);

}

**handleAdd**() {

**let** newItems = this.state.items;

newItems.push(random());

this.setState({ items: newItems });

}

render () {

**const** items = this.state.items.map((item, i) => (

<**Item**

item={item}

key={i}

keyDelete={i}

handleRemove={(i) => this.handleRemove(i)}

/>

));

**return** (

<**Fragment**>

<**Button** onClick={this.handleAdd}/>

<**div** className="project">

<**CSSTransitionGroup**

transitionName="example"

transitionEnterTimeout={500}

transitionLeaveTimeout={300}

>

{items}

</**CSSTransitionGroup**>

</**div**>

</**Fragment**>

);

}

};

# 11. Почему браузер не может читать JSX? Какие библиотеки для интерпретации вам известны?

Браузеры нативно не могут читать JSX, потому что JSX — это синтаксический сахар для JavaScript, используемый в основном в библиотеке React для удобства описания интерфейсов. JSX выглядит как HTML, но не является им. В итоге, браузеры понимают только чистый JavaScript, а не JSX.

Для того чтобы браузеры могли работать с JSX, этот код должен быть транспилирован (преобразован) в обычный JavaScript. Для этой цели используются специальные инструменты. Самые известные из них:

1. **Babel**: Это транспилятор, который преобразует JSX в JavaScript. Babel также широко используется для преобразования современного JavaScript (ES6 и выше) в более старую версию (ES5), которая поддерживается большинством браузеров.
2. **Webpack**: Хоть Webpack сам по себе не транспилирует JSX, он часто используется вместе с Babel. Webpack — это сборщик модулей, который помогает упаковывать разные части приложения в один или несколько файлов JavaScript, включая преобразование JSX с помощью Babel.
3. **Create React App**: Это среда для создания React-приложений, которая уже включает в себя Babel и Webpack, настроенные для работы с JSX. Это удобный инструмент для быстрого старта проектов на React без необходимости вручную настраивать Babel и Webpack.

# 12. Как можно объяснить фразу «В React все является компонентом».

Фраза "В React все является компонентом" отражает ключевую концепцию библиотеки React для разработки пользовательских интерфейсов. Вот несколько аспектов, которые помогут понять эту фразу:

1. **Модульность**: В React приложения строятся из маленьких, независимых частей, называемых компонентами. Каждый компонент представляет собой часть пользовательского интерфейса, и весь интерфейс строится путем комбинирования этих компонентов.
2. **Переиспользование**: Компоненты созданы для того, чтобы их можно было легко переиспользовать в разных частях приложения или даже в разных приложениях. Это упрощает разработку и поддержку кода.
3. **Инкапсуляция**: Каждый компонент инкапсулирует свое состояние и поведение. Это означает, что компоненты могут управлять своим внутренним состоянием и логикой, оставаясь при этом отделенными от остальной части приложения.
4. **Декларативность**: В React вы описываете, как компоненты должны выглядеть и вести себя, и библиотека заботится об обновлении DOM в соответствии с этими описаниями. Это упрощает процесс разработки, так как вам не нужно вручную управлять изменениями в DOM.
5. **Составные компоненты**: Компоненты могут включать в себя другие компоненты, что позволяет создавать сложные пользовательские интерфейсы из простых строительных блоков.
6. **Универсальность**: Компоненты в React могут быть как простыми, как кнопка или поле ввода, так и очень сложными и крупномасштабными, как целые страницы или даже приложения.

13. Как встроить два или более компонентов в один компонент?

С помощью дочерних компонентов

Вложенные компоненты

Вложенный компонент — это любой дочерний компонент, связанный с родительским компонентом. Эта связь между дочерними и родительскими компонентами формируется посредством композиции, а не наследования. Это означает, что вместо того, чтобы наследовать один компонент от другого, каждый компонент создается путем сборки более мелких компонентов.

Несмотря на то, что вложение компонентов помогает вам структурировать ваш код намного чище, если вы не сделаете вложение компонентов правильно, вы получите гораздо более сложный и менее эффективный код. Поэтому давайте рассмотрим различные стили вложения компонентов, когда их использовать и правила, которым нужно следовать.

Вложение без дочерних компонентов

При разработке презентационных компонентов этот стиль вложения идеален. Таким образом, вы можете эффективно отделять элементы пользовательского интерфейса и быстро передавать данные в качестве свойств вложенным компонентам. Взгляните на пример кода ниже.



В этом примере компонент Card вложен в компонент App. Вы можете поместить столько компонентов Card, сколько хотите, внутрь компонента App. Кроме того, этот дизайн дает больше возможностей дочернему компоненту.

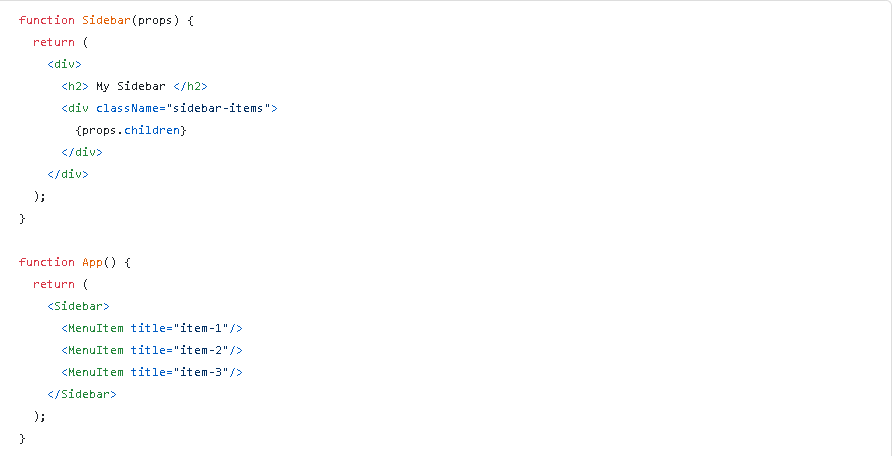
Однако этот тип вложения имеет более низкую возможность повторного использования компонентов и видимость архитектуры компонентов. Поэтому было бы лучше, если бы вы также проявили осторожность и избегали создания глубоко вложенных компонентов.

Глубоко вложенные компоненты приведут к тому, что данные будут передаваться через компоненты, которым не требуются эти конкретные данные. В результате отслеживание потока данных по мере роста приложения может стать затруднительным. Однако, если вы используете инструмент управления состоянием, например Redux, этого можно избежать. Поэтому, если в вашем проекте используется Redux, этот стиль вложения предпочтительнее.

Вложение с дочерними компонентами

При разработке таких компонентов, как Sidebar или Dialog, представляющих общие «блоки», вы можете не знать заранее всех их дочерних элементов. В таких случаях предпочтительнее передавать компоненты как дочерние элементы.

Свойство children позволяет вам получить доступ к этим переданным дочерним элементам во вложенном компоненте. Взгляните на код ниже, чтобы понять его лучше.



В этом примере компоненты MenuItem вложены как дочерние элементы компонента Sidebar. Этот способ вложения обеспечивает большую возможность повторного использования компонентов и наглядность архитектуры компонентов. Это также помогает избежать глубоко вложенных компонентов по мере роста вашего приложения. Однако в таком дизайне дочерние компоненты будут менее гибкими и мощными, а код родительского компонента может стать длинным.

Если ваш проект не использует Redux или Context API для управления состояниями, этот стиль вложения компонентов предпочтительнее, чтобы поддерживать чистоту вашего дерева компонентов.

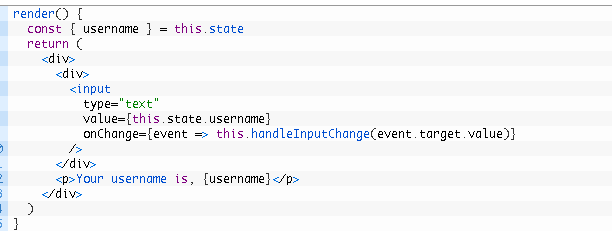
# 14. Как обновить состояния компонентов?

Обновление состояния

Единственный допустимый способ обновления состояния компонента – через setState(). Разберем это на примере. Во-первых, начну с создания метода, который будет вызываться для обновления username компонента. Этот метод должен получить аргумент, который будет использовать для обновления состояния.



Еще раз, обратите внимание, что я передаю объект в setState(). Далее мне необходимо передать эту функцию в обработчик события, который вызывается при изменении значения поля ввода. Обработчик события предоставит контекст события, которое было вызвано, что позволяет получить значение, введенное в поле, через event.target.value. Это аргумент, переданный в метод handleInputChange(). Метод render должен выглядеть следующим образом.



При каждом вызове setState() в React отправляется запрос на обновление DOM через обновленное состояние. Теперь вы понимаете, что обновление состояния можно отложить.

Компонент должен выглядеть следующим образом:



# 15. Как стрелочные функции работают в React? Как их передавать?

Стрелочные функции в React работают так же, как и в JavaScript.

Стрелочные функции

Начнем со стрелочных функций, синтаксис у них такой () => {}, круглые скобки не обязательны в некоторых случаях, так же как и фигурные скобки.

Разберем круглые скобки. Их можно не писать когда у функции один аргумент. Но если у функции нет аргументов, то скобки обязательны

Пример:

state **=>** {

**return** state

}

Более практический пример:

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

**const** arr2 **=** arr.map(item **=>** {

**return** item **+** 1

})

console.log(arr2) *// [2, 3, 4, 5, 6];*

Во 2 примере мы перебираем массив, с помощью метода *map* и если нам надо выводить еще и индекс элемента, то скобки писать придеться

Пример:

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

**const** arr2 **=** arr.map((item, index) **=>** {

**return** item **+** index

})

console.log(arr2) *// [1, 3, 5, 7, 9];*

На этом с круглыми скобками мы закончим, на очереде у нас фигурные скобки, с ними немного посложнее будет

Вот 2 примера:

**const** myFunc **=** (arr) **=>** {

**return** arr.push("some string");

}

**const** myFunc2 **=** (arr) **=>** arr.push("some string");

Эти 2 функции одинаковы, хотя и пишуться немного по разному. Главное правило, если вы сомневаетесь писать скобки или нет, это если нету скобок то функция автоматически возвращает все что после знака “стрелки” =>. Это можно использовать при написании более короткого кода.

**Особенности стрелочных функций в React:**

1. **Контекст this:** Стрелочные функции не создают собственный контекст **this**, так что **this** внутри стрелочной функции ссылается на контекст родительской функции или компонента. Это удобно, когда тебе нужно обращаться к свойствам или методам компонента.
2. **Краткость:** Стрелочные функции позволяют писать более лаконичный и читаемый код, особенно когда функция проста.

**Как передавать стрелочные функции:**

1. **В обработчиках событий:** Можно определить стрелочную функцию прямо в JSX. Например, **<button onClick={() => this.handleClick()}>Click me</button>**. Это удобно, но может привести к ненужному ререндерингу, поскольку каждый раз создаётся новая функция.
2. **Как методы класса:** Можно определить стрелочную функцию как свойство класса. Например:

class MyComponent extends React.Component {

handleClick = () => {

console.log('Clicked!');

};

render() {

return <button onClick={this.handleClick}>Click me</button>;

}

}

Здесь **handleClick** автоматически связан с контекстом экземпляра **MyComponent**.

**3. Передача в дочерние компоненты:** Стрелочные функции можно передавать в дочерние компоненты как пропсы. Это позволяет дочерним компонентам вызывать функции родительского компонента.

**Важные моменты:**

* Используй стрелочные функции осторожно, особенно в методах **render**, так как они могут приводить к ненужным ререндерам из-за постоянного создания новых экземпляров функций.
* Если функция не использует **this** или не передаётся вниз как проп, можно использовать обычные функции для лучшей производительности.

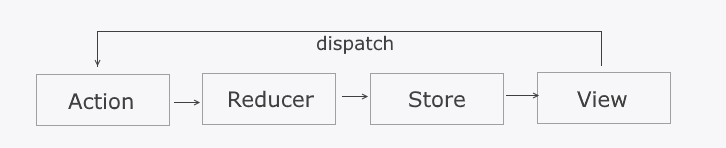
# 16. Что такое Redux? Для чего он нужен?

**Redux** представляет собой контейнер для управления состоянием приложения и во многом напоминает Flux. Redux не привязан непосредственно к React.js и может также использоваться с другими js-библиотеками и фреймворками.

Ключевые моменты Redux:

* **Хранилище** (store): хранит состояние приложения
* **Действия** (actions): некоторый набор информации, который исходит от приложения к хранилищу и который указывает, что именно нужно сделать. Для передачи этой информации у хранилища вызывается метод **dispatch()**.
* **Создатели действий** (action creators): функции, которые создают действия
* **Reducer** : функция (или несколько функций), которая получает действие и в соответствии с этим действием изменяет состояние хранилища

Общую схему взаимодействия элементов архитектуры Redux можно выразить следующим образом:



Из View (то есть из компонентов React) мы посылаем действие, это действие получает функция reducer, которая в соответствии с действием обновляет состояние хранилища. Затем компоненты React применяют обновленное состояние из хранилища.

Actions

Действия (actions) - это объекты, которые помогают доставить данные из компонентов в хранилище.

Хранят минимально необходимый набор информации.

Должны иметь свойство type, которое указывает тип выполняемого действия.

Помимо поля type, структура действия может быть произвольной.

const action = {

type: 'ADD\_NOTE',

payload: {

text: 'Redux is awesome!',

},

};

Действия создаются функциями (action creators), которые могут быть асинхронными и иметь побочные эффекты. В базовом варианте они просто возвращают объект-дейтсвие.

const addNote = text => ({

type: 'ADD\_NOTE',

payload: {

id: Date.now(),

text,

},

});

const deleteNote = id => ({

type: 'DELETE\_NOTE',

payload: {

id,

},

});

**Reducers**

**Редюсеры (reducers)** - это чистые функции, которые принимают предыдущее состояние приложения и действие, а затем возвращают новое следующее состояние.

Они определяют, как изменяется состояние приложения в ответ на действия, отправленные в хранилище. Помните, что действия описывают только то, что произошло, а не как изменяется состояние приложения.

(previousState, action) => newState

Вещи, которые нельзя делать внутри редюсера:

* Мутировать аргументы
* Выполнять побочные эффекты, такие как API-запросы и т. п.
* Вызывать нечистые функции, например Date.now()

Как выполнять побочные эффекты мы рассмотрим далее, пока что просто помните - редюсер должен быть чистым. Получая те же аргументы, он должен вычислить следующее состояние и вернуть его. Без сюрпризов. Никаких побочных эффектов. Никаких мутаций. Просто рассчет.

Вот редьюсер, который принимает текущее состояние и действие как аргументы, а затем возвращает следующее состояние.

const initialState = [];

function notesReducer(state = initialState, action) {

switch (action.type) {

case 'ADD\_NOTE':

return [...state, action.payload];

case 'DELETE\_NOTE':

return state.filter(note => note.id !== action.payload.id);

default:

return state;

}

}

Обратите внимание:

* Мы создаем копию state, а не мутируем его.
* Мы возвращаем предыдущее состояние по умолчанию. Важно вернуть предыдущее состояние для любого неизвестного действия.

**3. Store**

**Хранилище (store)** - js-объект, который содержит состояние приложения и методы доступа к нему, отправки действий и регистрации слушателей.

* Хранит состояние (state) приложения как один объект
* Позволяет получить доступ к состоянию через метод getState()
* Напрямую состояние доступно только для чтения
* Единственный способ изменить состояние - отправить действие (action), объект, описывающий, что произошло
* Для отправки дейсвтий есть метод dispatch(action)
* Изменения производятся с использованием чистых функций - редюсеров (reducers), которые реагируют на действия
* Регистрирация слушателей делается методом subscribe(listener)

Так как все состояние приложения хранится как один объект, стоит подумать о форме состояния прежде чем писать какой-либо код. Продумайте минимальное представление состояния приложения в виде объекта.

const state = {

notes: [],

filter: '',

session: {

user: {

name: null,

avatar: null,

},

token: null,

},

};

**3.1. Функция createStore**

Для того чтобы создать хранилище, используется функция createStore, которая принимает набор параметров и возвращает созданное хранилище.

createStore(reducer, [preloadedState], [enhancer])

* reducer - функция, которая возвращает следующее дерево состояния, учитывая текущее дерево состояния и действие для обработки.
* preloadedState - начальное состояние, к примеру сериализаванное состояние последнего пользовательского сеанса. Это должен быть объект той же формы, что и, как минимум, часть состояния.
* enhancer - расширяет возможности хранилища при помощи прослоек (middleware).

import { createStore } from 'redux';

// Используем редюсер-болванку

const reducer = (state = {}, action) => state;

const store = createStore(reducer);

# 17. Какие команды библиотеки npm нужно использовать, чтобы создать, запустить и подготовить к размещению на сервере React-проект?

Для создания, запуска и подготовки к размещению на сервере проекта на React с использованием npm, вам потребуется выполнить следующие команды:

1. **Создание нового React-проекта:**
   * Сначала установите **create-react-app**, если у вас его ещё нет. Это глобальный пакет, который помогает начать новый проект React:

luaCopy code

npm install -g create-react-app

* + Затем создайте новый проект React:

luaCopy code

npx create-react-app имя\_вашего\_проекта

Здесь **имя\_вашего\_проекта** - это название вашего нового проекта.

1. **Запуск проекта для разработки:**
   * Перейдите в каталог вашего проекта:

bashCopy code

cd имя\_вашего\_проекта

* + Запустите проект в режиме разработки:

sqlCopy code

npm start

Эта команда запустит локальный сервер разработки, и вы сможете увидеть ваше приложение в браузере, обычно по адресу **http://localhost:3000**.

1. **Сборка проекта для размещения на сервере:**
   * Когда вы закончите разработку и будете готовы опубликовать свой проект, вы можете создать оптимизированную версию вашего приложения, используя:

arduinoCopy code

npm run build

Эта команда создаст папку **build** в корне вашего проекта, которая содержит все статические файлы, необходимые для размещения на сервере.

1. **Размещение на сервере:**
   * Содержимое папки **build** готово к размещению на любом статическом хостинге. Вы можете использовать сервисы, такие как Netlify, Vercel, GitHub Pages или любой другой веб-сервер, который поддерживает статический хостинг.

# 18. Для чего нужен Node.js? Какие у него особенности?

**Что такое Node.js?**

Node.js (или просто Node) — это серверная платформа для работы с [JavaScript](https://netology.ru/programs/javascript) через движок V8. JavaScript выполняет действие на стороне клиента, а Node — на сервере. С помощью Node можно писать полноценные приложения. Node умеет работать с внешними библиотеками, вызывать команды из кода на JavaScript и выполнять роль веб-сервера.

Особенности и преимущества Node.js

**Совместимость с JavaScript и TypeScript**

Node.js обеспечивает возможность разработки приложений с использованием двух распространенных языков программирования — JavaScript и TypeScript. А это огромное преимущество, учитывая высокую популярность и востребованность этих языков на рынке труда.

Совместимость с JavaScript и TypeScript делает процесс разработки более удобным и доступным, позволяя разработчикам использовать в новых проектах уже сформированные навыки и опыт работы.

**Асинхронная и событийно-ориентированная природа**

Асинхронность — это одна из ключевых характеристик Node.js. Термин «асинхронный» в контексте серверов, созданных с использованием Node.js, означает, что они не ожидают завершения операций ввода-вывода, прежде чем продолжить выполнение других задач.

Кроме того, асинхронность подразумевает наличие неблокирующего ввода-вывода, что позволяет в процессе ожидания ответа на запрос, например, к базе данных выполнять другие операции и обрабатывать другие запросы.

**Однопоточность**

По умолчанию Node.js работает в однопоточном режиме, что означает, что все операции выполняются в одном основном потоке. Однако это не означает, что Node.js не может в достаточной мере обрабатывать множество одновременных соединений.

В настоящее время Node.js имеет эффективные встроенные средства для многопоточной и многопроцессной работы.

**Широкий выбор библиотек и модулей**

Node.js имеет огромное количество встроенных модулей, а также большое количество сторонних библиотек, доступных через пакетный менеджер NPM (Node Package Manager). Это позволяет разработчикам быстро интегрировать сторонние решения и ускоряет процесс разработки.

**Возможность создания API и серверов**

Node.js широко используется для создания веб-серверов и API. Его простая синтаксическая структура и возможность обработки запросов и ответов в режиме реального времени делают его отличным выбором для создания серверных приложений.

**Эффективная обработка событий**

Обработка событий означает, что код перед выполнением ждет некоего события. В Node.js, при начале выполнения какой-либо операции, можно сразу же обозначить функцию, которая должна быть выполнена после окончания данной задачи.

Такие функции называются функциями обратного вызова. Функции обратного вызова используют значительно меньше ресурсов на стороне сервера, а также занимают меньше памяти.

**Высокая производительность**

Благодаря асинхронной обработке и архитектуре V8, Node.js способен обрабатывать большое количество запросов с минимальной задержкой, что делает его хорошим выбором для создания высоконагруженных приложений.

**Создание приложений для реального времени**

Node.js отлично подходит для создания приложений, которые должны оперативно обрабатывать и передавать данные в реальном времени, такие как чат-серверы, онлайн-игры и системы мониторинга.

# 19. Опишите механизм работы метода render?

**render()**

render()

Метод render() — единственный обязательный методов в классовом компоненте.

При вызове он должен проверять this.props и this.state и возвращать один из следующих типов:

* **React-элементы.** Обычно создается через [JSX](https://ru.react.js.org/docs/introducing-jsx.html). Например, <div /> и <MyComponent /> — элементы React, которые указывают React отрисовывать DOM-узел или другой пользовательский компонент, соответственно.
* **Массивы и фрагменты.** Позволяет возвращать несколько элементов из метода render(). Подробнее смотрите документацию по [фрагментам](https://ru.react.js.org/docs/fragments.html).
* **Порталы**. Позволяет вам отрисовывать дочерние элементы в другое поддерево DOM. Подробнее смотрите документацию на [порталах](https://ru.react.js.org/docs/portals.html).
* **Строки и числа.** Они отрисовывается как текстовые узлы в DOM.
* **Логические значения или null**. Ничего не отрисовывает. (В основном существует для поддержки паттерна return test && <Child />, где test — логическое значение.)

Функция render() должна быть чистой, что означает, что она не изменяет состояние компонента, она возвращает один и тот же результат при каждом вызове и не взаимодействует напрямую с браузером.

Если вам нужно взаимодействовать с браузером, выполняйте все необходимые операции в componentDidMount() или других методах жизненного цикла. Сохранение render() чистым делает компонент легче для понимания.

**Примечание**

render () не будет вызываться, если [shouldComponentUpdate()](https://ru.react.js.org/docs/react-component.html#shouldcomponentupdate) возвращает false.

# 20. Опишите, каким образом можно использовать массивы в качестве состояний компонентов?

Массивы можно использовать в качестве состояний компонентов для управления коллекциями данных в приложениях. Например, представим, что у нас есть компонент веб-приложения, который отображает список задач. Мы можем использовать массив в состоянии компонента для хранения этих задач. Каждый элемент массива будет представлять отдельную задачу.

Вот как это может работать:

1. **Инициализация состояния**: Сначала мы инициализируем состояние компонента с пустым массивом или массивом начальных элементов. Это можно сделать в конструкторе классового компонента или используя хук **useState** в функциональном компоненте.

this.state = {

tasks: []

};

const [tasks, setTasks] = useState([]);

**2. Обновление состояния**: Когда нам нужно добавить новую задачу в список, мы можем использовать метод **setState** в классовом компоненте или функцию обновления состояния в хуке **useState**. Важно помнить, что состояние в React является неизменяемым, поэтому мы должны создавать новый массив, а не изменять существующий.

this.setState(prevState => ({

tasks: [...prevState.tasks, newTask]

}));

Или

setTasks(oldTasks => [...oldTasks, newTask]);

3. **Рендеринг**: В методе рендеринга или теле функционального компонента мы можем перебирать массив **tasks** и отображать каждую задачу. Это можно сделать, используя метод **map** для массивов.

<ul>

{this.state.tasks.map(task => <li key={task.id}>{task.text}</li>)}

</ul>

Или

<ul>

{tasks.map(task => <li key={task.id}>{task.text}</li>)}

</ul>

Использование массивов как состояний позволяет легко управлять коллекциями данных, такими как списки, галереи изображений, или даже сложные структуры данных в пользовательских интерфейсах.

21. С помощью какой библиотеки можно реализовать многостраничность в React-приложении? Приведите примеры?

С помощью библиотеки react-router-dom ( npm install react-router-dom)

**Настройка маршрутов**: После установки необходимо настроить маршруты в вашем приложении. Вы создаёте компоненты для каждой страницы, а затем используете **<Route>** для определения пути к этим компонентам. Для оборачивания всех маршрутов используется **<BrowserRouter>**.

Пример кода:

import React from 'react';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import HomePage from './HomePage';

import AboutPage from './AboutPage';

import ContactPage from './ContactPage';

function App() {

return (

<Router>

<Switch>

<Route exact path="/" component={HomePage} />

<Route path="/about" component={AboutPage} />

<Route path="/contact" component={ContactPage} />

</Switch>

</Router>

);

}

export default App;

В этом примере **<Switch>** используется для выбора только одного **<Route>** для отображения. Компоненты **HomePage**, **AboutPage** и **ContactPage** - это ваши страницы.

**Навигация между страницами**: Чтобы переходить между страницами, используйте компонент **<Link>** из **react-router-dom**.

Пример:

import { Link } from 'react-router-dom';

function Navbar() {

return (

<nav>

<Link to="/">Главная</Link>

<Link to="/about">О нас</Link>

<Link to="/contact">Контакты</Link>

</nav>

);

}

# 22. В чем особенность декларативного подхода по сравнению с императивным. Приведите примеры.

**Декларативное программирование** — это парадигма программирования … которая выражает логику вычисления без описания его потока управления.  
**Императивное программирование** — это парадигма программирования, в которой используются утверждения, которые изменяют состояние программы.

Продемонстрируй в коде

Ладно, ладно, вот несколько примеров кода. У нас есть кнопка, которая меняет цвет при нажатии. Я начну с императивного примера:

const container = document.getElementById(‘container’);  
const btn = document.createElement(‘button’);btn.className = ‘btn red’;  
btn.onclick = function(event) {  
 if (this.classList.contains(‘red’)) {  
 this.classList.remove(‘red’);  
 this.classList.add(‘blue’);  
 } else {  
 this.classList.remove(‘blue’);  
 this.classList.add(‘red’);  
 }  
};container.appendChild(btn);

И пример нашего декларатвного React:

class Button extends React.Component{this.state = { color: 'red' }handleChange = () => {  
 const color = this.state.color === 'red' ? 'blue' : 'red';  
 this.setState({ color });  
 }render() {  
 return (<div>  
 <button   
 className=`btn ${this.state.color}`  
 onClick={this.handleChange}>  
 </button>  
 </div>);  
 }  
}

Возможно, различия незначительны. У нас по-прежнему есть логика, которая говорит: если красный, значит синий, но есть одна огромная разница. Пример c React никогда не затрагивает элемент. Он просто объявляет, что элемент должен отображаться с учетом нашего текущего состояния (далее — state). Он фактически не манипулирует самим DOM.

# 23. Для чего нужно экспортировать компоненты в React?

**Польза экспорта компонентов в React.js**

Одна из главных польз экспорта компонентов — это возможность создания библиотеки компонентов. Это позволяет разработчикам объединять схожие компоненты в одной библиотеке и делиться ею с другими членами команды или сообществом разработчиков. Это существенно экономит время и силы, которые могут быть направлены на разработку других частей приложения.

Кроме того, экспорт компонентов предоставляет возможность реализации концепции компонентного подхода. Компонентный подход, основанный на разделении пользовательского интерфейса на отдельные независимые компоненты, позволяет создавать приложения, которые можно легко модифицировать и масштабировать. Компоненты могут быть использованы повторно в различных частях приложения и даже в других проектах.

Кроме того, экспорт компонентов снижает связность и повышает связь компонентов, что делает приложение более структурированным и понятным. Использование экспорта позволяет легко представлять и переиспользовать компоненты в разных частях приложения, что способствует повышению его гибкости и удобства разработки.

Итак, польза экспорта компонентов в React.js очевидна. Это позволяет создавать библиотеки компонентов, применять компонентный подход, а также повышает гибкость и структурированность приложения. Все это упрощает и ускоряет процесс разработки, делая React.js популярным выбором для разработки современных веб-приложений.

# 24. Каким образом представление и логика работы в React соединяются в одном месте? Приведите примеры?

В React представление и логика работы соединяются в компонентах. Компоненты — это основные строительные блоки в React, которые объединяют в себе как разметку (обычно используется JSX, который выглядит как HTML), так и JavaScript-код, который определяет логику поведения этой разметки.

пример простого компонента на React:

import React, { useState } from 'react';

function Counter() {

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

function handleIncrement() {

setCount(count + 1);

}

function handleDecrement() {

setCount(count - 1);

}

return (

<div>

<h1>Счётчик: {count}</h1>

<button onClick={handleIncrement}>Увеличить</button>

<button onClick={handleDecrement}>Уменьшить</button>

</div>

);

}

export default Counter;

В этом примере у нас есть компонент **Counter**, который реализует счётчик. Рассмотрим основные части:

1. **Импорт React и хука useState:** Это позволяет использовать в компоненте возможности React и его состояние.
2. **Функциональный компонент Counter:** Здесь определяется сам компонент. Внутри этой функции находится вся логика компонента.
3. **Хук useState:** Используется для создания переменной состояния **count** и функции **setCount** для её обновления. Это позволяет компоненту реагировать на изменения.
4. **Функции handleIncrement и handleDecrement:** Это функции, определённые внутри компонента, которые изменяют состояние **count**.
5. **JSX разметка:** Возвращает разметку, которая будет отображаться на странице. Здесь мы видим использование **{count}** для отображения текущего значения счётчика и обработчики событий **onClick**, привязанные к кнопкам.

# 25. Каким образом можно организовать работу компонент со связными состояниями?

Для организации работы компонент со связными состояниями можно использовать несколько подходов. Важно понимать, что связные состояния - это когда состояние одного компонента зависит или влияет на состояние другого. Вот несколько способов организации такой работы:

1. **Пропсы (Props):** Это один из самых простых способов. Мы передаем состояние из родительского компонента в дочерний через пропсы. Таким образом, дочерний компонент будет использовать или изменять состояние, которое приходит из родительского.
2. **Контекст (Context):** Это более продвинутый способ, который позволяет передавать данные напрямую между компонентами, минуя промежуточные уровни. Так, состояние, хранящееся в контексте, может быть доступно в любом компоненте, который подписан на этот контекст.
3. **Глобальное управление состоянием (например, Redux или MobX):** Это подходит для крупных приложений, где множество компонентов нуждаются в доступе к общим данным. Состояние хранится в глобальном хранилище, а компоненты подписываются на изменения в этом хранилище.
4. **Кастомные хуки (Custom Hooks):** Это способ создать свою логику управления состоянием, которую можно повторно использовать в разных компонентах.
5. **Event Bus или Pub/Sub системы:** Такие системы позволяют компонентам общаться друг с другом через глобальное событийное пространство. Компоненты могут отправлять события и подписываться на них, реагируя на изменения в других частях приложения.

Выбор метода зависит от конкретной задачи, размера приложения и предпочтений разработчика. Важно подходить к этому выбору с учетом архитектуры приложения и будущей масштабируемости.