**Что такое virtual dom ?**

Спросят с вероятностью 27%

Virtual DOM (виртуальный Document Object Model) - это концепция, широко используемая в разработке интерфейсов, особенно в библиотеке React от Facebook, а также в других современных фронтенд-фреймворках. DOM - это структура данных, используемая браузерами для представления веб-страниц. Она позволяет программам изменять структуру, стиль и содержание веб-страницы, взаимодействуя с HTML и CSS. Проблема обычного DOM заключается в том, что он может быть довольно медленным при частых обновлениях, поскольку изменения в нем приводят к перерисовке элементов страницы, что может быть ресурсоёмким процессом.

Эта концепция призвана решить данную проблему. Вместо того чтобы напрямую взаимодействовать с реальным DOM при каждом изменении, изменения сначала применяются к виртуальному, который является лёгкой копией реального DOM. После этого, с помощью процесса, называемого согласованием (reconciliation), виртуальный DOM сравнивается с предыдущей его версией, чтобы определить, какие именно изменения необходимо внести в реальный. Это позволяет минимизировать количество операций с реальным DOM, что значительно увеличивает производительность приложения.

Пример без Virtual DOM:

const element = document.getElementById('myElement');

element.textContent = 'Новый текст';

В этом случае каждое изменение непосредственно обновляет DOM, что может быть неэффективно при множественных обновлениях.

Пример с использованием Virtual DOM (пример на React):

class MyComponent extends React.Component {

render() {

return <div>Новый текст</div>;

}

В этом случае, при изменении состояния компонента, React сначала применяет изменения к виртуальному DOM, а затем, используя согласование, оптимально обновляет реальный.

Виртуальный Document Object Model позволяет писать код, как если бы можно было менять любую часть веб-страницы в любой момент, не беспокоясь о производительности. Это делает разработку интерфейсов более интуитивно понятной и эффективной.

Virtual DOM - это технология для оптимизации обновлений веб-интерфейсов, позволяющая ускорить и упростить разработку сложных пользовательских интерфейсов, минимизируя взаимодействие с медленным реальным DOM.

**Что происходит во фронтенде ?**

Спросят с вероятностью 3%

Во фронтенде происходит взаимодействие пользователя с веб-сайтом или веб-приложением через веб-браузер. Фронтенд-разработка включает в себя создание пользовательского интерфейса и пользовательского опыта (UI/UX), а также реализацию логики взаимодействия на клиентской стороне. Вот основные аспекты того, что происходит во фронтенде:

1️⃣ Рендеринг страницы

Когда пользователь заходит на веб-сайт, его браузер запрашивает HTML, CSS и JavaScript файлы с сервера. Браузер анализирует эти файлы и отображает страницу в соответствии с их указаниями. Этот процесс включает в себя:

- Построение DOM-дерева: Браузер строит DOM (Document Object Model) дерево из HTML-кода.

- Построение CSSOM-дерева: Аналогично, браузер строит CSSOM (CSS Object Model) дерево из CSS-кода.

- Рендеринг: С использованием DOM и CSSOM деревьев браузер рендерит страницу.

2️⃣ Интерактивность

Используется для добавления интерактивности на веб-страницу. Это может включать в себя:

- Обработку событий, таких как клики мыши и нажатия клавиш.

- Изменение DOM в ответ на действия пользователя.

- Асинхронные запросы к серверу с помощью AJAX или Fetch API для динамического обновления контента без перезагрузки страницы.

3️⃣ Адаптивный дизайн

Используют адаптивный дизайн для обеспечения корректного отображения и удобства использования веб-сайтов на различных устройствах, включая мобильные телефоны, планшеты и настольные компьютеры. Это достигается за счет медиа-запросов, которые позволяют применять разные стили в зависимости от размера экрана и других факторов.

4️⃣ Оптимизация производительности

Является важной частью фронтенд-разработки. Это включает в себя:

- Минификацию и объединение файлов CSS и JavaScript.

- Оптимизацию изображений.

- Ленивую загрузку медиа-контента.

- Использование кэширования и сервис-воркеров.

5️⃣ Обеспечение безопасности

Также участвуют в обеспечении безопасности веб-приложений, реализуя меры защиты от XSS (межсайтового скриптинга), CSRF (межсайтовой подделки запроса) атак и других угроз.

6️⃣ Использование фреймворков и библиотек

Для упрощения и ускорения разработки часто используются фреймворки и библиотеки, такие как React, Angular, Vue.js, Svelte и другие. Они предоставляют готовые компоненты и утилиты для разработки интерфейса и логики веб-приложений.

Фронтенд-разработка постоянно развивается, появляются новые инструменты, технологии и подходы, что делает её динамичной и интересной областью в мире веб-разработки.

**Какие есть компоненты ?**

Спросят с вероятностью 7%

Компоненты являются основными строительными блоками для построения пользовательских интерфейсов. Они позволяют разбивать интерфейс на независимые, переиспользуемые части, которые можно разрабатывать, тестировать и поддерживать отдельно друг от друга. Компоненты в React могут быть как простыми и представлять собой один элемент интерфейса (например, кнопку или поле ввода), так и сложными и включать в себя другие компоненты, формируя иерархию.

Типы:

1️⃣ Функциональные компоненты:

- Самый простой способ определить компонент.

- Это JavaScript-функции, которые принимают пропсы (свойства) в качестве аргумента и возвращают элемент, описывающий, что должно отображаться на экране.

- С появлением хуков, функциональные компоненты получили возможность использовать состояние и другие возможности React, аналогичные классовым компонентам.

function Welcome(props) {

return <h1>Привет, {props.name}</h1>;

}

2️⃣ Классовые компоненты:

- Перед введением хуков классовые компоненты были единственным способом использования состояния и жизненного цикла компонентов .

- Они определяются путем расширения React.Component и требуют метода render(), который возвращает элемент React.

class Welcome extends React.Component {

render() {

return <h1>Привет, {this.props.name}</h1>;

}

}

**Пропсы и состояние**

- Пропсы (Props): Коротко для "свойств", пропсы передаются от родительских компонентов к дочерним. Они неизменяемы в пределах компонента, что означает, что дочерний компонент не может изменять пропсы, полученные от родителя.

- Состояние (State): Состояние позволяет компонентам реагировать на ввод пользователя, серверные запросы и любые другие действия, сохраняя и изменяя данные во времени. В отличие от пропсов, состояние управляется и изменяется внутри самого компонента.

**Жизненный цикл компонента**

Классовые компоненты имеют несколько "жизненных циклов" — методов, которые автоматически вызываются в разные моменты жизни компонента, такие как монтирование (компонент вставляется в DOM), обновление (компонент реагирует на изменения пропсов или состояния) и размонтирование (компонент удаляется из DOM). Хуки позволяют использовать аналогичные возможности в функциональных компонентах.

Компоненты обеспечивают эффективный способ организации пользовательского интерфейса, делая код более читабельным, легким для тестирования и поддержки. Они позволяют разработчикам строить сложные приложения из простых частей, облегчая разработку и обеспечивая лучший пользовательский опыт.

**Какие проблемы решает react ?**

Спросят с вероятностью 10%

React — это популярная библиотека для разработки пользовательских интерфейсов. Она предлагает разработчикам эффективный и гибкий способ создавать интерактивные веб-приложения. React решает множество задач и проблем, с которыми разработчики сталкиваются при построении современных веб-приложений:

1️⃣ Создание динамических интерфейсов

Позволяет легко создавать динамические веб-интерфейсы с использованием компонентов. Благодаря декларативному подходу и использованию JSX (синтаксис, похожий на HTML внутри JavaScript), разработка становится более интуитивно понятной и менее подверженной ошибкам.

2️⃣ Управление состоянием компонентов

Предоставляет механизмы для управления состоянием (state) внутри компонентов, что позволяет разработчикам легко отслеживать и управлять изменениями данных в приложении. Это упрощает создание интерактивных интерфейсов, таких как формы, фильтры, навигация и т.д.

3️⃣ Оптимизация производительности

Использует виртуальный DOM (Virtual DOM), что позволяет оптимизировать обновления в реальном DOM. Это снижает затраты производительности при обновлении пользовательского интерфейса, особенно при работе с большими объемами данных и сложными интерфейсами.

4️⃣ Компонентный подход

Поощряет использование компонентного подхода к разработке, что позволяет разбивать интерфейс на мелкие, повторно используемые части. Это улучшает организацию кода, облегчает его поддержку и тестирование.

5️⃣ Универсальность и изоморфизм

Код может выполняться как на клиенте, так и на сервере, что позволяет создавать изоморфные (универсальные) приложения. Это улучшает SEO, скорость загрузки и обеспечивает лучший пользовательский опыт.

6️⃣ Мощный экосистема и сообщество

Обладает большим и активным сообществом разработчиков, а также мощной экосистемой инструментов и библиотек, таких как Redux для управления состоянием приложения, React Router для навигации, Next.js для серверного рендеринга и многих других.

7️⃣ Гибкость и интеграция

Может быть интегрирован в существующие приложения постепенно, что делает его удобным выбором для поэтапной модернизации проектов. Также он хорошо сочетается с другими библиотеками и фреймворками.

React предлагает разработчикам удобные инструменты для создания эффективных, масштабируемых и поддерживаемых веб-приложений, решая множество проблем современной веб-разработки, начиная от управления состоянием и заканчивая производительностью интерфейсов.

**Зачем нужен менеджер состояния ?**

Спросят с вероятностью 10%

Менеджер состояния используется для централизованного управления состоянием приложения, особенно когда речь идет о сложных или крупномасштабных приложениях с множеством компонентов, которым необходимо обмениваться данными. Состояние приложения может включать различные данные: от пользовательского ввода и серверных ответов до временных данных, таких как статус загрузки или авторизационный статус пользователя.

**Зачем нужен менеджер состояния:**

1️⃣ Централизация состояния: Позволяют хранить все данные приложения в одном месте, делая их легкодоступными для всех компонентов приложения независимо от их места в иерархии.

2️⃣ Упрощение обмена данными между компонентами: В больших приложениях компоненты часто нуждаются в доступе к одним и тем же данным. Облегчает передачу данных между компонентами, уменьшая необходимость в пропсах или колбэках.

3️⃣ Поддержка сложных взаимодействий и асинхронности: Предоставляют механизмы для управления асинхронными действиями, такими как запросы к API, и могут обрабатывать сложные последовательности действий, обновляя состояние соответствующим образом.

4️⃣ Отладка и разработка: Большинство предлагают инструменты для отладки, которые позволяют просматривать изменения состояния в реальном времени, отслеживать действия, вызывающие эти изменения, и даже перемещаться по истории изменений состояния.

5️⃣ Повышение производительности: Централизованное управление состоянием позволяет оптимизировать обновления интерфейса, минимизируя ненужные рендеринги и улучшая общую производительность приложения.

6️⃣ Повышение масштабируемости приложения: Предоставляет структурированный подход к управлению состоянием, который облегчает масштабирование приложения и добавление новых функций.

Примеры:

- Redux: Один из самых популярных менеджеров состояния для React, использующий концепцию однонаправленного потока данных.

- Vuex: Специализированный менеджер состояния для Vue.js.

- MobX: Альтернатива Redux, предлагающая более гибкий подход к управлению состоянием через наблюдаемые объекты.

- Context API и useReducer (React Hook): Встроенные механизмы в React для управления состоянием без добавления внешних библиотек.

В зависимости от сложности приложения и предпочтений можно выбирать между использованием внешних библиотек для управления состоянием или применением встроенных средств фреймворка.

**Какие бывают хуки ?**

Спросят с вероятностью 17%

Хуки позволяют использовать состояние и другие возможности React без написания классов. С их появлением в версии 16.8, разработчики получили доступ к мощному и выразительному инструменту для создания компонентов. Рассмотрим основные примеры, а также некоторые дополнительные, которые предоставляет React.

Основные хуки:

1️⃣ useState

Позволяет добавлять состояние к функциональным компонентам.

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

2️⃣ useEffect

Даёт возможность выполнять побочные эффекты в компоненте, такие как обращения к API, подписки или таймеры.

useEffect(() => {

document.title = Вы нажали ${count} раз;

}, [count]); // Зависимость от состояния count

3️⃣ useContext

Даёт возможность получить текущее значение контекста, что упрощает доступ к данным в глубоко вложенных компонентах без их явной передачи через пропсы.

const value = useContext(MyContext);

**Дополнительные хуки:**

1️⃣ useReducer

Альтернатива useState для более сложного управления состоянием, использует концепцию редьюсера для обработки сложных изменений состояния.

const [state, dispatch] = useReducer(reducer, initialState);

2️⃣ useCallback

Возвращает мемоизированный колбэк, который изменяется только если изменяются зависимости.

const memoizedCallback = useCallback(() => {

doSomething(a, b);

}, [a, b]);

3️⃣ useMemo

Возвращает мемоизированное значение, пересчитывается только при изменении зависимостей.

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

4️⃣ useRef

Возвращает изменяемый ref-объект, который можно использовать для доступа к DOM-элементам или хранения любого мутируемого значения.

const myRef = useRef(initialValue);

Основные хуки (useState, useEffect, useContext) позволяют добавлять состояние, работать с побочными эффектами и использовать контекст в функциональных компонентах

Дополнительные хуки предоставляют более специализированные или углублённые возможности для оптимизации, работы с рефами, мемоизации и других задач, связанных с жизненным циклом компонентов и управлением рендерингом.

**Что такое меморизация ?**

Спросят с вероятностью 17%

Меморизация — это оптимизационная техника, используемая для ускорения выполнения программ путём сохранения результатов дорогостоящих функций и повторного использования этих результатов, когда снова возникают одинаковые входные данные, вместо того чтобы повторно выполнять функцию.

**Как это работает**

При первом вызове функции с определённым набором аргументов результат выполнения функции сохраняется в каком-либо хранилище (чаще всего в виде объекта или карты). При последующих вызовах функции с теми же аргументами, вместо выполнения функции, возвращается сохранённый результат. Это значительно сокращает время выполнения, особенно если исходная функция требует значительных вычислительных ресурсов.

**Применение:**

Меморизация часто применяется для оптимизации производительности в различных областях, включая:

- Рекурсивные вычисления, например, вычисление чисел Фибоначчи.

- Дорогостоящие вычисления, которые часто повторяются с одними и теми же аргументами.

- Оптимизация производительности веб-приложений, например, в React, где меморизация помогает избежать ненужных перерисовок компонентов.

**Меморизация** — это техника сохранения результатов выполнения функций для их быстрого повторного использования, что позволяет существенно ускорить выполнение программы, особенно при работе с дорогостоящими вычислениями или рекурсивными функциями. Это достигается за счёт того, что вместо повторного выполнения функции с теми же аргументами, результат возвращается из специального хранилища (кэша).

**Как использовать useeffect ?**

Спросят с вероятностью 13%

useEffect — это хук, введённый в React 16.8, который позволяет выполнять побочные эффекты в функциональных компонентах. Побочные эффекты включают в себя операции, которые выходят за рамки возвращаемого результатом рендеринга, такие как работа с сетью (запросы к API), подписки, изменение документа и так далее. Он заменяет собой жизненные циклы классовых компонентов, такие как componentDidMount, componentDidUpdate и componentWillUnmount.

Синтаксис:

useEffect(() => {

// Ваш код для побочного эффекта

return () => {

// Очистка (необязательно)

};

}, [зависимости]);

✅ Функция побочного эффекта: Это его основное тело, где вы помещаете код, который должен выполняться после каждого рендеринга (по умолчанию) или когда изменяются определённые зависимости.

✅ Очистка: Функция очистки выполняется перед тем, как компонент будет удалён, а также перед следующим выполнением эффекта. Это идеальное место для отмены подписок, таймеров и других ресурсов, чтобы избежать утечек памяти.

✅ Массив зависимостей: Определяет, при изменении каких значений должен повторно выполняться эффект. Если массив пуст ([]), эффект выполнится один раз после первого рендеринга. Если массив не указан, эффект будет выполняться после каждого рендеринга.

**Примеры использования:**

1️⃣ Запуск эффекта один раз

Чтобы выполнить эффект один раз после первого рендеринга (аналог componentDidMount в классовых компонентах), используйте пустой массив зависимостей:

useEffect(() => {

// Код здесь будет выполнен один раз после инициализации компонента

}, []);

2️⃣ Выполнение при изменении зависимостей

Чтобы выполнить эффект при изменении определённых значений:

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

useEffect(() => {

// Код здесь будет выполнен каждый раз, когда изменяется значение count

document.title = Вы нажали ${count} раз;

}, [count]); // Зависимости

3️⃣ Очистка эффекта

Пример использования функции очистки:

useEffect(() => {

const timerID = setInterval(() => {

// Ваша логика здесь

}, 1000);

return () => {

clearInterval(timerID); // Очистка при размонтировании компонента

};

}, []); // Эффект выполнится один раз

useEffect позволяет вам выполнять побочные эффекты в функциональных компонентах. Он может заменить собой несколько методов жизненного цикла классовых компонентов, предоставляя более унифицированный и удобный способ работы с побочными эффектами. Через массив зависимостей вы контролируете, как часто эффект должен выполняться, что делает ваш компонент более оптимизированным и предсказуемым.

**Что такое state manager и с какими приходилось работать ?**

Спросят с вероятностью 13%

State manager (менеджер состояний) — это инструмент или библиотека, который помогает управлять состоянием приложения. Состояние приложения может включать различные данные: от пользовательского ввода и временных данных до кэшированных страниц и аутентифицированного пользователя. Менеджер состояний помогает централизованно управлять этими данными, облегчая их передачу между компонентами, упрощая тестирование и повышая производительность приложения.

**Зачем нужен менеджер состояний?**

✅ Упрощение доступа к данным: Предоставляют унифицированный интерфейс для доступа к данным из любого места в приложении.

✅ Повышение производительности: Эффективное управление состоянием может помочь избежать ненужных обновлений интерфейса и повысить производительность приложения.

✅ Улучшение масштабируемости: Упрощает масштабирование приложения и добавление новых функций.

✅ Упрощение отладки и тестирования: Делает приложение более предсказуемым, что упрощает отладку и тестирование.

Примеры:

1️⃣ Redux: Одна из самых популярных библиотек для управления состоянием в React-приложениях. Она предлагает однонаправленный поток данных, что делает изменения состояния предсказуемыми.

2️⃣ Vuex: Библиотека для управления состоянием в Vue.js. Подобно Redux, она предлагает централизованное хранилище для всех компонентов приложения.

3️⃣ MobX: Альтернатива Redux, которая использует наблюдаемые объекты для управления состоянием. Она предлагает более гибкий подход к управлению состоянием, автоматически отслеживая изменения.

4️⃣ Context API и Hooks в React: Начиная с версии 16.8, React предоставляет встроенный механизм для управления состоянием без необходимости добавления внешних библиотек. Хотя для небольших или средних проектов это может быть достаточно, в больших приложениях может потребоваться более мощное решение вроде Redux.

**Менеджер состояний** — это инструмент, который помогает управлять состоянием веб-приложения, обеспечивая эффективную передачу данных между компонентами, улучшая производительность и упрощая отладку и тестирование. В зависимости от технологического стека и требований проекта, разработчики могут выбирать из различных решений, таких как Redux, Vuex, MobX или использовать встроенные средства фреймворков, например, Context API.

**Что такое JSX ?**

Спросят с вероятностью 10%

JSX (JavaScript XML) — это расширение синтаксиса. Оно позволяет писать структуру компонентов React с использованием синтаксиса, похожего на HTML, непосредственно файлах. Этот синтаксический сахар делает код более читабельным и удобным для разработки пользовательских интерфейсов.

Основные особенности JSX:

- Сочетание HTML и JavaScript: Позволяет использовать HTML-теги и компоненты React внутри JavaScript-кода.

- Выражения JavaScript: Можно вставлять JavaScript-выражения в фигурные скобки {}. Это позволяет динамически управлять содержимым компонентов.

- Компонентный подход: Легко создавать и использовать повторно компоненты UI, что упрощает разработку сложных пользовательских интерфейсов.

- Преобразование: Не может быть выполнен напрямую браузерами и требует преобразования в обычный JavaScript. Это обычно делается с помощью транспиляторов, таких как Babel.

Преимущества использования JSX:

✅ Наглядность: Код с ним легче читать и писать, поскольку он напоминает HTML.

✅ Мощность: Сочетает в себе преимущества JavaScript, позволяя использовать всю его выразительную мощь при описании UI.

✅ Интеграция: Плотно интегрирован с React, облегчая создание интерактивных компонентов.

JSX — это расширение синтаксиса, которое делает разработку интерфейсов на React более интуитивной и выразительной за счёт смешивания HTML-подобного кода с JavaScript. Хотя использование JSX необязательно для работы с React, оно значительно упрощает процесс разработки и повышает читабельность кода.

**Что такое react ?**

Спросят с вероятностью 10%

React — это декларативная, эффективная и гибкая библиотека для создания пользовательских интерфейсов. Она позволяет строить сложные интерактивные UI из небольших и изолированных частей кода, называемых "компонентами". Он был создан инженерами Facebook и впервые представлен в 2013 году.

**Основные особенности React:**

1️⃣ Компонентный подход: Приложения строятся из компонентов, которые содержат свой собственный стейт (состояние) и логику отображения. Компоненты могут быть повторно использованы в разных частях приложения, что повышает удобство поддержки и разработки кода.

2️⃣ Декларативность: Позволяет описывать, как компоненты интерфейса выглядят в разных состояниях, автоматически обновляя их при изменении данных. Это делает код более читаемым и упрощает процесс разработки.

3️⃣ Виртуальный DOM (VDOM): Использует концепцию виртуального DOM для повышения производительности приложения. Виртуальный DOM — это легковесная копия реального DOM-дерева в памяти, позволяющая оптимизировать обновления интерфейса за счет минимизации взаимодействия с реальным DOM.

4️⃣ Однонаправленный поток данных: В нем данные передаются от родительских компонентов к дочерним через свойства (props), что обеспечивает предсказуемость и упрощает отладку приложений.

**Использование React**

Широко используется для разработки как небольших, так и крупных веб-приложений, предоставляя разработчикам мощные инструменты для создания интерактивных пользовательских интерфейсов. Он может быть использован как в одностраничных приложениях (SPA), так и в многостраничных сайтах.

**Какие библиотеки популярны в react ?**

Спросят с вероятностью 3%

В экосистеме React существует множество популярных библиотек, которые используются для улучшения разработки, повышения производительности и добавления дополнительного функционала в ваши приложения. Рассмотрим некоторые из них:

1️⃣ React Router

Это стандартная библиотека для маршрутизации в React-приложениях. Она позволяет вам внедрять динамическую навигацию в приложение, без перезагрузки страницы. React Router поддерживает как веб, так и мобильные приложения.

2️⃣ Redux

Это библиотека для управления состоянием приложения. Она чаще всего используется в сложных приложениях с большим объемом данных и множеством действий, которые могут изменять это состояние. Помогает упростить управление состоянием, делая его предсказуемым.

3️⃣ Axios

Это библиотека для выполнения HTTP-запросов. Она может использоваться в браузере и на сервере (Node.js). Предлагает широкий набор функций и удобный API для выполнения запросов к серверу и обработки ответов.

4️⃣ Material-UI

Содержит готовые компоненты, стилизованные согласно Material Design от Google. Это позволяет легко и быстро создавать красивые и функциональные пользовательские интерфейсы.

5️⃣ Styled Components

Используется для стилизации компонентов в приложениях React. Это библиотека, позволяющая писать CSS-код непосредственно в файлах, используя тегированные шаблонные строки. Это помогает создавать стилизованные компоненты, которые легко переиспользовать и поддерживать.

6️⃣ React Testing Library

Предназначена для тестирования компонентов. Она позволяет тестировать компоненты "как они есть", симулируя взаимодействие пользователя с интерфейсом, что делает тесты более надежными и удобными в поддержке.

7️⃣ React Query

Предлагает удобные инструменты для выполнения, кэширования и синхронизации асинхронных запросов в React-приложениях. Она автоматически обрабатывает обновление данных, кэширование и инвалидацию, значительно упрощая работу с асинхронными данными.

**Какие минусы у react ?**

Спросят с вероятностью 3%

React — это популярная библиотека для построения пользовательских интерфейсов, созданная и поддерживаемая Facebook. Хотя React обладает многими преимуществами, такими как компонентный подход, виртуальный DOM и реактивное обновление, он также имеет некоторые недостатки. Рассмотрим основные минусы React:

1️⃣Высокий порог вхождения для новичков

✅Сложность экосистемы: React сам по себе может быть относительно простым для понимания, но экосистема вокруг него (Redux, React Router, JSX, Webpack и т.д.) может быть сложной для новичков. Разработчикам нужно освоить много дополнительных инструментов и библиотек, чтобы использовать React эффективно.

✅JSX: Хотя JSX делает код более декларативным, он может показаться непривычным и сложным для разработчиков, не знакомых с ним.

2️⃣Быстро меняющаяся экосистема

✅Частые обновления: React и его экосистема быстро развиваются, и это может затруднить поддержание кода в актуальном состоянии. Разработчикам часто приходится адаптировать свои проекты к новым версиям и изменениям.

✅Несовместимости: Бывают случаи, когда обновления ломают обратную совместимость, что требует дополнительного времени и усилий на обновление проекта.

3️⃣Настройка проекта

✅Boilerplate код: Создание нового проекта на React может потребовать значительного количества начального кода и конфигурации. Хотя инструменты, такие как Create React App, упрощают этот процесс, для более сложных проектов часто требуется ручная настройка Webpack, Babel и других инструментов.

✅Сложность сборки: Настройка сборочного процесса для оптимизации производительности и поддержки различных окружений может быть сложной задачей.

4️⃣Производительность

✅Проблемы с производительностью при большом количестве компонентов: Несмотря на оптимизации, такие как виртуальный DOM, большие и сложные React-приложения могут столкнуться с проблемами производительности. Неправильное управление состоянием или неоптимизированные компоненты могут привести к избыточным перерисовкам и снижению производительности.

✅Рендеринг больших списков: Работа с большими списками данных может быть медленной, и разработчикам приходится использовать такие техники, как виртуализация (например, react-window или react-virtualized) для улучшения производительности.

5️⃣SEO (Search Engine Optimization)

✅Проблемы с индексированием: Хотя React позволяет создавать динамичные и интерактивные приложения, серверный рендеринг (Server-Side Rendering, SSR) или статическая генерация (Static Site Generation, SSG) могут потребоваться для улучшения SEO. Это добавляет сложность в настройку и разработку.

✅Дополнительные инструменты: Для улучшения SEO могут потребоваться дополнительные инструменты и библиотеки, такие как Next.js.

Хотя React предлагает множество преимуществ для разработки пользовательских интерфейсов, он также имеет свои минусы. К ним относятся высокий порог вхождения для новичков, быстро меняющаяся экосистема, сложности с настройкой проектов, проблемы с производительностью при большом количестве компонентов, сложности с SEO, отсутствие строгих правил и рекомендаций по структуре проекта и увеличение размера бандла из-за множества зависимостей. Эти недостатки могут затруднить работу с React.

**Зачем создавались библиотеки react и другие ?**

Спросят с вероятностью 3%

Библиотеки и фреймворки, такие как React, Angular и Vue.js, были созданы для решения проблем и улучшения процессов разработки веб-приложений. Веб-разработка эволюционировала, и требования к производительности, масштабируемости, поддержке и интерактивности веб-приложений значительно возросли. Эти инструменты помогают разработчикам создавать более сложные, быстрые и поддерживаемые приложения. Рассмотрим основные причины создания таких библиотек и фреймворков:

1️⃣Управление состоянием и реактивность

React

Был создан для решения проблем управления состоянием и реактивности в пользовательских интерфейсах. Он предоставляет компонентный подход к построению UI и использует виртуальный DOM для оптимизации обновлений пользовательского интерфейса.

✅Компонентный подход: Разделение пользовательского интерфейса на переиспользуемые компоненты улучшает структуру кода и его поддерживаемость.

✅Виртуальный DOM: Минимизирует прямые обновления реального DOM, используя виртуальный DOM для вычисления необходимых изменений и оптимизации производительности.

**Как работает react ?**

Спросят с вероятностью 7%

React — это библиотека для создания пользовательских интерфейсов. Она позволяет строить сложные, интерактивные UI из маленьких, изолированных частей кода, называемых компонентами. Он основывается на нескольких ключевых концепциях, которые обеспечивают его эффективность и гибкость.

1️⃣ Компонентный подход: Приложения состоят из компонентов. Компонент — это кусочек пользовательского интерфейса, который можно использовать повторно. Каждый компонент имеет своё состояние и пропсы (параметры, которые передаются от родительского компонента), что делает его независимым и способным к самостоятельной работе.

2️⃣ JSX: Использует JSX — синтаксис, похожий на HTML, для описания структуры интерфейса. Он делает код компонентов более читабельным и упрощает процесс их создания.

function App() {

return <div>Привет, мир!</div>;

}

3️⃣ Виртуальный DOM: Одной из главных особенностей является использование виртуального DOM — лёгкой копии реального DOM. Это позволяет React оптимизировать обновления интерфейса, сначала применяя изменения к виртуальному DOM, а затем, после вычисления наиболее эффективных изменений, к реальному DOM. Это существенно повышает производительность, особенно при работе с большими объёмами данных.

4️⃣ Однонаправленный поток данных: Данные течут строго от родителей к детям через пропсы, что упрощает отладку и понимание приложений, так как знаешь, откуда пришли данные и как они изменяются.

5️⃣ Хуки: С появлением хуков он стал ещё более мощным и гибким. Хуки позволяют использовать состояние и другие его возможности без написания классов. Например, хук useState позволяет добавить состояние к функциональному компоненту.

import { useState } from 'react';

function Counter() {

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

return (

<div>

<p>Вы кликнули {count} раз</p>

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

Нажми на меня

</button>

</div>

);

}

React предлагает мощный набор инструментов для быстрой и эффективной разработки интерактивных пользовательских интерфейсов. Он позволяет разрабатывать компоненты независимо, управлять их состоянием и пропсами, а также эффективно обновлять интерфейс, минимизируя взаимодействие с реальным DOM.

React помогает создавать веб-страницы, которые быстро реагируют на действия пользователя, делая взаимодействие с сайтом плавным и приятным, не перезагружая страницу при каждом действии.

**Зачем нужен виртуальный дом ?**

Спросят с вероятностью 7%

Виртуальный DOM (Virtual DOM) — это концепция, используемая в современных фреймворках и библиотеках, таких как React, для оптимизации и ускорения работы с реальным DOM. Реальный DOM медленный при частых изменениях, поэтому виртуальный DOM решает эту проблему, делая процесс более эффективным.

**Зачем он нужен**

1️⃣Улучшение производительности: Значительно уменьшает количество прямых манипуляций с реальным DOM, которые являются дорогостоящими с точки зрения производительности.

2️⃣Упрощение разработки: Работая с ним, можно более эффективно управлять состоянием приложения и его представлением, абстрагируясь от непосредственной работы с реальным DOM.

**Как он работает**

1️⃣Создание: Когда состояние приложения меняется, создаётся новый виртуальный DOM, представляющий обновлённое состояние.

2️⃣Сравнение: Сравниваются старый и новый виртуальные DOM, чтобы определить, какие части реального DOM нужно обновить. Этот процесс называется "диффинг" (diffing).

3️⃣Минимальные: После сравнения, в реальный DOM вносятся только необходимые изменения, что значительно снижает количество операций с ним.

**Преимущества**

1️⃣Повышение производительности: Благодаря уменьшению количества прямых операций с реальным DOM.

2️⃣Кроссбраузерная совместимость: Виртуальный DOM позволяет абстрагироваться от специфичных для браузера особенностей работы с DOM.

3️⃣Лёгкость обновлений и рендеринга: Использование виртуального DOM делает процесс обновления интерфейса приложения более предсказуемым и управляемым.

Виртуальный DOM — это оптимизация работы с реальным DOM, обеспечивающая более быструю и эффективную работу приложений. Он позволяет минимизировать количество изменений, вносимых в реальный DOM, что существенно повышает производительность и упрощает разработку.

**Назови компоненты жизненного цикл ?**

Спросят с вероятностью 7%

Жизненный цикл компонента состоит из нескольких этапов, которые можно разделить на три основные фазы: монтирование, обновление и размонтирование. Каждый из этих этапов включает в себя определённые методы цикла.

1️⃣Монтирование (Mounting):

✅constructor(): Вызывается перед монтированием компонента. Обычно используется для инициализации состояния и привязки методов.

✅staticgetDerivedStateFromProps(props, state): Вызывается перед рендером компонента как при монтировании, так и при обновлении. Используется для синхронизации состояния с пропсами.

✅render(): Обязательный метод, который возвращает JSX-разметку компонента.

✅componentDidMount(): Вызывается сразу после монтирования компонента. Здесь можно выполнять запросы к API или подписываться на события.

2️⃣Обновление (Updating):

✅staticgetDerivedStateFromProps(props, state): Как и при монтировании, вызывается перед рендером компонента для синхронизации состояния с пропсами.

✅shouldComponentUpdate(nextProps, nextState): Определяет, нужно ли компоненту перерисовываться. Можно использовать для оптимизации производительности.

✅render(): Ререндер компонента на основе новых пропсов или состояния.

✅getSnapshotBeforeUpdate(prevProps, prevState): Вызывается прямо перед тем, как изменённый DOM будет отображен. Полезно для сохранения информации о DOM, которая может измениться после обновления.

✅componentDidUpdate(prevProps, prevState, snapshot): Вызывается сразу после обновления компонента. Можно использовать для выполнения действий после обновления, таких как запросы к API на основе новых пропсов или состояния.

3️⃣Размонтирование (Unmounting):

✅componentWillUnmount(): Вызывается перед размонтированием и удалением компонента. Используется для очистки ресурсов, таких как таймеры, подписки или запросы.

**Что такое ssr ?**

Спросят с вероятностью 7%

SSR (Server-Side Rendering) — это технология рендеринга веб-страниц на сервере, а не на клиенте. В контексте современных JavaScript-фреймворков, таких как React или Vue.js, это означает, что HTML-код страницы генерируется на сервере и отправляется в браузер уже готовым для отображения. Это отличается от традиционного подхода клиентского рендеринга (Client-Side Rendering, CSR), где HTML генерируется непосредственно в браузере с использованием JavaScript.

Основные особенности и преимущества

1️⃣Улучшение производительности и скорости загрузки:

✅Первичный контент: Поскольку HTML генерируется на сервере, браузеры могут отображать готовый контент без необходимости ждать загрузки и выполнения JavaScript.

✅SEO: Поисковые системы лучше индексируют страницы, так как весь контент доступен сразу.

2️⃣Лучшая поддержка SEO:

✅Статические HTML-страницы, генерируемые сервером, легче индексируются поисковыми системами, что улучшает видимость сайта в поисковых результатах.

3️⃣Улучшенное восприятие пользователем (First Paint):

✅Пользователи видят контент быстрее, так как сервер отправляет уже готовую HTML-разметку.

**Как он работает**

1️⃣Запрос от клиента:

✅Браузер отправляет запрос на сервер для получения страницы.

2️⃣Генерация HTML на сервере:

✅Сервер запускает JavaScript-код для рендеринга компонента или всей страницы.

✅Сервер генерирует HTML-код и отправляет его клиенту.

3️⃣Отображение страницы в браузере:

✅Браузер получает готовую HTML-страницу и отображает её.

✅Затем JavaScript-фреймворк (например, React или Vue) «гидрирует» страницу, то есть связывает готовую HTML-разметку с JavaScript-кодом, чтобы сделать страницу интерактивной.

Next.js — популярный фреймворк для React, который поддерживает SSR из коробки.

**Как часто происходит рендер?**

💬 Спрашивают в 3%

В React рендеринг компонента происходит в нескольких ситуациях. Давайте рассмотрим основные из них:

1️⃣ Первоначальный рендеринг:

➕ Когда компонент впервые монтируется в DOM.

2️⃣ Обновление состояния (state):

➕ Если состояние компонента изменяется с помощью this.setState в классовых компонентах или useState в функциональных компонентах.

3️⃣ Обновление пропсов (props):

➕ Когда компонент получает новые пропсы от родительского компонента.

4️⃣ Обновление контекста (context):

➕ Когда значение контекста, используемого компонентом, изменяется.

5️⃣ Обновление при изменении ключа (key):

➕ Если ключ компонента изменяется (например, в списках), React будет рассматривать это как удаление старого компонента и добавление нового.

**Как отрисовать компонент по условию?**

💬 Спрашивают в 6%

В React отрисовка компонентов по условию осуществляется с помощью условных операторов JavaScript внутри JSX. Это позволяет отображать компоненты или элементы только тогда, когда выполняется определенное условие. Рассмотрим несколько способов условной отрисовки компонентов в React.

1️⃣ Использование тернарного оператора

Тернарный оператор ? : позволяет компактно проверять условие и отрисовывать один из двух компонентов.

<div>

{isLoggedIn ? <p>Welcome back!</p> : <p>Please log in.</p>}

</div>

2️⃣ Логическое И (&&)

Можно использовать логический оператор && для отрисовки компонента только тогда, когда условие истинно.

<div>

<h1>Item</h1>

{showDetails && <p>Here are the details...</p>}

</div>

3️⃣ Условный рендеринг с if...else

Для более сложной логики условного рендеринга можно использовать обычные операторы if...else внутри функции компонента.

const MyComponent = ({ status }) => {

if (status === 'loading') {

return <p>Loading...</p>;

} else if (status === 'error') {

return <p>Error occurred!</p>;

} else if (status === 'success') {

return <p>Data loaded successfully!</p>;

} else {

return null; // Возвращаем null, чтобы не отрисовывать ничего

}

};

4️⃣ Условный рендеринг с помощью функций

Можно вынести условную логику рендеринга в отдельную функцию для лучшей читаемости и повторного использования.

const renderContent = (status) => {

switch (status) {

case 'loading':

return <p>Loading...</p>;

case 'error':

return <p>Error occurred!</p>;

case 'success':

return <p>Data loaded successfully!</p>;

default:

return null;

}

**Что не стоит писать в рендере?**

💬 Спрашивают в 3%

В методе render (или в теле функционального компонента) в React не следует писать код, который выполняет побочные эффекты или вызывает ресурсоёмкие операции. Метод render должен быть чистым и идемпотентным, что означает, что его выполнение не должно изменять состояние компонента или вызывать другие побочные эффекты. Вот несколько примеров того, чего следует избегать в методе render:

1️⃣ Выполнение побочных эффектов

Побочные эффекты включают операции, которые изменяют состояние, делают запросы к серверу, подписываются на события или манипулируют DOM напрямую. Для этих целей следует использовать хуки, такие как useEffect, или методы жизненного цикла классовых компонентов, такие как componentDidMount и componentDidUpdate.

2️⃣ Изменение состояния (setState)

Изменение состояния компонента внутри метода render приведет к бесконечному циклу рендеринга.

3️⃣ Выполнение дорогостоящих вычислений

Метод render должен быть быстрым и легким, чтобы поддерживать высокую производительность. Дорогостоящие вычисления могут замедлить рендеринг.

4️⃣ Прямое манипулирование DOM

React управляет DOM за вас, поэтому вам не нужно и не следует манипулировать DOM напрямую в методе render. Вместо этого используйте состояние и пропсы для управления рендерингом.

🤔 Что не делать:

class MyComponent extends React.Component {

render() {

// Не манипулируйте DOM напрямую в render

document.getElementById('some-id').textContent = 'New Text';

return <div>Rendering...</div>;

}

}

🤔 Правильный подход:

class MyComponent extends React.Component {

render() {

return <div id="some-id">New Text</div>;

}

}

**Зачем нужен ref?**

💬 Спрашивают в 3%

В React ref (сокращение от reference) используется для доступа к DOM-элементам или компонентам напрямую. Он позволяет взаимодействовать с элементами, которые были созданы в процессе рендеринга, предоставляя механизм для манипуляции с ними, получения их размеров, положения или вызова методов у компонент. Это особенно полезно в ситуациях, когда необходимо выполнить операции, которые не могут быть выполнены исключительно через декларативный подход React.

🤔 Основные случаи использования `ref`:

➕ Доступ к DOM-элементам:

➕ Использование в сторонних библиотеках:

➕ Сохранение состояния вне дерева компонентов:

**В чем отличие между серверным рендерингом и спа?**

💬 Спрашивают в 3%

Серверный рендеринг (Server-Side Rendering, SSR) и одностраничные приложения (Single Page Applications, SPA) — это два различных подхода к созданию веб-приложений, каждый из которых имеет свои преимущества и недостатки. Рассмотрим их основные отличия.

🤔 Серверный рендеринг (SSR)

➕ При нем весь HTML-код страницы генерируется на сервере и отправляется в браузер в готовом виде.

➕ Сервер получает запрос от клиента, обрабатывает его, собирает необходимые данные, рендерит HTML и отправляет его обратно клиенту.

🤔 Преимущества:

1️⃣ Улучшение SEO:

➕ Поисковые системы получают полностью отрендеренную HTML-страницу, что облегчает индексацию контента.

2️⃣ Быстрый первый рендер (Time to First Paint):

➕ Пользователь видит содержимое страницы быстрее, так как браузер получает готовый HTML.

3️⃣ Совместимость с более старыми браузерами:

➕ Серверный рендеринг не требует от клиента выполнения сложного JavaScript-кода.

🤔 Недостатки:

1️⃣ Нагрузка на сервер:

➕ Сервер должен обрабатывать каждый запрос и рендерить HTML, что может увеличить нагрузку на сервер при большом количестве запросов.

2️⃣ Медленное взаимодействие после первого рендера:

➕ Каждое взаимодействие с приложением может потребовать полного обновления страницы, что делает взаимодействие медленнее и менее плавным.

🤔 Пример:

➕ Next.js для React.

🤔 Одностраничные приложения (SPA)

➕ В нем вся необходимая логика и компоненты загружаются на клиенте единожды при первой загрузке. После этого приложение взаимодействует с сервером через AJAX-запросы, загружая только необходимые данные.

➕ Обновляют контент динамически без перезагрузки всей страницы.

🤔 Преимущества:

1️⃣ Быстрое взаимодействие после первой загрузки:

➕ После начальной загрузки страницы взаимодействие с приложением становится очень быстрым, так как данные обновляются динамически.

2️⃣ Лучшая пользовательская опыта (UX):

➕ Плавные переходы и обновления контента без перезагрузки страницы.

3️⃣ Реактивное взаимодействие:

➕ Возможность реализации сложных пользовательских интерфейсов с высоким уровнем интерактивности.

🤔 Недостатки:

1️⃣ SEO-проблемы:

➕ Поисковые системы могут иметь сложности с индексацией контента, который загружается динамически.

2️⃣ Долгое время первой загрузки (Initial Load Time):

➕ Первоначальная загрузка может быть медленной, так как необходимо загрузить весь JavaScript-код приложения.

3️⃣ Совместимость с браузерами:

➕ Требует современных браузеров и может не работать в старых или менее популярных браузерах без полифиллов.

🤔 Пример:

➕ React с использованием React Router для маршрутизации.

➕ Vue.js с использованием Vue Router.

➕ Angular.

🤔 Гибридные подходы

Некоторые современные фреймворки поддерживают гибридный подход, совмещая преимущества SSR и SPA. Например, Next.js для React позволяет рендерить начальную загрузку на сервере для улучшения SEO и времени до первого рендера, а затем переключаться на SPA для интерактивных взаимодействий.

**Что такое компоненты высшего порядка?**

Это шаблон в React, который позволяет повторно использовать логику компонентов. HOC — это функция, которая принимает компонент и возвращает новый компонент с добавленной функциональностью.

🚩Зачем нужны HOC

🟠Повторное использование логики

HOC позволяет разделять и повторно использовать логику между разными компонентами, не дублируя код.

🟠Абстракция и композиция

HOC помогает создавать более абстрактные и композиционные компоненты, что улучшает архитектуру приложения.

🟠Разделение ответственности

HOC позволяет разделять обязанности между различными компонентами, улучшая структуру и читаемость кода.

🟠Как создается HOC

HOC создается как функция, которая принимает компонент и возвращает новый компонент с добавленной функциональностью.

🚩Плюсы

➕Работа с данными

HOC может использоваться для оборачивания компонента логикой загрузки данных из API или управления состоянием данных.

➕Обработка прав доступа

HOC может использоваться для управления доступом к компонентам, проверяя права пользователя и условно рендеря компоненты.

➕Общие методы

HOC может добавлять общие методы или состояние, которое нужно разделить между несколькими компонентами.

🚩Пример

В этом примере withAuthorization проверяет роль пользователя и условно рендерит WrappedComponent, если пользователь имеет соответствующую роль, или показывает сообщение о запрете доступа.

import React from 'react';

**В чём отличие между контролируемыми и неконтролируемыми объектами?**

В React различают два типа компонентов для обработки пользовательского ввода: контролируемые (controlled) и неконтролируемые (uncontrolled) компоненты. Понимание их различий важно для выбора подходящего способа управления состоянием формы.

**🚩Контролируемые компоненты**

Контролируемый компонент управляет своим состоянием через React. Это означает, что состояние элемента формы (например, значение текстового поля) хранится в состоянии (state) компонента и обновляется через React.

🚩Особенности контролируемых компонентов

🟠Состояние в React

Значение элемента формы хранится в состоянии компонента.

🟠Обработка событий

Все изменения отслеживаются через события (например, onChange) и обновляют состояние компонента.

🟠Односторонний поток данных

Данные идут от состояния компонента к элементу формы, обеспечивая контроль над значением.

**🚩Неконтролируемые компоненты**

Неконтролируемый компонент управляет своим состоянием через DOM. Значение элемента формы извлекается непосредственно из DOM, а не из состояния компонента.

import React, { useRef } from 'react';

**🚩Особенности неконтролируемых компонентов**

🟠Состояние в DOM

Значение элемента формы хранится в самом DOM-элементе.

🟠Использование рефов

Для доступа к значению элемента используется реф (например, через useRef).

🟠Двусторонний поток данных

Данные могут быть извлечены из DOM в любой момент времени, что делает управление состоянием менее очевидным.

**Как отследить демонтирование функционального компонента?**

В функциональных компонентах React можно отслеживать демонтирование компонента с помощью хука useEffect. Когда компонент демонтируется, React вызывает функцию очистки, которую можно определить внутри useEffect.

Хук useEffect позволяет выполнять побочные эффекты в функциональных компонентах. Функция очистки, возвращаемая из useEffect, выполняется при демонтировании компонента.

useEffect(() => {

console.log('Component mounted');

// Функция очистки вызывается при демонтировании компонента

**return () =>** {

console.log('Component will unmount');

};

}, []); // Пустой массив зависимостей означает, что эффект выполнится только при монтировании и демонтировании

**Расскажи про оптимизацию рендеринга в react?**

Это процесс снижения количества лишних перерисовок компонентов, что улучшает производительность приложения. React повторно рендерит компоненты, когда их состояние (state) или свойства (props) изменяются. Однако часто это приводит к ненужным рендерам, которые можно избежать.

**🚩Почему важна оптимизация рендеринга?**

🟠Производительность

Чем больше компонентов рендерится без необходимости, тем больше времени тратится на вычисления и обновления DOM.

🟠Экономия ресурсов

Избегая лишних рендеров, приложение работает быстрее, а пользовательский интерфейс становится более отзывчивым.

🟠Лучший UX

Плавная работа интерфейса критична для сложных приложений с большим количеством данных.

🚩Основные подходы к оптимизации рендеринга

🟠Мемоизация компонентов

React предоставляет утилиту React.memo, чтобы предотвращать ререндеринг компонента, если его props не изменились.

import React from 'react';

const MyComponent = React.memo(({ count }) => {

console.log('Рендер компонента');

return <div>Счётчик: {count}</div>;

});

🟠Использование `useMemo` и `useCallback`

Эти хуки используются для предотвращения повторных вычислений и создания функций при каждом рендере.

import React, { useMemo } from 'react';

🟠Разделение кода и ленивый рендеринг

Использование React.lazy и Suspense позволяет загружать компоненты только тогда, когда они необходимы.

import React, { Suspense } from 'react';

const LazyComponent = React.lazy(() => import('./LazyComponent'));

function App() {

return (

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

<LazyComponent />

</Suspense>

);

}

**Для чего нуден key при работе со списками?**

В React при рендере списков (например, через .map()) каждому элементу необходимо передавать уникальный key. Это помогает React оптимизировать перерисовку и правильно отслеживать изменения в списке.

**🚩Почему `key` важен?**

React использует виртуальный DOM и при обновлении сравнивает новый список с предыдущим.

Без key React не понимает, какие элементы изменились, удалились или добавились.

Это может привести к неожиданному поведению, например, сбросу состояния или неправильному рендеру.

**🚩Как использовать `key` правильно?**

const users = ["Alice", "Bob", "Charlie"];

function UserList() {

return (

<ul>

{users.map((user, index) => (

<li key={user}>{user}</li> // Хорошо, если `user` уникален

))}

</ul>

);

}

**🚩Что будет без `key`?**

Если не указать key, React выдаст предупреждение:

Warning: Each child in a list should have a unique "key" prop.

Кроме того, при изменении списка возможны проблемы с перерисовкой

{users.map((user, index) => (

<input key={index} defaultValue={user} />

))}