

JAVASCRIPT

1. [Разница между === и ==](#)
2. [Типы данных js](#)
3. [Разница между function arrow function / function declaration / function expression](#)
4. [Отличие null и undefined](#)
5. [This в strict и в обычном режиме](#)
6. [Что такое мемоизация и кеширование](#)
7. [Откуда у примитивов методы, автобоксинг](#)
8. [Почему typeof массива === object](#)
9. [Что такое Set](#)
10. [Что такое Map](#)
11. [Что такое WeakMap отличия от Map](#)
12. [Что такое WeakSet](#)
13. [Зачем нужен метод PreventDefault](#)
14. [Зачем нужен метод stopPropagation](#)
15. [Отличия Event.target от event.currentTarget](#)
16. [Распространение события и фазы](#)
17. [Стадия погружения события, как включить, всплытие события](#)
18. [Как работает Event-loop, микротаски и макротаски, порядок выполнения кода](#)
19. [Что такое DOM дерево](#)
20. [Отличия var let const](#)
21. [Поднятие или же hoisting переменных](#)
22. [Область видимости, scope, виды и как работает](#)
23. [Ключевое слово this, как работает и вычисляется](#)
24. [Отличия call, apply, bind](#)
25. [Прототипы, наследование, proto](#)
26. [Как работает promise](#)
27. [Как работает async await](#)
28. [Как работает Promise.all](#)
29. [Как работает Promise.allSettled отличие от Promise.all](#)
30. [Как работает Promise.race](#)
31. [Как работает Promise.any](#)
32. [Замыкание в js](#)
33. [Лексическое окружение](#)
34. [Как копировать или сравнивать 2 объекта](#)
35. [Как получить все ключи значения объекта](#)
36. [Ключевое слово super](#)
37. [Логические операторы ||, && !](#)
38. [Что делают операторы spread и rest, их отличия](#)
39. [Что такое функция, отличия от метода](#)
40. [Что такое static методы](#)
41. [Отличие синхронной функции от асинхронной](#)
42. [Как работает блок try catch](#)
43. [SOLID - принципы](#)
44. [dry kiss yagni](#)
45. [Babel зачем нужен](#)
46. [Методы массивов, какие из них мутирующие, а какие нет](#)

47. [Для чего нужны map / reduce / filter / concat, циклы для перебора массивов](#)
48. [Как сделать объект неизменяемым, методы Object.freeze Object.seal](#)
49. [Как сделать свойства объекта readonly или добавить новое Object.defineProperty](#)
50. [Как создать объект с указанным прототипом Object.create](#)
51. [Как проверить наличие свойства в объекте. in vs hasOwnProperty](#)
52. [Что такое SPA](#)
53. [Принципы ООП и классов](#)
54. [Конструкторы, функции без замыканий](#)
55. [Как работает ключевое слово new](#)
56. [Зачем нужны async defer и их отличия](#)
57. [Где лучше подключать script и почему](#)
58. [Что такое объект Proxy](#)
59. [Что такое функция высшего порядка](#)
60. [Что такое каррирование](#)
61. [Что такое compose и функция первого класса](#)
62. [Благодаря чему мы можем итерироваться по объектам, метод symbol.iterator](#)
63. [Как добавить задачу в очередь микротасок](#)
64. [Какие бывают структуры данных](#)
65. [Что такое service worker](#)
66. [Что такое web worker](#)
67. [Что такое ES module](#)
68. [Метод valueOf](#)
69. [Метод Symbol.toPrimitive](#)
70. [Операторы логического присваивания a &&= b](#)
71. [JS особенности](#)
72. [Рекурсия это](#)
73. [IIFE это](#)
74. [Что может заблокировать поток в js](#)
75. [Проваливание промисов это](#)
76. [Что такое boxing unboxing](#)
77. [Отличия примитивов от ссылочных типов данных](#)
78. [Что такое сборщик мусора](#)
79. [requestAnimationFrame что это](#)
80. [requestIdleCallback что это](#)
81. [Зачем нужен instanceof](#)
82. [Что такое микрофронтенд, webpack module federation](#)
83. [Что такое функции генераторы](#)
84. [Что такое гидрация](#)
85. [Что такое чистая функция](#)
86. [Дескрипторы свойств объектов](#)

TYPESCRIPT

1. [Отличия type от interface \(type, interface\)](#)
2. [Что такое Generics \(Дженерики\)](#)
3. [Что такое union](#)
4. [Что такое Intersection тип пересечения](#)
5. [Какие знаешь Utility types - record, pick, omit, required, partial](#)
6. [any vs unknown](#)
7. [Что такое Type Guard](#)
8. [Тип never](#)
9. [Что такое утверждение типов](#)
10. [Как работает keyof / typeof](#)
11. [Mapped types для чего](#)
12. [Что такое Implements](#)
13. [Как работают декораторы](#)
14. [Для чего нужно ключевое слово infer](#)
15. [Абстрактные классы, перегрузки функций](#)
16. [Для чего нужен Typescript, плюсы и минусы](#)
17. [Ошибки в typescript](#)
18. [Можно ли использовать typescript в серверной разработке](#)
19. [Такие же как и в JS: глобальные, блочные и функциональные](#)
20. [enum](#)
21. [Модификаторы доступа в typescript](#)
22. [как typescript поддерживает необязательные параметры и дефолтные параметры](#)

HTML, CSS

1. [CSS псевдоклассы и псевдоэлементы](#)
2. [CSS селекторы](#)
3. [CSS специфичность селекторов, как рассчитать специфичность](#)
4. [CSS normalize vs reset](#)
5. [CSS пост- и пре-процессоры зачем нужны](#)
6. [Семантика и семантическая верстка](#)
7. [Какие есть свойства position](#)
8. [Какие есть свойства display](#)
9. [Способы отцентровать блок](#)
10. [Margin vs padding, схлопывание margin](#)
11. [rem em](#)
12. [бэм](#)

13. [keyframes, transform](#)
14. [Какие способы есть вывести элемент из потока](#)
15. [Разница div span](#)

React

1. [Что такое Virtual Dom, зачем нужен, принципы работы, эвристический алгоритм](#)
2. [Зачем нужен key](#)
3. [Порталы в React](#)
4. [Что такое HOC, примеры хоков](#)
5. [Redux](#)
6. [Flux паттерн в редукс](#)
7. [Что такое store в redux](#)
8. [Что такое action в redux](#)
9. [Что такое reducer в redux](#)
10. [Что такое dispatch в redux](#)
11. [Как работает useState, зачем нужен](#)
12. [Как работает useEffect, зачем нужен](#)
13. [Как работает useEffect, зачем нужен](#)
14. [Как работает useRef, зачем нужен](#)
15. [Что такое контекст в реакт, хук useContext](#)
16. [Как работает useMemo, зачем нужен](#)
17. [Как работает useCallback, зачем нужен](#)
18. [В каких случаях нужно использовать мемоизацию и зачем](#)
19. [Как работает useReducer, зачем нужен](#)
20. [Offscreen](#)
21. [Что такое jsx](#)
22. [Какие хуки появились в React 18, useId, useTransition](#)
23. [Что такое контролируемые и неконтролируемые компоненты](#)
24. [Что такое React Fiber](#)
25. [Что такое React мемо и зачем нужен](#)
26. [React преимущества и недостатки](#)
27. [Что такое React lazy suspense](#)
28. [Порядок рендера компонентов и вызова хуков](#)
29. [Правила вызова хуков](#)

30. [Что такое батчинг, его плюсы](#)
31. [Зачем нужен хук forwardRef](#)
32. [Хук useImperativeHandle](#)
33. [extra reducer redux toolkit](#)
34. [Что такое middleware](#)
35. [Error boundaries что такое](#)
36. [Жизненный цикл компонента и методы жц в классах, порядок вызова](#)
37. [Почему ушли от классов к функциям в реакте](#)
38. [Причины перерисовки в реакт](#)
39. [Redux vs Context плюсы и минусы](#)
40. [Что такое виртуализация и зачем нужна](#)
41. [React переходы](#)

Браузер, сети, общение

1. [CORS, как работает, зачем нужен](#)
2. [CORS хедеры](#)
3. [Что такое preflight запросы, http метод OPTIONS](#)
4. [Какие http коды статусов есть](#)
5. [REST, какие есть принципы REST](#)
6. [Как работает HTTP, из чего состоит HTTP запрос](#)
7. [Как работает HTTPS и отличие от HTTP](#)
8. [HTTP отличия от HTTP 2](#)
9. [Отличие REST от GraphQL](#)
10. [Паттерны MVP MVC](#)
11. [Атрибуты для lazy загрузки медиа в html](#)
12. [Server Sent Events, Polling, Long Polling что такое и как юзается](#)
13. [TCP, UDP что такое и где используется, различия](#)
14. [Как работает протокол Websockets](#)
15. [Как работает браузер при вводе запроса, этапы рендера](#)
16. [Когда происходит Reflow Repaint](#)
17. [Что такое CSSOM](#)
18. [Какие есть способы оптимизации приложений](#)
19. [Как ты будешь дебажить приложение, из-за чего бывают утечки память](#)
20. [Отличия стека от очереди](#)
21. [Оценка сложности алгоритма, Big O](#)
22. [Что такое DNS](#)
23. [Что такое Shadow dom](#)
24. [Что такое jwt](#)

25. [Авторизация это](#)
26. [Access и refresh токены, что это и где безопасно хранить](#)
27. [Что делать если JWT токен истек](#)
28. [Cookie что это](#)
29. [Параметры настройки cookie \(secure, httpOnly...\)](#)
30. [SessionStorage и LocalStorage + отличия](#)
31. [Git merge vs rebase](#)
32. [Что такое подмодули \(submodules\) в гите и для чего они нужны?](#)
33. [В чём отличие ReactDOM.render от ReactDOM.hydrate \(или createRoot от hydrateRoot \)](#)
34. [Что такое простой запрос](#)
35. [HTTP методы, их отличия и особенности](#)
36. [Что такое WebRTC](#)
37. [OWASP уязвимости в браузере](#)
38. [Что такое Same Origin Policy](#)
39. [Основные концепции Webpack, loaders, plugins](#)
40. [Отличия webpack modules, chunks, bundle](#)
41. [Набор директив Content Security Policy \(CSP\). Зачем они нужны? Приведите примеры директив.](#)
42. [Опишите принцип работы http заголовка If-Modified-Since](#)
43. [Что такое Идемпотентность](#)
44. [Безопасные http методы](#)
45. [Что выполняет команда npm ci](#)
46. [Метрики core web vitals](#)

Разница между === и ==

Оператор строгого равенства === проверяет равенство без приведения типов. Если значения имеют разные типы, то они не могут быть равными. Перед сравнением оператор == равенства приводит обе величины к общему типу.

Оператор строгого равенства ===:

Сравнивает не только значения, но и типы значений.

Возвращает true, только если оба значения имеют один и тот же тип и равны.

Является более предсказуемым и обычно предпочтительней для использования, так как не происходит преобразование типов.

Оператор абстрактного равенства ==:

Сначала выполняет неявное преобразование типов, если значения сравниваемых переменных относятся к разным типам, и только затем сравнивает их уже как значения одного типа.

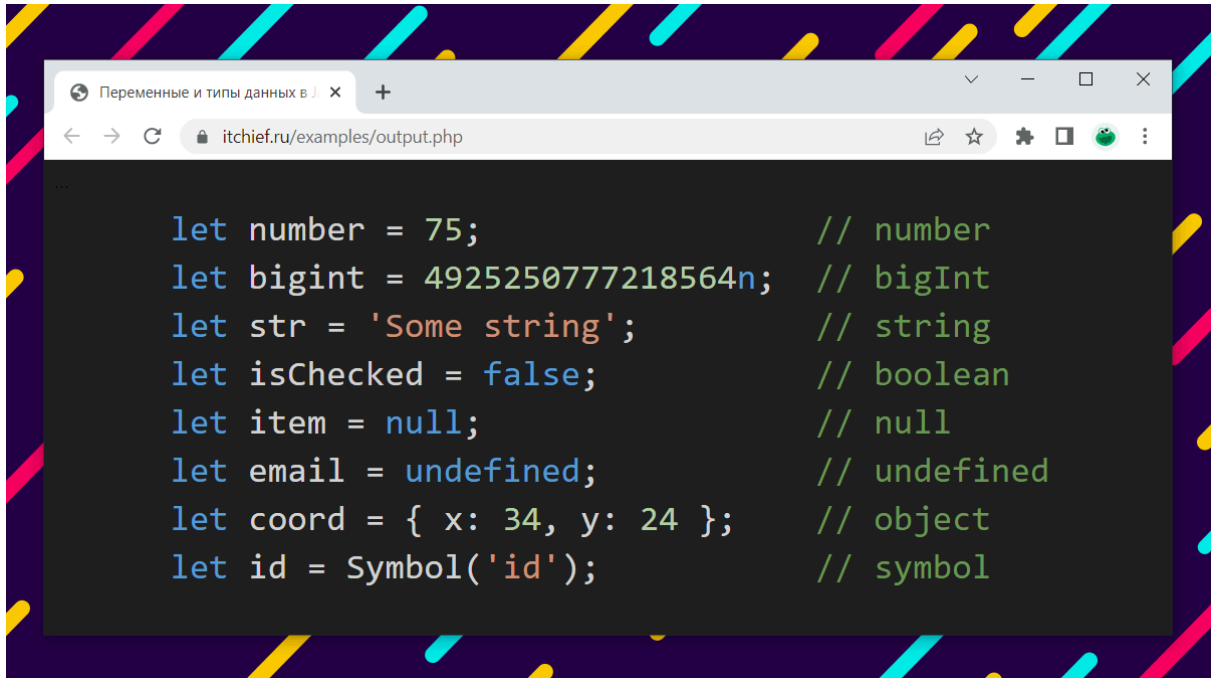
Возвращает true, если значения равны после преобразования типов.

Может приводить к неочевидному поведению из-за преобразования типов

```
let result = "5" - 1; // 4, строка "5" преобразуется в число
```

```
let result2 = "5" + 1; // "51", число 1 преобразуется в строку
```

Типы данных js



The image shows a web browser window with a dark-themed code editor. The browser's address bar shows the URL `itchief.ru/examples/output.php`. The code editor contains several JavaScript variable declarations, each followed by a comment indicating the data type. The code is as follows:

```
let number = 75;           // number
let bigint = 4925250777218564n; // bigInt
let str = 'Some string';    // string
let isChecked = false;      // boolean
let item = null;            // null
let email = undefined;      // undefined
let coord = { x: 34, y: 24 }; // object
let id = Symbol('id');      // symbol
```


Разница между arrow function / function declaration / function expression

Объявление функции (Function Declaration - базовое объявление функции)

Наиболее классический способ создания функций в JavaScript. Функция объявляется с указанием имени и может быть вызвана до объявления благодаря механизму поднятия (hoisting).

```
function имяФункции(параметры ) {  
    // Тело функции  
};  
  
// Пример декларативного объявления  
function greet(name) {  
    console.log("Привет, " + name + "!");  
};
```

Минусы:

- Такую функцию нельзя присвоить переменной или передать в качестве аргумента другой функции.
- Функцию можно использовать до её объявления — преимущество легко превращается в недостаток, потому что код сложнее читать и про функцию можно забыть.

Объявление функции (Function Expression - (функциональные выражения))

Для чего использовать: для функции, которую нужно записать в переменную. А ещё способ полезен, когда функция должна быть доступна только после её объявления.

Сначала мы объявляем переменную — у нас это greet. Затем присваиваем ей значение — функцию. Эта функция пишется так же, как и при декларативном объявлении. Но в отличие от него здесь имя функции чаще всего опускается — то есть она является анонимной.

```
const имяПеременной = function(параметры) {  
    // Код функции  
};  
  
// Пример функционального выражения  
const greet = function(name) {  
    console.log("Привет, " + name + "!");  
};
```

Плюсы:

Так как функция присваивается переменной, её легко передавать в другие функции. Можно создавать функции, которые используются только внутри определённого контекста.

Минусы:

Нельзя использовать функцию до её объявления, если она анонимна — а так чаще всего и есть.

Объявление функции (Arrow function - (стрелочные функции))

Function имеет свой [this](#) контекст, а у стрелочных функций **собственного контекста выполнения нет**. Они связываются с ближайшим по иерархии контекстом, в котором они определены. Проще говоря, функция не создает собственный контекст исполнения, она использует внешний.

В обычной функции `This` равен контексту в котором функция была вызвана.

Особенности стрелочных функций:

1. Лексический контекст [this](#):

Стрелочные функции не создают своего собственного `this`. Вместо этого, `this` внутри стрелочной функции определяется в момент создания функции и это значение `this` захватывается из окружающей (то есть, внешней) лексической области, где стрелочная функция была определена.. Лексический контекст - это, по сути, контекст, где функция была создана, а не где она вызывается.

Таким образом, стрелочные функции наследуют `this` от родительской области видимости в момент своего создания. Такое поведение особенно полезно при использовании стрелочных функций в качестве обработчиков событий или при использовании методов с коллбэками, таких как `setTimeout` и массивные операции вроде `map`, `filter` и `reduce`.

2. Отсутствие `arguments`:

Стрелочные функции не имеют своего объекта **`arguments`**, который существует в функциях, созданных с использованием функционального выражения или объявления. Если вам нужно работать с аргументами в стрелочной функции, вы можете использовать оператор расширения `REST` оператор (`...`), чтобы получить "список" аргументов как массив.

`arguments` — это объект, напоминающий массив, который содержит значения всех аргументов, переданных функции, с которой он связан. Он доступен внутри традиционных функций, объявленных с помощью ключевых слов `function` или `function*`, но не доступен в стрелочных функциях, классах или методах класса.

Свойство `arguments` позволяет функции работать с переменным числом аргументов, не указывая их явно. Вы можете обращаться к элементам `arguments` так же, как и к элементам массива, хотя на самом деле это не массив, и ему не доступны все методы массива, такие как `map`, `filter` или `reduce`.

Следует учитывать, что использование объекта `arguments` может снизить производительность, также с появлением синтаксиса остаточных параметров (`rest parameters`) `[...args]` употребление `arguments` стало менее актуальным, так как `rest parameters` предоставляют более удобный и прозрачный способ работы с переменным числом аргументов.

3. Стрелочные функции не могут быть конструкторами:

Стрелочные функции не имеют свойства [prototype](#), которое есть у обычных функций. Именно это свойство используется при создании новых объектов с `new`. Попытка использовать стрелочную функцию с `new` приведет к ошибке, так как стрелочные функции не могут быть использованы в качестве конструктора.

4. Нельзя изменить `this` с использованием [bind](#), [call](#) или [apply](#):

Поскольку `this` в стрелочной функции лексически связано с окружающим контекстом, методы `bind`, `call` и `apply` не изменяют значение `this` внутри стрелочной функции. Эти методы будут безэффектны.

****Конструктор функций (Function Constructor):****

```
```javascript
const sum = new Function('a', 'b', 'return a + b');
```
```

- Этот способ создания функций используется редко, и он позволяет создавать функции динамически на основе строковых аргументов.

Отличие `null` и `undefined`

`null` и `undefined` в JavaScript представляют отсутствие значения, но в разных аспектах: `undefined` - это значение переменной, которое не было инициализировано, то есть, когда переменная объявлена, но ей не присвоено никакого значения. `null` - это намеренное отсутствие значения и используется для обозначения, что переменная должна быть "пуста" или "не иметь значения". `Undefined` получаем, когда значение переменной не определено.

Если в массиве нет значения, которого мы от него требуем

Если в объекте нет такого значения

Если в функции ничего не возвращает

```
let varWithoutValue;
console.log(varWithoutValue); // Вывод: undefined
```

```
let varWithNull = null;
console.log(varWithNull); // Вывод: null
```

При нестрогом сравнении

```
console.log(null == undefined); // Вывод: true
```

При строгом сравнении

```
console.log(null === undefined); // Вывод: false
```

Тип undefined

```
console.log(typeof undefined); // Вывод: "undefined"
```

Тип null

```
console.log(typeof null); // Вывод: "object"
```

Что такое мемоизация и кеширование

. **Мемоизация** – сохранение результата каких-либо вычислений, чтобы потом не пересчитывать заново.

Кеширование - процесс сохранения данных локально, для быстрого доступа при повторном запросе.

Откуда у примитивов методы, автобоксинг

Каждый примитив имеет свой собственный объект-обёртку, которые называются: String, Number, Boolean, Symbol и BigInt.

А прототипы этих объектов имеют разный набор методов. Это Autoboxing.

Когда вы пытаетесь вызвать метод на примитиве, например, 'hello'.toUpperCase(), движок JavaScript временно оборачивает примитив в соответствующий объект-обертку (String, Number, или Boolean), который содержит методы и свойства, доступные для работы с типом данных.

Этот процесс называется "boxing" (автоупаковка), и во время его выполнения примитивный тип данных временно превращается в объект, чтобы предоставить доступ к его методам. Непосредственно после вызова метода объект-обертка удаляется (это "unboxing"), и результат возвращается как примитивное значение.

Почему typeof массива === object

В JavaScript массивы являются разновидностью объектов, поэтому, когда вы используете оператор typeof для проверки типа массива, он возвращает "object".

это происходит потому, что в JavaScript массивы фактически являются объектами с дополнительными возможностями - они имеют специальное свойство length, которое автоматически обновляется при добавлении или удалении элементов, и набор встроенных методов для работы с массивами, таких как .push(), .pop(), .slice(), .map() и многих других.

Чтобы более точно определить, является ли объект массивом, можно использовать глобальный метод Array.isArray()

Array.isArray() возвращает true, если переданное ему значение является массивом, и false в противном случае

Что такое Set

Set в JavaScript (ES6) — это структура данных, которая позволяет хранить уникальные значения любого типа, будь то примитивы или ссылочные типы данных. Set гарантирует, что значение может появиться в нем только один раз, что делает его идеальным для ситуаций, когда необходимо исключить повторяющиеся данные.

Основные особенности и методы Set:

.add(value) — добавляет значение в Set.

.delete(value) — удаляет значение из Set.

.has(value) — проверяет, существует ли значение в Set.

.clear() — удаляет все значения из Set.

.size — возвращает количество элементов в Set.

Set часто используют для удаления дубликатов из массива, так как все значения в Set автоматически становятся уникальными

Что такое Map

Map — коллекция для хранения записей вида ключ:значение. В отличие от объектов, в которых ключами могут быть только string и symbol (другие типы данных приводятся к string), в Map ключом может быть

произвольное значение, такое как объект, массив, функция. Также по умолчанию map не содержит никаких ключей, а объект наследует свойства и методы из своего прототипа.

Вот некоторые из основных методов и свойств Map:

.set(key, value) — добавляет элемент с ключом key и значением value в Map.

.get(key) — возвращает значение, ассоциированное с ключом key, либо undefined, если ключ не найден.

.has(key) — возвращает true, если Map содержит элемент с ключом key.

.delete(key) — удаляет элемент по ключу key.

.clear() — удаляет все элементы из Map.

.size — возвращает количество элементов в Map.

Что такое WeakMap отличия от Map

В отличие от **Map** в **WeakMap** ключи должны быть объектами, а не примитивными значениями, и если мы используем объект в качестве ключа и на этот объект нет ссылок, то он будет удалён из памяти.

WeakMap не поддерживает перебор и методы keys(), values(), entries(), так что нет способа взять все ключи или значения из неё.

Что такое WeakSet

В **WeakSet** мы можем добавлять в только объекты (не примитивные значения). Объект присутствует в сете только до тех пор, пока доступен где-то ещё. Как и Set, она не поддерживает size, keys() и не является перебираемой

Зачем нужен метод PreventDefault

JavaScript метод **preventDefault()** объекта **Event** при вызове отменяет действие события по умолчанию (Клик по ссылке: Переход на новую страницу или загрузка ресурса., Отправка формы: Переход на другую страницу или перезагрузка текущей

страницы с результатами отправки формы., Нажатие клавиш в текстовом поле: Ввод текста в форму. и прочие). Событие продолжает распространяться как обычно, только с тем исключением, что событие ничего не делает.

Зачем нужен метод `stopPropagation`

Метод **`stopPropagation`** у объекта `Event` предназначен для предотвращения всплытия события. Всплытие события – это процесс, при котором событие, возникшее на одном из элементов веб-страницы, поочерёдно передается вверх по DOM-дереву от самого вложенного элемента (цели события) до корня документа.

По пути всплытия событие может быть обработано любым элементом (имеющим соответствующий обработчик), не только элементом, на котором оно сначала сработало. **Если не использовать метод `stopPropagation`**, событие достигнет самого верхнего уровня DOM-дерева (обычно это объект `document`).

Отличия `Event.target` от `event.currentTarget`

`Event.target` — это элемент, в котором происходит событие, или элемент, вызвавший событие.

`Event.currentTarget` - элемент, к которому прикреплен прослушиватель событий

Распространение события и фазы

Событийная модель в браузере описывает способ обработки событий, таких как клики мыши, нажатия клавиш, изменения размеров окна и других пользовательских взаимодействий. В этой модели события обрабатываются с использованием концепции всплытия и погружения.

Событие распространяется от объекта **`Window`** до вызвавшего его элемента (**`event.target`**). При этом

событие затрагивает всех предков целевого элемента. Распространение события имеет три фазы:

- Фаза **погружения** (**`capturing phase`**) – событие сначала идёт сверху вниз.
- Фаза **цели** (**`target phase`**) – событие достигло целевого элемента.
- Фаза **всплытия** (**`bubbling stage`**) – событие начинает всплывать.

Стадия погружения события, как включить

Стадия погружения по умолчанию не используется и чтобы ее включить нужно в **`addEventListener`** передать 3 аргумент **`{capture: true}`**

Когда на элементе происходит событие, обработчик сначала срабатывает на нём, потом всплывает по цепочке предков. **Всплытие** позволяет делегировать

события, повесив обработчик события на родительский элемент, мы можем обрабатывать клики по дочерним элементам.

Как работает Event-loop, микротаски и макротаски, порядок выполнения кода

event loop не является частью js. Отдельный механизм, который позволяет использовать не блокирующий модель ввода вывода.

Пример: сайт, отправлять запрос, выводить анимацию, обработка кнопки была независима...

event loop (цикл событий) - отвечает за очередь задач (**task queue**). Он не является частью движков. Цикл событий предоставляется средой. Например, браузер или node js.

call stack - (стек вызовов) - за его обработку отвечает движок javascript'a (V8: Движок, разработанный Google, используется в Google Chrome и Chromium, а также является сердцем среды выполнения Node.js и многих других сред, таких как Deno и Electron.)

Что решает js движок:

- Куча (heap) и стек вызовов (call stack)
- Работа с памятью (выделение и сбор мусора)
- Компиляция js в машинный код
- Оптимизация (кеши, скрытые классы и прочее)

Если у нас **движок js** предоставляет **call stack** (стек вызовов), а **task queue** (очередь задачи) предоставляет **event loop**, то как они общаются между собой?

Ответ. У каждого браузера есть **Web api**.

Web API предоставляет JavaScript набор асинхронных и синхронных функциональностей, которые расширяют возможности работы в веб-браузере. Включают следующее:

1. ****Доступ к элементам DOM****: API для манипуляции с HTML и стилями (например, `document.querySelector`, `document.getElementById`, `element.innerHTML`, `element.style`).
2. ****События****: Обработка пользовательских событий, таких как клики мыши, нажатия клавиш и события сенсорного экрана.
3. ****AJAX (XMLHttpRequest) и Fetch API****: Функциональность для выполнения HTTP-запросов к серверу и получения ответов без перезагрузки страницы.
4. ****Таймеры****: Функции установки и сброса таймеров с задержкой (`setTimeout`, `setInterval` и их "сброски" `clearTimeout`, `clearInterval`).
5. ****Веб-хранилище****: Интерфейсы для хранения данных в браузере, такие как `localStorage` и `sessionStorage`.

6. ****Service Workers****: Скрипты, работающие в фоновом режиме, обеспечивающие возможности offline-работы, кэширования и фоновой синхронизации.
7. ****WebSockets****: API для двустороннего взаимодействия между браузером и сервером по протоколу WS.
8. ****File и FileReader****: API для работы с файлами, которые позволяют читать данные файлов, выбранных пользователем.
9. ****Canvas и WebGL****: API для рисования в 2D и 3D.
10. ****WebRTC****: API для организации потоковой передачи аудио, видео и общения между браузерами в реальном времени.
11. ****Geolocation****: API для получения географической информации о местоположении пользователя.
12. ****History API****: Функциональность для манипуляции историей браузера и URL без перезагрузки страницы.

Есть стек вызовов (call stack). Стек вызовов - определенная структура данных, хранилище.

Call stack не бесконечный. При переполнении приложение может крашиться. Пример: рекурсия, передаем огромное число.

Задачи (асинхронщина) из очереди задач (task queue) могут попасть в стек (call stack) ТОЛЬКО после вызова всех функций из стека

Пример работы:

```
function log(value) {  
    console.log(value);  
}  
  
log('start');  
  
setTimeout(() => {  
    log('timeout');  
}, 3000);  
  
log('end');
```

Ответ: сначала 'start', затем сразу 'end' и после 3 секунд ожидания 'timeout'.

Почему так?

после log 'start' интерпретатор добавляет setTimeout в call stack, после этого он регистрируется в web api, на этом этапе запускается таймер. После того, как таймер истек,

мы отправляем наш call back в очередь задач (task queue) и он после того, как наш call stack (стек вызовов) очистился, он берет задачу из очереди задачи (task queue).

То есть, выполнились все синхронные задачи, после истечения таймера, задача попадает в очередь и после того, как все синхронные задачи выполнены, мы берем задачу из очереди и выполняем его.

Немного усложненный теперь пример:

```
function log(value) {  
  console.log(value);  
}  
log('1');  
setTimeout(() => {  
  log('2');  
}, 0);  
Promise.resolve().then(() => {  
  log('3');  
});  
log('4');
```

Результат выполнения этого кода 1,4,3,2

Почему такой порядок:

Все дело в том, что очередей 2. Очередь микрозадач и очередь макрозадач. У этих очередей есть свой приоритет. Event loop (берет в опр-ом порядке)

Промисы всегда попадают в очередь микротасок. Затем у нас выполняется log 4, на текущем этапе все синхронные задачи выполнены, call stack (стек вызовов) пустой,

В приоритете выполняются микротаски, причем сразу все и потом выполняется одна макротаска

log('1') вызывается синхронно и выводит 1.

setTimeout с задержкой 0 мс помещает log('2') в очередь макротасок.

Promise.resolve().then(...) добавляет log('3') в очередь микротасок, которая будет обработана немедленно после текущего блока выполнения скрипта, но до обработки задач из очереди макротасок (например, вызовов setTimeout).

log('4') вызывается синхронно и сразу выводит 4.

После того, как основной поток выполнения скрипта завершён, Event Loop обрабатывает микротаски — выводится 3.

Только после завершения микротасок обрабатываются макротаски, ожидающие в своей очереди - выводится 2.

итог: сначала выполняются все синхронные задачи, потом выполняются все микротаски (промисы), потом выполняется одна макротаска (setTimeout), если она порождает другие микротаски, выполняются они все, затем выполняется макротаска.

все микротаски, одна макротаска и так по кругу.

Что создает микротаски?

1. Промисы (на 99 процентов)
2. `queueMicrotask` - можно создать явно промисы метод, позволяющий явно добавить микротаску в очередь.
3. `MutationObserver` - то API, которое можно использовать для отслеживания изменений в DOM дереве. Когда изменения происходят, колбэки `MutationObserver` помещаются в очередь микротасок.

Что создает макротаски?

1. Таймеры (`setTimeout`, `setInterval`)
2. События (клик, загрузка изображения и т.д.)
3. Браузерные нюансы (рендер, `input`, `output` (ввод, вывод) и т.д.)

Повторение: выполняются все синхронные задачи из `call stack`, затем `event loop` обрабатывает все таски, которые находятся в `microtask`, затем выполняется одна макротаска

Отличия var let const

До прихода escript 6 можно было создавать только с помощью ключевого слова **var**. Теперь можно создавать еще с 2 ключевыми словами, как **let** и **const**

Переменная - контейнер для хранения информации

1. `var`:

- Область видимости: Имеет функциональную область видимости, что означает, что переменная, объявленная с помощью нее внутри функции, доступна в любом месте этой функции. Если она объявлена вне функции, она становится глобальной.
- Поднятие (Hoisting): Переменные, объявленные через нее, поднимаются в начало функции или глобальной области видимости, но инициализация остаётся на своём месте. Это значит, что переменную можно использовать до её объявления в коде.
- Перезаписываемость: Можно повторно объявить и изменить переменную, используя ее.

2. `let`:

- Область видимости: Имеет блочную область видимости, ограниченную фигурными скобками `{}`, в которых она была объявлена, например, внутри циклов, условий или блоков кода.
- Поднятие (Hoisting): Поднятие происходит, но в отличие от `var`, доступ к переменной до её объявления вызывает ошибку `ReferenceError`.
- Перезаписываемость: Можно изменить значение переменной, но нельзя повторно объявить её в той же области видимости.

3. `const`:

- Область видимости: Как и `let`, имеет блочную область видимости.
- Поднятие (Hoisting): Поднимается так же, как и `let`, с теми же ограничениями доступа до объявления.
- Перезаписываемость: Нельзя изменить значение. Однако, если переменная ссылается на объект или массив, то можно изменить содержимое объекта или массива, но не саму ссылку.
- Необходима инициализация: При объявлении переменной с `const` необходимо сразу же инициализировать её значением.
- При объявлении в глобальном контексте выполнения - добавляется как свойство в объект window

Что такое DOM дерево

DOM —объектная модель документа представлена в виде дерева, которую создает браузер на основе html кода, полученного от сервера, с помощью dom json можем взаимодействовать с элементами

Virtual DOM - это легковесная копия **DOM**, хранимое в оперативной памяти, которое синхронизируется

с **RDOM**. Помогает избежать прямой работы с **DOM** из-за производительности (т.к изменения **VDOM** не

отрисовываются на экране. Из-за того, что это простой отдельный js объект, мы можем свободно

изменять его свойства, не затрагивая актуальный **RDOM**, до тех пор, пока нам это не понадобится).

При изменении состояния компонента, React обновляет **VDOM**. После обновления **VDOM**, React

сравнивает его текущую версию дерева с предыдущей. (**diffing**)

-

- Затем вычисляется разница между предыдущим и новым представлениями DOM для определения того, какие части должны быть обновлены. (**алгоритм reconciliation** выполняет обход в глубину, начиная с корня дерева и сравнивает каждый элемент и его потомков)

- После этого используя наработки из предыдущей фазы, вызываются методы жизненного цикла, хуки и обновляются только те части реального DOM, которые подверглись изменениям.

В ответ на событие строится новое Work-in-progress дерево, которое, рекурсивно сравнивается с Current деревом для определения различий. Эти различия передаются в среду Rendering Environment для повторной отрисовки и обновления DOM. После этого Work-in-progress дерево становится в Current деревом, после следующего изменения, цикл повторяется.

****BOM (Browser Object Model)**** - это объектная модель браузера, которая предоставляет доступ к функциональности браузера, которая не связана с документом HTML. Она включает в себя объекты, которые представляют *окно браузера*, *историю браузера*, параметры *URL-адреса*, *cookies* и т.д.

Некоторые из основных объектов BOM включают в себя:

1. **``window``** - объект, представляющий окно браузера. Он содержит множество свойств и методов, таких как **``alert()``**, **``setTimeout()``**, **``setInterval()``** и т.д.

2. **``document``** - объект, представляющий текущий HTML-документ в окне браузера.

3. **``location``** - объект, представляющий текущий URL-адрес, который отображается в адресной строке браузера.

4. **``history``** - объект, позволяющий перемещаться по истории браузера.

5. **``navigator``** - объект, содержащий информацию о браузере и операционной системе пользователя.

6. `screen` - объект, представляющий информацию о характеристиках экрана пользователя.

7. `cookies` - объект, позволяющий установить, получить или удалить cookie-файлы.

BOM позволяет создавать взаимодействие между пользователем и браузером, например, показывать сообщения, устанавливать и получать cookie-файлы, управлять историей браузера и т.д.

Таким образом, основное отличие между DOM и BOM заключается в том, что DOM фокусируется на структуре и содержимом документа, тогда как BOM предоставляет инструменты для взаимодействия с окном браузера и другими элементами браузерного окружения.

Порталы Portal - это способ визуализации элемента в узле DOM, который существует вне иерархии DOM родительского компонента.

Поднятие или же hoisting переменных

Поднятие или hoisting — это механизм JS, в котором переменные и объявления функций, передвигаются вверх своей области видимости перед тем, как код будет выполнен. Из-за того что ещё до выполнения кода интерпретатор загружает в память функции и переменные.

Область видимости (scope), виды и как работает

Область видимости scope, это место откуда мы имеем доступ к переменным, функциям и другим данным, и можем к ним обратиться. Разделить область видимости можно на глобальную, блочную и функциональную. Глобальная – переменные доступны из любого места в кода. Блочная – когда переменные доступны только, например в блоке if. Функциональная— это область видимости в пределах тела функции. При создании локальной области видимости она сохраняет ссылку на внешнюю область видимости, таким образом получаются цепочки областей видимости и мы можем в 1 ОВ обращаться к переменным из 2 ОВ. ОВ создается каждый раз при вызове функции.

Область видимости (scope) переменных определяется местом в коде, где эти переменные объявлены, и влияет на доступность переменных в различных частях программы

```
function outer() { // Первая область видимости (внешняя)
  let varOuter = 'Я внешняя переменная';

  function inner() { // Вторая область видимости (внутренняя, локальная
    для outer)
      let varInner = 'Я внутренняя переменная';
      console.log(varOuter); // Доступ к переменной из внешней области
видимости
    }

    inner();
  }

  outer();
}
```


Ключевое слово **this**, как работает и ВЫЧИСЛЯЕТСЯ

this это ключевое слово в JS которое указывает на контекст исполняемого кода. То есть это ссылка на объект, который «владеет» текущим исполняемым кодом или функцией. Когда функция вызывается как метод объекта, **this** принимает значение объекта, по отношению к которому вызван метод, то значение объекта перед точкой.

Но в стрелочных методах **this** относится к вышестоящему контексту т.к они не привязаны к собственным сущностям и не имеют своего **this**. В стрелочных функциях **this** определяется в момент создания, а в обычных **this** вычисляется в момент вызова и равен объекту перед точкой. Если такого объекта нет — тогда **this** будет указывать на глобальный контекст (window)

This - это ключевое слово, которое ссылается на объект, в контексте которого был вызван код. **This** в обычной функции является динамическим, так как он определяется во время выполнения функции, а в стрелочной функции - статическим, так как будет ссылаться всегда на окружение, внутри которого функция была определена.

This в strict и в обычном режиме

Если вы попытаетесь обратиться к ключевому слову **this** в глобальной области видимости, при включенном strict-mode оно будет === undefined, в обычном режиме - window

Значение **this** в JavaScript зависит от контекста, в котором функция вызывается, а также от режима, в котором выполняется код — строгим ("strict mode") или обычном.

В обычном режиме (non-strict mode):

Если функция вызывается обычным способом, то есть не как метод объекта, **this** ссылается на глобальный объект (window в браузере, global в Node.js).

```
javascript
function myFunc() {
```

```
    console.log(this); // Выведет глобальный объект (window или global)
}
myFunc();
```

Когда функция вызывается как метод объекта, `this` ссылается на объект.

```
javascript
const myObject = {
  myMethod() {
    console.log(this); // Выведет myObject
  }
};
myObject.myMethod();
```

В конструкторе (функция вызвана с `new`), `this` ссылается на новый создаваемый экземпляр объекта.

В строгом режиме (strict mode):

Если функция вызывается как свободная функция (не как метод объекта), то `this` будет `undefined`, вместо ссылки на глобальный объект.

```
javascript

"use strict";
function myFunc() {
  console.log(this); // Выведет undefined
}
myFunc();
```

При вызове функции как метода объекта, [this](#) по-прежнему будет ссылаться на объект.

В конструкторе строгое поведение `this` идентично нестрогому режиму — `this` ссылается на новый экземпляр объекта.

Отличия call, apply, bind

Методы `call`, `apply` и `bind` принадлежат к функциональному объекту `Function` и используются для указания контекста `this` при вызове функции. Хотя все три метода позволяют контролировать значение `this` внутри функции, между ними существуют ключевые различия в способе использования и поведении.

Метод `bind()` создаёт новую функцию и позволяет указать какой объект будет привязан к `this` во время ее вызова. `bind` не вызывает функцию, а лишь возвращает "обёртку", которую можно вызвать позже. функция не вызывается мгновенно

`Bind`, `call`, `apply` не работает для стрелочных функций, т.к у них нельзя изменить `this`, но переданные в `bind` аргументы забиндятся

`call` и `apply` очень похожи – они вызывают функцию с указанным контекстом `this` и дополнительными аргументами. Единственная разница между `call` и `apply` заключается в том, что `call` требует, чтобы аргументы передавались по одному, а `apply` принимает их в виде массива

Прототипы, наследование, proto

Прототипы – объекты, имеют объект-прототип, который выступает как шаблон, от которого объект наследует методы и свойства. Объект-прототип так же может иметь свой прототип. Прототипное наследование — позволяет создавать объекты на основе других объектов, наследуя их свойства и методы,

prototype - это объект, который используется для построения `_proto_`, когда вы создаете объект с помощью оператора `new`. Это СВОЙСТВО ЕСТЬ ТОЛЬКО У КЛАССОВ И FUNCTION т.е только у функций-конструкторов.

Геттер `__proto__` возвращает прототип объекта

Сеттер `__proto__` позволяет установить прототип для объекта

`_proto_` - это свойство setter/getter, которое присутствует у ВСЕХ объектов в js кроме **`Object.create(null)`**, которое ссылается на `.prototype` функции конструктора с помощью которого этот объект был создан. То есть это ссылка на прототип родительского объекта. То есть `[1,2].__proto__ === Array.prototype`

Если свойство не найдено в текущем объекте, оно будет искаться в цепочке прототипов, и так по цепочке прототипов, пока не дойдет до `null`.

Помимо `proto`, для доступа к прототипу можем юзать - `getPrototypeOf`, `setProootypeOf`, `Object.create`. Можем добавлять методы - `Object.prototype.sayHi = () => {}`

Использование `__proto__` напрямую считается устаревшей практикой и не рекомендуется к использованию в новом коде. Вместо этого предпочтительнее использовать функции `Object.getPrototypeOf()` и `Object.setPrototypeOf()` для получения и изменения прототипов объектов.

Как работает promise

Promise (обещание) — это объект, представляющий завершение (или неудачу) асинхронной операции и её результат. Он позволяет ассоциировать обработчики с асинхронным действием, тем самым избавляя от необходимости использовать обратные вызовы (callback-функции). Они упрощают работу с асинхронными операциями, такими как AJAX-запросы или чтение файлов, позволяя написать код, который проще понять и поддерживать.

Состояния:

1. **Pending** (Ожидание): Начальное состояние; асинхронная операция не завершена.
2. **Fulfilled** (Исполнено): Операция завершена успешно, и promise возвращает результат.
3. **Rejected** (Отклонено): Операция завершена с ошибкой, и promise возвращает причину отказа.

Promise поддерживает цепочки вызовов (`then`), что позволяет организовывать асинхронный код последовательно и читабельно. Кроме того, существуют вспомогательные методы, такие как `Promise.all`, `Promise.race`, `Promise.resolve`, и `Promise.reject`, которые облегчают работу с группами асинхронных операций.

Promise.all принимает массив промисов и возвращает промис, который ждёт, пока все переданные промисы завершатся, и переходит в состояние fulfilled с массивом их результатов или rejected, если хотя бы один из переданных промисов

завершится с ошибкой. Метод используют, когда нужно запустить несколько промисов параллельно и дождаться их выполнения.

Promise.allSettled() принимает массив промисов, выполняет их параллельно, и в отличие от **Promise.all**

всегда ждёт завершения всех промисов, неважно завершились они с ошибкой, или нет. В итоге

возвращает промис с массивом результатов

{status:"fulfilled", value:результат}

Promise.race() принимает массив промисов и возвращает новый промис. Он завершится, когда завершится самый быстрый из всех переданных (неважно успешно или с ошибкой). Остальные промисы будут проигнорированы.

Promise.any() в отличие от **race** ожидает первый успешно выполненный промис, возвращает его, остальные игнорируются. Если все переданные промисы отклонены, возвращается **AggregateError**

Promise — это способ организации асинхронного кода, который предоставляет более удобный и понятный интерфейс для работы с асинхронными операциями, чем традиционные **callback**-функции. У каждого обещания есть три состояния: ожидание, исполнено и отклонено, которые помогают управлять результатом асинхронных операций.

Промис обеспечивает несколько методов:

then(onFulfilled, onRejected): добавляет обработчики успешного исполнения (**onFulfilled**) и/или отклонения (**onRejected**) промиса.

catch(onRejected): добавляет обработчик отклонения и является сокращением для **then(null, onRejected)**.

finally(onFinally): добавляет обработчик, который выполняется после завершения промиса, независимо от его исхода.

Как работает **async await**

async и **await** — ключевые слова в JavaScript, которые используются для упрощения работы с асинхронным кодом, особенно с промисами. Они позволяют писать асинхронный код, который выглядит и ведёт себя больше как синхронный код, делая его более читаемым и понятным.

В качестве результата **async** функция всегда возвращает промис. В **async** функциях можно

использовать ключевое слово **await**, которое указывается перед промисом и заставит интерпретатор

ждать до тех пор, пока этот промис не выполнится, после чего оно вернёт его результат, и выполнение

кода продолжится. JS в это время может выполнять другие задачи. Т.к асинхронный код выполняется в

фоне и не блокирует основной поток. Для избежания **callback hell**'а, и блокировки потока

Замыкание в js

Замыкание — это комбинация функции и лексического окружения, в котором эта функция была определена. Когда у вас одна функция находится внутри другой, то внутренняя функция имеет доступ к переменным внешней функции. Замыкание - это возможность для внутренней функции получать доступ к переменным которые находятся в родительской функции даже после того как родительская функция завершила свое выполнение! создаются каждый раз при создании функции.

Лексическое окружение

Лексическое окружение - это скрытый объект, который связан с функцией и создаётся во время ее вызова! В нём находятся все локальные переменные этой функции, и ссылка на внешнее лексическое окружение.

Как копировать или сравнивать 2 объекта

Копировать объект - не сериализуемые данные: undefined, функция, symbol, Date - при вызове

Для копирования объекта в JavaScript существует несколько способов, в том числе поверхностное (shallow) и глубокое (deep) копирование.

****Поверхностное копирование.****

1. С помощью распространения объектов (Object Spread):

```
```javascript
let original = { a: 1, b: 2 };
let copy = { ...original };
```
```

2. С помощью `Object.assign`:

```
```javascript
let original = { a: 1, b: 2 };
let copy = Object.assign({}, original);
```
```


...

Оба эти метода создают новый объект с копиями свойств исходного объекта. Однако, если исходный объект содержит вложенные объекты, то в копии они останутся ссылками на оригинальные вложенные объекты (неглубокое копирование).

****Глубокое копирование:****

1. JSON сериализация/десериализация:

```
```javascript
let original = { a: 1, b: { c: 3 } };
let copy = JSON.parse(JSON.stringify(original));
```
```

2. Использование сторонних библиотек, таких как Lodash, функция `_.cloneDeep`:

```
```javascript
let original = { a: 1, b: { c: 3 } };
let copy = _.cloneDeep(original);
```
```

Глубокое копирование создаёт полную копию объекта, включая все вложенные объекты. Однако, метод JSON имеет недостатки: он не справляется с копированием методов объекта, а также не может копировать объекты со сложными типами данных (такими как `Map`, `Set`, функции, символы, даты и т.д.).

Сравнение объектов в JavaScript можно выполнять на предмет "поверхностного" сходства (shallow comparison) или "глубокого" сходства (deep comparison).

****Поверхностное сравнение (shallow comparison):****

Это сравнение первого уровня свойств объектов:

```
```javascript
function shallowEqual(object1, object2) {
 const keys1 = Object.keys(object1);
 const keys2 = Object.keys(object2);

 if (keys1.length !== keys2.length) {
 return false;
 }

 for (let key of keys1) {
 if (object1[key] !== object2[key]) {
 return false;
 }
 }
}
```

```

 }

 return true;
}
...

```

Такое сравнение подходит, если уверены, что объекты не содержат вложенных объектов или их вложенность не требуется учитывать.

**\*\*Глубокое сравнение (deep comparison):\*\***

Сравнение всех уровней вложенных свойств и объектов. Для глубокого сравнения часто используют библиотеки, такие как Lodash с функцией `_.isEqual`:

```

```javascript
let isEqual = _.isEqual(object1, object2);
...

```

Или же можно написать собственную функцию для глубокого сравнения:

```

```javascript
function deepEqual(object1, object2) {
 const keys1 = Object.keys(object1);
 const keys2 = Object.keys(object2);

 if (keys1.length !== keys2.length) {
 return false;
 }

 for (let key of keys1) {
 const val1 = object1[key];
 const val2 = object2[key];
 const areObjects = isObject(val1) && isObject(val2);

 if (
 areObjects && !deepEqual(val1, val2) ||
 !areObjects && val1 !== val2
) {
 return false;
 }
 }

 return true;
}

```

```
function isObject(object) {
 return object !== null && typeof object === 'object';
}
...
```

Глубокое сравнение рекурсивно проверяет каждое свойство и подобъект на равенство. Однако это может быть ресурсоемкой операцией и значительно медленнее поверхностного сравнения, особенно для больших или сложных объектов.

## Как получить все ключи значения объекта

`Object.keys(obj)` – возвращает массив ключей.

`Object.values(obj)` – возвращает массив значений.

`Object.entries(obj)` – возвращает массив пар [ключ, значение].

`Object.keys(user) = ["name", "age"], Object.entries(user) = [ ["name", "John"], ["age", 30] ]`

## Ключевое слово **super**

Ключевое слово `super()` в `constructor` используется как функция, вызывающая родительский конструктор. Её необходимо вызвать до первого обращения к ключевому слову `this` в теле конструктора.

**`super()`** — это функция, используемая в классах для вызова конструктора родительского класса в JavaScript. Это позволяет дочернему классу получать доступ к свойствам и методам родительского класса.

Когда используется `super()` в конструкторе дочернего класса:

Это необходимо сделать, чтобы правильно инициализировать `this` в дочернем классе. Вы должны вызвать `super()` перед тем, как использовать `this` в конструкторе дочернего класса, в противном случае будет ошибка ссылки.

## Логические операторы ||, && !

ИЛИ || - возвращает первое истинное значение или последнее falsy, если такое значение не найдено.

И && - возвращает true, если оба аргумента истинны, а иначе – false:

!val – логическое НЕ возвращает false, если операнд может быть преобразован в true  
**falsy значения**

В JavaScript существует несколько значений, которые считаются "falsy", то есть они преобразуются в `false`, когда оцениваются в логическом контексте. Вот список этих значений:

?? (Оператор нулевого слияния) - возвращает правый операнд, если левый операнд равен null или undefined, иначе возвращает левый операнд.

1. `false` — логическое значение "ложь".
2. `0` — число ноль.
3. `-0` — отрицательный ноль.
4. `0n` — BigInt ноль.
5. `""` — пустая строка.
6. `null` — отсутствие какого-либо объектного значения.
7. `undefined` — значение не было присвоено.
8. `NaN` — "Не число" (Not a Number).

Эти значения ведут себя как `false` в условных операторах и других ситуациях, когда производится преобразование к логическому типу.

`a ||= b` Если `a` ложно, присваивает `a` значение `b`, Конвертирует `a` в логическое значение  
`a &&= b` логическое присваивания И - присвоит `a` значение `b` только в том случае, если `a` истинно.

`a ??= b` нулевое присваивание присвоит `a` значение `b`, если она содержит `null` или `undefined`

## Что делают операторы `spread` и `rest`, их отличия

Операторы `spread` и `rest` имеют одинаковый синтаксис (`"..."`). Разница состоит в том, что с помощью `spread` мы передаем или распространяем данные массива на другие данные, а с помощью `rest` — получаем все параметры функции и помещаем их в массив

## **Что такое функция, отличия от метода**

Функция – это тип объекта, который можно вызвать по имени, который не связан с каким то объектом. А метод всегда определяется внутри объекта и работает с его данными.

# Что такое static методы

Static методы в JavaScript — это методы, ассоциированные с классом, а не с экземпляром класса. Это означает, что они вызываются на самом классе, а не на объектах, созданных с помощью этого класса (экземплярах).

Такие методы полезны, когда вы хотите реализовать функциональность, которая относится к классу в целом, а не к отдельному объекту, например, фабричные методы (методы, создающие экземпляры) или вспомогательные функции, которые выполняют операции независимо от конкретных объектов.

Пример использования статических методов:

```
```javascript
class MyClass {
  static staticMethod() {
    console.log('Это статический метод.');
```



```
  }
}

MyClass.staticMethod(); // Вызов статического метода
// Выводит: "Это статический метод."
```
```

В данном примере `staticMethod` можно вызвать только через имя класса (`MyClass.staticMethod()`), а не через инстанс (экземпляр) этого класса. Вызов статического метода через экземпляр класса приведет к ошибке.



## Отличие синхронной функции от асинхронной

Синхронные функции блокируют выполнение операций прежде чем функция будет выполнена и

выполняются последовательно, асинхронные не блокируют.

Синхронность означает то, что строка 2

не может запуститься до тех пор, пока строка 1 не закончит своё выполнение

## Как работает блок try catch

Сначала выполняется код внутри блока **try**{}. Если в нём нет ошибок, то блок **catch(err)** игнорируется: выполнение доходит до конца try и потом далее, полностью

пропуская catch. если же в нём возникает ошибка, то выполнение try прерывается, и поток управления переходит в начало catch(err). **Finally** – выполнится в любом случае.

## SOLID - принципы

**S** - Принцип единственной ответственности (The Single Responsibility Principle) каждый класс должен выполнять лишь одну задачу и все сервисы класса должны быть направлены на обеспечение этой обязанности. Например компонент должен отрисовать JSX и нежелательно чтобы axios/fetch запросы делались прямо в компонентах, это нужно выносить в отдельные переиспользуемые классы, функции, хуки.

**O** - Принцип открытости/закрытости (The Open Closed Principle) программные сущности должны быть открыты для расширения, но закрыты для модификации. В общем, этот принцип имеет в виду следующее: у нас должна быть возможность добавлять новый функционал, не трогая существующий код класса.

**L** - Принцип подстановки Барбары Лисков (The Liskov Substitution Principle) объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения

правильности выполнения программы. Наследующий класс должен дополнять, а не изменять базовый. Для того чтобы следовать принципу необходимо в базовый (родительский) класс выносить только общую логику, характерную для классов наследников, которые будут ее реализовывать и, соответственно, можно будет базовый класс без проблем заменить на его класс-наследник.

**I** - Принцип разделения интерфейса (The Interface Segregation Principle) много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения. Классы не должны реализовывать и зависеть от методов, которые они не используют. Классы необходимо разделять на более специфические. В react – компоненты не должны зависеть от пропсов, которые они не юзают.

**D** - Принцип инверсии зависимостей (The Dependency Inversion Principle) классы должны зависеть от интерфейсов или абстрактных классов, а не от конкретных классов и функций. Высокоуровневые модули не должны зависеть от низкоуровневых модулей. Оба типа модулей должны зависеть от абстракций.

Чистая архитектура, предложенная Робертом С. Мартином (Uncle Bob), представляет собой методологию проектирования программного обеспечения с целью обеспечить легкость поддержки, тестирования и расширения программных систем. В ней определены следующие слои (или уровни), упорядоченные от более высоких уровней абстракции к нижним:

1. **\*\*Политики уровня предприятия (Entities):\*\***

- Этот слой содержит бизнес-правила приложения и бизнес-сущности. Это самый внутренний уровень и он представляет собой набор операций, которые описывают, как данные могут быть созданы, хранятся и изменяться. Он не должен зависеть от каких-либо других слоёв.

2. **\*\*Политики приложения (Use Cases):\*\***

- Этот уровень содержит бизнес-правила, специфичные для приложения. Он описывает, как данные должны протекать между бизнес-сущностями (Entities) и интерфейсами пользователя или внешними системами. Use cases координируют действия и направляют поток данных к и от сущностей, и обеспечивает, что все бизнес-правила соблюдаются.

3. **\*\*Интерфейсы адаптеров (Interface Adapters):\*\***

- Этот слой преобразует данные из формата, удобного для использования в use cases и entities, в формат, удобный для внешних агентов, таких как база данных или веб. Включает в себя контроллеры, представления и презентеры/трансформеры.

4. **\*\*Фреймворки и драйверы (Frameworks & Drivers):\*\***

- Самый внешний слой, который включает в себя все детали, зависящие от платформы: UI, инструменты для работы с базой данных, фреймворки, системы ввода-вывода и т.д. Этот слой должен изменяться только из-за изменений во внешних агентах, системах или библиотеках.

Важной концепцией чистой архитектуры является принцип зависимости: зависимости в коде должны направляться только от внешних слоёв к внутренним. То есть, более высокоуровневые слои не должны зависеть от деталей, предоставляемых более низкоуровневыми слоями, а должны взаимодействовать с ними через абстракции. Это помогает в создании системы, которая является гибкой, тестируемой и легко поддерживаемой.

## dry kiss yagni

Это 3 принципа, чтобы создать более эффективный, понятный и поддерживаемый код.

- **DRY** - Don't Repeat Yourself - избегать повторения блоков кода

- **KISS** - Keep It Simple, Stupid - поддерживать простоту, читаемость и интуитивную ясность кода

- **YAGNI** - You Aren't Gonna Need It - не использовать функциональность, в которой нет необходимости.

1. DRY (Don't Repeat Yourself): Означает, что каждая часть программы или системы должна иметь единственный источник истины. Если возникает необходимость в изменении логики или функциональности, это должно быть сделано только в одном месте. Избегайте повторения кода, так как это может привести к дублированию ошибок, сложности поддержки кода и излишней сложности.

2. KISS (Keep It Simple, Stupid): Этот принцип призывает к простоте и интуитивной ясности кода и архитектуры. Сложность может привести к ошибкам и затратам времени на поддержку. Поэтому следует избегать излишней сложности и стремиться к простоте в решении задачи или реализации функции.

3. YAGNI (You Aren't Gonna Need It): Этот принцип говорит о том, что в программировании необходимо избегать реализации функциональности, которая не требуется на данный момент. Не следует предполагать будущие потребности и добавлять функциональность, которая не является непосредственным требованием. Это помогает избежать излишней сложности и потери времени на реализацию ненужных функций.

Применение этих принципов поможет создать более эффективный, понятный и поддерживаемый код, а также избежать излишней сложности в разработке и управлении проектом.

# Babel зачем нужен

**Babel** — это компилятор, который преобразует современный JavaScript для запуска в старых браузерах. Он также выполняет преобразование JSX синтаксиса в js.

## Методы массивов, какие из них мутирующие, а какие нет

Методы массивов в JavaScript можно разделить на мутирующие (изменяющие исходный массив) и немутрующие (не изменяющие исходный массив).

**### Мутирующие** методы (изменяют исходный массив):

```
- push() - добавляет элементы в конец массива.
- pop() - удаляет последний элемент из массива.
- shift() - удаляет первый элемент из массива.
- unshift() - добавляет элементы в начало массива.
- reverse() - переворачивает массив на месте.
- sort() - сортирует элементы массива на месте.
- splice() - добавляет/удаляет/заменяет элементы в массиве.
- fill() - заполняет массив статическими значениями.
- copyWithin() - копирует последовательность элементов массива внутри массива.
```

**### Немутрующие** методы (не изменяют исходный массив):

```
- concat() - объединяет два или более массивов.
```

- **`map()`** - создаёт новый массив с результатами вызова функции для каждого элемента.
- **`filter()`** - создаёт новый массив со всеми элементами, прошедшими проверку.
- **`slice()`** - возвращает новый массив, являющийся копией части исходного массива.
- **`join()`** - объединяет все элементы массива в строку.
- **`indexOf()`** / **`lastIndexOf()`** - возвращают первый/последний индекс, по которому данный элемент может быть найден в массиве.
- **`find()`** / **`findIndex()`** - возвращают значение первого элемента, удовлетворяющего условию.
- **`includes()`** - определяет, содержит ли массив определённый элемент.
- **`every()`** / **`some()`** - проверяют соответствие всех/некоторых элементов массива заданному условию.
- **`reduce()`** / **`reduceRight()`** - применяют функцию к аккумулятору и каждому значению массива (слева-направо/справа-налево), возвращая одно конечное значение.

Понимание разницы между мутлирующими и немутлирующими методами значительно важно, особенно при работе с иммутабельными структурами данных, как это делается в функциональном программировании и в некоторых фреймворках, например, Redux.

## Для чего нужны `map` / `reduce` / `filter` / `concat`, циклы для перебора массивов

`map`, `reduce`, `filter`, `concat` и циклы — это инструменты в JavaScript для работы с массивами, каждый из которых используется для разных целей:

1. **`map()`**: Применяет функцию к каждому элементу массива и возвращает новый массив с результатами этих применений, не изменяя исходный массив. Используется для преобразования массива.

```
javascript
let numbers = [1, 2, 3];
let squares = numbers.map(x => x * x); // [1, 4, 9]
```

2. **`reduce()`**: Применяет функцию-редуктор к каждому элементу массива, возвращая одно результирующее значение. Используется для свёртки массива.

```
javascript
let sum = numbers.reduce((total, value) => total + value, 0); // 6
...

```

3. **filter()**: Создаёт новый массив со всеми элементами, которые прошли проверку, реализованную предоставленной функцией. Используется для выборки определённых элементов массива.

```
```javascript
let evens = numbers.filter(x => x % 2 === 0); // [2]
```
```

4. **concat()**: Объединяет два или более массивов в один новый массив, не изменяя исходные массивы.

```
```javascript
let moreNumbers = [4, 5, 6];
let allNumbers = numbers.concat(moreNumbers); // [1, 2, 3, 4, 5, 6]
```
```

**Циклы для перебора массивов**, такие как `for`, `for...of`, `forEach()`, также используются для итерации по массивам:

- **for**: Традиционный цикл, который даёт больший контроль над процессом итерации.

```
```javascript
for (let i = 0; i < numbers.length; i++) {
    console.log(numbers[i]);
}
```
```

- **for...of**: Современный способ итерации по значениям итерируемых объектов, включая массивы.

```
```javascript
for (let value of numbers) {
    console.log(value);
}
```
```

- **forEach()**: Выполняет функцию один раз для каждого элемента в массиве.

```
```javascript
numbers.forEach(value => console.log(value));
```
```

Цикл `for of` в JavaScript позволяет перебирать итерируемые объекты, такие как массивы, множества, Map, строки. НЕ ПОДХОДИТ ДЛЯ ПЕРЕБОРА ОБЪЕКТОВ. `for (let a of arr) { }`

Для перебора всех свойств из объекта используется цикл по свойствам `for in`.

Единственный способ перебрать все свойства объекта. Подходит для объектов и массивов. `for (key in obj) { }`

## Как сделать объект неизменяемым, методы `Object.freeze` `Object.seal`

Чтобы сделать объект в JavaScript неизменяемым, можно использовать методы `Object.freeze()` или `Object.seal()`.

**`Object.freeze()`**: Замораживает объект, делая его неизменяемым. Свойства не могут быть добавлены, удалены или изменены, а объекты в глубоких структурах остаются неизменными.

```
```javascript
const obj = { prop: 42 };
Object.freeze(obj);

obj.prop = 33; // Не будет выполнено, так как объект заморожен
delete obj.prop; // Не будет выполнено
obj.newProp = 17; // Не будет выполнено

console.log(obj.prop); // 42
```
```

**`Object.seal()`**: Запечатывает объект. Можно изменять существующие свойства, но нельзя добавлять новые свойства или удалять старые.

```
```javascript
```

```
const obj = { prop: 42 };
Object.seal(obj);

obj.prop = 33; // Будет выполнено, свойство изменится
delete obj.prop; // Не будет выполнено
obj.newProp = 17; // Не будет выполнено

console.log(obj.prop); // 33
````
```

И `Object.freeze()`, и `Object.seal()` делают свойства объекта ненастраиваемыми, но `Object.seal()` позволяет изменять значения существующих свойств на объекте. В обоих случаях после применения методов нельзя добавить или удалить свойства объекта.

`(Object.preventExtensions())`, а все его собственные свойства – ненастраиваемыми. Это предотвращает добавление новых свойств и удаление существующих. Можно изменять существующие свойства

## Как сделать свойства объекта `readonly` или добавить новое `Object.defineProperty`

`Object.defineProperty(obj, propertyName, descriptor)` определяет новое или изменяет существующее свойство в объекте, возвращая этот объект. **Descriptor** – объект со свойствами `configurable` (можем менять), `enumerable` (можем ли итерироваться), `writable` (можем ли присвоить значение или же свойство **readonly** - **readonly**

обозначает свойство, доступное только для чтения.), `value`.

## Как создать объект с указанным прототипом `Object.create`

`Object.create(proto, properties)` - создаёт новый объект с указанным прототипом и свойствами, позволяет конфигурировать у свойств `configurable`, `enumerable`, `writable`, `value`.



# Как проверить наличие свойства в объекте, `in` vs `hasOwnProperty`

Используйте `in`, если хотите проверить существование свойства в цепочке прототипов объекта, и `hasOwnProperty`, если нужно удостовериться, что свойство принадлежит именно данному объекту и не было унаследовано

- **Оператор `in`**: Проверяет, существует ли свойство в объекте или его прототипе.

```
```javascript
if ('property' in object) {
    // свойство существует в объекте или его прототипе
}
```
```

- **Метод `hasOwnProperty`**: Проверяет, является ли свойство собственным (то есть непосредственно принадлежащим) свойством объекта, не учитывая прототип.

## Что такое SPA

**SPA** — это приложение, использующее единственный HTML-документ как оболочку и организующий взаимодействие с пользователем через динамически подгружаемые HTML, CSS, JavaScript.

SPA (Single Page Application) — это тип веб-приложения или веб-сайта, который взаимодействует с пользователем, динамически переписывая текущую страницу, а не загружая новые страницы с сервера. Это создает впечатление работы с настольным приложением, так как в процессе работы страница не перезагружается полностью, и это может значительно увеличить скорость и удобство интерфейса.

В SPA весь необходимый HTML, JavaScript и CSS обычно загружаются один раз, или же соответствующие ресурсы динамически подгружаются и добавляются к странице по мере необходимости, часто в ответ на действия пользователя. Обновление данных на странице обычно происходит через AJAX запросы к серверу и обмен данными в формате JSON, что уменьшает время отклика и повышает скорость работы приложения.

Примеры популярных фреймворков и библиотек, используемых для создания SPA:

- React (созданный Facebook)
- Angular (разработанный Google)
- Vue.js
- Ember.js

- Svelte

SPA имеет как преимущества (быстрая интерактивность, унификация с мобильными приложениями, лучшие возможности для UX/UI), так и недостатки (проблемы с SEO, необходимость управления состоянием, сложность в поддержке старых браузеров, первая загрузка может занимать больше времени).

## Принципы ООП и классов

**Объектно-ориентированное программирование.** В ООП программы состоят из взаимодействующих между собой объектов. Объекты в свою очередь представляют собой сущности, обладающие свойствами и методами.

Свойства - данные объекта, а методы - действия, которые можно выполнять с объектом.

object является ссылочным типом. Это означает то, что работа со всеми объектами в JavaScript ведётся по ссылке. Ни одна переменная в JavaScript не содержит объектов. Все они лишь ссылаются на них в памяти.

Данные в объектах хранятся в виде пар “ключ: значение”, где ключ - это string (или symbol), а значение может являться чем угодно (в том числе другим объектом).

В целом, говоря про объектно-ориентированное программирование в js, кроме самих объектов нужно подчеркнуть такие важные элементы данной парадигмы, как:

**Классы:** С ES6, в JavaScript добавлена поддержка классов — синтаксический сахар над прототипным наследованием. Классы определяют конструктор для объектов и могут содержать описание методов и свойств.

**Прототипы:** Все объекты в JavaScript имеют прототипы, от которых они наследуют свойства и методы. Прототипное наследование позволяет объекту использовать методы и свойства другого объекта.

**Конструкторы:** Функции, вызываемые с помощью оператора new, для создания новых экземпляров объекта. Конструкторы устанавливают начальные свойства и могут вызывать методы при создании нового объекта.

**Наследование:** Способность одного класса унаследовать свойства и методы другого. Реализуется через прототипы или ключевое слово extends в классах.

**Инкапсуляция:** Практика сокрытия внутренних деталей компонентов объекта, чтобы предотвратить доступ к ним извне и упростить интерфейс.

**Полиморфизм:** Возможность объектов использовать методы базового (родительского) класса так, как если бы это были методы производного (дочернего) класса, что позволяет индивидуально для каждого класса определить реализацию этих методов.

**Абстракция:** Способность создавать простой интерфейс взаимодействия с объектом, скрывая его сложную внутреннюю структуру.

Методы: Функции, связанные с объектами, которые могут использовать их контекст (this).

## Конструкторы, функции без замыканий

Constructor это специальный метод, служащий для инициализации классов, и установки начальных значений экземпляра. Функции без замыкания – классы и функции конструкторы, которые вызываются с new

## Как работает ключевое слово new

Ключевое слово **new** - создает экземпляр объекта, при этом вызывая конструктор. При вызове new создаётся новый пустой объект, и он присваивается this. Выполняется тело функции. Обычно оно както модифицирует this. Далее возвращается значение this.

## Зачем нужны async defer и их отличия

Атрибут **defer** сообщает браузеру, что он должен продолжать обрабатывать страницу и загружать скрипт в фоновом режиме. выполняться они будут в указанном порядке и после того как DOM дерево будет построено. УЧИТЫВАЕТСЯ порядок загрузки скриптов

**Async** указывает браузеру, что скрипт может быть выполнен асинхронно. Парсеру HTML нет необходимости останавливаться, когда он достигает тега <script> для загрузки и выполнения. Выполнение может произойти после того, как скрипт будет получен параллельно с разбором документа. НЕ УЧИТЫВАЕТСЯ порядок загрузки скриптов

У async и defer есть кое-что общее: они не блокируют отрисовку страницы. Их отличия:

|       | Порядок                                                    | DOMContentLoaded                                                                                                                                                                           |
|-------|------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| async | Порядок загрузки (кто загрузится первым, тот и сработает). | Не имеет значения. Может загрузиться и выполниться до того, как страница полностью загрузится. Такое случается, если скрипты маленькие или хранятся в кеше, а документ достаточно большой. |
| defer | Порядок документа (как расположены в документе).           | Выполняется после того, как документ загружен и обработан (ждёт), непосредственно перед DOMContentLoaded.                                                                                  |

## Где лучше подключать script и почему

Скрипты обычно подключают либо в `head` с атрибутом `defer`, либо перед закрывающим тегом `</body>`.

Скрипты с `defer` лучше размещать в секции `head` HTML-документа. Они загружаются асинхронно и выполняются после того, как весь HTML будет разобран, но до события `DOMContentLoaded`.

```

<<html
<head>
 <script src="script.js" defer></script>
</head>
...

```

Скрипты с `async` также можно размещать в `head`, так как они загружаются асинхронно и выполняются независимо от остальной части страницы, как только окажутся загруженными.

**<script> <script> Скрипты лучше подключать в конце HTML. Если подключать JavaScript в начале то body не будет отрисовываться, пока не выполнится подключаемый js скрипт. потому рекомендуется подключать js в конце.**

Также при использовании `async` важно, чтобы скрипт не зависел от DOM или других скриптов из-за неопределенного времени выполнения.

## Что такое объект Proxy

Объект **Proxy** позволяет обернуть другой объект и может перехватывать и обрабатывать разные действия с ним, например чтение/запись свойств и другие **new Proxy(target, handler)**

## Что такое функция высшего порядка

Функции высшего порядка — это функции, которые либо принимают другие функции в виде аргументов, и/или возвращают их (**map**, **filter**).

Функции высшего порядка часто встречаются в JavaScript, вот несколько примеров:

1. **Методы массивов `map`, `filter`, `reduce`**: Это встроенные функции высшего порядка в JavaScript.

```
```javascript
// map принимает функцию и возвращает новый массив
const numbers = [1, 2, 3];
const doubled = numbers.map(num => num * 2);

// filter принимает функцию и возвращает новый массив, фильтруя
элементы
const evens = numbers.filter(num => num % 2 === 0);

// reduce принимает функцию и возвращает единственное значение
const sum = numbers.reduce((total, num) => total + num, 0);
```
```

2. **Функции, возвращающие функции**: Например, фабрика функций.

```
```javascript
function greaterThan(n) {
  return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11)); // true
```
```

3. **Функции, принимающие функции как аргументы**: Например, функции обратного вызова (callbacks).

```
```javascript
function withLogging(func) {
  console.log('Starting function execution');
  let result = func();
  console.log('Finished function execution');
  return result;
}

function sayHello() {
  console.log('Hello!');
}

withLogging(sayHello); // Логирует выполнение функции sayHello
```
```

Эти примеры демонстрируют суть функций высшего порядка: они либо принимают другие функции как аргументы, либо возвращают функции, либо делают и то, и другое.

## Что такое каррирование

Каррирование (currying) — это процесс преобразования функции с множеством аргументов в последовательность функций, каждая из которых принимает один аргумент. Главная цель каррирования — это сделать функцию более гибкой, позволяя зафиксировать некоторые аргументы и создать новую функцию, готовую принять оставшиеся аргументы позже.

**\*\*Пример каррирования:\*\***

```
```javascript
function curryAdd(a) {
  return function(b) {
    return a + b;
  };
}
```

```
// Использование каррированной функции
let addFive = curryAdd(5); // Фиксируем первый аргумент, а, как 5
console.log(addFive(3)); // Выводит 8, так как b принимает значение 3
```
```

В этом примере `curryAdd` является каррированной функцией, которая сначала принимает один аргумент, затем возвращает функцию, которая принимает следующий аргумент и в конце производит вычисление.

## Что такое compose и функция первого класса

Функция `compose` в функциональном программировании позволяет соединить несколько функций в одну. Результат выполнения одной функции передается как аргумент в следующую. При комбинировании функций они выполняются справа налево (от последней к первой).

**\*\*Пример функции `compose`:\*\***

```
```javascript
function double(x) {
  return x * 2;
}

function increment(x) {
  return x + 1;
}

function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}

let incrementAndDouble = compose(double, increment);
```

```
console.log(incrementAndDouble(3)); // Сначала прибавляет 1, затем удваивает:  
(3+1)*2 = 8  
'''
```

В данном примере, функция `compose` создает новую функцию, которая сначала применяет `increment` к аргументу, а затем результат передает в функцию `double`.

Функция первого класса - это функция, которая была создана с целью передачи другим функциям.

Она, не имеет побочных эффектов и не предназначена для прямого вызова, а скорее для использования «другими функциями»

Благодаря чему мы можем итерироваться по объектам, метод `symbol.iterator`

String, Array, Map и Set являются итерируемыми, потому что их прототипы реализуют метод `Symbol.iterator`. Он позволяет итерироваться по какой-то структуре данных.

Итераторы предоставляют метод `next()`, который возвращает объект с двумя свойствами: `{ value, done }`

Ключ `Symbol.iterator` в объекте позволяет определить итератор для этого объекта.

Итератор предоставляет способ обхода элементов объекта по одному за раз.

Итератор - это объект, который умеет последовательно обращаться к элементам по одному за раз, при этом отслеживая свое текущее положение внутри этой последовательности. У него есть метод `next`, который возвращает объект со свойствами `done` и `value`

Итераторы используются в JavaScript для перебора элементов коллекций, таких как массивы или строки, и могут быть созданы для любого объекта, реализующего интерфейс итератора.

Использование ключа `Symbol.iterator` позволяет объекту быть итерируемым, то есть объект может быть использован в циклах `for...of` или использовать другие методы и операторы, которые работают с итерируемыми объектами (например, оператор распространения или деструктуризация).

Какие бывают структуры данных

- **Стэк (Stack)**: Структура данных, работающая по принципу LIFO (Last In, First Out), то есть последний пришёл — первый вышел. Элементы добавляются (push) и удаляются (pop) с одного конца стека.
- **Очередь (Queue)**: Структура данных, работающая по принципу FIFO (First In, First Out), то есть первый пришёл — первый вышел. Элементы добавляются с одного конца и удаляются с другого.
- **Set**: Коллекция уникальных значений, не содержащая дубликатов.
- **Map**: Коллекция пар ключ-значение, где каждый ключ уникален.
- **Хеш-таблица (Hash Table)**: Структура данных, использующая хеш-функцию для вычисления индекса в массиве, куда будет помещён или найден элемент.
- **Дерево (Tree)**: Иерархическая структура данных, где каждый "узел" содержит данные и ссылки на "дочерние" узлы (ноль или больше).
- **Графы (Graphs)**: Набор узлов, связанных рёбрами, могут быть направленными (со стрелками) или ненаправленными (без стрелок). Узлы иногда называют вершинами, а рёбра могут иметь вес или стоимость.
- **Связный список (Linked List)**: Все элементы (узлы) содержат данные и ссылку на следующий узел. В двусвязном списке каждый узел содержит ссылку и на предыдущий узел.

Что такое service worker

Service Worker - это скрипт, который браузер запускает в отдельном потоке, отдельно от страницы, у них нет доступа к DOM, они являются не блокирующими и асинхронными. Имеют возможность перехватывать и обрабатывать сетевые запросы, работают только по HTTPS. Также могут кешировать статические ресурсы, для того чтобы приложение корректно работало в режиме офлайн, используются в PWA, могут юзаться для Push-уведомления

Что такое Web Worker

Web Worker - это отдельные потоки JS, которые работают параллельно с основным потоком JS и доступны только веб-приложению. Они используются для выполнения тяжелых задач, чтобы избежать блокировки цикла событий и страницы. Не могут манипулировать DOM

Что такое ES module

ES Модуль – это просто файл. Модули могут загружать друг друга и использовать директивы `export` и `import`, чтобы обмениваться функциональностью, вызывать функции одного модуля из другого. `export` отмечает переменные, которые должны быть доступны вне текущего модуля. `import` позволяет импортировать функциональность из других модулей.

Метод valueOf

Метод **valueOf ()** возвращает примитивное значение объекта. Если объект не имеет примитивного значения,

valueOf возвращает сам объект, который отображается как: `[object Object]`

Метод `valueOf()` в JavaScript возвращает примитивное значение указанного объекта. **По умолчанию метод `valueOf()` унаследован каждым объектом от `Object.prototype`** и может быть переопределён в пользовательских объектах для возвращения значения, которое считается естественным представлением объекта.

Когда операции в JavaScript требуют примитивного значения (например, математические операции), автоматически вызывается метод `valueOf()`. Если метод возвращает значение, которое не является объектом, операция использует это примитивное значение. В противном случае, если `valueOf()` возвращает объект, то операция пытается вызвать метод `toString()`.

****Пример:****

```
```javascript
function MyNumberType(n) {
 this.number = n;
}
```

```
MyNumberType.prototype.valueOf = function() {
 return this.number;
};

let myNumberInstance = new MyNumberType(4);
console.log(myNumberInstance + 3); // Выведет 7, так как valueOf
возвращает примитивное значение
````
```

Таким образом, `valueOf` используется для получения примитивной версии объекта, когда таково значение необходимо.

Метод **Symbol.toPrimitive**

С помощью свойства `Symbol.toPrimitive`, объект может быть приведён к примитивному типу. Функция вызывается с аргументом `hint`, который передаёт желаемый тип примитива. Значением аргумента `hint` может быть `"number"`, `"string"`, и `"default"`. Если `hint` равен `"string"` - происходит попытка вызвать `obj.toString`, если `default` или `number` – `valueOf`

JS особенности

JavaScript имеет следующие особенности:

- отсутствие строгой типизации данных; динамически типизируемый
- полная интеграция с HTML-страницами и CSS;
- обработка действий и событий браузера, взаимодействие с DOM.
- Является однопоточным и синхронным - это означает, что он может выполнять только одну задачу за раз и у него есть только один стек вызовов.

```
const changeBalance = () => {  
  let balance = 0;  
  
  return (num: number) => {  
    balance += num;  
    console.log("balance now = " + balance);  
  };  
};  
  
// ВЫЗЫВАЕМ 1 ФУНКЦИЮ => ОНА ВЕРНЕТ ДРУГУЮ => ВЫЗЫВАЕМ ДРУГУЮ И  
// ПЕРЕДАЕМ ЕЙ АРГУМЕНТ  
changeBalance()(50);  
  
// doChange это функция, которую вернет вызов changeBalance. (num: number) => void  
const doChange = changeBalance();  
  
doChange(100); // balance = 100  
doChange(200); // balance = 300  
doChange(500); // balance = 800
```

Рекурсия это

Рекурсия - это когда функция вызывает саму себя до тех пор, пока не сработает какое либо условия для прекращения.

Да, рекурсия — это техника в программировании, при которой функция вызывает саму себя для решения подзадачи.

Пример рекурсивной функции (факториал числа):

```
```javascript
function factorial(n) {
 if (n === 0 || n === 1) { // условие прекращения
 return 1;
 } else {
 return n * factorial(n - 1); // рекурсивный вызов
 }
}

console.log(factorial(5)); // 120
```

В этом примере функция `factorial` вызывает саму себя с уменьшенным на единицу аргументом, пока `n` не станет 0 или 1, что является базовым случаем для прекращения рекурсии.

## IIFE это

**IIFE** (Immediately Invoked Function Expression) это JavaScript функция, которая выполняется сразу же после того, как она была определена.

Да, IIFE (немедленно вызываемое функциональное выражение) — это функция в JavaScript, которая выполняется сразу после того, как она была определена.

```
javascript
(function() {
 var localVar = 'Я существую только внутри IIFE';
 console.log(localVar); // Выводит: "Я существую только внутри IIFE"
})();
```

```
// localVar не доступна в глобальной области видимости
console.log(typeof localVar); // Выводит: "undefined"
```

IIFE обычно используется для создания новой области видимости и изоляции переменных, чтобы избежать их конфликта с другими частями кода.

## Что может заблокировать поток в js

Операции, которые могут заблокировать поток (main thread) в JavaScript, так называемые "блокирующие операции", включают:

1. Тяжёлые вычисления: длительные циклы и рекурсивные функции.
2. Синхронные запросы к серверу: `XMLHttpRequest` синхронный или использование `fetch` без `await`.
3. Долгие обработчики событий.
4. Синхронное чтение из локального хранилища.
5. Использование `alert`, `prompt`, `confirm` (модальные окна браузера).

Для предотвращения блокировок используют асинхронное программирование, в том числе `promises`, `async/await`, `Web Workers`.

## Проваливание промисов это

Проваливание промисов - когда вы передаете в `then()` что-то отличное от функции (например, промис), это интерпретируется как `then(null)` и в следующий по цепочке промис «проваливается» результат предыдущего.

```
Promise.resolve('foo').then(Promise.resolve('bar')).then((result)=> console.log(result));
```

## Что такое boxing unboxing

**Boxing** – это когда примитивное значение (string, number) преобразуется в соответствующий объект-обертку `new Number(7)`.

**Unboxing** - когда значение извлекается из объекта-обертки и преобразуется обратно в примитивный тип данных `new String("word").valueOf()`;

## Отличия примитивов от ссылочных типов данных

Особенность **примитивных** типов данных заключается в том, что они иммутабельны и хранятся в памяти по значению. В отличие от **ссылочных** типов, которые передаются по ссылке на место в

```
let person = { ez300k: true };
let array = [person];

person = null;
console.log(array); // [{ ez300k: true }]
```

памяти.

Объект person не будет удаляться из памяти сборщиком мусора, так как на него ещё

существует ссылка в коде.  
(array)

## Что такое сборщик мусора

При объявлении переменных и функций в js выделяется память. Сборщик мусора нужен для освобождения памяти, которая больше не используется. Его работа заключается в определении и удалении объектов, которые больше не нужны приложению. Для этого сборщик мусора периодически выполняет проход по всем объектам в куче и проверяет, есть ли на них ссылки.

## requestAnimationFrame что это

**requestAnimationFrame** - позволяет выполнять переданный колбэк, перед обновлением интерфейса и стадией layout в браузере. Нужен для задач, которые нужно выполнить перед обновлением следующего кадра. Его плюс в том, что он не будет вызываться чаще или реже, чем браузер выполняет layout.

## requestIdleCallback

**requestIdleCallback** принимает колбэк, который будет вызываться в период простоя браузера, когда цикл событий будет в режиме ожидания.

## Зачем нужен instanceof

Как работает **usePrevious** – обновляется value => вызывается рендер => после рендера вызывается эффект => ref.current = value присваивается новое значение => обновление ref не вызывает рендер

Оператор **instanceof** позволяет проверить, принадлежит ли объект указанному классу, с учётом наследования.

## Что такое микрофронтенд, webpack module federation

**Микрофронтенды** — это подход, при котором можно разделить приложение на отдельные модули и в рантайме объединять в единое приложение. Плюсы: улучшение масштабируемости, сервисы не зависят друг от друга, ошибка в 1 сервисе не крашнет другие. Минусы: более сложная конфигурация, проблемы с ci-cd, с шарингом состояния.

**host** — сборка, которая будет подгружать другие модули

**remote** — сборка, которая будет загружена в host

**exposed** - модули, которые экспортируются из `remote` и будут доступны для импорта  
**shared**: зависимости, которые будут совместно использоваться между различными приложениями  
**singleton**: если установлено значение `true`, то общий модуль будет загружаться только один раз, даже если он используется в нескольких приложениях.  
**requiredVersion**: версия общего модуля, которая должна быть загружена  
**eager**: если установлено значение `true`, то общий модуль будет загружен сразу при инициализации приложения, а не по требованию.

**Unit тесты** нужны для тестирования отдельных модулей, компонентов.

**E2e Интеграционные** тесты проверяют несколько частей кода на правильность совместной работы.

## Что такое функции генераторы

**Генераторы** -- вид функций, которые могут приостанавливать своё выполнение, возвращать промежуточный результат и далее возобновлять его позже, в произвольный момент времени. Генератор является итерируемым объектом (через `for...of`). Т.к реализует метод `Symbol.iterator`.

С генераторами не нужно беспокоиться о создании бесконечного цикла – вы можете остановить и возобновить выполнение по требованию, `next` последовательно возвращает значения в цикле.

Ключевое слово `yield` вызывает остановку функции-генератора и возвращает текущее значение выражения. `yield*` - применима только к другому генератору и делегирует ему выполнение.

Метод `next()` – возобновляет выполнение кода и возвращает следующий элемент последовательности.

С использованием `async/await` код выглядит более предсказуемо и читабельно, мы можем обрабатывать ошибки в `try/catch`. Генератор возвращает объект итератор, а `async`-функция промис.

В генераторах обычно требуется использование какой-либо внешней вспомогательной функции для обработки промисов, она запускается синхронно, последовательно выполняя свой код до тех пор, пока не дойдет до `yield`. Полезны, когда вам нужен более детальный контроль над потоком выполнения.

## Что такое гидрация

**Гидрация** – это процесс, при котором серверно-сгенерированный HTML, прикрепляется и "гидрируется" на клиентской стороне, то есть превращается в действительные компоненты React. Процесс гидрации в SSR приложениях подразумевает, что сервер уже выполнил рендеринг компонентов React в виде HTML-кода. Когда клиент получает этот HTML-код, React берет на себя задачу гидрации, то есть он прикрепляет HTML-код к соответствующим компонентам на



клиентской стороне и запускает клиентскую логику. Это позволяет клиенту продолжить интеракцию с приложением, отображать изменения без необходимости повторной загрузки всей страницы.

## Отличия type от interface (type, interface)

В отличие от interface, type может задать алиас для любой разновидности типов, включая примитивы. А interface позволяет нам описать поля и типы, которые должны быть реализованы в классе или объекте. Также интерфейс может расширять с помощью ключевого слова extends (даже от type). Если дважды объявить интерфейс с одним и тем же именем, то интерфейсы объединятся.

## Что такое Generics (Дженерики)

**Generic** - позволяют создавать типы, способные работать с различными типами, а не только с одним строго определенным типом. Т.е. мы определяем типы в момент использования. Дженерики позволяют нам писать переиспользуемые, гибкие функции, классы и типы, сохраняя при этом строгую типизацию. Generic можно extend от каких либо interface и class

## Что такое Union

С помощью конструкции Union | вы можете создать новую структуру (тип), которая будет являться объединением нескольких указанных типов.

## Что такое Intersection тип пересечения

Тип пересечения &. Эта конструкция позволяет комбинировать несколько типов в один. Это означает, что если вы создадите два разных объекта и захотите написать одну общую функцию для них, то вам потребуется что-то универсальное, что может принимать два разных типа и возвращать их комбинацию.

## Какие знаешь Utility types - record, pick, omit, required, partial

Record - создает тип с набором полей переданных 1 аргументом, типом которых будет 2й аргумент `const num : Record<keyof IUser, number> = {prop: 777 },`

`Pick<T, 'name'>` - создает тип, выбирая поля из типа, переданного 1 аргументом `Omit<T, 'name'>` - создает тип, исключая поля из типа, переданного 1 аргументом `Exclude<'a' | 'b', 'a'>` - создает тип, исключает из `union` типа все типы, переданные 2 аргументом. Простыми словами из типа `T` будут исключены ключи присущие также и типу `U`.

`Extract<'a' | 'b', 'a'>` - создает тип выбирая типы из двух типов, которые присутствуют в обоих / Создает тип путем извлечения из `union T` всех свойств, которые могут быть присвоены `U`.

`Readonly` - делает все поля типа доступными только для чтения `Partial` - делает все поля типа необязательными

`Required` - делает все поля типа обязательными

`ReturnType` - позволяет извлечь тип, который возвращает функция - `ReturnType<typeof getStore>`

`Parameters<typeof getUser>` - Получает тип параметров функции в виде кортежа `[id: number, name: string]`

`Awaited` - тип которым можно развернуть промис и вернуть тип.

## any vs unknown

Как `any`, так и типу `unknown` можно присвоить любое значение, но тип `unknown` не может быть присвоен почти никакому другому без утверждения типа, в отличии от `any`. Также мы не можем обращаться к свойствам объекта типа `unknown`, вызывать их или конструировать. `Unknown` безопаснее.

## Тип never

`never` — это примитивный тип, это признак для значений, которых никогда не будет. Или, признак для функций, которые никогда не вернут значения, например по причине бесконечного цикла, или по причине выбрасывания ошибки `throw Error`. Также пересечение двух несовместимых типов (`string & number`). Типу `never` мы не можем присвоить никакие значения, в том числе типа `any`.

Тип `**void**` используется для функций, которые не возвращают значения.

Тип `**never**` используется для функций, которые никогда не завершаются (например, функция, которая всегда выбрасывает исключение).

Тип `**unknown**` используется для значений, о типе которых мы не знаем на момент компиляции. Он похож на тип `any`, но более безопасен, так как `TypeScript` не позволяет

присваивать значения типа `unknown` переменным других типов без явного приведения тип

## Что такое Type Guard

TypeGuard это функции которые возвращают Boolean значение и позволяют сузить тип переменной или определить к какому типу относится эта переменная, с помощью `typeof`, `instanceof`, `in`.

это механизм в TypeScript, позволяющий выполнить проверку типов во время выполнения кода. Он позволяет более точно определять типы переменных и использовать функциональные блоки кода в зависимости от типов.

## Что такое утверждение типов

Type Assertion, утверждение типов с использованием `as` чтобы сообщить TypeScript, каким должен быть тип, когда он выводит его неправильно. `as const` делает поля `readonly` и тип полей ограничен значением;

Литеральные типы - более конкретные типы строк, чисел, лог. значений.

## Как работает keyof / typeof

`keyof` Type возвращает тип, который ограничен ключами типа; `typeof` возвращает тип данных.

## Mapped types для чего

**Mapped types** - используется для перебора всех ключей типа и создает новый тип `{ [key in keyof T]: T[key] }`

## Что такое Implements

**Implements** в классе используется для того чтобы проверить, удовлетворяет ли класс определенному интерфейсу, можно делать `implement` нескольких интерфейсов

Методы - это функции, определённые внутри объектов, классов.

## Как работают декораторы

Декораторы могут существовать только в классе. Декорировать в TypeScript можно классы, методы, параметры метода, accessors и свойства.

Паттерн декоратора позволяет обернуть объект, класс в некоторую функцию-декоратор, которая модифицирует его поведение.

Декоратор Класа объявляется перед объявлением класса. Декоратор класса применяется к конструктору класса и может быть использован для наблюдения, модифицирования или замещения определения класса. В декоратор класса передается функция-конструктор декорируемого класса. Доступ к конструктору класса позволяет, например, полностью переопределить его, или изменить его прототип.

Декораторы исполняются в обратном порядке поэтому здесь будет 170000

Декоратор Метода может переопределить метод класса, дополнять поведение метода его, или изменить его прототип

## Для чего нужно ключевое слово infer

Ключевое слово infer позволяет нам использовать условный оператор для вывода неизвестного типа.

## Абстрактные классы, перегрузки функций

Абстрактные классы определяются с ключевым словом abstract. Они похожи на обычные классы, но у нас нет возможности создать экземпляр абстрактного класса с помощью new, но можем от них наследоваться.

В Абстрактных классах можно определить абстрактные методы и свойства, это методы, у которых объявлена сигнатура, но нет тела метода. При наследовании классы обязаны реализовать все абстрактные методы, или будет ошибка.

namespace - пространство имен позволяют избежать конфликтов с именами других объектов. Чтобы получить доступ к данным внутри namespace нужно экспортировать значения, к которым мы хотим получить доступ. Это обертка позволяющая инкапсулировать какую-либо логику

Да, поддерживает. Перегрузка подразумевает под собой способ определения нескольких версий работы функции с целью поддержания читабельности кода. TypeScript будет выбирать правильную версию на основе передаваемых аргументов.

Перегрузки функций - это способ сообщить TypeScript, что эта функция может принимать разные аргументы или возвращать разные типы данных. Для перегрузки функции надо сверху этой функции расположить сигнатуры перегрузки. Сигнатуры перегрузки не содержат кода, в ней указывается кол-во аргументов, их тип, и тип, который вернет функция.

## Для чего нужен Typescript, плюсы и минусы

TypeScript — строго типизированный язык, основанный на JavaScript. Используется для статической типизации данных, для того чтобы избежать ошибок на этапе компиляции, для написания более понятного и читаемого кода, чтобы мы понимали какие данные приходят с бэка, передаются в аргументах функции, передаются в пропсах. Минусы - перед запуском его необходимо транспилировать в JS, Увеличение объема кода, Не защищает от ошибок в runtime

## Ошибки в typescript

1. Компиляция TypeScript кода с помощью TypeScript компилятора (tsc). Компилятор выдаст ошибки и предупреждения о типах и других проблемах в коде.
2. Использование интегрированных сред разработки (IDE) с поддержкой TypeScript, таких как Visual Studio Code, WebStorm, Atom и другие. Они обычно выделяют ошибки в коде, подсвечивая их красным цветом.
3. Использование сторонних инструментов для статического анализа кода, таких как TSLint или ESLint с плагинами для TypeScript. Они также могут выделять ошибки в коде и предлагать исправления.
4. Запуск тестовых сценариев для проверки правильности работы программы. Тесты могут помочь выявить ошибки, которые не были обнаружены при компиляции или статическом анализе кода.

## Можно ли использовать typescript в серверной разработке

Да, TypeScript может быть использован в серверной разработке. TypeScript может быть использован для написания серверных приложений на Node.js, а также для создания API и веб-сервисов.

# Такие же как и в JS: глобальные, блочные и функциональные

Такие же как и в JS: глобальные, блочные и функциональные

## enum

Enum представляет собой некую смесь объекта и массива. При обращении к enum возвращается его индекс. Получать данные мы можем по значению, а также по ключу.

Все элементы enum по умолчанию нумеруются и нумерация начинается по порядку от 0

```
```Typescript
enum Directions {
    Up,
    Down,
    Left,
    Right
}

Directios.Up;    // 0
Directions.Down; // 1
Directions.Left; // 2
Directions.Right; // 3
Directios.[0];   // 'Up'
Directions.[1];  // 'Down'
Directions.[2];  // 'Left'
Directions.[3];  // 'Right'
```
```

Также в enum мы можем задавать собственные индексы, что делает этот тип более гибким

```
```Typescript
enum Directions {
    Up = 2,
    Down = 4,
    Left = 6,
    Right
}

Directios.[2];   // 'Up'
```

```
Directions.[4]; // 'Down'
Directions.Left; // 6
Directions.Right; // 7 - Присвоился по умолчанию от предыдущего индекса
...`
```

Пример использования enum

```
`TypeScript
enum Links {
    youtube = 'https://youtube.com',
    vk = 'https://vk.com',
    google = 'https://google.com',
}

Links.vk // 'https://vk.com'
Links.youtube // 'https://youtube.com'
...`
```

Результатом компиляции enum будет анонимная самовызываемая функция, которая конструирует объект, так как типа enum в JS нет

```
`javascript
var Links;
(function(links) {
    links[youtube] = 'https://youtube.com',
    links[vk] = 'https://vk.com',
    links[google] = 'https://google.com',
})(links || (links = {}));
...`
```

Модификаторы доступа в typescript

1. ****public**** - значение по умолчанию. Свойство или метод доступны из любого места в коде, включая внешний код и наследующие классы.
2. ****private**** - свойство или метод доступны только внутри класса, в котором они определены.
3. ****protected**** - свойство или метод доступны только внутри класса, а также в классах, которые наследуют этот класс.
4. ****readonly**** - только для чтения

как typescript поддерживает необязательные параметры и дефолтные параметры

1. **Необязательные параметры:**

Можно объявить параметры функции необязательными путем добавления вопросительного знака ? после имени параметра. Это позволяет вызывать функцию с опущенными значениями для таких параметров. Внутри функции необязательные параметры будут иметь значение undefined, если не передано явное значение.

```
``typescript
function greet(name: string, age?: number) {
  if (age) {
    console.log(`Привет, ${name}! Тебе ${age} лет.`);
  } else {
    console.log(`Привет, ${name}!`);
  }
}
```

```
greet("Алексей"); // Привет, Алексей!
greet("Мария", 25); // Привет, Мария! Тебе 25 лет.
...
```

2. **Дефолтные параметры:**

Вы также можете определить значения по умолчанию для параметров функции. Если при вызове функции не передано значение для таких параметров, будут использованы значения по умолчанию.

```
``typescript
function greet(name: string, age: number = 30) {
  console.log(`Привет, ${name}! Тебе ${age} лет.`);
}
```

```
greet("Алексей"); // Привет, Алексей! Тебе 30 лет.
greet("Мария", 25); // Привет, Мария! Тебе 25 лет.
...
```

CSS псевдоклассы и псевдоэлементы

CSS псевдоклассы – hover, disabled, enabled, first-child, active

CSS псевдоэлементы – before, after, placeholder (нельзя получить доступ через javascript)

CSS селекторы

Селекторы: * - универсальный селектор выбирает все элементы страницы. button – селектор по элементу, .class_btn – селектор по классу

Приоритеты селекторов в CSS определяют, как браузер применяет стили к элементам при наличии конфликтующих правил.

Перечисление селекторов от самого приоритетного к менее приоритетных:

- inline стили
- селекторы id (#elem)
- селекторы классы (.elem)
- селекторы элементы (div)
- селекторы псевдоклассы и псевдоэлементы

(::last-child)

!important - ключевое слово, которое перебивает любые стили. НЕ СТОИТ ИСПОЛЬЗОВАТЬ!**

Приоритеты селекторов в CSS определяют, как браузер применяет стили к элементам при наличии конфликтующих правил. Приоритеты определяются на основе специфичности селекторов, веса важности и порядка записи стилей. Вот как работают эти факторы:

1. **Специфичность селекторов:**

- Специфичность измеряет, насколько точный и уникальный селектор. Чем выше специфичность, тем больший приоритет имеет правило.

2. **Вес важности (!important):**

- Правило, помеченное как !important, имеет наивысший приоритет и переопределяет другие стили, даже если они более специфичны.

3. **Порядок записи стилей:**

- Если правила имеют одинаковую специфичность и важность, то порядок их записи в таблице стилей

определяет, какое правило будет применено. Последнее правило будет иметь приоритет.

Когда возникает конфликт между стилями, рекомендуется избегать использования `!important`, поскольку это может сделать код менее поддерживаемым. Лучше стремиться к четким иерархиям селекторов и управлению конфликтами через специфичность.

CSS специфичность селекторов, как рассчитать специфичность

Специфичность селектора может быть представлена в виде трехзначного числа 0-0-0. Первое число

— это количество селекторов по ID, второе — количество селекторов по классу/атрибуту/псевдоклассу, а третья — количество селекторов по тегу/псевдоэлементам.

СПЕЦИФИЧНОСТЬ: 1 - `!important`, 2 – `INLINE STYLE`, 3 – селектор по ID, 4 – селектор класса, 5 – селектор тэга.

Каскад в CSS, означает, что порядок следования правил в CSS имеет значение; когда применяются два правила, имеющие одинаковую специфичность, используется то, которое идёт в CSS последним.

Например у нас есть табы, они отрисовывают один и тот же компонент, но с разными пропсами и там отрисовывается форма, и вот когда мы переключаемся между ними у нас не обнуляется состояние формы, так как react при сравнении деревьев видит, что корневой элемент не изменился и не перерисовывает этот компонент, чтобы это пофиксить можно передавать динамический `key`, и при обновлении ключа, компонент будет уничтожаться и строиться заново с новым состоянием.

Когда компонент обновляется, его экземпляр остаётся прежним, поэтому его состояние сохраняется между рендерами.

CSS normalize vs reset

CSS reset – сброс дефолтных стилей, марджинов, паддингов, шрифтов, он нужен чтобы сайт выглядел одинаково во всех браузерах.

Normalize – не сбрасывает дефолтные стили, а фиксирует некоторые стандартные браузерные стили и так же добавляет некоторые стили по умолчанию для улучшения кроссбраузерности.

CSS пост- и пре-процессоры зачем нужны

PostCSS это инструмент CSS, но с добавлением плагинов и сторонних инструментов. Препроцессор работает так же, как и Sass и LESS, трансформирует расширенный синтаксис и свойства в современный и дружелюбный CSS код.

Постпроцессоры решают самую назойливую проблему – автоматическое добавление вендорных префиксов к новым свойствам CSS3.

CSS препроцессор — это надстройка над CSS, которая добавляет ранее недоступные возможности для, с помощью новых синтаксических конструкций. SCSS позволяет использовать вложенности, переменные, миксины, наследование.

Семантика и семантическая верстка

Семантика - это верстка с помощью набора тегов, которым четко задан смысл и роль. Это такие тэги как: h1, header, footer, aside, section, main, nav, ul, li, table, button. Нужны для: улучшения SEO оптимизации, улучшают доступность для скринридеров которыми пользуются люди с ОВ.

Какие есть свойства position

Static	Дефолтный тип позиционирования
Relative	Элемент позиционируется относительно своего текущего положения
Absolute	Позиционирование относительно другого элемента у которого позиционирование не static. Если такого нет, то относительно окна браузера
Fixed	Позиционирование только относительно окна браузера
Sticky	В видимой области экрана элемент ведёт себя, как fixed. При дальнейшей прокрутке, скроллится вместе с родителем

Какие есть свойства display

Свойства **display**: **list-item**, **table**, **table-cell**, **table-column**, **grid**.

none - полностью удаляет элемент из интерфейса. Он не занимает места, но находится в HTML.

block (блочные) - Блок стремится расширяться на всю доступную ширину родителя. Можно указать ширину и высоту явно.

inline (строчные) - Ширина и высота определяется содержимым и нельзя задать ширину и высоту. Думайте об этом как о плавающем в абзаце слове.

inline-block (строчно-блочные) - Такой же, как и **block**, за исключением того, что ширина определяется содержимым. Можно указать ширину и высоту явно.

Способы отцентровать блок

Способы отцентровать – **text-align**, **flex**, **grid**, **position: absolute**, **table (vertical-align: middle;)**, **margin: 0 auto**

****ОТВЕТ:** Потому что этот метод сохраняет поток и не надо использовать **margin** и **position****

Использование ``transform`` для центрирования элемента имеет несколько преимуществ:

1. ****Простота кода:****

- Свойства ``top: 50%; left: 50%; transform: translate(-50%, -50%);`` предоставляют простой и лаконичный способ центрирования элемента относительно его родительского контейнера.

2. ****Динамичность:****

- Этот метод позволяет центрировать элементы различных размеров без необходимости изменения значений ``margin`` или ``position`` в зависимости от размеров элемента. ``translate(-50%, -50%)`` адаптируется к размерам элемента.

3. ****Отсутствие неопределенных размеров:****

- При использовании ``transform`` для центрирования нет необходимости задавать фиксированные размеры элемента. Это особенно полезно при работе с динамическим содержимым или при изменении размеров экрана.

4. ****Сохранение потока:****

- Используя ``transform``, элемент сохраняет свое место в потоке документа, что может быть важно, чтобы не нарушать структуру документа.

5. ****Производительность:****

- Браузеры эффективно обрабатывают ``transform``, и этот метод обычно имеет небольшое воздействие на производительность.

Хотя есть и другие способы центрирования элементов, `transform` является одним из наиболее распространенных и удобных методов для решения этой задачи.

Margin vs padding, схлопывание margin

margin - задает внешние отступы Padding - задаёт внутренние отступы

Схлопывание margin - Если блоку задан нижний отступ, а следующему за ним — верхний, то итоговый отступ будет равен большему отступу из двух.

rem em

Почему transform лучше для анимаций чем top/left. При использовании top/left и тд, для анимаций, на каждый кадр анимации браузер пересчитывает размеры и позиции элементов, то есть происходит layout происходит перерисовка элементов – repaint Размещает элементы относительно друг друга (композиция или composite), выносит все в GPU и показывает на экране.

Из-за этого страница может тормозить, поэтому лучше использовать transform, так как он триггерит только этап композиции

Отличие display:none от visibility: hidden или opacity: 0
visibility: hidden скрывает содержимое тега, но оставляет элемент в обычном потоке страницы то есть он по-прежнему занимает место
none - полностью удаляет элемент из интерфейса. Он не занимает места, но находится в HTML.

1em – единица, равная размеру текущего шрифта h1 { font-size: 20px } /* 1em = 20px */
p { font-size: 16px } /* 1em = 16px */

1rem - Это единица, равная корневому значению font-size. 1rem всегда будет равен значению font-size, которое было определено в html. (в большинстве браузеров по умолчанию это 16px).

БЭМ

БЭМ (Блок, Элемент, Модификатор) — это методология разработки, предложенная Яндексом для создания веб-приложений с использованием повторно используемых компонентов. Она сосредоточена на упрощении разработки и поддержки кода. В БЭМ различают три основные сущности: Блок, Элемент и Модификатор.

1. ****Блок**** — это логически и функционально самостоятельный компонент страницы, который может быть повторно использован. Примером блока может быть кнопка, заголовок, меню и т.д. Блоки предназначены для описания "главных героев" интерфейса.

2. ****Элемент**** — это составная часть блока, которая не может использоваться в отрыве от него. Например, пункт меню (`menu item`) является элементом блока меню (`menu`). Элементы делают структуру блока более детализированной.

3. ****Модификатор**** — это свойство блока или элемента, которое изменяет их внешний вид или поведение. Модификатор может использоваться для задания разных стилей одной кнопке — например, сделать кнопку большой (`button_size_large`) или изменить цвет (`button_color_red`).

Важность БЭМ-сущностей заключается в возможности создавать сложные интерфейсы из небольших и понятных частей, облегчая тем самым разработку и последующую поддержку кода. БЭМ также способствует уменьшению взаимосвязей в CSS, делая стили более предсказуемыми и уменьшая риск неожиданных переопределений.

keyframes transform

keyframes и transform связаны с анимацией.

- keyframes - определяют набор стилей, который применяется в указанной последовательности и с указанной частотой. В этот набор также может входить transform. Это очень гибкий инструмент для создания анимации.

- transform - свойство, для реализации более простой анимации.

`@keyframes` и `transform` - это два разных понятия в CSS, связанных с анимациями, и они выполняют разные функции.

1. ****`@keyframes` (Кадры):****

- **`@keyframes`** - это правило CSS, которое используется для создания анимаций. Оно позволяет определить набор стилей, которые должны применяться поочередно в течение определенного времени анимации. Ключевые кадры обозначают начальное, промежуточное и конечное состояния анимации.

```
```css
@keyframes slide {
 0% {
 transform: translateX(0);
 }
 50% {
 transform: translateX(100px);
 }
 100% {
 transform: translateX(200px);
 }
}
```

```
.element {
 animation: slide 2s infinite;
}
...
```

В этом примере создается анимация с именем `slide`, которая перемещает элемент вправо относительно его исходного положения.

## 2. `transform`:

- `transform` - это свойство CSS, которое применяется для преобразования элемента. Оно может использоваться для множества видов преобразований, таких как перемещение (`translate`), вращение (`rotate`), масштабирование (`scale`) и другие.

```
```css  
.element {  
  transform: translateX(100px) rotate(45deg) scale(1.5);  
}  
...
```

Это пример использования `transform` для перемещения элемента на 100 пикселей вправо (`translateX`), поворота элемента на 45 градусов (`rotate`) и увеличения размера элемента в 1.5 раза (`scale`).

Таким образом, `@keyframes` используется для определения последовательности стилей в течение анимации, в то время как `transform` - это свойство, которое применяется для конкретных преобразований элемента. Ключевые кадры могут использовать `transform` для задания конкретных стилей в различных точках анимации.

Какие способы есть вывести элемент из потока

- `position: absolute`
- `position: fixed`
- `float: left` или `float: right`

Вывести элемент из потока (take an element out of the normal flow) означает изоляцию элемента от окружающих элементов, так что остальные элементы не будут влиять на его позицию и размер

Разница div span

- div - используется для создания блочных контейнеров, а span - для выделения части текста внутри строки.

- div - блочный элемент (block), а span - строчный (inline)

`div` и `span` - это два различных HTML-элемента, каждый из которых предназначен для различных целей в структуре веб-страницы.

***`<div>` (Block-level element):**

`div` (сокращение от division) является блочным элементом и используется для группировки других элементов в блоки. Он создает блочный контейнер, который занимает всю ширину родительского контейнера и начинается с новой строки. `div` часто используется для стилизации с помощью CSS, форматирования разметки и создания структуры страницы.

***`` (Inline element):**

`span` является строчным элементом и используется для выделения или стилизации части текста внутри строки. Он не создает новой строки и занимает только столько места, сколько необходимо для его содержимого. `span` часто используется внутри текста для применения стилей к определенным словам или фразам.

Зачем нужен key

key - это атрибут, который нужно использовать при создании списков из элементов массива.

Проп key помогает React определять, какие элементы массива подверглись изменениям, были добавлены или удалены. Он должен быть уникальным, их необходимо указывать, чтобы React мог сопоставлять элементы массива с течением времени, по умолчанию будут использоваться index.

Использовать индексы в качестве ключей не рекомендуется, если порядок расположения элементов может измениться, потому что это может повлечь за собой неожиданное поведение состояний. При изменении ключа компонент демонтируется и монтируется заново.

Если ключи основаны на индексах массива и порядок элементов изменится, React будет неправильно определять, какие элементы изменились, и может неправильно переиспользовать существующие компоненты, что приведет к ошибочному поведению.

При добавлении, удалении или сортировке элементов в списке, элементам с новыми индексами нужно будет заново перерендериваться, даже если их содержимое не изменилось. Это может негативно сказаться на производительности, особенно для длинных списков.

Компоненты могут не правильно обновляться при изменении списка, так как React может неправильно ассоциировать предыдущее состояние компонента с новым элементом списка, основываясь только на индексе массива.

Порталы в React

Порталы в React представляют собой механизм для рендеринга компонентов в узле DOM, который находится вне текущей иерархии компонента родителя. Это полезно в ряде ситуаций, например, при создании модальных окон, всплывающих подсказок или в случаях, когда необходимо избежать проблем со стилизацией из-за наложения CSS или "спортивных проблем" в позиционировании элементов.

Порталы создаются с использованием функции `ReactDOM.createPortal()`, которая принимает два аргумента: реактовый элемент, который вы хотите отрендерить, и DOM-узел, в который вы хотите его поместить.

```
import React from 'react';
import ReactDOM from 'react-dom';

class Modal extends React.Component {
  render() {
    // Здесь modalRoot -- это DOM-элемент, в который будет вставлен портал.
    // modalRoot должен быть определен в вашем HTML файле.
    return ReactDOM.createPortal(
      this.props.children,
      document.getElementById('modalRoot')
    );
  }
}

class App extends React.Component {
  render() {
    return (
      <div onClick={() => console.log('Клик внутри App')}>
```

```

    Кликните на App для события.
    <Modal>
      <div onClick={e => e.stopPropagation()}>Модальное окно</div>
    </Modal>
  </div>
);
}
}

```

В этом примере, компонент `Modal` использует `ReactDOM.createPortal()` для рендера дочерних элементов в `modalRoot`, который находится вне текущего компонента `App`. Это позволяет модальному окну быть визуально и функционально независимым от иерархии родительского компонента, что особенно полезно для UI-элементов, которые должны "выходить" за рамки своих родителей.

Что такое HOC

HOC — это особая техника в React, при которой функция принимает аргумент Component и возвращает новый компонент добавляя в него данные и/или функционал. HOC являются "**чистыми**" функциями. Встроенные функции высшего порядка - **forwardRef, memo, withRouter**

Говоря просто, компонент высшего порядка — это функция, которая принимает компонент и возвращает новый компонент.

можно создать собственный HOC (компонент высшего порядка) в React. HOC в React — это мощный механизм для повторного использования компонентной логики, который позволяет разрабатывать абстрактные компоненты, внедряющие в себя пропсы и поведение, которые можно использовать многократно.

Render Props является альтернативой использованию Higher-Order Components (HOC) для композиции логики в React-компонентах. Этот паттерн позволяет более гибко и динамично компоновать компоненты и их логику.

Render Props (передача через рендеринг) - это паттерн в React, который представляет собой способ передачи функции (которая возвращает React-элемент) в компонент, чтобы этот компонент мог использовать эту функцию для рендеринга чего-то внутри себя.**

Суть этого паттерна в том, чтобы делегировать часть логики по рендерингу компоненту, который может быть более универсальным.

Когда вы используете Render Props, вы передаете функцию в качестве пропса, и этот компонент вызывает эту функцию, передавая ей какие-то данные. Эта функция должна возвращать React-элемент, который будет включен в рендер компонента.

Пример использования Render Props:

```

```JSX
import React from 'react';

// Компонент с Render Props
const RenderPropsComponent = ({ render }) => {
 // Логика компонента...
 const data = 'Some data';

 // Рендер компонента с использованием переданной функции
 return <div>{render(data)}</div>;
};

// Компонент, использующий Render Props
const App = () => {
 return (
 <RenderPropsComponent
 render={(data) => (
 <p>The data is: {data}</p>
)}
 />
);
};

export default App;
```

```

В этом примере `RenderPropsComponent` принимает пропс `render`, который является функцией. Эта функция вызывается внутри компонента, и результат ее выполнения включается в рендер компонента.

Redux

Redux - это стейт менеджер для хранения состояния приложения, основанный на паттерне проектирования **Flux**.

принципы Redux:

единственный источник истины (**store**).

Состояние только для

чтения (единственный способ обновить состояние – **action**).

Изменения вносятся с помощью чистых функций (**reducers**)

Flux паттерн в редукс

Flux — это архитектурный шаблон, разработанный Facebook для создания клиентских веб-приложений. Flux предлагает однонаправленный поток данных:

1. ****Диспетчер (Dispatcher)****: Центральный хаб, который управляет потоком данных и уведомляет хранилища о действиях.
2. ****Действия (Actions)****: Пакеты информации, которые предоставляют данные и тип операции хранилищу.
3. ****Хранилища (Stores)****: Контейнеры для состояния приложения и бизнес-логики, реагирующие на действия.
4. ****Представления (Views)****: React-компоненты, которые реагируют на изменения в хранилищах и обновляются соответственно.

Всё начинается с генерации действий, которые диспетчеризуются в хранилища, где обновляется состояние. После обновления состояния хранилища извещают представления, что ведет к их обновлению.

Что такое store в redux

Store — это объект, который соединяет эти части вместе. Стор берет на себя следующие задачи:

содержит состояние приложения (application state); предоставляет доступ к состоянию с помощью `getState()`; предоставляет возможность обновления состояния с помощью `dispatch(action)`

Что такое action в redux

Action Creators – это функции, возвращающие объекты действий.

Экшены — это обычные JavaScript-объекты. Экшены должны иметь поле `type`, которое указывает на

тип исполняемого экшена и если нужно `payload` чтобы передать какие то данные. В Redux, действие (action) — это объект JavaScript, который передает информацию от приложения к хранилищу (store). Каждое действие имеет поле ``type``, которое указывает тип выполняемой операции, и часто включает другие данные, которые необходимы для обновления состояния хранилища.

Действия отправляются в хранилище с помощью метода ``dispatch()``, после чего они обрабатываются редьюсерами (reducers) для обновления состояния приложения.

Что такое reducer в redux

Редьюсер (reducer) — это чистая функция, которая принимает предыдущее состояние и экшен (state и action) и возвращает следующее состояние. В редьюсере нельзя:

- Выполнять какие-либо сайд-эффекты: обращаться к API или осуществлять переход по роутам;
- Вызывать не чистые функции, например `Date.now()` или `Math.random()`

Что такое dispatch в redux

dispatch() – это метод, который принимает экшн и позволяет изменить состояние. В контексте Redux и других библиотек управления состоянием, dispatch — это метод, используемый для отправки (или "диспатчинга") действия в хранилище. Это главный механизм для триггера изменений состояния в приложении. Когда действие отправляется с помощью dispatch, хранилище запускает свои редьюсеры и передает им текущее состояние и отправленное действие, чтобы вычислить и вернуть новое состояние.

Что такое чистая функция

Чистая функция в программировании — это функция, которая удовлетворяет двум основным условиям:

1. ****Детерминированность****: Она возвращает одинаковый результат при одинаковых аргументах, не имея побочных эффектов. Это означает, что вызов функции может быть заменён её результатом без изменения поведения программы.
2. ****Отсутствие побочных эффектов****: Функция не вызывает изменений за пределами своего контекста, то есть не изменяет глобальные переменные, не изменяет входные данные и не производит I/O операций (ввод/вывод).

```
``javascript
function sum(a, b) {
  return a + b;
}
```

Эта функция всегда возвращает один и тот же результат для одних и тех же аргументов и не изменяет никакие внешние состояния.

Дескрипторы свойств объектов

Дескрипторы свойств объектов (*property descriptors*) - это объекты, которые описывают, какое поведение должно быть у свойства объекта при чтении, записи или удалении свойства.

1. ``value`` - значение свойства;
2. ``writable`` - можно ли менять свойство с помощью присваивания `true/false`;
3. ``enumerable`` - является ли свойство доступным для итерации в цикле `for..in` или методе `Object.keys()` (`true/false`);
4. ``configurable`` - является ли свойство доступным для удаления или его дескриптор быть изменен (`true/false`);
5. ``get`` - функция, вызываемая при чтении значения свойства;
6. ``set`` - функция, вызываемая при записи значения свойства.

Они используются для более тонкой настройки свойств объектов в JavaScript.

Дескрипторы свойств включают следующие свойства:

1. `value`: Значение свойства.
2. `writable`: Определяет, можно ли изменить значение свойства. Если установлено значение `true`, то свойство может быть изменено с помощью присваивания. Значение по умолчанию - `false`.
3. `enumerable`: Определяет, будет ли свойство видимым при итерации по свойствам объекта с использованием цикла `for...in` или метода `Object.keys()`. Значение `true` делает свойство видимым, а `false` - скрывает его. Значение по умолчанию - `false`.
4. `configurable`: Определяет, может ли свойство быть удалено из объекта или его дескриптор быть изменен. Значение `true` позволяет изменять или удалять свойство, а `false` - делает его неподлежащим изменению. Значение по умолчанию - `false`.
5. `get`: Функция, вызываемая при доступе к свойству. Когда свойство читается, вызывается функция `get`. Если свойство не имеет аксессуара, значение этого свойства равно `undefined`. Значение по умолчанию - `undefined`.
6. `set`: Функция, вызываемая при присваивании значения свойству. Когда свойству присваивается новое значение, вызывается функция `set`. Если свойство не имеет аксессуара, присваивание значения игнорируется. Значение по умолчанию - `undefined`.

Дескрипторы свойств позволяют изменять поведение свойств объектов при различных операциях, например, запрещать изменение или удаление свойства, скрывать свойство от перечисления, создавать свойства с геттерами и сеттерами и т.д.

Для настройки дескрипторов свойств объектов в JavaScript используются методы `Object.defineProperty()` и `Object.defineProperties()`, которые позволяют определять новые свойства или изменять существующие свойства объектов.

Как работает `useState`, зачем нужен

Хук **`useState`** принимает один аргумент - начальное состояние и возвращает массив, первый элемент которого - текущее состояние, а второй - колбек с помощью которого нужно обновлять состояние. После каждого вызова `setState` React перендерит компонент. **`useState`** — это асинхронный хук, и он не меняет состояние сразу, он должен ждать повторного рендеринга компонента. Когда `state === true` и мы ставим `setState(true)`, то `console.log()` сработает только в первый раз, не вызовет перендер чайд-компонентов! `const [state, setState] = useState(compute()) – initValue` сохраняется, но вычислительная функция вызывается при каждом рендеринге. `useState(() => compute())` так только 1 раз.

State иммутабельный, и может обновляться асинхронно, поэтому не следует напрямую мутировать состояние, так же `setState` вызовет метод `render` для обновления.

Как работает useEffect, зачем нужен

useEffect дает возможность выполнять side effects в функциональном компоненте. Побочные эффекты это запросы, изменение DOM, подписки. Данный хук React вызывает асинхронно, после того, как браузер отрисует компоненты (Вызывается после стадии painting). Что бы не выполнять эффекты постоянно, можно передавать массив зависимостей вторым параметром. Также в useEffect можно вернуть cleanup функцию и после каждого повторного рендеринга с измененными зависимостями React сначала запускает cleanup со старыми значениями, а затем запускает effect с новыми значениями. Также при удалении компонента из DOM, запускается cleanup.

- массив **не** указан: эффект запускается при каждом рендере
- **пустой** массив: эффект запускается только при маунте
- массив **с элементами**: эффект запускается при изменении любого элемента

Как работает useLayoutEffect, зачем нужен

useLayoutEffect обладает таким же API, как и **useEffect**, с тем отличием, что он вызывается синхронно, после всех вычислений мутаций в DOM, то есть блокирует отрисовку браузера, в то время как **useEffect** вызывается **асинхронно** и **не** блокирует рендер. (срабатывает до отрисовки в браузере). Срабатывает когда компоненты уже находятся в virtual dom (можно прочитать/установить различные свойств), но еще не были отрисованы браузером. Вызывается до стадии painting. (Примеры использования: Измерение размеров DOM элементов; Обновление положения скrolла чтобы не было мерцания)

Как работает useRef, зачем нужен

useRef возвращает мутабельный **ref-объект**, свойство `.current` которого инициализируется переданным аргументом. С помощью **useRef** можно получить доступ к DOM элементу, или сохранять значение, которое не будет триггерить рендеры при изменении. Мутирование свойства `.current` не вызывает повторный рендер. То есть это один и тот же объект при каждом рендере.

Что такое контекст в реакт, хук useContext

useContext Принимает объект контекста и возвращает текущее значение контекста для этого контекста, переданного в `provider`.

Контекст в React — это концепция, которая позволяет вам снабжать дочерние компоненты глобальными данными, независимо от того, насколько глубоко они находятся в дереве компонентов, обернув все `provider`. Для избежания **prop drilling'a** (процесс передачи данных от родительского компонента через промежуточные компоненты к вложенным дочерним компонентам, даже если эти промежуточные

компоненты не используют эти данные. Это может привести к усложнению кода, так как приходится передавать пропсы через множество уровней компонентов, что делает компоненты сильно связанными и менее переиспользуемыми.

В больших приложениях это может стать проблематичным, и для управления состоянием на глобальном уровне часто используют стейт-менеджеры, такие как Redux или контекст (Context API) в React.

).

Context, при изменении значения в провайдере, вызывает рендер для каждого компонента. Redux вызывает рендер только для тех компонентов, которые непосредственно используют измененные значения. Но при использовании Redux, для каждого компонента запускается селектор и сравниваются значения.

как работает useCallback

useCallback позволяет мемоизировать переданный колбек, то есть при каждом рендере, ссылка на функцию, будет неизменна. Ссылка будет новой только, когда изменится одна из переданных зависимостей. Стоит юзать только когда передаем через пропсы другому компоненту, использующему React.memo, или в зависимости другим хукам, например useEffect.

Как работает useМемо, зачем нужен

useМемо – принимает колбэк и позволяет мемоизировать возвращаемое значение. будет пересоздано только тогда, когда изменятся значения его зависимостей. useМемо полезен для вычислений, потребляющих много ресурсов, где повторное вычисление можно избежать.

useМемо возвращает мемоизированное значение после выполнения переданной функции, а useCallback возвращает мемоизированную функцию.

Используйте useМемо, когда нужно избежать дорогостоящих вычислений при ререндере.

Используйте useCallback, чтобы не потерять ссылку на функцию и предотвратить ненужные ререндеры дочерних компонентов, которые зависят от этой функции как от пропа.

Что такое React мемо и зачем нужен

React.memo — это компонент высшего порядка, который позволяет оптимизировать производительность функциональных компонентов, избегая их ненужных ререндеров.

React.memo призван помочь в ситуациях, когда компонент всегда рендерит одно и то же при неизменных пропсах.

Компонент, обёрнутый в React.memo, будет повторно рендериться только в случае, если изменились его пропсы. Это сравнение пропсов происходит поверхностно, что значит, что React.memo сравнивает пропсы и их значения на одном уровне глубины.

React мемо - это компонент высшего порядка, который будет обновляться только если его

предыдущие пропсы не равны новым пропсам. Вторым аргументом принимает функцию для сравнения пропсов, возвращающую Boolean значение, если false – перерисуем

В каких случаях нужно использовать мемоизацию и зачем

Мемоизация недешевая. Она работает только тогда, когда мы оборачиваем компонент в мемо, и когда все пропс-ы обернуты в `useMemo` и `useCallback`. Ее стоит использовать: в зависимостях к `useEffect`, для дорогих вычислений и для высоконагруженных компонентов, например списков.

Как работает `useReducer`, зачем нужен

Хук `useReducer(reducer, initialState)` принимает 2 аргумента: чистую функцию `reducer` которая определяет, как обновляется состояние и начальное состояние. И возвращает массив с: текущим состоянием и функцией `dispatch`.

`useReducer` — это хук в React, который используется для управления сложным состоянием компонента. Он предоставляет альтернативный `useState` способ работы со состоянием, особенно когда:

- Состояние содержит несколько значений, которые зависят друг от друга.
- Следующее состояние зависит от предыдущего.
- Логика обновления состояния достаточно сложна (например, содержит несколько условий).
- Действия по изменению состояния делятся на множество подзадач.

`useReducer` принимает редьюсер функцию и начальное состояние. Редьюсер — это функция, которая принимает текущее состояние и действие, а затем возвращает новое состояние. Также `useReducer` возвращает текущее состояние и функцию `dispatch`, которую можно использовать для исполнения действий.

Оба хука отвечают за управление состоянием, но `useState` лучше использовать в примитивной логике, а `useReducer` — когда есть какая-то сложная логика.

Разница между `useState` и `useReducer` связана с их различными способами управления состоянием в React компонентах.

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

Этот хук делает управление состоянием более предсказуемым и упрощает тестирование, так как логика обновления состояния концентрируется в одном месте — в редьюсере. Также упрощается передача обработчиков действий компонентам, поскольку достаточно передать одну функцию `dispatch` вместо множества колбэков.

Offscreen

Компонент `Offscreen` в React представляет новую возможность, предоставляемую React 18, которая позволяет рендеринг компонентов вне экрана (offscreen).

****Вот зачем он нужен:****

- ****Отложенный Рендеринг (Deferred Rendering)**:** Он позволяет разработчикам отложить рендеринг ненужного в данный момент контента, например, пока он не в зоне видимости. Это может помочь улучшить производительность, позволяя браузеру сосредоточиться на наиболее важном содержимом.

- ****Подготовка Контента**:** `Offscreen` можно использовать для предварительной загрузки и рендеринга интерфейсов, которые пользователь может понадобится в будущем. Так, когда пользователь получит доступ к этим интерфейсам, переключение будет мгновенным.

- ****Проработка Интерактивности**:** С помощью этого компонента можно рендерить компоненты в состоянии "скрыто", пока они не станут активными. Это полезно для вкладок, меню, модальных окон и других элементов, которые пользователь не видит сразу.

****Пример использования:****

```
```jsx
import { Offscreen } from 'react';

function MyComponent() {
 return (
 <Offscreen mode="hidden">
 { /* Этот компонент будет предварительно рендериться, но скрыт до активации */ }
 <ExpensiveUI />
 </Offscreen>
);
}
```
```

В ранних версиях React 18 компонент `Offscreen` ещё не полностью реализован, и используется в основном внутри Facebook. Планируется, что в будущем он станет

открытым и доступным для использования в широком диапазоне задач, связанных с улучшением производительности и пользовательского опыта.

Что такое jsx

JSX - это расширение синтаксиса JavaScript, который позволяет использовать **XML**-подобный синтаксис в JS, это синтаксический сахар для функции `React.createElement()`. В итоге с помощью `babel` компилируется в обычный JS и вызовы `createElement`, а после в объект **{props, key, ref, type}**.

Нельзя вызывать компонент как функцию(), реакт расценит это как кастомный хук. Кастомный хук в React — это функция, которую вы создаете для повторного использования логики состояния и побочных эффектов между различными компонентами. Как и встроенные хуки, такие как `'useState'` и `'useEffect'`, **кастомный** хук может использовать другие хуки внутри себя.

Кастомные хуки предлагают мощный способ абстрагировать компонентную логику, делая код чище и проще в обслуживании, и помогают избежать дублирования кода. В названии кастомных хуков обычно используют префикс `'use'`.

Пример кастомного хука:

```
```javascript
import { useState, useEffect } from 'react';

function useWindowSize() {
 const [size, setSize] = useState({ width: window.innerWidth, height: window.innerHeight });

 useEffect(() => {
 function handleResize() {
 setSize({ width: window.innerWidth, height: window.innerHeight });
 }

 window.addEventListener('resize', handleResize);
 return () => window.removeEventListener('resize', handleResize);
 }, []);

 return size;
}

// Использование кастомного хука в компоненте
```

```
function MyComponent() {
 const { width, height } = useWindowSize();

 return (
 <div>
 {width} x {height}
 </div>
);
}
...

```

В этом примере кастомный хук `useWindowSize` позволяет легко получить и отслеживать размер окна браузера в разных компонентах.

## Какие хуки появились в React 18, `useId`, `useTransition`

**`useId`** – хук который возвращает уникальный id, который можно использовать в атрибутах HTML элементов по типу `input`, `label`.

**`useTransition`** позволяют помечать некоторые обновления состояния как несрочные.

**`useSyncExternalStore`** — добавляет поддержку параллельного чтения для внешних хранилищ;

**`useDeferredValue`** - можно обернуть значение и пометить его изменения как менее важные и отложить повторный рендеринг. до тех пор, пока браузер не будет бездействовать.

## Что такое контролируемые и неконтролируемые компоненты

Контролируемые и неконтролируемые компоненты в React относятся к разным способам управления формами:

- **\*\*Контролируемые компоненты\*\***:

- Элементы форм (например, ``, ``, `<select>`) управляются состоянием React компонента.</li>
<li>- Изменения ввода пользователя обрабатываются функциями обработчиками (например, `onChange`) и отражаются в состоянии компонента.</li>
<li>- Значение элемента формы всегда синхронизировано со значением состояния.</li>
</ul>
</div>
<div data-bbox="144 844 627 896" data-label="Text">
<pre>```jsx
function Form() {
 const [value, setValue] = React.useState("");
</pre>
</div>

```

function handleChange(event) {
 setValue(event.target.value);
}

return <input type="text" value={value} onChange={handleChange} />;
}
...

```

- **\*\*Неконтролируемые компоненты\*\***:

- Элементы форм управляются напрямую DOM, а не состоянием React.
- Для доступа к значениям этих элементов используются refs (ссылки на DOM-узлы).
- Используются, когда хочется избежать управления состоянием формы в коде React или при интеграции React с неконтролируемым кодом или сторонними библиотеками.

```

```jsx
function Form() {
  const inputRef = React.useRef();

  function handleSubmit(event) {
    alert('A name was submitted: ' + inputRef.current.value);
    event.preventDefault();
  }

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
}
...

```

В контролируемых компонентах React ответственен за управление состоянием и интерфейсом, в то время как в неконтролируемых компонентах это делают нативные элементы HTML.

Что такое React Fiber

Также в реакт 16 появился Fiber — это новый движок согласования, который тоже улучшает производительность.

React Fiber — позволяет React воспользоваться планированием. Его цель предоставить возможность: остановить работу и вернуться к ней позже,

приоритизировать разные типы работы, переиспользовать работу проделанную ранее, отменить работу. (Работа – какие-нибудь обновления состояния).

Также он позволяет приостанавливать и возобновлять процесс рендеринга, чтобы избежать блокировки всего интерфейса.

Во время согласования (reconciliation) данные каждого React элемента, возвращенные из метода render, объединяются в дерево fiber-узлов. Каждый React элемент имеет соответствующий fiber-узел. В отличие от React элементов, fibers не создаются заново при каждом рендере. Это мутабельные структуры данных, которые хранят состояние компонентов и DOM.

React Fiber — это переписанный алгоритм согласования (reconciliation) и рендеринга в библиотеке React. Это не отдельная библиотека, а внутреннее улучшение основного алгоритма React, внедрённое начиная с версии 16.

Основные цели React Fiber:

1. ****Инкрементальный рендеринг****: Возможность разбивать работу рендеринга на части и выполнять их по очереди. Это помогает React более гибко управлять приоритетами задач и ответить на действия пользователя напрямую.
2. ****Приоритизация задач****: В Fiber есть возможность устанавливать приоритеты различным процессам согласования, что позволяет React сосредотачивать ресурсы на более важных с точки зрения UX задачах.
3. ****Поддержка конкурентности (concurrency)****: React может готовить несколько версий UI для обновления одновременно и обработать неожиданные прерывания (например, анимации или загрузка данных).
4. ****Лучшая обработка ошибок****: Fiber вводит новые API для обработки ошибок, такие как `getDerivedStateFromError` и `componentDidCatch`, позволяя компонентам «ловить» ошибки и обрабатывать их более изящно.

Fiber использовал своего рода виртуальный стек кадров для управления работой, что помогает обрабатывать сложные приложения более плавно, избегая фризов и заиканий.

React преимущества и недостатки

React преимущества:

- **Virtual DOM** - в отличие от реального DOM, занимает мало места и быстро обновляется, тем самым повышая производительность.
- Повторное применение компонентов – уменьшение количества кода, использование JSX.
- Односторонний поток данных – передача пропсов от родителя к ребенку.

Минусы: вес библиотеки, не полное решение (т.е. нужно юзать доп. либы для роутинга и тд.), отсутствие четкой структуры как в ангуляр например??

Что такое React lazy suspense

React.lazy()

— позволяет рендерить динамически импортируемые компоненты как обычные компоненты.

Компоненты, загружаемые с помощью React.lazy(), должны быть обернуты в компонент

Suspense, который позволяет отображать резервный контент (например, индикатор загрузки).

Позволяет уменьшить размер бандла приложения путём отображения только тех компонентов, которые нужны пользователю.

Атрибуты для lazy подгрузки медиа в хтмл

Атрибуты loading="lazy" у img и preload="none" у audio/video позволяет подгружать файлы, в тот момент когда они будут нужны

Порядок рендера компонентов и вызова хуков

алгоритм reconciliation выполняет обход в глубину, и рендеринг компонента завершается только после завершения рендеринга всех его дочерних элементов. В результате корневой компонент в вашем дереве всегда будет завершать рендеринг последним.

Хуки нельзя вызвать в циклах/условиях, их можно вызывать только внутри тела компонента. Потому что React полагается на порядок вызова хуков. Т.к хуки должны вызываться в одинаковой последовательности при каждом рендере компонента. Это позволит React правильно кешировать состояние хуков между рендерами. Также в devtools можно увидеть, что каждый хук пронумерован.

Что такое батчинг, его плюсы

Батчингом в React называют процесс группировки нескольких вызовов обновления состояния в один

этап ререндера. Это положительно сказывается на производительности.

В реакт 18 здесь будет 1 ререндер, в остальных версиях 2:

```
Const click = () => {  
  setState1()  
  setState2()  
}
```

Зачем нужен хок `forwardRef`

`forwardRef` - это функция, которая принимает функциональный компонент и возвращает новый функциональный компонент, который может иметь атрибут `ref`. Если просто передать `ref` в компонент не использующий `forwardRef`, то будет ошибка.

`React.forwardRef` — это применяемая в React функция, позволяющая перенаправить `ref`, которые передаются к компоненту, в один из его дочерних компонентов. Это очень полезно в тех случаях, когда компоненты более высокого порядка (например, функциональные компоненты) должны иметь возможность передавать `ref` вложенным компонентам.

Хук `useImperativeHandle`

`useImperativeHandle` принимает `ref` переданный от родительского компонента, Вторым аргументом принимает функцию (`init`), которая возвращает объект, что будет записан в `ref`. Позволяет записывать в `ref` любые данные изнутри функционального компонента.

`useImperativeHandle` — это хук в React, который используется совместно с `forwardRef` для настройки значения `ref`, передаваемого от родительского компонента. Он позволяет дочернему компоненту явно указать, какие функции или значения должны быть доступны родительскому компоненту через `ref`.

Этот хук обычно используется, когда необходимо предоставить родительскому компоненту доступ к определенным методам внутри дочернего компонента, при этом скрывая детали реализации дочернего компонента.

Error boundaries что такое

Error boundaries — это способ обрабатывать ошибки в приложении. Это классовый компонент который

предоставляют резервный UI интерфейс, если это применимо. Можно реализовать только на классах

т.к в хуках не получится отловить ошибку, нет метода `componentDidCatch`.

Жизненный цикл компонента и методы жц в классах, порядок вызова

`componentDidMount()` вызывается после монтирования компонента в DOM-дерево. подходящее место для подписок, запросов на сервер.

componentDidUpdate(prevProps, prevState,) вызывается сразу после обновления. Этот метод не вызывается при первоначальной отрисовке. Это также подходит для выполнения запросов, если нужно сделать запрос при изменении текущих свойств.

componentWillUnmount() вызывается перед размонтированием компонента. Выполните необходимую очистку в этом методе, такую как отмена таймеров, сетевых запросов или очистка любых подписок, созданных в **componentDidMount**().

shouldComponentUpdate(nextProps, nextState) вызывается перед отрисовкой при получении новых свойств или состояний. Значение по умолчанию равно true.

getDerivedStateFromProps вызывается непосредственно перед вызовом метода **render()**, как при начальном монтировании, так и при последующих обновлениях. Он должен вернуть объект для обновления состояния или null, чтобы ничего не обновлять.

Инициализация – компонент готовит установку начального состояния и параметров по умолч. Монтирование – компонент готов для монтирования в ДОМ дерево браузера **willMount**, **didMount**. Обновление – обновление состояния компонента, **didUpdate**, **willUpdate**, **shouldUpdate**.

Размонтирование – компонент удаляется из ДОМ дерева браузера.

extra reducer redux toolkit

extraReducer позволяет **createSlice** реагировать и обновлять свое собственное состояние в ответ на другие типы действий, помимо сгенерированных им типов.

Extra reducer в Redux Toolkit нужен для обработки специфичной логики аккаунта пользователя, которая может быть независима от остальных состояний в приложении.

Что такое middleware

****ОТВЕТ:**

Middleware (мидлвар) - это функция, которая обрабатывает `actions` перед тем, как они попадают в `reducer`.

Middleware используется для повышения гибкости и расширяемости приложений. Они упрощают реализацию сложной логики, которая не может быть реализована с помощью простых действий и редьюсеров.

Например, `middleware` может быть использован для выполнения следующих функций:

- ***Логирование***: middleware может логировать действия и состояние приложения для отладки и анализа производительности. `createLogger`
- ***Асинхронные запросы***: middleware может выполнять асинхронные запросы к серверу и обновлять состояние приложения, когда ответ будет получен. `thunk`
- ***Авторизация***: middleware может проверять, авторизован ли пользователь, и блокировать доступ к защищенным страницам, если пользователь не прошел аутентификацию.
- ***Оптимизация***: middleware может использоваться для оптимизации и кеширования запросов, чтобы уменьшить нагрузку на сервер и ускорить производительность приложения.

``Middleware`` ***работает по принципу цепочки обязанностей (chain of responsibility)***. Каждый middleware получает действие и может изменять его, передавать его следующему middleware или передавать управление редьюсеру. Middleware может быть добавлено в Redux с помощью функции ``applyMiddleware``, которая принимает список middleware и возвращает функцию, которая принимает `createStore` и возвращает усовершенствованный экземпляр ``store``.

Поскольку редьюсер - это чистая функция, которая принимает action и предыдущий state и возвращает новую версию state, мы не можем в нем делать те или иные побочные действия. Для этого и нужен middleware - для того чтобы, выполнять логирование, запросы и так до того момента пока action не попадет в reducer

Почему ушли от классов к функциям в реакте

Почему ушли от классов к функциям: 1 – функции более читабельные и легче для понимания, 2 – не нужно беспокоиться об `this`, биндить методы, 3 – не нужно использовать методы по типу `constructor`, `render`, 4 – в функциях можно юзать хуки, а в классах – методы жизненного цикла, 5 – в функциональных функции пересоздаются при каждом рендере, в классах - нет

Причины перерисовки в реакт

Причины перерисовки react – изменение `props`, изменение состояния, контекста, `forceUpdate` в классах, ререндер родительского компонента (вызывает ререндер дочерних), изменение `key`

Redux vs Context плюсы и минусы

REDUX:

- + поддержка асинхронных экшнов и middleware, такие как saga, thunk и т.д;
- + явное отделение бизнес-логики от UI, более удобно писать код;
- + devtools для отслеживания всех обновлений состояния.
- + более оптимизирован в отличие от context, есть reselect, entityAdapter для нормализации.

вес npm пакета

приходится писать бойлерплейт код - Бойлерплейт код (boilerplate code) для Redux относится к повторяющимся участкам кода, которые мало меняются от проекта к проекту и приходится писать каждый раз при создании новых действий (actions), типов действий (action types), редьюсеров (reducers) и хранилищ (stores). Это стандартные шаблоны и конструкции, необходимые для поддержания унифицированной структуры управления состоянием, но которые могут порождать много дублирующегося кода.

CONTEXT:

- + не нужно скачивать npm пакеты
 - + позволяет избавиться от prop-drilling и создать глобальное состояние
- оптимизация, при изменении состояния ререндерит всех потребителей контекста
- не такое явное отделение логики от UI
- нет поддержки middleware и асинхронных экшнов, нет devtools

Что такое виртуализация и зачем нужна

Виртуализация – позволяет рендерить только те элементы, которые видны на экране.

Рендер предполагает создание множества объектов в DOM и в Virtual DOM, все это нужно хранить в памяти, рендерить в браузере, отслеживать изменения.

это техника для улучшения производительности при рендеринге больших списков данных. При её использовании на экране пользовательского интерфейса отображается только то количество элементов, которое может быть видимо пользователем, а остальные элементы рендерятся по мере необходимости при прокрутке.

React переходы

Функционал React "Переходы" (Transitions) является частью новых особенностей React 18, связанных с конкурентным рендерингом (Concurrent Features). Эти переходы предназначены для улучшения пользовательского опыта и производительности путём управления приоритетами обновлений состояния.

****Зачем нужны Переходы (Transitions):****

1. ****Управление приоритетами****: Переходы позволяют разработчикам указывать меньший приоритет для некоторых обновлений состояния. Это означает, что более важные обновления, которые должны произойти немедленно (например, ввод текста пользователем), могут иметь более высокий приоритет, чем обновления, которые могут подождать (например, загрузка данных).
2. ****Улучшение восприятия производительности****: Переходы позволяют обновлениям, которые не требуют немедленного внимания пользователя, происходить "в фоновом режиме". Это помогает предотвратить нежелательные прерывания в пользовательском опыте, такие как дерганье UI во время ввода текста или при прокрутке страницы.
3. ****Улучшенная стабильность UI****: Во время перехода состояние UI может быть сохранено в стабильном состоянии на время выполнения перехода, что предотвращает визуальный "джанк".

Для использования этого функционала, React предоставляет хук `useTransition`, который позволяет начинать обновления с меньшим приоритетом. Вы можете начать переход с помощью функции, возвращаемой этим хуком, что отложит применение этих обновлений на потом и позволит более важным обновлениям применяться сразу.

Таким образом, "Переходы" (Transitions) являются мощным средством для создания более плавного и отзывчивого интерфейса в приложениях React.

CORS, как работает, зачем нужен

CORS (Cross-Origin Resource Sharing) механизм, который использует дополнительные заголовки HTTP, чтобы дать браузерам указание предоставить веб-приложению, работающему в одном источнике, доступ к ответу на запрос к ресурсам из другого источника. Это браузерная политика. Например чтобы не было возможности у вредоносного сайта делать межсайтовый запрос и получить результат этого запроса, потому что с запросом могут отправиться файлы cookie вашего действующего сеанса на ориг- сайте и авторизационные куки могут попасть на вредоносный сайт.

CORS хедеры

Access-Control-Allow-Origin - заголовок определяет, какие источники могут получать ответ от сервера. Механизм CORS, проверяет совпадение значений заголовка ответа Access-Control-Allow-Origin и заголовка запроса Origin (Источник с которого ушел запрос).

Access-Control-Allow-Credentials заголовок сообщает браузерам, разрешает ли сервер запросам отправлять учетные данные, такие как куки, заголовки авторизации и тд.

Access-Control-Allow-Methods - заголовок указывает, какие HTTP-методы разрешены при доступе к ресурсам.

Способы пофиксить корс ошибку:

Написать бекендеру, чтобы добавил наш домен в список доступных источников, чтобы бэк мог обрабатывать запросы с нашего домена. И тогда в ответе на запрос будет приходить хедер Access- Control-Allow-Origin с нашим адресом

Что такое preflight запросы, http метод OPTIONS

Если запрос не является простым, в случае cross-origin запроса (на другой домен), перед отправкой фактического запроса, делается **preflight OPTIONS** запрос с информацией о запросе: о его методе, дополнительных заголовках и тд. Сервер получает этот запрос и отправляет ответ, содержащий CORS- заголовки. Браузер получает этот ответ и проверяет, будет ли разрешен фактический запрос.

preflight заранее чекает есть ли разрешение на доступ к домену, только после этого позволяет отправить основной запрос.

Какие http коды статусов есть

HTTP статусные коды — это стандартизированные коды ответа, которые серверы используют для сообщения клиенту о результате обработки его запроса. Они разделены на пять классов:

1. ****Информационные (100 - 199)**:**
 - `100 Continue`: Ожидание продолжения запроса.
 - `101 Switching Protocols`: Уведомление об изменении протоколов.
2. ****Успех (200 - 299)**:**
 - `200 OK`: Стандартный ответ для успешных HTTP-запросов.
 - `201 Created`: Запрос успешно выполнен и в результате был создан новый ресурс.
 - `204 No Content`: Запрос успешно выполнен, но содержимое не было найдено.
3. ****Перенаправления (300 - 399)**:**
 - `301 Moved Permanently`: Данный ресурс окончательно перенесен на другой URI.
 - `302 Found`: Временное перенаправление на другой URI.
 - `304 Not Modified`: Данные не изменились с момента последнего обращения.
4. ****Клиентские ошибки (400 - 499)**:**
 - `400 Bad Request`: Сервер не понимает запрос из-за неверного синтаксиса.
 - `401 Unauthorized`: Требуется аутентификация пользователя.
 - `403 Forbidden`: Сервер понял запрос, но отказывается его выполнить.
 - `404 Not Found`: Запрошенный ресурс не был найден.
 - `429 Too Many Requests`: Клиент отправил слишком много запросов за короткий временной интервал.
5. ****Серверные ошибки (500 - 599)**:**
 - `500 Internal Server Error`: Общая ошибка сервера, когда сервер столкнулся с ситуацией, которую он не знает, как обрабатывать.
 - `501 Not Implemented`: Сервер либо не распознает метод запроса, либо не имеет возможности выполнить его.
 - `503 Service Unavailable`: Сервер недоступен, обычно из-за технического обслуживания или перегрузки

Можно отключить CORS-проверки в браузере
Отключить корсы с помощью расширения для браузера, но работает не всегда и не для всех запросов
Настроить прокси сервер, чтобы перенаправлял запросы

REST, какие есть принципы REST

REST (Representational State Transfer) — это архитектурный стиль разработки веб-сервисов, предложенный Роем Филдингом. Основные принципы REST включают:

1. ****Клиент-серверная архитектура****: Разделение ответственности между клиентом и сервером. Сервер предоставляет API и данные, а клиент (например, браузер или мобильное приложение) использует эти API для рендеринга пользовательских интерфейсов.
2. ****Stateless (без сохранения состояния)****: Каждый запрос от клиента к серверу должен содержать всю информацию, необходимую для его выполнения. Сервер не хранит состояние клиента между запросами.
3. ****Cacheable (кэшируемость)****: Ответы должны быть неявно или явно отмечены как кэшируемые или некаэшируемые. Если ответ кэшируемый, клиент может повторно использовать ответ данных для ускорения будущих запросов.
4. ****Uniform Interface (единообразный интерфейс)****: Стандартизированный способ коммуникации между клиентом и сервером. Типично включает использование HTTP-методов (GET, POST, PUT, DELETE и т.д.), URI для адресации ресурсов, медиа-типов для определения формата передачи данных.
5. ****Layered System (слоистая система)****: Клиенту не должно быть известно, общается ли он напрямую с сервером или с посредниками (например, прокси-серверами, балансировщиками нагрузки).
6. ****Code on Demand (опционально)****: Сервер может временно расширять или настраивать функциональность клиента путём передачи ему исполняемого кода (например, JavaScript).

Соблюдение этих принципов позволяет создавать масштабируемые, надежные и удобные в реализации веб-сервисы.

Как работает HTTP, из чего состоит HTTP запрос

HTTP — протокол прикладного уровня передачи данных, изначально предназначенный для передачи гипертекстовых документов. Используется для передачи произвольных данных между клиентом и сервером. HTTP является однонаправленным (запросы идут от клиента к серверу) и работает по принципу запрос — ответ. При каждом запросе браузер устанавливает TCP соединение, получает ответ с сервера и разрывает соединение.

Передача данных по протоколу HTTP осуществляется через TCP/IP- соединения

request содержат: HTTP-метод, определяющее операцию, которую клиент хочет выполнить. Путь к ресурсу url,

Версию HTTP-протокола, Объект request содержит .query .cookies .params .body .headers

response содержат: Версию HTTP-протокола. HTTP код состояния, сообщающий об успешности или причине неудачи. HTTP заголовки\headers. Опционально: body, содержащее пересылаемый ресурс.

Отличие – response имеет статус, а request нет | request имеет метод, а response нет.

Как работает HTTPS и отличие от HTTP

HTTPS — защищённый протокол передачи гипертекста. HTTPS обеспечивает шифрованную связь между клиентом и сервером. расширение протокола HTTP для поддержки шифрования в целях повышения безопасности. Протокол HTTPS шифрует передаваемые сообщения с помощью протокола TLS или уже устаревшего SSL. TLS протокол нужен для подтверждения подлинности сайтов и шифрования сетевого соединения между сайтом и браузером.

HTTP отличия от HTTP 2

В отличие от текстового HTTP протокола, HTTP/2 – бинарный (двоичный), бинарные сообщения обрабатываются быстрее текста. МУЛЬТИПЛЕКСИРОВАНИЕ - позволяет увеличить скорость передачи данных в HTTP2. В HTTP1.1 для каждого запроса требуется устанавливать отдельное TCP-соединение. Мультиплексирование же позволяет браузеру выполнять множество запросов в рамках одного TCP- соединения.

Отличие REST от GraphQL

Различие: в REST структура и объем ресурса определяются сервером. В GraphQL клиент может запрашивать именно те данные, которые ему нужны, указывая в запросе необходимые поля. Все запросы проходят через один endpoint – /graphql, есть система типов, Для запросов создаются mutations, resolvers, query, schema.

Паттерны MVP MVC

MVC – model view controller - это шаблон, который разделяет логику приложения на три части:

Model (модель). Получает данные от контроллера, выполняет необходимые операции и передаёт их в вид.

View (представление). Получает данные от модели и выводит их для пользователя.

Интерфейс. Controller (контроллер). Обрабатывает действия пользователя, проверяет полученные данные и передаёт их модели. Изменить данные, сделать запрос.

Model View Presenter - MVP позволяет ускорить разработку и разделить ответственность разных специалистов; приложение удобнее тестировать и поддерживать.

Model (Модель) работает с данными, проводит вычисления и руководит всеми бизнес-процессами. View (Вид или представление) показывает пользователю интерфейс и данные из модели.

Presenter (Представитель) служит прослойкой между моделью и видом

Основное отличие MVP и MVC в том, что в MVC обновлённая модель сама говорит виду, что нужно показать другие данные. Если же этого не происходит и приложению нужен посредник в виде представителя, то паттерн стоит называть MVP.

Server Sent Events, Polling, Long Polling что такое и как юзается

Polling - периодический опрос сервера, то есть например мы каждые 5сек. делаем запрос за данными. Минусы: задержка между запросами, запросы отправляются каждые 5сек, а не только когда надо, т.е более ресурсоемкий + на каждый запрос устанавливается новое ТСП соединение

Long Polling — это способ, когда сервер получает запрос, но отправляет ответ на него не сразу, а лишь тогда, когда произойдет какое-либо событие либо истечет таймаут соединения. После отправляется еще 1 запрос. Минусы: имеет большую задержку чем WS, более ресурсоемкий, возможно потеря сообщений.

Server Sent Events - позволяет поддерживать HTTP соединение с сервером и получать от него события. В отличие от WebSockets, SSE является односторонним: сообщения отправляются в одном направлении - от сервера к клиенту. Если соединение разрывается, класс EventSource автоматически пытается восстановить его. Стоит юзать, например для уведомлений, или ленты новостей, чтобы обновлять когда появляется новая.

TCP, UDP что такое и где используется, различия

TCP (Протокол Управления Передачей) - протокол сети, который позволяет двум хостам создать соединение и обмениваться потоками данных. Он гарантирует доставку данных в том же порядке, в котором они были отправлены. Чтобы обеспечить гарантию доставки данных, TCP использует подтверждение получения сегментов, при ошибке, если не пришло подтверждения получения и сегмент отправляется повторно.

UDP (User Datagram Protocol), более прост. Для передачи данных ему не обязательно устанавливать соединение между отправителем и получателем. Информация передается без проверки готовности принимающей стороны. Это делает протокол менее надежным – при передаче некоторые фрагменты данных могут теряться. Также, упорядоченность данных не соблюдается, но имеет более высокую скорость передачи данных. Подходит для видео/аудио звонков.

Отличие от HTTP: TCP и UDP - это протоколы транспортного уровня, а HTTP-это протокол прикладного уровня, который работает поверх TCP.

Как работает протокол Websockets

WebSockets - это протокол, позволяющий открыть постоянное двунаправленное сетевое соединение между клиентом и сервером с возможностью обмена данными.

Этот процесс начинается с того, что

клиент отправляет серверу HTTP-запрос с заголовком Upgrade: websocket.

- С WebSocket: браузер единоразово устанавливает TCP соединение для обмена данными. А в REST,

при каждом HTTP-запросе браузер устанавливает TCP соединение, получает данные с сервера и

разрывает соединение.

- WebSocket является двунаправленным, то есть любой клиент / сервер может отправить сообщение

другой стороне, а HTTP является однонаправленным т.к запрос всегда отправляется клиентом, а ответ

обрабатывается сервером

- WebSocket - протокол с отслеживанием состояния, а REST не хранит состояние, т.е. клиенту не нужно

знать о сервере, и то же самое относится и к серверу.

TCP/IP — это стек протоколов, которые задают правила передачи данных по Сети.

TCP (Transmission Control Protocol) отвечает за обмен данными. Он управляет их отправкой и следит за тем, чтобы они дошли до получателя в целости. IP (Internet Protocol) отвечает за передачу пакетов данных между устройствами в сети. Его задача — связывать друг с другом устройства.

Как работает браузер при вводе запроса, этапы рендера

Браузер парсит URL ищет в своём кэше запись о DNS сервере соответствующего IP-адреса. Если ее нет, то запрашивает DNS запись. При нахождении нужного DNS

сервера IP сохраняется в кеше устройства. Запрос содержит имя сервера, который должен быть преобразован в IP-адрес.

Когда IP адрес становится известен, браузер начинает установку соединения к серверу с помощью TCP рукопожатия. Выполняется обмен флагами в 3 этапа: SYN, SYN-ACK и ACK для согласования параметров и подтверждения соединения.

Когда мы установили соединение, браузер отправляет GET запрос для получения HTML файла.

Полученный от сервера HTML-документ браузер преобразует в DOM дерево.

Загружаются и парсятся css-стили, формируется CSSOM (CSS Object Model).

Загружается JS, если при парсинге html встречается script, то он блокирует дальнейший рендер, пока скрипт не отработает

На основе DOM и CSSOM формируется дерево рендеринга, или render tree — набор объектов рендеринга. Render tree дублирует структуру DOM, но сюда не попадают невидимые элементы (например — <head>, или элементы со стилем display:none;). Также, каждая строка текста представлена в дереве рендеринга как отдельный renderer. Каждый объект рендеринга содержит соответствующий ему объект DOM и рассчитанный для этого объекта стиль. Проще говоря, render tree описывает визуальное представление DOM.

Для каждого элемента render tree рассчитывается положение на странице и его размеры, высота, ширина, происходит layout. Браузеры используют поточный метод (flow), при котором в большинстве случаев достаточно одного прохода для размещения всех элементов

Происходит отрисовка каждого отдельного узла в браузере — painting.

Происходит Composition - это единственный этап, который выполняется на GPU.

Конечная отрисовка элементов на странице. Браузер на этом этапе группирует различные элементы по слоям. выполняет только определенные стили CSS, такие как transform и opacity

Есть два свойства которые вызывают задачу composite — это opacity и transform. Эти два свойства являются самыми дешевыми для анимации.

Когда происходит Reflow Repaint

Reflow

Если же изменения затрагивают содержимое, структуру документа, положение элементов — происходит reflow. Причинами таких изменений обычно являются:

Манипуляции с DOM (добавление, удаление, изменение, перестановка элементов);

Изменение содержимого, в т.ч. текста в полях форм;

Расчёт или изменение CSS-свойств; Добавление, удаление таблиц стилей;

Манипуляции с атрибутом «class»; Активация псевдо-классов (например, hover).

Манипуляции с окном браузера — изменения размеров, прокрутка;

Repaint

В случае изменения стилей элемента, не влияющих на его размеры и положение на странице (например, background-color, border-color, visibility), браузер просто отрисовывает его заново, с учётом нового стиля — происходит repaint.

Что такое CSSOM

CSSOM - описывает все селекторы CSS на странице, их иерархию и их свойства. Также стили, которые браузер вставляет по умолчанию. Это набор API-интерфейсов, позволяющих манипулировать CSS из JavaScript

Какие есть способы оптимизации приложений

Оптимизация приложения:

Можем использовать lazy loading и разбиение на чанки, например с помощью React.lazy импортов и Suspense, чтобы у нас код подгружался не весь сразу при первой загрузке приложения, а небольшими частями в тот момент, когда это будет нужно.

Можем использовать webpack минимайзеры css и js файлов (UglifyJsPlugin, Terser, MiniCssExtract), плагин для tree-shaking в webpack, кеширование статических файлов. Tree shaking – это удаление кода и импортов, которые фактически не используются. Можем использовать react.memo, usecallback, usememo для уменьшения количества ререндеров, - например в больших списках

- Использование debounce/throttle, например в фильтрах, инпутах для поиска, чтобы запросы

отправлялись не на каждый ввод в инпут, а с какой-то задержкой

- Использование виртуализации больших списков, чтобы рендерилось например не 1000 элементов, а

только те которые юзер видит на экране

- Оптимизация изображений - использование CDN например s3 для хранения изображений, помогает

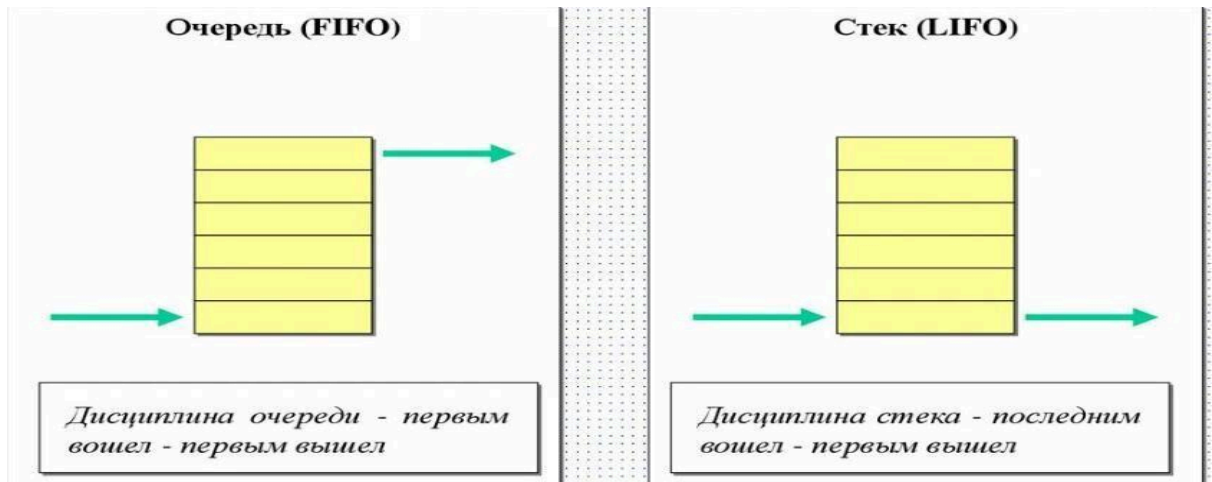
уменьшить время подгрузки, lazy loading images, Progressive Images,

Как ты будешь дебажить приложение, из-за чего бывают утечки памяти

Debug – посмотреть нет ли ошибок в консоли, чекнуть profiler. Утечки памяти могут быть из-за

большого количества eventListeners и setTimeout, так как они остаются в памяти, если мы не сделали

Отличия стека от очереди



Стек и очередь — это две структуры данных, которые позволяют хранить и управлять элементами данных, но они делают это разными способами:

Стек (Stack):

- **LIFO** (Last-In-First-Out): Последний добавленный элемент обрабатывается первым.
- Операции:
 - **push**: добавление элемента на вершину стека.
 - **pop**: удаление и возвращение элемента с вершины стека.
- Пример из реальной жизни: стопка тарелок; последняя положенная тарелка будет первой взятой.

Очередь (Queue):

- **FIFO** (First-In-First-Out): Первый добавленный элемент обрабатывается первым.
- Операции:
 - **enqueue** (или **offer**): добавление элемента в конец очереди.
 - **dequeue** (или **poll**): удаление и возвращение элемента из начала очереди.
- Пример из реальной жизни: очередь в магазине или банке; первый человек в очереди будет обслужен первым.

Вкратце, ключевое отличие между стеком и очередью — порядок, в котором элементы входят и выходят из структуры данных.

Оценка сложности алгоритма, Big O

O(n) — линейная сложность. Например, алгоритм поиска max элемента в не отсортированном

массиву, коллекции это O(n), где n — размер входных данных

O(n²) — квадратичная сложность, например - два вложенных цикла.

O(1) - Константная сложность, одна операция для всех возможных входных данных, например получение элемента массива по индексу - `nums[1]`.

O(log n) - Означает, что сложность алгоритма растёт логарифмически с увеличением входных

он больше искомого, то отбросим вторую половину массива — там его точно нет. Если же меньше, то наоборот — отбросим начальную половину. Быстрее чем O(n).

массиве. Нам надо пройти по всем n элементам массива, чтобы найти максимальный.

Перебор по
данным.

Например — бинарный поиск. Мы делим массив пополам. Проверим средний элемент, если

Big O notation - описывает оценку сложности алгоритма. То есть максимальное количество операций, которое алгоритм может выполнить в худшем случае.

Что такое DNS

DNS — это Система доменных имён которая ведёт список доменных имён вместе с их числовыми IP- адресами или местонахождениями.

1 - Браузер получает запрос от пользователя и направляет его DNS-серверу сети, который ищет совпадение доменного имени и сетевого адреса. Если ответ обнаружен, то страница сайта загружается сразу. В противном случае запрос будет отправлен серверу более высокого уровня или корневому.

Что такое Shadow dom

Теневой DOM Shadow DOM используется для инкапсуляции. Это изолированное DOM дерево со своими дефолтными элементами и стилями к которому нельзя обратиться из главного документа, у него могут быть изолированные CSS-правила и т.д. (например стили для input range). Позволяет избежать ситуаций, когда наши CSS стили вашего сайта может быть по ошибке применена к компонентам внутри Shadow DOM. Можно добавлять свои веб-компоненты.

Что такое jwt

JWT

JWT токен — это строка, которая состоит из трех частей: заголовок header (информация о хешировании), полезные данные payload (например id или роль юзера), и подпись signature. **Чтобы обезопасить** - токены лучше хранить в куках с флагами secure и httponly, не передавать в токенах чувствительные пользовательские данные, ограничить время жизни JWT

Авторизация это

Аутентификация- это процесс проверки учётных данных пользователя

(логин/пароль). Проверка подлинности пользователя путём сравнения введённого им логина/пароля с данными сохранёнными в базе данных.

Авторизация- это проверка прав пользователя на доступ к определенным ресурсам.

Access и refresh токены, что это и где безопасно хранить

access token - используется для авторизации запросов и хранения дополнительной информации о пользователе (user_id, user_role или еще что либо). Более короткоживущий токен.

refresh token - выдается сервером по результатам успешной аутентификации (логин/рег) и используется для получения новой пары **access/refresh** токенов. Имеет большой срок жизни.

Что делать если JWT токен истек

Если JWT токен истек, то для его обновления можно использовать axios-interceptors, redux-middleware, делать проверку в RTK Query FetchBaseQuery, и если нужно перезапрашивать токен.

Cookie что это

HTTP cookie - это фрагмент данных, который сервер отправляет браузеру пользователя. Браузер может сохранить этот фрагмент и отправлять на сервер с каждым последующим запросом. Максимальный размер для браузера – 4 КБ. Чтобы включить отправку кук – withCredentials. Безопаснее хранить токен в куках, с флагами secure+httponly, потому что можно запретить доступ к кук из javascript. Таким образом, если кто-то проведет XSS атаку и сможет выполнить js-код у другого пользователя, он не сможет украсть у него токен, так как к кук нет доступа из JS.

Параметры настройки cookie (secure, httpOnly...)

httpOnly - Эта настройка запрещает любой доступ к куки из JavaScript. Мы не можем видеть такое куки или манипулировать им с помощью document.cookie. Так можно избежать XSS-атак

samesite - определяет, может ли данная кука быть отправлена при кросс-доменном запросе. Значение параметра strict будет предотвращать отправку на другие домены, а lax разрешит отправлять куки с GET-запросами.

secure - указывает, что данная кука может быть передана только при запросах по защищённому протоколу HTTPS. **max-age** и **expires** - определяет время жизни куки.

Domain - хосты на которые отсылаются куки, по умолчанию будет равен хосту сайта.

SessionStorage и LocalStorage + отличия

SessionStorage хранит данные (до 5мб) только во время текущей сессии (для вкладки, пока она открыта). Закрытие вкладки приводит к очищению этих данных. При этом данные сохраняются при обновлении страницы или отображение в этой вкладке другой страницы из этого же источника. В отличие от sessionStorage, localStorage хранит данные в течение неограниченного количества времени. Они сохраняются при закрытии браузера и выключения компьютера.

LocalStorage это свойство объекта window, предназначенное для хранения пар ключ/значение в браузере. В зависимости от браузера, мы можем сохранять до 5 мб данных.

cookies:

- Вместимость - 4кбайта,
- Можем отправить в запросе на бэкенд,
- Можно управлять жизненным циклом в ручную

localStorage:

- Вместимость - 10мбайт,
- Не можем отправить в запросе на бэкенд,
- Информация хранится до тех пор, пока не почистим кэш

браузера

sessionStorage:

- Вместимость - 5мбайт,
- Не можем отправить в запросе на бэкенд

- Информация хранится до тех пор, пока не закроем вкладку или окно браузера**

Что такое простой запрос

Запрос является простым, если он отправлен с помощью методов GET, POST, или HEAD и не содержит дополнительных заголовков. Другой запрос предварительным.

HTTP методы, их отличия и особенности

HTTP методы — это набор определенных операций, поддерживаемых протоколом HTTP, позволяющих выполнять различные действия с ресурсами на сервере. Вот некоторые из наиболее часто используемых HTTP методов, их особенности и отличия:

1. **GET**:

- Используется для получения данных с сервера.
- Должен быть идемпотентен (повторные запросы не изменяют состояние сервера).
- Данные запроса передаются через URL.

2. **POST**:

- Используется для отправки данных на сервер для создания нового ресурса.
- Не является идемпотентным (повторные запросы могут привести к созданию нескольких ресурсов).
- Данные отправляются в теле запроса, что позволяет передавать большие объемы данных.

3. **PUT**:

- Используется для обновления существующего ресурса или создания нового по конкретному URI.
- Является идемпотентным.
- Данные также отправляются в теле запроса.

4. **DELETE**:

- Используется для удаления ресурса по указанному URI.
- Является идемпотентным.

5. **PATCH**:

- Используется для частичного обновления ресурса.
- Может быть не идемпотентным, в зависимости от того, как реализован на сервере.
- Только изменения передаются в теле запроса, а не целиком обновлённый ресурс.

6. **HEAD**:

- Так же, как и GET, используется для получения данных с сервера, но не возвращает тело ответа.
- Идемпотентен.

- Используется для получения метаданных ответа без полного тела ответа.

7. ****OPTIONS****:

- Используется для описания параметров связи с ресурсом, возврат списка поддерживаемых методов сервером.
- Идемпотентен.

Каждый из этих методов предназначен для определенного типа действия в рамках модели запрос-ответ, и их правильное использование обеспечивает соблюдение лучших практик веб-разработки и HTTP-протокола.

Что такое идемпотентность

Идемпотентность - свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом.

Идемпотентный методы - GET, PUT, DELETE, HEAD

Не идемпотентные методы - POST, PATCH**

****Идемпотентность**** — свойство объекта или операции при повторном применении операции к объекту давать тот же результат, что и при первом. Другими словами, если вы выполняете идемпотентную операцию несколько раз, состояние ресурса будет таким же, как при однократном выполнении. В противном случае операция является не идемпотентной

Идемпотентные:

****GET**** - каждый запрос на получение ресурса будет возвращать одинаковый результат

****PUT**** - используется для обновления или создания ресурса, при этом повторное выполнение запроса не приведет к изменению состояния ресурса.

****DELETE**** - используется для удаления ресурса, повторное выполнение запроса не приведет к ошибке, так как ресурс уже удален, но если не УДАЛИТЬ ПОСЛЕДНЮЮ ЗАПИСЬ

****HEAD**** - также является идемпотентным, так как не изменяет состояние ресурса

Не идемпотентные:

****POST**** - НО! может являться идемпотентным, если будет передан ключ, по которому будет проверяться наличие

****PATCH**** - Повторное выполнение PATCH-запроса может изменить состояние ресурса. Например, обновление только определенных полей в профиле пользователя.

Безопасные http методы

****Безопасный метод HTTP**** - это метод, который не изменяет состояние ресурса на сервере и не имеет побочных эффектов на другие ресурсы. Такие методы могут быть

выполнены безопасно, без опасности повреждения данных или нарушения безопасности.

- GET - используется для получения ресурса без его изменения
- HEAD - аналогичен методу GET, но возвращает только заголовки без тела ответа
- OPTIONS - используется для получения информации о возможностях сервера и поддерживаемых методах запросов

Что такое WebRTC

WebRTC (Web Real-Time Communications) - это технология, которая позволяет приложениям захватывать и передавать аудио или видео медиа-поток в браузере.

OWASP уязвимости в браузере

OWASP уязвимости – XSS; инъекции SQL/NOSQL; Межсайтовая подделка запроса (Cross-Site Request Forgery, CSRF); использование deprecated пакетов

Что такое Same Origin Policy

В браузере действует политика безопасности Same Origin Policy. Это означает, что доступ к ресурсам можно получить, только если источник этих ресурсов и источник запроса совпадают. Два URL имеют «одинаковый источник» в том случае, если они имеют совпадающие протокол, домен и порт.

вся функциональная ветка feature окажется поверх главной ветки main, включая в себя все новые коммиты в ветке main

Основные концепции Webpack, loaders, plugins

Webpack - это инструмент сборки JavaScript-приложений, который позволяет управлять и оптимизировать различные ресурсы проекта, такие как JS файлы, CSS, изображения. Для расширения возможностей и оптимизации можем использовать различные вебпак loaders и plugins.

Entry (точка входа) - это файл или набор файлов, с которых Webpack начинает создание своего графа зависимостей. Этот файл является начальной точкой

для вашего приложения, и Webpack использует их для определения всех зависимостей, необходимых для запуска вашего приложения.

Output в Webpack указывает, куда помещать создаваемые им пакеты и как называть эти файлы.

Loaders предназначены для обработки файлов при сборке проекта, т.к **по умолчанию поддерживается только JS и JSON**. Они могут преобразовать файлы из одного формата в другой, например ES6 JS в ES5, компилировать SCSS или LESS в CSS и т.д. То есть лодеры применяются к каждому файлу индивидуально.

Plugins в вебпаке используются для выполнения более широкого спектра задач, которые недоступны с помощью loaders и работают на уровне всего проекта в отличие от лодеров. Они могут выполнять, оптимизацию бандла, генерировать HTML файлов, env переменные и тд.

Отличия webpack modules, chunks, bundle

1. **Module (модуль):** Модуль в Webpack представляет собой отдельный файл JavaScript, CSS, изображение или любой другой ресурс, содержащий код или данные. Модули могут иметь зависимости от других модулей, которые Webpack будет автоматически разрешать и объединять в итоговый bundle. Модули позволяют организовать код проекта на более мелкие и понятные части.
2. **Chunk (кусок):** Chunk – это фрагмент bundle, который содержит набор связанных модулей. Webpack может разбивать бандл на несколько чанков для оптимизации загрузки и кэширования. Чанки могут быть динамически загружаемыми или статическими в зависимости от настроек сборки. Использование чанков помогает уменьшить размер бандла и ускорить загрузку приложения.
3. **Bundle** - это итоговый файл (сборка), в котором комбинируются все модули и чанки проекта. Это один или несколько файлов, содержащих весь необходимый код и ресурсы для работы приложения. Бандл создается Webpack после обработки всех модулей, разрешения зависимостей и оптимизации кода.

Таким образом, модули представляют собой отдельные файлы с кодом, чанки -

фрагменты бандла, содержащие набор модулей, а бандл - это итоговый файл с объединенными модулями и чанками.

Бандл — это приложение. Он родитель **чанков** — частей этого приложения. И все они состоят из

модулей — «атомов», подключаемых через `import / require`, которые мы пишем.

[hash]: это хеш всего пакета. Он генерируется на основе всего содержимого пакета и меняется при любых изменениях в любом из файлов пакета.

[chunkhash]: это хеш фрагмента. Он генерируется на основе содержимого каждого фрагмента и меняется только тогда, когда изменяется содержимое этого фрагмента.

[contenthash]: это хеш содержимого файла. Он генерируется на основе содержимого каждого файла и меняется только тогда, когда изменяется содержимое этого файла.

[name]: это имя файла, которое указывается в конфигурации Webpack. Оно не изменяется при изменении содержимого файла.

В чём отличие ReactDOM.render от ReactDOM.hydrate (или createRoot от hydrateRoot)?

`ReactDOM.render` и `ReactDOM.hydrate` - это две функции из библиотеки React DOM, используемые для рендеринга React-компонентов. Они были частью React до версии 18, когда была представлена новая модель параллельной (concurrent) обработки.

`ReactDOM.render` использовался для начального рендеринга React-компонентов в DOM. Он привязывает React-элементы к DOM-элементу и является точкой входа для React-приложений:

`ReactDOM.hydrate` выполняет ту же основную функцию, но используется для «оживления» (hydration) серверно-отрендеренных компонентов. В среде серверного рендеринга (SSR - Server Side Rendering), первоначальный HTML генерируется на сервере. Клиентский JavaScript затем "оживляет" этот HTML, позволяя прикреплять события и дополнительные интерактивные функции к уже существующим элементам. Для гарантии консистентности между серверным и клиентским рендерингом, `hydrate` обеспечивает привязку событий без перерендеривания содержимого:

С появлением React 18, `createRoot` и `hydrateRoot` стали предпочтительными методами для инициализации приложения и его гидрации:

`createRoot` и `hydrateRoot` в новой конкурентной модели React 18 позволяют использовать функции, такие как параллельная загрузка (concurrent features), что дает дополнительный контроль и лучшую производительность для сложных приложений

Набор директив Content Security Policy (CSP). Зачем они нужны? Приведите примеры директив.

Content Security Policy (CSP) — это дополнительный слой безопасности, который помогает защитить веб-приложения от определенного типа атак, таких как Cross-Site Scripting (XSS) и другие виды инъекций кода. CSP позволяет веб-разработчикам контролировать ресурсы, которые пользовательский агент должен загружать для данной страницы. С помощью политик, заданных в CSP, сайты могут определять, откуда браузер может загружать ресурсы, такие как скрипты, изображения, стили CSS и многое другое.

CSP реализуется на стороне сервера через HTTP заголовок `Content-Security-Policy` или через тег `<meta>` в HTML документе.

****Примеры директив CSP:****

- ****default-src****: Устанавливает политику источника по умолчанию для загрузки ресурсов, таких как скрипты, изображения или стили. Например:

```
```http
Content-Security-Policy: default-src 'self'
```
```

Это означает, что все ресурсы должны загружаться из того же источника, что и сам документ.

- ****script-src****: Задает допустимые источники для загрузки скриптов. Например:

```
```http
Content-Security-Policy: script-src 'self' https://apis.google.com
```
```

Это указывает браузеру загружать скрипты только с того же домена (самодостаточно) и с домена `apis.google.com`.

- ****img-src****: Определяет, откуда можно загружать изображения. Например:

```
```http
Content-Security-Policy: img-src 'self' https://images.example.com
```
```

Изображения могут загружаться только с текущего домена и с `images.example.com`.

- **style-src**: Устанавливает политику для загрузки стилей. Например:

```
```http
Content-Security-Policy: style-src 'self' 'unsafe-inline'
```
```

Это разрешает загрузку стилей из источника страницы и использование встроенных стилей ('unsafe-inline' — использование этой опции не рекомендуется из-за риска XSS атак).

- **object-src**: Задаёт источники для загрузки плагинов, таких как ``, `` и ``. Например:

```
```http
Content-Security-Policy: object-src 'none'
```
```

Это полностью запрещает загрузку плагинов.

- **connect-src**: Ограничивает источники, к которым может быть установлено соединение через интерфейсы, такие как ``, `fetch`, `XMLHttpRequest`, `WebSocket`. Например:

```
```http
Content-Security-Policy: connect-src 'self' https://api.example.com
```
```

CSP служит хорошим дополнением к стандартным практикам безопасности и помогает защитить ваши веб-приложения от множества угроз. Правильная настройка CSP может значительно снизить риск выполнения вредоносного кода на вашем сайте.

Опишите принцип работы http заголовка If-Modified-Since

HTTP заголовок `If-Modified-Since` используется для условного запроса ресурса со стороны клиента. Этот заголовок позволяет клиенту запросить ресурс только в том случае, если он был изменен после указанной даты и времени. Принцип работы заголовка `If-Modified-Since` участвует в механизме кеширования и помогает уменьшить количество передаваемых данных и загрузку на сервер, предотвращая повторную загрузку ресурса, который не изменился.

Как работает `If-Modified-Since`:

1. **Первоначальный запрос:** Когда клиент (например, веб-браузер) впервые запрашивает ресурс, сервер отвечает, возвращая содержимое ресурса вместе с HTTP заголовком `Last-Modified`, который указывает дату и время последнего изменения ресурса.

2. ****Повторный запрос:**** При следующем запросе того же ресурса клиент может включить заголовок `If-Modified-Since`, устанавливая в нем дату и время из заголовка `Last-Modified`, полученного ранее. Этим клиент сообщает серверу: "Отправь мне ресурс только в том случае, если он был изменен после этой даты".

3. ****Ответ сервера:****

- Если ресурс был изменен после указанной даты в `If-Modified-Since`, сервер отправит новую версию ресурса с кодом состояния `200 (OK)`.
- Если ресурс не изменился, сервер ответит кодом `304 (Not Modified)` и не включит содержимое ресурса в ответ, тем самым сигнализируя клиенту использовать версию ресурса из кеша.

****Примеры использования:****

1. ****Первоначальный запрос без `If-Modified-Since`:****

```
```http
GET /image.png HTTP/1.1
Host: example.com
```
```

2. ****Ответ сервера с `Last-Modified`:****

```
```http
HTTP/1.1 200 OK
Last-Modified: Wed, 21 Oct 2015 07:28:00 GMT
Content-Type: image/png
(бинарные данные изображения)
```
```

3. ****Повторный запрос с `If-Modified-Since`:****

```
```http
GET /image.png HTTP/1.1
Host: example.com
If-Modified-Since: Wed, 21 Oct 2015 07:28:00 GMT
```
```

4. ****Ответ сервера, если ресурс не изменился:****

```
```http
HTTP/1.1 304 Not Modified
```
```

Использование заголовка `If-Modified-Since` способствует оптимизации веб-трафика и ускорению загрузки веб-страниц за счет уменьшения необходимости повторной передачи неизменившихся ресурсов.

Что такое core web vitails

Core WEb Vitals - это набор метрик от компании Google, которые оценивают скорость загрузки, интерактивность, визуальную стабильность.

1. LCP - Largest Contentful Paint - Отрисовка самого крупного контента
2. FID - First Input Delay - Задержка после первого действия
3. CLS - Cumulative Layout Shift - Неожиданное изменение макета.

Core Web Vitals - это набор метрик, которые используются Google для оценки качества пользовательского опыта на веб-сайтах. Эти метрики включают в себя:

1. ****Largest Contentful Paint (LCP)**** - время загрузки самого большого контентного элемента на странице, такого как изображение или текст. Это значение должно составлять менее 2,5 сек с начала скачивания страницы.
2. ****First Input Delay (FID)**** - время задержки между первым взаимодействием пользователя с сайтом и реакцией на это действие. Это значение должно составлять менее 100 мс.
3. ****Cumulative Layout Shift (CLS)**** - мера стабильности макета сайта и отсутствия неожиданных перемещений элементов при загрузке страницы.

Google рекомендует, чтобы все три метрики Core Web Vitals были удовлетворены на сайте, чтобы улучшить пользовательский опыт и рейтинг в поисковой выдаче.

Что выполняет команда npm ci

npm ci - это команда, которая удаляет все зависимости из node_modules и устанавливает зависимости из файла package.lock.json точно соблюдая их версии

Git merge vs rebase

Обе команды предназначены для включения изменений из одной ветки в другую.

merge создает в ветке новый «коммит слияния», связывающий истории обеих веток.

Существующие ветки никак не изменяются.

rebase —(берет коммиты из 1 ветки и перемещает на

2) эта команда перезаписывает историю проекта, создавая новые коммиты для каждого коммита в исходной ветке. В результате `git checkout feature => git rebase main`

Что такое подмодули (submodules) в гите и для чего они нужны?

Подмодули (submodules) в Git позволяют вам включить и управлять зависимыми репозиториями внутри вашего основного Git репозитория. Этот механизм используется для того, чтобы сохранять исходный код отдельных библиотек или компонентов в своих собственных репозиториях, а затем добавлять их к вашему проекту как подмодули.

****Для чего нужны подмодули в Git:****

1. ****Управление зависимостями****: Подмодули позволяют вам легко интегрировать и менеджерить внешние зависимости или библиотеки. Это особенно удобно для больших проектов, где разные части могут разрабатываться независимо.
2. ****Отдельное версионирование****: Каждый подмодуль отслеживается как отдельный Git репозиторий, что позволяет ему иметь собственную историю коммитов и версии. Это значит, что вы можете контролировать, какая версия внешнего компонента или библиотеки используется в вашем проекте.
3. ****Обновление и поддержка****: Вы можете легко обновлять подмодули до последних версий или конкретных тэгов, не изменяя основной код вашего проекта.
4. ****Совместная работа****: Подмодули помогают в организации совместной работы. Разные команды могут работать на разных репозиториях, но их изменения могут быть легко интегрированы в основной проект как подмодули.

****Как добавить подмодуль:****

Чтобы добавить подмодуль в ваш Git репозиторий, используйте следующую команду:

...

```
git submodule add <repository-url> <path-to-submodule-directory>
```

Несмотря на их полезность, подмодули также имеют сложности в управлении, особенно в больших проектах, такие как сложность обновления и требования к дополнительным шагам при клонировании репозитория. Поэтому применение подмодулей требует внимательного планирования и управления.