[**Doc-style**](#_cxd4jn1efd) **4**

[**Общая информация**](#_jyremwjs5yt5) **4**

[**Готовность билетов**](#_vgd6stfbpm7a) **4**

[**Билет №1 (Крамер Константин, Ибрагимов Эмиль)**](#_i8wk0etlrqod) **5**

[Типы узлов в DOM; атрибуты узлов для навигации по братьям, детям, родителям](#_njp6me5y0piy) 5

[**Билет №2 (Крамер Константин, Ибрагимов Эмиль)**](#_uar23a9ndnoz) **10**

[Методы поиска узлов в DOM; атрибуты текстового содержимого узла; работа с атрибутами и классами](#_56vjiduagkr5) 10

[**Билет №3 (Крамер Константин, Ибрагимов Эмиль)**](#_eh78ldsvzbbo) **14**

[Методы модификации DOM-дерева; добавление и удаление событий](#_e2fqw2kou8gj) 14

[**Билет №4 (Аркадий Сюрис)**](#_wm3vd21sy1y5) **17**

[Захват и всплытие события. Паттерны работы с событиями](#_pgr5v2fq4rc3) 17

[Ссылки:](#_dbvtqgmd4xex) 17

[**Всплытие и погружение:**](#_5n6pnlsgabka) **17**

[**Паттерны:**](#_7ao9o2npz1nv) **19**

[Делегирование:](#_whp3oot2c9ns) 19

[Debounce:](#_e58g98wvj0uw) 19

[**Билет №5 (Кочетков Никита, Тая Пенская)**](#_dsqvtm4bekw0) **20**

[Функциональные и классовые компоненты в React. Жизненный цикл компонента](#_36y0k4frntbo) 20

[Ссылки](#_dfgzs1ek760a) 20

[Компоненты](#_rc6ma6ugkplx) 20

[Жизненный цикл компонента](#_jd1ouq8atwmn) 21

[**Билет №6 (Кочетков Никита, Тая Пенская)**](#_pe9gxa78swcc) **24**

[Props, Context, State в React - способы использования, различия](#_2ir832u06pps) 24

[Ссылки:](#_3fxnvwb8k8ts) 24

[Пропсы](#_a8bq0itk0coi) 24

[Состояние](#_frlnfk8cdhgz) 25

[Контекст](#_cofqs2ij2u6j) 26

[Контекст позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на промежуточных уровнях.](#_bs1reid4wo00) 26

[Контекст разработан для передачи данных, которые можно назвать «глобальными» для всего дерева React-компонентов (например, текущий аутентифицированный пользователь, UI-тема или выбранный язык).](#_tncjxl2zocly) 26

[React.createContext](#_mwfg55wtohcr) 26

[Context.Provider](#_gocltaqatryd) 26

[Class.contextType](#_cidpxroyh6lk) 26

[Context.Consumer](#_l60mhqcslg1q) 26

[Context.displayName](#_6lbhw3pb19o0) 27

[**Билет №7 (Лапшина Вероника)**](#_biwpeu5nzvht) **28**

[Протокол HTTP. Формат, механики, проблемы](#_3l8b4tz3tt26) 28

[**Билет №8 (Лапшина Вероника)**](#_wb4pkruxraff) **33**

[REST. Принципы, рекомендации, проблемы](#_pd3icug78dsj) 33

[**Билет №9 (Али, Диана)**](#_bkijziaz99ia) **38**

[React Hooks. Стандартные хуки, их применение](#_m3ng5wm6pujx) 38

[**Билет №10 (Донат, Ксюша)**](#_qnvu8vxm6rdz) **41**

[React. Повторное использование логики. Компоненты высшего порядка. Как работают Render Props.](#_8obl0ltem99c) 41

[React.](#_ntlody69rxnt) 41

[React - это декларативная, эффективная и гибкая библиотека JavaScript для создания пользовательских интерфейсов (UI). Она позволяет вам создавать сложные UI из небольших и изолированных частей кода, называемых «компонентами».](#_6kz06n3koi5q) 41

[React дает вам язык шаблонов и некоторые callback-функции для отрисовки HTML. Весь результат работы React — это HTML. Ваши связки HTML/JavaScript, называемые компонентами, занимаются тем, что хранят свое внутреннее состояние в памяти (например: какая закладка выбрана), но в итоге вам просто выплевывается HTML. Разумеется, вы не можете построить полно функционирующее динамическое приложение только с React.](#_wltci1tk0d9q) 41

[Особенности.](#_jv6cqpol9r67) 41

[HOC(Higher-Order Components) - принимают компонент и возвращают тот же компонент, но с дополнительной функциональностью.  
Функция высшего порядка - функция, которая принимает функцию в качестве аргумента или возвращает функцию в качестве результата](#_ko6w3rznv7) 44

[>>>Пример из презентации](#_lpxn2xkj4ui8) 44

[**Билет №11 (Стас, Ерофей)**](#_jy9peoap33bp) **46**

[Redux. Нарисовать классическую flux-архитектуру. Нарисовать flux-архитектуру в Redux. Объяснить преимущества flux-архитектуры перед MV\*. Объяснить преимущества redux перед классическим flux.](#_whq2w5cebl3q) 46

[MV-паттерн (Model-View)](#_gi9xcmx5sqwd) 46

[MVVM-паттерн (Model-View-ViewModel)](#_9a7ttvdidvs6) 47

[Flux-архитектура](#_2vv6rs8cenu) 47

[Redux](#_wp52vs58ft4w) 48

[Послесловие](#_oltifk7awfmm) 48

[**Билет №12 (Никита)**](#_g2bdj0mjs925) **49**

[Redux. Какие существуют проблемы с асинхронностью в redux и как их решить. Рассказать про одну из популярных библиотек (thunk, promise, saga или observable).](#_bgzef0fr6hs7) 49

[redux-thunk: добавление миддлвары](#_pt146vthupms) 49

[redux-saga: добавляем отдельный тред](#_xec3d4cfcthz) 50

[redux-promise: как redux-thunk, только для Promise](#_j6j312pu6tqc) 50

[redux-observable: самая сложная библиотека](#_9nl29l4qamo7) 51

[**Билет №13 (Рома, Саша, Галя)**](#_6upvaxx0c0y9) **53**

[Безопасность. Логические уязвимости. Примеры и предпосылки](#_wjftbapwxigp) 53

[Угроза-](#_994d7ppe057u) 53

[**Билет №14 (Рома, Саша, Галя)**](#_trcyihyj1as5) **55**

[Безопасность. Инъекции. Виды и способы борьбы с ними.](#_9zj4nqfdf828) 55

[**Билет №15 (Рома, Саша, Галя)**](#_1kkiw6fqwivt) **59**

[Безопасность. CORS. СSP. Как обезопасить работу с Cookie.](#_qs4x7mf9qvmn) 59

[**Билет №16 (Соня, Артемий)**](#_xybujomouywy) **63**

[Процесс тестирования](#_etztm85ty4yh) 63

[Testdesign](#_6d26dcbshmww) 64

[Что такое тест-кейс, примеры тест-кейсов.](#_s1nbpedh8qx0) 66

[BugReports, примеры багрепорта.](#_3oddt5humspv) 67

[Виды тестирования](#_2rjb8avxpf7m) 68

[**Билет №17 (Костя)**](#_eqilwdeku334) **72**

[Оптимизация загрузки страницы. Ключевые моменты, способы оптимизации. Инструменты для измерения скорости загрузки, измеряемые показатели.](#_4ogzwd5ybjlt) 72

[Зачем оптимизировать производительность](#_urnupgdeytvn) 72

[RAIL — performance model](#_qq2mxh93766n) 72

[Воспринимаемая производительность](#_bgtdm23f1b3v) 72

[Optimistic UI](#_53p6swdqs44d) 73

[Измерение производительности — критерии](#_9nqiu9t22id2) 73

[Инструменты](#_9clm0lxesr5z) 74

[Оптимизация](#_njqcy6rs3c13) 75

[**Билет №18 (Михаил)**](#_nujcqof7r0in) **78**

[Стратегии рендеринга. Отличия и преимущества. Профилирование компонентов.](#_nu37vavp9p68) 78

[Server-side rendering (SSR)](#_1c1z92lqnrdi) 78

[Static rendering](#_yix9tudkpb5s) 79

[Client-side rendering (CSR)](#_wps2i96n2y6f) 79

[SSR with rehydration](#_6hn3am7ntlwc) 80

[**Билет №19 (Дмитрий)**](#_hmaqecbx8k4x) **82**

[Cookies, web storage, IndexedDB. Особенности, различия](#_s7xplq59o23h) 82

[Cookies:](#_wjihjh7ln42) 82

[WebStorage:](#_lrwmgizdflcd) 84

[IndexedDB:](#_n6duvl5wgopj) 85

[Различия:](#_qh2jp23oj9a) 89

# 

# Общая информация

[Ссылка на GitLab](https://gitlab.com/yandex-courses/frontend/ifmo/2019-2020/webdev/general) (записи лекций там же в таблице)

[Презентации](https://frontend-ifmo-2019.now.sh/webdev)

[Если что-то не осилил, то вот в помощь](http://neerc.ifmo.ru/lgd.pdf)

# Билет №1

## Типы узлов в DOM; атрибуты узлов для навигации по братьям, детям, родителям

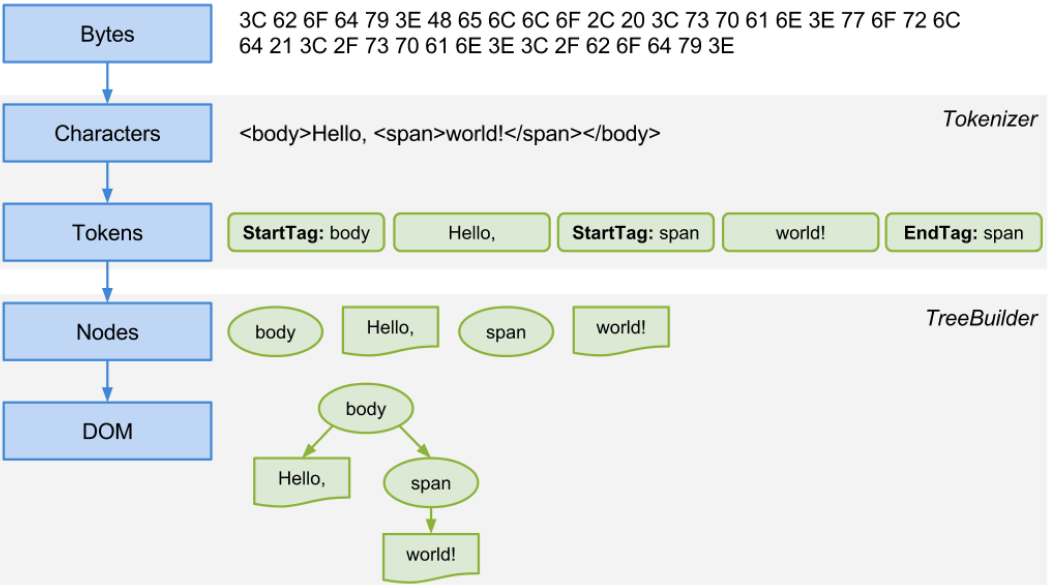
**Вся презентация:** <https://urfu-2017.github.io/webdev-slides/06-browser-and-js/lection/index.html#/>

**Слайды по теме:**

<https://urfu-2017.github.io/webdev-slides/06-browser-and-js/lection/index.html#/3>

<https://javascript.ru/tutorial/dom/intro> - классная статья с примерами

**DOM** (Document Object Model) - это программный интерфейс для работы с XML- и HTML-документами. Это дерево, узлами которого могут быть атрибуты, элементы, свойства и многое другое.



В DOM 12 типов узлов, но обычно все работают с 4:  
1) document - “входная точка” в DOM

2) узлы-элементы - HTML-теги, основные строительные блоки

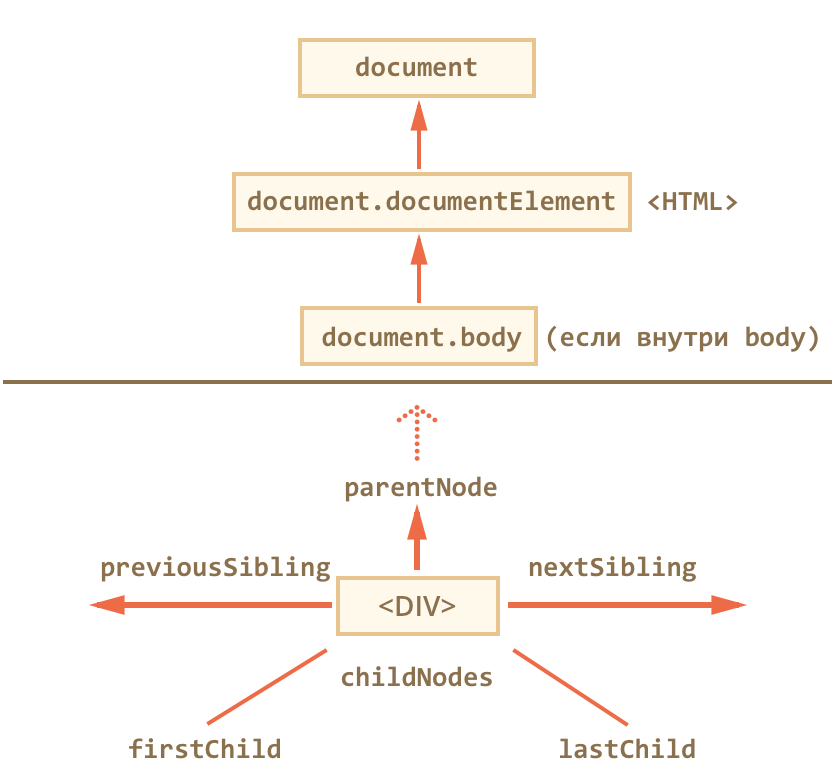
3) текстовые узлы - содержат текст

4) Комментарии - в них можно включить информацию, которая будет доступна из JS

**Навигация по узлам:**

DOM позволяет нам делать что угодно с элементами и их содержимым, но для начала нужно получить соответствующий DOM-объект.

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



**Самые верхние элементы дерева доступны как свойства объекта document**:

1. <html> = document.**documentElement -** самый верхний узел документа.

В DOM он соответствует тегу <html>.

1. <body> = document.**body**

Другой часто используемый DOM-узел - узел тега <body>.

1. <head> = document.**head**

Тег <head> доступен как document.head.

**null**

Есть одна тонкость: document.body может быть равен null:

Нельзя получить доступ к элементу, которого еще не существует в момент выполнения скрипта.

**Дети:**

1. **childNodes** - коллекция, содержит список всех детей, включая текстовые узлы.
2. **firstChild** - обеспечивает быстрый доступ к первому дочернему элементу.
3. **lastChild** - обеспечивает быстрый доступ к последнему дочернему элементу.

(childNodes похож на массив, но это коллекция - особый перебираемый объект - псевдомассив)

**Соседи и родители:**

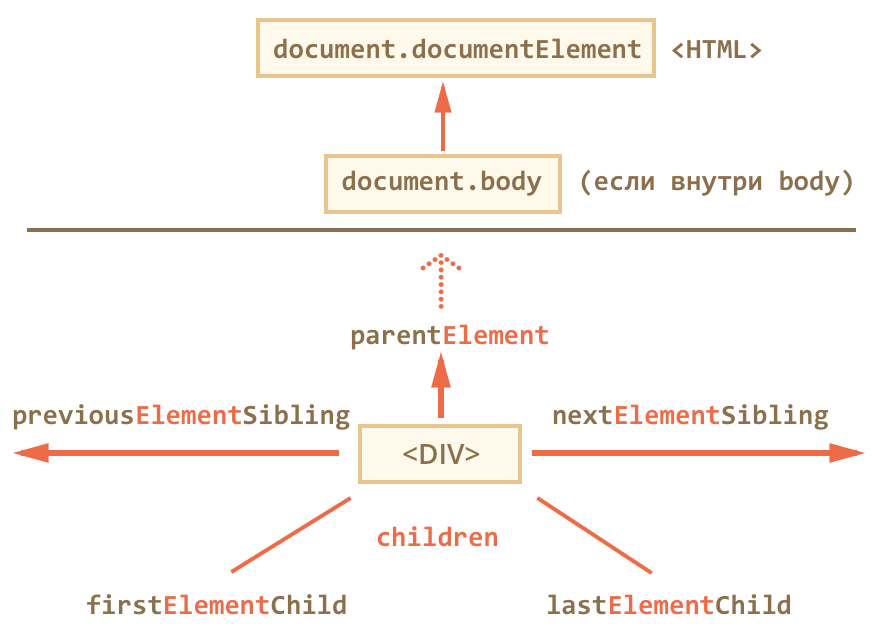
1. **parentNode** - родитель узла.
2. **nextSibling** - следующий узел того же родителя.
3. **previousSibling** - предыдущий узел того же родителя.

**Навигация только по элементам:**

Для большинства задач текстовые узлы и узлы-комментарии нам не нужны. Мы хотим манипулировать узлами-элементами, которые представляют собой теги и формируют структуру страницы.

Давайте рассмотрим дополнительный набор ссылок, которые учитывают только узлы-элементы:

1. **children** - коллекция детей, которые являются элементами.
2. **firstElementChild** - первый дочерний элемент.
3. **lastElementChild** - последний дочерний элемент.
4. **previousElementSibling** - предыдущий элемент того же родителя.
5. **nextElementSibling** - следующий элемент того же родителя.
6. **parentElement** - родитель-элемент.



**Замечание:**

**parentElement** и **parentNode** обычно одинаковы, за исключением:

document.documentElement

alert( document.documentElement.parentNode); // document

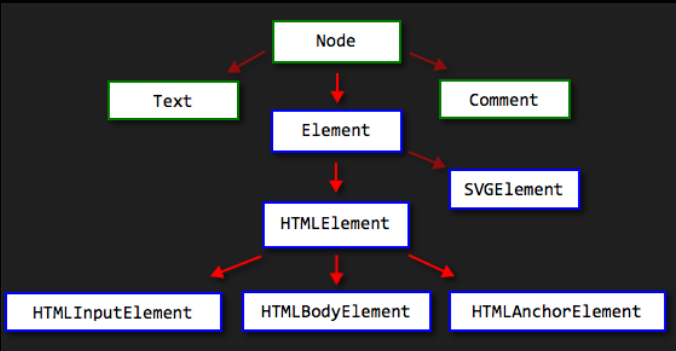
alert( document.documentElement.parentElement); // null

Причина - document не узел-элемент.

(Некоторые виды DOM-элементов, например таблицы, предоставляют дополнительные ссылки и коллекции для доступа к своему содержимому.)

**Свойства узлов:**

Каждый DOM-узел принадлежит соответствующему встроенному классу. Корнем иерархии является EventTarget, от него наследует Node и остальные DOM-узлы.



**Классы:**

1. **EventTarget** - «абстрактный» класс. Объекты этого класса никогда не создаются. Он служит основой, благодаря которой все DOM-узлы поддерживают так называемые «события».
2. **Node** - также является «абстрактным» классом, и служит основой для DOM-узлов. Он обеспечивает базовую функциональность: parentNode, nextSibling, childNodes и т.д. (это геттеры). Объекты класса Node никогда не создаются. Но есть определённые классы узлов, которые наследуют от него:
3. **Text** - для текстовых узлов.
4. **Element** - для узлов-элементов.
5. **Comment** - для узлов-комментариев.
6. **Element** - это базовый класс для DOM-элементов. Он обеспечивает навигацию на уровне элементов: nextElementSibling, children и методы поиска: getElementsByTagName, querySelector. Браузер поддерживает не только HTML, но также XML и SVG. Класс Element служит базой для классов: SVGElement, XMLElement и HTMLElement.
7. HTMLElement - является базовым классом для всех остальных HTML-элементов. От него наследуют конкретные элементы:
8. HTMLInputElement - класс для тега <input>
9. HTMLBodyElement - класс для тега <body>
10. HTMLAnchorElement - класс для тега <a>
11. и т.д.

**Свойства:**

1. node.**nodeType -** для того, чтобы узнать тип узла. Возвращает число:
2. 1 == ELEMENT\_NODE
3. 2 == ATTRIBUTE\_NODE
4. 3 == TEXT\_NODE
5. 4 == CDATA\_SECTION\_NODE
6. 5 == ENTITY\_REFERENCE\_NODE
7. 6 == ENTITY\_NODE
8. 7 == PROCESSING\_INSTRUCTION\_NODE
9. 8 == COMMENT\_NODE
10. 9 == DOCUMENT\_NODE
11. 10 == DOCUMENT\_TYPE\_NODE
12. 11 == DOCUMENT\_FRAGMENT\_NODE
13. 12 == NOTATION\_NODE
14. node.**nodeName** и node.**tagName**
15. Оба свойства содержат имя тега
16. Свойство tagName есть только у узлов-элементов (поскольку они происходят от класса Element), а nodeName определено для любых узлов Node.
17. **innerHTML**
18. Позволяет получить HTML-содержимое элемента в виде строки.
19. Мы можем изменять его.
20. “innerHTML +=” осуществляет перезапись.
21. Скрипты не выполняются. Если innerHTML вставляет в документ тег <script> - он становится частью HTML, но не запускается.
22. **outerHTML**
23. Содержит HTML элемента целиком.
24. Доступно на запись, но в результате получается новый элемент.
25. **nodeValue / data**

Свойство innerHTML есть только у узлов-элементов, у других есть аналоги.

1. **textContent**
2. Содержит только текст внутри элемента, за вычетом всех <тегов>.
3. **hidden**
4. Виден ли элемент.
5. Работает как для CSS, так и для соответствующего атрибута.

В зависимости от своего класса DOM-узлы имеют и другие свойства. Например у элементов <input> (HTMLInputElement) есть свойства value, type, у элементов <a> (HTMLAnchorElement) есть href и т.д. Большинство стандартных HTML-атрибутов имеют соответствующие свойства DOM.

# Билет №2

## Методы поиска узлов в DOM; атрибуты текстового содержимого узла; работа с атрибутами и классами

**Вся презентация:** <https://urfu-2017.github.io/webdev-slides/06-browser-and-js/lection/index.html#/>

**Слайды по теме:**

<https://urfu-2017.github.io/webdev-slides/06-browser-and-js/lection/index.html#/4>

Работа с DOM-деревом всегда начинается с поиска элементов.

Поиск элемента:

1. document.**getElementById**(‘id’)

Ищет элемент с заданным идентификатором. Возвращает первый найденный элемент. Объект, который возвращается нам, имеет класс HTMLFormElement.   
Метод getElementById всегда возвращает ровно один элемент, так как по спецификации элемент не может быть повторен с одинаковым идентификатором на странице. И в DOM-дереве всегда должен быть ровно один элемент с одним идентификатором. Как следствие, этот метод самый быстрый.

1. **getElementsByClassName**(class)
2. Есть не только у document, но и у любого элемента.
3. Ищет элементы с соответствующим классом.
4. Возвращает коллекцию.
5. document.getElementsByClassName(‘class class2’)

не вернет

<div class=”class class2”>

1. **getElementsByTagName**(‘tag’)
2. Есть не только у document, но и у любого элемента.
3. Ищет элементы соответствующего тега.
4. Возвращает (коллекцию) объект класса HTMLCollection.

Этот метод достаточно устаревший и не очень удобный для применения, так как вам редко нужно находить все теги какого-то вида на странице.

На смену устаревшим методам для поиска были добавлены более современные, и прежде всего это метод querySelector, который позволяет искать внутри DOM-дерева элементы в зависимости от CSS-селектора.

Механизм поиска элементов по CSS-селектору очень гибкий, так как с помощью различных CSS правил их комбинаций вы можете задавать очень сложные правила для поиска.

Например: document.querySelector(‘[name=”login”]’) // [object HTMLInputElement]

(в данном случае мы находим все элементы, у которого есть атрибут name, значение которого - login)

Поиск по идентификатору: document.querySelector(‘#auth’).

Поиск по классу: document.queryselector(‘.auth’).

jQuery:

1. **querySelectorAll**(selector)
2. Возвращает коллекцию по заданному CSS селектору.

([object NodeList])

1. Псевдоселекторы тоже работают, т.е. :hover вернёт всё дерево элементов на которые мы навелись.
2. **querySelector**(selector)
3. Возвращает первый найденный элемент по заданному селектору.
4. document.querySelector(‘.class.class2’)

вернет

<div class=”class class2”>

1. **matches**(selector)
2. Проверяет соответствует ли текущий элемент селектору.
3. Возвращает true или false
4. **closest**(selector)
5. Ищет ближайший элемент, соответствующий селектору вверх по DOM - дереву относительно текущего.

Для **итерирования по коллекциям**, и HTMLCollection, и ModeList самый простой способ - **цикл for**. И тот и другой являются подобиями массива. И у того и у другого есть свойство **len**, которое говорит о количестве элементов внутри коллекции. И к любому элементу внутри коллекции можно обратиться по индексу.

var collection = document.querySelectorAll(‘input,button’);

for (var i = 0, len = collection.length; i < len; i++) {

var elem = collection[i];

}

Другой вариант - с помощью **метода Array.prototype.slice** преобразовать нашу коллекцию к настоящему массиву и уже воспользоваться всеми методами, которые есть у массива.

var elems = document.querySelectorAll(‘input,button’);

var elemsList = Array.prototype.slice.call(elems);

elemsList.forEach(function(elem) {…})

И последний вариант, который предоставляют нам современные браузеры - метод **Array.from**, который на вход получает что-то, подобное массиву, и преобразует это в полноценный, настоящий массив. В нашем случае мы получаем массив с элементами и уже можем применить любые методы для работы с массивами к ним.

var elemsList = Array.from(document.querySelectorAll(‘input,button’));

elemsList.forEach(function(elem) {...});

<https://learn.javascript.ru/searching-elements-internals>

**Атрибуты**:

1. **element.[get/has/set/remove]Attribute**
2. **getAttribute** - в качестве параметра он получает имя атрибута, который мы хотим, к которому мы хотим обратиться, а в качестве возвращаемого значения в виде строки - содержимое этого атрибута.
3. **hasAttribute** - проверяет наличие атрибута, и метод **setAttribute**, который устанавливает значение атрибута. При повторном вызове метод has уже вернет нам true, т.к. новый атрибут уже был добавлен в DOM-дерево.
4. **removeAttribute** - метод для удаления атрибута по его имени.
5. **attributes** - коллекция всех атрибутов элемента

**Свойства**:

У всех элементов есть как атрибуты, так и свойства. И порой это совершенно разные вещи.



Порой атрибуты и свойства ведут себя очень предсказуемо, и значения у них могут сильно отличаться. Поэтому лучше использовать атрибуты в большинстве случаев вместо свойств, а если вы хотите пользоваться свойствами, то использовать самые простые и общие свойства: id и className.

**Классы**:

**Атрибут class** - один из самых широко используемых атрибутов в html. С помощью него вы можете помечать элементы для того, чтобы наложить на них css-свойства и настроить их визуальное отображение, для того чтобы выбрать их с помощью селектора, и вообще объединить в какие-то логические связанные группы.

Для работы с атрибутом class есть свойство className:

**className** - в строковом представлении содержит значение атрибута class в html и представляет весь набор классов, который указан у этого html-элемента в виде строки.

Для того чтобы отредактировать набор классов: убрать класс, добавить класс, вам необходимо устанавливать это свойство и заменять строчку уже существующую на новую с помощью конкатенации, с помощью регулярных выражений или с помощью замены внутри строки. Это усложняет и поиск, и выборку, и проверку наличия класса.

Поэтому был создан новый, достаточно современный метод **classList**, который содержит набор методов для работы с классами.

1. **classList** - объект для работы с классами
2. **classList.[add/remove/toggle/contains/item]**
3. **add** - добавляет класс к списку уже существующих
4. **item** - получает класс по его индексу и по порядковому расположению его в html-элементе
5. **contains** - позволяет по имени класса проверить, есть ли он внутри этого элемента
6. **remove** - удалить класс из html-элемента
7. **toggle** - если класс у элемента отсутствует - добавляет, иначе - убирает. Когда вторым параметром передано false - удаляет указанный класс, а если true - добавляет.

Если вторым параметром передан undefined или переменная с typeof == 'undefined', поведение будет аналогичным передаче только первого параметра при вызове toggle.

**data-\***

1. Все **data-\*** атрибуты доступны в объекте dataset.

Существует специальный объект **dataset**, который есть внутри любого DOM-элемента, который содержит значения всех data-атрибутов, которые есть внутри этого элемента. Он преобразовывает имя атрибута, убирает от него префикс data-, а все дефисы и последующие буквы преобразуют к CamelCase. Таким образом имя атрибута становится валидным идентификатором для переменной в JavaScript, и мы можем обращаться к нему через так называемую дот-нотацию. И мы можем с помощью этого как бы обратиться к атрибуту на чтение, так и установить новое значение атрибута, которое будет прописано в DOM-дереве уже в развернутом data- виде.

1. **data-user-location** будет доступно в element.dataset.userLocation

# Билет №3

## Методы модификации DOM-дерева; добавление и удаление событий

**Вся презентация:** <https://urfu-2017.github.io/webdev-slides/06-browser-and-js/lection/index.html#/>

**Слайды по теме:**

<https://urfu-2017.github.io/webdev-slides/06-browser-and-js/lection/index.html#/6>

Добавление и удаление узлов:

Добавление:

1. document.**createElement** - отвечает за создание элемента.

Созданный элемент находится в памяти и не находится внутри тела страницы (не связан с DOM-деревом).

1. **appendChild** - добавляет элемент внутрь нашей страницы.

С помощью appendChild мы можем добавить наш элемент к любому другому, как вложенный новый дочерний элемент.

1. **insertBefore** - добавляет элемент в список дочерних элементов родителя перед указанным элементом. Если указанный элемент не задан или равен null, вставляет в конец списка.

Удаление:

1. parentElement.**removeChild**(elem) - удаляет elem из детей parentElem.
2. elem.**remove**() - то же самое, но не поддерживается IE 11-.
3. parentElem.**replaceChild**(newElem, elem) - удаляет elem из детей parentElem и вставляет на его место newElem.
4. classList.**contains** - есть ли класс.

elem.**cloneNode:**

При создании нового DOM-элемента нам не всегда нужно создавать его с нуля: можно взять за основу уже существующий, который есть на странице, скопировать его и поменять в нем какие-то свойства. Для этого существует метод cloneNode, который принимает в качестве необязательного параметра true или false. False - значение по умолчанию, а если мы передадим true, то все дочерние элементы внутри нашего элемента, который мы хотим склонировать, будут скопированы. Все это будет помещено во вновь созданный склонированный объект.

Коротко:

1. Создает копию текущего элемента.
2. Если в качестве аргумента передать true, то создаст глубокую копию элемента, включая атрибуты и подэлементы.

**События:**

*Мышь***:**

1. **click** – происходит, когда кликнули на элемент левой кнопкой мыши
2. **contextmenu** – происходит, когда кликнули на элемент правой кнопкой мыши
3. **mouseover** – возникает, когда на элемент наводится мышь
4. **mousedown** и **mouseup** – когда кнопку мыши нажали (down) или отжали (up)
5. **mousemove** – при движении мыши над элементом

*Тачскрин*:

1. **touchstart** – происходит, когда элемента коснулись
2. **touchmove** – происходит, когда по элементу провели пальцем
3. **touchend** – возникает, когда касание закончилось и палец убрали
4. **touchcancel** – палец переместился на интерфейс браузера или тач-событие нужно отменить

*Дополнительно*:

1. **submit** – отправка формы
2. **focus** – фокус на элементе
3. **keyup/keydown** – печать на клавиатуре
4. **DOMContentLoaded -** браузер полностью загрузил HTML, было построено DOM-дерево, но внешние ресурсы, такие как картинки <img> и стили, могут быть ещё не загружены.
5. **transitionend** – закончена CSS-анимация

Обработчик событий:

Самый простой способ добавить обработчик события на какой-то элемент - явно прописать его в качестве атрибута с префиксом on, который будет ему соответствовать. А далее мы указываем в качестве значения атрибута js-функцию, которая будет вызвана в качестве обработчика этого события.

1. Встроить его в DOM-элемент, например:

**<li onclick="alert('привет');">Щелкни меня</li>**

1. Или назначить выбранному DOM-элементу функцию-обработчик в коде:

**var li = document.getElementByTagName('li')[0];**

**li.onclick = function () { alert('привет'); }**

Это рабочий способ, и он будет работать даже в самых старых браузерах, но настоятельно не рекомендован, т.к.:

1. Мы не можем навешать более одного обработчика на какое-то конкретное событие.
2. JS код и наш HTML код становятся очень тесно связаны.
3. Меняя HTML код или меняя JavaScript код, нам очень важно всегда помнить, что мы не можем переименовать значение/тип атрибута, иначе это будет другое событие.
4. Мы не можем навешивать эти обработчики динамически, должны изначально присутствовать в теле HTML-документа.

Для решения всех этих проблем был создан метод **addEventListener**, который добавляет слушателя на какое-то событие, которое может произойти на DOM-элемент.

1. Воспользоваться методом **addEventListener**, передав название обработчика аргументом:

**li.addEventListener('click', () => {**

**alert('привет');**

**});**

Но обработчиков может быть значительно больше, и мы можем добавить их целый пул, таким образом создать стек обработки событий и разбить нашу обработку на несколько отдельных функций. Все обработчики всегда вызывают в том порядке, в котором они добавляются в качестве слушателей. Контекстом любой функции обработки события всегда будет являться DOM-элемент, на котором это событие произошло, и на котором был вызван addEventListener для того, чтобы его послушать. В любую функцию обработчика события в качестве первого параметра передается объект события. И у этой переменной есть несколько свойств:

1. **target** - указывает на фактический DOM-узел (самый последний листок DOM-дерева, на котором произошло событие).
2. **altKey, ctrlKey, shiftKey** - говорят о том, была ли нажата соответствующая клавиша во время события и значение type, которое указывает строкой тип события, который фактически сейчас произошел.

Чтобы удалить обработчик событий, нужно воспользоваться методом **removeEventListener**, в который нужно передать **ссылку на назначенный обработчик**. Другими словами, вот это не сработает, несмотря на то, что переданные обработчики делают одно и то же:

**var li = document.getElementByTagName('li')[0];**

**li.addEventListener('click', ()=>alert('привет'));**

**// не работает**

**li.removeEventListener('click', ()=>alert('привет'));**

А вот это сработает, ведь передана ссылка на тот же обработчик onClickHandler:

**var li = document.getElementByTagName('li')[0];**

**var onClickHandler = () => {**

**alert('привет');**

**}**

**li.addEventListener('click', onClickHandler);**

**li.removeEventListener('click', onClickHandler);**

# Билет №4

## Захват и всплытие события. Паттерны работы с событиями

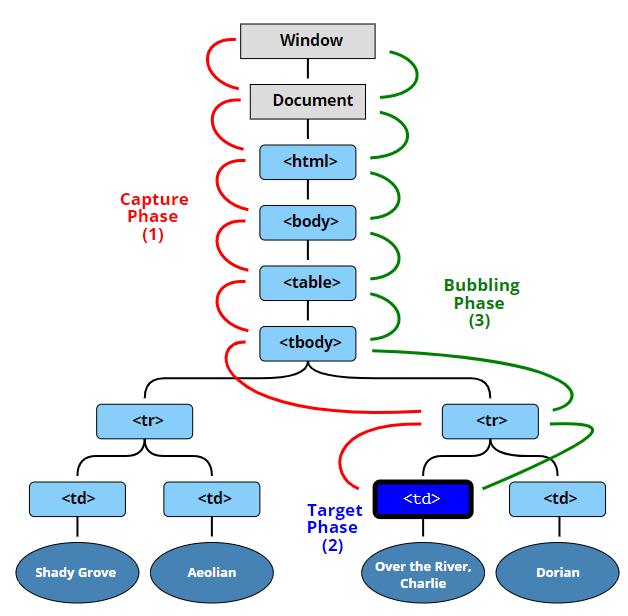
### Ссылки:

* Слайды: <https://urfu-2017.github.io/webdev-slides/06-browser-and-js/lection/index.html#/8>
* Полезное: <https://learn.javascript.ru/events>
* <https://learn.javascript.ru/bubbling-and-capturing>

## Всплытие и погружение:

При наступлении события – самый глубоко вложенный элемент, на котором оно произошло, помечается как «целевой» (event.target).

* Затем событие сначала двигается вниз от корня документа к event.target, по пути вызывая обработчики, поставленные через addEventListener(...., true), где true – это сокращение для {capture: true}.
* Далее обработчики вызываются на целевом элементе.
* Далее событие двигается от event.target вверх к корню документа, по пути вызывая обработчики, поставленные через on<event> и addEventListener без третьего аргумента или с третьим аргументом равным false. ({capture: false})



По умолчанию все обработчики работают на стадии всплытия или захвата. Большинство событий всплывают, но не все. Например, focus не всплывает.

Чтобы поймать событие на стадии погружения, нужно использовать третий аргумент capture вот так:

elem.addEventListener(..., {capture: true})

// или просто "true", как сокращение для {capture: true}

elem.addEventListener(..., true)

Каждый обработчик имеет доступ к свойствам события event:

* event.target – самый глубокий элемент, на котором произошло событие.
* event.currentTarget (=this) – элемент, на котором в данный момент сработал обработчик (тот, на котором «висит» конкретный обработчик)
* event.eventPhase – на какой фазе он сработал (погружение=1, фаза цели=2, всплытие=3).

Любой обработчик может остановить всплытие вызовом:

* event.stopPropagation() - прекратить всплытие
* event.stopImmediatePropagation() - прекратить всплытие и все оставшиеся обработчики

Также event.preventDefault() - отменяет обработчик по умолчанию.

## Паттерны:

### Делегирование:

Идея в том, что если у нас есть много элементов, события на которых нужно обрабатывать похожим образом, то вместо того, чтобы назначать обработчик каждому, мы ставим один обработчик на их общего предка.

Из него можно получить целевой элемент event.target, понять на каком именно потомке произошло событие и обработать его.

### Debounce:

Debouncing функции означает, что все вызовы будут игнорироваться до тех пор, пока они не прекратятся на определённый период времени. Только после этого функция будет вызвана. Например, если мы установим таймер на 2 секунды, а функция вызывается 10 раз с интервалом в одну секунду, то фактический вызов произойдёт только спустя 2 секунды после крайнего (десятого) обращения к функции.

# Билет №5

## Функциональные и классовые компоненты в React. Жизненный цикл компонента

### Ссылки

* Преза: <https://frontend-ifmo-2019.now.sh/webdev/05-react#/>
* [Компоненты и пропсы – React](https://ru.reactjs.org/docs/components-and-props.html)
* [Состояние и жизненный цикл](https://ru.reactjs.org/docs/state-and-lifecycle.html)
* <https://habr.com/ru/post/461541/#section23>

(преза - поверхностно, можно просто полистать, чтобы понять, че там за темы вообще, далее идут две годные ссылки - если их прочитать(пункты 1-5), то всё понятно в принципе, ну и хабр - там тож неплохо все написано)

**Совет от души:** если у вас мало времени, а посмотреть перед экзом - не посмотрел ни черта, то презу даже не открывай, читай просто пункты 1-5 из основных понятий(красные ссылки), а при ответе пользуйся нашим конспектом :)

**Концепции React**:

1. Декларативность
2. Компонентный подход
3. Эффективная абстракция над DOM
4. Изоморфный код
5. Реактивный рендеринг
6. Нисходящий поток данных

### Компоненты

**React компонент** — функция возвращающая React-элемент или JavaScript класс реализующий метод render() который возвращает React-элемент.

**Важное различие:**

Компоненты, объявленные как функции JavaScript не могут содержать «состояния», поэтому их также принято называть **stateless components**, то есть, компоненты без состояния, а компонентные объявленные как JavaScript класс могу содержать «состояние», поэтому их называют **stateful components**, то есть, компоненты с состоянием.

\*На самом деле вы можете добиться похожей функциональности и от функциональных компонентов используя [хуки](https://ru.reactjs.org/docs/hooks-overview.html#state-hook). (но это уже другой билет)

Поэтому кратко на заметку:

**Хуки** — это функции, с помощью которых вы можете «подцепиться» к состоянию и методам жизненного цикла React из функциональных компонентов. Хуки не работают внутри классов — они дают вам возможность использовать React без классов.

**React.Component** — это базовый класс для компонентов React.

Вот тут я очень советую обратиться к след. билету и рассказать про **пропсы.**

**Композиция компонентов**

Компоненты могут ссылаться на другие компоненты в возвращенном ими дереве. Это позволяет нам использовать одну и ту же абстракцию — компоненты — на любом уровне нашего приложения.

В приложениях, написанных на React с нуля, как правило, есть один компонент **App**, который находится на самом верху.

Вот тут я очень советую обратиться к след. билету и рассказать про **состояние.**

### Жизненный цикл компонента

[React lifecycle methods diagram](https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/) --- оч классная вещь

Каждый компонент имеет несколько «методов жизненного цикла».

Первоначальный рендеринг компонента в DOM называется «монтирование» (mounting).

Каждый раз когда DOM-узел, созданный компонентом, удаляется, происходит «размонтирование» (unmounting).

**Монтирование**

При монтировании запускается следующие методы в указанном порядке:

* constructor() - нужен для инициализации внутреннего состояния и привязки обработчиков событий к экземпляру
* getDerivedStateFromProps() - должен вернуть объект для обновления состояния или null, чтобы ничего не обновлять. Этот метод существует для редких случаев, когда состояние зависит от изменений в пропсах.
* render() - обязательный метод, должен быть чистым
* **componentDidMount()** - вызывается сразу после монтирования (то есть, вставки компонента в DOM), в этом методе должны происходить действия, которые требуют наличия DOM-узлов.

**Обновление**

Обновление происходит при изменении пропсов или состояния компонента. При повторном рендере компонента закономерно вызываются следующие методы в указанном порядке:

* static getDerivedStateFromProps()
* shouldComponentUpdate() - указывает на необходимость следующего рендера на основе изменений состояния и пропсов.

По умолчанию происходит повторный рендер при любом изменении состояния.

* render()
* getSnapshotBeforeUpdate() - позволяет вашему компоненту брать некоторую информацию из DOM (например, положение прокрутки) перед её возможным изменением.

Хотя этот метод и вызывается после render() в нем вы все еще можете получить «снимок», то как выглядит DOM до рендера.

* componentDidUpdate() - вызывается сразу после обновления DOM. Он подходит для выполнения, таких действии которые требуют наличие DOM. Также он подходит для выполнения сетевых запросов, которые выполняются на основании результата сравнения текущих пропсов с предыдущими.

componentDidUpdate() не вызывается, если shouldComponentUpdate() возвращает false.

Если ваш метод render() зависит от некоторых других данных, вы можете указать React необходимость в повторном рендере, вызвав **forceUpdate()**.

Вызов forceUpdate() приведёт к выполнению метода render() в компоненте, **пропуская shouldComponentUpdate()**. Это вызовет обычные методы жизненного цикла для дочерних компонентов, включая shouldComponentUpdate() каждого дочернего компонента.

**Размонтирование**

При удалении компонента из DOM вызывается метод componentWillUnmount().

**componentWillUnmount()** вызывается непосредственно перед размонтированием и удалением компонента. В этом методе выполняется необходимый сброс: отмена таймеров, сетевых запросов и подписок, созданных ранее.

# Билет №6

## Props, Context, State в React - способы использования, различия

### Ссылки:

* Преза: <https://frontend-ifmo-2019.now.sh/webdev/05-react#/>
* [Компоненты и пропсы – React](https://ru.reactjs.org/docs/components-and-props.html)
* [Состояние и жизненный цикл](https://ru.reactjs.org/docs/state-and-lifecycle.html)
* [Контекст](https://ru.reactjs.org/docs/context.html#when-to-use-context)

(преза - поверхностно, можно просто полистать, чтобы понять, че там за темы вообще, далее идут три годные ссылки - если их прочитать(пункты 1-5 из основных понятий + контекст), то всё понятно)

Советую посмотреть предыдущий билет для понимания.

Стоит начать с темы **компонент** из предыдущего билета.

### Пропсы

**Пропсы** — это просто аргументы передаваемые компоненту. Когда React встречает JSX-атрибуты он собирает их в один объект и передаёт компоненту. Этот объект и называется «пропсы» (props).

**Что такое JSX?**

Достаточно быстро пробежаться глазками по [Знакомство с JSX](https://ru.reactjs.org/docs/introducing-jsx.html) и все станет понятно.

Но кратко: расширение JS; синтаксический сахар для функции React.createElement(..); html-ка, наделенная силой JS.

JavaScript XML (можем фигачить разметку в файлах)

В React мы связываем логику и разметку с помощью компонент. В J SX мы можем использовать обычные теги, а можем создать компоненту и передать ей какие-нибудь атрибуты.

React-компоненты обязаны вести себя как **чистые функции** по отношению к своим пропсам.

Компонент никогда **не должен что-то записывать в свои пропсы** — вне зависимости от того, функциональный он или классовый.

Пример: <MyComponent myProp=”value” />

Дёргать внутри MyComponent: props.myProp

### Состояние

**Состояние компонента** (State) — это приватное свойство (state) классовых React компонентов. Обновляя состояние с помощью метода setState() компонент рендерится повторно. Из этого следует запомнить что компонент не будет рендериться заново если изменять состояние напрямую (без вызова setState()).

setState() использует текущее значение состояния для его обновления. Поэтому вместо объекта с новым состоянием передаётся функция получающая текущее состояние как аргумент. Дело в том что вызовы setState являются **асинхронными** из-за чего this.state в некоторых случаях может отображать неправильное значение.

setState(updater[, callback]

**Обновления состояния объединяются!**

Например, состояние может состоять из нескольких независимых полей:

* если состояние хранит несколько значений, то изменение одного значения не повлияет на другое
* оба поля можно обновить по отдельности с помощью отдельных вызовов setState()

Состояние доступно **только для самого компонента и скрыто от других**. То есть this.state является приватным свойством.

Компонент может передать свое состояние вниз по дереву в виде пропсов дочерних компонентов.

### Контекст

### Контекст позволяет передавать данные через дерево компонентов без необходимости передавать пропсы на промежуточных уровнях.

### Контекст разработан для передачи данных, которые можно назвать «глобальными» для всего дерева React-компонентов (например, текущий аутентифицированный пользователь, UI-тема или выбранный язык).

### **React.createContext**

Создание объекта Context. Когда React рендерит компонент, который подписан на этот объект, React получит текущее значение контекста из ближайшего подходящего Provider выше в дереве компонентов.

### **Context.Provider**

Каждый объект Контекста используется вместе с Provider компонентом, который позволяет дочерним компонентам, использующим этот контекст, подписаться на его изменения.

Принимает проп value, который будет передан во все компоненты, использующие этот контекст и являющиеся потомками этого Provider компонента. Один Provider может быть связан с несколькими компонентами, потребляющими контекст. Так же Provider компоненты могут быть вложены друг в друга, переопределяя значение контекста глубже в дереве.

### **Class.contextType**

В свойство класса contextType может быть назначен объект контекста, созданный с помощью [React.createContext()](https://ru.reactjs.org/docs/context.html#reactcreatecontext). Это позволяет вам использовать ближайшее и актуальное значение указанного контекста при помощи this.context. В этом случае вы получаете доступ к контексту, как во всех методах жизненного цикла, так и в рендер методе.

### **Context.Consumer**

Consumer — это React-компонент, который подписывается на изменения контекста.

Consumer принимает [функцию в качестве дочернего компонента](https://ru.reactjs.org/docs/render-props.html#using-props-other-than-render). Эта функция принимает текущее значение контекста и возвращает React-компонент. Передаваемый аргумент value будет равен ближайшему (вверх по дереву) значению этого контекста, а именно пропу value Provider компонента. Если такого Provider компонента не существует, аргумент value будет равен значению defaultValue, которое было передано в createContext().

### **Context.displayName**

Объекту Context можно задать строковое свойство displayName. React DevTools использует это свойство при отображении контекста.

### 

# Билет №7

## Протокол HTTP. Формат, механики, проблемы

Презентация: <https://urfu-2017.github.io/webdev-slides/05-client-server/index.html#/>

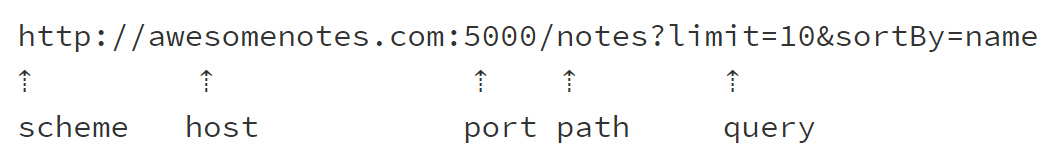
<https://urfu-2017.github.io/webdev-slides/05-client-server/index.pdf> (то же самое в формате pdf)

Слайды: 1 - 30

<https://habr.com/en/post/308846/>

HTTP - HyperText Transfer Protocol

URI:



Формат http

Запрос:

POST /notes HTTP/1.1 ← Request line

Accept: application/json ← Headers

Content-Length: 35

Content-Type: application/json; charset=utf-8

Host: localhost:8080

User-Agent: HTTPie/0.9.3

← Empty line

{ ← Body

"name": "films",

"text": "Films to watch"

}

Ответ:

HTTP/1.1 200 OK ← Status line

Content-Length: 67 ← Headers

Content-Type: application/json; charset=utf-8

Date: Wed, 16 Mar 2016 14:32:18 GMT

X-Powered-By: Express

← Empty line

{ ← Body

"createdAt": 1458138738899,

"id": 33,

"name": "films",

"text": "Films to watch"

}

Обязательные - Content-Length

Стандартные - Content-Length, Content-Type, Date

Методы

GET получение ресурса

POST создание ресурса

PUT обновление ресурса

PATCH частичное изменение ресурса

DELETE удаление ресурса

HEAD запрос заголовков ресурса

OPTIONS определение возможностей сервера для ресурса

Код ответа

1xx информационные

101 Switch protocol

2xx успех транзакции

200 Ok

201 Created

204 No content

3xx перенаправления

301 Moved Permanently

304 Not modified

4xx ошибки клиента

400 Bad request

403 Forbidden

404 Not found

5xx ошибки сервера

500 Internal Server Error

504 Gateway Timeout

Сжатие

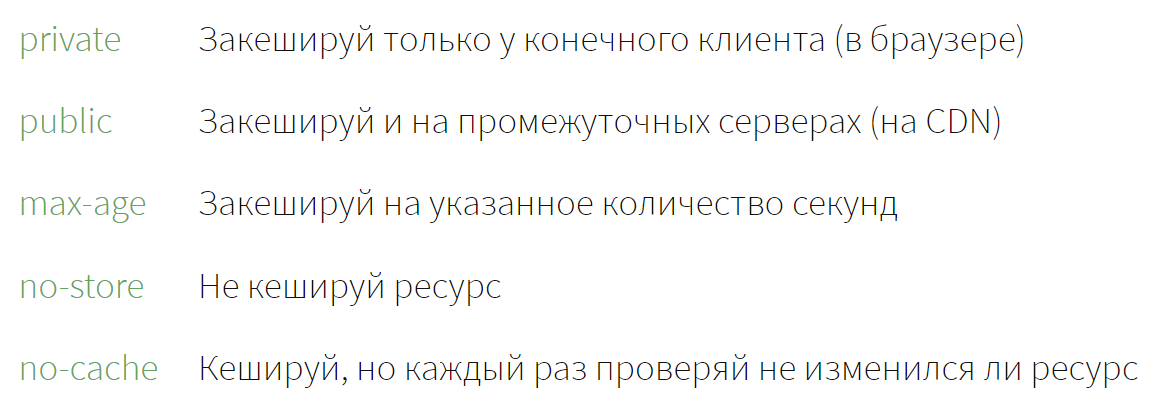
Http позволяет сжимать данные, получаемые от сервера. Для этого в запросе указывается заголовок: Accept-Encoding: gzip, br. Тогда в ответе приходит соответствующий заголовок: Content-Encoding: gzip

Кэширование

В http возможно кэшировать приходящие ресурсы. Для этого вместе с ресурсом должен прийти заголовок, явно указывающий на возможность кэширования.



Возможные значения:



Если запрос идет на закешированный ресурс, которые не изменился, возвращается код 304 - Not Modified

Проблемы HTTP 1.1

Head-of-Line blocking - ситуация, при которой первый пакет сдерживает последующие пакеты в очереди

Ограниченное количество параллельных запросов (6 запросов)

Повторяющийся блок заголовков

HTTP/2

Решение проблем http1:

Head-of-Line blocking -> Multiplexing (вы можете выдавать новые запросы по одному и тому же TCP соединению, не дожидаясь завершения предыдущих)

Ограниченное количество параллельных потоков -> Multiplexing

Повторяющийся блок заголовков -> Сжатие заголовков (HPACK)

Мы постоянного обращаемся к серверу из одного и того же клиента, то в заголовках раз за разом посылается огромное количество повторяющихся данных. А иногда к этому добавляются ещё и куки, раздувающие размер заголовков, что снижает пропускную способность сети и увеличивает время задержки. Чтобы решить эту проблему, в HTTP/2 появилось сжатие заголовков по алгоритму Хаффмана.

Проблемы:

Бинарный - HTTP/2 пытается решить проблему выросшей задержки, существовавшую в HTTP/1.x, переходом на бинарный формат. Бинарные сообщения быстрее разбираются автоматически, но, в отличие от HTTP/1.x, не удобны для чтения человеком.

Server push - Сервер, зная, что клиент собирается запросить определённый ресурс, может отправить его, не дожидаясь запроса. Вот, например, когда это будет полезно: браузер загружает веб-страницу, он разбирает её и находит, какой ещё контент необходимо загрузить с сервера, а затем отправляет соответствующие запросы.

Head-of-Line blocking на уровне TCP -

Если один пакет отброшен или потерян в сети где-то между двумя конечными точками, которые говорят на HTTP/2, это означает, что все TCP-соединение остановлено, в то время как потерянный пакет повторно передается и находит свой путь к месту назначения. Поскольку TCP-это такая "цепочка", это означает, что если одно звено внезапно отсутствует, то все, что придет после потерянного звена, должно подождать.

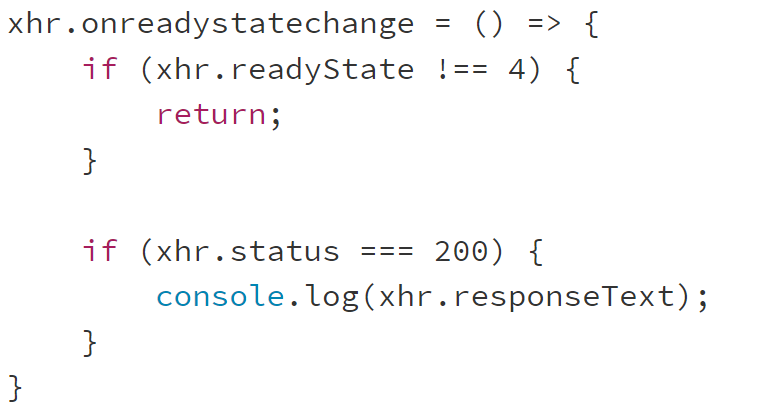
Инструменты для работы с http:

* XMLHttpRequest - API для работы с запросами

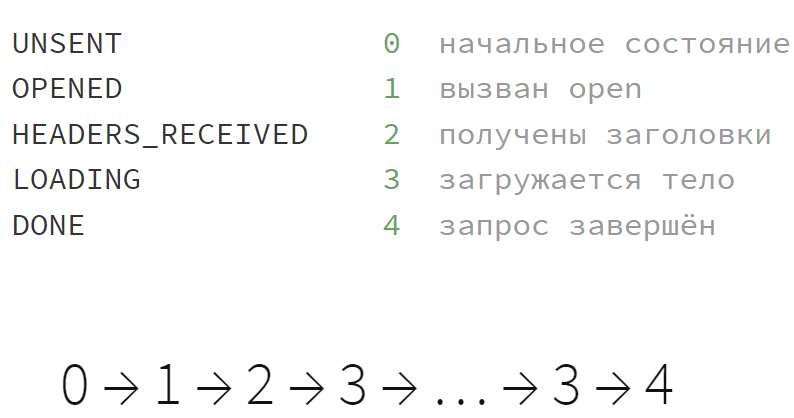
Отправка запроса:



Получение ответа:



Этапы у xhr.readyState



* Запросы через Fetch



Недостатки данной техники - неудобно отследить прогресс запроса.

Не во всех браузерах есть возможность отмены запроса.

# Билет №8

## REST. Принципы, рекомендации, проблемы

REST - REpresentational State Transfer

ссылки:

<https://frontend-ifmo-2019.now.sh/webdev/06-client-server#/32>

<https://www.restapitutorial.com/> (видео на 20 минут на английском)

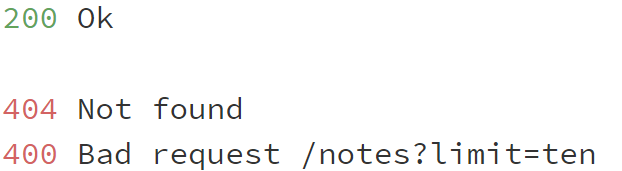
REST (основные принципы)

* Клиент-сервер - Отделяя пользовательский интерфейс от хранилища данных, мы улучшаем переносимость пользовательского интерфейса на другие платформы и улучшаем масштабируемость серверных компонент за счет их упрощения.
* Stateless - Каждый запрос от клиента к серверу должен содержать в себе всю необходимую информацию и не может полагаться на какое-либо состояние, хранящееся на стороне сервера. Таким образом, информация о текущей сессии должна целиком храниться у клиента.
* Механизм кэширования - Требует, чтобы для данных в ответе на запрос явно было указано -- можно их кэшировать или нет. Если ответ поддерживает кэширование, то клиент имеет право повторно использовать данные в последующих эквивалентных запросов без обращения на сервер.
* Идентификация ресурсов - В REST ресурсом является все то, чему можно дать имя. Например,пользователь, изображение, предмет (майка, голодная собака, текущая погода) и т.д. Каждый ресурс в REST должен быть идентифицирован посредством стабильного идентификатора, который не меняется при изменении состояния ресурса.
* Самодостаточные сообщения - Каждое сообщение содержит достаточно информации, чтобы понять, каким образом его обрабатывать.
* HATEOAS (связность ресурсов) - (hypermedia as the engine of application state). Статус ресурса передается через содержимое body, параметры строки запроса, заголовки запросов и запрашиваемый URI (имя ресурса). Это называется гипермедиа (или гиперссылки с гипертекстом). HATEOAS также означает, что, в случае необходимости ссылки могут содержатся в теле ответа (или заголовках) для поддержки URI , извлечения самого объекта или запрошенных объектов.

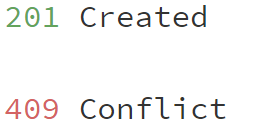
Все кроме связности соответствует идеям http

Запросы

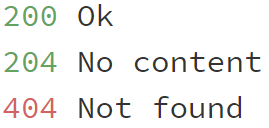
GET - Получает состояние ресурса



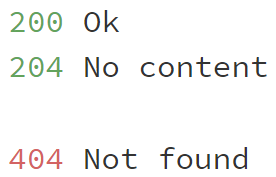
POST - Создаёт новый ресурс с начальным состоянием, когда мы не знаем его ID



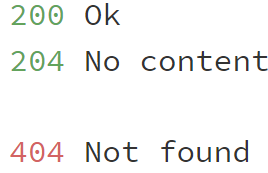
PUT - Создаёт новый ресурс с начальным состоянием, когда мы знаем его ID, обновляет состояние целиком



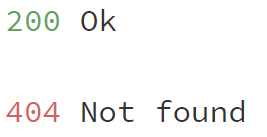
DELETE - Удаляет существующий ресурс



PATCH - Обновляет состояние существующего ресурса частично



HEAD - Запрашивает заголовки, чтобы проверить существование ресурса



OPTIONS - Запрашивает правила взаимодействия, например, доступные методы

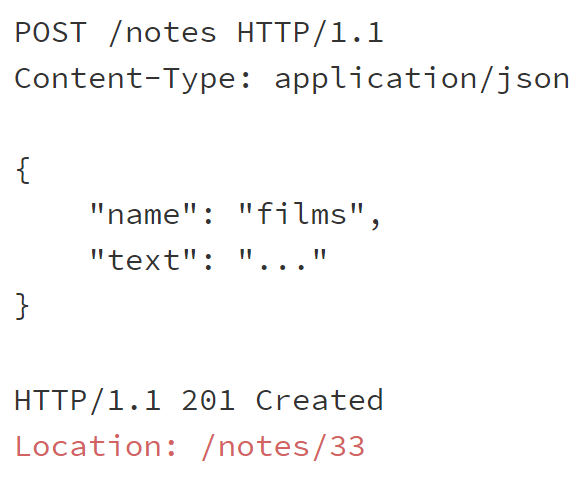
**Безопасный** запрос не меняет состояния приложения

**Идемпотентный** запрос - запрос эффект которого от многократного выполнения равен эффекту от однократного выполнения



Связность - при запросе возвращается дополнительная информация

К примеру при создании - id созданного элемента или ссылка на него



Hypertext Application Language - способ поддерживать связность. В результат запроса добавляются ссылки на связанные ресурсы. Пример:



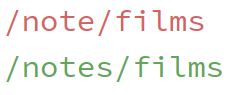
Подробнее про HAL - <http://stateless.co/hal_specification.html>

Best practices

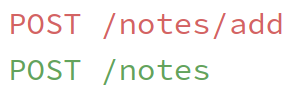
Используйте path, а не query



Используйте множественное число, а не единственное



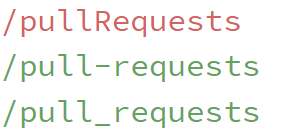
Используйте только существительные, не глаголы



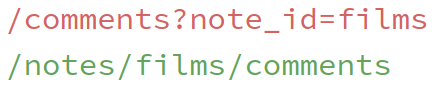
Избегайте избыточности



Используйте kebab-case или snake\_case для разделения слов



Используйте вложенность



Минусы REST

-Не протокол - rest это скорее просто набор правил по работе с http

-Ограниченный словарь - можно использовать только методы http и конечный набор адресов

-Размазанные данные - на каждый объект один запрос. Для получения данных часто необходимо множество запросов

-Привязка к HTTP - невозможно организовать работу с использованием другого протокола

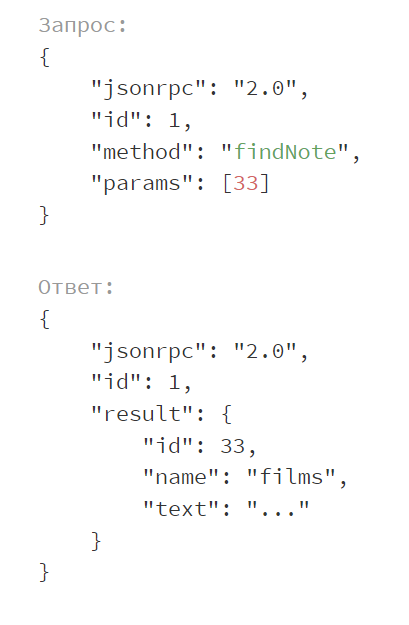
-Один запрос - один ресурс (нет batching-a) - нельзя за один запрос получить несколько объектов(ресурсов), получать ресурсы пакетами.

Альтернативы

* JSON-RPC - remote procedure call

Подробнее, примеры: <https://www.jsonrpc.org/specification>

Обмен между сервером и клиентом json. Запрос содержит id и метод, ответ - id запроса и результат.



* GraphQL - на лекции ничего про него не рассказывали, есть документация <https://graphql.org/>

# Билет №9

## React Hooks. Стандартные хуки, их применение

Небольшое вступление и советы. Чтобы понять, зачем хуки, советую посмотреть [это](https://learn-reactjs.ru/basics/state-and-lifecycle). По ссылке описывается создание класса-компонента и, что особенно полезно, его поведение (как рендерится, что вызывается). Классы-компоненты включают себя большую функциональность в отличии от функциональных компонентов, так вот хуки придуманы, чтобы добавить им всякие плюшки как для классов. После этой статьи можно прочитать статью непосредственно про [хуки](https://ru.reactjs.org/docs/hooks-overview.html), после предыдущей статьи выглядит очень понятно и понятно зачем оно надо.

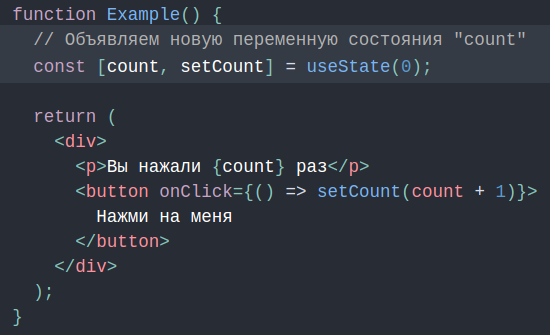
Ссылка на презентацию, там довольно [суховато](https://frontend-ifmo-2019.now.sh/webdev/10-react-2#/17)

[Справочник API хуков](https://ru.reactjs.org/docs/hooks-reference.html) - почитать про какие-то конкретные хуки подробнее. Для общего понимание хватает тех двух ссылок.

Хуки - специальные функции, которые позволяют функциональным!! [компонентам](#_rc6ma6ugkplx) «подцепиться» к возможностям React.

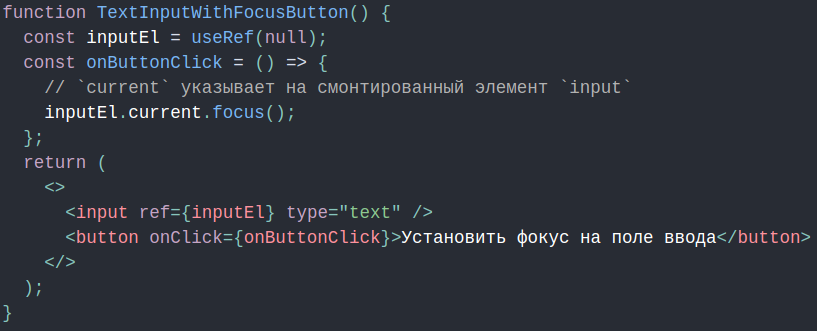
useState

Добавляет состояние. Состояние сохраняется между рендерами, изменение состояния вызывает ререндер. При обновлении состояние заменяется целиком. Принимает единственный аргумент - начальное состояние ( состояние может, но не обязано, быть объектом) и возвращает текущее состояние и функцию для его обновления (сеттер).



useRef

Добавляет контейнер. Объект остается одним и тем же между рендерами, изменение содержимого контейнера **не** вызывает ререндер. Контейнер может хранить ссылку на любое мутируемое значение или на DOM-узел.



useContext

Добавляет [контекст](https://ru.reactjs.org/docs/context.html#when-to-use-context). Контекст можно читать, а также подписываться на его изменения. Изменение значения контекста вызывает ререндер.

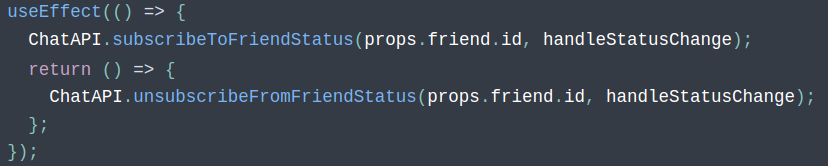
const value = useContext(MyContext); - принимает объект контекста (значение, возвращённое из React.createContext).

[useEffect](https://ru.reactjs.org/docs/hooks-overview.html#effect-hook)

(как componentDidMount, componentDidUpdate, componentWillUnmount в классах-компонентах)

Добавляет эффект. Эффект и его очистка запускаются после каждого рендера (есть оптимизации). Можно управлять тем, при изменении каких параметров перезапускать эффект. Запуск отложенный: после текущей отрисовки, но до следующей. Для синхронного выполнения есть useLayoutEffect.

Поскольку эффекты объявляются внутри компонента, у них есть доступ к его пропсам и состоянию. По умолчанию, React запускает эффекты после каждого рендера, включая первый рендер.

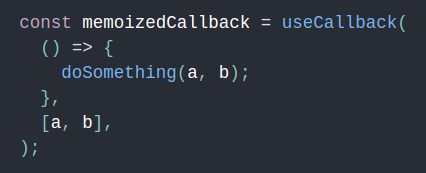


useMemo



Добавляет значение, которое перевычисляется только при изменении зависимостей. Это может быть полезно для оптимизации. Фабрика, вычисляющее значение, запускается во время рендера, в ней не должно быть сайд-эффектов.

useCallback



Добавляет коллбэк, который сохраняется между рендерами. Функция пересоздаётся при изменении зависимостей. Оборачивание коллбэков в хук нужно для передачи их дочерним компонентам для предотвращения ненужных рендеров.

useCallback(fn, deps) — это эквивалент useMemo(() => fn(), deps)

Можно писать свои [хуки](https://ru.reactjs.org/docs/hooks-overview.html#building-your-own-hooks)

# Билет №10

## React. Повторное использование логики. Компоненты высшего порядка. Как работают Render Props.

## React.

## React - это декларативная, эффективная и гибкая библиотека JavaScript для создания пользовательских интерфейсов (UI). Она позволяет вам создавать сложные UI из небольших и изолированных частей кода, называемых «компонентами».

## React дает вам язык шаблонов и некоторые callback-функции для отрисовки HTML. Весь результат работы React — это HTML. Ваши связки HTML/JavaScript, называемые компонентами, занимаются тем, что хранят свое внутреннее состояние в памяти (например: какая закладка выбрана), но в итоге вам просто выплевывается HTML. Разумеется, вы не можете построить полно функционирующее динамическое приложение только с React.

## Особенности.

Имена пользовательских компонентов должны начинаться с заглавной буквы

**<Article /> React.createElement(Article, null);**

**<article /> React.createElement("article", null);**

Может быть только один корневой элемент

**function MyComponent() {**

**return (**

**<p>Первый абзац</p>**

**<p>Второй абзац</p>**

**);**

**}**

Может быть только один корневой элемент

**function MyComponent() {**

**return (**

**<div>**

**<p>Первый абзац</p>**

**<p>Второй абзац</p>**

**</div>**

**);**

**}**

Props

Значимая часть при работе с React это его многократное использование, означающее возможность написать компонент один раз и затем использовать его в разных случаях, для примера — показывать разные сообщения.

Чтобы достигнуть такого рода повторного использования, мы сейчас добавим props. Это то, как мы передадим props компоненту. (подсвечено жирным):

**ReactDOM.render(**

**<Hello message="my friend" />,**

**document.getElementById("root")**

**);**

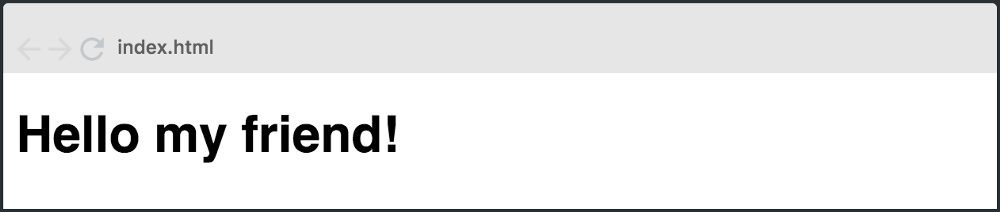
Props назван message и имеет значение *"my friend"*. Мы можем получить доступ к этому prop внутри компонента Hello, ссылаясь **this.props.message**, как тут:

**class Hello extends React.Component {**

**render() {**

**return <h1>Hello {this.props.message}!</h1>;**

**}**

**}**  
Как результат на экране мы увидим:

State

1. Может быть только у классов
2. Компонент реагирует на изменения состояния
3. Инициализация происходит в конструкторе
4. Изменять состояние напрямую нельзя

Другой способ хранения данных в React это state компонента, то есть его внутреннее состояние. И в отличие от props — который не может быть изменен напрямую компонентом — state это умеет.

Так что, если вы хотите изменить данные в вашем приложении — для примера основываясь на взаимодействиях пользователя — то это должно находиться в state компонента внутри приложения.

*Почитать более продробно про state:* <https://medium.com/@stasonmars/%D0%BF%D0%BE%D0%BD%D0%B8%D0%BC%D0%B0%D0%B5%D0%BC-react-%D0%B7%D0%B0-5-%D0%BC%D0%B8%D0%BD%D1%83%D1%82-2611122c6fbf>

Объединение компонентов

**import Form from './Form';**

**import Notes from './Notes';**

**function NotesApp() {**

**return (**

**<div className="notes-app">**

**<Notes />**

**<Form />**

**</div>**

**);**

**}**

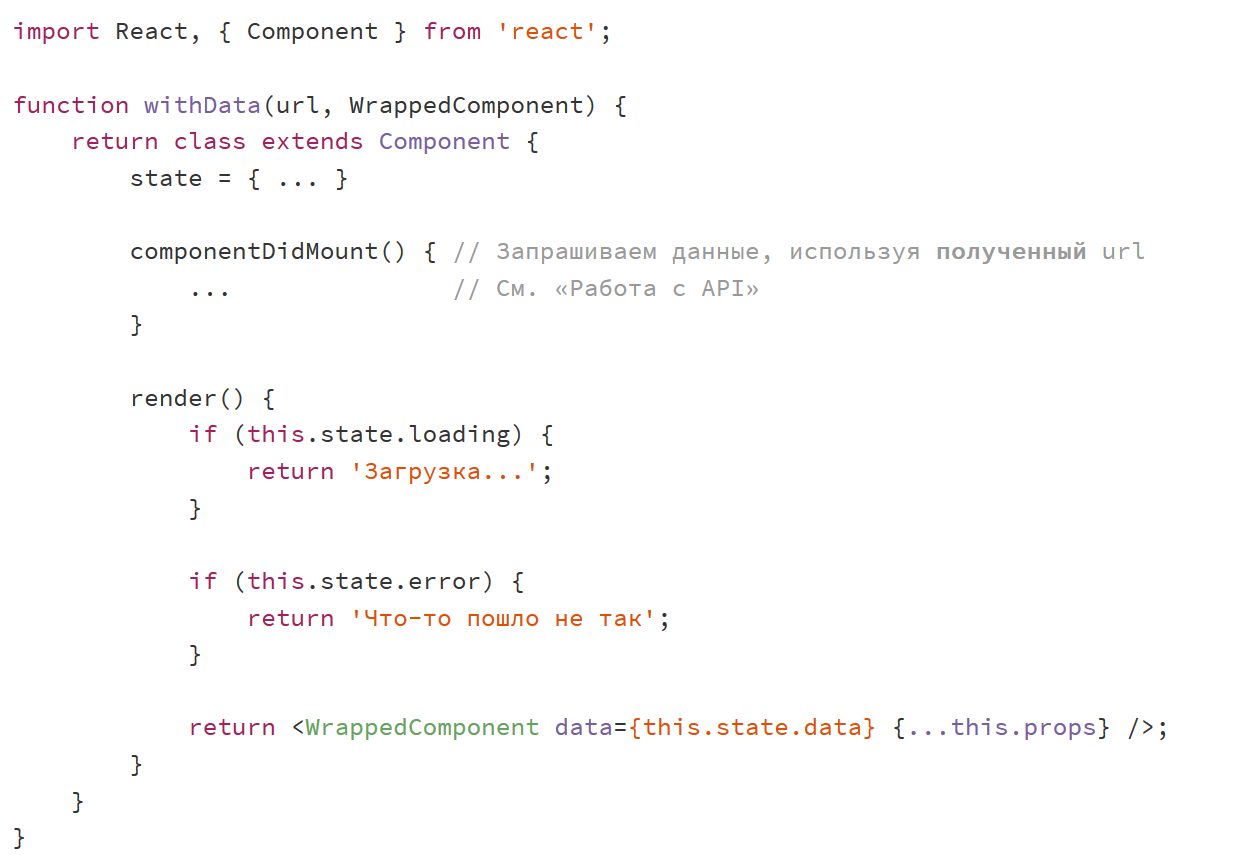
Компоненты высшего порядка

>>> Презентация: <https://frontend-ifmo-2019.now.sh/webdev/10-react-2#/51> (слайды 52-72)

>>> Полезно: <https://ru.reactjs.org/docs/higher-order-components.html>

## **HOC(Higher-Order Components) - принимают компонент и возвращают тот же компонент, но с дополнительной функциональностью.** Функция высшего порядка - функция, которая принимает функцию в качестве аргумента или возвращает функцию в качестве результата

## >>>Пример из презентации



Что это? HOC, который принимает на вход некоторый компонент и возвращает его же, но с добавленной в пропы data.

Для чего? Если у нас будет несколько компонентов, которые требуют data, то достаточно будет вызвать HOC для каждого такого компонента.

## 

Render Props

>>> Презентация: <https://frontend-ifmo-2019.now.sh/webdev/10-react-2#/72> (слайды 73-81)

>>> Полезно: <https://ru.reactjs.org/docs/render-props.html>

Для чего? Для повторного использования кода.

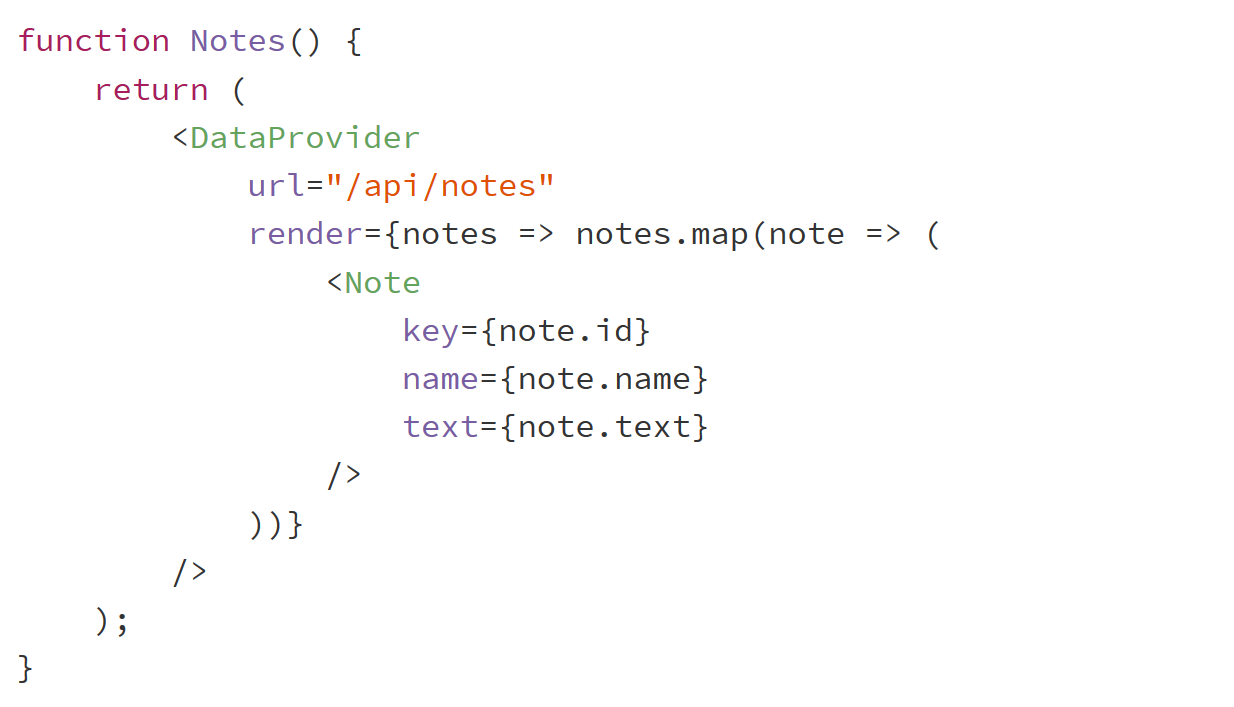
Как именно? Вместо того, чтобы создавать разные компоненты с похожим поведением, мы “вынесем” это поведение и через проп render будем ему передавать что необходимо отрендерить.

**Рендер-проп — функция, которая сообщает компоненту что необходимо рендерить.**

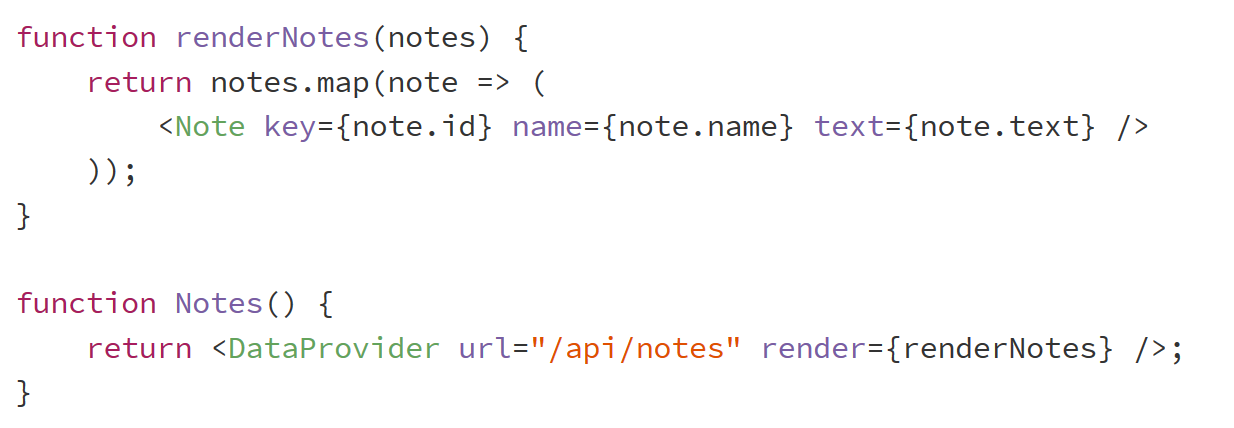
Может ли быть какой-то другой проп вместо render? Да, например, children. Или другой проп отвечающий за рендеринг (пример :( ).

>>>Пример из презентации

Было:



Стало:



Что поменялось? Теперь если понадобится по-другому отрендерить компонент с функциональностью DataProvider, будет необходимо лишь вместо renderNotes передать другую функцию в проп render.

Отличие от HOC?

# Билет №11

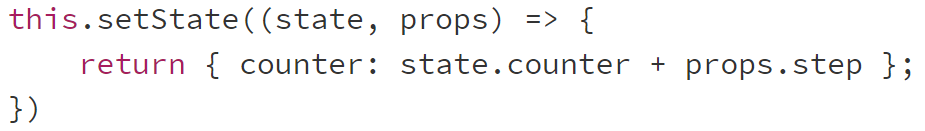
## Redux. Нарисовать классическую flux-архитектуру. Нарисовать flux-архитектуру в Redux. Объяснить преимущества flux-архитектуры перед MV\*. Объяснить преимущества redux перед классическим flux.

[презентация](https://frontend-ifmo-2019.now.sh/webdev/13-react-3/index.html#/), [видеозапись лекции](https://yadi.sk/i/0SzJbSiB5XF8TQ) (смотреть только самое начало)

**Application State** (также просто **State**) – структурированные данные о конкретном экземпляре приложения для одного конкретного пользователя  
**Model** – логика работы с этими данными (часто их объединяют вместе с State)  
**View** – часть приложения, отвечающая за отображение данных на странице

### MV-паттерн (Model-View)тест

Суть: пользователь во View совершает какие-либо события, State их обрабатывает и инициирует перерисовку View.

React: например, любой вызов функции setState принудительно инициирует render(), поскольку будет изменён state.

Недостатки:

* обработчики событий могли влиять на абсолютно несвязанные компоненты приложения, слишком сложные связи между событиями и тем, что за ними может последовать
* нужно следить за своевременным созданием/удалением обработчиков
* события могут приходить отовсюду (не только из UI, например, из другого UI, с помощью AJAX или вообще откуда блин угодно), за этим нужно постоянно следить
* сложно понимать, почему сработал тот или иной обработчик (приходится отлаживать при помощи блин консоли)

**ViewModel** – абстракция для отображения данных, представляющая собой обёртку над данными. Содержит сами данные и команды, с помощью которых из интерфейса можно влиять на данные и состояние.

### 

### MVVM-паттерн (Model-View-ViewModel)

Суть: появляется прослойка между View и Model.

Преимущества:

* можно тестировать и отлаживать ViewModel отдельно
* местоположение обработчиков событий куда более конкретизировано некоторой ViewModel, код обработки событий более разграничен

Недостатки:

* нужно помнить, откуда пришло событие, инициирующее команду в Model, чтобы обновить нужный View
* непонятно, как отлаживать изменения View после запроса к Model
* нет гарантий корректности Model после изменений

### Flux-архитектура

Суть: есть **Actions** (аналоги Commands, действия, которые поступают в диспетчер с целью отработать в Store), **Stores** (аналоги Model/State) и **Views**. События во Views *(например, нажатие на кнопку)* конвертируются в Actions (например, *editComment*), которые порождают изменения в Stores (изменение комментария), которые, в свою очередь, влекут за собой перерисовку View (обновление текста комментария). Также опциональным является наличие **Dispatcher**, который упорядочивает и синхронизирует запросы к Stores.

Преимущества:

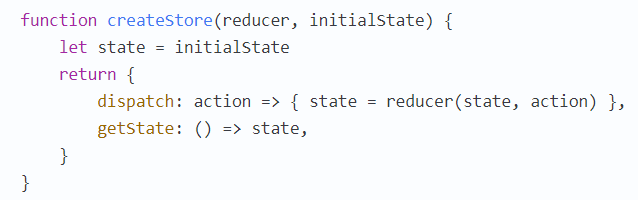
* отсутствие каскадных событий, которые влекут за собой непонятно что
* каждый Action отвечает за своё событие => есть возможность контролировать в Store, что мы меняем, и перерисовывать нужный View
* Dispatcher+Store+View работают синхронно => есть нормальная отладка

Недостатки:

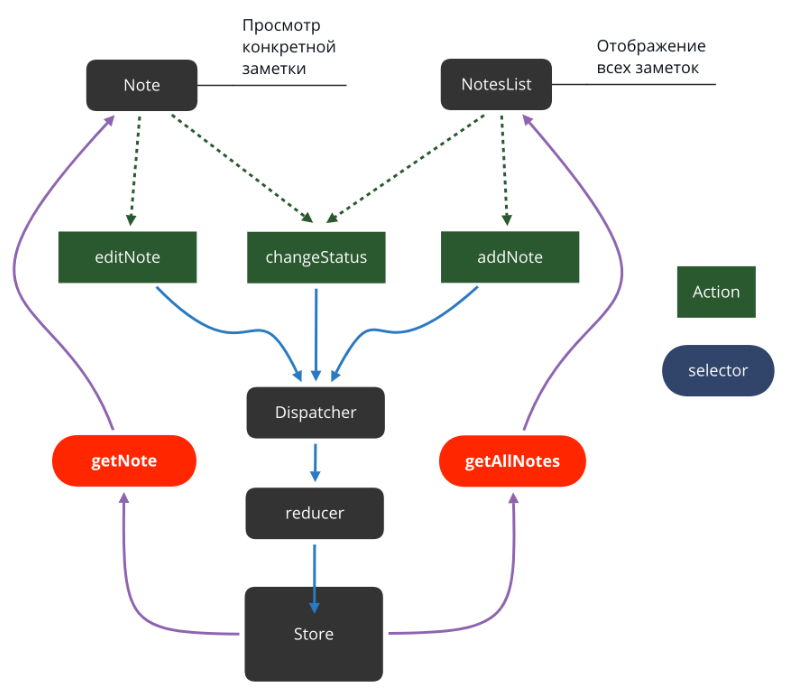
* нет гарантий корректности Model после изменений (как и в MVVM)

### 

### Redux

Суть: JS-библиотека Redux является реализацией Flux-архитектуры, в которой отсутствует Dispatcher, есть лишь один общий Store, а также используются **Reducers** – чистые функции, создающие новый Store на основе его старой версии и данных из Action – и **Selectors** – функции, отвечающие за передачу обновленных данных из Store во View.

Преимущества:

* Store гарантированно не сломается, так как нет побочных эффектов при его изменении

Недостатки:

* см. Проблемы Redux

### Проблемы Redux

* Если Store стал большой помойкой

Разделяем данные на 3 категории:

* данные, которые могут меняться из многих мест
* данные, которые нужны в разных несвязных компонентах
* Начальные данные приложения

Начальные данные приложения провайдим через **React Context API** во все компоненты без необходимости писать пропсы

* Как избежать render’a во всех местах, даже где не надо?

Библиотека **Reselect** - дает функционал, в который оборачиваются селекторы

* Асинхронность - см. следующий билет

### Послесловие

Надо понимать, что хоть каждый последующий паттерн архитектуры приложения решает проблемы предыдущего, все они не являются единственными существующими. Как минимум, из известных существуют MVC (Model-View-Controller) и MVP (Model-View-Presenter), идеи которых можно было встретить на вебе (Controller в Spring) или на андроиде (Activity или Fragment).

Кстати говоря, Redux не является библиотекой, специально написанной для React, она может использоваться и с другими фреймворками. Есть даже отдельная библиотека react-redux для более тесной интеграции, кажется.

Важный момент: обычно говорят, что Dispatcher отсутствует в Redux, но яндексоиды в презентации его всё-таки указывают, понимая под ним dispatch. Это какой-то странный момент.

# Билет №12

## Redux. Какие существуют проблемы с асинхронностью в redux и как их решить. Рассказать про одну из популярных библиотек (thunk, promise, saga или observable).

* [Презентация (слайды 136+)](https://frontend-ifmo-2019.now.sh/webdev/13-react-3/index.html#/136)
* [Релевантная документация Redux](https://redux.js.org/advanced/async-flow)
* [Лекция (с 1:27:06)](https://yadi.sk/i/0SzJbSiB5XF8TQ)

Основная и единственная проблема с асинхронностью в redux — он **не поддерживает** асинхронность из коробки. Никак.

Async Flow

По умолчанию, экшены в Redux являются синхронными, что, является проблемой для приложения, которому нужно взаимодействовать с серверным API, или выполнять другие асинхронные действия.

Главная идея всех подходов к решению этой проблемы в том, что в Redux есть миддлвары, которые являются прослойками между dispatch и reducer. В частности, эти миддлвары могут изменять action или делать вообще что угодно с ним.

Рассмотрим различные реализации и библиотеки, которые дают возможности для асинхронности в redux.

### **redux-thunk**: добавление миддлвары

redux-thunk позволяет писать action creators таким образом, что они возвращают функцию вместо action.

Эта функция вызывается и лишь её результат (сам action) передаётся в reducer. Каким образом? Эта функция принимает настоящий dispatch, что-то делает, а потом вызывает dispatch уже с нужным action.

Также можно применять для условного срабатывания события (dispatch вызывать необязательно). Суть библиотеки очень проста, но она подходит в большинстве случаев.

<https://redux.js.org/api/applymiddleware>

The key feature of **middleware** is that it is composable. Multiple middleware can be combined together, where each middleware requires no knowledge of what comes before or after it in the chain.

For example, [redux-thunk](https://github.com/gaearon/redux-thunk) lets the action creators invert control by dispatching functions. They would receive [dispatch](https://redux.js.org/api/store#dispatchaction) as an argument and may call it asynchronously. Such functions are called *thunks*.

Полезно:

* интересные ответы на вопросы в документации redux-thunk: <https://github.com/reduxjs/redux-thunk#why-do-i-need-this>

### **redux-saga**: добавляем отдельный тред

Раньше в redux-thunk была куча проблем с тестированием (приходилось писать много кода), Серёжа сказал, что это не так давно починили.

Помимо этого, иногда возникает потребность отменять какие-либо действия. Пример: форма входа. Если человек нажал быстро кнопку входа дважды — есть смысл отменить первый запрос.

Используется та же идея — saga подключается как миддлвара. Но помимо этого, она создаёт отдельный тред для обработки всех действий.

Все действия — генераторы (function\*). Мы используем yield, когда нужно что-то сделать, таким образом избавляясь от огромной кучи колбеков.

saga предоставляет некоторое количество хелперов:

* put(action) — задиспетчить action
* takeEvery(actionType, task) — указать, что action с типом actionType обрабатываются с помощью генератора task
* takeLatest(actionType, task) — то же, что takeEvery, но при этом гарантируется, что два события этого типа не смогут обрабатываться одновременно. Если поступит второе событие, обработка первого тут же остановится

Плюс, который мы получаем — можно чейнить экшны: put может быть обработан с помощью takeEvery/takeLatest, он снова отдаст событие, и оно снова может быть обработано и т.д., и только в конце попадёт в reducer.

Полезно:

* понять концепт по документации: <https://redux-saga.js.org/>
* некоторое мнение экспертов со StackOverflow: <https://stackoverflow.com/questions/54302091/when-should-i-use-redux-saga-instead-of-redux-thunk-and-when-should-i-use-redux>

### **redux-promise**: как redux-thunk, только для Promise

Не знаю, есть ли смысл рассказывать — недавно мейнтейнер написал, что проект заброшен. Вообще, это как redux-thunk, только принимает promises (фактически, async-функции). Позволяет делать примерно так:

createAction('FETCH\_THING', async id => {

const result = await somePromise;

return result.someValue;

});

### **redux-observable**: самая сложная библиотека

Решает те же проблемы, что и redux-saga. Основана на RxJS. Оверкилл примерно всегда.

Главная сущность — Epic: вещь, которая ест стрим экшнов (на самом деле, это и есть Observable, но оно очень похоже на джавовые стримы), и отдаёт стрим модифицированных экшнов. Пример из документации:

const pingEpic = action$ => action$.pipe(

filter(action => action.type === 'PING'),

mapTo({ type: 'PONG' })

);

// later...

dispatch({ type: 'PING' });

Таким образом, pingEpic для событий типа PING генерирует события PONG, они уже передаются в dispatch.

**Важно:** эпики здесь работают «рядом» с dispatch из redux, а не «вместо», как это было ранее. Изначальный экшн также будет обработан. То есть код выше создаст два экшна: PING и PONG.

Но здесь можно делать и асинхронные вещи:

const pingEpic = action$ => action$.pipe(

ofType('PING'),

delay(1000), // Asynchronously wait 1000ms then continue

mapTo({ type: 'PONG' })

);

Полезно:

* сравнение redux-saga и redux-observable, куча примеров: <https://hackmd.io/@2qVnJRlJRHCk20dvVxsySA/H1xLHUQ8e?type=view>

Важный момент (из документации): Since sagas and observables have the same use case, an application would normally use one or the other, but not both. However, note that **it's absolutely fine to use both thunks and either sagas or observables together**, because they solve different problems.

# 

# Билет №13

## Безопасность. Логические уязвимости. Примеры и предпосылки

Ссылки:

* <https://yadi.sk/d/O-3uvKmHtb8iug/%D0%91%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%98%D0%A2%D0%9C%D0%9E_2019-2020.mp4> - видео лекции
* <https://frontend-ifmo-2019.now.sh/webdev/14-security/#/> - презентация

## Угроза-

Потенциально возможное событие, которое посредством работы с компонентами сервиса, может нанести ущерб

Уязвимость-

Свойство сервиса, использование которого злоумышленником приводит к реализации угрозы

Атака-

Реализация угрозы, путём использования уязвимостей

Open Web Application Security Project (OWASP) - сообщество по безопасности веб-приложений  
Разработало список топ-10 популярных уязвимостей

Безопасность:

* Серверная
  + Логические уязвимости
    - Отсутствие разграничений прав доступа

Например, к каким-то файлам доступ должны иметь не все, но проверок нет

* + - Избыточная небезопасная функциональность

***Секретный урл***

<https://my-awesome-site.ru/admin> - переход к админскому функционалу (пример)

***Сканеры безопасности*** умеют перебрать url-ы и найти те, которые отвечают не 404, admin можно даже вручную понять и взломать

***Скандал с паролями в гугл-доках -*** документы с доступом по ссылке индексировались браузером

* + - Нарушение логики работы приложения

***Insecure direct object reference -*** проблемы с предоставлением пользователям прямого доступа к объектам (коррелирует с разграничением прав доступа)

*Примеры:*

* <https://online-shop.ru/payment/step/4> - можно миновать шаг оплаты и сразу перейти к получению
* <https://disk.yandex.ru/client/123> - доступ к данным клиента по ссылке
  + Ошибки эксплуатации (в следующих билетах)
  + Ошибки реализации (в следующих билетах)
* Клиентская

Предпосылки к появлению уязвимостей:

* **Сложности реализации**

Делаем что-то очень сложное, легко посадить багу

* **Плохо продуманная архитектура**

Как в сервисе с оплатой по шагам

* **Человеческий фактор**

Человек устал закрывать дыры в безопасности

# Билет №14

Ссылки:

* <https://yadi.sk/d/O-3uvKmHtb8iug/%D0%91%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%98%D0%A2%D0%9C%D0%9E_2019-2020.mp4> - видео лекции (с 17 минуты)
* <https://frontend-ifmo-2019.now.sh/webdev/14-security/#/23> - презентация

## Безопасность. Инъекции. Виды и способы борьбы с ними.

* Ошибки реализации
  + **Инъекции и небезопасная сериализация**

Инъекции - уязвимости, направленные на возможность внедрения управляющего кода в язык запросов

* SQL инъекции - самые популярные

const sql = `SELECT \* FROM locations WHERE id='${id}';`;

GET /locations?id=123

Казалось бы, получение локации по id - нормас

Но

GET /locations?id=123' or '1'='1

SELECT \* FROM locations WHERE id='123' or '1'='1';

Передали управляющую конструкцию, условие всегда верное - получили все локации

Могли например передать droptable и убить бд

Защита от SQL - инъекций

* Санитайзить / эскейпить данные

**Санитайзинг** - разрешение определенных управляющих последовательностей, но ограниченного количества (whitelist)

**Эскейпинг** - эскейпим последовательности - превращаем код в текст.

const { name } = req.body;

const escaped = name.replace('\'', '\'\'');

const sql = `

SELECT \* FROM locations WHERE name='${escaped}';

`;

Работает только в *частных случаях* - если в ваш сервер приходят

не только ковычки - эскейпите остальное тоже

* Использовать ORM

ORM - технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».По сути, ее задача - избавиться от необходимости писать SQL-код для взаимодействия в СУБД

ORM обычно реализованы таким образом, чтобы нельзя было произвести инъекцию при их использовании. Но нужно постоянно следить за чейнджлогами ORM, к-ым вы пользуетесь - бывает там появляются уязвимости.

* Использовать prepared statements

Prepared statement - механизм, который используется для многократного выполнения одинаковых или похожих запросов к базе данных с высокой эффективностью.

По сути функция:

PREPARE findLocation (text) AS

SELECT \* FROM locations WHERE name = $1;

EXECUTE findLocation('home');

Пример sql prepared statement

Суть в том, что такой запрос подготавливается один раз, а выполняется много (с разными аргументами).

Говорят, prepared statement под капотом безопасны

вот можно почитать https://stackoverflow.com/questions/1582161/how-does-a-preparedstatement-avoid-or-prevent-sql-injection

* Command injection

exec(`cd ${name}`);

name: 'suggest && sudo rm -rf /app/' - уничтожили репозиторий

**Разбивать сервис на микросервисы -** запуск на другом сервере, чем принятие запроса, запускающий ничего не знает и не сможет ничего украсть

**Не передавать пользовательский ввод в параметры**

**Не запускать новые процессы из пользовательских запросов**

**Изолировать запускаемый код -** как в 1 пункте

* CRLF - injection

Путь на фронте преобразуется в путь на бэке:

/handle => /backend/handle

/handle %20HTTP/1.0%0D%0AHost%3A%20test%0D%0A%0D%0A

Перевод строки

GET /backend/handle HTTP/1.0\r\n

Host: test\r\n

\r\n

HTTP/1.0\r\n

Host: backend\r\n

*Часть данных отбросится и может попасть в response*

* HTTP Header Splitting

То же, что в прошлом примере, но инъекция в header

Referer: test%0D%0ASet-Cookie:%20name=Alice%0D%0A\r\n

\r\n

Referer: test\r\n

Set-Cookie: name=Alice\r\n

* HTTP parameter contamination

Инъекция в параметр

GET /handle?payload=1%26login=admin

GET /backend?payload=1&login=admin&login=user

Почему так??

* Path traversal

GET /handler?filename=image.jpg

GET /handler?filename=../../../../../etc/passwd

***Нужно разграничивать права доступа к директориям в файловой системе***

* + **Открытые редиректы**
  + **Слабая криптография**
  + **Бинарные уязвимости**

Race conditions (гонки)

Активация промокода - можно успеть активировать два промокода до отметки об использовании

Нужно использовать транзакции, блокировки…

О Раскрытии информации

Что стоит скрывать:

* + Заголовки запроса в ответе сервера
  + Ссылки на внутреннюю документацию / описание внутренних структур
  + Язык и фреймворк
  + Отладочные данные в production
  + Traceback в production
  + Небезопасное логирование

О криптографии

Криптография !== Безопасность (если вы все зашифровали и положили пароль рядом, то криптография не спасет)

Последний рубеж (все взломали, тогда возможно в конце спасет шифровка)

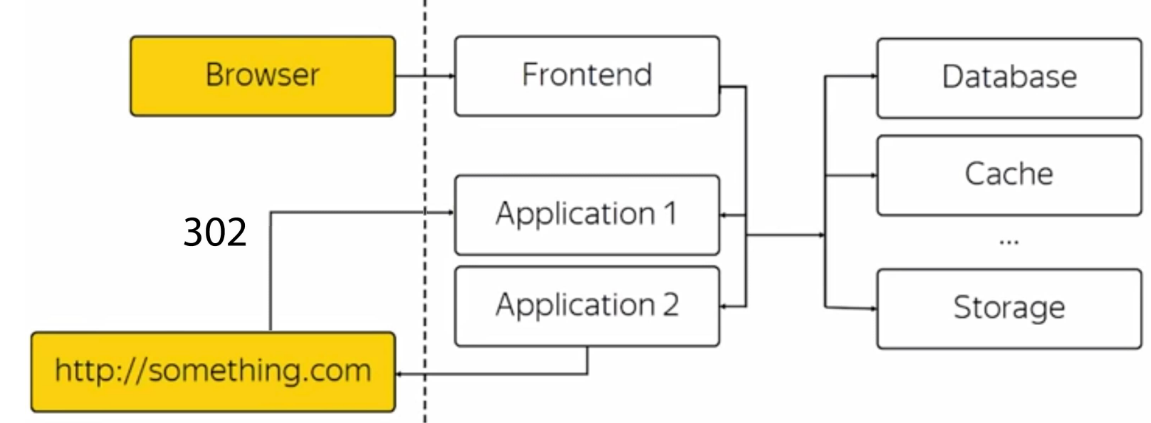
Нужно использовать соль (добавка к паролю, а потом хэшировать), так как многие хэш-функции уже посчитаны

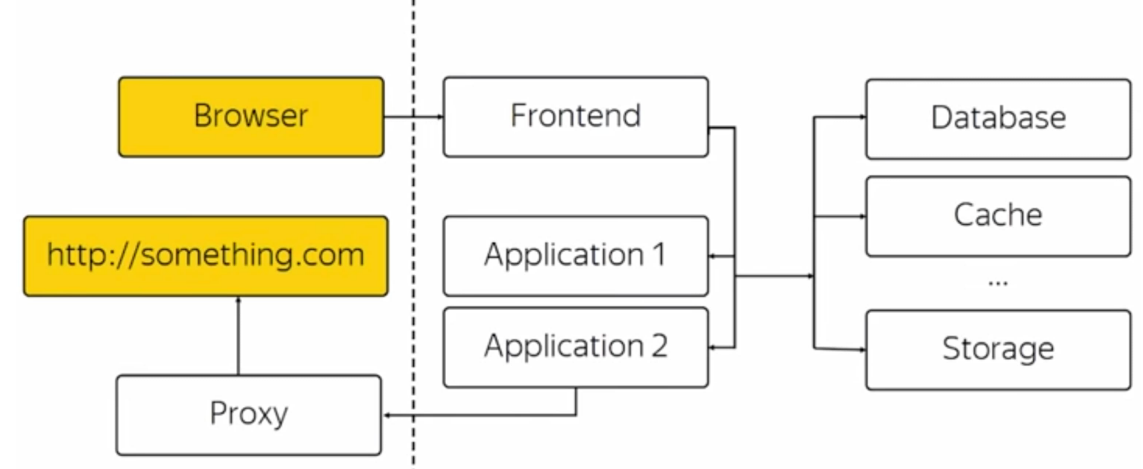
Denial of service - DOS-атака

Отправляем очень много запросов, сервер не справляется и падает

Ошибки архитектуры

Атака Server Side Request Forgery (SSRF) и защита от нее

Пример атаки SSRF, вызванной уязвимостью архитектуры. Есть сервис, который получает урл от пользователя (Browser), идет по данному урлу (<http://something.com>), берет полученные данные и возвращает их пользователю. Допустим <http://something.com> устроен так, что умеет редайректить запросы, тогда он может перенаправить запрос на наш сервис и попросить какие-то закрытые данные. Appllication 1 видит, что данные запрашивает Application 2 (тк запрос именно с него) и отдает их ему. Затем данные идут на Frontend и отдаются браузеру злоумышленника.



Фикс: будем использовать прокси вне сети сервиса, тогда Application 1 увидит, что данные запрашивает кто-то левый (Proxy) и ничего не отдаст.

# Билет №15

## Безопасность. CORS. СSP. Как обезопасить работу с Cookie.

Ссылки:

* <https://yadi.sk/d/O-3uvKmHtb8iug/%D0%91%D0%B5%D0%B7%D0%BE%D0%BF%D0%B0%D1%81%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%98%D0%A2%D0%9C%D0%9E_2019-2020.mp4> - видео лекции (с 1:13:50)
* <https://frontend-ifmo-2019.now.sh/webdev/14-security/#/63> - презентация

Cookie -

заголовок с авторизационными данными пользователя

Атрибуты - id, path, http-only, secure

Same Origin Policy -

Система, позволяющая делить разные url на origin’ы - протокол + домен + порт

http://a.yandex.ru/dir1

<http://a.yandex.ru/dir1/dir2>

https://a.yandex.ru/dir1

http://a.yandex.ru:8080/dir1

http://b.yandex.ru/dir1

Первые два имеют один origin, остальные разные

Запрещается получать данные и делать запросы к другим origin’ам

CORS -

Cross-Origin Resource Sharing

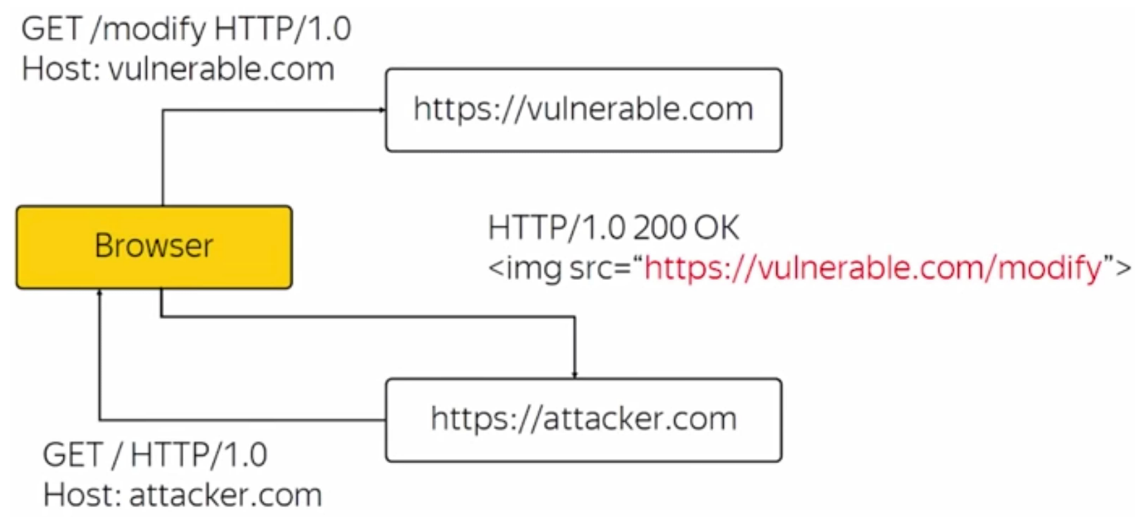
Система, позволяющая указать специальный заголовок на сервере “с этого сервера можно брать данные не смотря на то, что у него другой origin”

**Access-Control-Allow-Origin**: <https://attacker.com> - можно обменивать данными

**Access-Control-Allow-Credentials**: true - можно передавать авторизационные данные

CSRF-уязвимость -

Cross Site Request Forgery - наиболее частая и древнейшая уязвимость клиентской части



Есть клиент (Browser), авторизованный на vulnerable.com (клиент ввел логин/пароль и ему выставили cookie - теперь пользователь может в течение какого-то времени туда ходить со своей кукой и vulnerable.com будет считать, что это это пришел пользователь). Пусть у vulnerable.com есть хендлер /modify, который работает по методу Get(так делать ооочень плохо). Пусть браузер заходит на attacker.com и на этой странице есть

<img src=”<https://vulnerable.com/modify>”>. При рендеринге страницы браузер пользователя сделает Get запрос на <https://vulnerable.com/modify> и так как он там авторизован, то при запросе он передаст авторизационные данные. В итоге клиент сделал запрос /modify на vulnerable.com c какого-то левого сайта сам того не подозревая, это очень плохо.

Фикс: использование CSRF-токенов. CSRF-токен может быть полем в cookie или отдельным заголовком. По сути это трудно предсказуемая последовательность, которая генерируется сервисом (vulnerable.com) на каждый запрос. Пользователь может сделать запрос на сервис только если он передаст токен в заголовке. Тогда attacker.com не сможет сделать запрос, потому что не знает токен. Токен генерируется сервером на каждый запрос, при этом инвалидириует предыдущий токен.



XSS (Cross-site request scripting)-

Атака, при которой вредоносный код внедряется в код вашего сайта и может быть запущен (инъекция)

**Защита** -

* санитайзинг - позволять только конкретные теги/атрибуты
* эскейпинг на выходе
* бескуковый домен для пользовательских скриптов - песочница - изолированное место для запуска пользовательских скриптов
* Заголовок X-XSS-Protection - механизм защиты бразуера от этой атаки
* Content Security Policy

<XSS>’’;!—«<hr/>&{()} - вставьте это туда где хотите чекнуть XCC, если все развалится, то уязвимо

CSP (content security policy)-

Заголовок, говорящий пользователю, какие данные и откуда он может получать (с какого ориджина).

Content-Security-Policy:

default-src 'self';

img-src \*;

script-src: trusted.com;

media-src: <https://yandex.ru> - оттуда можно видюшки

Поддерживается не во всех браузерах

ClickJacking-



Полупрозрачная ссылка поверх основной.

X-Frame-Options: ALLOW-FROM http://trusted.com

X-Frame-Options: SAMEORIGIN

X-Frame-Options: DENY

Ограничения для использования frame’ов.

В google-chrome ALLOW-FROM не работает, можно использовать CSP frame-ancestors.

Многократная отправка формы-

Защищаемся капчей

**Наша защита-стартерпак:**

* **Унификация серверов**

Если один сервер и принимает, и обрабатывает, и отсылает, и все и сразу, больше вероятности что нас обкрадут

* **Не доверяем пользовательским данным**

Могут содержать управляющие последовательности, код, который все сломает

* **Автоматизация**

Автоматический анализ безопасности, сканеры.

* **Общие безопасные компоненты**

Используете протестированные безопасные компоненты, не пишите свои велосипеды

* **npm audit**

Проверит безопасность в зависимостях node js

# Билет №16

Процесс тестирования. Что такое тест-кейс, примеры тест-кейсов. Testdesign, BugReports, примеры багрепорта. Виды тестирования (всё из [тестирование ч1](https://docviewer.yandex.ru/view/534650980/?page=7&*=1ykQkTzKSZyZFj8jLrqLmHtj0%2B17InVybCI6InlhLWRpc2stcHVibGljOi8vM0l5ZkF3MlZaV0s5anVEUUJCcmIrTnExa0dUaU0zVGNVUVUzUitycm43emNwb29xdHFLQzBPZ05zNm1FOFJ0MnEvSjZicG1SeU9Kb25UM1ZvWG5EYWc9PSIsInRpdGxlIjoiMTYtdGVzdC0xLnBwdHgiLCJub2lmcmFtZSI6ZmFsc2UsInVpZCI6IjUzNDY1MDk4MCIsInRzIjoxNTk0MjI1NTE1NTI2LCJ5dSI6IjU4NjA2NTgyNTE1ODMzNDUwMTkifQ%3D%3D))

[Ссылка на презентацию в pdf](https://drive.google.com/file/d/1iSkPLHjRKkgFe59qGuair3jVeTT8LTvA/view?usp=sharing)

<http://www.protesting.ru/testing/> полезная ссылка

Что было в презентации, но нет в билете:

* требования (со стр 27)
* приоритизация (Сколько пользователей пострадает если это требование будет не реализовано/при реализации будет дефект? Сколько пользователей заметит, что не реализовано/сломано? Можно ли выполнить действие иначе?)

Другое про тестирование:

* [Автотесты Assert. Mocha. Организация тестов](https://frontend-ifmo-2019.now.sh/webdev/17-unit-test/index.html#/)
* (Была ещё часть 3, но нет презентации и видео)

### Процесс тестирования

**Цели**

* Предоставление обратной связи о состояния продукта
* Повышение вероятности отказоустойчивости приложения
* Повышение вероятности соответствия системы требованиям

**Задача**

Выявить проблемы, которые могут возникнуть при работе с продуктом и

сделать все возможное для улучшения его качества

Процесс, содержащий в себе все активности жизненного цикла, как динамические, так и статические, касающиеся планирования, подготовки и оценки программного продукта и связанных с этим результатов работ с целью определить, что они соответствуют описанным требованиям, показать, что они подходят для заявленных целей и для определения дефектов

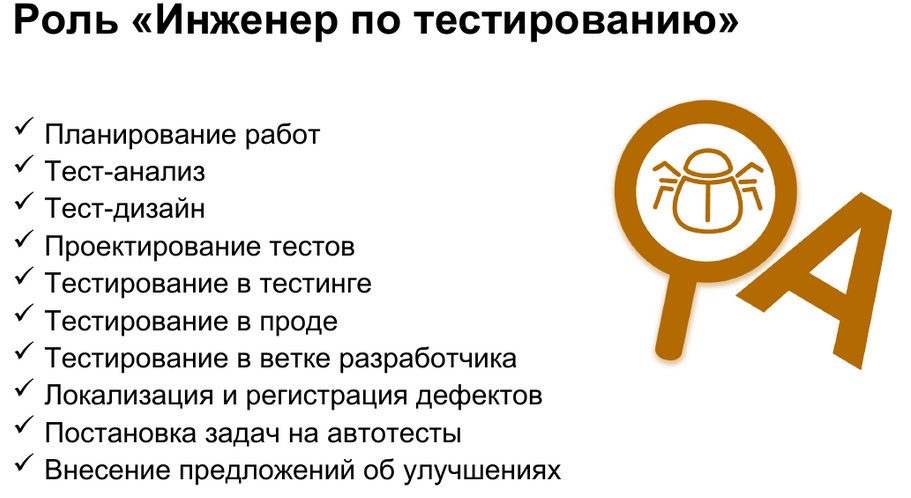
[ISTQB, глоссарий] (из презентации)

Несмотря на то, что тестировать долго и дорого, это в результате сэкономит время и деньги. Чем раньше обнаружен дефект, тем дешевле его править

Кто тестирует:



Ещё тестировать может и пользователь на уровне private alpha-beta-earlyaccess программ



### Testdesign

Это этап процесса тестирования ПО, на котором проектируются и создаются тестовые случаи (тест кейсы), в соответствии с определёнными ранее критериями качества и целями тестирования.

Главное: их должно быть достаточно, чтобы покрыть все требования(к функционалу и тд), но они не должны быть избыточны.

Техники тест-дизайна:

* **Классы эквивалентности -** техника ТД, где подбираются диапазоны входных данных такие, что все значения в рамках одного диапазона одинаково влияют на систему. То есть можно было бы взять одно значение из диапазона и считать, что был проверен весь диапазон.
* **Граничные значения**
* **Позитивное тестирование**

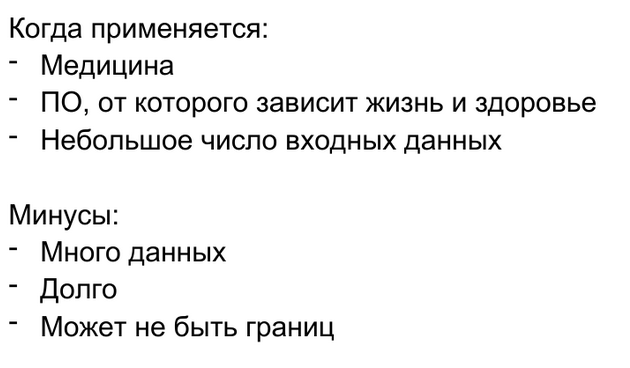
Тестирование на выполнение каких-либо условий

* **Негативное тестирование** (ещё называют предугадывание ошибки)

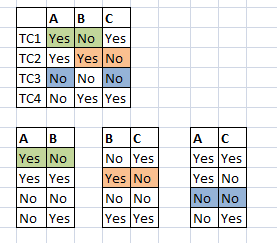
Это когда тест аналитик использует свои знания системы и способность к интерпретации спецификации на предмет того, чтобы "предугадать" при каких входных условиях система может выдать ошибку. По-простому: ассерт что на данных Х программа должна упасть.

Правило: Нельзя объединять две и больше негативных проверок. Потому что можно гарантировать только первую негативную проверку. Результат последующих предугадать уже нельзя так как система находится в невалидном состоянии, и первая ошибка уже могла повлиять на результат следующих проверок.

* **Исчерпывающее тестирование**

Проверить все возможные комбинации входных значений.  


* **Попарное тестирование**  
  Каждое тестируемое значение каждого из проверяемых параметров хотя бы единожды сочетается с каждым тестируемым значением всех остальных проверяемых параметров.   
  РУССКИМ ЯЗЫКОМ: попарное тестирование = перебрать все ПАРЫ тестируемых значений, и при этом МИНИМИЗИРОВАТЬ количество тестов.  
  Пример с презентации но МАКСИМАЛЬНО упрощенный:  
  Представим, что у нас есть параметры A, B и C принимающие значения Yes или No. Максимальное количество комбинаций значений этих параметров – 8. Но при использовании попарного тестирования достаточно четырех комбинаций, так как **учитываются всевозможные пары параметров (пара A и B, пара B и C, пара A и C):**



!!Достаточно внимательно вглядеться в картинку и все станет понятно!!

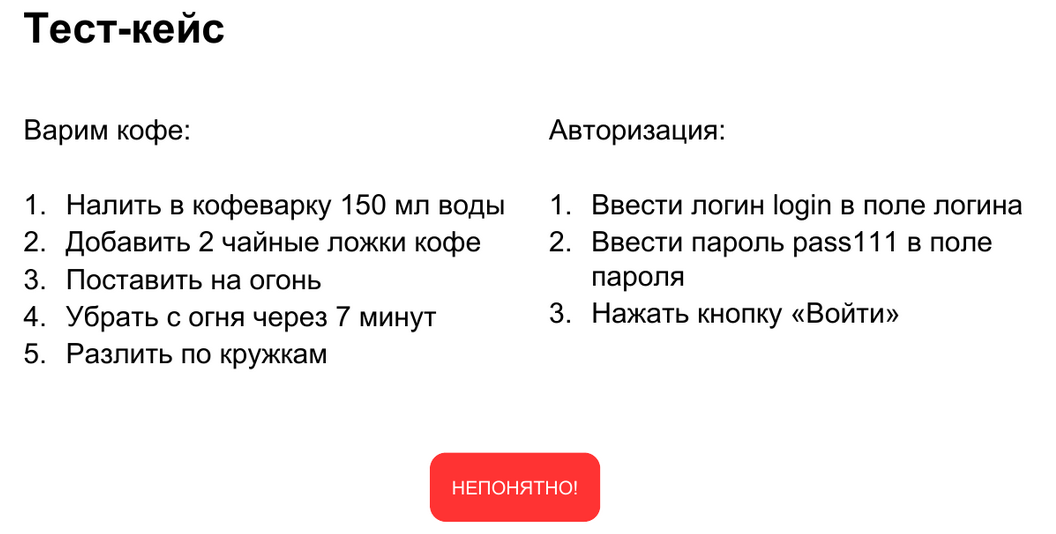
### Что такое тест-кейс, примеры тест-кейсов.

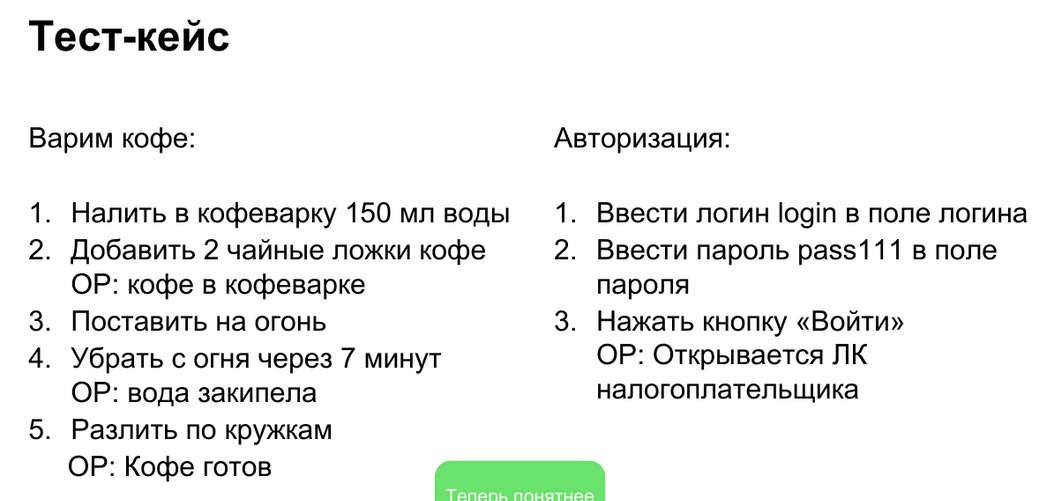
Тест-кейс: способ описания теста (шагов, конкретных условий и параметров), не более. Юзается в случаях когда проект живет долго, они сложные и долгие. ОР - ожидаемый результат.

с Хабра:

Тест кейс должен иметь 3 части:

* **PreConditions** - список действий, которые приводят систему к состоянию пригодному для проведения основной проверки. Либо список условий, выполнение которых говорит о том, что система находится в пригодном для проведения основного теста состояния.
* **Test Case Description** - список действий, переводящих систему из одного состояния в другое, для получения результата, на основании которого можно сделать вывод об удовлетворении реализации поставленным требованиям.
* **PostConditions** - список действий, переводящих систему в первоначальное состояние (состояние до проведения теста — initial state).

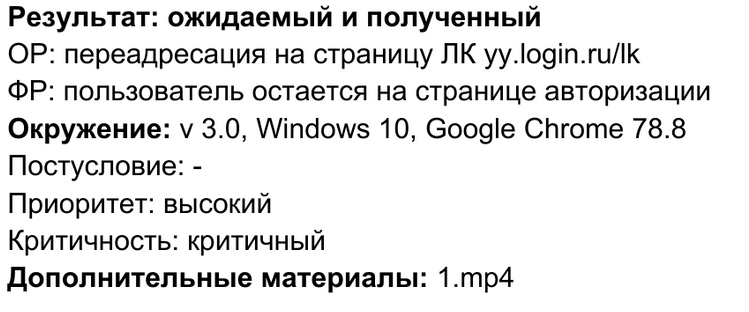
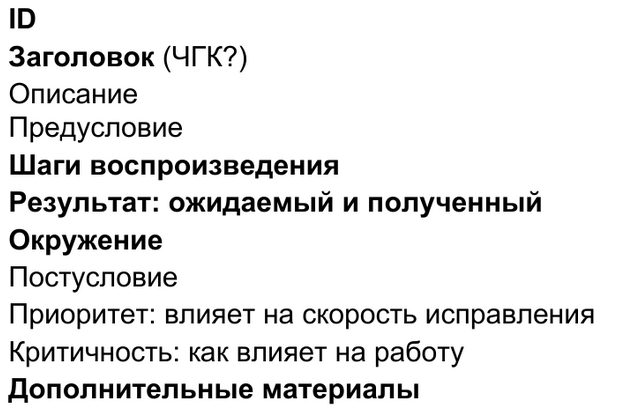
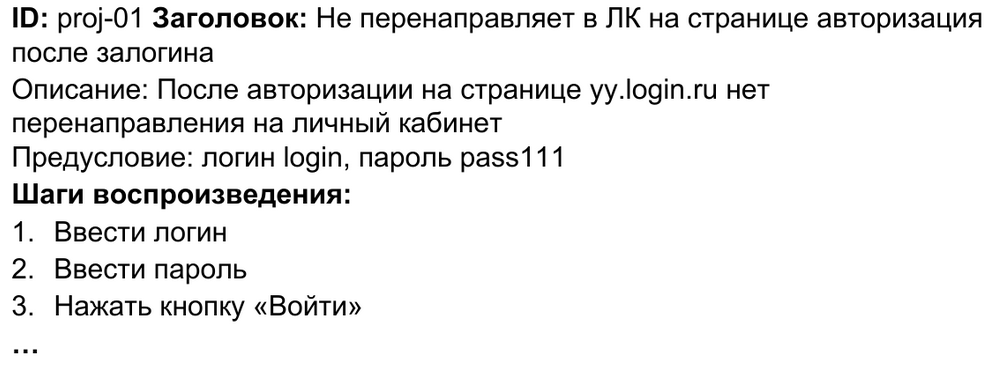




### BugReports, примеры багрепорта.

Пишут и читают все, ниже из чего bugreport состоит, максимально просто надо писать, чтобы все понимали. Заголовок отвечает на вопрос - что где когда, и должен быть уникальным и простым.

Пример



### Виды тестирования



Далее более подробное описание

**По целям:**

1. **Функциональное**

Функциональное тестирование — это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям. Русским языком - проверка что прога делает че тебе надо.

1. **Нефункциональные**

Нефункциональное тестирование – тестирование свойств, которые не относятся к функциональности системы. Данные свойства определяются нефункциональными требованиями, которые характеризуют продукт с таких сторон, как:

1. Безопасности
2. Установки

проверка успешности установки приложения, его настройки и удаления. Снижает риски потери пользовательских данных, потери работоспособности приложения и пр.

1. UI/UХ

имеется в виду самое понятие удобства и красоты, а не наличие кнопок. Наличие кнопок как-раз функциональное.

1. На отказ и восстановление

На предмет восстановления после ошибок, сбоев. Оценивание реакции защитных свойств приложения.

1. Конфигурационное

Как влияют разные настройки на приложение

1. Взаимодействия

(Между приложениями)

1. Производительности/нагрузочное

* Стресс

Для выявления максимально возможного пика, при котором система работает правильно.

* Стабильности  
  Проверка работоспособности приложения при длительном (многочасовом) тестировании со средним уровнем нагрузки.
* Объёмов

Получение оценки производительности при увеличении объемов данных в базе данных приложения

* Нагрузочное

Проверка работоспособности при стандартных нагрузках

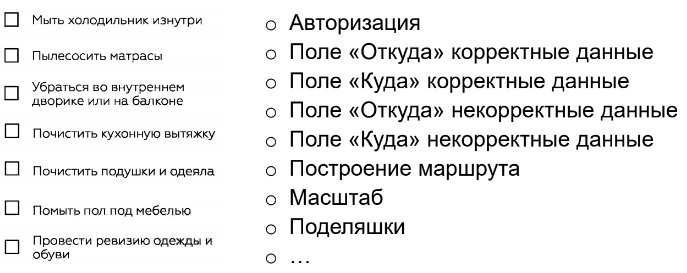
**По позитивности**

1. **+**
2. Важнее чем -.
3. **-**

**По формальности**

1. **По алгоритмам, по которым тесты выстраивались, 2 варианта**

* Созданы с помощью тест кейса
* С помощью чек-листа:



1. **Исследовательское**

Когда не знаем как устроено приложение, изучение проекта. Отличие от черного ящика - в процессе тестирования появляется новая информация о приложении и тест подстраивается под него. Больше [здесь](https://habr.com/ru/post/148479/)

1. **Специальное (свободное)**

**По знанию системы**

1. **Белый ящик**

На соответствие программного продукта требованиям со знанием внутренней структуры реализации системы (есть в наличии исходный код и технические спецификации).ь

1. **Серый ящик**

Про устройство что-то знаем, что-то нет

1. **Черный ящик**

Без знания внутренней структуры

**По хронологии выполнения**

1. **Первичное**
2. **Повторное**

После исправлений, например

1. **Регрессионное**

описано в следующей секции

1. **Приемочное**

Тесты перед заказчиком

**По изменениям:**

Выполняется после изменений, например после устранения дефектов.

1. **Дымовое**

"При вводе в эксплуатацию нового оборудования ("железа") считалось, что тестирование прошло удачно, если из установки не пошел дым."

Короткий цикл тестов, выполняемый для подтверждения того, что после сборки кода (нового или исправленного) устанавливаемое приложение, стартует и выполняет основные функции.

1. **Регрессионное**Проводится с целью проверить, не влияют ли новые функции, улучшения и исправленные дефекты на существующую функциональность продукта и не возникают ли старые дефекты.
2. **Повторное**
3. **Тестирование сборки**

Что приложение вообще устанавливается

1. **Санитарное/проверка исправности**

Узконаправленное тестирование доказывающее, что конкретная функция работает согласно заявленным в спецификации требованиям. Является подмножеством регрессионного тестирования. Обычно выполняется вручную.

В отличии от дымового (Smoke testing), санитарное тестирование (Sanity testing) направлено вглубь проверяемой функции, в то время как дымовое направлено вширь, для покрытия тестами как можно большего функционала в кратчайшие сроки.

**По степени автоматизации**

1. **Ручное**
2. **Автоматизированное**

**По уровню тестирования**

1. **Модульное**

Как работает отдельно реализованный модуль

1. **Интеграционные тесты**Как работает несколько модулей между собой
2. **Системное тестирование**

Проверка всей системы в целом

### Требования

* Анализируем требования

Что за ПО?

Зачем?

Кто пользователь?

Что ПО должно делать?

Что должен делать каждый элемент ПО?

* Декомпозировать

Каждое требование раскладывается на элементы

Требования: авторизация перенаправляет на ЛК

• Есть раздел авторизации

• Есть раздел регистрации/добавления пользователей

• Есть система ролей

* Серые зоны и уточнение требований

Поиск несоответствий, недостаточности информации, ошибок и т.д.

Серые зоны

Скрытые требования

Требований нет

* Визуализация требований

Создание наглядной схемы всех объектов и элементов тестирования

Объект тестирования – система или элемент системы, которую

необходимо протестировать

Элемент тестирования – часть объекта тестирования

* Приоритезация требований

Какая степень влияния каждого требования, с чего начать работать, что

нужно для первой версии, что нужно для минимального покрытия

потребностей пользователя

# [Билет №17](https://frontend-ifmo-2019.now.sh/webdev/17-client-performance/assets/player/KeynoteDHTMLPlayer.html)

<https://yadi.sk/d/g8Oc_tshGk3gdA/%D0%9F%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C.mp4> - видео

## Оптимизация загрузки страницы. Ключевые моменты, способы оптимизации. Инструменты для измерения скорости загрузки, измеряемые показатели.

### Зачем оптимизировать производительность

* Удержание пользователей
* Конверсия (как правило, посещения, от которых появляется прибыль)
* UX

### [RAIL — performance model](https://web.dev/rail/)

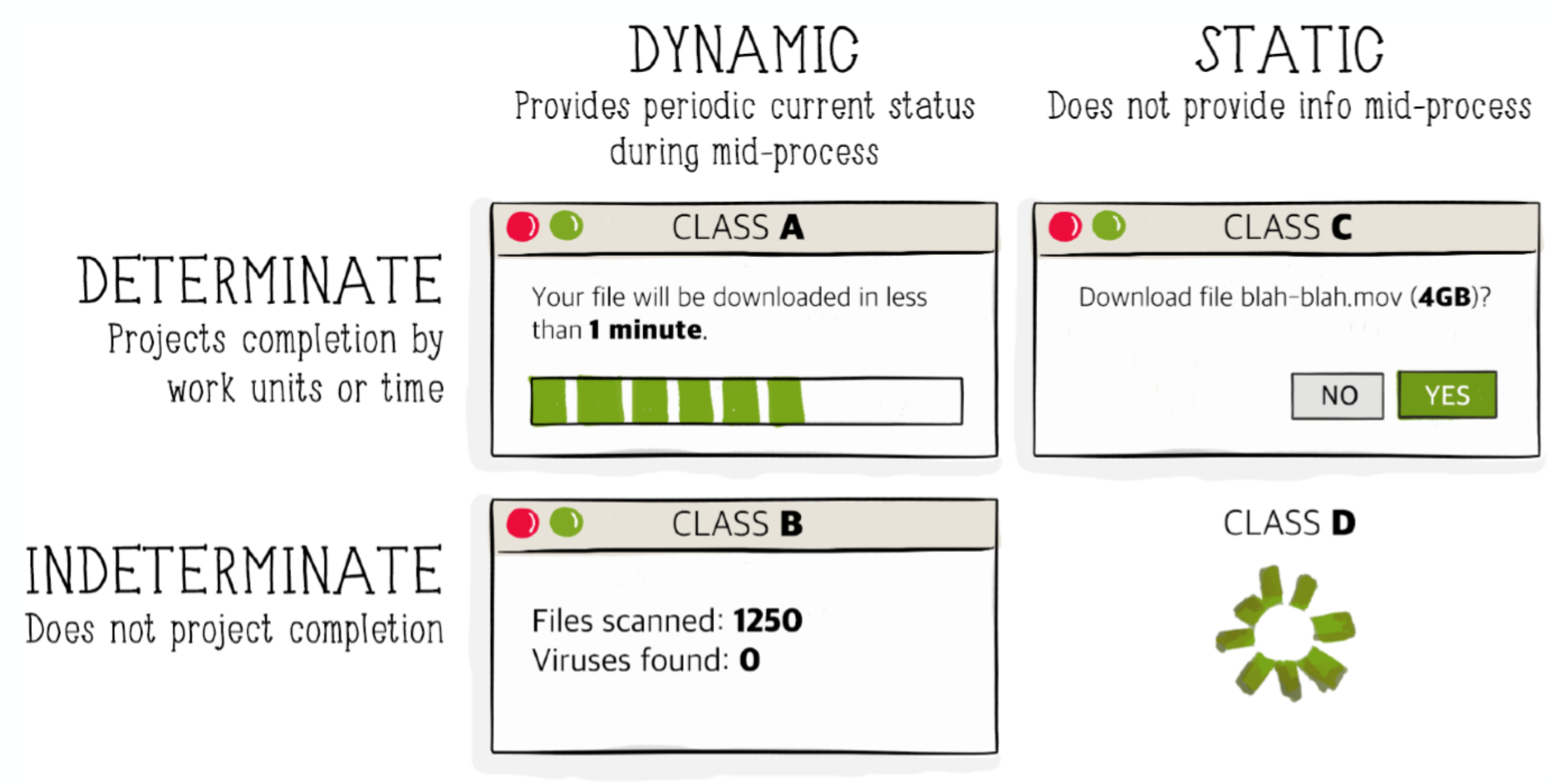
(последним пунктом в строке указано оптимальное время)

* **R**esponse — отклик на действие — 100 мс
* **A**nimation — скорость анимации — 60 fps/10 мс на кадр
* **I**dle — состояние свободного основного потока — нужно максимизировать это время
* **L**oad — время на доставку контента — 1000 мс

### Воспринимаемая производительность

Идея в том, что объективно прошедшее время воспринимается по-разному в зависимости от того, что происходит на экране. То есть, например, 10 секунд белого экрана могут субъективно восприниматься пользователем сложнее, чем 20 секунд с прогресс-баром.

Классификация прогресс-баров:



#### Optimistic UI

Давайте будем отображать результат действия до его реального исполнения.

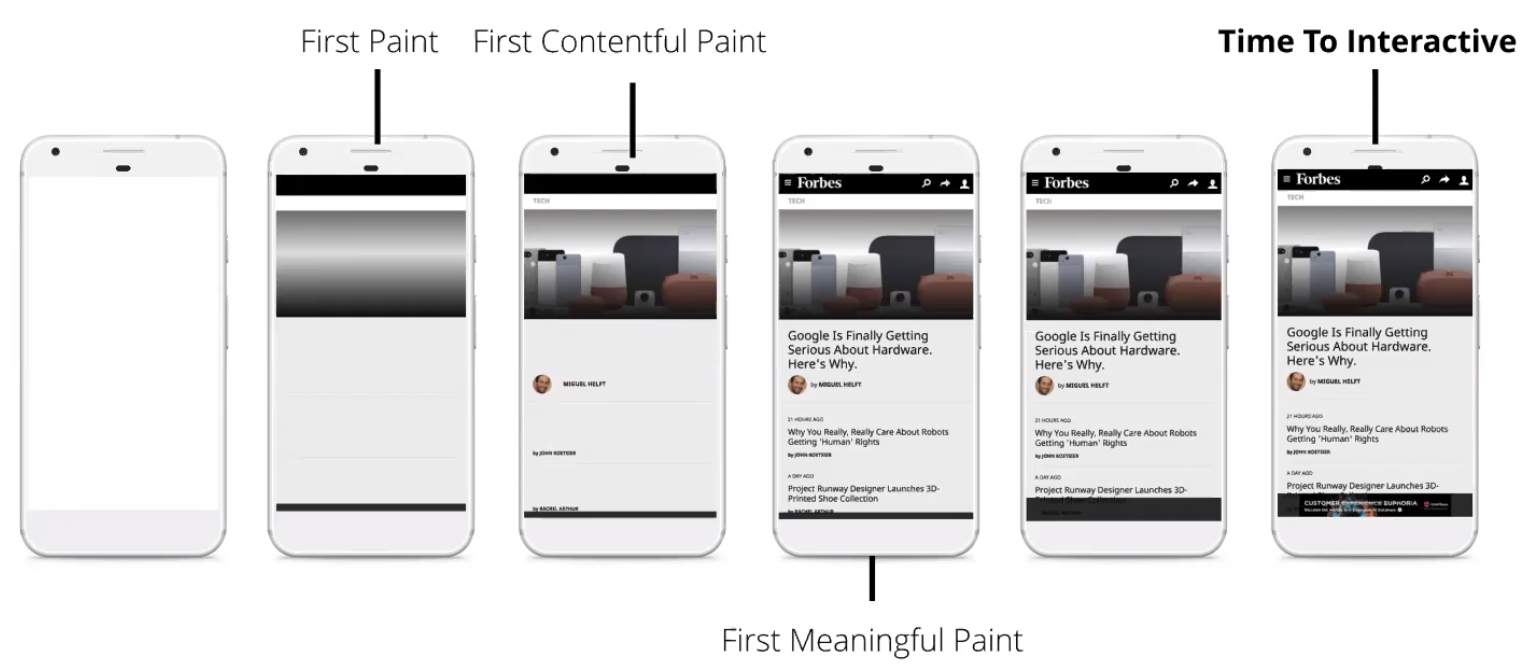
Используется для не особо критичных действий — например, поставить лайк в соцсети. При нажатии на кнопку отобразим, что лайк поставился и параллельно с этим отправим запрос. Если зафейлится — откатим изменения и покажем сообщение об ошибке.

### Измерение производительности — критерии

1. Time to first byte — время от начала запроса до получения первого байта с сервера
2. First paint — время от начала загрузки до первого рендера (например, отрендерился бэкграунд, но пока без элементов)
3. First contentful paint — время от начала загрузки до первого появления видимого контента (какого-то), пользователь будет видеть, что загрузка идёт
4. First meaningful paint — время от начала загрузки до отображения основного контента
5. First interactive (First CPU idle) — некоторые элементы готовы реагировать на действия пользователя

\* освободился основной поток, закончились сетевые запросы \*

1. Time to interactive — 5 секунд с последней задачи и последнего сетевого запроса



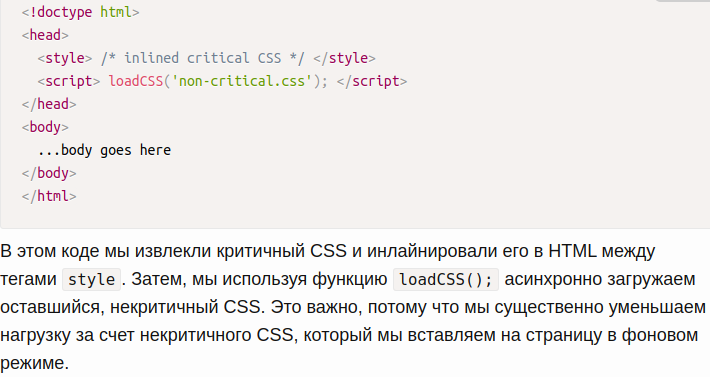
### Инструменты

1. DevTools браузера (вкладки Network, Performance, Audit)
   1. Network показывает инфу по запросам
   2. Performance показывает различные тайминги
   3. Audit показывает тайминги вышеописанных критериев, даёт общую оценку и предлагает пути решения проблем
2. Яндекс.Метрика и прочие аналитики
   1. Показывают статистику реальных пользователей
   2. География, условия сети, устройства, браузеры
3. Navigation Timing API
   1. Позволяет получить из кода различные показатели, недоступные другими способами
   2. Время выгрузки предыдущей страницы, время DNS запроса, время полной загрузки страницы
4. Также есть сервисы, которые дают рекомендации: PageSpeed Insights, Lighthouse

### 

### Оптимизация

1. Уменьшение количества запросов
   * Инлайн критических ресурсов в HTML
     + [WebPack URL Loader](https://github.com/webpack-contrib/url-loader) — используется для инлайна маленьких изображений, например иконок
     + Преобразует файлы в Base64
     + Заинлайненные изображения не кешируются
     + Большие файлы инлайнить не нужно, ибо будут замедлять загрузку страницы



* + Кеширование
  + Использование CDN (Content Delivery Network — распределённое хранение данных, что позволяет получать данные с ближайшего географически сервера)
  + Объединение ресурсов одного типа в бандлы/спрайты
    - Bundle — объединение нескольких HTML в один файл
    - Спрайт — объединение svg-шек, например
  + Использовать HTTP/2 и его ServerPush
    - HTTP/2 позволяет серверу “пушить” клиенту не только те данные, которые он запросил — таким образом, на один запрос нам приходит сразу пачка результатов

1. Уменьшение объёма ресурсов
   * Компрессия файлов (gzip, brotli-webpack-plugin)
     + brotli — примерно на 20% лучше сжимает, чем gzip; IE11 не поддерживает (на момент создания презентации)
     + brotli-webpack-plugin для настройки сжатия
   * Минификация кода и изображений
     + Оптимизация изображений — [image-webpack-loader](https://github.com/tcoopman/image-webpack-loader)

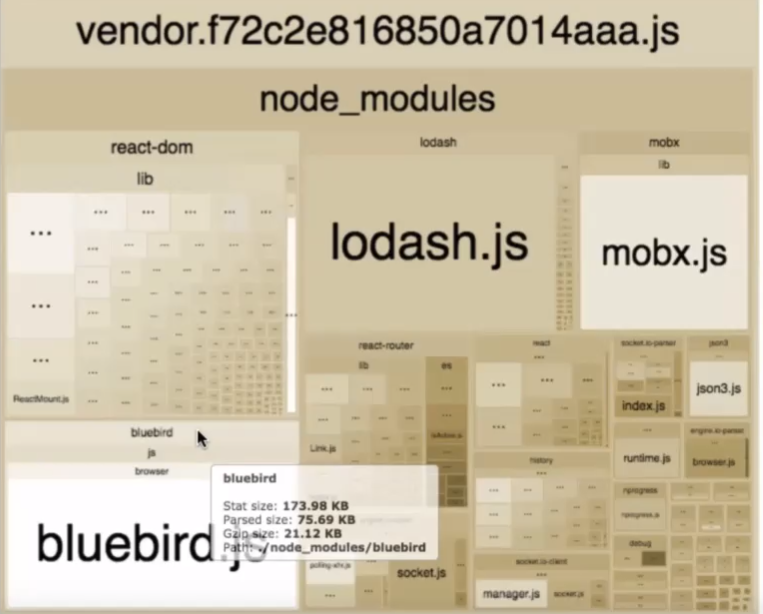
идет с file-loader, который сначала файлы качает. Оптимизация замедляет сборку, используется в продакшне.

* + - **Лишние пробелы, табуляция, переносы строк** и комментарии в коде применяются для лучшего его восприятия разработчиками при написании и редактировании, но **«раздувают» его объём**, увеличивая общий **вес веб-страницы**. Поэтому целесообразно их **удалять перед применением на сайт**е, оставляя копию оригинала для возможного редактирования.

Минификация кода **включена для production-сборок по умолчанию с Webpack 4+** (только для продакшн, потому что минификация замедляет время билда); есть также [terser-webpack-plugin](https://webpack.js.org/plugins/terser-webpack-plugin/)

* + Использование SVG
  + Tree shaking — удаление неиспользуемого кода
    - [Гайд на webpack.js.org](https://webpack.js.org/guides/tree-shaking/)
    - Работает с ES2015 import-export синтаксисом
    - sideEffects-оптимизация — вырезает файлы и модули целиком, если они не используются и не имеют сайд-эффектов (лучше про это почитать в гайде — как я понял, имеет ли модуль сайд-эффекты, нужно указывать руками в конфигах)
  + Постепенная подгрузка контента при скроллинге страницы
  + Виртуализация больших списков
    - Суть в том, что мы рендерим только тот контент списка, который мы видим, что ускоряет отрисовку для больших списков (если у нас 500 элементов, очевидно мы не хотим рендерить всё сразу)
    - [Либа (react-virtualized)](https://github.com/bvaughn/react-virtualized)
  + Webpack Bundle Analyzer — проверка того, что попало в бандл — ненужные зависимости и тяжелые зависимости.

bundler - это инструмент для сборки модулей в единые пакеты, имеющий доступ к файловой системе. Получающиеся пакеты совместимы с браузером, которому не нужен



1. Не допускать блокирование отрисовки
   * Стили в шапку, скрипты вниз (либо асинхронно с помощью аттрибутов async/defer) async - асинхронно в любом порядке, defer - асинхронно по порядку
     + Потому что когда парсер встречает <script>, парсинг останавливается и начинает выполняться скрипт
   * Загрузка только необходимого
   * Использовать минифицированные версии библиотек
   * Обращать внимание на шрифты
     + FOUT (Flash Of Unstyled Text) — мелькание стандартного текста
     + FOIT (Flast Of Invisible Text) — мелькание невидимого текста
     + Шо це таке вище? У шрифтов есть определённая последовательность отображения:
       1. Период блокировки шрифта — шрифт не загружен ⇒ рендерится невидимый запасной шрифт. Если основной за это время подгрузится, то используется он.
       2. Период подмены шрифта — шрифт не загружен ⇒ рендерится запасной шрифт. Если за это время подгрузится основной, используется он.
       3. Период отказа шрифта — шрифт не загружен ⇒ загрузка считается неудачной и используется запасной шрифт
     + Регулируются эти периоды свойством font-display
       1. block — короткий (1) и ∞ (2) (ждем загрузку красивого с запасным)
       2. swap — нет (1) и ∞ (2)
       3. fallback — короткие (1) и (2)
       4. optional — короткий (1) и нет (2)
2. Отзывчивость анимации
   * Использовать CSS-анимации (быстрее JS)
   * Не дёргать DOM без причины (перестроения требуют рассчетов, это долго)
   * Просто забейте на анимацию для пользователей со слабыми устройствами
   * Разбивание сложных вычислений на части (чтобы между выполнениями были периоды отзывчивости ~100ms и пользователь не скучал)
   * Выносить вычисления в отдельный поток
   * Загружать ресурсы заранее (prefetch/preload) ((подробно рассказывалось в том семе, кажется))
     + <link rel=”X” href=”${url}”> внутри <head> блока, где X — это:
     + dns-prefetch — зарезолвить имя домена
     + preconnect — инициация коннекта (handshake) (установка TCP-соединения)
     + prefetch — загрузка файлов
     + prerender — загрузка страницы целиком



1. Если всё описанное выше уже значимо не помогает, то надо уже менять сценарии
   * Например, подключать не критичный контент асинхронно

# 

# [Билет №18](https://frontend-ifmo-2019.now.sh/webdev/17-client-performance/assets/player/KeynoteDHTMLPlayer.html#59)

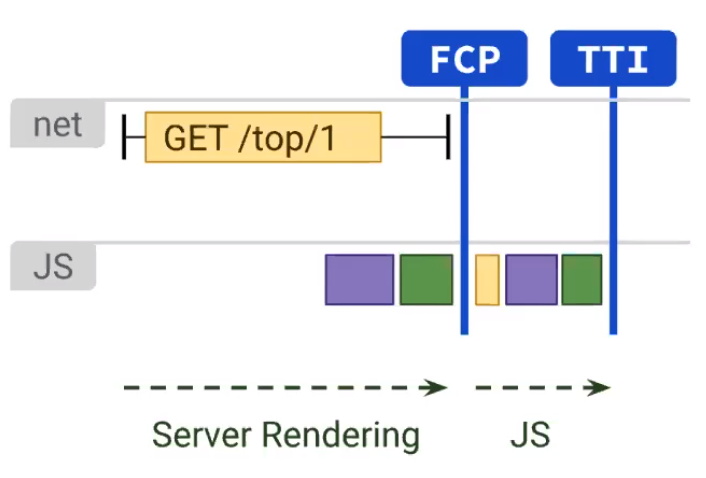
## Стратегии рендеринга. Отличия и преимущества. Профилирование компонентов.

<https://yadi.sk/d/g8Oc_tshGk3gdA/%D0%9F%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C.mp4> - видео (тайминг 1:08:35)  
<https://frontend-ifmo-2019.now.sh/webdev/17-client-performance/assets/player/KeynoteDHTMLPlayer.html#59> - лекция (59 и до конца)

Подходы к загрузке и отрисовке данных:

### Server-side rendering (SSR)

Рендеринг на стороне сервера — генерируем HTML на сервере и возвращаем на фронт

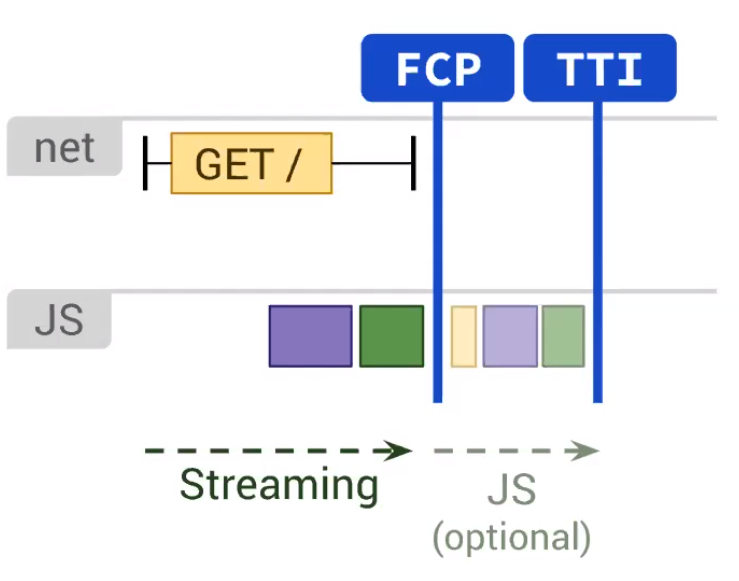


|  |  |
| --- | --- |
| **Плюсы** | **Минусы** |
| Быстрый First Paint (FP) | Дольше Time To First Byte (TTFB) |
| Быстрый First Contentful Paint (FCP) | Часто нужно отдельно настраивать и поддерживать (например, React настроен на clien-side-rendering) |
| Обычно быстрый Time To Interact (TTI) |  |

Плохо если медленный сервер, JS вставляется ссылкой в HTML и подгружается на фронте, часто нужно специально настраивать под различные фреймворки (например React, с ним лучше client-side rendering)

### Static rendering

Генерируем полный HTML при сборке

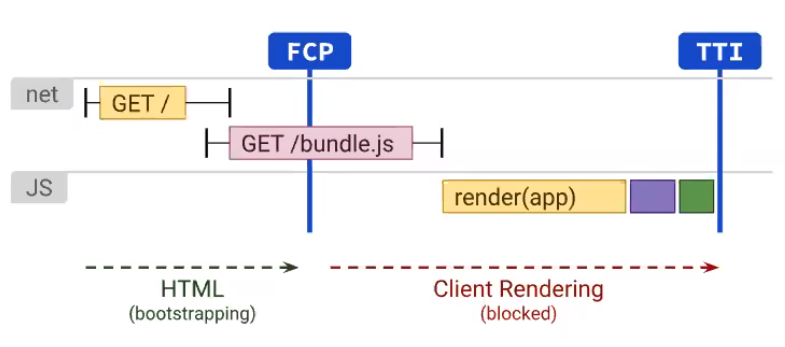


|  |  |
| --- | --- |
| **Плюсы** | **Минусы** |
| Быстрый FP, FCP, TTI, TTFB | Отдельный файл для каждой HTML страницы |
|  | Не подходит для сайтов с большим количеством страниц |

Самый простой метод, подходит для простых сайтов (мало страниц, нет JS, обращений к DB)

### Client-side rendering (CSR)

Вся логика, выборка данных, шаблоны обрабатываются в браузере

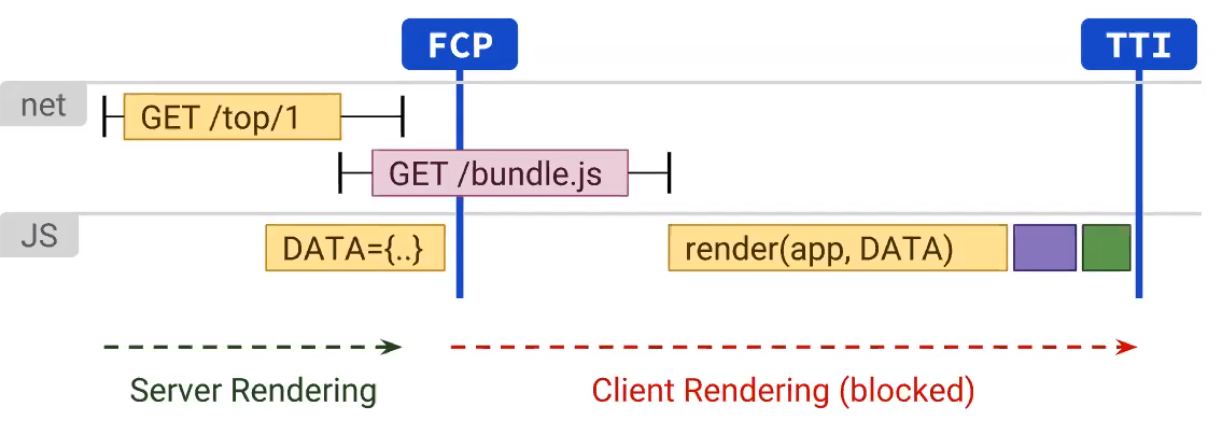


|  |  |
| --- | --- |
| **Плюсы** | **Минусы** |
| Быстрый TTFB и FCP | Долгий TTI |
|  | Замедляется с увеличением количества и сложности кода |

Отдаем минимальную страницу с <head> и <div class=”root”/>, дальше все обычно рисуется фреймворком. Быстрое FCP, так как фреймворки оптимизированы. Долгий TTI, так как после построения DOM-дерева начинают подключаться обработчики событий и JS, которых обычно много в фреймворках. Больше интерактивности на сайте - дольше подгрузка.

### SSR with rehydration

SSR + CSR



|  |  |
| --- | --- |
| **Плюсы** | **Минусы** |
| Более быстрый FCP, чем при SSR | Может ухудшить TTI и TTFB |
|  | Двойная загрузка данных |
|  | Страница может быть полностью прорисована, но неинтерактивна |

Запрос страницы -> обработка на сервере (как SSR) -> отправляет клиенту страницу + данные -> клиент подгружает данные в отрендеренную страницу. Данные используются для рендеринга на сервере и отправляются на клиент - двойная загрузка - большой трафик. Основной поток может быть долго заблокирован - плохой TTI. Чаще используется частично.

Пример - зашли на сайт с поиском, тыкаем в поисковую строку - не тыкается, печатаем - не печатается, заблокирован поток и ждет клиентского рендеринга.

# Билет №19

## Cookies, web storage, IndexedDB. Особенности, различия

Ссылка на слайды: <https://frontend-ifmo-2019.now.sh/webdev/20-webapi/index.html#/>

Полезные ссылки:

* <https://learn.javascript.ru/cookie>
* <https://learn.javascript.ru/localstorage>
* <https://learn.javascript.ru/indexeddb>

### Cookies:

**Cookie** - это небольшие строки данных, которые хранятся непосредственно в браузере. Они являются частью HTTP-протокола, определённого в спецификации [RFC 6265](https://tools.ietf.org/html/rfc6265).

Куки обычно устанавливаются веб-сервером при помощи заголовка **Set-Cookie**. Затем браузер будет автоматически добавлять их в (почти) каждый запрос на тот же домен при помощи заголовка **Cookie**.

Один из наиболее частых случаев использования куки – это **аутентификация**:

1. При входе на сайт сервер отсылает в ответ HTTP-заголовок **Set-Cookie** для того, чтобы установить куки со специальным уникальным идентификатором сессии (“**session** **identifier**”).
2. Во время следующего запроса к этому же домену браузер посылает на сервер HTTP-заголовок **Cookie**.
3. Таким образом, сервер понимает, кто сделал запрос.

Обращаться непосредственно к “печенькам” в **JS** можно посредством свойства **document.cookie**.

Значение **document.cookie** состоит из пар **ключ=значение**, разделённых **“;”**. Каждая пара представляет собой отдельное куки. Поэтому код:

**alert( document.cookie ); // cookie1=value1; cookie2=value2;...** - выведет все куки.

Чтобы найти определённое куки, достаточно разбить строку из **document.cookie** по **“;,”** и затем найти нужный ключ. Для этого мы можем использовать как регулярные выражения, так и функции для обработки массивов.

Мы можем писать в **document.cookie**. Но это не просто данные, а акcессор (геттер/сеттер). Присваивание обрабатывается особым образом.

Запись в **document.cookie** обновит только упомянутые в ней куки, но при этом не затронет все остальные:

**document.cookie = "user=John";** - обновит лишь одно куки.

На запись правда существуют ограничения:

* После **encodeURIComponent** пара **name=value** не должна занимать более **4Кб**. Таким образом, мы не можем хранить в куки большие данные.
* Общее количество куки на **один** домен ограничивается примерно **20+**. Точное ограничение зависит от конкретного браузера.

У куки есть ряд настроек, многие из которых важны и должны быть установлены.

* **path=/mypath -** URL-префикс пути, куки будут доступны для страниц под этим путём. Должен быть абсолютным. По умолчанию используется текущий путь. Как правило, указывают в качестве пути корень **path=/**, чтобы наше куки было доступно на всех страницах сайта.
* **domain=site.com** - Домен, на котором доступны наши куки. На практике, однако, есть ограничения – мы не можем указать здесь какой угодно домен. По умолчанию куки доступно лишь тому домену, который его установил и мы не сможем получить эти куки на поддомене. Но с помощью этой опции - можно, нужно лишь указать корневой домен.
* **expires=Tue, 19 Jan 2038 03:14:07 GMT** - дата истечения срока действия куки, когда браузер удалит его автоматически. Дата должна быть точно в этом формате, во временной зоне GMT. Если мы установим прошедшую дату, то куки будет удалено.
* **max-age=3600** - Альтернатива **expires**, определяет срок действия куки в секундах с текущего момента. Если задан ноль или отрицательное значение, то куки будет удалено.
* **secure** - Куки следует передавать только по HTTPS-протоколу. По умолчанию куки, установленные сайтом **http://site.com**, также будут доступны на сайте **https://site.com** и наоборот. С этой настройкой, если куки будет установлено на сайте **https://site.com**, то оно не будет доступно на том же сайте с протоколом HTTP, как **http://site.com**.
* **httpOnly** - веб-сервер использует заголовок **Set-Cookie** для установки куки. И он может установить настройку **httpOnly**. Эта настройка запрещает любой доступ к куки из **JavaScript**. Мы не можем видеть такое куки или манипулировать им с помощью **document.cookie**. Эта настройка используется в качестве меры предосторожности от определённых атак, когда хакер внедряет свой собственный **JavaScript**-код в страницу и ждёт, когда пользователь посетит её.

Для работы с **cookies** есть удобное **API** - [**js-cookie**](https://github.com/js-cookie/js-cookie).

**Cookies.set('subject', 'Song', { expires: 7, path: '' });**

**Cookies.get('subject');**

**Cookies.remove('subject');**

Также в связке в [**express**](https://expressjs.com/ru/) можно реализовать отправку на сервер примерно так:

**const express = require('express')**

**const app = express();**

**const cookieParser = require('cookie-parser');**

**app.use(cookieParser());**

**app.use((req, res) => {**

**console.log(req.cookies);**

**// { subject: 'Song' }**

**});**

А отправку на клиент:

**var express = require('express')**

**var app = express();**

**app.use((req, res) => {**

**res.cookie('subject', 'Song', { path: '/users' });**

**});**

**Советы** при работе с “печеньками”:

Также важно отметить что заголов не сжимаются(у HTTP 2 и HTTP 3 есть сжатие). Полезно для статики использовать **cookieless** домены (CDN).

В **cookie** храните **id**, по которому можно на сервере получить полные данные.

### WebStorage:

**WebStorage** - программные методы и протоколы веб-приложения, используемые для хранения данных в браузере. Рассматриваются объекты веб-хранилища **localStorage** и **sessionStorage**, которыепозволяют хранить пары ключ/значение в браузере.

**SessionStorage** – хранит данные до окончания сессии (до закрытия вкладки/окна).

**LocalStorage** – хранит данные перманентно, пока скрипт или пользователь не удалит их.

Что в них важно – данные, которые в них записаны, сохраняются после обновления страницы (в случае **sessionStorage**) и даже после перезапуска браузера (при использовании **localStorage**).

* В отличие от куки, объекты веб-хранилища не отправляются на сервер при каждом запросе. Поэтому мы можем хранить гораздо больше данных. Большинство браузеров могут сохранить как минимум 2 мегабайта данных (или больше, максимум 10мб), и этот размер можно поменять в настройках.
* Ещё одно отличие от куки – сервер не может манипулировать объектами хранилища через HTTP-заголовки. Всё делается при помощи **JavaScript**.
* Хранилище привязано к источнику (домен/протокол/порт). Это значит, что разные протоколы или поддомены определяют разные объекты хранилища, и они не могут получить доступ к данным друг друга.

Объекты хранилища **localStorage** и **sessionStorage** предоставляют одинаковые методы и свойства:

* **setItem(key, value)** – сохранить пару ключ/значение.
* **getItem(key)** – получить данные по ключу **key**.
* **removeItem(key)** – удалить данные с ключом **key**.
* **clear()** – удалить всё.
* **key(index)** – получить ключ на заданной позиции.
* **length** – количество элементов в хранилище.

Интерфейс похож на **Map**, но также запоминается порядок элементов, и можно получить доступ к элементу по индексу.

Ключ и значение должны быть **строками**.

Если мы используем любой другой тип, например число или объект, то он автоматически **преобразуется** в строку

Основные особенности **localStorage**:

* Этот объект один на все вкладки и окна в рамках источника (один и тот же домен/протокол/порт).
* Данные не имеют срока давности, по которому истекают и удаляются. Сохраняются после перезапуска браузера и даже ОС.

Особенности **sessionStorage**:

* **sessionStorage** существует только в рамках текущей вкладки браузера.
  + Другая вкладка с той же страницей будет иметь другое хранилище.
  + Но оно разделяется между ифреймами на той же вкладке (при условии, что они из одного и того же источника). IFrame — это компонент HTML-элемента, который позволяет встраивать документы, видео и интерактивные медиафайлы на страницу. Делая это, вы можете отобразить дополнительную веб-страницу на главной странице. Например, видос с ютуба встроенный в страницу.
* Данные продолжают существовать после перезагрузки страницы, но не после закрытия/открытия вкладки.

Когда обновляются данные в **localStorage** или **sessionStorage**, генерируется событие [**storage**](https://www.w3.org/TR/webstorage/#the-storage-event) со следующими свойствами:

* **key** – ключ, который обновился (**null**, если вызван **.clear()**).
* **oldValue** – старое значение (**null**, если ключ добавлен впервые).
* **newValue** – новое значение (**null**, если ключ был удалён).
* **url** – url документа, где произошло обновление.
* **storageArea** – объект **localStorage** или **sessionStorage**, где произошло обновление.

Важно! Событие срабатывает на всех остальных объектах **window**, где доступно хранилище, кроме того окна, которое его вызвало. Это позволяет разным окнам одного источника обмениваться сообщениями.

Используется в основном для хранения настроек или промежуточных данных.

Имеет **синхронный** интерфейс, из-за чего довольно медленная штука.

### IndexedDB:

**IndexedDB** – это встроенная база данных, более мощная, чем **localStorage**.

* Хранилище ключей/значений: доступны несколько типов ключей, а значения могут быть (почти) любыми.
* Поддерживает транзакции для надёжности.
* Поддерживает запросы в диапазоне ключей и индексы.
* Позволяет хранить больше данных, чем **localStorage**.

Для традиционных клиент-серверных приложений эта мощность обычно чрезмерна. **IndexedDB** предназначена для оффлайн приложений, можно совмещать с **ServiceWorkers** и другими технологиями

Для начала работы с **IndexedDB** нужно открыть базу данных.

Синтаксис:

**let openRequest = indexedDB.open(name, version);**

* **name** – название базы данных, строка.
* **version** – версия базы данных, положительное целое число, по умолчанию 1.

Метод open возвращает объект **IDBOpenDBRequest** с тремя обработчиками: **onerror**, **onsuccess**, **onupgradeneeded**

* **success**: база данных готова к работе, готов “объект базы данных” **openRequest.result**, его следует использовать для дальнейших вызовов.
* **error**: не удалось открыть базу данных.
* **upgradeneeded**: база открыта, но её схема устарела.

**IndexedDB** имеет встроенный механизм “версионирования схемы”, который отсутствует в серверных базах данных.

Если локальная версия базы данных меньше, чем версия, определённая в **open**, то сработает специальное событие **upgradeneeded**, и мы сможем сравнить версии и обновить структуры данных по мере необходимости.

Допустим у нас две в вкладки с разными версиями, что тогда? При попытке обновления на объекте базы возникает событие **versionchange**. Нам нужно слушать его и закрыть соединение к базе (а также, возможно, предложить пользователю перезагрузить страницу, чтобы получить обновлённый код). Если мы его не закроем, то второе, новое соединение будет заблокировано с событием **blocked** вместо **success**.

Чтобы сохранить что-то в **IndexedDB**, нам нужно хранилище объектов.

Хранилище объектов – это основная концепция **IndexedDB**. В других базах данных это «таблицы» или «коллекции». Здесь хранятся данные. В базе данных может быть множество хранилищ: одно для пользователей, другое для товаров и так далее. Мы можем хранить почти любое значение, в том числе сложные объекты.

Важно!Каждому значению в хранилище должен соответствовать уникальный ключ.Ключ должен быть одним из следующих типов: **number**, **date**, **string**, **binary** или **array**. Это уникальный идентификатор: по ключу мы можем искать/удалять/обновлять значения.

Синтаксис для создания хранилища объектов:

**db.createObjectStore(name[, keyOptions]);**

* **name** – это название хранилища, например "books" для книг,
* **keyOptions** – это необязательный объект с одним или двумя свойствами:
* **keyPath** – путь к свойству объекта, которое **IndexedDB** будет использовать в качестве ключа, например **id**.
* **autoIncrement** – если **true**, то ключ будет формироваться автоматически для новых объектов, как постоянно увеличивающееся число.

Важно! Хранилище объектов можно создавать/изменять только при обновлении версии базы данных в обработчике **upgradeneeded**.

Все операции с данными в **IndexedDB** могут быть сделаны только внутри **транзакций**.

Для начала транзакции:

**db.transaction(store[, type]);**

* **store** – это название хранилища, к которому транзакция получит доступ, например, "books". Может быть массивом названий, если нам нужно предоставить доступ к нескольким хранилищам.
* **type** – тип транзакции, один из:
  + **readonly** – только чтение, по умолчанию.
  + **readwrite** – только чтение и запись данных, создание/удаление самих хранилищ объектов недоступно.

Несколько **readonly** транзакций **могут одновременно** работать с одним и тем же хранилищем объектов, а **readwrite** транзакций – **не могут**. Транзакции типа **readwrite** “блокируют” хранилище для записи. Следующая такая транзакция должна дождаться выполнения предыдущей, перед тем как получит доступ к тому же самому хранилищу.

В первую очередь чтобы работать с хранилищем его нужно получить в рамках транзакции с помощью:

**transaction.objectStore(name)**

Хранилища объектов поддерживают два метода для добавления значений:

* **put(value, [key])** - добавляет значение **value** в хранилище. Ключ **key** необходимо указать, если при создании хранилища объектов не было указано свойство **keyPath** или **autoIncrement**. Если уже есть значение с таким же ключом, то оно будет заменено.
* **add(value, [key]) -** То же, что **put**, но если уже существует значение с таким ключом, то запрос не выполнится, будет сгенерирована ошибка с названием "**ConstraintError**".

Запросы на запись могут выполниться неудачно. При ошибке в запросе соответствующая транзакция отменяется полностью, включая изменения, сделанные другими её запросами.

Все методы поиска принимают аргумент query, который может быть либо точным ключом, либо диапазоном ключей:

* **store.get(query)** – поиск первого значения по ключу или по диапазону.
* **store.getAll([query], [count])** – поиск всех значений, можно ограничить, передав count.
* **store.getKey(query)** – поиск первого ключа, который удовлетворяет запросу, обычно передаётся диапазон.
* **store.getAllKeys([query], [count])** – поиск всех ключей, которые удовлетворяют запросу, обычно передаётся диапазон, возможно ограничить поиск, передав count.
* **store.count([query])** – получить общее количество ключей, которые удовлетворяют запросу, обычно передаётся диапазон.

Хранилище объектов внутренне сортирует значения по ключам. Поэтому запросы, которые возвращают много значений, всегда возвращают их в порядке сортировки по ключу.

Для поиска по другим полям объекта нам нужно использовать дополнительную структуру данных, называемую “индекс” (index). Индекс – это “надстройка“ к хранилищу, отслеживающая заданное поле объекта.

Синтаксис:

**objectStore.createIndex(name, keyPath, [options]);**

* **name** – название индекса,
* **keyPath** – путь к полю объекта, которое индекс должен отслеживать (мы собираемся сделать поиск по этому полю),
* **option** – необязательный объект со свойствами:
  + **unique** – если **true**, тогда в хранилище может быть только один объект с заданным значением в keyPath. Если мы попытаемся добавить дубликат, то индекс сгенерирует ошибку.
  + **multiEntry** – используется только, если **keyPath** является массивом. В этом случае, по умолчанию, индекс обрабатывает весь массив как ключ. Но если мы укажем **true** в **multiEntry**, тогда индекс будет хранить список объектов хранилища для каждого значения в этом массиве. Таким образом, элементы массива становятся ключами индекса.

Такие методы как **getAll/getAllKeys** возвращают массив ключей/значений. Но хранилище объектов может быть огромным, больше, чем доступно памяти. Тогда метод **getAll** вернёт ошибку при попытке получить все записи в массиве.

Объект **cursor** идёт по хранилищу объектов с заданным запросом (**query**) и возвращает пары ключ/значение по очереди, а не все сразу. Это позволяет экономить память.

Синтаксис:

**let request = store.openCursor(query, [direction]);** - как **getAll**, но с использованием курсора

* **query -** ключ или диапазон ключей, как для **getAll**.
* **direction -** необязательный аргумент, доступные значения:
  + "**next**" – по умолчанию, курсор будет проходить от самого маленького ключа к большему.
  + "**prev**" – обратный порядок: от самого большого ключа к меньшему.
  + "**nextunique**", "**prevunique**" – то же самое, но курсор пропускает записи с тем же ключом, что уже был (только для курсоров по индексам, например, для нескольких книг с price=5, будет возвращена только первая).

Чтобы получить ключи, а не значения (как **getAllKeys**): **store.openKeyCursor**

Так как хранилище объектов внутренне отсортировано по ключу, курсор проходит по хранилищу в порядке хранения ключей (по возрастанию по умолчанию).

Важно! Основным отличием курсора является то, что **request.onsuccess** генерируется многократно: один раз для каждого результата.

Основные методы курсора:

* **advance(count)** – продвинуть курсор на count позиций, пропустив значения.
* **continue([key])** – продвинуть курсор к следующему значению в диапазоне соответствия (или до позиции сразу после ключа **key**, если указан).

### Различия:

Отличия между **cookies** и **LocalStorage**:

* Размер (**cookies** ограничены 4 Кбайт, а размер **LocalStorage** - 2-10 Мбайт);
* Куки в отличие от данных локального хранилища включаются в состав запроса при отправке его на сервер;
* Печеньки можно прочитать и установить как на сервере, так и на клиенте; на клиенте доступны все куки, кроме тех, у которых установлен флаг **HttpOnly**; **LocalStorage** доступны только в браузере посредством **JavaScript API**;
* Куки хранятся ограниченное время (до конца сеанса или истечения указанной даты), нахождение данных в локальном хранилище не ограничено по времени;
* Работа с **LocalStorage** в **JavaScript** организовано намного удобнее чем с **cookies**;
* Для использования **cookies** может потребоваться разрешение.
* **Сookies** в основном используются для управления сеансом, персонализации и отслеживания действий пользователя, в то время как **LocalStorage** применяется в качестве обычного локального хранилища информации на компьютере пользователя.

**IndexedDB** же можно рассматривать как “**localStorage** на максималках”. Это простая база данных типа ключ-значение, достаточно мощная для оффлайн приложений, но простая в использовании. Она асинхронная, может хранить не только строки и т.д.