

Курс «Веб-технологии: уязвимости и безопасность»

JavaScript

Основы JavaScript, работа с DOM, XHR-запросы, формат JSON и сериализация.

Оглавление

[Введение](#)

[Видеоурок 1](#)

[Язык JavaScript](#)

[Пример работы с JavaScript](#)

[Функции в JavaScript](#)

[Переменные в JavaScript](#)

[Аргументы функции](#)

[Итоги](#)

[Видеоурок 2](#)

[Практика на JavaScript](#)

[Пример функции](#)

[Цикл в JavaScript](#)

[Итоги](#)

[Видеоурок 3](#)

[Функции для работы с DOM](#)

[Практика](#)

[Эскейпинг строк в JavaScript](#)

[Итоги](#)

[Видеоурок 4](#)

[XHR-запросы \(XMLHttpRequest\)](#)

[Итоги](#)

[Видеоурок 5](#)

[Формат данных JSON](#)

[Практика работы с JSON](#)

[Запись в JSON](#)

[Сериализация и десериализация](#)

[Итоги](#)

[Итоги урока](#)

[Ссылки к уроку](#)

Введение

В этом уроке мы обзорно рассмотрим язык программирования JavaScript.

На предыдущих уроках мы успешно освоили основы веба — URL, HTTP, HTML и CSS. Теперь научимся программировать на основном языке для веба — JavaScript, — и поймем, какие проблемы безопасности с ним связаны. План урока:

- 1) Что такое JavaScript.
- 2) Основы программирования на JavaScript.
- 3) Работа с DOM и эскейпингом строк в JavaScript.
- 4) Технология веб-запросов XHR.
- 5) Формат записи веб-объекта JSON и сериализация данных.

К концу урока вы напишете свою первую программу на JavaScript, поймёте, как работает DOM и как с помощью JavaScript с ним взаимодействовать, узнаете, что такое сериализация и JSON, научитесь делать XHR-запросы.

Видеоурок 1

Язык JavaScript

С помощью CSS сайт можно сделать красивым, но он будет интерактивным: будут только формы, ссылки, текст и, может быть несколько картинок.

Интерактивным сайт можно сделать при помощи языка для динамического отображения веб-контента — JavaScript. Он делает так, что в браузерах можно исполнять какие-то более сложные действия, чем отображение картинок, текста и отправки форм.

Пример работы с JavaScript

Откроем браузер Firefox. Мы видим простую HTML-страницу с заголовком зеленого цвета, текст и кнопку Display. Если мы нажмём на кнопку Display, цвет заголовка изменится с зеленого на красный и обратно.

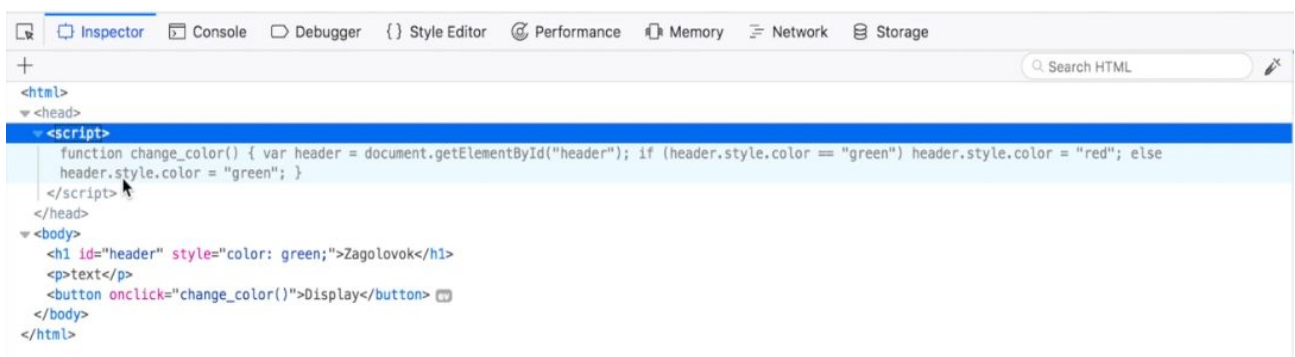
Это возможно благодаря JavaScript. Давайте рассмотрим исходный код этой страницы:

```
<html>
  <head></head>
  <body>
    <h1 id="header" style="color: green;">Zagolovok</h1>
    <p>text</p>
    <button onclick="change_color()">Display</button>
  </body>
</html>
```

Мы видим, что у кнопки есть атрибут `onclick="change_color()"`, который вызывает функцию `change_color()` при нажатии Display.

Также существуют другие атрибуты, например `onload=""` — он срабатывает при загрузке элемента, действует функция, которая записана аргументом в параметре `onload=""`. Что такое функция, мы узнаем чуть позже.

Также здесь есть скрипт, который и меняет цвет на странице:



```
function change_color() { var header = document.getElementById("header"); if (header.style.color == "green") header.style.color = "red"; else header.style.color = "green"; }
</script>
<body>
  <h1 id="header" style="color: green;">Zagolovok</h1>
  <p>text</p>
  <button onclick="change_color()">Display</button>
</body>
</html>
```

Откроем HTML-файл в текстовом редакторе, чтобы увидеть все более наглядно:

```
nano js-1.html

<script>
  function change_color() {
    var header = document.getElementById("header");

    if (header.style.color == "green")
      header.style.color = "red"
    else
      header.style.color = "green";
  }
</script>

<h1 id="header" style="color:green">Zagolovok</h1>
```

```
<p>text</p>
<button onclick="change_color()">Display</button>
```

В этом html мы используем теги `<script></script>` — именно в них пишется JavaScript-код. В данном случае написана функция `change_color()`.

Функции в JavaScript

Функция — преобразование, которое принимает данные на вход и на выходе производит с ними операции. Иногда функция ничего не принимает на вход, а просто оперирует данными, как здесь.

Чтобы объявить функцию JavaScript, мы должны написать слово `function`, имя функции и скобочки `()`. В скобки можно записать аргументы функции, то есть то, что подается на входе. В следующих уроках мы увидим, как это работает, но сейчас для простоты оставим функцию без аргументов.

И, наконец, описание тела функции заключается в фигурные скобки `{}`, так же, как в случае с CSS, только там мы вписывали содержимое стиля, а здесь — JavaScript-код, который выполнится.

Чтобы вызвать функцию в коде, необходимо написать ее имя и добавить скобки, например, `getElementById()` — функция и мы видим, что она принимает аргумент — строку. Функция `getElementById()` ищет на странице HTML-тег, у которого атрибут `id` равен той строке, которую мы зададим, например, тег `<h1>` в примере имеет `id="header"`.

Особенность `id` в том, что он должен быть уникален по всей странице, то есть в пределах одной страницы не может быть два одинаковых `id="header"`. Именно поэтому мы однозначно идентифицируем элемент `<h1 id="header">Zagolovok</h1>` и получаем его в переменную `header`. Здесь могло бы быть другое название: то, что совпали и значение атрибута `id` и название самой переменной — ничего не значит, на самом деле здесь можно написать все, что разрешено в качестве названия переменной в JavaScript.

Дальше в примере кода на JavaScript появляется условие `if`. Условие говорит о том, что если в скобках истина (текущий цвет элемента равен `green`), цвет элемента меняется на красный, в противном случае, если это условие не выполняется, `header` становится зеленым.

Переменные в JavaScript

Переменная — контейнер, в котором мы храним значение. Мы уже встречались с переменными, когда использовали GET и POST-запросы. HTTP-заголовки, или Cookie, по сути, тоже переменные, просто у них есть свои специальные названия и они используются в конкретных протоколах. В переменную мы записываем значение и в дальнейшем можем обратиться к ней, чтобы узнать ее значение в текущий момент времени.

Например, если мы хотим посмотреть цвет заголовка, это можно сделать через запрос значения `header.style.color`. А если мы захотим изменить цвет заголовка, можно положить туда вместо зеленого нужный нам цвет.

Аргументы функции

Посмотрим на пример функции с аргументом. Допустим нас есть функция `plus()`, которая складывает все ее аргументы:

```
function plus(a, b, c) {  
    console.log(a + b + c)  
}
```

То есть на вход мы, например, дадим 1, 2 и 3, на выходе получим 6, потому что $1+2+3=6$. Ее вывод мы можем посмотреть, например, с помощью функции `console.log()` — так мы выведем результат в консоль. Давайте вызовем функцию `plus()` из функций `change_color()`:

nano js-1.html

```
<script>  
    function change_color() {  
        var header = document.getElementById("header");  
  
        if (header.style.color == "green")  
            header.style.color = "red"  
        else  
            header.style.color = "green";  
  
        plus(2, 3, 4);  
    }  
  
    function plus(a, b, c) {  
        console.log(a + b + c)  
    }  
</script>  
  
<h1 id="header" style="color:green">Zagolovok</h1>  
<p>text</p>  
<button onclick="change_color()">Display</button>
```

Откроем страницу и откроем консоль разработчика, обновим страницу, чтобы подгрузить новый JavaScript. Теперь нажмем кнопку Display — JavaScript сложил все числа, которые мы написали, то есть 2, 3 и 4, и результат — 9, как и должно быть. Функция в конце вывела это в консоль:



Также JavaScript умеет выводить более явным методом, например функцией `alert()`:

```
nano js-1.html

<script>
  function change_color() {
    var header = document.getElementById("header");

    if (header.style.color == "green")
      header.style.color = "red"
    else
      header.style.color = "green";

    plus(2, 3, 4);
  }

  function plus(a, b, c) {
    alert(a + b + c)
  }
</script>

<h1 id="header" style="color:green">Zagolovok</h1>
<p>text</p>
<button onclick="change_color()">Display</button>
```

Обновим страницу, нажмем Display — JavaScript умеет выводить текст на страницу в специальном диалоговом окне:



Подробнее, как это работает, мы узнаем в следующих уроках и еще в следующих уроках мы научимся записывать результат работы функции в саму страницу, то есть изменять ее DOM-дерево.

Итоги

Подведем итоги:

- 1) Познакомились с JavaScript.
- 2) Узнали, что такое функции, посмотрели на пример функции. Разобрали аргументы и переменные функции и научились работать с ними в JavaScript.

Видеоурок 2

В этом видеоуроке мы поговорим про основы программирования на JavaScript. План видео:

- 1) Что такое функция на JavaScript, зачем она нужна и как её можно написать.
- 2) Циклы и частный пример цикла — оператор for.
- 3) Базовый синтаксис JavaScript.

Практика на JavaScript

Сразу перейдём к примеру: откроем Ubuntu и создадим файл этого урока:

```
nano 1-5-2.html
```

```
<body>
<script>
  var pElem = document.createElement("p");
</script>
</body>
```

Как вы знаете, тег `<body>` можно не писать, но чтобы создать JavaScript-функции, нам нужен валидный HTML-документ. Внутри напомним вызов объекта `<document>` — это будет ссылка на текущий HTML-документ в браузере. Это корень всего, в нем функции для работы с DOM (подробнее DOM мы разберем в следующем видеоуроке, когда будем разбирать DOM-дерево). Далее мы вызовем функцию `createElement()` — эта функция создаст элемент и вернет нам его объект, а мы положим его в переменную `var pElem`.

Ключевое слово `var` означает, что `pElem` — нужная нам переменная (`var` — от английского слова *variable*). `pElem` — имя, а справа находится значение, так же, как и с GET-параметрами. Чтобы достать значение, которое в нем лежит, мы можем обратиться к переменной по имени. Сейчас в переменной находится элемент `<p>` (когда вы лучше разберетесь с JavaScript и другими языками программирования, то поймете, что на самом деле там лежит).

Дальше нам необходимо создать текст:

```
<body>
<script>
```

```

    var pElem = document.createElement("p");
    var textNode = document.createTextNode("Hi, mom!");
  </script>
</body>

```

Эта функция создаёт так называемый `TextNode` — он так называется, потому что это узел или, по-другому, вершина (от английского `node` — узел). Представьте, что HTML — дерево, вспомните, что у дерева есть вершины, которые также называются узлами: по сути это разные элементы HTML-документа, например, текстовое поле. То есть, вызывая функцию `createTextNode()` с аргументом, мы говорим о том, что нам требуется написать в этом текстовом поле.

Теперь нам необходимо к элементу `pElem` прикрепить `textNode`, и мы сделаем это с помощью функции `appendChild()`. В прямом переводе на русский это означает прикрепить потомка или ребенка. Так и есть: к элементу дерева `<p>` мы прикрепляем потомка, то есть узел, который находится ниже по дереву:

```

<body>
<script>
  var pElem = document.createElement("p");
  var textNode = document.createTextNode("Hi, mom!");
  pElem.appendChild(textNode);
</script>
</body>

```

Теперь нам необходимо прикрепить на страницу сам элемент `<p>`:

```

<body>
<script>
  var pElem = document.createElement("p");
  var textNode = document.createTextNode("Hi, mom!");
  pElem.appendChild(textNode);
  document.body.appendChild(pElem);
</script>
</body>

```

Напишем `<document>` и обратимся к `<body>`, так как это тоже HTML-элемент. Затем ставим `appendChild()`, но в этот раз нам необходимо прикрепить к `<body>` текст, который завернут в параграф `<p>`, — это переменная `pElem`.

Сохраняем и переходим в браузер — посмотрим, что получилось:



Все успешно прикрепилося и надпись добавилась на экран.

По большому счету JavaScript и нужен для того, чтобы добавлять надписи, скрывать их, менять атрибуты HTML-тегов, то есть взаимодействовать с DOM-деревом, с самим HTML. Просто HTML в итоге представляется в браузере как DOM-дерево и с ним идет работа.

Пример функции

Представим, что нам нужно добавить не одну надпись, а несколько и в разных местах кода. Для этого скопируем четыре строчки, вставим их и напишем здесь другой текст:

```
<body>
<script>
  var pElem = document.createElement("p");
  var textNode = document.createTextNode("Hi, mom!");
  pElem.appendChild(textNode);
  document.body.appendChild(pElem);

  var pElem = document.createElement("p");
  var textNode = document.createTextNode("I'm on TV!");
  pElem.appendChild(textNode);
  document.body.appendChild(pElem);
</script>
</body>
```

Перезагрузим страницу:



Если мы хотим добавить еще текст, то, соответственно, еще раз напишем это:

```
<body>
<script>
  var pElem = document.createElement("p");
  var textNode = document.createTextNode("Hi, mom!");
  pElem.appendChild(textNode);
  document.body.appendChild(pElem);

  var pElem = document.createElement("p");
  var textNode = document.createTextNode("I'm on TV!");
  pElem.appendChild(textNode);
  document.body.appendChild(pElem);

  var pElem = document.createElement("p");
  var textNode = document.createTextNode("Amazing!");
  pElem.appendChild(textNode);
  document.body.appendChild(pElem);
</script>
</body>
```

```
pElem.appendChild(textNode);
document.body.appendChild(pElem);
</script>
</body>
```

И еще раз перезагрузим страницу:



Но, согласитесь, каждый раз один и тот же код очень неудобно копировать, вставлять и изменять. Более того, если вы вдруг захотите сделать что-то другое, например, добавить надпись с другим цветом, то вам придется добавлять еще три строчки. А если их будет 1000, 2000?

Конечно, никому не хочется копировать и вставлять что-либо вручную, и для этого люди придумали функции. Мы познакомились с ними на предыдущем видео, сейчас еще раз повторим и закрепим материал.

Функция — это участок кода, который мы можем удобно вызывать несколько раз. Рассмотрим ее на примере и создадим функцию `appendText()`:

```
<body>
<script>
function appendText(text) {
    var pElem = document.createElement("p");
    var textNode = document.createTextNode(text);
    pElem.appendChild(textNode);
    document.body.appendChild(pElem);
}

appendText("Hi, mom! (2)");
</script>
</body>
```

Ключевое слово `function` — то же самое что и `var`, только `function` обозначает, что дальше будет идти имя функции, в данном случае — `appendText()`. По этому имени мы обратимся к функции, чтобы она выполнила те действия, которые мы в ней напишем. Сами действия будут писаться между двух фигурных скобок `{}`. `text` — это так называемый параметр функции, кто-то называет его аргументом. Его нужно будет передать, когда мы будем вызывать функцию. Рассмотрим вызов функции.

Каждый раз, когда мы пишем `appendText()`, открываем скобки и передаём туда параметр (или аргумент функции) — `Hi, mom! (2)`, мы вызываем эту функцию.

Бывает, что у функции несколько параметров, но это не наш случай. Мы не можем написать сюда еще 123, 456, потому что эта функция принимает только один параметр, и нам нужно передать только его. Сохраним наши изменения и посмотрим, как это отобразится:



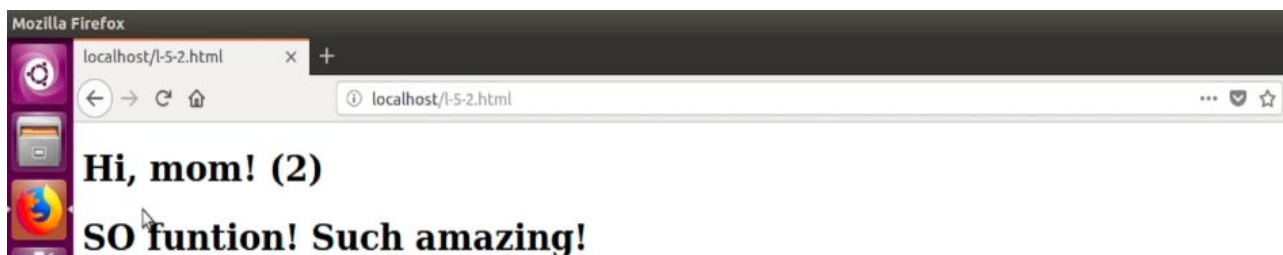
Текст успешно добавлен.

Вы можете сказать, что результат тот же самый, но это не совсем так. Теперь мы можем вызывать функцию `appendText()` несколько раз и нам не нужно копировать каждый раз четыре строчки. Это только в нашем примере их 4, а в реальных проектах сотни тысячи строк кода. Так нам не нужно их заново копировать и вставлять, можно просто написать имя функции, передать ей параметр и она сделает то, что нужно. А если мы захотим в ней что-то поменять, например, сделать так, чтобы текст стал заголовком `<h1>`, мы легко сделаем это, поправив код в одном месте:

```
<body>
<script>
  function appendText(text) {
    var pElem = document.createElement("h1");
    var textNode = document.createTextNode(text);
    pElem.appendChild(textNode);
    document.body.appendChild(pElem);
  }

  appendText("Hi, mom! (2)");
  appendText("SO funtion! Such amazing!");
</script>
</body>
```

Теперь все вызовы функции будут добавлять заголовки с тегом `<h1>`:



Если бы у нас было 50 надписей и мы не пользовались функцией, нам бы пришлось в каждой из 50 надписей заменить `<p>` на `<h1>`, а так мы меняли это всего в одном месте, так что везде, где мы вызываем функцию, произойдут изменения.

Цикл в JavaScript

С функциями мы разобрались, перейдём к циклам.

Допустим, что мы хотим сделать много надписей на странице. Конечно, можно вызвать много раз функцию `appendText()`, но придется копировать и вставлять один и тот же фрагмент кода несколько раз, а, как мы уже видели, это неудобно. Воспользуемся циклом:

```
<body>
<script>
  function appendText(text) {
    var pElem = document.createElement("h1");
    var textNode = document.createTextNode(text);
    pElem.appendChild(textNode);
    document.body.appendChild(pElem);
  }
  var i = 0;
  for (i = 1; i <= 5; i++) {
    appendText(i);
  }
</script>
</body>
```

Посмотрим на результат:



На экране появились цифры от 1 до 5 — это говорит о том, что `appendText()` выполнялась, как мы и хотели, 5 раз.

Теперь разберём подробнее, что же произошло. Сначала мы создали переменную (`i`), равную нулю — это наш счетчик. В цикле мы говорим, что счетчик сначала равен единице и мы выполним его, пока он будет меньше либо равен 5. Последний операнд (`i++`) — инструкция увеличивать текущее значение счетчика на единицу — она будет выполняться каждый раз при каждой итерации цикла.

Итерация — это однократное выполнение тела цикла. Тело цикла находится между фигурными скобками {}.

Разберем действия, происходящие в цикле, по шагам. Первое выражение (`i = 1;`) до первой точки с запятой выполнится лишь один раз, когда мы дойдем до оператора цикла (`for`). Второе условие (`i <= 5;`) будет проверяться в конце каждой итерации — мы выполнили тело цикла, а затем проверили условие на истину. И если условие истинно, мы продолжаем и переходим на следующую итерацию. Выполняем еще раз тело цикла, и если условие ложно, то есть (`i`) стало больше пяти, выходим из цикла. Третье выражение (`i++`) тоже выполняется в конце каждой итерации, но при этом оно ничего не проверяет, а просто выполняется — в данном случае мы увеличиваем счетчик (`i`) на единицу.

Также есть циклы `while`, `do while` — по сути они делают то же, что и `for`, но в некоторых ситуациях их удобнее использовать; вы можете самостоятельно разобраться и изучить их.

Итоги

Давайте подведем итоги. Вы узнали:

- 1) Как написать функции на JavaScript, зачем это может понадобиться.
- 2) Как писать циклы, зачем нужен `for` и как работают циклы.
- 3) Разобрали примеры базового синтаксиса JavaScript: ключевые слова `function`, `var`, параметры и аргументы, цикл `for`.

Видеоурок 3

На этом уроке мы разберем работу с DOM и эскейпингом строк в JavaScript:

- 1) Что такое DOM (Document Object Model), зачем он нужен и как им пользоваться.
- 2) Базовые функции для работы с DOM в JavaScript. Как JavaScript взаимодействует с DOM.
- 3) Что такое эскейпинг строк в JavaScript и когда его необходимо применять.

Функции для работы с DOM

JavaScript был создан, чтобы изменять HTML-элементы, добавлять новые, удалять старые, скрывать, прятать и показывать их.

Мы уже видели, как HTML представляется в виде дерева: проще говоря, это и есть DOM-дерево, с которым JavaScript в итоге взаимодействует. Посмотрим несколько основных примеров функций для работы с DOM в JavaScript.

Как мы помним, есть специальная переменная `document` — это корень DOM-дерева и основа всей DOM-иерархии, через которую можно вызвать основные элементы — `<head>` и `<body>`. Также через

нее вызываются методы для работы с DOM-деревом, ищутся, добавляются и удаляются узлы — через `document` делается практически все.

Функции для работы с DOM

`document` -- корень всего DOM дерева

`document.getElementById(id)` и др. -- получение доступа к поддеревьям DOM

`element.innerHTML` -- изменить содержимое элемента

`document.createElement(element)` и др. -- создать новый элемент

Есть целый класс функций, которые ищут элементы, находят их в DOM-дереве, чтобы потом мы могли их сохранить в переменную и затем пользоваться этим (добавлять к элементам что-то, менять их стили и так далее). Например, есть функция `getElementById(id)` и, как вы помните, `id` — это специальный HTML-атрибут, который на странице должен быть уникален. Функция `getElementById()` — хороший способ получить элемент с HTML-страницы.

Другой объект — `element.innerHTML` — содержимое элемента. Если мы выберем элемент абзаца `<p>`, его `innerHTML` будет находиться внутри этого абзаца. Другой пример: `innerHTML` элемента `<body>` — все, что мы запишем в `<body>`, то есть все содержимое страницы. Собственно говоря, `innerHTML` — это так называемое поддерево текущего элемента, все, что в него включено.

Также есть целый класс функций для создания элементов, например `createElement()`. Эта функция позволяет создать любой новый HTML-элемент, например, элемент абзаца `<p>` или картинки ``, и любые другие элементы.

Практика

Откроем Ubuntu, скопируем файл из предыдущего урока и на его основе познакомимся ближе с тем, как работают функции взаимодействия с DOM-деревом:

```
cd /var/www/html && cp 1-5-2.html 1-5-3.html && nano 1-5-3.html
```

```
<body>
<script>
  function appendText(text) {
    var pElem = document.createElement("h1");
    var textNode = document.createTextNode(text);
    pElem.appendChild(textNode);
    document.body.appendChild(pElem);
  }

  var i = 0;
```

```
    for (i = 1; i <= 5; i++) {  
        appendText(i);  
    }  
</script>  
</body>
```

Теперь работаем с тем, что есть: у нас 5 элементов `<h1>` с текстом, возьмем любой, например второй, и изменим то, что у него внутри. Для этого допишем функцию `getElementsByName()`:

```
<body>  
<script>  
    function appendText(text) {  
        var pElem = document.createElement("h1");  
        var textNode = document.createTextNode(text);  
        pElem.appendChild(textNode);  
        document.body.appendChild(pElem);  
    }  
    var i = 0;  
    for (i = 1; i <= 5; i++) {  
        appendText(i);  
    }  
  
    var secondH1 = document.getElementsByTagName("h1");  
</script>  
</body>
```

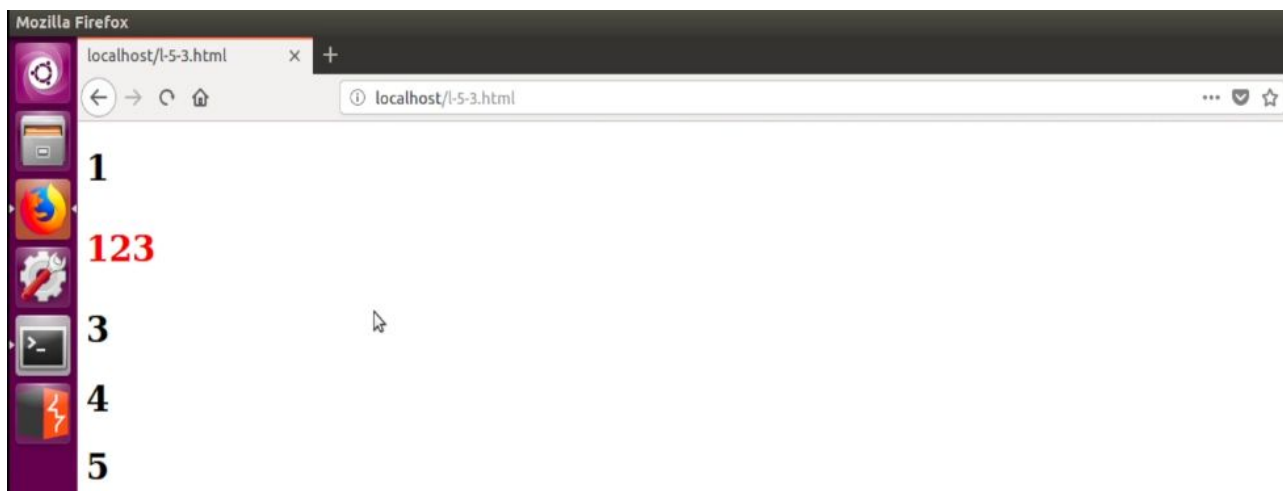
Мы выбрали все элементы с тегом `<h1>` и это очень удобно — мы вызываем одну функцию и она нам собирает все элементы. Теперь в переменной `secondH1` хранится массив, содержащий все элементы с тегом `h1`. Условно массив — это несколько переменных, сгруппированных в одну так, чтобы к ним удобно и быстро можно было обращаться.

Теперь получим доступ ко второму элементу массива:

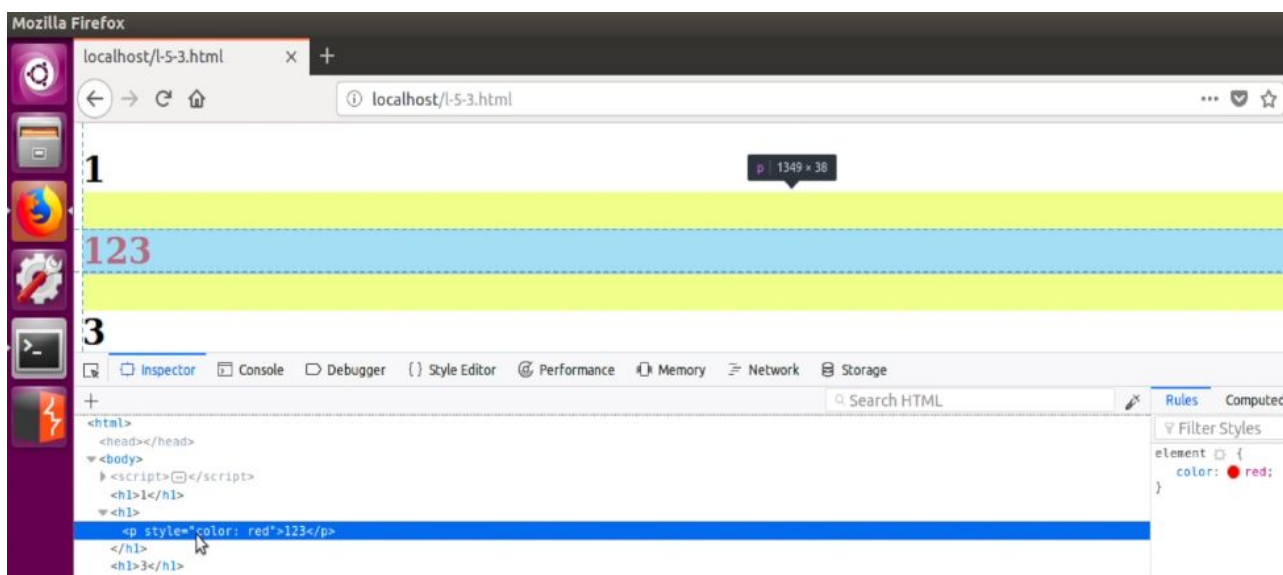
```
<body>  
<script>  
    function appendText(text) {  
        var pElem = document.createElement("h1");  
        var textNode = document.createTextNode(text);  
        pElem.appendChild(textNode);  
        document.body.appendChild(pElem);  
    }  
    var i = 0;  
    for (i = 1; i <= 5; i++) {  
        appendText(i);  
    }  
  
    var secondH1 = document.getElementsByTagName("h1");  
    secondH1[1].innerHTML = "<p style='color: red'>123</p>";  
</script>  
</body>
```

Вы можете подумать что мы ошиблись, сказав, что обращаемся ко второму, а написали в квадратных скобках [1]. Но на самом деле программисты выбирают нумерацию не с единицы, а с нуля, в том числе и для элементов массива. У первого элемента индекс [0], поэтому, если нужен второй элемент массива, пишем в квадратных скобках [1].

Проверим, как это будет выглядеть в браузере:



Второй элемент изменился и стал красным — 123. Если мы откроем это в браузере и посмотрим на код, то увидим второй элемент `<h1><p style="color: red">123</p></h1>` — всё, как мы написали, только браузер сам поменял одинарные кавычки на двойные:



Мы познакомились с тем, как можно искать элементы, как можно через `innerHTML` их менять. Теперь вы знаете что `<document>` — это корневой узел, через который мы имеем доступ к DOM-дереву созданию изменений в нем. Добавлять и вставлять элементы можно через `innerHTML`, но лучше всего использовать для этого существующие специальные методы JavaScript.

Эскейпинг строк в JavaScript

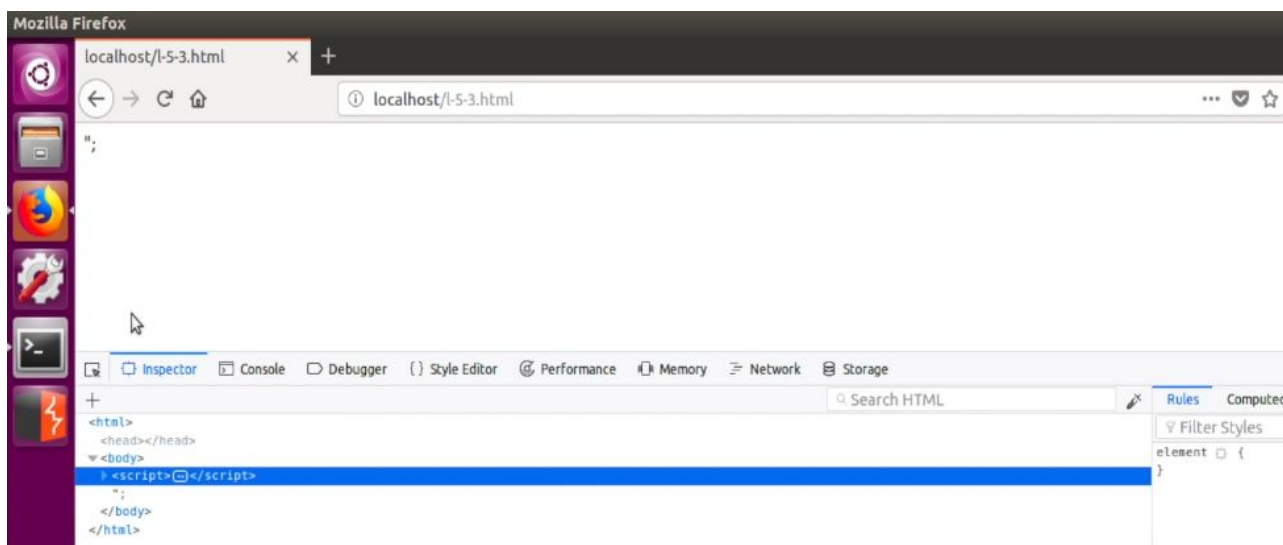
Если мы напишем JavaScript-код и вставим его через innerHTML, он не выполнится:

```
<body>
<script>
  function appendText(text) {
    var pElem = document.createElement("h1");
    var textNode = document.createTextNode(text);
    pElem.appendChild(textNode);
    document.body.appendChild(pElem);
  }

  var i = 0;
  for (i = 1; i <= 5; i++) {
    appendText(i);
  }

  var secondH1 = document.getElementsByTagName("h1");
  secondH1[1].innerHTML = "<p style='color:
red'>123</p><script>alert(1)</script>";
</script>
</body>
```

Проверим это:



Ничего не получилось, вывелось что-то совсем другое.

Это произошло потому, что закрывающий тег `</script>` в составе innerHTML-фрагмента кода парсер воспринял как закрытие всего JavaScript-кода. Чтобы избежать таких ошибок, существует JavaScript-эскейпинг — это обратный слеш (`\`). Добавим его в innerHTML:

```
<body>
<script>
  function appendText(text) {
    var pElem = document.createElement("h1");
```

```

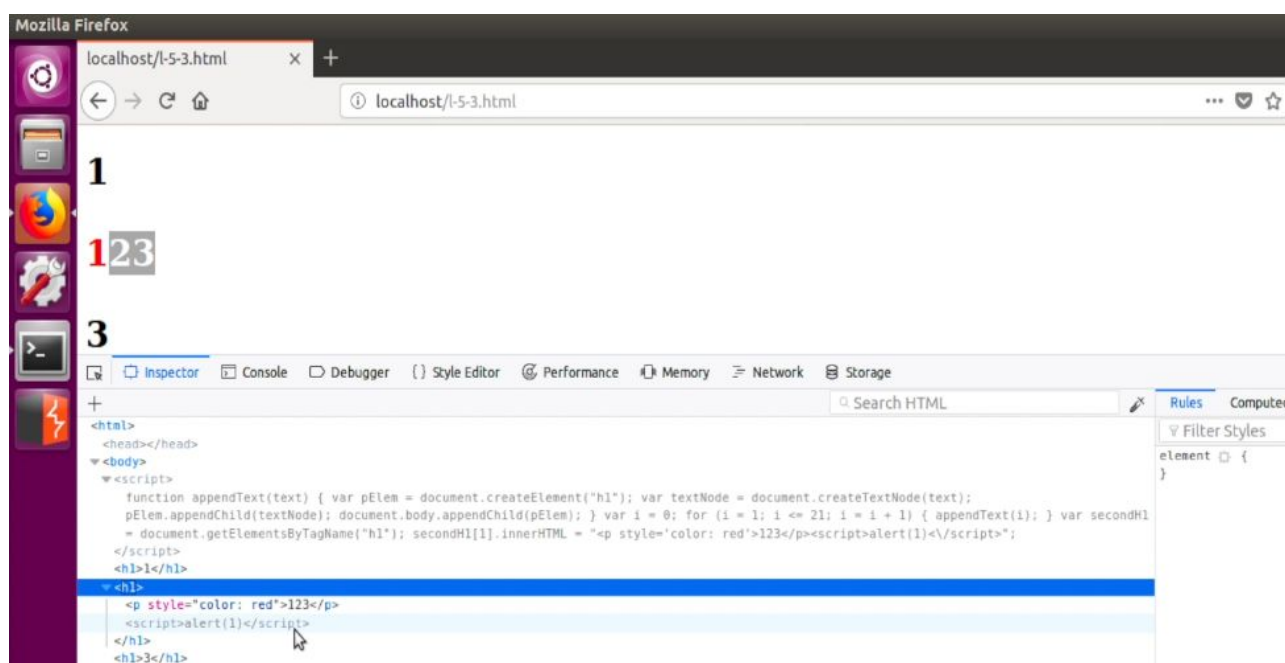
var textNode = document.createTextNode(text);
pElem.appendChild(textNode);
document.body.appendChild(pElem);
}

var i = 0;
for (i = 1; i <= 5; i++) {
    appendText(i);
}

var secondH1 = document.getElementsByTagName("h1");
secondH1[1].innerHTML = "<p style='color:
red'>123</p><script>alert(1)<\\script>";
</script>
</body>

```

Снова перезагрузим страницу:



HTML вернулся в прежнее состояние, второй элемент (123) снова стал красным, но при этом скрипт не исполнился. В коде есть тег `<script>`, но он не исполняется.

Теперь подробнее рассмотрим, как работает эскейпинг строк в JavaScript. Мы уже увидели, что, если добавить закрывающий тег `</script>` в строку JavaScript, парсер может воспринять это как HTML-тег. Чтобы этого избежать, мы добавляем обратный слеш (`\`) перед символом, который мы хотим заэскейпить. Вспомните про HTML-эскейпинг — там мы заменяли некоторые символы специальной последовательностью, которая преобразовывалась в символ, когда отображалась в браузере.

В JavaScript, как и в некоторых других языках программирования, для эскейпинга мы вставляем специальный символ перед тем, который мы хотим отобразить как символ, а не что-то специальное.

После этого символ не имеет специального значения, а эскейпится, или, по-другому, мы экранируем его.

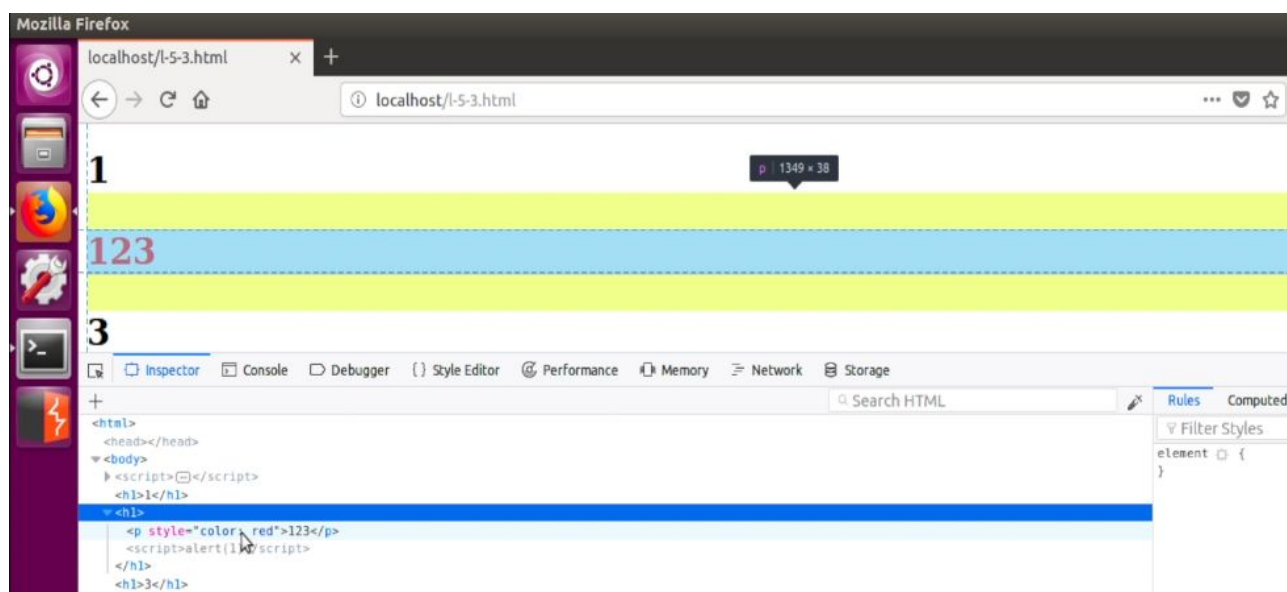
Например, мы можем переписать весь innerHTML с двойными кавычками. Но, чтобы они воспринимались не как закрытие всей строки, а как двойная кавычка, добавим обратный слеш перед внутренними двойными кавычками (\"):

```
<body>
<script>

..

var secondH1 = document.getElementsByTagName("h1");
secondH1[1].innerHTML = "<p style=\"color:
red\">123</p><script>alert(1)</script>";
</script>
</body>
```

Перезагрузим страницу и увидим, что все отобразилось так, как мы ожидали, ничего не сломалось:



Эскейпинг нужен, чтобы интерпретатор или компилятор воспринимали спецсимволы как обычные символы строки, а не как часть синтаксиса языка программирования. В целом, эскейпинг — это популярная техника, она применяется во всех языках программирования, а не только в HTML или JavaScript.

Итоги

Давайте подведем итоги, в этом видео вы узнали:

- 1) Что такое DOM, углубили его понимание.

- 2) Базовые функции JavaScript для работы с DOM. Теперь вы умеете искать элементы, выбирать их, изменять их значения, добавлять и удалять их.
- 3) Зачем нужен JavaScript-эскейпинг, как и где его необходимо применять.

Видеоурок 4

Поговорим про XHR request:

- 1) Что такое XHR, или XMLHttpRequest, зачем он нужен.
- 2) Как использовать XHR, какие методы есть в JavaScript, чтобы его использовать.
- 3) Какие ограничения существуют на XHR.

XHR-запросы (XMLHttpRequest)

XHR нужен, чтобы JavaScript мог делать веб-запросы — GET, POST и другие. Это очень полезно, когда мы хотим не перегружать страницу целиком, а получить данные сервера без перезагрузки страницы. Пользователь не увидит этого, JavaScript в фоне все сделает и доставит необходимые данные на страницу, возможно, как-то при этом ее изменит.

Откроем Ubuntu и создадим новый файл:

```
cd /var/www/html && nano 1-5-4.html
```

Чтобы использовать XHR, нужно создать переменную с объектом XHR:

```
<script>
  var xhr = new XMLHttpRequest();

</script>
```

Тут мы видим новую функцию, ключевые слова, название, которое на самом деле не совсем название функции. Слово `new` применяется, когда мы хотим создать новый экземпляр объекта или класса — это всё термины из объектно-ориентированного программирования. Чтобы создать объект XHR, которым потом мы будем пользоваться для запросов, нужно написать `new XMLHttpRequest()`.

XHR называется XML Http Request потому, что изначально предполагалось, что JavaScript будет делать запросы, в теле которых будет находиться XML. По сути, XML — это формат документа. Он отличается от формата HTML тем, что в XML больше возможностей в целом и он гораздо более строгий. Например, если вы в XML где-то пропустите хотя бы одну кавычку или не закроете тег, или забудете одну скобку в теге, или перепутаете теги местами, он обязательно на это отреагирует и не будет парситься. XML очень строгий в этом плане, в отличие от HTML. HTML-парсер может самостоятельно достраивать неполные теги и исправлять мелкие ошибки.

Сейчас веб-запросы в основном делаются не через XML, а через более новый стандарт — JSON. Запросим простой текст:

```
<script>
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "http://localhost/test.txt", false);
  xhr.send();
</script>
```

Нужно вызвать метод `xhr.open()`. Указываем метод запроса GET: он самый простой. Далее указываем ссылку, откуда нужно этот запрос получить — это может быть абсолютный или относительный URL. Сначала укажем абсолютный, но потом его можно легко поменять и на относительный. Далее пишем `false` — третий аргумент в `xhr.open()`; указывает, будет запрос синхронным или асинхронным. По умолчанию эти запросы асинхронные, поэтому мы должны явно указать `false`, то есть, что он синхронный. Синхронный означает, что выполнение JavaScript-кода остановится, пока запрос полностью не выполнится, то есть пока мы не получим ответ от сервера.

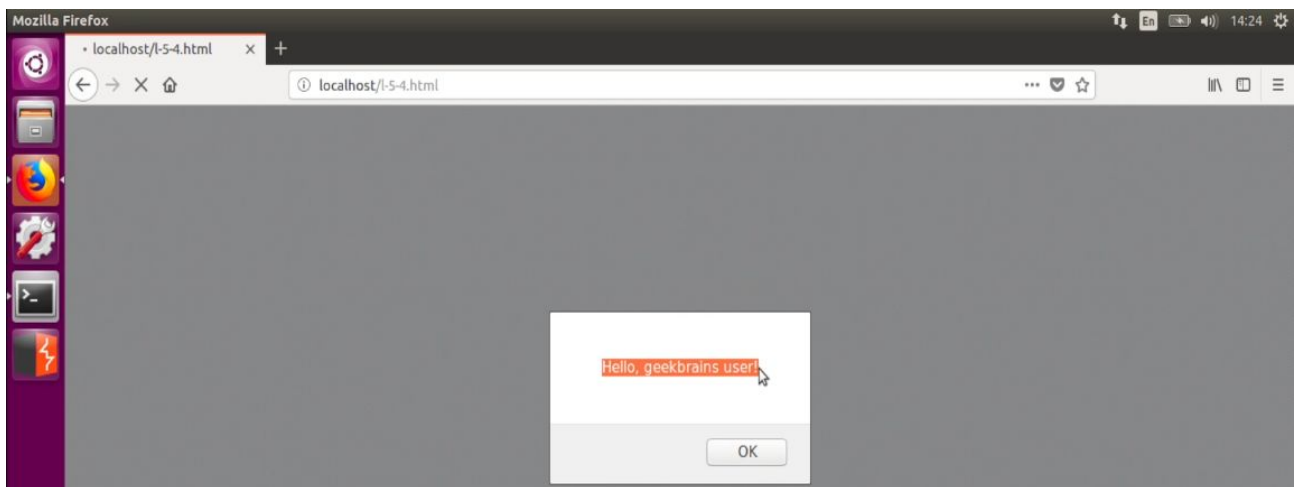
Если запрос асинхронный, это значит, что когда исполнение кода дойдет до строки с `xhr.send()`, JavaScript не будет ждать ответ на запрос, а просто дальше выполнит код. Когда придёт ответ на запрос, вызовется обработчик события и дальнейшие события происходят в зависимости от обработчика, который мы напишем. Подробнее про обработчики вы узнаете, когда будете глубже изучать JavaScript. В целом, обработчик — это функция, которая вызывается при определенном событии.

Еще XHR удобен тем, что в нём много полей. Поля — это свойства или методы нужного нам объекта, вызываются через точку (`.`). Например, они позволяют нам получить код ответа от сервера:

```
<script>
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "http://localhost/test.txt", false);
  xhr.send();

  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText);
  } else {
    alert(xhr.responseText);
  }
</script>
```

Выделенное желтым значит: если код ответа от сервера не равен 200, то есть если возникла какая-то ошибка, система выведет ее на экран и укажет ее код. Иначе выведет результат — то, что вернулось в GET-запросе. Мы написали девять строк кода, теперь проверим, что они работают. Зайдём в Firefox и откроем страницу:



Как только мы открываем страницу, сразу высвечивается `alert()` и текст с файлом, который мы получили. Файл тестовый, и мы видим надпись **Hello, geekbrains user!** — это содержимое выводится в алерте.

JavaScript-код можно вызывать, например, по нажатию кнопки. Давайте обернём код в функцию, которую назовем `xhrTest()`, добавим кнопку `<button>` и поместим новую функцию в ее событие `OnClick` — это атрибут `onclick=".."`:

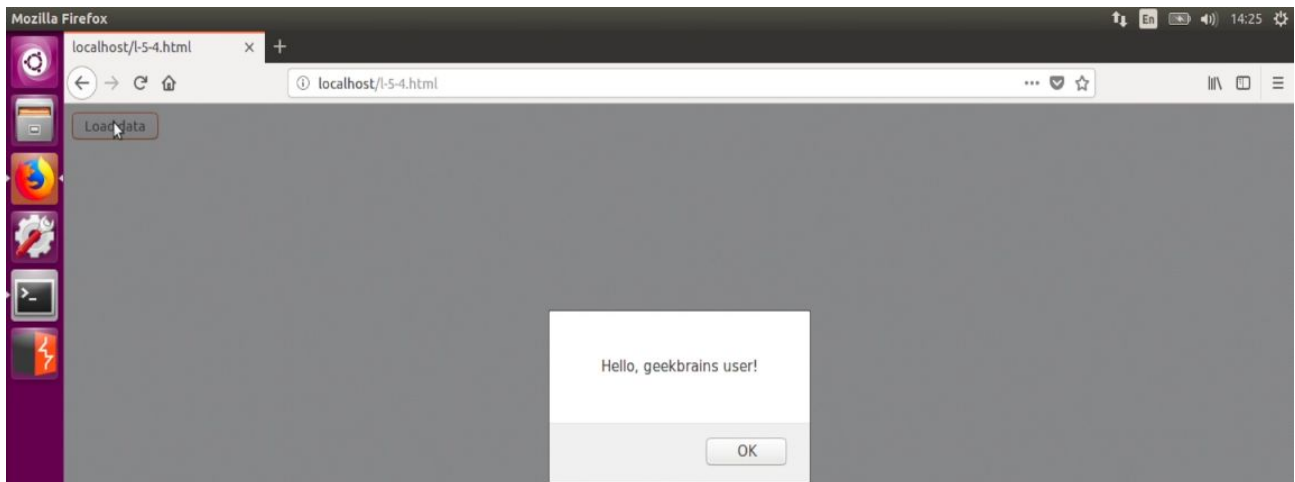
```
<script>
function xhrTest() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "http://localhost/test.txt", false);
    xhr.send();

    if (xhr.status != 200) {
        alert(xhr.status + ': ' + xhr.statusText);
    } else {
        alert(xhr.responseText);
    }
}
</script>
<button onclick="xhrTest()">Load data</button>
```

Когда мы нажмем на кнопку, выполнится то, что мы напишем в атрибуте `onclick`. Например, мы напишем `onclick="xhrTest()"` — то есть выполнять функцию `xhrTest()` — и назовем кнопку `Load data`. Закроем, сохраним и перезагрузим страницу:



Увидим кнопку Load data, но алерта не будет. И только после того, как мы нажмем на кнопку, появляется то, что мы загружаем:



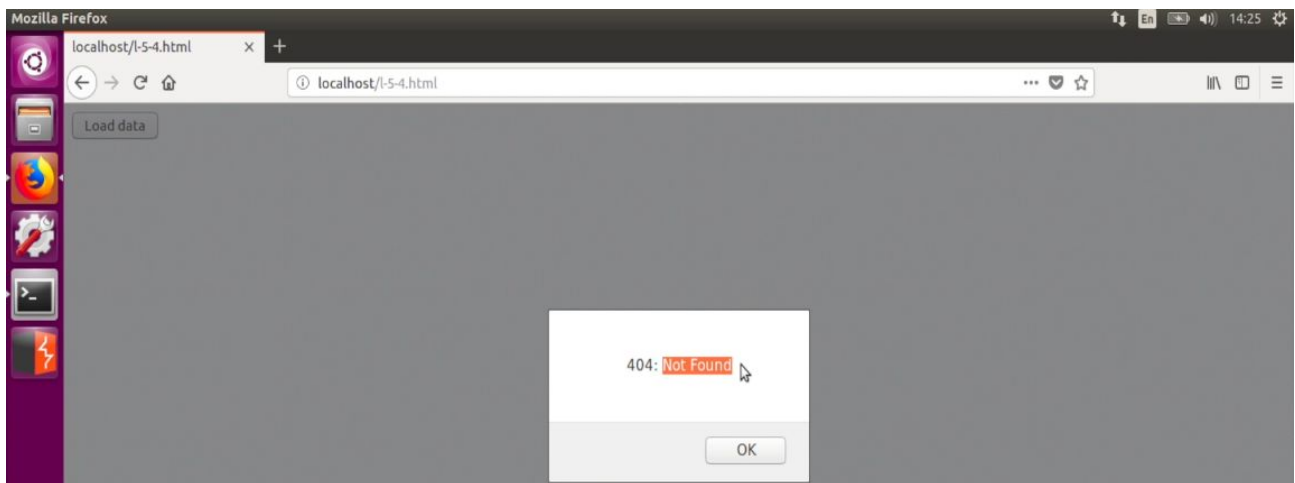
Мы добавили событие OnClick на кнопку, когда мы кликаем на нее, она выполняет указанное действие.

Загрузим данные со страницы, которой нет:

```
<script>
  function xhrTest() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "http://localhost/test.txt123", false);
    xhr.send();

    if (xhr.status != 200) {
      alert(xhr.status + ': ' + xhr.statusText);
    } else {
      alert(xhr.responseText);
    }
  }
</script>
<button onclick="xhrTest()">Load data</button>
```

В ответ высвечивается код и пояснение к ошибке:



Все работает!

XHR позволяет по умолчанию делать запросы только в пределах того домена, в котором вы сейчас находитесь. Подробнее об этом мы поговорим в одном из следующих уроков, когда будем разбирать политики безопасности браузеров и Same Origin Policy (SOP). Там мы разберемся, почему это так работает. Запомните что, например, если XHR делается с домена `geekbrains.ru` на домен `mail.ru`, ничего не получится — запрос уйдет, но сам ответ вы прочитать не сможете. Это все из-за SOP — политики одного origin. Но если запрос с `geekbrains.ru` идет на `geekbrains.ru`, вы сможете прочитать ответ.

Ещё здесь можно было написать не полный путь, а относительный — и все замечательно сработает:

```
<script>
function xhrTest() {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "test.txt", false);
  xhr.send();

  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText);
  } else {
    alert(xhr.responseText);
  }
}
</script>
<button onclick="xhrTest()">Load data</button>
```

Мы перезагрузим страницу и при этом вывод будет абсолютно такой же, как раньше.

Итоги

Подведем итоги:

- 1) Узнали, что такое XML HTTP Request — он нужен, чтобы мы могли с помощью JavaScript загружать страницу, менять в ней данные, но при этом не перезагружать её. Это нужно, чтобы

пользователь не видел перезагрузок страниц и не ждал, а все происходило на лету и без явного обновления веб-страниц.

- 2) Научились использовать XHR, создавать объект XHR, указывать метод, адрес URL, узнали, что такое синхронные и асинхронные запросы, попробовали сделать синхронный запрос, получили данные и отобразили их.
- 3) Узнали, какие ограничения на использование XHR существуют.

Видеоурок 5

На этом уроке мы разберем JSON и сериализацию:

- 1) Что такое JSON, зачем они нужны и посмотрим их на примере.
- 2) Чтение и написание JSON.
- 3) Сериализация и десериализация.

Формат данных JSON

JSON придумали, чтобы можно было передавать сложный объект из браузера на сервер или наоборот. До этого мы сталкивались с довольно простыми объектами: строка, число и т. п. Но есть и гораздо более сложные объекты, в которые входит, например, массив, в котором каждый элемент — еще один массив, и так далее. Еще есть такие элементы, как словари. Важно представить, что бывают очень сложные и большие объекты, которые просто так не передать на сервер. Для этого в JavaScript есть специальный формат данных — JSON. Это акроним от JavaScript Object Notation. Рассмотрим его:

```
cd /var/www/html && nano name.json
```

```
{ "name": "Nikita", "lastname": "Stupin" }
```

Фигурные скобки здесь обозначают не тело функции или цикла, а означают то, что это словарь.

Словарь — тип данных, который состоит из уже знакомых нам записей вида «ключ-значение»; таких пар может быть очень много. Вы можете, например, провести ассоциацию с обычным словарем, в котором содержатся слова и объяснение их значения. Например, толковый словарь русского языка будет содержать в себе список слов, и каждое из них — ключ, у которого есть значение. Здесь фактически то же самое: у нас есть `name` — это ключ, а после двоеточия (`:`) идет значение. Значение может быть любым типом данных, а элементы в словаре разделяются запятой. Дальше мы можем написать `lastname` и значение. В итоге мы указали имя: Никита и фамилию: Ступин.

Практика работы с JSON

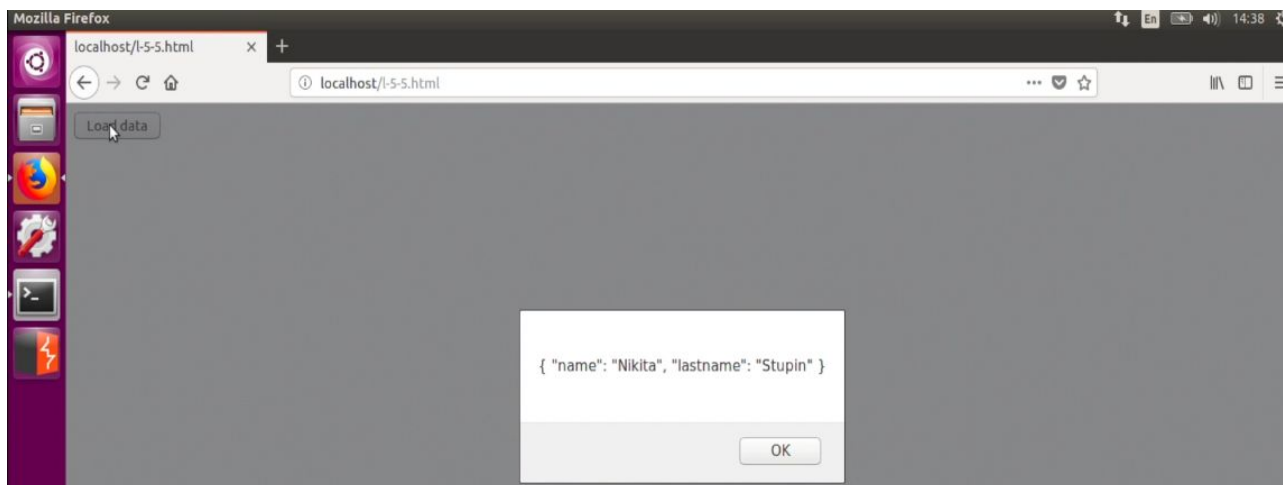
Теперь посмотрим, как взаимодействовать с JSON из JavaScript. Скопируем файл из предыдущего урока:

```
cp 1-5-4.html 1-5-5.html && nano 1-5-5.html
```

```
<script>
function xhrTest() {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "name.json", false);
  xhr.send();

  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText);
  } else {
    alert(xhr.responseText);
  }
}
</script>
<button onclick="xhrTest()">Load data</button>
```

Теперь мы будем загружать наш файл — name.json. Проверим, что произойдет, загрузим:



JSON отобразился просто как строка. На самом деле, с сервера и пришла строка, чтобы превратить её в объект. мы должны применить специальную функцию.

В JavaScript для этого есть функция `JSON.parse()` — от слова парсинг, парсер. Это значит, что мы разберем строку как JSON:

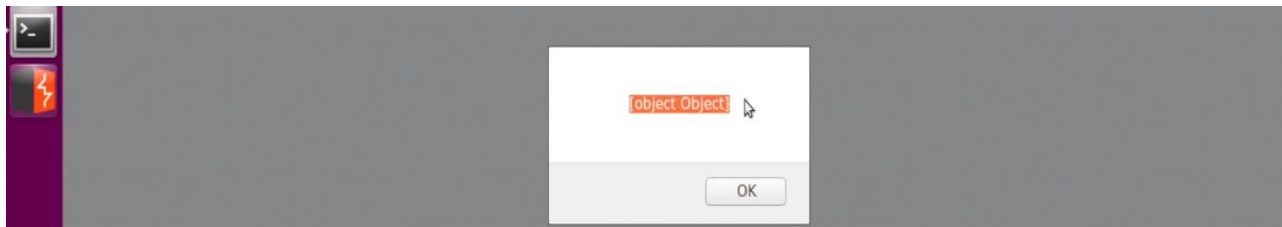
```
<script>
function xhrTest() {
  var xhr = new XMLHttpRequest();
  xhr.open("GET", "name.json", false);
  xhr.send();
```

```

    if (xhr.status != 200) {
        alert(xhr.status + ': ' + xhr.statusText);
    } else {
        var j = JSON.parse(xhr.responseText);
        alert(j);
    }
}
</script>
<button onclick="xhrTest()">Load data</button>

```

Запомним все в переменной (j) от JSON и посмотрим, что у нас получилось:



Теперь высвечивается не строка, а объект Object.

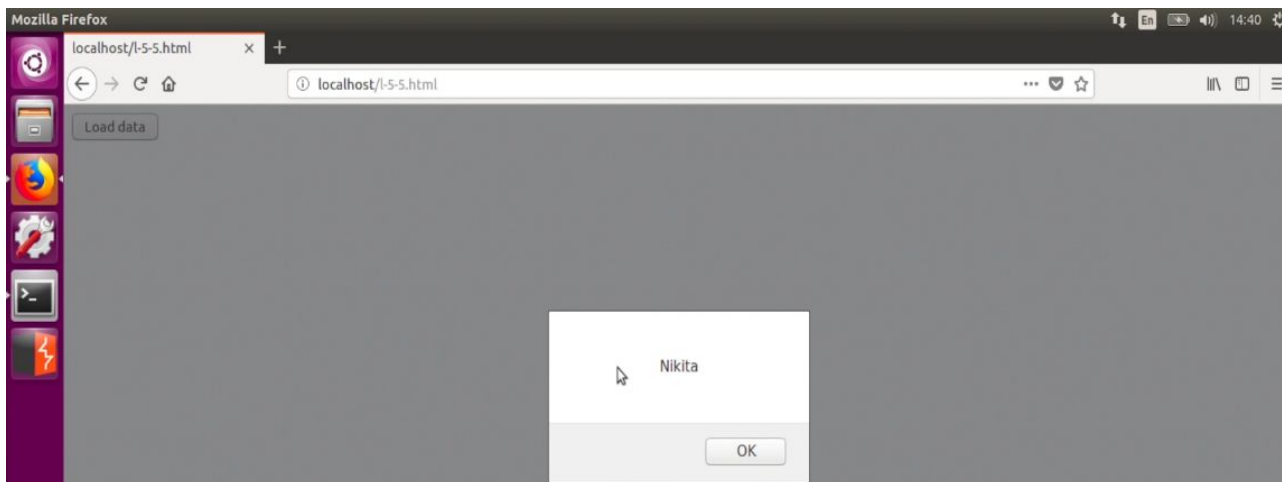
В переменной (j) находится объект и мы можем обратиться к его методам и свойствам. Напишем j.name и перезагрузим страницу:

```

<script>
function xhrTest() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "name.json", false);
    xhr.send();

    if (xhr.status != 200) {
        alert(xhr.status + ': ' + xhr.statusText);
    } else {
        var j = JSON.parse(xhr.responseText);
        alert(j.name);
    }
}
</script>
<button onclick="xhrTest()">Load data</button>

```



Теперь здесь выводится имя — Никита. Мы обратились к переменной `name`, которую задали ранее в виде JSON. Затем обратились к нему и достали значение. Мы уже работаем с JSON не как со строкой, а как с объектом. Мы превратили строку в объект, у которой есть два поля: `name` и `lastname` — фактически это даже не объект, а словарь.

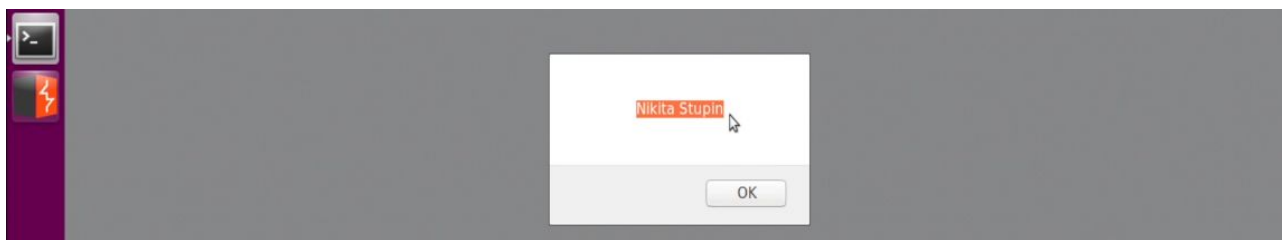
Также мы можем получить значение `name`, если обратимся к переменной `j["name"]` — то есть напишем так, как будто мы обращаемся к массиву, но не по индексу-числу, а по индексу-названию поля. Обычно так к словарям и обращаются. Можете проверить — будет то же самое.

Чаще всего к элементам словаря обращаются как раз вторым способом — через квадратные скобки, но можно и первым способом — через точку. Таким образом, мы можем вывести оба поля:

```
<script>
function xhrTest() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "name.json", false);
    xhr.send();

    if (xhr.status != 200) {
        alert(xhr.status + ': ' + xhr.statusText);
    } else {
        var j = JSON.parse(xhr.responseText);
        alert(j["name"] + " " + j["lastname"]);
    }
}
</script>
<button onclick="xhrTest()">Load data</button>
```

Это конкатенация, то есть операция соединения строк. Вывелось Nikita Stupin:



Теперь мы умеем отображать JSON.

Запись в JSON

JSON можно не только парсить, то есть читать данные, которые пришли на сервер в формате JSON, но и записать значения в JSON. Сделаем это:

```
<script>
function xhrTest() {
    var xhr = new XMLHttpRequest();
    xhr.open("GET", "name.json", false);
    xhr.send();

    if (xhr.status != 200) {
        alert(xhr.status + ': ' + xhr.statusText);
    } else {
        var j = JSON.parse(xhr.responseText);
        alert(j["name"] + " " + j["lastname"]);
        var serialized = JSON.stringify(j);
        alert(serialized);
    }
}
</script>
<button onclick="xhrTest()">Load data</button>
```

Искомая функция называется `JSON.stringify(j)` — то есть превращаем объект в строку. Выведем этот объект:



Наш объект — исходная строка.

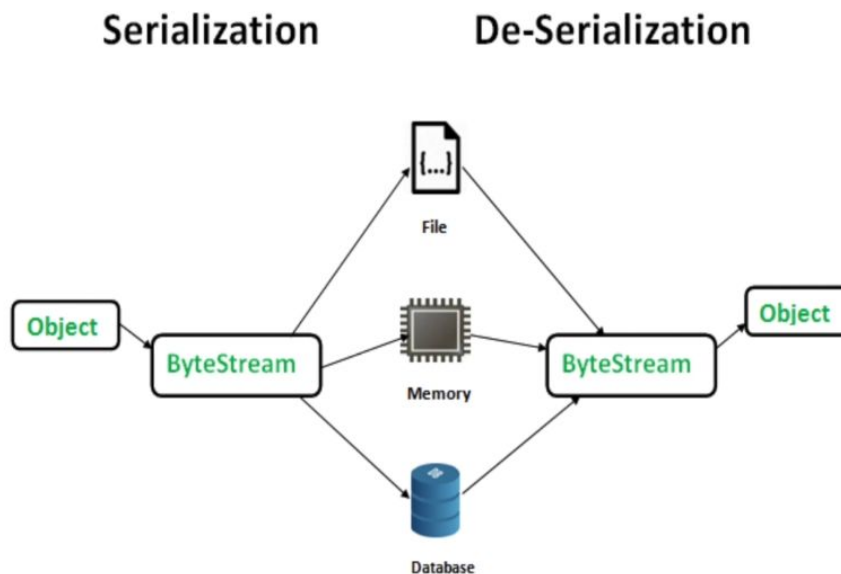
JSON так и работает: у нас есть метод, как доставать данные из него — `JSON.parse()` и метод, как записать их обратно — `JSON.stringify()`. Далее мы переменную `serialized` могли бы отправить на сервер и ему было бы удобно ее принять, а потом распаковать обратно.

Сериализация и десериализация

```
{  
  "firstName": "Иван",  
  "lastName": "Иванов",  
  "address": {  
    "streetAddress": "Московское ш., 101, кв.101",  
    "city": "Ленинград",  
    "postalCode": 101101  
  },  
  "phoneNumbers": [  
    "812 123-1234",  
    "916 123-4567"  
  ]  
}
```

На слайде выше вы видите пример более сложного JSON: здесь есть имя, фамилия, адрес, номера телефонов. Здесь словарь, у него есть 2 простых пары имя:значение. Есть еще один словарь в словаре, а в четвертом значении — массив. Таким образом, словарь — очень мощная структура данных и его передать на сервер уже не так просто. JSON с этим помогает.

Теперь в общем посмотрим, что такое сериализация и десериализация. Собственно, процесс записи в JSON, то есть превращение объекта в строку, называется сериализацией. В общем случае, когда мы из строки делаем объект, это называется десериализацией.



В общем случае, сериализация — превращение объекта в так называемый поток байтов или ByteStream. Потом мы этот поток пишем в файл, в память, в базу данных или передаем по интернет-каналу. А на другом конце тот, кто принял эти данные, должен их десериализовать, то есть

из нулей и единиц (байтов) сделать то, что будет представляться как объект — это и называется десериализацией.

Сериализация и десериализация используются не только для передачи данных между клиентом и сервером, но и чтобы сохранять сложные объекты в файлы. Например, между запусками программы сохранить данные и объекты, чтобы потом загрузить их и восстановить её состояние. Также сериализацию/десериализацию возможно делать в память, flash-память и базы данных.

Итоги

Подведем итоги, в этом видео вы узнали:

- 1) Что такое JavaScript Object Notation или, коротко, JSON, зачем он нужен и как применяется. Как его можно отправлять и принимать на сервере.
- 2) Как можно читать или писать в формате JSON (или, что вернее, сериализовать или десериализовать).
- 3) Что такое сериализация и десериализация, где она применяется и для чего нужна.

Итоги урока

На этом мы завершили изучение JavaScript. В следующем уроке мы узнаем, как обеспечивается безопасность в браузере:

- 1) Мы познакомимся с основными защитными механизмами браузеров. Изучим технологии, которыми пользуются разработчики при разработке, например, веб-приложений. Узнаем, как их сейчас защищают и как их правильно защищать.
- 2) В деталях разберем, что такое Content Security Policy (CSP) и освоим Same Origin Policy (SOP).

Ссылки к уроку

1. [Современный учебник JavaScript.](#)
2. [Basic JavaScript: Escaping Literal Quotes in Strings.](#)
3. [Получаем данные в JavaScript с помощью XMLHttpRequest.](#)
4. [Работа с JSON.](#)
5. [Формат JSON, метод toJSON.](#)
6. [JSON APIs and Ajax: Get JSON with the JavaScript XMLHttpRequest Method.](#)