JavaScript (jQuery)

LoftSchool
от мыслителя к создателю

Оглавление

jQuery. Работа с DOM

jQuery — библиотека JavaScript, фокусирующаяся на взаимодействии JavaScript и HTML. Библиотека jQuery помогает легко получать доступ к любому элементу DOM, обращаться к атрибутам и содержимому элементов DOM, манипулировать ими. Также библиотека jQuery предоставляет удобный API для работы с AJAX.

Чтобы обратиться к jQuery, достаточно написать **jQuery** либо **$**.

Для того, чтобы выбрать тот или иной элемент (или группу элементов) на странице, обращаемся к jQuery и передаем нужный селектор.
Например:

```
jQuery('p'); //выбрать все элементы p
jQuery('.className'); //выбрать все элементы с классом className
jQuery('#idName'); //выбрать все элементы с id idName
jQuery('ul li'); //выбрать все элементы li, которые находятся в элементе ul
```

То же самое можно сделать с помощью знака $:

```
$('p'); //  выбрать все элементы p
$('.className'); //выбрать все элементы с классом className
$('#idName'); //выбрать все элементы с id idName
$('ul li'); //выбрать все элементы li, которые находятся в элементе ul
```

Зная css-селекторы, мы просто передаем их объекту jQuery.

Мы можем применять к элементу те или иные методы: изменить содержимое, задать ширину, высоту, добавить/ удалить/ изменить атрибут и т.д.

Мы уже знаем, как выбирать соседей, родителей, потомков на чистом Javascript, теперь давайте посмотрим, как это делается с помощью jQuery.

closest() - Находит ближайший, соответствующий заданному селектору элемент, из числа следующих: сам выбранный элемент, его родитель, его прародитель, и так далее, до начала дерева DOM.

prev() - Находит элементы, которые лежат непосредственно перед каждым из выбранных элементов.

next() - Находит элементы, которые лежат непосредственно после каждого из выбранных элементов.

siblings() - Находит все соседние элементы (под соседними понимаются элементы с общим родителем).

first() - Возвращает первый элемент в наборе.

last() - Возвращает последний элемент в наборе.

Например, нам нужно скрыть все соседние элементы, но оставить видимым второй элемент:

```
<div>first</div>
<div class="second">second</div>
<div>third</div>
<div>fourth</div>

$('.second').siblings().hide();
```

Давайте посмотрим, как можно вытащить текст из выбранного элемента:

```
<p>Lorem ipsum dolor sit amet, consectetur adipisicing
elit. Dolorem laudantium, modi molestiae consectetur!
At dolore possimus optio temporibus dolor, eum est
debitis qui, totam soluta ratione veritatis, quos
tenetur. Eum necessitatibus vero, nam accusamus eos
minima ea quibusdam deserunt laboriosam autem. At cum
quod, laboriosam ab dolorum, alias illo iste.</p>

var text = $('p').text();
console.log(text);
```

И еще некоторые методы, которые могут вам пригодится:

attr() - Возвращает/изменяет (в зависимости от числа параметров) значение атрибута у элементов на странице;

removeAttr() - Удаляет атрибут у элементов на странице;

addClass() - Добавляет класс элементам на странице;

removeClass() - Удаляет класс(ы) у элементов на странице;

toggleClass() - Изменяет наличие класса у элементов на противоположное (добавляет/удаляет);

css() - Возвращает/изменяет (в зависимости от числа входных параметров) CSS параметры элемента;

html() - Возвращает/изменяет (в зависимости от числа параметров) html-содержимое элементов на странице;

text() - Возвращает/изменяет (в зависимости от числа параметров) текст, находящийся в элементах на странице;

remove() - Удаляет элементы на странице

События

Механизм DOM-события призван помочь разработчику перехватить и обработать различные действия пользователей (клики мышкой по элементам, нажатия клавиш и прочее).

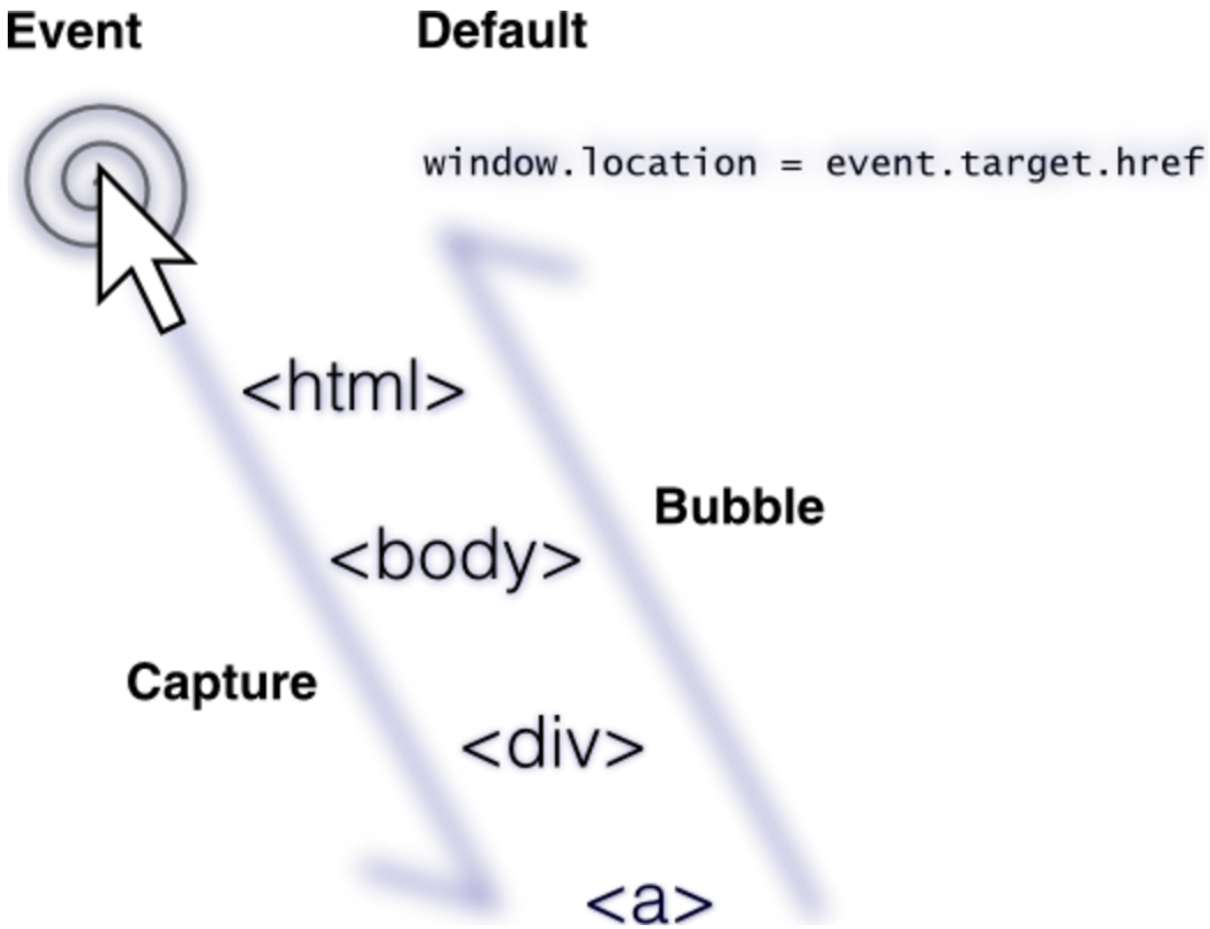
Почти все (не все) события состоят из двух фаз:

- **capturing(захват)**
- **bubbling(всплытие)**

Когда происходит событие, информация о нем спускается от корневого элементам dom-дерева вниз, до того элемента, на котором произошло событие (например, если кликнули по ссылке, то событие спускается от корня дерева до ссылки). Этот процесс называется **фазой захвата**.

После этого, процесс начинается в обратном порядке. То есть информация о событии поднимается от элемента, на котором произошло событие до корневого элемента dom-дерева. Этот процесс называется **фазой всплытия**. После того, как событие “всплывет” происходит действие по умолчанию. Например, для ссылок, действием по умолчанию является переход на сайт, заданный в атрибуте href.

Ниже приведен рисунок, который поможет лучше понять суть происходящего:



Разработчик может установить свой обработчик события на любую из фаз.

Регистрация обработчиков событий

Для того, чтобы добавить обработчик события на элемент, есть несколько способов.

1) Добавить атрибут **on*** (вместо звездочки подставить **имя события**)

```
<button onclick="alert('click')">click me</button>
```

Мы добавили обработчик кликов мышкой по элементу.
Таким образом, при клике мышкой по элементу, сработает код, который указан в значении атрибута **onclick** (в нашем случае, это **alert('click')**).

Мы не рекомендуем добавлять таким образом обработчики события, т.к. это считается плохим стилем и влечет за собой путаницу между логикой и ее представлением.

2) Присвоить в свойство **on*** элемента **функцию**, которая отработает при наступлении событии. Например:

```
<button>click me</button>

<script>
  var button = document.querySelector('button'); // находим элемент

  button.onclick = function () {
    alert('click');
  }
</script>
```

Этот вариант лучше первого, но существует недостаток - мы не можем добавить несколько обработчиков событий. И хотя данную проблему можно обойти, существует и третий, наиболее предпочтительный метод.

3) Установка обработчиков через метод **addEventListener**

Это наиболее предпочтительный вариант. Мы рекомендуем использовать именно его. При помощи данного метода, вы можете задавать несколько обработчиков события.

```
var button = document.querySelector('button');

button.addEventListener('click', function() {
    alert('click');
})

button.addEventListener('click', function() {
    alert('second');
})
```

При клике на кнопку сработают оба обработчика. В том порядке, в котором они были установлены.

Первым параметром необходимо передать **имя события**, на которое нужно установить обработчик.
Вторым параметром - **функцию**, которая будет выполнять роль обработчика.

Есть так же третий параметр.
Дело в том, что любой обработчик по умолчанию выполняется на фазе всплытия.
Если в addEventListener третьим параметром передать **true**, то обработчик будет выполняться на фазе захвата.

Для того, чтобы удалить обработчик события с элемента, необходимо воспользоваться методом **removeEventListener**:

```
function handler1() {
    console.log('обработчик 1');
}

function handler2() {
    console.log('обработчик 2');
}

var button = document.querySelector('button');

button.addEventListener('click', handler1);
button.addEventListener('click', handler2);
button.removeEventListener('click', handler2);
```

При клике на кнопку, сработает только первый обработчик, т.к. второй обработчик мы удалили сразу после того, как добавили.
Обратите внимание, что при удалении обработчика, необходимо указать переменную с функцией-обработчиком. Иначе браузер не сможет удалить обработчик из элемента. Для этого, просто храните обработчики в именованных функциях или в переменных.

Объект события

Когда браузер вызывает обработчик события, он передает в эту функцию первым параметром специальный объект с описанием события. Вот основные свойства этого объекта:

type - имя события
target - элемент, для которого изначально было предназначено событие
currentTarget - элемент, который перехватил событие в данный момент
eventPhase - фаза события (захват, выполнение, всплытие)

```
button.addEventListener('click', function(e){
    console.log(e);
});
```

У данного объекта так же есть метод **preventDefault**, который мы сейчас рассмотрим.

Действие по умолчанию

У html-элементов есть действие по умолчанию.
Например:
При клике на элементы **<a>** - происходит переход по ссылке из атрибута **href**.
При нажатии клавиш в текстовое поле - вывод введенных символов.

Механизм событий позволяет отменить действие по умолчанию при помощи вызова метода **preventDefault** из объекта с информацией о событии. Например, вот как можно запретить переход по ссылке:

```
<a href="http://ya.ru" id="link">link</a>

    <script>
    link.addEventListener('click', function(e){
        e.preventDefault();
    });
</script>
```

Теперь, при клике по ссылке, перехода осуществлено не будет.

События мыши

Наверное, одно из самых распространенных событий, это клик “мышкой” по элементу.

```
<button>Click me</button>

<script>
var button = document.querySelector('button'); // выбираем кнопку

button.addEventListener('click', function() { //вешаем обработчик события на клик
    alert('click'); //выводим сообщение по клику
});
</script>
```

Менее распространенный - **dblclick**, срабатывает при двойном клике.

mousedown - это событие отрабатывает на нажатие левой клавиши. Разница между им и click в том, что событие click - это нажатие и отпускании левой кнопки мышки, а mousedown - только нажатие.

mouseup - это событие отрабатывает при отпускании кнопки мышки.

По факту, событие click вмещает в себя два события - mousedown и mouseup. Выполняются они в таком порядке: mousedown, mouseup, click.

mouseover - срабатывает по наведению курсора на элемент;

mouseout - срабатывает по отводу курсора с элемента;

mousemove - срабатывает при каждом движении курсора над элементом;

События клавиатуры

keydown - отрабатывает по нажатию на клавишу
keyup - отрабатывает по отжатию клавиши

```
<input type="text" />

<script>
var input = document.querySelector('input');

input.addEventListener('keydown', function() { //по
нажатию на
    alert('key'); //клавишу в текстовом поле отработает alert
})
</script>
```

keypress - отрабатывает по нажатию на клавишу.
В отличии от keydown, событие keypress срабатывает только на те ***клавиши, которые печатают символы***. То есть такие клавиши как ctrl, alt, shift не входят в эту категорию, так как не печатают символы.

```
<input type="text" />

<script>
var input = document.querySelector('input');

input.addEventListener('keypress', function() {
    alert('key');
});
</script>
```

По нажатию на shift, fn, ctrl, alt, backspace - код выше не сработает.

В объект событий keydown, keypress, и keyup, передается свойство keyCode, которое содержит в себе код введенного символа по ASCII-таблице(пример таблицы по ссылке).
Мы можем получить непосредственно введенный символ из его кода, при помощи Метода **String.fromCharCode(code)**.
Либо можем обрабатывать “сырые” коды клавиш. Например, код с 48 по 57 - это числа от 0 до 9. Зная это, мы можем очень просто сделать обработчик ввода, который не будет давать вводить в текстовое поле ничего кроме цифр:

```
inp.addEventListener('keydown', function(e) {
    if (e.keyCode < 48 || e.keyCode > 57) {
        e.preventDefault();
    }
});
```

input - событие, которое выполняется для текстовых полей, при каждом изменении значения в поле. В отличие от остальных событий клавиатуры, в объект события не передается дополнительных свойств типа charCode. input просто сообщает о том, что содержимое текстового поля было изменено.

Указанные события выполняются в такой последовательности: ***keydown, keypress, input, keyup***.

События загрузки документа

Загрузка документа делится на два этапа:

- **загрузка html и скриптов**
- **загрузка внешних ресурсов**

Каждый из этих этапов можно перехватить и обработать.

DOMContentLoaded - событие, которое сработает после того, как весь html будет загружен браузером и js в документе будет выполнен. То есть все dom-элементы будет созданы и доступны через JS.
Данное событие срабатывает на объекте **document**:

```
document.addEventListener("DOMContentLoaded",
function() {
    console.log("html-структура загружена");
});
```

load - событие, которое сработает, когда браузер загрузит все внешние ресурсы (картинки, css) и страница будет полностью готова к работе.
Данное событие срабатывает на объекте **window**:

```
window.addEventListener('load', function() {
    console.log("Страница полностью готова");
});
```

Делегирование

Делегирование - это когда элементы возлагают обязанности по обработке своих событий на вышележащие элементы.

Предположим, что у нас есть div, внутри которого располагаются 1000 кнопок.
Вешать обработчик события на каждую кнопку - не разумно.
Проще повесить один обработчик на div.
Ведь при клике на кнопку, события об этом сначала погрузится от корня документ до кнопки, по которой произошел клик, пройдя через div, а затем всплывет от кнопки, по которой произошел клик до корня документа, тоже пройдя через div.
Мы можем повесить обработчик события click для div на любую фазу, и клики по любой кнопке будут так и иначе перехвачены всего лишь одним общим обработчиком. Это и есть делегирование.

Предотвращение распространения события

Как вы помните, бОльшая часть событий сначала погружается вниз по дереву, а затем - всплывает. А по умолчанию, обработчики событий выполняются на фазе всплытия.

Внутри обработчика события мы можем “приказать” событию не следовать дальше по дереву, после выполнения данного обработчика.

Это можно сделать при помощи метода **stopPropagation** объекта события:

```
<div id="div1">
  <div id="div2">
    <div id="div3">
      <button id="button">click me</button>
    </div>
  </div>
</div>

<script>
  var els = document.querySelectorAll('body *');

  for (var i = 0; i < els.length; i++) {
    els[i].addEventListener('click', function (e) {
      console.log(e.currentTarget.id); //эквивалентно
console.log(this.id);
    });
  }
</script>
```

Здесь мы выбираем все элементы внутри body (3 дива и кнопка) и на каждый из них вешаем обработчик события **click**. Обработчики этих элементов сработаеют при клике на кнопку, при всплытии события, то есть когда событие будет идти уже от кнопки к корню дерева.

Задача данного обработчика проста - вывести **id** элемента, который в данный момент обрабатывает событие. Для этого, мы обращаемся с свойству **currentTarget** объекта события. Мы не можем обратиться к свойству target, потому что, как вы помните, свойство target всегда

ссылается на тот элемент, на котором произошло событие, а не на тот элемент, на котором сейчас сработал обработчик.

Соответственно, после клика на кнопку, выведет в консоль сообщения в таком порядке:

button
div3
div2
div1

Именно в таком, потому что обработчики выполнятся на фазе всплытия события от элемента.

Теперь давайте добавим к элементу с id=div3 такой обработчик, который будет препятствовать дальнейшему всплытию события, после того, как этот обработчик сработает.
После цикла for добавим еще один обработчик к элементу div3:

```
div3.addEventListener('click', function(e) {
  e.stopPropagation();
});
```

Соответственно, когда событие, всплывая, дойдет до элемента div3, то сначала сработает первый обработчик на этом элементе, который выведет id данного элемента, а затем сработает тот обработчик, который мы только что написали, который, в свою очередь, прервет дальнейшее всплытия события засчет stopPropagation.

Соответвенно, обработчики на элементах div2 и div1 выполнены не будут, т.к. событие просто не всплывет к ним.

Убедитесь сами, в консоль будет выведено:

button
div3

jQuery события

Библиотека jQuery помогает нам решить много проблем, связанных с кроссбраузерностью, в ней есть как стандартные методы JavaScript, так и свои собственные.

on() - С помощью этого метода мы будем устанавливать обработчики событий.

```
<button>click</button>
<script>
  var button = $('button');

  button.on('click', function() { // по нажатию на
    alert('click'); //  кнопку отработает alert
  });
</script>
```

Вешаем обработчик события с помощью on(), передавая первым аргументом тот или иной обработчик события.

click() - Устанавливает обработчик клика левой клавишей мыши по элементу.

dblclick() - Устанавливает обработчик двойного "клика" мышью по элементу, либо запускает это событие.

hover() - Устанавливает обработчик двух событий - появления/ исчезновения курсора над элементом.

Например:

```
<button>click</button>
<script>
  var button = $('button');

  button.hover(
          function() { // по наведению
            alert(first); // кнопку отработает 1-й alert
          },

          function() { //  при отведении курсора сработает второй alert
            alert('second');
          });
</script>
```

mousedown() - Устанавливает обработчик нажатия кнопки мыши.

mouseup() - Устанавливает обработчик поднятия кнопки мыши.

mouseenter() - Устанавливает обработчик появления курсора в области элемента. Появление этого события отработано лучше, чем стандартного mouseover.

mouseleave() - Устанавливает обработчик выхода курсора из области элемента. Появление этого события отработано лучше, чем стандартного mouseout.

mousemove() - Устанавливает обработчик движения курсора в области элемента.

mouseout() - Устанавливает обработчик выхода курсора из области элемента, либо запускает это событие.

toggle() - Поочередно выполняет одну из двух или более заданных функций, в ответ на "клик" по элементу.

Deffered object

Deferred объект (в jQuery) — это всего лишь хранилище состояния асинхронной функции. Таких состояний обычно несколько:

- **pending** — ожидание завершения процесса
- **rejected** — процесс закончен падением
- **resolved** — процесс закончен успешно

Кроме того, у Deferred объекта есть ряд методов, которые могут менять его состояние.
По состоянию Deferred объекта мы можем судить, закончен ли процесс, состояние которого мы отслеживаем.

Список методов:

.done() .fail() .then() .always()
регистрируют обработчики перехода объекта deferred в состояние "выполнено"/"ошибка выполнения" (resolved/rejected) (then() регистрирует два обработчика сразу, а .always() общий обработчик на оба события)

.progress()
регистрирует обработчики прогресса выполнения объекта deferred

.resolved() .reject()
переводят объект deferred из состояния "не выполнено" в "успешно выполнено"/"ошибка выполнения"

.notify()
вызывает событие частичного выполнения deferred (его прогресса выполнения).

```
var deferred = $.Deferred(function(obj) {
    obj.done(someCallback);
});
```

При создании, объект jQuery.Deferred находится в состоянии unresolved (еще не выполнено). После этого, состояние объекта можно изменить на resolved (выполнено) с помощью метода .resolve() или .resolveWith(), а так же в состояние rejected (ошибка при выполнении), если вызвать метод .reject() или .rejectWith().

Важно отметить, что объект jQuery.Deferred может изменить свое состояние только один раз!

С помощью метода **.done()** можно установить обработчик удачного выполнения объекта Deferred, **.fail()** установит обработчик неудачного выполнения. В методе **.then()** можно задать оба вида обработчиков, а **.always()** установит обработчик, который будет вызван при переходе в любое из состояний. Если установить обработчик на объект Deferred, который уже находится в выполненном состоянии, то он (обработчик) будет запущен незамедлительно:

```
// Реализуем функцию test, которая запустит someAction() в течении 10 секунд.
//  Возвращаемый объект Deferred будет оповещать о выполнении someAction()

function test() {
    var d = $.Deferred(),
        actTime = 10000 * Math.random(); // время запуска 0-10 сек

    setTimeout(function () {
        someAction(); // выполняем интересующую функцию
        d.resolve();  // изменяем состояние Deferred-объекта на "выполнено"
    }, actTime);

    return d;
}

var defrr = test();

// Устанавливаем обработчик выполнения Deferred-объекта
defrr.done(function () {
    alert("someAction выполнен");
```

Ajax - реализация в jQuery

jQuery.ajax

Это основной метод, а все последующие методы лишь обертки для метода jQuery.ajax. У данного метода лишь один входной параметр – объект включающий в себя все настройки

- **async** – асинхронность запроса, по умолчанию true
- **cache** – вкл/выкл кэширование данных браузером, по умолчанию true
- **contentType** – по умолчанию “application/x-www-form-urlencoded”
- **data** – передаваемые данные – строка или объект
- **dataFilter** – фильтр для входных данных
- **dataType** – тип данных возвращаемых в callback функцию (xml, html, script, json, text, _default)
- **global** – триггер – отвечает за использование глобальных AJAX Event’ов, по умолчанию true
- **ifModified** – триггер – проверяет были ли изменения в ответе сервера, дабы не слать еще запрос, по умолчанию false
- **jsonp** – переустановить имя callback функции для работы с JSONP (по умолчанию генерируется на лету)
- **processData** – по умолчанию отправляемые данный заворачиваются в объект, и отправляются как “application/x-www-form-urlencoded”, если надо иначе – отключаем
- **scriptCharset** – кодировка – актуально для JSONP и подгрузки JavaScript’ов
- **timeout** – время таймаут в миллисекундах
- **type** – GET либо POST
- **url** – url запрашиваемой страницы

Локальные AJAX Event’ы:

- **beforeSend** – срабатывает перед отправкой запроса
- **error** – если произошла ошибка
- **success** – если ошибок не возникло
- **complete** – срабатывает по окончанию запроса

Для организации HTTP-авторизации:

- **username** – логин
- **password** – пароль

Пример AJAX-запроса:

```
$.ajax({
    url: ‘/ajax/example.html', // указываем URL и
    dataType: «json", // тип загружаемых данных
    success: function(data, textStatus) { // вешаем свой
обработчик на функцию success
        $.each(data, function(i, val) { // обрабатываем
полученные данные
            /* ... */
        });
    }
});
```

jQuery.get

Загружает страницу, используя для передачи данных GET запрос. Может принимать следующие параметры:

- **url** запрашиваемой страницы
- передаваемые **данные** (необязательный параметр)
- **callback функция**, которой будет скормлен результат (необязательный параметр)
- **тип данных** возвращаемых в callback функцию (xml, html, script, json, text, _default)

jQuery.post

Данный метод аналогичен предыдущему, лишь передаваемые данные уйдут на сервер посредством POST’а. Может принимать следующие параметры:

- **url** запрашиваемой страницы
- передаваемые **данные** (необязательный параметр)
- **callback функция**, которой будет скормлен результат (необязательный параметр)
- **тип данных** возвращаемых в callback функцию (xml, html, script, json, text, _default)

jQuery.getJSON

Загружает данные в формате JSON (удобней и быстрее нежели XML). Может принимать следующие параметры:

- **url** запрашиваемой страницы
- передаваемые **данные** (необязательный параметр)
- **callback функция**, которой будет скормлен результат (необязательный параметр)

AJAX-события в jQuery

Для удобства разработки, на AJAX запросах висит несколько event'ов, их можно задавать для каждого AJAX запроса в отдельности, либо глобально. На все event'ы можно повесить свою функцию:

```
$.ajax({
    beforeSend: function() {
        // Handle the beforeSend event
    },
    complete: function() {
        // Handle the complete event
    }
    // ...
});
```

Список всех event'ов:

- **ajaxStart** – Вызывается в случае когда побежал AJAX запрос, и при этом других запросов нету
- **beforeSend** – Срабатывает до отправки запроса, позволяет редактировать XMLHttpRequest. Локальное событие
- **ajaxSend** – Срабатывает до отправки запроса, аналогично beforeSend
- **success** – Срабатывает по возвращению ответа, когда нет ошибок ни сервера, ни вернувшихся данных. Локальное событие
- **ajaxSuccess** – Срабатывает по возвращению ответа, аналогично success
- **error** – Срабатывает в случае ошибки. Локальное событие
- **ajaxError** – Срабатывает в случае ошибки
- **complete** – Срабатывает по завершению текущего AJAX запроса (с ошибкои или без – срабатывает всегда).Локальное событие
- **ajaxComplete** – Глобальное событие, аналогичное complete
- **ajaxStop** – Данный метод вызывается в случае когда больше нету активных запросов

JSONP

Отдельно стоит отметить использование JSONP – ибо это один из способов осуществления кросс-доменной загрузки данных.

При работе с jQuery, имя callback функции генерируется автоматически для каждого обращения к удаленному серверу, для этого достаточно использовать GET запрос ввида:

http://api.domain.com/?type=jsonp&query=test&callback=?

Вместо последнего знака вопроса (?) будет подставлено имя callback функции:

```
$.getJSON("http://example.com/something.json?
callback=?", function(result) {
    alert(result); // Выводим результат
});
```

CORS (Cross-Origin Resource Sharing)

По умолчанию - нельзя отправлять AJAX-запросы на хосты, отличные от того, с которого пришла страничка. Браузер просто не даст этого сделать.
Если только сервер, в ответ на запрос браузера, не отдаст специальные заголовки, по которым браузер поймет, что сервер совершенно не против, чтобы к нему обратились с посторонней странички.

CORS - это набор HTTP заголовков, которые позволяют объяснить браузеру и серверу, что они хоть и из разных доменов, но работать могут вместе. Т.е. обеспечивается поддержка кросс-доменных запросов

Сервер, предоставляющий поддержку кросс-доменных запросов, должен сообщить клиенту об этом, отправив следующие заголовки:

- **Access-Control-Allow-Origin**
- **Access-Control-Allow-Methods**
- **Access-Control-Allow-Headers**

Access-Control-Allow-Origin — (обязательный) список (через пробел) допустимых доменов (источников), которые могут делать запросы на данный сервер. Из особенностей: регистрозависим, поддерживает маски, например, http://api.superservice.com/, http://*.superservice.com/ или *.
Этот заголовок будет сравниваться с заголовком **Origin** клиентского запроса.

Access-Control-Allow-Method — (не обязательный) это список доступных HTTP методов, разделенных запятыми.

Access-Control-Allow-Headers — (не обязательный) список (через запятую) заголовков разрешенных в запросе.

Пример настройки для Apache:
CORS заголовки (добавте это, например, в .htaccess)

```
<ifmodule mod_headers.c>

Header always set Access-Control-Allow-Origin: "*"
   Header always set Access-Control-Allow-Methods
"POST, GET, PUT, DELETE, OPTIONS"
   Header always set Access-Control-Allow-Headers "X-
Requested-With, content-type»

</ifModule>
```

Тот же пример для PHP:

```
<?php
header('Access-Control-Allow-Origin: *');
header('Access-Control-Allow-Methods: POST, GET, PUT,
DELETE, OPTIONS');
header('Access-Control-Allow-Headers: X-Requested-
With, content-type');
?>
```

Теперь пример самого запроса на jQuery:

```
jQuery.ajax({
    url: 'http://api.superservice.com/credit-card-ids',
    type: 'GET',
    contentType: 'application/json', // Формат ответа от сервера
    headers: {}, // Разные заголовки, нестандартные заголовки, указываем
их в Access-Control-Allow-Headers
    success: function(res) {
        console.log(res);
    },
    error: function() {
        //.....
    }
});
```

Для выполнения запросов типа POST/PUT/DELETE, сначала надо получить список заголовков типа Access-Control-*, делается это предварительным OPTIONS запросом. Который может вернуть тот же набор Access-Control-*, что и обычный метод GET.

Этот предварительный запрос (prefligth-запрос), делается автоматически, по тому же URL что и ваш основной запрос POST/PUT/DELETE поэтому будьте внимательны - если вы не сделаете правильной обработку запросов типа OPTIONS то и запрос POST/PUT/DELETE у вас сделать не получиться, из-за политики безопасности, который следуют браузеры:

```
jQuery.ajax({
    ulr: 'http://api.superservice.com/credit-card-ids',
    type: 'POST',
    data: '{"content": "' + content + '"}', // отправляемые json-
данные
    contentType: 'application/json', // тип ответа от сервера
    headers: {}, // разные заголовки, нестандартные заголовки, не
забудьте их указать в Access-Control-Allow-Headers
    success: function(res) {
        console.log(res);
    },
    error: function() {
        //....
    }
});
```

Шаблонизация на JavaScript, Handlebars.js

Handlebars является одним из наиболее популярных, быстрых и многофункциональных ***шаблонизаторов для JavaScript***.

Он практически не позволяет добавлять логику и произвольный JavaScript в шаблоны (кроме циклов и условных выражений), тем самым он как бы «заставляет» разработчика содержать логику отдельно от разметки.

Handlebars используют такие JavaScript-фреймворки, как Meteor.js, Derby.js, Ember.js, также с ним отлично взаимодействуют и другие фреймворки, например Backbone.js.

Использование Handlebars.js

Чтобы начать использовать Handlebars в своем проекте, необходимо подключить файл **handlebars.js**. Самый простой способ определения шаблона Handlebars — внедрение его непосредственно на страницу внутри тегов <script>.

```
<script id="header" type="text/x-handlebars-template">
  <div>Title: {{title}}</div>
</script>
```

Переменные шаблона, а также выражения заключаются в двойные фигурные скобки **{{...}}**. Данные в шаблон передаются в виде обычного объекта, переменным шаблона соответствуют свойства этого объекта. И наконец, HTML-структура с внедренными данными формируется функцией **Handlebars.compile()**.

```
<script id="book" type="text/x-handlebars-template">
  <h2>{{bookTitle}}</h2>
  <div>Author: {{bookAuthor}}</div>
</script>
```

```
// Инициализируем объект с данными
var book = { bookTitle: '2001: A Space Odyssey',
bookAuthor: 'Arthur Clarke' };

// Получаем шаблон
var templateScript = $('#book').html();

// Функция Handlebars.compile принимает шаблон и возвращает
новую функцию
var template = Handlebars.compile(templateScript);

// Формируем HTML и вставляем в документ
$(document.body).append(template(book));
```

Синтаксис Handlebars.js

Выражения Handlebars заключаются в двойные фигурные скобки. Выражением может быть как переменная, так и хелпер — предварительно определенная функция, результат которой будет вставлен в шаблон.

Также Handlebars поддерживает несколько встроенных хелперов, или блоков, каждый из которых ***открывается символами «{{#», а закрывается символами «/}}»***.
Про встроенные хелперы мы еще поговорим подробнее, примером такого блока может служить блок if:

```
{{#if someValueI}} Content {{/if}}
```

Комментарии Handlebars заключаются в символы **{{! }}** (конечно же, можно использовать обычные HTML-комментарии).

```
{{! Comment content }}
```

Если свойство объекта данных, в свою очередь, тоже является **объектом**, можно вставлять в шаблон значения его свойств через точку, например:

```
// Объект с данными
var data = { name: { firstName: 'John', lastName:
'Smith' } }

// Шаблон
    <div>{{name.firstName}}</div>
```

Если свойство объекта данных является **массивом**, можно реализовать вывод элементов массива с помощью цикла each:

```
var data = {
    groupName: 'Customers',
    users: [
        { name: { firstName: 'John', lastName: 'Smith'
}},
        { name: { firstName: 'Thomas', lastName:
'Anderson' }}
    ]
};


<script id="users-template" type="x-handlebars-
template">
    {{#each users}}
<li>
{{name.firstName}} {{name.lastName}} is in the {{../
groupName}} group.
</li>
{{/each}}
</script>
```

Обратите внимание - при заходе в цикл меняется объект данных (в терминах Handlebars он называется контекстом). Вернуться в родительский контекст позволяет выражение «../».

Handlebars экранирует любые HTML-строки, переданные в шаблон. Чтобы вывести исходный HTML, необходимо использовать тройные фигурные скобки {{{ ... }}}.

Browserify

Browserify - это библиотека, которая позволяет вам использовать стиль node.js модулей для работы в браузере.

Мы определяем зависимости и потом Browserify собирает их в один JavaScript файл.

Вы подключаете ваши JavaScript файлы используя require("./ваш_файл.js"); выражение. Также вы можете использовать публичные модули из npm.

Для Browserify не составляет никакого труда создание source map'ов (карт исходных файлов до компрессии), так что даже не смотря на конкатенацию, вы сможете отлаживать отдельные части пакета ровно так же, как вы и привыкли это делать с отдельными файлами.

Для подключения собственных модулей используем директиву **require()** . Следующий код импортирует ваш файл и присваивает его значению переменной greatestModuleEver:

```
greatestModuleEver = require('./your_module.js');
```

При этом, в импортируемом файле всего лишь необходимо использовать структуру модуля, используя **module.exports**

```
module.exports = function(vars) {
    // Your code
}
```

Чтобы начать работать с Browserify, вам необходимо иметь следующее:

node.js
npm – по умолчанию поставляется с node.js
Browserify

И, конечно же, набор JS-файлов, с которыми вы хотите работать. Устанавливаем Browserify используя следующую команду:

```
npm install –g browserify
```

Пример запуска из командной строки:

```
browserify js/main.js –o js/findem.js –d
```

Эта команда читает main.js файл, обрабатывает, собирает зависимости и заисывает в findem.js Ключ -d Обеспечивает поддержку source map что очень удобно при дальнейшей отладке.

Пример использования:

app.js

```
var hideElement = require('./hideElement');

hideElement('#some-id');
```

hideElement.js

```
var $ = require('jquery');
module.exports = function(selector) {
    return $(selector).hide();
};
```

gulpfile.js

```
var browserify = require('browserify');
var bundle = browserify('./app.js').bundle()
```

Запуск ***app.js*** через Browserify приведет к следующему:

- app.js требует hideElement.js
- hideElement.js требует модуль, который называется jquery
- Browserify собирает jQuery, hideElement.js, и app.js в один файл, гарантируя, что зависимости будут разрешены, когда это понадобится

Пример автоматизации процесса через gulp:

```
var browserify = require('browserify'),
    watchify = require('watchify'),
    gulp = require('gulp'),
    source = require('vinyl-source-stream'),
    sourceFile = './js/main.js',
    destFolder = './js/',
    destFile = 'findem.js';

gulp.task('browserify', function() {
    return browserify(sourceFile)
        .bundle()
        .pipe(source(destFile))
        .pipe(gulp.dest(destFolder));
});

gulp.task('watch', function() {
    var bundler = watchify(sourceFile);
    bundler.on('update', rebundle);

    function rebundle() {
        return bundler.bundle()
            .pipe(source(destFile))
            .pipe(gulp.dest(destFolder));
    }

    return rebundle();
```

```
});

gulp.task('default', gulp.parallel('browserify',
'watch'));
```

Полезные ссылки:

- The Browserify Handbook – Карманный справочник Джеймса Халлидея о начале работы с Browserify.
- Gulp + Browserify: The Everything Post by Dan Tello – Интересная статья, показывающая продвинутые техники использования
- And just like that Grunt and RequireJS are out, it's all about Gulp and Browserify now – Мартин Дженей говорит о его неожиданной любви к Browserify и Gulp, с приведением примера.
- An introduction to Gulp.js – Больше информации о том, как использовать Gulp от Крейга Баклера.