Ajax и JSON

Оглавление

[Синтаксис языка 0](#_Toc444164282)

[Внешние скрипты и порядок их исполнения 1](#_Toc444164283)

[Структура кода 2](#_Toc444164284)

[Комментарии 2](#_Toc444164285)

[Переменные 3](#_Toc444164286)

[Константы 3](#_Toc444164287)

[Типы данных 4](#_Toc444164288)

[Оператор typeof 5](#_Toc444164289)

[Современный стандарт, "use strict" 6](#_Toc444164290)

[Домашнее задание 6](#_Toc444164291)

[Используемая литература 7](#_Toc444164292)

# Что такое Ajax?

AJAX (аббревиатура от «Asynchronous Javascript And Xml») – технология обращения к серверу без перезагрузки страницы.

За счет этого уменьшается время отклика и веб-приложение по интерактивности больше напоминает десктоп.

Несмотря на то, что в названии технологии присутствует буква X (от слова XML), использовать XML вовсе не обязательно. Под AJAX подразумевают любое общение с сервером без перезагрузки страницы, организованное при помощи JavaScript.

# Что можно сделать с помощью Ajax?

**Элементы интерфейса**

В первую очередь AJAX полезен для форм и кнопок, связанных с элементарными действиями: добавить в корзину, подписаться, и т.п.

Сейчас – в порядке вещей, что такие действия на сайтах осуществляются без перезагрузки страницы.

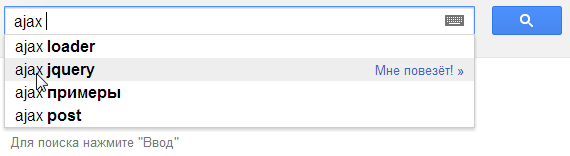
**Динамическая подгрузка данных**

Например, дерево, которое при раскрытии узла запрашивает данные у сервера.

**Живой поиск**

Живой поиск – классический пример использования AJAX, взятый на вооружение современными поисковыми системами.

Пользователь начинает печатать поисковую фразу, а JavaScript предлагает возможные варианты, получая список самых вероятных дополнений с сервера.



Код, который это обеспечивает, работает следующим образом.

1. Код активируется примерно при каждом нажатии клавиши, но не чаще чем раз в 100 мс (примерно).

2. Создается скрытый DIV и заполняется ответом сервера:

- Текущий результат подсвечен, можно перемещаться и выбирать

- При нажатии правой стрелки или при клике -- поиск в подрезультатах

3. Результаты запросов кешируются, повторных обращений к серверу не происходит.

4. В Google не только предлагаются варианты, но система тут же инициирует и сам поиск, т.е. не нужно даже нажимать `key:Enter`.

**Технически, с помощью AJAX можно обмениваться любыми данными с сервером.**

Обычно используются форматы:

* JSON – для отправки и получения структурированных данных, объектов.
* XML – если сервер почему-то работает в формате XML, то можно использовать и его, есть средства.
* HTML/текст – можно и просто загрузить с сервера код HTML или текст для показа на странице.
* Бинарные данные, файлы – гораздо реже, в современных браузерах есть удобные средства для них.

# Формат JSON

JSON (англ. JavaScript Object Notation) — текстовый формат обмена данными, основанный на JavaScript. Как и многие другие текстовые форматы, JSON легко читается людьми. Формат JSON был разработан Дугласом Крокфордом.

Несмотря на происхождение от JavaScript (точнее, от подмножества языка стандарта ECMA-262 1999 года), формат считается независимым от языка и может использоваться практически с любым языком программирования. Для многих языков существует готовый код для создания и обработки данных в формате JSON.

Это один из наиболее удобных форматов данных при взаимодействии с JavaScript. Если нужно с сервера взять объект с данными и передать на клиенте, то в качестве промежуточного формата – для передачи по сети, почти всегда используют именно его.

JSON-текст представляет собой (в закодированном виде) одну из двух структур:

* Набор пар ключ: значение. В различных языках это реализовано как объект, запись, структура, словарь, хэш-таблица, список с ключом или ассоциативный массив. Ключом может быть только строка (регистрозависимая: имена с буквами в разных регистрах считаются разными[3]), значением — любая форма.
* Упорядоченный набор значений. Во многих языках это реализовано как массив, вектор, список или последовательность.

Это универсальные структуры данных: как правило, любой современный язык программирования поддерживает их в той или иной форме. Они легли в основу JSON, так как он используется для обмена данными между различными языками программирования.

В качестве значений в JSON могут быть использованы:

* Объект — это неупорядоченное множество пар ключ:значение, заключённое в фигурные скобки «{ }». Ключ описывается строкой, между ним и значением стоит символ «:». Пары ключ-значение отделяются друг от друга запятыми.
* Массив (одномерный) — это упорядоченное множество значений. Массив заключается в квадратные скобки «[ ]». Значения разделяются запятыми.
* Число.
* Литералы true, false и null.
* Строка — это упорядоченное множество из нуля или более символов юникода, заключенное в двойные кавычки. Символы могут быть указаны с использованием escape-последовательностей, начинающихся с обратной косой черты «\» (поддерживаются варианты \", \\, \/, \t, \n, \r, \f и \b), или записаны шестнадцатеричным кодом в кодировке UTF-8 в виде \uFFFF.

Следующий пример показывает JSON-представление объекта, описывающего человека. В объекте есть строковые поля имени и фамилии, объект, описывающий адрес, и массив, содержащий список телефонов. Как видно из примера, значение может представлять собой вложенную структуру.

{

"firstName": "Иван",

"lastName": "Иванов",

"address": {

"streetAddress": "Московское ш., 101, кв.101",

"city": "Ленинград",

"postalCode": 101101

},

"phoneNumbers": [

"812 123-1234",

"916 123-4567"

]

}

На языке XML подобная структура выглядела бы примерно так:

**<person>**

**<firstName>**Иван**</firstName>**

**<lastName>**Иванов**</lastName>**

**<address>**

**<streetAddress>**Московское ш., 101, кв.101**</streetAddress>**

**<city>**Ленинград**</city>**

**<postalCode>**101101**</postalCode>**

**</address>**

**<phoneNumbers>**

**<phoneNumber>**812 123-1234**</phoneNumber>**

**<phoneNumber>**916 123-4567**</phoneNumber>**

**</phoneNumbers>**

**</person>**

В современных браузерах есть замечательные методы, знание тонкостей которых делает операции с JSON простыми и комфортными.

Данные в формате JSON (RFC 4627) представляют собой:

* JavaScript-объекты { ... } или
* Массивы [ ... ] или
* Значения одного из типов:
  + строки в двойных кавычках,
  + число,
* логическое значение true/false,
* null.

Почти все языки программирования имеют библиотеки для преобразования объектов в формат JSON.

Основные методы для работы с JSON в JavaScript – это:

* JSON.parse – читает объекты из строки в формате JSON.
* JSON.stringify – превращает объекты в строку в формате JSON, используется, когда нужно из JavaScript передать данные по сети.

**Метод JSON.parse**

Вызов JSON.parse(str) превратит строку с данными в формате JSON в JavaScript-объект/массив/значение.

Например:

var numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] ); // 1

Или так:

var user = '{ "name": "Вася", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] ); // 1

Данные могут быть сколь угодно сложными, объекты и массивы могут включать в себя другие объекты и массивы. Главное чтобы они соответствовали формату.

Объекты в формате JSON похожи на обычные JavaScript-объекты, но отличаются от них более строгими требованиями к строкам – они должны быть именно в двойных кавычках.

В частности, первые два свойства объекта ниже – некорректны:

{

name: "Вася", // ошибка: ключ name без кавычек!

"surname": 'Петров',// ошибка: одинарные кавычки у значения 'Петров'!

"age": 35, // .. а тут всё в порядке.

"isAdmin": false // и тут тоже всё ок

}

Кроме того, в формате JSON не поддерживаются комментарии. Он предназначен только для передачи данных.

Есть нестандартное расширение формата JSON, которое называется JSON5 и как раз разрешает ключи без кавычек, комментарии и т.п, как в обычном JavaScript. На данном этапе, это отдельная библиотека.

**Умный разбор: JSON.parse(str, reviver)**

Метод JSON.parse поддерживает и более сложные алгоритмы разбора.

Например, мы получили с сервера объект с данными события event.

Он выглядит так:

// title: название события, date: дата события

var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';

…И теперь нужно восстановить его, то есть превратить в JavaScript-объект.

Попробуем вызвать для этого JSON.parse:

var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';

var event = JSON.parse(str);

alert( event.date.getDate() ); // ошибка!

…Увы, ошибка!

Дело в том, что значением event.date является строка, а отнюдь не объект Date. Откуда методу JSON.parse знать, что нужно превратить строку именно в дату?

Для интеллектуального восстановления из строки у JSON.parse(str, reviver) есть второй параметр reviver, который является функцией function(key, value).

Если она указана, то в процессе чтения объекта из строки JSON.parse передаёт ей по очереди все создаваемые пары ключ-значение и может возвратить либо преобразованное значение, либо undefined, если его нужно пропустить.

В данном случае мы можем создать правило, что ключ date всегда означает дату:

// дата в строке - в формате UTC

var str = '{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}';

var event = JSON.parse(str, function(key, value) {

if (key == 'date') return new Date(value);

return value;

});

alert( event.date.getDate() ); // теперь сработает!

Кстати, эта возможность работает и для вложенных объектов тоже:

var schedule = '{ \

"events": [ \

{"title":"Конференция","date":"2014-11-30T12:00:00.000Z"}, \

{"title":"День рождения","date":"2015-04-18T12:00:00.000Z"} \

]\

}';

schedule = JSON.parse(schedule, function(key, value) {

if (key == 'date') return new Date(value);

return value;

});

alert( schedule.events[1].date.getDate() ); // сработает!

**Сериализация, метод JSON.stringify**

Метод JSON.stringify(value, replacer, space) преобразует («сериализует») значение в JSON-строку.

Пример использования:

var event = {

title: "Конференция",

date: "сегодня"

};

var str = JSON.stringify(event);

alert( str ); // {"title":"Конференция","date":"сегодня"}

// Обратное преобразование.

event = JSON.parse(str);

При сериализации объекта вызывается его метод toJSON.

Если такого метода нет – перечисляются его свойства, кроме функций.

Посмотрим это в примере посложнее:

var room = {

number: 23,

occupy: function() {

alert( this.number );

}

};

event = {

title: "Конференция",

date: new Date(Date.UTC(2014, 0, 1)),

room: room

};

alert( JSON.stringify(event) );

/\*

{

"title":"Конференция",

"date":"2014-01-01T00:00:00.000Z", // (1)

"room": {"number":23} // (2)

}

\*/

Обратим внимание на два момента:

1. Дата превратилась в строку. Это не случайно: у всех дат есть встроенный метод toJSON. Его результат в данном случае – строка в таймзоне UTC.
2. У объекта room нет метода toJSON. Поэтому он сериализуется перечислением свойств.

Мы, конечно, могли бы добавить такой метод, тогда в итог попал бы его результат:

var room = {

number: 23,

toJSON: function() {

return this.number;

}

};

alert( JSON.stringify(room) ); // 23

**Исключение свойств**

Попытаемся преобразовать в JSON объект, содержащий ссылку на DOM.

Например:

var user = {

name: "Вася",

age: 25,

window: window

};

alert( JSON.stringify(user) ); // ошибка!

// TypeError: Converting circular structure to JSON (текст из Chrome)

Произошла ошибка! В чём же дело, неужели некоторые объекты запрещены? Как видно из текста ошибки – дело совсем в другом. Глобальный объект window – сложная структура с кучей встроенных свойств и круговыми ссылками, поэтому его преобразовать невозможно. Да и нужно ли?

Во втором параметре JSON.stringify(value, replacer) можно указать массив свойств, которые подлежат сериализации.

Например:

var user = {

name: "Вася",

age: 25,

window: window

};

alert( JSON.stringify(user, ["name", "age"]) );

// {"name":"Вася","age":25}

Для более сложных ситуаций вторым параметром можно передать функцию function(key, value), которая возвращает сериализованное value либо undefined, если его не нужно включать в результат:

var user = {

name: "Вася",

age: 25,

window: window

};

var str = JSON.stringify(user, function(key, value) {

if (key == 'window') return undefined;

return value;

});

alert( str ); // {"name":"Вася","age":25}

В примере выше функция пропустит свойство с названием window. Для остальных она просто возвращает значение, передавая его стандартному алгоритму. А могла бы и как-то обработать.

**Красивое форматирование**

В методе JSON.stringify(value, replacer, space) есть ещё третий параметр space.

Если он является числом – то уровни вложенности в JSON оформляются указанным количеством пробелов, если строкой – вставляется эта строка.

Например:

var user = {

name: "Вася",

age: 25,

roles: {

isAdmin: false,

isEditor: true

}

};

var str = JSON.stringify(user, "", 4);

alert( str );

/\* Результат -- красиво сериализованный объект:

{

"name": "Вася",

"age": 25,

"roles": {

"isAdmin": false,

"isEditor": true

}

}

\*/

# Объект XMLHttpRequest

Объект XMLHttpRequest (или, как его кратко называют, «XHR») дает возможность из JavaScript делать HTTP-запросы к серверу без перезагрузки страницы.

Несмотря на слово «XML» в названии, XMLHttpRequest может работать с любыми данными, а не только с XML.

Использовать его очень просто.

Как правило, XMLHttpRequest используют для загрузки данных.

Для начала посмотрим на пример использования, который загружает файл phones.json из текущей директории и выдаёт его содержимое:

// 1. Создаём новый объект XMLHttpRequest

var xhr = new XMLHttpRequest();

// 2. Конфигурируем его: GET-запрос на URL 'phones.json'

xhr.open('GET', 'phones.json', false);

// 3. Отсылаем запрос

xhr.send();

// 4. Если код ответа сервера не 200, то это ошибка

if (xhr.status != 200) {

// обработать ошибку

alert( xhr.status + ': ' + xhr.statusText ); // пример вывода: 404: Not Found

} else {

// вывести результат

alert( xhr.responseText ); // responseText -- текст ответа.

}

Далее мы более подробно разберём основные методы и свойства объекта XMLHttpRequest, в том числе те, которые были использованы в этом коде.

**Настроить: open**

Синтаксис:

xhr.open(method, URL, async, user, password)

Этот метод – как правило, вызывается первым после создания объекта XMLHttpRequest.

Задаёт основные параметры запроса:

* method – HTTP-метод. Как правило, используется GET либо POST, хотя доступны и более экзотические, вроде TRACE/DELETE/PUT и т.п.
* URL – адрес запроса. Можно использовать не только http/https, но и другие протоколы, напримерftp:// и file://. При этом есть ограничения безопасности, называемые «Same Origin Policy»: запрос со страницы можно отправлять только на тот же протокол://домен:порт, с которого она пришла. В следующих главах мы рассмотрим, как их можно обойти.
* async – если установлено в false, то запрос производится синхронно, если true – асинхронно.

«Синхронный запрос» означает, что после вызова xhr.send() и до ответа сервера главный поток будет «заморожен»: посетитель не сможет взаимодействовать со страницей – прокручивать, нажимать на кнопки и т.п. После получения ответа выполнение продолжится со следующей строки.

«Асинхронный запрос» означает, что браузер отправит запрос, а далее результат нужно будет получить через обработчики событий, которые мы рассмотрим далее.

* user, password – логин и пароль для HTTP-авторизации, если нужны.

Заметим, что вызов open, в противоположность своему названию (open – англ. «открыть») не открывает соединение. Он лишь настраивает запрос, а коммуникация инициируется методом send.

**Отослать данные: send**

Синтаксис:

xhr.send([body])

Именно этод метод открывает соединение и отправляет запрос на сервер.

В body находится тело запроса. Не у всякого запроса есть тело, например у GET-запросов тела нет, а у POST – основные данные как раз передаются через body.

**Отмена: abort**

Вызов xhr.abort() прерывает выполнение запроса.

**Ответ: status, statusText, responseText**

Основные свойства, содержащие ответ сервера:

**status**

HTTP-код ответа: 200, 404, 403 и так далее. Может быть также равен 0, если сервер не ответил или при запросе на другой домен.

**statusText**

Текстовое описание статуса от сервера: OK, Not Found, Forbidden и так далее.

**responseText**

Текст ответа сервера.

Есть и ещё одно свойство, которое используется гораздо реже:

**responseXML**

Если сервер вернул XML, снабдив его правильным заголовком Content-type: text/xml, то браузер создаст из него XML-документ. По нему можно будет делать запросыxhr.responseXml.querySelector("...") и другие.

Оно используется редко, так как обычно используют не XML, а JSON. То есть, сервер возвращает JSON в виде текста, который браузер превращает в объект вызовомJSON.parse(xhr.responseText).

# Синхронные и асинхронные запросы

Если в методе open установить параметр async равным false, то запрос будет синхронным.

Синхронные вызовы используются чрезвычайно редко, так как блокируют взаимодействие со страницей до окончания загрузки. Посетитель не может даже прокручивать её. Никакой JavaScript не может быть выполнен, пока синхронный вызов не завершён – в общем, в точности те же ограничения как alert.

// Синхронный запрос

xhr.open('GET', 'phones.json', false);

// Отсылаем его

xhr.send();

// ...весь JavaScript "подвиснет", пока запрос не завершится

Если синхронный вызов занял слишком много времени, то браузер предложит закрыть «зависшую» страницу.

Из-за такой блокировки получается, что нельзя отослать два запроса одновременно. Кроме того, забегая вперёд, заметим, что ряд продвинутых возможностей, таких как возможность делать запросы на другой домен и указывать таймаут, в синхронном режиме не работают.

Из всего вышесказанного уже должно быть понятно, что синхронные запросы используются чрезвычайно редко, а асинхронные – почти всегда.

Для того, чтобы запрос стал асинхронным, укажем параметр async равным true.

Изменённый JS-код:

var xhr = new XMLHttpRequest();

xhr.open('GET', 'phones.json', true);

xhr.send(); // (1)

xhr.onreadystatechange = function() { // (3)

if (xhr.readyState != 4) return;

button.innerHTML = 'Готово!';

if (xhr.status != 200) {

alert(xhr.status + ': ' + xhr.statusText);

} else {

alert(xhr.responseText);

}

}

button.innerHTML = 'Загружаю...'; // (2)

button.disabled = true;

Если в open указан третий аргумент true (или если третьего аргумента нет), то запрос выполняется асинхронно. Это означает, что после вызова xhr.send() в строке (1) код не «зависает», а преспокойно продолжает выполняться, выполняется строка (2), а результат приходит через событие (3), мы изучим его чуть позже.

**Событие readystatechange**

Событие readystatechange происходит несколько раз в процессе отсылки и получения ответа. При этом можно посмотреть «текущее состояние запроса» в свойстве xhr.readyState.

В примере выше мы использовали только состояние 4 (запрос завершён), но есть и другие.

Все состояния, по спецификации:

const unsigned short UNSENT = 0; // начальное состояние

const unsigned short OPENED = 1; // вызван open

const unsigned short HEADERS\_RECEIVED = 2; // получены заголовки

const unsigned short LOADING = 3; // загружается тело (получен очередной пакет данных)

const unsigned short DONE = 4; // запрос завершён

Запрос проходит их в порядке 0 → 1 → 2 → 3 → … → 3 → 4, состояние 3 повторяется при каждом получении очередного пакета данных по сети.

**HTTP-заголовки**

XMLHttpRequest умеет как указывать свои заголовки в запросе, так и читать присланные в ответ.

Для работы с HTTP-заголовками есть 3 метода:

**setRequestHeader(name, value)**

Устанавливает заголовок name запроса со значением value.

Например:

xhr.setRequestHeader('Content-Type', 'application/json');

**getResponseHeader(name)**

Возвращает значение заголовка ответа name, кроме Set-Cookie и Set-Cookie2.

Например:

xhr.getResponseHeader('Content-Type')

**getAllResponseHeaders()**

Возвращает все заголовки ответа, кроме Set-Cookie и Set-Cookie2.

Заголовки возвращаются в виде единой строки, например:

Cache-Control: max-age=31536000

Content-Length: 4260

Content-Type: image/png

Date: Sat, 08 Sep 2012 16:53:16 GMT

Между заголовками стоит перевод строки в два символа "\r\n" (не зависит от ОС), значение заголовка отделено двоеточием с пробелом ": ". Этот формат задан стандартом.

Таким образом, если хочется получить объект с парами заголовок-значение, то эту строку необходимо разбить и обработать.

**Таймаут**

Максимальную продолжительность асинхронного запроса можно задать свойством timeout:

xhr.timeout = 30000; // 30 секунд (в миллисекундах)

При превышении этого времени запрос будет оборван и сгенерировано событие ontimeout:

xhr.ontimeout = function() {

alert( 'Извините, запрос превысил максимальное время' );

}

# Домашнее задание

* Создайте код, который загрузит файл phones.json из текущей директории и выведет все названия телефонов из него в виде списка.
* Добавить вывод фигур в шахматную доску из предыдущего урока.
* Написать инициализацию расположения фигур с помощью ajax-запроса к json-файлу. Формат данного файла придумать и осуществить самостоятельно.
* \* Добавить возможность отсутствия фигур (например, если они были срублены) и предусмотреть проверку корректности данных (например, что на доске сейчас находится не больше двух королей и тп.), в случае, если данные некорректны – выдать человекопонятную ошибку.

# Используемая литература

* [Mozilla Developer Network](https://developer.mozilla.org/)
* [MSDN](http://msdn.microsoft.com/)
* [Safari Developer Library](https://developer.apple.com/library/safari/navigation/index.html)
* [Современный учебник JavaScript](http://learn.javascript.ru)