# Динамическое клиент-серверное программирование, AJAX

В этой статье AJAX описывается на уровне возможностей и примеров. Рассмотрены особенности асинхронного взаимодействия и примеры использования, но с минимумом технических деталей.

Надеюсь, она будет полезна для понимания, что такое AJAX и с чем его едят.

**Что такое AJAX ? Пример реализации.**

AJAX, или, более длинно, Asynchronous Javascript And Xml - технология для взаимодействия с сервером без перезагрузки страниц.

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

Например, при нажатии кнопки голосовать - из браузера на сервер будет отправлено сообщение, а сервер ответит браузеру, что голос принят.

Здесь будет ответ сервера

Технология AJAX, как указывает первая буква A в ее названии - асинхронна, т.е браузер, отослав запрос, может делать что угодно, например, показать сообщение  
об ожидании ответа, прокручивать страницу, и т.п.

Вот код кнопки в примере выше:

<input value="Голосовать!" onclick="vote()" type="button" />

При нажатии она вызывает функцию vote, которая отправляет запрос на сервер, ждет ответа, а затем показывает сообщение об этом в div'е под кнопкой:

<div id="vote\_status">Здесь будет ответ сервера</div>

Далее мы разберем, как она работает, более подробно.

Для обмена данными с сервером используется специальный объект XmlHttpRequest, который умеет отправлять запрос и получать ответ с сервера. Кроссбраузерно создать такой объект можно, например, так:

function getXmlHttp(){

var xmlhttp;

try {

xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");

} catch (e) {

try {

xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");

} catch (E) {

xmlhttp = false;

}

}

if (!xmlhttp && typeof XMLHttpRequest!='undefined') {

xmlhttp = new XMLHttpRequest();

}

return xmlhttp;

}

Более подробно о деталях реализации AJAX с использованием XmlHttpRequest и других транспортов можно почитать в [разделе про общение с сервером](http://javascript.ru/ajax/transport).

Здесь мы не будем на этом останавливаться и перейдем сразу к функции vote:

// javascript-код голосования из примера

function vote() {

// (1) создать объект для запроса к серверу

var req = getXmlHttp()

// (2)

// span рядом с кнопкой

// в нем будем отображать ход выполнения

var statusElem = document.getElementById('vote\_status')

req.onreadystatechange = function() {

// onreadystatechange активируется при получении ответа сервера

if (req.readyState == 4) {

// если запрос закончил выполняться

statusElem.innerHTML = req.statusText // показать статус (Not Found, ОК..)

if(req.status == 200) {

// если статус 200 (ОК) - выдать ответ пользователю

alert("Ответ сервера: "+req.responseText);

}

// тут можно добавить else с обработкой ошибок запроса

}

}

// (3) задать адрес подключения

req.open('GET', '/ajax\_intro/vote.php', true);

// объект запроса подготовлен: указан адрес и создана функция onreadystatechange

// для обработки ответа сервера

// (4)

req.send(null); // отослать запрос

// (5)

statusElem.innerHTML = 'Ожидаю ответа сервера...'

}

Поток выполнения, использованный vote, довольно типичен и выглядит так:

1. Функция создает объект XmlHttpRequest
2. назначает обработчик ответа сервера onreadystatechange
3. открывает соединение open
4. отправляет запрос вызовом send (ответ сервера принимается срабатывающей в асинхронном режиме функцией onreadystatechange)
5. показывает посетителю индикатор состояния процесса

Серверный обработчик, к которому обращен AJAX-запрос (в примере это vote.php) по сути ничем не отличается от обычной страницы. AJAX-запрос, отправляемый XmlHttpRequest, ничем не отличается от обычного запроса.

Просто текст, который возвращает сервер, не показывается как HTML, а читается и обрабатывается функцией onreadystatechange.

Пример: vote.php для примера с голосованием

<?php

sleep(3);

echo 'Ваш голос принят!';

**Смысл AJAX - в интеграции технологий**

Технология AJAX использует комбинацию:

* (X)HTML, CSS для подачи и стилизации информации
* DOM-модель, операции над которой производятся javascript на стороне клиента, чтобы обеспечить динамическое отображение и взаимодействие с информацией
* XMLHttpRequest для асинхронного обмена данными с веб-сервером. В некоторых AJAX-фреймворках и в некоторых ситуациях, вместо XMLHttpRequest используется IFrame, SCRIPT-тег или другой аналогичный транспорт.
* JSON часто используется для обмена данными, однако любой формат подойдет, включая форматированный HTML, текст, XML и даже какой-нибудь EBML

Типичное AJAX-приложение состоит как минимум из двух частей.

Первая выполняется в браузере и написана, как правило, на JavaScript, а вторая - находится на сервере и написана, например, на Ruby, Java или PHP .

Между этими двумя частями происходит обмен данными через XMLHttpRequest(или другой [транспорт](http://javascript.ru/ajax/transport)).

**Что я могу сделать с помощью AJAX ?**

Смысл AJAX - в интерактивности и быстром времени отклика.

**Небольшие элементы управления**

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

|  |  |  |
| --- | --- | --- |
|  |  |  |

**Динамическая подгрузка данных с сервера.**

Например, дерево, узлы которого подгружаются по мере раскрытия.

Бесконечное дерево

**Незаметные для пользователя действия.**

Например, при редактировании статьи - каждые 10 минут результаты автосохраняются на сервере.

**Непрерывная подзагрузка информации с сервера.**

Самый типичный пример - чат. В окошко постоянно поступают все новые сообщения, непрерывно подгружаемые с сервера. И, опять же, через AJAX, без перезагрузки страницы, пользователь может  
отсылать сообщения на сервер.

Существуют другие предложения подобного рода, например, отображающие биржевые котировки в реальном времени.

**Пример. Google suggest.**

Google - одна из первых систем, которая предложила "живой поиск", live search. Пользователь печатает поисковую фразу, а система автодополняет ее, получая  
список самых вероятных дополнений с сервера.

Код, который это обеспечивает, работает следующим образом.

* Активируется примерно при каждом нажатии клавиши
  + Время посылки последнего запроса отслеживается
  + Для "обычной" скорости печати - запрос отсылается при каждом нажатии
  + Для "программистской" скорости - каждые несколько нажатий
* Создается скрытый DIV, который показывается при начале печати
* DIV заполняется ответом сервера
  + Текущий результат подсвечен, можно перемещаться и выбирать
  + При нажатии правой стрелки, поиск в подрезультатах
* Результаты кэшируются
  + при нажатии на "удалить", обращения к серверу не происходит
* Время на осуществление запроса отслеживается для управления частотой запросов к серверу
  + Обычный модем будет обращаться к серверу меньше,
  + Подключение по выделенной линии - запросы идут чаще.

**Пример. Gmail.**

Раз уж взялись за Google - рассмотрим почтовый сервис той же компании, <http://gmail.com>.

На момент его появления он явился единственным открытым почтовым сервисом, использующим AJAX для следующих фич.

* Проверка ошибок ввода формы ДО сабмита

На сервер передается имя пользователя, результат проверки возвращается на страницу.

* "Мгновенная" загрузка

Браузер обращается к серверу только за данными, а не обновляет весь громоздкий интерфейс

* Автоматическая "доставка" писем в открытую папку

Время от времени отправляется запрос на сервер и, если пришли новые письма, они появляются в браузере.

* Автодополнение

Достаточно ввести первые буквы имени адресата, и остальные дополняются автоматически, как в десктоп-приложениях.

Результат: обширная популярность, высокий спрос на account'ы с момента открытия.

**Синхронная модель VS Асинхронная модель**

В обычном программировании все операции носят синхронный характер, т.е выполняются одна за другой.

Условно говоря, мы действуем так:

1. закидываем удочку
2. ждем, когда клюнет
3. клюнуло - включаем подтяжку спиннинга

При асинхронном подходе мы:

1. вешаем на удило специальный детектор клева, задаем ему тянуть спиннинг, когда клюнет
2. закидываем удочку
3. занимаемся другими делами
4. детектор клева срабатывает, запуская подтяжку спиннинга

Т.е, в синхронном случае удочка постоянно приковывает наше внимание. Ловля рыбы - последовательный процесс.

В асинхронном варианте - мы сначала задали программу, что делать при клеве, а затем опустили удочку ловить и занялись другими делами.  
Например, поставили еще 5 таких удочек.

Асинхронное программирование сложнее, чем синхронное, и поначалу непривычно, т.к в нем заранее задается то, что сработает после.  
Т.е, программу "что делать, когда клюнет" нужно задать до того, как клюнуло, и вообще неизвестно, есть ли в водоеме рыба.

Существуют приемы, облегчающие асинхронное программирование, например, отложенный объект Deferred (Twisted,Dojo,Mochikit), но об этом - в отдельной статье.

**Синхронная и асинхронная модель в AJAX**

Вернемся к нашим баранам: браузеру, серверу и, скажем, базе данных.

В синхронной модели браузер отправляет запрос на сервер и висит, ждет, пока тот совершит всю необходимую работу. Сервер выполняет запросы к базе данных, заворачивает ответ в необходимый формат и выводит его. Браузер. получив ответ, вызывает функцию показа.

Все процессы выполняются последовательно, один за другим.

Сетевые задержки включены во время ожидания, обозначенное на схеме серым цветом.

Пользователь не может заниматься чем-то другим на этой же странице, пока происходит синхронный обмен данными.

В асинхронной модели запрос отсылается ("удочка поставлена"), и можно заняться чем-то другим. Когда запрос выполнился ("клюнуло") - запускается заранее  
подготовленная программистом функция ("подтянуть спиннинг") показа сообщения сервера.

Здесь сервер сразу же уведомляет браузер о том, что запрос принят в обработку и освобождает его для дальнейшей работы. Когда ответ будет готов - сервер перешлет его, и на браузере будет вызвана соответствующая функция показа, но пока этот ответ формируется и пересылается - браузер свободен.

Пользователь может написать комментарии, заполнить и отослать форму и т.п: Могут производиться новые асинхронные запросы.

Асинхронная модель характеризуется почти мгновенной реакцией на действия пользователя, так что создается впечатление удобного и быстрого приложения.

Например, в примере с голосованием выше - кнопка срабатывает сразу, хотя реальный учет голоса происходит позднее, после обработки сообщения сервером.

Из-за такого разрыва между действием и реальным результатом приложение становится гораздо более чувствительно к ошибкам.

Особенно в случае нескольких одновременных асинхронных запросов, нужно заботиться об очередности выполнения и ответа (race-conditions) и, в случае ошибки, оставлять приложение в целостном (consistent) состоянии.

**Особенности асинхронной модели**

* Сложность в реализации
  + Недостаточные возможности браузера (javascript)
  + Асинхронная модель сложнее для отладки
* Race conditions
  + Неопределена последовательность выполнения
  + Можно делать много одновременных задач ("удочек"), но задача, начатая первой, может окончиться последней.
* Реакция тут же, но неизвестно, какой будет результат. Усложнена обработка ошибок
  + Ошибок коммуникации - разрыв связи, и т.п.
  + Пользовательских ошибок - например, не хватило привилегий
* Контроль целостности (bugproof)
  + Например, редактор отправил асинхронный запрос на удаление ветки дерева. Добавление в нее нужно отключить, пока не придет ответ сервера. Если вдруг не хватило привилегий, то операция не удалась.
* Интерактивность
* Быстрый интерфейс

Плюсов всего два, зато какие! Овчинка стоит выделки.

**Асинхронный drag'n'drop.**

Иногда для асинхронных операций необходимо делать различные "финты ушами". Например, хочется сделать drag'n'drop в дереве, т.е перетаскивать статьи из одного раздела в другой мышкой, и чтобы они на сервере в базе данных меняли родителя.

Drag'n'drop - это "взял мышей объект - положил куда надо - готово". Но в асинхронной модели не может быть все прям сразу "готово".  
Надо проверить привилегии на сервере, проверить, существует ли еще объект, вдруг его удалил другой пользователь.

Надо как-то показать, что процесс пошел, но результат "ща будет..". А как? В асинхронной модели указатель мыши не может просто так зависнуть над объектом, превратившись в часики.

В таком случае применяют либо синхронные запросы к серверу - и тогда все действительно подвисает, либо оригинальный выход - положить объект, как будто он перенесен, и проинформировать анимированной иконкой об ожидании ответа.  
Если ответ отрицателен - обработчик ответа переносит объект обратно.

**Stale context, устаревший контекст**

В примере с drag'n'drop также затронута проблема "stale context" - устаревшего контекста.

Веб - многопользовательская среда. Если для навигации используется,  
скажем, дерево статей, то над ним работают много человек. Один из них может удалить ветку дерева, над которой работает другой: конфликт.

Как правило, для преодоления таких казусов используются следующие средства:

**Политика редактирования**

Это когда все знают кто чего делает и на уровне деления полномочий и личного общения такие удаления согласовывают. Вариант заведомо небезопасный, но обычно работающий.

**Локинг и/или версионный контроль**

Локинг - блокирование редактируемых документов.

Версионный контроль - каждый новый документ становится версией, так что изменения никогда не теряются. Версионность влечет за собой конфликты, когда Петя начал редактировать документ раньше Васи, а сохранил - позже. При этом в последней версии изменения Васи оказались потеряны, хотя предпоследняя (Васина) версия в системе обязательно есть.

Более подробно о локинге и версионности можно почитать, например, в документации к системе версионного контроля [Subversion](http://svnbook.red-bean.com/nightly/ru/svn.basic.vsn-models.html).

**Автообновление контекста**

Проблема устаревшего контента может быть на 99% решена при помощи мгновенного автообновления.

Браузер держит постоянное соединение с сервером (или делает время от времени корректирующие запросы) - и нужные изменения отсылаются по этому каналу.

Например, в раскрытую ветку дерева иногда подгружаются новые статьи, в открытый почтовый интерфейс - новые письма.

Вообще, проблема устаревшего контекста напрямую относится к задаче целостности данных. За конечную проверку целостности, как и при валидации форм, в любом случае несет ответственность сервер.

# [XMLHTTPRequest](http://javascript.ru/ajax/transport/xmlhttprequest)

Объект XMLHTTPRequest - низкоуровневая основа большинства AJAX-приложений. Знание его методов, свойств и особенностей не только помогает писать приложения на низком уровне с минимумом javascript-кода, но и позволяет понять, что происходит внутри фреймворков, которые, увы, часто несовершенны или тянут много лишнего.

**Основные методы**

Основные методы для посылки запросов [XMLHttpRequest](http://xmlhttprequest.ru):

* open(Method, Url, **async**)
* send(data)
* onreadystatechange

Правильная последовательность open/onreadystatechange/send:

req.open("GET", url, true)

req.onreadystatechange = handler

req.send(null)

Методы для манипуляции заголовками:

* setRequestHeader()
* getResponseHeader()
* getAllResponseHeaders()

Обрыв соединения осуществляет метод

* abort()

Ответ сервера находится в

* responseText
* responseXML
* status
* statusText

Причем responseXML заполняется только в случае, когда Content-Type с сервера имеет значение text/xml (кроме overrideMimeType-метода, но он есть только в Firefox).

Краткая сводка плюсов и минусов

* Возможна синхронная передача (хотя и подвешивает браузер)
* Полный контроль над заголовками
* Можно отсылать документы из браузера в виде файлов, составляя POST-запрос вручную
* В явном виде выдает ошибки коммуникации
* Ограничение безопасности при помощи same origin policy
* Невозможна отправка файлов, выбранных в форме
* Утечки памяти при неправильном применении в старых IE

**Пример обертки**

Пример небольшой функции getUrl, которая делает запросы:

function getXmlHttp(){

try {

return new ActiveXObject("Msxml2.XMLHTTP");

} catch (e) {

try {

return new ActiveXObject("Microsoft.XMLHTTP");

} catch (ee) {

}

}

if (typeof XMLHttpRequest!='undefined') {

return new XMLHttpRequest();

}

}

// Получить данные с url и вызывать cb - коллбэк c ответом сервера

function getUrl(http://javascript.ru/forum/url, cb) {

var xmlhttp = getXmlHttp();

// IE кэширует XMLHttpRequest запросы, так что добавляем случайный параметр к URL

// (хотя можно обойтись правильными заголовками на сервере)

xmlhttp.open("GET", url+'?r='+Math.random());

xmlhttp.onreadystatechange = function() {

if (xmlhttp.readyState == 4) {

cb(

xmlhttp.status,

xmlhttp.getAllResponseHeaders(),

xmlhttp.responseText

);

}

}

xmlhttp.send(null);

}

Более подробно о свойствах и применении XMLHTTPRequest - на <http://xmlhttprequest.ru>

**XMLHTTPRequest: описание, применение, частые проблемы**

Здесь Вы найдете полное описание объекта XMLHTTPRequest, способы использования, форматы данных и разбор частых проблем. На отдельной страничке - [спецификация объекта XMLHTTPRequest](http://xmlhttprequest.ru/w3c), согласно W3C.

Полезного чтения.

* [Введение](http://xmlhttprequest.ru/#start)
* [Использование XMLHTTPRequest](http://xmlhttprequest.ru/#use)
* [Методы объекта XMLHTTPRequest](http://xmlhttprequest.ru/#method)
* [Использование XMLHTTPRequest](http://xmlhttprequest.ru/#props)
* [GET и POST-запросы. *Кодировка*](http://xmlhttprequest.ru/#encoding)
* [Частые проблемы](http://xmlhttprequest.ru/#problem)
* [Ограничения безопасности. Кросс-доменный XMLHTTPRequest](http://xmlhttprequest.ru/#security)
* [Поддержка в библиотеках](http://xmlhttprequest.ru/#frameworks)

**Объект XMLHttpRequest**

Объект XMLHttpRequest (или, сокращенно, XHR) дает возможность браузеру делать HTTP-запросы к серверу без перезагрузки страницы.

Несмотря на слово XML в названии, XMLHttpRequest может работать с данными в любом текстовом формате, и даже c бинарными данными. Использовать его очень просто.

**Кроссбраузерное создание объекта запроса**

В зависимости от браузера, код для создания объекта может быть разный.  
Кроссбраузерная функция создания XMLHttpRequest:

function getXmlHttp(){

var xmlhttp;

try {

xmlhttp = new ActiveXObject("Msxml2.XMLHTTP");

} catch (e) {

try {

xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");

} catch (E) {

xmlhttp = false;

}

}

if (!xmlhttp && typeof XMLHttpRequest!='undefined') {

xmlhttp = new XMLHttpRequest();

}

return xmlhttp;

}

Функция тупо перебирает возможные внутренние реализации и возвращает начальный объект XMLHttpRequest. Существует и масса других рабочих кроссбраузерных функций, однако все они по сути делают то же самое.

**Использование XMLHTTPRequest**

Различают два использования XmlHttpRequest. Первое - самое простое, синхронное.

**Синхронный XMLHttpRequest**

В этом примере через XMLHTTPRequest с сервера запрашивается страница http://example.org/, и текст ответа сервера показывается через alert().

var xmlhttp = getXmlHttp()

xmlhttp.open('GET', '/xhr/test.html', false);

xmlhttp.send(null);

if(xmlhttp.status == 200) {

alert(xmlhttp.responseText);

}

Здесь сначала создается запрос, задается открытие ([open](http://xmlhttprequest.ru/" \l "method_open)) синхронного соединение с адресом /xhr/test.html и запрос отсылается с null, т.е без данных: [send(null)](http://xmlhttprequest.ru/" \l "method_send).

При синхронном запросе браузер "подвисает" и ждет на строчке 3, пока сервер не ответит на запрос. Когда ответ получен - выполняется строка 4, код ответа сравнивается с 200 (ОК), и при помощи alert печатается текст ответа сервера. Все максимально просто.

Свойство responseText получит такой же текст страницы, как браузер, если бы Вы в перешли на /xhr/test.html. Для сервера GET-запрос через XmlHttpRequest ничем не отличается от обычного перехода на страницу.

**Асинхронный XMLHttpRequest**

Этот пример делает то же самое, но асинхронно, т.е браузер не ждет выполнения запроса для продолжения скрипта. Вместо этого к свойству onreadystatechange подвешивается функция, которую запрос вызовет сам, когда получит ответ с сервера.

var xmlhttp = getXmlHttp()

xmlhttp.open('GET', '/xhr/test.html', **true**);

xmlhttp.onreadystatechange = function() {

if (xmlhttp.readyState == 4) {

if(xmlhttp.status == 200) {

alert(xmlhttp.responseText);

}

}

};

xmlhttp.send(null);

Асинхронность включается третьим параметром функции open. В отличие от синхронного запроса, функция send() не останавливает выполнение скрипта, а просто отправляет запрос.

Запрос xmlhttp регулярно отчитывается о своем состоянии через вызов функции xmlhttp.onreadystatechange. Состояние под номером 4 означает конец выполнения, поэтому функция-обработчик при каждом вызове проверяет - не настало ли это состояние.

Вообще, список состояний readyState такой:

* 0 - *Unitialized*
* 1 - *Loading*
* 2 - *Loaded*
* 3 - *Interactive*
* 4 - *Complete*

Состояния 0-2 вообще не используются.

Вызов функции с состоянием *Interactive* в теории должен происходить каждый раз при получении очередной порции данных от сервера. Это могло бы быть удобным для обработки ответа по частям, но Internet Explorer не дает доступа к уже полученной части ответа.   
Firefox дает такой доступ, но для обработки запроса по частям состояние *Interactive* все равно неудобно из-за сложностей обнаружения ошибок соединения. Поэтому *Interactive* тоже не используется.

На практике используется только последнее, *Complete*.

Если хотите углубиться в тонкости багов браузеров c readyState, отличными от 4, то многие из них рассмотрены в статье на [Quirksmode (англ.)](http://www.quirksmode.org/blog/archives/2005/09/xmlhttp_notes_r_2.html).

**Не используйте синхронные запросы**

Синхронные запросы применяются только в крайнем случае, когда кровь из носу необходимо дождаться ответа сервера до продолжения скрипта. В 999 случаях из 1000 можно использовать асинхронные запросы. При этом общий алгоритм такой:

1. Делаем асинхронный запрос
2. Рисуем анимированную картинку или просто запись типа "Loading..."
3. В onreadystatechange при достижении состояния 4 убираем Loading и, в зависимости от status вызываем обработку ответа или ошибки.

Кроме того, иногда полезно ставить ограничение на время запроса. Например, хочется генерировать ошибку, если запрос висит более 10 секунд.

Для этого сразу после send() через setTimeout ставится вызов обработчика ошибки, который очищается при получении ответа и обрывает запрос с генерацией ошибки, если истекли 10 секунд.

Таймаут на синхронный запрос ставить нельзя, браузер может висеть долго-долго.. А вот на асинхронный - пожалуйста.

Этот пример демонстрирует такой таймаут.

var xmlhttp = getXmlHttp()

xmlhttp.open("POST", "/someurl", true);

xmlhttp.onreadystatechange=function(){

if (xmlhttp.readyState != 4) return

clearTimeout(timeout) // очистить таймаут при наступлении readyState 4

if (xmlhttp.status == 200) {

// Все ок

...

alert(xmlhttp.responseText);

...

} else {

handleError(xmlhttp.statusText) // вызвать обработчик ошибки с текстом ответа

}

}

xmlhttp.send("a=5&b=4");

// Таймаут 10 секунд

var timeout = setTimeout( function(){ xmlhttp.abort(); handleError("Time over") }, 10000);

function handleError(message) {

// обработчик ошибки

...

alert("Ошибка: "+message)

...

}

**Методы объекта XMLHttpRequest**

**open()**

Варианты вызова:

* open( method, URL )
* open( method, URL, async )
* open( method, URL, async, userName )
* open( method, URL, async, userName, password )

Первый параметр **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, например FTP и FILE://. При этом есть ограничения безопасности, так называемая "same origin policy": запрос со страницы можно отправлять только на тот домен и порт, с которого она пришла.

Ниже это ограничение и способы обхода будут рассмотрены подробнее.

**async** = true задает асинхронные запросы, эта тема была поднята выше.

**userName**, **password** - данные для HTTP-авторизации.

**send()**

Отсылает запрос. Аргумент - *тело* запроса. Например, GET-запроса тела нет, поэтому используется send(null), а для POST-запросов тело содержит параметры запроса.

**abort()**

Вызов этого метода xmlhttp.abort() обрывает текущий запрос.

Здесь есть одно НО для браузера Internet Explorer. Успешный вызов abort() на самом деле может не обрывать соединение, а оставлять его в подвешенном состоянии на некоторый таймаут (20-30 секунд). Отловить такие повисшие соединения можно через прокси для отладки, например, Fiddler.

У браузера есть лимит: не более 2 одновременных соединений с одним доменом-портом. Т.е, если два соединения уже висят (и отвиснут по таймауту), то третье открыто не будет, пока одно из них не умрет. Надеюсь, Вы с такой проблемой не столкнетесь. Ее можно обойти использованием кросс-доменных XmlHttpRequest.

**setRequestHeader(name, value)**

Устанавливает заголовок **name** запроса со значением **value**. Если заголовок с таким **name** уже есть - он заменяется. Например,

xmlhttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')

**getAllResponseHeaders()**

Возвращает строку со всеми HTTP-заголовками ответа сервера.

**getResponseHeader(headerName)**

Возвращает значение заголовка ответа сервера с именем **headerName**.

**Свойства объекта XMLHttpRequest**

**onreadystatechange**

Ссылается на функцию-обработчик состояний запроса. В некоторых браузерах функция имеет аргумент - событие. Не используйте его, он совершенно лишний.

**readyState**

Номер состояния запроса от 0 до 4. Используйте только 4 ("completed").

**responseText**

Текст ответа сервера. Полный текст есть только при readyState=4, ряд браузеров дают доступ к полученной части ответа сервера при readyState=3.

**responseXML**

Ответ сервера в виде XML, при readyState=4.

Это свойство хранит объект типа XML document, с которым можно обращаться так же, как с обычным document. Например,

var authorElem = xmlhttp.responseXML.getElementById('author')

Чтобы браузер распарсил ответ сервера в свойство responseXML, в ответе должен быть заголовок Content-Type: text/xml.   
Иначе свойство responseXML будет равно null.

**status**

Для HTTP-запросов - статусный код ответа сервера: 200 - OK, 404 - Not Found, и т.п. Браузер Internet Explorer может также присвоить status код ошибки WinInet, например 12029 для ошибки "cannot connect".

Запросы по протоколам FTP, FILE:// не возвращают статуса, поэтому нормальным для них является status=0.

**statusText**

Текстовая расшифровка status, например "Not Found" или "OK".

**GET и POST-запросы. Кодировка.**

Во время обычного submit'а формы браузер сам кодирует значения полей и составляет тело GET/POST-запроса для посылки на сервер. При работе через XmlHttpRequest, это нужно делать самим, в javascript-коде. Большинство проблем и вопросов здесь связано с непониманием, где и какое кодирование нужно осуществлять.

Вначале рассмотрим общее кодирование запросов, ниже - правильную работу с русским языком для windows-1251.

Существуют два вида кодирования HTTP-запроса. Основной - urlencoded, он же - стандартное кодирование URL. Пробел представляется как %20, русские буквы и большинство спецсимволов кодируются, английские буквы и дефис оставляются как есть.

Способ, которым следует кодировать данные формы при submit'е, задается в ее HTML-таге:

<form method="get"> // метод GET с кодировкой по умолчанию

<form method="post" enctype="application/x-www-form-urlencoded"> // enctype явно задает кодировку

<form method="post"> // метод POST с кодировкой по умолчанию (urlencoded, как и предыдущая форма)

Если форма submit'ится обычным образом, то браузер сам кодирует (urlencode) название и значение каждого поля данных (input и т.п.) и отсылает форму на сервер в закодированном виде.

Формируя XmlHttpRequest, мы должны формировать запрос "руками", кодируя поля функцией encodeURIComponent.

Конечно, пропускать через encodeURIComponent стоит только те переменные, в которых могут быть спецсимволы или не английские буквы, т.е которые и будут как раз закодированы.

Например, для посылки GET-запроса с произвольными параметрами name и surname, их необходимо закодировать вот так:

// Пример с GET

...

var params = 'name=' + encodeURIComponent(name) + '&surname=' + encodeURIComponent(surname)

xmlhttp.open("GET", '/script.html?'+params, true)

...

xmlhttp.send(null)

В методе POST параметры передаются не в URL, а в теле, посылаемом через send(). Поэтому params нужно указывать не в адресе, а при вызове send()

Кроме того, при POST **обязателен** заголовок Content-Type, содержащий кодировку. Это указание для сервера - как обрабатывать (раскодировать) пришедший запрос.

// Пример с POST

...

var params = 'name=' + encodeURIComponent(name) + '&surname=' + encodeURIComponent(surname)

xmlhttp.open("POST", '/script.html', true)

xmlhttp.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')

...

xmlhttp.send(params)

Заголовки Content-Length, Connection в POST-запросах, хотя их и содержат некоторые "руководства", обычно не нужны. Используйте их, только если Вы действительно знаете, что делаете.

**Запросы multipart/form-data**

Второй способ кодирования - это отсутствие кодирования. Например, кодировать не нужно для пересылки файлов. Он указывается в форме (только для POST) так:

<form method="post" enctype="multipart/form-data">

В этом случае при отправке данных на сервер ничего не кодируется. А сервер, со своей стороны, посмотрев на Content-Type(=multipart/form-data), поймет, что пришло.

Возможности XmlHttpRequest позволяют создать запрос с любым телом. Например, можно вручную сделать POST-запрос, загружающий на сервер файл. Функционал создания таких запросов есть, в частности, во фреймворке [dojo](http://dojotoolkit.org). Но можно реализовать его и самому, прочитав о нужном формате тела POST и заголовках.

**Кодировка (языковая)**

Если Вы используете только UTF-8 - пропустите эту секцию.

Все идущие на сервер параметры GET/POST, кроме случая multipart/form-data, кодируются в UTF-8. Не в кодировке страницы, а именно в UTF-8. Поэтому, например, в PHP их нужно при необходимости перекодировать функцией iconv.

// ajax.php

$name = iconv('UTF8','CP1251',$\_GET['name']);

С другой стороны, ответ с сервера браузер воспринимает именно в той кодировке, которая указана в заголовке ответа Content-Type. Т.е, опять же, в PHP, чтобы браузер воспринял ответ в windows-1251 и нормально отобразил данные на странице в windows-1251, нужно послать заголовок с кодировкой в php-коде, например так:

// ajax.php

header('Content-Type: text/plain; charset=windows-1251');

Или же, такой заголовок должен добавить сервер. Например, в apache автоматически добавляется кодировка опцией:

# в конфиге апача

AddDefaultCharset windows-1251

**Частые проблемы**

**Кеширование**

Многие браузеры поддерживают кеширование ответов на XmlHttpRequest запросы. При этом реализации кеширования немного разные.

Например, при повторном XmlHttpRequest на тот же URL, Firefox посылает запрос с заголовком "If-Modified-Since" со значением, указанным в заголовке "Last-Modified" предыдущего ответа.

А Internet Explorer делает так, только когда кешированный ответ устарел, т.е после времени из заголовка "Expires" предыдущего ответа. Поэтому, кстати, многие думают, что Internet Explorer вообще не очищает кеш ответов.

Самое простое решение проблемы - просто убрать кеширование. Например, при помощи заголовков, или добавлением случайного параметра в URL типа:

xmlhttp.open("GET", "/service.php?r="+Math.random(), true)

Есть, однако, ряд случаев, когда кеширование XMLHttpRequest браузером полезно, улучшает время ответа и экономит трафик, просто нужно уметь его использовать.

Пример демонстрирует универсальный код работы с кешем для Internet Explorer и Firefox. Этот пример обеспечивает посылку "If-Modified-Sinse"-заголовка IE при обращениях к закешированному запросу.

var xmlhttp = getXmlHttp()

xmlhttp.open("GET", uri, false); // синхронный запрос для примера

xmlhttp.send(null);

if(!xmlhttp.getResponseHeader("Date")) { // 1

var cached = xmlhttp;

xmlhttp = getXmlHttp()

var ifModifiedSince = cached.getResponseHeader("Last-Modified");

ifModifiedSince = (ifModifiedSince) ? ifModifiedSince : new Date(0); // January 1, 1970

xmlhttp.open("GET", uri, false);

xmlhttp.setRequestHeader("If-Modified-Since", ifModifiedSince);

xmlhttp.send(null);

if(xmlhttp.status == 304) {

xmlhttp = cached;

}

}

**Разбор примера работы с кешем**

Внешний тест (1) опирается на то, что в Internet Explorer, если запрос возвращается из кеша без перепроверки, заголовок Date - пустая строка. Поэтому при этом нужно сделать дополнительный запрос, который как раз и будет реальным запросом к серверу.

Когда делаем дополнительный запрос, что ссылку на кешированый запрос сохраняем, т.к если код ответа дополнительного запроса - "304 Not Modified", то его тело будет пустой строкой, и нужно будет вернуться к кешированному объекту.

Для оптимизации, можно не создавать новый объект XmlHttpRequest, а сохранить данные из существующего и использовать заново его же.

Пример выше опирается на то, что сервер всегда выдает заголовок "Date", что верно для большинства конфигураций. В нем делается синхронный запрос. В асинхронном случае, проверку на Date и т.д нужно делать после получения ответа в функции-обработчике onreadystate.

**Повторное использование объекта XmlHttpRequest**

В Internet Explorer, если open() вызван после установки onreadystatechange, может быть проблема с повторным использованием этого XmlHttpRequest.

Чтобы использовать заново XmlHttpRequest, сначала вызывайте метод open(), а затем - присваивайте onreadystatechange. Это нужно из-за того, что IE самостоятельно очищает объект XmlHttpRequest в методе open(), если его статус "completed".

Вызывать abort() для перенаправления запроса на другой URL не нужно, даже если текущий запрос еще не завершился.

**Повторный XmlHttp-запрос после abort() зависает**

С этой проблемой я сталкивался только в IE под Windows. Ее причины - в том, что abort() не обрывает TCP-соединение, а оставляет его висеть до наступления таймаута (см. [метод abort()](http://xmlhttprequest.ru/" \l "method_abort)). Если же к домену есть два TCP-соединения (даже ждущие таймаута), то третье будет висеть, пока какое-то из них не помрет.

**XmlHttpRequest виснет в IE7 (много табов)**

Проблема иногда возникает при отладке приложений с длинными XmlHttpRequest, которые висят и ждут события с сервера.

Она связана с ограничением в 2 одновременных соединения к одному домену. Точнее, с тем фактом, что это ограничение в IE7 действует не на один таб, а на все. Так что, если есть два таба с непрерывным соединением, то при открытии третьего таба - XmlHttpRequest с него к тому же домену просто зависнет и будет ждать окончания одного из двух предыдущих запросов.

**Утечки памяти**

В Internet Explorer объект XmlHttpRequest принадлежит миру DOM/COM, а Javascript-функция - миру Javascript. Присваивание xmlhttp.onreadystatechange = function() { ... } задает неявную круговую связь: xmlhttp ссылается на функцию через onreadystatechange, а функция, через свою область видимости - видит (ссылается на) xmlhttp.

Невозможность обнаружить и оборвать такую связь во многих (до IE 6,7 редакции июня 2007?) версиях Internet Explorer приводит к тому, что XmlHttpRequest вместе с ответом сервера, функция-обработчик и всё замыкание прочно оседают в памяти до перезагрузки браузера.

Чтобы этого избежать, ряд фреймворков (YUI, dojo...) вообще не ставят onreadystatechange, а вместо этого через setTimeout проверяют его readyState каждые 10 миллисекунд. Это разрывает круговую связку xmlhttp <-> onreadystatechange, и утечка памяти не грозит даже в самых глючных браузерах.

**Firefox ставит responseXML вида <parseerror>...</parseerror>**

Да, у браузеров типа Mozilla это такой способ сказать, что документ невалидный.

**Ограничения безопасности. Кросс-доменный XMLHttpRequest**

Для ограничения XmlHttpRequest используется философия "Same Origin Policy". Она очень проста - каждый сайт в своей песочнице. Запрос можно делать только на адреса с тем же протоколом, доменом, портом, что и текущая страница.

Т.е, со страницы на адресе http://site.com нельзя сделать XmlHttpRequest на адрес **https**://site.com, http://site.com:**81** или http://**othersite.com**

Это создает проблему, если хочется взять контент с другого сайта. Как правило, в этом случае вместо XmlHttpRequest используются другие средства, например, загрузка через динамически создаваемый тег <script>. Но, конечно, XmlHttpRequest удобнее и мощнее, поэтому некоторые средства для кросс-доменных запросов все же придуманы.

**Проксирование**

Самый простой способ обойти это ограничение - проксирование. Допустим, мы хотим сделать запрос с http://site.com на http://remote.com/get.html.

Чтобы обойти ограничение, вместо указания remote.com в методе open(), там ставится специальный URL вида http://site.com/proxy/remote.com/get.html. Так что запрос приходит на наш веб-сервер, который проксирует его на сервер site.com, который в свою очередь обрабатывает этот запрос, как нужно.

Если remote.com находится на другом сервере, то серверу site.com придется проксировать посетителю как запрос, так и ответ. При этом, разумеется, никак не будут задействованы куки remote.com, так что не получится отдельной авторизации, учета пользователей или чтото в этом роде с отдельными куками.

Проксирование настраивается соответствующим модулем (mod\_proxy, proxy module и т.п.) веб-сервера для всех адресов, начинающихся на /proxy.

Например, при использовании web-сервера Apache, для проксирования нужны директивы ProxyPass, ProxyPassReverse. Кроме того, доступны еще модули, которые по необходимости правят урлы, разархивируют контент [и т.п.](http://www.askapache.com/htaccess/reverse-proxy-apache.html)

**Использование наддомена**

Часто кроссбраузерные запросы - это

1. Способ обойти ограничения в 2 одновременных соединения к одному домену-порту.
2. Способ использовать два разных сервера в общении с посетителем. Например, на chat.site.ru - чат-демон, на www.site.ru - веб-сервер.

Кросс-доменные запросы с поддомена типа http://a.site.com, http://b.site.com на базовый домен site.com допустимы при использовании свойства document.domain, которое надо установить в site.com

// на странице a.site.com

...

document.domain='site.com'

...

// все, теперь могу делать XmlHttpRequest на site.com

xmlhttp.open(..'http://site.com/feedme.php'..)

**Запрос на старый домен**

В браузере Internet Explorer, чтобы сделать запрос на старый домен a.site.com, нужно вернуть свойство document.domain обратно. В остальных браузерах это приводит к ошибке, поэтому можно оформить код типа такого:

var oldDomain = document.domain

document.domain = "site.com"

... работаем с site.com ...

try {

// для IE, в остальных браузерах ошибка...

document.domain = oldDomain;

} catch(e) { /\* ... но в них все и так работает \*/ }

... работаем с a.site.com ...

**Same origin и фреймы**

Приятным бонусом свойства document.domain является возможность коммуникации между фреймами/ифреймами на одном домене.

То есть, например, если

* во фрейме с адреса http://a.site.com установлен document.domain='site.com',
* на фрейме с адреса http://b.site.com установлен домен document.domain='site.com'
* на фрейме с адреса http://site.com установлен (обязательно!) домен document.domain='site.com'

То эти три фрейма могут свободно общаться посредством javascript и XmlHttpRequest.

Обычно такая коммуникация используется при создании чатов/событий с сервера, когда на site.com находится основной веб-сервер, а на chat.site.com висит чат-демон.

**Internet Explorer trusted zone**

Любые запросы допустимы между сайтами, находящимися в доверенной (trusted) зоне Internet Explorer. Так что, внутренний корпоративный портал может быть у всех пользователей в этой зоне, и он сможет делать запросы к любым сайтам.

**XhrIframeProxy**

Еще один хитрый подход называется [XHRIframeProxy](http://www.google.ru/search?q=xhriframeproxy), и позволяет делать XmlHttpRequest к любым доменам при помощи хитрого iframe-хака. Он основан на том, что фреймы с разных доменов могут читать и менять друг у друга anchor, т.е часть адреса после решетки '#'. За счет этого организуется специальный протокол, по которому "проксируется" XmlHttpRequest.

Этот метод, в принципе, вполне жизнеспособен, особенно для небольшого объема данных.

**Кросс-доменные запросы в FF3/IE8/Opera9..**

В спецификации HTML 5 предусмотрены кросс-доменные запросы [postMessage](http://www.whatwg.org/specs/web-apps/current-work/multipage/section-crossDocumentMessages.html" \l "crossDocumentMessages).

Создатели Firefox и Opera реализовали этот вариант, см. например MDC: [DOM:window.postMessage](http://developer.mozilla.org/en/docs/DOM:window.postMessage).

Разработчики IE8 пошли другим путем и предлагают [XDomainRequest](http://msdn.microsoft.com/en-us/library/cc288060.aspx).

Оба способа вполне жизнеспособны и уже пригодны для использования в интранет-приложениях, когда на всех машинах администратор ставит одинаковый браузер, например, Firefox 3 ;)

**Поддержка в библиотеках и фреймворках**

Практически каждая javascript-библиотека или javascript-фреймворк включает в том или ином виде поддержку XmlHttpRequest-запросов и других способов прозрачного общения с сервером. Берите фреймворк по другим параметрам, а какая-то поддержка так обязательно будет.

**Javascript-библиотеки**

**Dojo toolkit**

Наиболее профессионально общение с сервером, на мой взгляд, сделано в [dojo](http://dojotoolkit.org). Для удобства работы с асинхронными вызовами, в dojo и [Mochikit](http://mochikit.com) используется специальный объект Deferred. Умеет посылать формы, отменять запросы, позволяет строить сложные цепочки асинхронных вызовов. В dojo для этого используется вызов dojo.xhrGet, который позволяет указывать обработчик, таймаут и формат запроса (например, JSON). Также умеет предотвращать кеширование (preventCache), передавать объекты/формы с файлами.

Надо сказать, что в dojo есть еще масса других транспортов, которые позволяют вытворять со связью клиент-сервер все, что только возможно и невозможно... Надо только разобраться как, на момент написания доки, откровенно говоря, слабоваты.

**Yahoo UI (YUI)**

В [Yahoo UI](http://developer.yahoo.com/yui/) соединениями с сервером заведует [Connection Manager](http://developer.yahoo.com/yui/docs/module_connection.html). Главная фунция asyncRequest принимает в качестве одного из параметров (callback) объект, который позволяет [подписываться на события](http://developer.yahoo.com/yui/examples/connection/global_customevents.html), [указывать timeout](http://developer.yahoo.com/yui/examples/connection/abort.html) и посылать на сервер объект. Кроме того можно указывать временной промежуток для автоматических опросов. Например, опрашивать новости с сервера каждые 3 секунды. Метод setForm передает форму, умеет загружать файлы.

**Prototype**

Во фреймворке [prototype](http://www.prototypejs.com/) Ajax представлен рядом классов вида Ajax.\*. В сочетании с другими методами библиотеки - предоставляет весь стандартный функционал. Кроме того - приятный бонус: Ajax.PeriodicalUpdater умеет легко обновлять HTML-элемент с сервера и гибко увеличивать промежуток между опросами при проблемах серверной части.

**JsHttpRequest**

Есть еще библиотека [JsHttpRequest](http://dklab.ru/lib/JsHttpRequest/), которая набрала популярность за счет русской документации и коммунити. Весь базовый функционал у нее есть. Лично я ни разу не пользовался, но говорят - работает. Если Вы не знаете английского языка и не нуждаетесь в интеграции AJAX с более общим javascript-фреймворком - возможно, эта библиотека подойдет.

**Серверные библиотеки**

Есть специальные серверные библиотеки, которые упрощают работу с XmlHttpRequest, организуя не только javascript-часть, но и серверную тоже. Они обычно умеют, например, отображать серверные функции на php в javascript-аналоги. При вызове такого javascript-аналога библиотека сама сделает запрос на сервер, обработает его на сервере, вызовет серверную функцию и вернет ее результат.

Для PHP одной из лучших библиотек является [XAJAX](http://www.xajaxproject.org/), для Java - [DWR](http://getahead.org/dwr).

**...А если...**

... Ну а если фреймворка не хочется, или надо то, чего во фреймворках нет, надеюсь, после прочтения этой доки, Вы без проблем реализуете все сами.

<http://javascript.ru/ajax/intro>

<http://javascript.ru/ajax/transport/xmlhttprequest>

<http://xmlhttprequest.ru/>