**AJAX**

Ajax, состоит из HTML, технологии JavaScript, DHTML и DOM. Ниже раскрываются основные концепции Ajax, включая объект XMLHttpRequest.

**Что делает Ajax**

Web-приложения работают где-то на Web-сервере, а вы обращаетесь к этим приложениям через ваш Web-браузер.

От того, где выполняется код этих приложений, зависит их поведение и способ вашего взаимодействия с ними. Настольные приложения обычно достаточно быстрые (они работают на вашем компьютере; вы не ждете интернет-подключения), имеют отличные пользовательские интерфейсы (обычно взаимодействующие с вашей операционной системой) и невероятно динамичны. Вы можете щелкать мышкой, вводить текст, пользоваться ниспадающими и всплывающими меню, перемещаться по окнам практически без каких-либо задержек.

Web-приложения обычно самые свежие по времени и предоставляют возможности, которые вы никогда бы не смогли иметь на вашем компьютере (вспомните Amazon.com и eBay). Однако с могуществом Web приходит ожидание – ожидание ответа от сервера, ожидание обновления экрана, ожидание ответа на запрос и генерирования новой страницы.

Ajax пытается преодолеть разрыв между функциональностью и интерактивностью настольного приложения и всегда обновленным Web-приложением. Вы можете использовать динамические пользовательские интерфейсы, аналогичные имеющимся в настольном приложении, но доступные в Web-приложении.

**Определение Ajax**

Ajax – это аббревиатура от Asynchronous JavaScript and XML (и DHTML, и т.д.).

Ajax охватывает несколько различных технологий:  
\* HTML используется для создания Web-форм и указания полей для использования в вашем приложении.  
\* JavaScript-код – это основной код, выполняющий Ajax-приложения и обеспечивающий взаимодействие с серверными приложениями.  
\* DHTML, или Dynamic HTML, помогает динамически обновлять формы. Вы будете использовать div, span и другие динамические HTML-элементы для разметки вашего HTML.  
\* DOM, Document Object Model (объектная модель документов), будет использоваться (через код JavaScript) для работы и со структурой вашего HTML, и (в некоторых случаях) с XML, полученным от сервера.

**В старых Web-приложениях** пользователи заполняют поля форм и нажимают кнопку Submit (подтвердить). Затем форма передается на сервер полностью, сервер обрабатывает сценарий (обычно PHP или Java, возможно, CGI-процесс или что-то в этом роде), а потом передает назад всю новую страницу, а **пользователи должны ждать**. Их экраны очистятся и будут перерисовываться по мере поступления новых данных от сервера.

Ajax действует иначе: Некое действие пользователя запускает код JavaScript, который  
1) читает данные из формы  
2) передает запрос на сервер  
3) получает ответ от сервера.  
4) изменяет веб-страницу

Запрос передается асинхронно, а это означает, что ваш код JavaScript (и пользователь) не ожидают ответа сервера. Пользователи могут продолжать вводить данные, прокручивать страницу и работать с приложением. Пользователь даже не замечает, что происходит запрос на сервер.

Для выполнения задач 1 и 4 надо знать о методе getElementById()

Листинг 2. Чтение данных из формы на веб-странице

// Получить значение поля "phone" и записать его в переменную phone

var phone = document.getElementById("phone").value;

Листинг 3. Установка значений полей при помощи кода JavaScript

// Установить значения в форме, используя массив response

document.getElementById("order").value = response[0];

document.getElementById("address").value = response[1];

**Наиболее полный контроль над веб-страницей вы можете получить с помощью DOM.** DOM - Document Object Model (объектная модель документа) в JavaScript.

**Создание объекта Request**

XMLHttpRequest является центральным для Ajax-приложений. Он управляет взаимодействием вашего кода JavaScript с сервером и является основой Ajax. Но в разных браузерах он работает по разному.

**Microsoft Internet Explorer** для обработки XML использует анализатор MSXML. В ходу две различных версии MSXML. Версия MSXML зависит от версии JavaScript, установленной в Internet Explorer, поэтому нужен код, подходящий для обеих версий:

**Создание объекта XMLHttpRequest в браузерах Microsoft**

var xmlHttp = false;

try { xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");}

catch (e){

try {xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");}

catch (e2){ xmlHttp = false;}

}

Обратите внимание на:  
**xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");**  
и  
**xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");**.  
  
**Работа с Mozilla и браузерами не от Microsoft**

var xmlHttp = new XMLHttpRequest();.

Эта строка создает объект XMLHttpRequest в Mozilla, Firefox, Safari, Opera и в большой степени в каждом браузере не от Microsoft, поддерживающем Ajax в любой форме или разновидности.

**Универсальный код создания XMLHttpRequest**

var xmlHttp = false;

try {xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");}

catch (e1){

try { xmlHttp = new ActiveXObject("Microsoft.XMLHTTP"); }

catch (e2) {

try{xmlHttp = new XMLHttpRequest();}

catch(e3){xmlHttp = false;}

}

}

Этот код можно разделить на три шага:  
1. Создать переменную xmlHttp для ссылки на объект XMLHttpRequest, который вы создадите.  
2. В блоке try создать объект в браузерах Microsoft:  
• В блоке try создать объект с использованием объекта Msxml2.XMLHTTP.  
• Если это не получится, В блоке try создать объект с использованием объекта Microsoft.XMLHTTP.  
3. Если xmlHttp все еще не установлен, создать объект для остальных браузеров.  
  
В конце этого процесса переменная xmlHttp должна ссылаться на корректный объект XMLHttpRequest, независимо от используемого пользователем браузера.

**Обработка ошибок**

Написанный выше код не обеспечивает обработки всех ошибок. Например, многие старые браузеры не поддерживают XMLHttpRequest, и вам нужно предупредить таких пользователей, что что-то не получилось.

**Создание XMLHttpRequest с некоторыми способностями обработки ошибок**

var xmlHttp = false;

try {xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");}

catch (e1){

try { xmlHttp = new ActiveXObject("Microsoft.XMLHTTP"); }

catch (e2) {

try{xmlHttp = new XMLHttpRequest();}

catch(e3){xmlHttp = false;}

}

}

if (!xmlHttp) alert("Error initializing XMLHttpRequest!");

Теперь у вас есть более защищенный от ошибок код, создающий объект XMLHttpRequest и даже предупреждающий вас о том, что что-то прошло не так.

**О защите**

Современные браузеры предлагают пользователям возможность повысить уровень безопасности, отключить технологию JavaScript и запретить множество настроек в своих браузерах. В этих ситуациях ваш код, вероятно, не будет работать ни при каких обстоятельствах. Для таких ситуаций вы должны элегантно решить проблемы. Пока же вы пишете устойчивый, но не совершенный код, который отлично подходит для изучения Ajax. Вы вернетесь к более качественному коду позже.

**Выполнение запроса**

Нам понадобится функция JavaScript для инициализации запроса, в которой нужно:  
1. Получить какие-либо данные из Web-формы.  
2. Создать URL для подключения.  
3. Открыть соединение с сервером.  
4. Установить функцию для сервера, которая выполнится после его ответа.  
5. Передать запрос.

Листинг 5. Выполнить запрос с Ajax

function callServer() {

// Получить city и state из web-формы

var city = document.getElementById("city").value;

var state = document.getElementById("state").value;

// Продолжать только если есть значения обоих полей

if ((city == null) || (city == )) return;

if ((state == null) || (state == )) return;

// Создать URL для подключения

var url = "/scripts/getZipCode.php?city=" + escape(city) + "&state=" + escape(state);

// Открыть соединение с сервером

xmlHttp.open("GET", url, true);

// Установить функцию для сервера, которая выполнится после его ответа

xmlHttp.onreadystatechange = updatePage;

// SПередать запрос

xmlHttp.send(null);

}

Свойство onreadystatechange позволяет установить в браузере обработчик события "изменение состояния" (обычно это получение ответа от сервера). Поскольку код не собирается ждать ответа, браузер должен знать что делать с полученным ответом. В данном случае будет инициирован конкретный метод (называемый updatePage()) после завершения сервером обработки вашего запроса.  
  
Наконец, вызывается send() со значением null. Поскольку вы добавили данные для передачи на сервер (city и state) в URL запроса, вам не надо передавать что-либо в запросе. (Аргумент метода send() – это данные, отправляемые методом POST).

**Обработка ответа сервера**

Пока вы должны знать только два момента:  
\* Не делать ничего, пока свойство xmlHttp.readyState не будет равно 4.  
\* Сервер будет записывать свой ответ в свойстве xmlHttp.responseText.  
  
Листинг 6. Обработка ответа от сервера

function updatePage() {

if (xmlHttp.readyState == 4) {

var response = xmlHttp.responseText;

document.getElementById("zipCode").value = response;

}

}

Инициализация запроса

<form>

<p>City: <input type="text" name="city" id="city" size="25"

onChange="callServer();" />

<p>State: <input type="text" name="state" id="state" size="25"

onChange="callServer();" /></p>

<p>Zip Code: <input type="text" name="zipCode" id="zipCode" size="5" /></p>

</form>

**Как использовать Ajax**

Представьте себе Web-форму, которая реагирует не только тогда, когда вы нажимаете кнопку, но и когда вы вводите данные в поле, когда выбираете вариант из списка вариантов и даже когда перемещаете курсор по экрану. Подумайте о том, что на самом деле означает слово асинхронный. Подумайте о JavaScript-коде, выполняющемся и не ожидающем, пока сервер возвратит ответы на его запросы. С какими проблемами вы можете столкнуться? С какими вещами следует проявлять осторожность? И как дизайн ваших форм изменится, принимая во внимание этот новый подход в программировании?

**Выполнение асинхронных запросов с JavaScript и Ajax**

Большинство Web-приложений используют модель запрос/ответ, в которой получают от сервера HTML-страницу полностью. В результате получаем работу по типу "вперед и назад", обычно состоящую из нажатия кнопки, ожидания сервера, нажатия другой кнопки и опять ожидания сервера. При помощи Ajax и объекта XMLHttpRequest вы можете использовать модель запрос/ответ, которая никогда не оставляет пользователей в ожидании ответа сервера. Асинхронность позволяет послать серверу несколько запросов одновременно.  
  
Недостатки и преимущества асинхронности:  
+ приложение работает субъективно быстрее  
+ передаётся меньшее количество данных (только изменения)  
+- серверу можно отправлять несколько запросов  
- сервер трудится больше  
- надо следить, чтобы один не возникали конфликты среди желающих использовать один и тот же объект  
- быстрее передать 1 раз 10 кб, чем 10 раз по 1 кб. Если запросов много, то сеть забивается служебными запросами

**Подробнее о XMLHttpRequest**

Основные методы и атрибуты:  
\* open(): Устанавливает новый запрос к серверу;  
\* send(): Передает запрос серверу;  
\* abort(): Прекращает текущий запрос;  
\* readyState: Хранит текущее состояние готовности HTML;  
\* responseText: Текст, который сервер передал назад как ответ на запрос.  
  
Каждый из этих методов и свойств относится к передаче запроса и работает с ответом.

**Работа с браузерами Microsoft**

Ожидается, что Internet Explorer версия 7.0, намеченная к выходу в 2006 г., будет поддерживать XMLHttpRequest напрямую, позволяя вам использовать ключевое слово new вместо всего кода создания Msxml2.XMLHTTP. Но не слишком обольщайтесь; вам все равно необходимо поддерживать старые браузеры, поэтому кросс-браузерный код уйдет со сцены еще не скоро.  
  
Microsoft поддерживает Ajax, но называет свою версию XMLHttpRequest по-другому. Вернее он называет этот объект несколькими разными именами. Если вы используете более новую версию Internet Explorer, то должны использовать объект Msxml2.XMLHTTP; некоторые старые версии используют Microsoft.XMLHTTP.  
  
Листинг 4. Добавление поддержки для браузеров Microsoft

<script language="javascript" type="text/javascript">

var request = false;

try {

request = new XMLHttpRequest();

} catch (trymicrosoft) {

try {

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

} catch (othermicrosoft) {

try {

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

} catch (failed) {

request = false;

}

}

}

if (!request) alert("Error initializing XMLHttpRequest!");

</script>

**Статика против динамики**

Обратите внимание, что код JavaScript выполняется в какой-то момент времени до того, как страница отобразится пользователю (из спецификации не ясно с точностью 100%, когда этот код выполняется, и браузеры поступают по-разному; но все же есть гарантия, что код выполняется до того, как пользователи смогут взаимодействовать с вашей страницей). Обычно Ajax-программисты именно так и создают объект XMLHttpRequest.  
  
Как было сказано, вы определенно можете поместить этот код в метод:  
  
Листинг 5. Перемещение кода создания XMLHttpRequest в метод

<script language="javascript" type="text/javascript">

var request;

function createRequest() {

try {

request = new XMLHttpRequest();

} catch (trymicrosoft) {

try {

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

} catch (othermicrosoft) {

try {

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

} catch (failed) {

request = false;

}

}

}

if (!request)

alert("Error initializing XMLHttpRequest!");

}

</script>

Поступив так, вы должны вызвать этот метод до начала любой работы с Ajax. То есть, вы должны иметь что-то похожее на листинг 6.

Листинг 6. Использование метода для создания XMLHttpRequest

<script language="javascript" type="text/javascript">

var request;

function createRequest() {

try {

request = new XMLHttpRequest();

} catch (trymicrosoft) {

try {

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

} catch (othermicrosoft) {

try {

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

} catch (failed) {

request = false;

}

}

}

if (!request) alert("Error initializing XMLHttpRequest!");

}

function getCustomerInfo() {

createRequest();

// Сделать что-то с переменной request

}

</script>

Единственное замечание по поводу этого кода (и причина того, почему многие Ajax-программисты не используют такой подход) – он задерживает вывод сообщения об ошибке. Представьте, что у вас имеется сложная форма с 10 или 15 полями, рамками выбора вариантов и т.п., и вы активизируете какой-либо Ajax-код при вводе пользователем текста в поле 14 (внизу формы). Активизируется getCustomerInfo(), который пытается создать объект XMLHttpRequest, и (для данного примера) терпит неудачу. Пользователю выводится предупреждение, сообщающее о том (во многих словах), что он не может использовать это приложение. Но пользователь уже потратил время для заполнения формы! Это довольно неприятно, и эта неприятность не входит в число тех вещей, из-за которых пользователь захотел бы вернуться на ваш сайт.  
  
В случае использования статического JavaScript, пользователь получит сообщение об ошибке, как только зайдет на вашу страницу. Это так же неприятно? Возможно. Можно взбесить пользователя тем, что ваше Web-приложение не работает в его браузере. Однако это определенно лучше, чем выдать эту же ошибку после 10 минут ввода пользователем информации. Только по этой причине я советую вам устанавливать ваш код статически и заранее предупреждать пользователей о возможных проблемах.

**Передача запросов с XMLHttpRequest**

**Ajax имеет модель безопасности по типу "песочницы"**

В результате ваш Ajax-код (и, в частности, объект XMLHttpRequest) может посылать запросы только тому домену, на котором выполняется.

**Установка URL сервера**

Первое, что вам необходимо определить – это URL сервера для подключения. Это не является специфичным для Ajax (очевидно вы должны знать, как составить URL), но это существенно для выполнения соединения. Листинг 7. Создание URL-запроса

<script language="javascript" type="text/javascript">

var request = false;

// здесь создаётся объект request

function getCustomerInfo() {

var phone = document.getElementById("phone").value;

var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);

}

</script>

Листинг 8. Форма "Break Neck Pizza"

<body>

<form action="POST">

<p>Enter your phone number:

<input type="text" size="14" name="phone" id="phone"

onChange="getCustomerInfo();" />

</p>

<p><input type="submit" value="Order Pizza" id="submit" /></p>

</form>

</body>

Обратите также внимание, что когда пользователи вводят телефонный номер или изменяют номер, вызывается метод getCustomerInfo(), приведенный в листинге 8. Затем этот метод собирает этот номер и использует его для составления строки URL, сохраняемой в переменной url. Помните: поскольку Ajax-код выполняется в "песочнице" и может соединяться только с тем же самым доменом, вам фактически не нужно указывать домен в URL. В данном примере имя сценария - /cgi-local/lookupCustomer.php. Наконец, номер телефона добавляется к этому сценарию как параметр GET: "phone=" + escape(phone).

**Внимание: метод escape() нетривиально работает с русскими кодировками.**

**Открытие запроса**

Имея URL, вы можете сконфигурировать запрос. Это делается при помощи метода open() вашего объекта XMLHttpRequest. Этот метод принимает пять параметров:  
\* request-type: Тип передаваемого запроса. Обычными значениями являются GET или POST, но вы можете также передать HEAD-запросы.  
\* url: URL для соединения.  
\* asynch: True, если вы хотите, чтобы запрос был асинхронным, и false, если он должен быть синхронным. Этот параметр необязателен и по умолчанию равен true.  
\* username: Если требуется аутентификация, то здесь вы можете указать имя пользователя. Это необязательный параметр и он не имеет значения по умолчанию.  
\* password: Если требуется аутентификация, здесь вы можете указать пароль. Это необязательный параметр и он не имеет значения по умолчанию.

**Открывает ли open()?**  
Internet-разработчики не пришли к согласию насчет того, что точно делает метод open(). Чего он действительно не делает, так это не открывает запрос. Если бы вы проследили за сетью и передачей данных между вашей XHTML/Ajax-страницей и сценарием, с которым она связана, то не увидели бы никакого трафика при вызове метода open(). Непонятно почему было выбрано именно это имя, но ясно, что это не лучший выбор.  
  
Обычно вы будете использовать три первых параметра. На самом деле, даже если вы хотите асинхронный запрос, желательно указывать "true" в качестве третьего параметра. Это значение по умолчанию, но это приятная мелочь для самодокументирования. Всегда видно, асинхронный запрос или нет.  
Объедините все это вместе, и у вас, скорее всего, получится примерно такая строка, какая показана в листинге 9.  
  
Листинг 9. Открытие запроса

function getCustomerInfo() {

var phone = document.getElementById("phone").value;

var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);

request.open("GET", url, true);

}

Как только вы вычислили URL, все становится тривиальным. Для большинства запросов использования GET достаточно (в будущих статьях вы увидите ситуации, в которых, возможно, захотите использовать POST); это все, что вам нужно для использования open().

**Асинхрония**

Почему последний параметр в open() настолько важен? В обычной модели запрос/ответ клиент (ваш браузер или код, выполняющийся на вашей локальной машине) посылает запрос серверу. Этот запрос является синхронным; другими словами, клиент ждет ответа от сервера. Пока клиент ждет, вы обычно получаете одну из нескольких форм уведомления о том, что вы ждете:  
\* Песочные часы (главным образом в Windows).  
\* Крутящийся мяч (обычно на Mac-компьютерах).  
\* Приложение замирает и иногда меняется курсор.  
Именно это дает ощущение, что Web-приложения неуклюжи или медленны – отсутствие реальной интерактивности.  
Асинхронный запрос не ждет ответа от сервера. Вы посылаете запрос, и ваше приложение продолжает работу. Пользователи могут вводить данные в Web-форму, нажимать другие кнопки, даже уйти с формы. Нет крутящегося мяча или вращающихся песочных часов, и нет больших замираний приложения. Сервер молча отвечает на запрос, и когда завершается передача ответа, он дает знать об этом инициатору запроса (способами, о которых вы вскоре узнаете). В результате мы имеем приложение, которое не выглядит неуклюжим или медленным, а, наоборот, чувствительным, интерактивным и быстрым.

**Передача запроса**

Метод для передачи запроса - send().  
send() принимает только один параметр – содержимое для передачи.  
  
И хотя вы можете передать данные с использованием send(), вы можете также передать их через сам URL. В действительности в GET-запросах (которые будут составлять до 80% типичного использования Ajax) намного легче передавать данные в URL.  
Когда вы начинаете передавать защищенную информацию или XML, нужно рассмотреть передачу содержимого через send()  
Если вам не надо передавать данные через send(), установите аргумент в значение null.  
  
Листинг 10. Передача запроса

function getCustomerInfo() {

var phone = document.getElementById("phone").value;

var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);

request.open("GET", url, true);

request.send(null);

}

**Указание обработчика ответа**

Свойство XMLHttpRequest с названием onreadystatechange.  
Запрос настраивается и затем передается. Поскольку это асинхронный процесс, JavaScript-метод (getCustomerInfo() в примере) не будет останавливаться в ожидании ответа. Поэтому код будет продолжать выполняться; в данном случае это означает, что произойдет выход из метода и управление вернется в форму.  
Объект XMLHttpRequest должен иметь инструкцию некоего типа о том, что делать, когда он придёт ответ от сервера. Свойство onreadystatechange позволяет указать функцию обработки ответа.  
  
Листинг 11. Установка обработчика события

function getCustomerInfo() {

var phone = document.getElementById("phone").value;

var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);

request.open("GET", url, true);

request.onreadystatechange = updatePage;

request.send(null);

}

Обратите особое внимание, что это свойство устанавливается ПЕРЕД перед вызовом send().

**Обработка ответов сервера**

Вы уже увидели, как дать браузеру знать, что делать после завершения обработки запроса: установить свойство onreadystatechange объекта XMLHttpRequest в имя функции, которая будет выполнена. Затем, когда браузер получит ответ от сервера, он автоматически вызовет эту функцию. Этот метод параметров не имеет.  
  
Листинг 12. Код обработчика

<script language="javascript" type="text/javascript">

var request = false;

try {

request = new XMLHttpRequest();

} catch (trymicrosoft) {

try {

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

} catch (othermicrosoft) {

try {

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

} catch (failed) {

request = false;

}

}

}

if (!request) alert("Error initializing XMLHttpRequest!");

function getCustomerInfo() {

var phone = document.getElementById("phone").value;

var url = "/cgi-local/lookupCustomer.php?phone=" + escape(phone);

request.open("GET", url, true);

request.onreadystatechange = updatePage;

request.send(null);

}

function updatePage() {

alert("Server is done!");

}

</script>

Он просто выводит предупреждение о том, что объект изменил состояние готовности (прибытие ответа от сервера тоже изменяет состояние)  
В зависимости от вашего браузера вы получите два, три или даже четыре предупреждения, прежде чем форма прекратит отображать всплывающее окно.

**Состояния готовности HTTP**

Состояние готовности HTTP отображает состояние или статус запроса. Оно используется для определения того, начат ли запрос, принимается ли ответ, или завершен ли цикл запрос/ответ. Это состояние помогает также при определении того, можно ли читать текст или данные, которые сервер мог предоставить в ответе. Вы должны знать о пяти состояниях готовности в ваших Ajax-приложениях:  
\* 0: Запрос не инициализирован (перед вызовом open()).  
\* 1: Запрос инициализирован, но не был передан (перед вызовом send()).  
\* 2: Запрос был передан и обрабатывается (на данном этапе вы можете обычно получить заголовки содержимого из ответа).  
\* 3: Запрос обрабатывается; часто в ответе доступны некоторые частичные данные, но сервер не закончил свой ответ.  
\* 4: Ответ завершен; вы можете получить ответ сервера и использовать его.  
Как и почти все кросс-браузерные проблемы, эти состояния готовности используются несколько непоследовательно. Вы могли бы ожидать, что состояния всегда изменяются от 0 к 1, от 1 к 2, от 2 к 3, от 3 к 4, но на практике это редкий случай. Некоторые браузеры никогда не выдают 0 или 1, а сразу перепрыгивают к 2, затем к 3 и 4. Другие браузеры выдают все состояния. Но некоторые выдают состояние 1 несколько раз. Как вы видели в последнем разделе, сервер вызывал updatePage() несколько раз, и каждый вызов приводил к появлению всплывающего окна предупреждения – наверное, это не то, чего вы хотели!  
Для Ajax-программирования единственное состояние, с которым вы должны иметь дело, - это состояние готовности 4, указывающее на то, что ответ сервера завершен и можно проверить ответные данные и использовать их.  
  
Листинг 13. Проверка состояния готовности

function updatePage() {

if (request.readyState == 4)

alert("Server is done!");

}

Это изменение проверяет, действительно ли сервер завершил процесс. Попробуйте поработать с этой версией кода, и вы должны получить только одно предупреждающее сообщение, как и должно быть.

**Коды состояния http**

Что если сервер ответил на ваш запрос и завершил обработку, но возвращает ошибку?  
То есть, кроме состояния готовности вы должны также проверять HTTP-состояние. Вам подходит код состояния 200, просто означающий, что все в порядке.  
  
Листинг 14. Проверка кода состояния http

function updatePage() {

if (request.readyState == 4)

if (request.status == 200)

alert("Server is done!");

}

**Чтение текста ответа**

Текст ответа сохранен в свойстве responseText объекта XMLHttpRequest. Тип возвращаемого значения – обычный текст  
Листинг 16. Работа с ответом сервера

function updatePage() {

if (request.readyState == 4) {

if (request.status == 200) {

var response = request.responseText.split("|");

document.getElementById("order").value = response[0];

document.getElementById("address").innerHTML =

response[1].replace(/\n/g, "  
");

} else

alert("status is " + request.status);

}

}

**Ответ в формате XML**

Еще одно важное свойство XMLHttpRequest - responseXML. Это свойство содержит XML-ответ в случае, если сервер решил ответить в XML-формате. Работа с XML ответом существенно отличается от работы с обычным неформатированным текстом и включает работу с синтаксическим анализатором, Document Object Model (DOM) и некоторые другие соображения.

**Усовершенствованные запросы и ответы в Ajax**

**Подробно о состояниях готовности HTTP**

Вы должны помнить из предыдущей статьи, что объект XMLHttpRequest имеет свойство readyState. Это свойство удостоверяет, что сервер завершил запрос, и обычно функция обратного вызова использует данные от сервера для обновления Web-формы или страницы.  
  
Листинг 1. Работа с ответом сервера в функции обратного вызова

function updatePage() {

if (request.readyState == 4) {

if (request.status == 200) {

var response = request.responseText.split("|");

document.getElementById("order").value = response[0];

document.getElementById("address").innerHTML =

response[1].replace(/\n/g, "  
");

} else

alert("status is " + request.status);

}

}

Это определенно наиболее общее (и наиболее простое) использование состояний готовности. Как вы, возможно, догадались по числу "4", существует и несколько других состояний готовности:  
\* 0: Запрос не инициализирован (перед вызовом open()).  
\* 1: Запрос инициализирован, но не был передан (перед вызовом send()).  
\* 2: Запрос был передан и обрабатывается (на данном этапе вы можете обычно получить заголовки содержимого из ответа).  
\* 3: Запрос обрабатывается; часто в ответе доступны некоторые частичные данные, но сервер не закончил свой ответ.  
\* 4: Ответ завершен; вы можете получить ответ сервера и использовать его.  
  
Если вы хотите выйти за рамки основ Ajax-программирования, то должны знать не только эти состояния, но и когда они возникают, а также как вы можете использовать их. Первое и самое главное – вы должны изучить, на каком этапе запроса возникает каждое состояние. К сожалению, это не такая уж интуитивная вещь, и имеются специальные случаи.

**Скрытые значения состояний готовности**

Первое состояние готовности, обозначаемое значением свойства readyState равным 0 (readyState == 0), представляет неинициализированный запрос. Как только вы вызовете метод open() вашего объекта запроса, это свойство устанавливается в 1. Поскольку вы почти всегда вызываете open() сразу после инициализации вашего запроса, редко можно увидеть readyState == 0. Более того, неинициализированное состояние готовности достаточно бесполезно в реальных приложениях.  
Очевидно, это не очень хорошо; существует очень мало случаев, когда вы должны проверять, что open() не была вызвана. Единственным использованием этого состояния готовности в "почти реальном" Ajax-программировании будет ситуация, когда вы выполняете несколько запросов, используя один и то же объект XMLHttpRequest из нескольких функций. В этой ситуации (довольно необычной) вы, возможно, захотите убедиться, что объект запроса находится в неинициализированном состоянии (readyState == 0) перед выполнением новых запросов. Это, в сущности, гарантирует, что другая функция не использует объект в это же время.

**Обзор состояния готовности выполняющегося запроса**

От состояния готовности 0 ваш объект запроса должен проходить через каждое другое состояние в обычном запросе и ответе, и, в конце концов, закончить на состоянии 4. Вот почему вы видите строку кода if (request.readyState == 4) в большинстве функций обратного вызова; она гарантирует, что сервер закончил свою работу с запросом и можно без опасений обновить Web-страницу или выполнить действие, основанное на полученных от сервера данных.

**Несовместимость браузеров**

Например, в **Firefox** 1.5 вы увидите следующие состояния готовности: 1 => 2 => 3 => 4  
  
А вот состояния в **Safari** 2.0.1: 2 => 3 => 4  
Safari на самом деле пропускает первое состояние, и нет здравого объяснения.  
  
**Opera** 8.5 : 3=> 4  
  
**Internet Explorer** отображает следующие состояния: 1 => 2 => 3 => 4  
  
Если с запросом имеются проблемы, это самое первое место, где следует их искать. Добавьте вывод предупреждения, чтобы увидеть состояние готовности запроса и убедиться в том, что все работает нормально. Еще лучше - протестируйте и в Internet Explorer, и в Firefox – вы получите все четыре состояния готовности и сможете проверить каждый этап запроса.

**Данные ответа под микроскопом**

Cвойство объекта XMLHttpRequest responseText.  
  
Как только браузер завершит обработку запроса и ответа, он размещает все данные, необходимые для ответа на запрос, в свойство responseText объекта XMLHttpRequest. После этого ваша функция обратного вызова может использовать эти данные  
Листинг 4. Использование ответа от сервера

function updatePage() {

if (request.readyState == 4) {

var newTotal = request.responseText;

var totalSoldEl = document.getElementById("total-sold");

var netProfitEl = document.getElementById("net-profit");

replaceText(totalSoldEl, newTotal);

/\* Определить новую чистую прибыль \*/

var boardCostEl = document.getElementById("board-cost");

var boardCost = getText(boardCostEl);

var manCostEl = document.getElementById("man-cost");

var manCost = getText(manCostEl);

var profitPerBoard = boardCost - manCost;

var netProfit = profitPerBoard \* newTotal;

/\* Обновить чистую прибыль на форме продаж \*/

netProfit = Math.round(netProfit \* 100) / 100;

replaceText(netProfitEl, netProfit);

}

**Просмотр текстового ответа во время запроса**

Аналогично состоянию готовности значение свойства responseText изменяется на всем протяжении жизненного цикла запроса. Чтобы увидеть это в действии, используйте код, аналогичный приведенному в листинге 5, для тестирования текстового ответа, также как и состояния готовности.

**Получение надежных данных**

Во всей документации и спецификациях утверждается, что только при состоянии готовности 4 данные можно использовать безопасно. Поверьте мне, вы редко найдете случаи, когда данные не могут быть получены из свойства responseText при состоянии готовности равном 3. Однако полагаться на это в вашем приложении – плохая идея. Если вы напишете код, зависящий от завершенности данных при состоянии готовности 3, почти гарантируется, что наступит момент, когда данные будт не полны.

**Пристальный взгляд на коды состояния HTTP**

Коды состояния HTTP существуют в Web со времен его появления и браузеры их отображают:  
\* 401: Unauthorized (Не авторизован)  
\* 403: Forbidden (Запрещен)  
\* 404: Not Found (Не найден)  
\* 200 Все OK  
Гарантировать то, что сервер не только завершил обработку запроса, но и возвратил код состояния "Все в порядке", требует минимальных усилий. Этот код равен "200" и возвращается в свойстве status объекта XMLHttpRequest.  
Листинг 7. Проверка кода состояния

function updatePage() {

if (request.readyState == 4) {

if (request.status == 200) {

var response = request.responseText.split("|");

document.getElementById("order").value = response[0];

document.getElementById("address").innerHTML =

response[1].replace(/\n/g, "  
");

} else

alert("status is " + request.status);

}

}

С этими небольшим числом дополнительных строк кода вы можете быть уверены, что если что-то пойдет не так, ваши пользователи получат (под вопросом) полезное сообщение об ошибке вместо страницы с искаженными данными без всяких объяснений.

**Переадресация и перенаправление**

Перед детальным рассмотрением ошибок имеет смысл поговорить о том, о чем вы, возможно, не должны беспокоиться при использовании Ajax – переадресации. В кодах состояния HTTP есть семейство кодов состояния 300, включающих:  
\* 301: Moved permanently (Перемещен постоянно)  
\* 302: Found (Найден) - запрос переадресован на другой URL/URI  
\* 305: Use Proxy (Использовать прокси) - запрос должен использовать прокси для доступа к затребованному ресурсу  
  
Ajax-программисты, возможно, не беспокоятся о переадресации по двум причинам:  
\* Прежде всего, Ajax-приложения почти всегда пишутся для конкретного серверного сценария, сервлета или приложения. Исчезновение или перемещение в другое место этого компонента без вашего, Ajax-программиста, ведома встречается довольно редко. Поэтому почти всегда вы знаете, что ресурс переместился (потому что вы его переместили, или он уже был перемещен), меняете URL в вашем запросе и никогда не сталкиваетесь с такого рода проблемами.  
\* И еще одна, даже более важная, причина: Ajax-приложения и запросы выполняются в "песочнице" безопасности. Это значит, что доменом, обслуживающим Web-страницу, с которой выполняются Ajax-запросы, является домен, который должен на них отвечать. Поэтому Web-страница, обслуживаемая ebay.com, не может выполнить Ajax-запрос сценарию, выполняющемуся на amazon.com; Ajax-приложения ibm.com не могут выполнять запросы сервлетам, работающим на netbeans.org.  
  
В результате ваши запросы не могут быть переадресованы на другой сервер без генерирования ошибки защиты. В этих случаях вы вовсе не получите код состояния. Вы просто получите ошибку JavaScript в консоли отладки. Поэтому, думая о многообразии кодов состояния, вы можете почти совершенно игнорировать коды переадресации.

**Крайние ситуации и тяжелые ситуации**

На данный момент программисты-новички могут удивиться, из-за чего здесь весь этот шум. Совершенно ясно, что менее 5 процентов Ajax-запросов требуют работы с состояниями готовности 2 и 3 и с кодами состояния аналогичными 403 (на самом деле возможно ближе к 1 проценту или меньше). Эти случаи важны и называются крайними случаями – редкими ситуациями, которые возникают при совпадении очень необычных условий. Оставаясь редкими, крайние случаи могут быть причиной до 80 процентов наибольших пользовательских разочарований!  
  
Обычные пользователи забывают 100 случаев, когда приложение работало правильно, но ясно помнят один, когда оно работало не правильно. Если вы можете обработать крайние случаи (и тяжелые случаи) незаметно, ваши пользователи будут довольны и вернутся на ваш сайт.

**Ошибки**

Единственной группой кодов, о которой надо подумать – это 400-е семейство, указывающее на ошибки различного типа;  
Прежде всего, добавьте поддержку отсутствующих страниц. Они действительно не должны часто появляться в реальных системах, но это нередко случается при тестировании перемещенного сценария или при вводе программистом неверного URL. Если вы можете изящно обрабатывать ошибки 404, то намного лучше поможете поставленным в тупик пользователям и программистам.  
Пользователь не может узнать, в чем заключается проблема: в аутентификации, отсутствующем сценарии (как в данном случае), ошибке пользователя, или ошибке в исходном коде. Небольшие дополнения к коду могут сделать это сообщение об ошибке намного более точным. Посмотрите на листинг 8, который обрабатывает отсутствие сценария и ошибки аутентификации.  
  
Листинг 8. Проверка кода состояния

function updatePage() {

if (request.readyState == 4) {

if (request.status == 200) {

var response = request.responseText.split("|");

document.getElementById("order").value = response[0];

document.getElementById("address").innerHTML =

response[1].replace(/\n/g, "  
");

} else if (request.status == 404) {

alert ("Requested URL is not found.");

} else if (request.status == 403) {

alert("Access denied.");

} else

alert("status is " + request.status);

}

}

**Дополнительные типы запроса**

Если вы действительно хотите управлять объектом XMLHttpRequest, рассмотрим еще одну, последнюю тему. Добавьте HEAD-запросы в ваш репертуар. В первых двух статьях я рассказал, как выполнять GET-запросы; в следующей статье вы научитесь всему, что связано с передачей данных на сервер при помощи POST-запроса. В духе усовершенствованной обработки ошибок и сбора информации вы должны научиться выполнять HEAD-запросы.

**Выполнение запроса**

В действительности выполнение HEAD-запроса является довольно тривиальной задачей; вы просто вызываете метод open() с "HEAD" вместо "GET" или "POST" в качестве первого параметра.  
Листинг 9. Выполнение HEAD-запроса с Ajax

function getSalesData() {

createRequest();

var url = "/boards/servlet/UpdateBoardSales";

request.open("HEAD", url, true);

request.onreadystatechange = updatePage;

request.send(null);

}

При выполнении подобного HEAD-запроса сервер не возвращает реальный ответ, как для GET или POST-запросов. Вместо этого сервер должен только возвратить заголовки ресурса, в которые включается последнее время модификации содержимого запроса, месторасположение запрашиваемого ресурса и довольно много другой интересной информации. Вы можете использовать эту информацию о ресурсе еще до того, как сервер должен будет обработать и возвратить этот ресурс.  
  
Самой простой вещью, которую вы можете сделать с подобным запросом, является выдача всех заголовков ответа. Это даст вам представление о том, что доступно вам через HEAD-запросы. В листинге 10 приведена простая функция обратного вызова, которая отображает все заголовки ответа из HEAD-запроса.  
  
Листинг 10. Вывод всех заголовков ответа их HEAD-запроса

function updatePage() {

if (request.readyState == 4) {

alert(request.getAllResponseHeaders());

}

}

Вы можете использовать любой из этих заголовков (от типа сервера до типа содержимого) индивидуально для получения дополнительной информации или функциональности в Ajax-приложении.

**Проверка URL**

Вы увидели, как проверить ошибку 404 при отсутствии URL. Если это превращается в общую проблему (возможно, определенный сценарий или сервлет в данный момент времени находитcя в режиме offline) – вы, вероятно, захотите проверить URL перед выполнением полного GET или POST-запроса. Для этого выполните HEAD-запрос и проверьте 404-ю ошибку в вашей функции обратного вызова; в листинге 11 приведен пример такой функции.  
  
Листинг 11. Проверка существования URL

function updatePage() {

if (request.readyState == 4) {

if (request.status == 200) {

alert("URL exists");

} else if (request.status == 404) {

alert("URL does not exist.");

} else {

alert("Status is: " + request.status);

}

}

}

Это имеет небольшое значение. Сервер должен ответить на запрос и обработать его для заполнения значения длины содержимого заголовка ответа, поэтому вы не экономите время обработки. Кроме того, выполнение запроса и определение существования URL при помощи HEAD-запроса занимает столько же времени, сколько и выполнение GET или POST-запроса и последующая обработка ошибок (как показано в листинге 7. Хотя иногда может быть полезно знать точно, что доступно; вы никогда не знаете, когда вас пробьет на творчество и вам понадобится HEAD-запрос!

**Полезные HEAD-запросы**

Одной из ситуаций, когда HEAD-запрос может оказаться полезным, является необходимость узнать длину содержимого, или даже его тип. Это позволяет определить, будет ли возвращен процессу большой объем данных, или сервер будет пытаться возвратить двоичные файлы вместо HTML, текстовых данных или XML (которые намного проще обработать в JavaScript, чем двоичные данные).  
  
В таких случаях вы просто используете соответствующее имя заголовка и передаете его в метод getResponseHeader() объекта XMLHttpRequest. То есть, для получения длины ответа вызовите request.getResponseHeader("Content-Length");. Для получения типа содержимого используйте request.getResponseHeader("Content-Type");.  
  
Во многих приложениях выполнение HEAD-запросов не добавляет функциональности и может даже замедлить запрос (путем передачи HEAD-запроса для получения информации об ответе и последующий GET или POST-запрос для реального получения данных). Однако если вы не уверены в сценарии или серверном компоненте, HEAD-запрос может помочь вам получить некоторые основные данные без работы с данными ответа или без снижения пропускной способности для передачи этого ответа

**Лабораторна робота №12. Знайомство з технологією AJAX**

**Завдання**

1) Створити текстові файлы 1.txt та 2.txt Файл 1.txt містить назви фруктів в форматі JSON Файл 2.txt містить назви овочів в форматі JSON

2) Створити файл index.html, який містить JavaScript та два посилання: "фрукти" і "овочі". Натискання на посилання "фрукти" мусить завантажувати на поточну сторінку вміст файла 1.txt, і натискання на посилання "овочі" мусить завантажувати на поточну сторінку вміст файла 2.txt.

Веб-сторінка має такий вигляд:

[фрукти](http://sites.znu.edu.ua/webdevelopment/labs/1414.ukr.html) [овочі](http://sites.znu.edu.ua/webdevelopment/labs/1414.ukr.html)

Після натискання на ссылку \_\_фрукти\_\_ в зелному прямокутнику має з'явитись перелік фруктів із файла 1.txt, а після натискання на посилання \_\_овочі\_\_ в зеленому прямокутнику має з'явитись перелік овочів із файла 2.txt.

\*\*УВАГА\*\* змінюватись мусить лише вміст зеленого прямокутника. Всі інші частини сторінки залишаються незмінними.

Приклад

<script type="application/javascript">

function loadJSON()

{

var data\_file = "1.txt";

var http\_request = new XMLHttpRequest();

try{

// Opera 8.0+, Firefox, Chrome, Safari

http\_request = new XMLHttpRequest();

}catch (e){

// Internet Explorer Browsers

try{

http\_request = new ActiveXObject("Msxml2.XMLHTTP");

}catch (e) {

try{

http\_request = new ActiveXObject("Microsoft.XMLHTTP");

}catch (e){

// Something went wrong

alert("Your browser broke!");

return false;

}

}

}

http\_request.onreadystatechange = function(){

if (http\_request.readyState == 4 )

{

// Javascript function JSON.parse to parse JSON data

//var jsonObj = JSON.parse(http\_request.responseText);

// document.getElementById("Name").innerHTML = jsonObj.name;

//Javascript function JSON to not JSON data

document.getElementById("Name").innerHTML=http\_request.responseText;

}

}

http\_request.open("GET", data\_file, true);

http\_request.send();

}

</script>

<div id="Name">Цветы по англ</div>

<div class="central">

<button type="button" onclick="loadJSON()">Load </button>

1.txt

rose violet