Статья, по которой делался доклад на PHPConf 2006, Version 1.0

Дополнения, комментарии и отзывы welcome.

Илья Кантор, 2006
ICQ 820317
ilia [at] manual.ru

1 Оглавление

2 Основы AJAX

2.1 Введение в AJAX

Интернет начинался как коллекция статических страниц, предоставляющих информацию. Соответственно, и используемый протокол (HTTP) тоже выбран достаточно простым.

Классическое веб-приложение так и работает: пользователь получает страницу, переходит по ссылке на другую. При этом состояние приложения может храниться в куках, сессии и т.п. Каждое действие пользователя, даже если оно затрагивает небольшую часть интерфейса, приводит к полной перегрузке страницы.

Основной смысл AJAX заключается в возможности взаимодействовать с сервером без загрузки новой страницы. Это по сути небольшое изменение открывает массу новых возможностей и сложностей.

2.2 Пример

2.2.1 Gmail

- 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 остальное, при желании, дополняется автоматически, как в десктоп-приложениях.

Результат: обширная популярность, высочайший спрос на account’ы.

2.3 Определение AJAX

Asynchronous **J**avaScript **A**nd **X**ML, или, сокращенно, AJAX - технология веб-разработки для создания интерактивных интернет-приложений. Смысл в том, что время отклика уменьшается, так как происходит скрытый небольшой обмен данными с сервером, и не требуется перезагружать веб-страницу при каждом изменении со стороны пользователя.

Технология AJAX использует комбинацию:

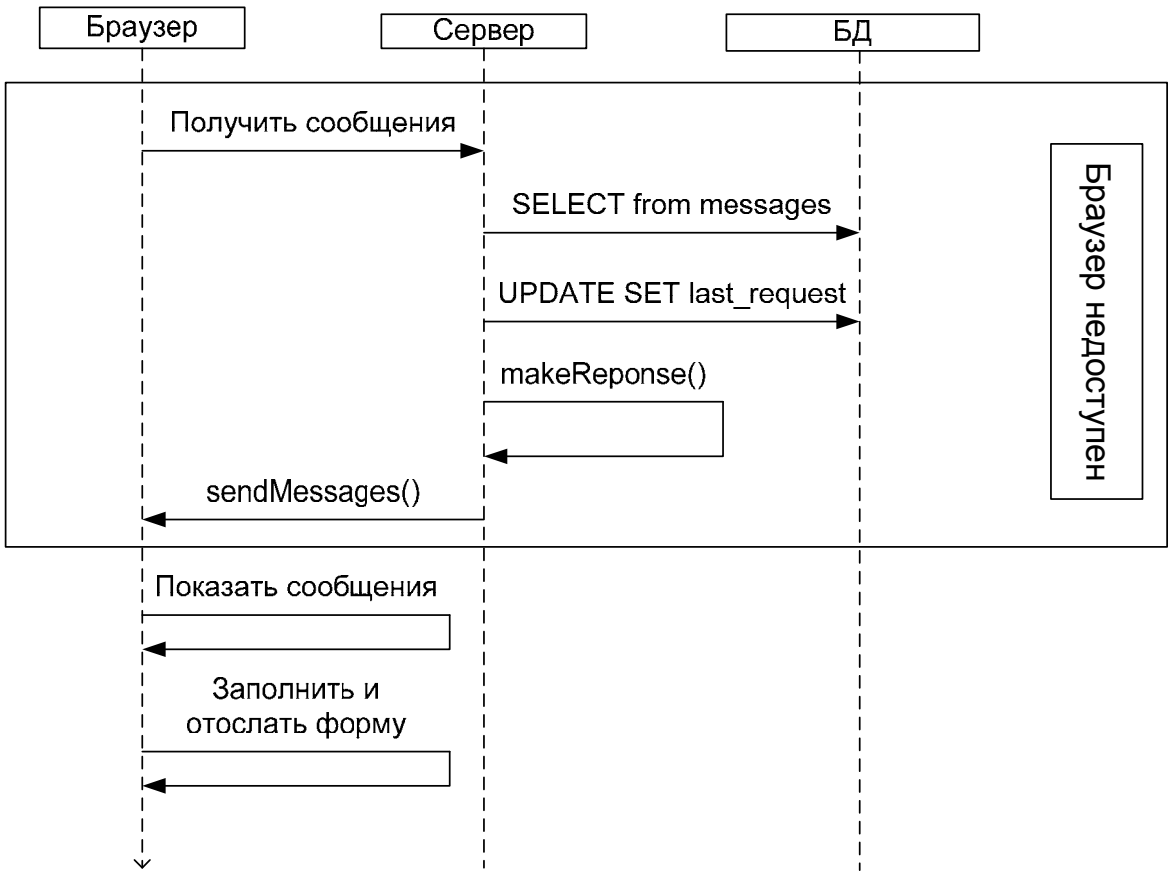
- (X)HTML, CSS для подачи и стилизации информации
- DOM-модель, операции над которой производятся на стороне клиента (особенно при помощи реализаций ECMAScript таких как JavaScript и Jscript), чтобы обеспечить динамическое отображение и взаимодействие с информацией
- XMLHttpRequest для асинхронного обмена данными с веб-сервером. В некоторых AJAX-фреймворках и в некоторых ситуациях, вместо XMLHttpRequest используется IFrame, SCRIPT-тег или другой аналогичный *транспорт*.

- XML часто используется для обмена данными, однако любой формат подойдет, включая форматированный HTML, текст, JSON и даже EBML

Таким образом, AJAX-приложение состоит как бы из двух частей. Первая выполняется в браузере и написана, как правило, на JavaScript, а вторая – на сервере, например, на PHP ;).
Между этими двумя частями происходит обмен данными через XMLHttpRequest(или др. транспорт).

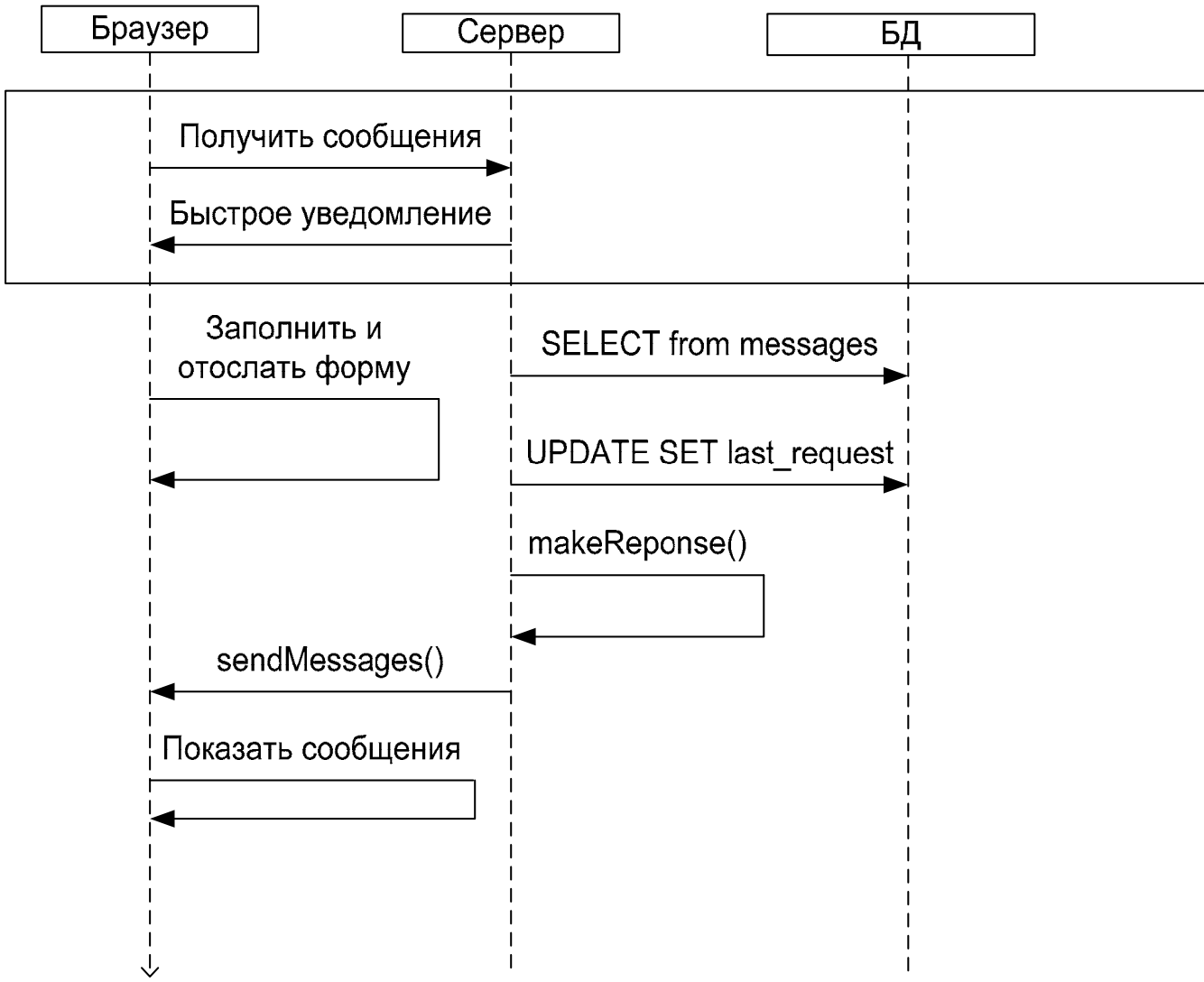
2.3.1 Асинхронная коммуникация

Как правило, веб-приложения работают синхронно.



В синхронной модели браузер отправляет запрос на сервер, ждет пока тот совершит всю необходимую работу и перешлет ответ, затем вызывает функцию показа.
Все сетевые задержки включены во время ожидания, обозначенное серым цветом.
Пользователь не может заниматься чем-то другим на этой же странице, пока происходит синхронный обмен данными.

В противоположность такой модели выступает *асинхронная коммуникация*.



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.

С другой стороны, приложение становится гораздо более чувствительным 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.

2.4 Rich Client в Web

При помощи AJAX-технологий становится возможным создание эффективного “богатого клиента”, *Rich Client.* Смысл этого словосочетания достаточно прост.

- Слово *Rich* (богатый) – относится к модели взаимодействия. Rich Client поддерживает множество методов ввода-вывода. Он организуется так, чтобы работа была максимально понятной и интуитивно очевидной.
  Как правило, для этого используется набор стандартных инструментов: меню, драг-н-дроп, работа с выделением и т.п.

- *Client* – означает, что приложение является клиентом, т.е отделено от сервера. В нашем случае это означает, что клиент исполняется на браузере и взаимодействует с сервером при помощи технологии AJAX.

На организацию Rich Client в интернет дополнительно накладываются серьезные ограничения.

2.4.1 Сетевые задержки, проблемы связи

Очевидно, они требуют внутренней обработки ошибок коммуникации и лагов. Есть и еще одно, более важное, следствие для интерфейса.

Хороший интерфейс должен отвечать на интуитивно понятном уровне. Как правило, это реализуется через моделирование событий реального мира, когда пользователь “нажимает”, “открывает” или “перелистывает” что-либо. При этом реакция на действие видна тотчас же.
В десктоп-приложениях этот принцип соблюдается.
Веб-интерфейс должен учитывать задержки, ошибки и делать по возможности интуитивно понятными и их тоже.

2.4.2 Ограниченность ресурсов, ошибки и несовместимость браузеров

JavaScript – далеко не самая производительная платформа. Ее реализации различаются от браузера к браузеру, имеют свои особенности и ошибки.

Ряд серьезных ошибок, к сожалению, имеет статус “won’t fix”. То есть, разработчики знают о них, но даже не планируют исправлять. Как пример, отметим многочисленные утечки памяти.

Кросс-браузерные несовместимости, как правило, решаются созданием специализированных библиотек. Однако, такие библиотеки занимают место и снижают производительность, поэтому их часто не используют для простых приложений.

3 Внедрение AJAX

3.1 Что я могу сделать с помощью AJAX ?

- ➔ Кнопка “Подписаться”, “Голосовать”, при нажатии на которую название на кнопке почти сразу меняется на “Отписаться”, “Спасибо, Ваш голос принят”
- ➔ Дерево разделов сайта, которое подгружает узлы с сервера по мере того, как пользователь их открывает.
- ➔ Кросс-браузерный чат… Да, это тоже реализуется с помощью AJAX-технологий..
- ➔ И многое, многое другое…

3.2 Начальный уровень

Используется AJAX-библиотека, основанная на PHP, которая предоставляет возможность добавить сервисы без особой модификации и усложнения HTML-кода.

Со стороны разработчика, для пересылки используется готовый HTML, а библиотека сама формирует JavaScript, который будет действовать на клиенте.

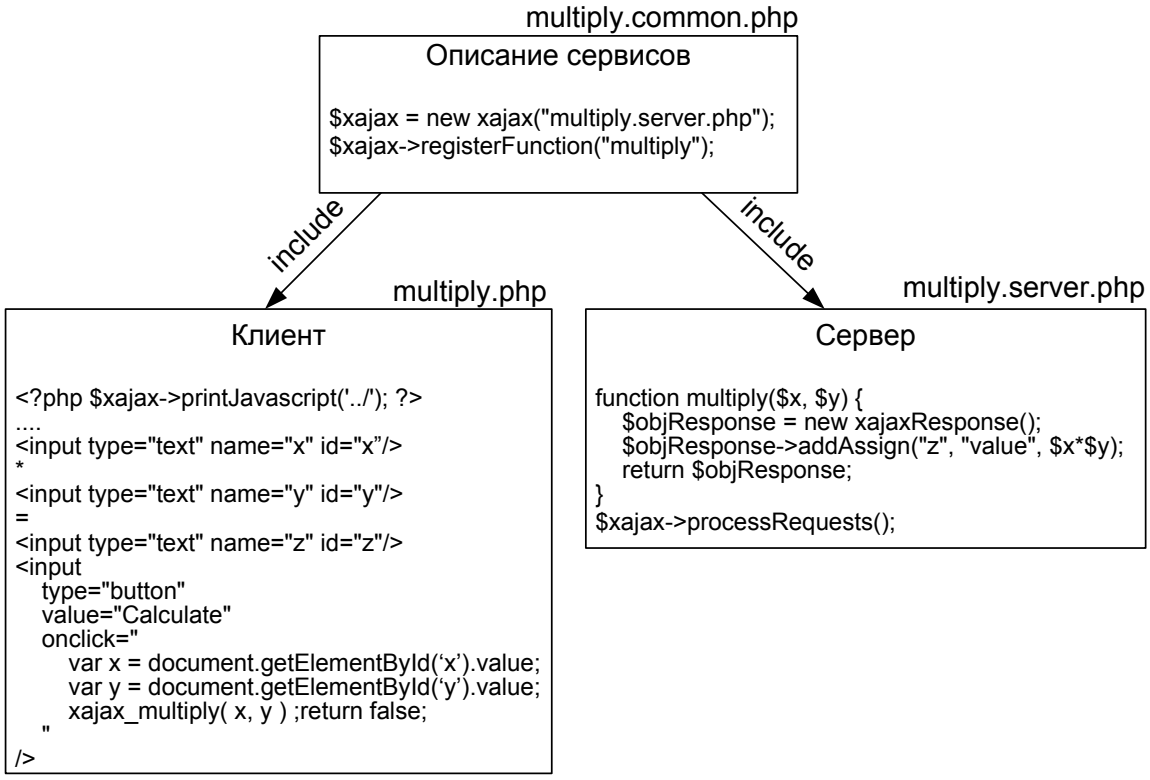
3.2.1 Библиотека XAJAX

Существует достаточно много библиотек начального уровня для AJAX. Многое в них совпадает.

XAJAX – просто одна из них, достаточно функциональная:

- Не требует(хотя допускает) JavaScript на клиенте, все задается через PHP
- Умеет работать с формами и массивами
- Кросс-браузерный
- Объектно-ориентированный
- LGPL

3.2.2 AJAX-умножение двух полей в форме с показом результата в третьем



В общем файле multiply.common.php регистрируется URL сервера, задаются параметры XAJAX и функции, доступные для удаленного вызова. Эта информация нужна и на клиенте и на сервере.

Все запросы идут на сервер multiply.server.php. Каждая функция должна возвратить объект типа xajaxResponse, который задает различные манипуляции с вызвавшей страницей

Некоторые методы xajaxResponse:

- addAssign(<id элемента>, <свойство>, <значение>) - простое присвоение значения свойству
- addAppend / addPrepend / addReplace аналогично задают другие операции
- addScript(<javascript code>) добавляют в ответ JavaScript, который будет выполнен на клиенте

Клиент включает в себя JavaScript, сгенерированный XAJAX по описанию сервисов и вызовы соответствующих функций с префиксом. В примере это – xajax_multiply(x,y).
В xajax.js также входит (небольшое) количество дополнительных функций.

Некоторые методы объекта xajax

- getFormValues(<id формы> [, <игнорировать disabled> [, <префикс>]]) сериализует форму для отправки на сервер, позволяя игнорировать отключенные элементы и включать только элементы с заданным префиксом.
- loadingFunction / doneLoadingFunction - этим свойствам можно присвоить указатель на функцию, которая будет вызвана в начале/конце асинхронной загрузки.

3.3 Средний уровень. Элементы Rich Client

Наряду с AJAX требуется некоторый набор JavaScript-компонент, которые должны быть минимальны по размеру – т.к их будет загружать каждый посетитель.

Для этого может быть использована, например, X library http://cross-browser.com/

В ней есть довольно обширный набор кросс-браузерных компонент.
Кроме того, пакет включает в себя систему сборки и зависимости на уровне объектов и функций, так что единый JavaScript-файл со всеми компонентами для страницы, скорее всего, не будет включать много лишнего.

3.4 Высокий уровень интерактивности. Сложный клиент

Для страниц со сложным функционалом на стороне клиента, подход совсем другой. Можно использовать сложный AJAX-фреймворк, активно использующий JS на стороне клиента. При этом серверная часть уже не занимается задачей формирования JS, она просто предоставляет вебсервисы и возвращает результат в нужном формате.

Здесь уже требуется более сложная архитектура, профессиональное JavaScript – программирование и т.п.

3.4.1 Четыре базовых принципа

Успешное использование AJAX, как правило, следует нескольким важным принципам.

1. **Браузер не показывает документ, а выполняет приложение.**

   В классических веб-приложениях сервер хранит все необходимые данные для генерации страниц. В ответ на действия пользователя он, учитывая сессию, создает новую страницу. Браузер выбрасывает старый документ и показывает новый.
   Роль браузера проста и незатейлива.

   Однако, занесение товара в корзину покупателя вряд ли изменит список категорий и продуктов. Поэтому можно при помощи AJAX отправить информацию о добавленном продукте на сервер. Например,
   ```
   <action>addToChart</action>
   <productId>151</action>
   ```

   Если такой товар может быть добавлен, то сервер ответит
   ```
   <status>OK</status>
   ```
   и браузер (а, точнее, наше JavaScript-приложение) самостоятельно обновит корзину на экране.

   Информация о текущем состоянии корзины может, вообще, отсутствовать на сервере и храниться только на клиенте.
   Таким образом, виден побочный эффект – уменьшение нагрузки на сервер, т.к часть ее берет на себя “умный” клиент, приложение на стороне браузера.

2. **Сервер посылает данные, а не документ**

   Как видно из примера выше, браузер получает с сервера совсем не HTML документ. Со своей стороны он отправляет специальный запрос, а со стороны сервера получает стандартный ответ, который использует для обновления информации.

   Вообще говоря, AJAX может использоваться и для пересылки готового HTML. Такой подход, как правило, применяется либо для облегчения клиентской части, либо для небольшого внедрения AJAX в классическое приложение.
   Большинство разработчиков соглашаются, что чем сложнее клиентское приложение, тем выгоднее обмениваться данными вместо документов.

3. **Возможно непрерывное взаимодействие с пользователем**

   При работе синхронного приложения пользователь запрашивает данные с сервера и оказывается в своего рода “лимбо” – нечего делать, пока создается и загружается новая страница

   В AJAX-корзине пользователь может добавлять все новые товары, открывать новые категории дерева товаров непрерывно.

   Паузы в работе приложения отвлекают мысли пользователя, особенно начиная с некоторой границы, которую разные исследователи трактуют по-разному. Их отсутствие открывает новый уровень интерактивности в работе с веб-приложениями.

4. **Сложные AJAX-приложения требуют грамотного программирования и подхода**

   Роль JavaScript в классических веб-приложениях невелика. Как правило, она сводится к созданию различных эффектов и оригинальных меню.
   Такое программирование, как правило, не требует ни глубоких знаний, ни ООП и принесло JavaScript(незаслуженную) репутацию непонятного и сложного для освоения языка.

   С другой стороны, AJAX-приложение, нацеленное на долгую работу в браузере, должно иметь расширяемую базу, грамотную архитектуру и учитывать особенности среды, в которой работает: быстродействие, управление памятью и т.п.

4 Способы общения с сервером

4.1 Транспорты

4.1.1 XMLHttpRequest

Используется специальный объект XMLHttpRequest с методами

- `send(данные)` - отсылает данные
- `abort()` – останавливает текущий запрос
- `setRequestHeader/getResponseHeader` манипуляция заголовками
- `open(метод, url, флаг асинхронного запроса...)` открывает запрос

Основные свойства:

- `onreadystatechangeEvent` – обработчик, который вызывается при каждом изменении статуса обработки запроса
- `readyStateObject` – текущий статус запроса: `uninitialized/loading/loaded/interactive/complete`
- `responseText – строка с данными, которые вернул сервер`
- `responseXML – DOM полученных данных, если они представляют собой XML`
- `statusNumeric` – код ответа сервера: 404/200/…
- `statusText – расшифровка кода ответа: “NOT FOUND”, “OK”…`

Вот пример небольшой функции, которая делает запросы.

```
// Получить объект XMLHttpRequest для запроса (кросс-браузерно)
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;
}

// Получить данные с url и вызывать cb – коллбэк с ответом сервера
function getUrl(url, cb) {
  var xmlhttp = getXmlHttp();
  // IE кэширует XMLHttpRequest запросы, так что добавляем случайный параметр
к URL
  xmlhttp.open("GET", url+'?r='+Math.random());
  xmlhttp.onreadystatechange = function() {
    if (xmlhttp.readyState == 4) {
      cb(xmlhttp.status,
         xmlhttp.getAllResponseHeaders(),
         xmlhttp.responseText);
      }
  }
  xmlhttp.send(null);
}
```

- Два вида передачи: синхронный и асинхронный
  - синхронный способ подвешивает браузер
- Можно проставлять заголовки через XMLHttpRequest.setRequestHeaders и делать различные типы запросов. Таким образом можно, например, работать с SOAP-сервисами.
- Невозможна отправка файлов, выбранных в форме
  - однако, можно отсылать документы из браузера в виде файлов, составляя POST-запрос вручную
- В IE до 7.0 необходим включенный ActiveX.
- Ограничение в 2 потока

Тонкости реализации:

В связке XMLHttpRequest <-> JS – при событиях происходят утечки памяти из-за циклических ссылок. Разрушить ссылки методами JS не удается, поэтому безопасная реализация – отправить запрос и по setTimeout отслеживать readyState.

4.1.2 IFrame

Динамически создается один (или два, но не больше из-за ограничения HTTP 1.1) невидимый IFrame.

- По умолчанию однопоточный транспорт, хотя можно распределять запросы между ифреймами

- Возможна отправка файлов пользователя с диска через <input type=file>

- Звук(клик) при сабмите в IE: событие “start navigation”. Особенно видна эта проблема при частом (раз в 5 сек) поллинге-рефреше ифрейма.
  Обычные сабмиты и переходы по ссылке внутрь iframe добавляют ненужную запись в document.history и таким образом влияют на кнопку Back.

  Эти проблему решает
  1. Динамическая генерация IFrame через DOM
  2. Использование iframe.contentDocument.location.replace(src) для загрузки новых данных.

  При сабмите форм внутрь ифрейма такой способ, увы, не работает.

4.1.3 Script

К странице динамически добавляются теги SCRIPT, данные пересылаются в URL. Например[1],

```
<script src="http://site.com/service.php?name=data"></script>
```

Браузер тут же обрабатывает тег: запрашивает скрипт с заданного URL и выполняет его.

Как основная программа может получить данные из скрипта?

4.1.3.1 Передача данных через простое присваивание

В процессе выполнения скрипт присваивает данные некоторой переменной.
`data=7`
Наш обработчик по setTimeout проверяет каждые 0.01 сек, не появилась ли такая переменная.
Если появилась, то работа закончена, тег <script> удаляется из документа.

4.1.3.2 Передача данных через callback

[1] Для краткости опущен обязательный атрибут type=”text/javascript”

В запросе указывается коллбэк для вызова.
```
<script src="http://site.com/service.php?callback=done&name=data"></script>
```

Полученный скрипт в конце работы запускает callback с полученными данными:
```
done(7)
```

В практической реализации транспорта, как правило, в скрипт передается специальная функция транспорта:
```
<script src="http://site.com/service.php?callback=sciptload[123]&name=data"></script>
```

Так что вызывается служебная функция `sciptload[123](7)`, обрабатывает конец загрузки, после чего вызывает `done(7)`

В примерах число 7 означает текстовую запись JavaScript-объекта (JSON).
Второй способ получил название JSON with Padding: JSONP.

4.1.3.3 Multipart-отправка данных

Длина URL в GET-запросах ограничена. Она зависит от браузера-сервера, приблизительно 1kb – безопасное значение. Поэтому для отправки на сервер бОльших фрагментов используется multipart-протокол. Каждый запрос снабжается двумя параметрами: уникальным ID **dsrid** и номером части **part.**
Первый элемент multipart-запроса имеет **part=1,** последний - не имеет параметра **part.**

+ Запросы на другие домены
- Ошибки коммуникации отслеживаются только через timeout
- Только GET

4.2 Cross-site scripting

Кросс-сайтовый скриптинг общее название для случая, когда страницы с одного домена производят запрос на другой.

***site1.net* делает запрос на *site2.net* – в этом случае домены совершенно разные**.

Документы, полученные в одном фрейме с *site1.net* не смогут обращаться к другому через JS, если он с *site2.net*. Вообще, документы с разных доменов, протоколов или с разных портов(кроме IE) одного домена не могут(точнее, почти не могут, см. например http://msdn.microsoft.com/workshop/author/om/xframe_scripting_security.asp) общаться друг с другом в модели безопасности JavaScript.

Единственный транспорт, позволяет такие запросы – это SCRIPT. Но он, естественно, накладывает ограничения на заголовки, метод и формат ответа.
Скажем, XML-данные так получить нельзя.

Как правило, сервисы, в которых нужны такие запросы, делаются через reverse proxy. То есть, *site1.net* делает специальный запрос, например, на особенный URL *http://site1.net/site2/*, а сервер *site1.net* проксирует его(например, через mod_proxy) на *site2.net.*

***domain1.site.*net делает запрос *на domain2.site.net* – есть общий наддомен.**

Основной вопрос – зачем такое вообще может понадобиться?

Первый сценарий – наш сайт *domain.site.net* является частью некой системы, в которой адрес news.site.net предоставляет ленту новостей, *goods.site.net* – товары,
и т.п., так что мы, имея такой домен, можем с удобством пользоваться этими вебсервисами.

Второй сценарий - оба таких сайта находятся под нашим контролем… Скажем, *www.site.net* и *site.net* формально на разных доменах, но могут одинаково обрабатываться сервером.
В таком случае все необходимые запросы можно сделать и на текущий домен – они так и так попадут к нам на сервер.
Но здесь появляется второе применение кросс-сайтового скриптинга. А именно, обход ограничения HTTP 1.1 на соединения: не более двух одновременных запросов к серверу(домену).

Не так давно мне пришлось писать AJAX-компонент, который делает запросы к нескольким вебсервисам, причем время отклика может варьироваться (в зависимости от запроса) между 1-20 сек.

При этом одно соединение было постоянно занято подгрузкой бесконечного ифрейма с сервера, через которое поступают обновления (push данных со стороны сервера в виде <script>-тагов).
Оставался один канал на все про все – явно недостаточно для асинхронных запросов.

Также возможны случаи, когда нужно поддерживать несколько push-каналов, например, дополнительно открыто окно мини-чата, который реализован отдельно от общих обновлений.

Кросс-сайтовый скриптинг позволяет обойти лимит путем использования нескольких доменов. Бесконечный ифрейм подгружаем с *updates.site.net*, чат работает на *chat.site.net*, и свободные основные 2 канала для запросов с *site.net*.

4.2.1 Кросс-скриптинг с общим наддоменом

Яваскрипт из одного фрейма может как-то вызывать другой фрейм только если они с одного сайта.
Но у документа есть еще свойство document.domain, которое можно менять. Так что если два фрейма имеют один document.domain , то они могут делать друг с другом что угодно.

Конечно, есть ограничения безопасности…
Document.domain можно присваивать:

- на текущий домен,
- наддомен не меньше уровня сайта (т.е *job.site.com.ua* можно поменять на *site.com.ua*, но не на *com.ua*),
- IE допускает обратное переименование с наддомена обратно на полный домен.

Так что в случае разных сайтов на одном общем домене можно присвоить свойству document.domain этот общий домен, и тогда кросс-сайтовый скриптинг между этими двумя сайтами возможен.

Перед следующим запросом в IE нужно вернуть document.domain обратно. В Opera/FF это невозможно (домен может быть установлен только в текущий или в домен высшего уровня), но в Opera 9/FF 1.5 ситуация поправлена тем, что повторные запросы с исходного домена **разрешаются**.

Итак, алгоритм для IE/FF 1.5/Opera 9

1. Создать IFrame с другим поддоменом
2. Сделать запрос с этого IFrame
3. Установить document.domain такой же как у фрейма, куда надо отправить информацию и вызвать нужные функции
4. Для IE – вернуть document.domain обратно перед следующим запросом

4.2.2 test.html

```
<html>
  <head>
  <script type="text/javascript">
   <!-- обрезаем домен текущей страницы до базового -->
   document.domain="tmp.x";
   function gotTime(result) { document.getElementById('time').innerHTML =
result }
  </script>
  </head>
  <body>
    Счетчик
    <div id="time"></div>
    <!-- iframe с другого домена -->
    <iframe src="http://www.tmp.x/crossdomain/test-iframe.html"></iframe>
  </body>
</html>
```

4.2.3 test-iframe.html

```
<html>
  <head>
  <script type="text/javascript" src="xmlhttp.js"></script>
  <script type="text/javascript">
    var AJAX_URL="http://www.tmp.x/crossdomain/time.php";
    function gotResult(status, headers, result) {
      var oldDomain = document.domain;
      document.domain = "tmp.x";
      window.parent.gotTime(result);

      try {
        // для IE, в остальных браузерах ошибка...
        document.domain = oldDomain;
       } catch(e) {  /* ... но там это не нужно */ }

      setTimeout(getTime, 1000);
    }
    function getTime(){
      getUrl(AJAX_URL, gotResult);
    }
    getTime();
  </script>
  </head>
```

```
</html>
```

4.2.4 time.php

```
<?php echo time();
```

4.2.5 xmlhttp.js

см. описание XMLHttpRequest

Пользователи Opera/FF, как правило, быстро обновляют свои браузеры, но тем не менее.
Решение в Opera 8: cross-domain messaging
http://virtuelvis.com/archives/2005/12/cross-document-messaging
Решение в FF 1.0 – промежуточный IFrame:
http://fettig.net/weblog/2005/11/28/how-to-make-xmlhttprequest-connections-to-another-server-in-your-domain/
http://fettig.net/weblog/2005/11/30/xmlhttprequest-subdomain-update/

4.3 Передача данных со стороны сервера (server push)

Наиболее распространенная реализация AJAX – это запросы со стороны клиента. Браузер инициирует запросы и посылает их серверу. На этом коммуникация заканчивается до следующего запроса.

Однако, современные веб-приложения являются мультипользовательскими. Данные могут обновляться независимо от клиента, который что-то набирает в браузере.
Это происходит при редактировании материалов сайта, просмотре форума и т.п.
Самый показательный случай здесь, пожалуй, чат.

К сожалению, широко поддерживаемые протоколы (HTTP, FTP…) не предоставляют эффективные способы “подписки” на события с сервера, но некоторые решения, все же, существуют.

4.3.1 Частые опросы сервера (поллинг)

Первое решение, которое приходит в голову – это “поллинг” (polling), т.е периодический опрос сервера стандартными пакетами: “я тут, изменилось ли что-нибудь?”

В ответ сервер во-первых помечает у себя, что клиент онлайн, а во-вторых посылает датаграмму, в которой в специальном формате содержится весь пакет событий, накопившихся к данному моменту.

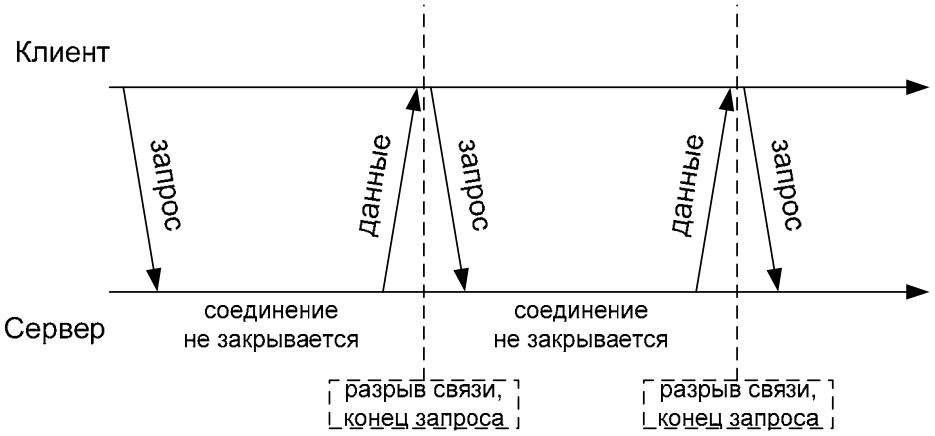
У этого способа есть одна большая проблема, а именно - большие задержки между созданием и получением данных. Сервер отсылает их не тогда, когда они появились, а когда настанет время очередного запроса.

Задержка = время между опросами + установление соединения + передача данных.

4.3.2 Очередь ожидающих запросов

Уходит запрос на сервер, и соединение не закрывается, пока ответ не придет. После прихода ответа сразу же отсылается новый запрос.
Каждый пакет данных, таким образом, означает новое соединение, которое будет открыто столько, сколько нужно, пока сервер не решит прислать информацию

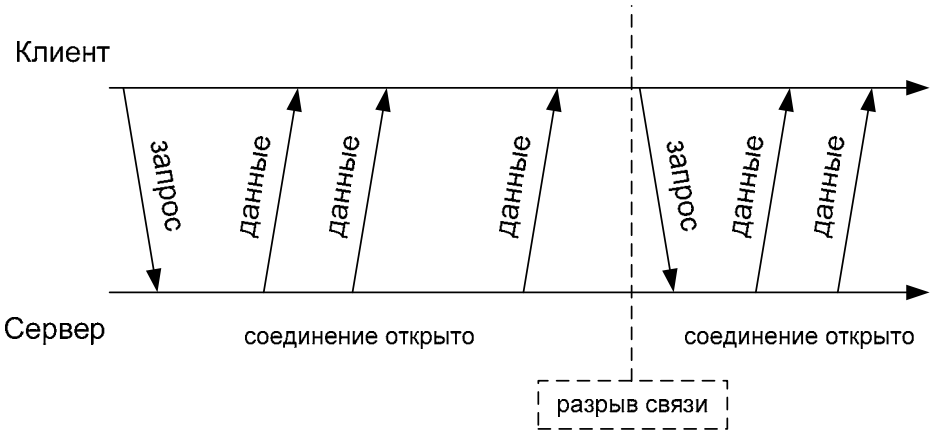
- Передача данных закрывает соединение
- Клиент открывает соединение заново для каждого пакета данных



4.3.3 Бесконечный IFrame

Способ основан на том, что браузер обрабатывает страницу последовательно и обрабатывает все новые теги по мере того, как сервер их присылает.

Клиент открывает IFrame, в который сервер присылает <script>-блоки, оставляя соединение открытым. JavaScript тут же выполняется браузером и вызывает функцию на родительской странице, которая обрабатывает событие.
Таким образом, получается “бесконечная” страница, через которую сервер присылает все новые данные. Этот способ самый известный и кросс-браузерный.



- Соединение всегда открыто, завершается только при ошибках или время от времени для очистки IFrame'а от принятых объектов.
- Данные передаются в <script>-конвертах

Проблема – при такой бесконечной передаче данных постоянно активирован индикатор прогресса. В Firefox – это просто надпись в строке статуса, В IE – анимированный индикатор прогресса внизу и переливающийся при значок вверху.

Для IE общая схема решения (применяется Google) использует ActiveX. Вместо создания IFrame непосредственно на странице, он добавляется к специальному объекту htmlfile (свойства см. тут: http://msdn.microsoft.com/workshop/browser/mshtml/reference/ifaces/document2/document2.asp)

```
// текущий документ с child.example.com,
// но document.domain=example.com для общения с фреймами
var currentDomain = "http://example.com/";
var dataStreamUrl = currentDomain+"path/to/server.php";
var transferDoc = new ActiveXObject("htmlfile");
// тут проверить, можно ли использовать transferDoc
transferDoc.open();
transferDoc.write("<html>");
transferDoc.write("<script>document.domain='"+currentDomain+"';</script>");
transferDoc.write("</html>");
transferDoc.close();
// создать ифрейм, через который будут производиться связь с сервером
var ifrDiv = transferDoc.createElement("div");
transferDoc.appendChild(ifrDiv);
// открыть соединение
ifrDiv.innerHTML = "<iframe src='"+dataStreamUrl+"'></iframe>";
```

Чтобы обратиться из ифрейма к объекту вне htmlfile, нужно установить ссылку через свойство parentWindow:

```
function foo() {…}
transferDoc = new ActiveXObject(”htmlfile”);
transferDoc.parentWindow.foo = foo; // установить ссылку
// вызов изнутри ифрейма
parent.foo();
```

В Mozilla такая схема не работает, но и сообщение далеко не такое отвлекающее, статическая надпись в адресной строке.
Там применимо другое решение. У XMLHttpRequest есть readyState=interactive, которое наступает каждый раз после получения очередного пакета данных. Полученный пакет можно получить из responseText , отделив от предыдущего пакета через substr.
Время от времени, конечно же, приходится повторять запрос, чтобы очистить память (gmail).

4.3.4 XMLHttpRequest с флагом multipart

В Mozilla есть возможность устанавливать флаг multipart у XMLHttpRequest. Таким образом, ответ сервера может состоять из нескольких частей, при получении каждой вызывается onload().
Учебник по данному вопросу находится здесь
http://www.xulplanet.com/tutorials/mozsdk/serverpush.php

Теоретически способ самый правильный и хороший, однако на текущий момент он отсутствует в других браузерах (Opera, Safari, не уверен насчет новых IE).

4.3.5 Частый опрос против длинных соединений

Всегда ли длинное соединение лучше частых опросов?
Решение с длинными соединениями с виду оптимальнее, но гораздо сложнее и обладает рядом особенностей.

1. Реализация длинного коннекта, как правило, усложняет архитектуру. Возможно, можно обойтись решением проще?

2. Большинство веб-серверов плохо оптимизированы под большое количество длинных соединений. Как правило, используются потоки или процессы, которые отъедают фиксированное количество ресурсов и не освобождают их до конца соединения.

   На уровне OS проблема решается использованием kqueue(FreeBSD) или epoll(Linux).
   На уровне веб-сервера можно использовать
   1. Apache MPM event для apache 2.2 (экспериментальный и ограниченный MPM, специальный поток обрабатывает Listening и Keep-Alive сокеты)
   2. Jetty (Java) / Twisted(Python), nginx и другие специализированные серверы

   Потянет ли текущая серверная архитектура длинные соединения? Здесь, конечно, ответ становится неочевидным лишь с сотен, а то и тысяч соединений.

3. Насколько долго пользователи находятся на одной и той же странице? При переходах коннект, скорее всего, придется открывать заново в любом случае.

4. Если допустимы задержки, то, может быть, хватит частого опроса?

4.3.6 Замечания к использованию server push

1. Проблема интеграции изменений с сервера в рабочую среду не является особенностью AJAX. Посмотрите, как с этим справляются десктоп-приложения

2. Пользователи должны понимать КТО и ПОЧЕМУ изменил данные

3. Как всегда, ошибки коммуникации должны обрабатываться понятным образом

4. Сервер должен пересылать обновления данных, а не изменения функционала.

5 Форматы данных для AJAX

5.1 HTML

Сервер возвращает данные.

```
<div class="report">
        <h3>Доклад об AJAX</h3>
        <p>Рассказ о создании AJAX-приложений</p>
</div>
<div class="report">
        <h3>Доклад про ORM</h3>
        <p>Раскрыты способы задания отображений</p>
</div>
```

Клиент помещает их внутрь тега

```
domObject.innerHTML = data
```

+ Не требует постобработки
- Большой объем
- Ограниченное применение. Это часть документа, а не данные.
- Проблемы с формами, создаваемыми через innerHTML (не отсылаются, некорректно показываются и т.п.)

5.2 XML

Сервер возвращает xmlDoc

```
<reports>
        <report>
                <title>Доклад об AJAX</title>
                <description>Рассказ о создании AJAX-приложений</description>
        </report>
        <report>
                <title>Доклад про ORM</title>
                <description>Раскрыты способы задания отображений</description>
        </report>
</reports>
```

На клиенте уже хранится или передается дополнительно xslDoc:

```
<xsl:template match="/reports/report">
        <h3><xsl:value-of select="title"/></h3>
        <p><xsl:value-of select="description"/></p>
</xsl:template>
```

Клиент производит XSLT-преобразование и присваивает innerHTML

```
domObject.innerHTML = xmlDoc.transformNode(xslDoc)
```

+ Понятен для человека
* XSLT/XPath
    - поддерживается IE 6.0+, Firefox/Galeon, Opera 9.0+,
    - не поддерживается Safari/Konqueror, рядом других агентов
    - требует ActiveX в IE
    - трансформирует данные напрямую в HTML
    - хорошо работает (быстрее eval) на больших объемах данных

+ Давно известный и широко используемый формат
    - огромное количество стандартных сервисов и известных классов отдают информацию в XML-формате

- Те же проблемы с формами

5.3 JSON (JavaScript Object Notation)

Сервер возвращает JavaScript-объект

```
{"reports":[
        {"report":
                {
                "title":"Доклад об AJAX",
                "description":"Рассказ о создании AJAX-приложений"
                }
        },
        {"report":
                {
                "title":"Доклад про ORM",
                "description":" Раскрыты способы задания отображений"
                }
        }
]}
```

Клиент просто пропускает текст через встроенный парсер

```
var reports = eval(data)
```

Затем HTML может быть сформирован средствами JavaScript, собственной шаблонной системой и т.п.

```
domObject.innerHTML = JSmarty.fetch(reportsTemplate)
```

Конечно же, никто не ограничивает в передаче непосредственно инструкций JavaScript, но это может существенно усложнить приложение.

- Как правило, малопонятен для человека, сложно заметить ошибку на глаз
+ eval(), как правило, работает достаточно быстро, но его можно запускать, только если яваскрипт гарантированно безопасный.
Cross-site scripting требует специального парсинга JSON, чтобы извлечь строку данных без запуска потенциально вредного кода, и такой парсинг в любом случае гораздо медленнее, чем eval/XML.

+ Существуют парсеры/сериализаторы, для php есть pear-класс и pecl-extension.

+ Менее избыточен, по сравнению с XML

5.4 Какой формат выбирать?

- Для небольшого внедрения AJAX в существующее приложение удобен HTML-формат. Известные библиотеки (jpspan, sajax, xajax…) делают работу с ним очень простой. Кроме того, HTML не требует клиентского кода.

- XML является стандартом де-факто для множества приложений.
  Так что работать с ним, вполне возможно, придется вне зависимости от пристрастий разработчиков.

- XML может быть намного проще, чем с JavaScript-объекты, так как доступны сложные XPath-выборки.
  Для JavaScript есть небольшая, менее мощная библиотека jQuery http://jquery.com/

- JSON удобен, когда нужно переслать данные, и непосредственная переработка данных в HTML не нужна.

- JSON является “родным” форматом для JavaScript и десериализуется в объекты. Для самых разных языков есть библиотеки (де)сериализации, однако XML все же более распространен.

- При выборке-отображении большого количества данных (начиная с десятков записей) XML+XSLT работает быстрее.

В разных частях веб-приложения вполне можно использовать несколько форматов Например:

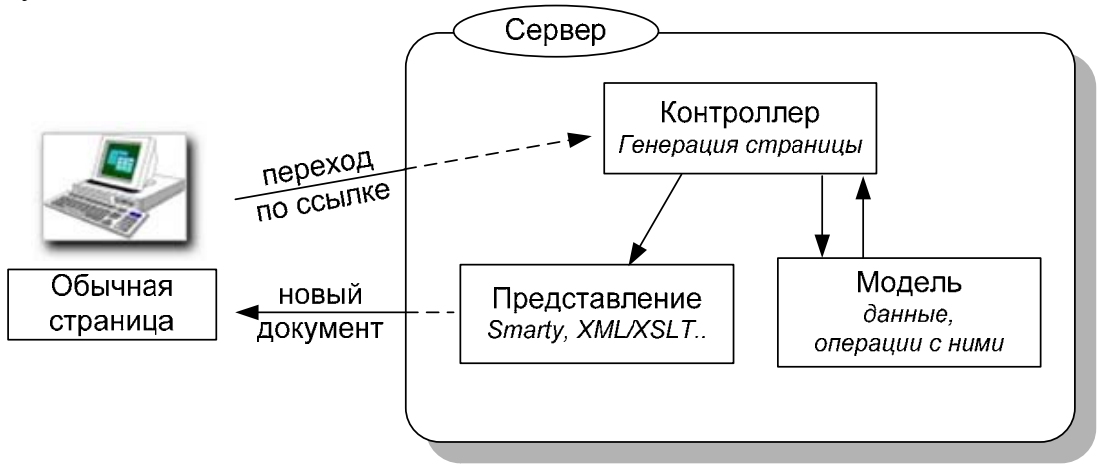
1. XML для AJAX-подгрузки ленты новостей
2. JSON для работы с узлами JavaScript-дерева
3. HTML для модулей, которые слабо адаптированы под работу в режиме сервиса

6 Паттерн MVC на клиенте и сервере

6.1.1 На сервере

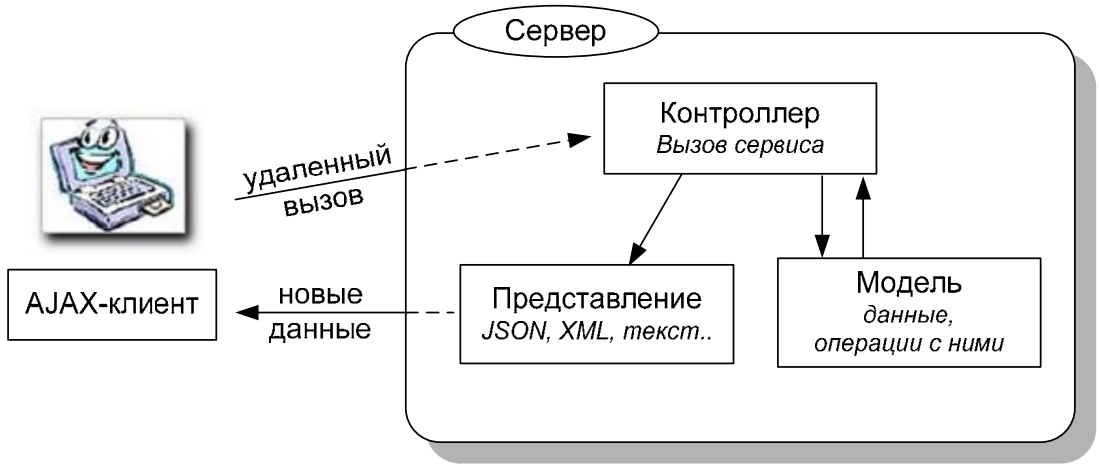
Сейчас почти повсеместно используется архитектура MVC(Model-View-Controller) на сервере. Модель(Model) представляет собой данные, отделенные от Представления (View), связь между ними осуществляет Контроллер(Controller).

Клиент отправляет запрос -> он попадает в Контроллер, который производит необходимые изменения в Модели и Представлении[2], которое пересылается в ответ клиенту.



Эта модель работает и в AJAX, но архитектура сервера становится *сервис-ориентированной*. Т.е, вместо запроса на новую страницу, браузер отправляет запросы на сервер, а в ответ получает новые данные.

[2] Это, конечно, весьма приближенное описание, разбор различных вариантов паттерна, видов его реализации и использования лежит за рамками доклада.



6.1.2 На клиенте

- **Модель**

Внутреннее представление данных: состояние корзины, ее свойства и т.п., как правило, реализуются через JavaScript-объекты.

- **Представление**

Отображение на экран происходит при помощи DOM-объектов. Различные эффекты и элементы интерфейса организуются через манипуляцию их свойствами.

- **Контроллер**

Код, который связывает модель с представлением и обрабатывает события является контроллером.

7 Введение в профессиональный JavaScript

Как правило, веб-разработчики недолюбливают JavaScript.. Вообще, для этого есть объективные причины: несовместимость браузеров, неудобная среда разработки, слабая производительность.

Однако, и на JS можно писать мощные, расширяемые программы с понятным кодом – это действительно так!

➔ Спецификация - http://www.mozilla.org/js/language/
➔ Дополнительно http://jibbering.com/faq/faq_notes/closures.html (ECMA-262 Edition 3)

7.1 ООП в JavaScript

Наследование «на прототипах» и, вообще, модель объектов в JS достаточно необычна для человека, который сталкивается с ней впервые.

В Интернете о ней можно прочитать, например, по ссылке

➔ http://google.ru/search?q=javascript+inheritance+prototype

Материалов, как это часто бывает, очень много, и разобраться в свалке, надеюсь, поможет эта глава.

7.1.1 Явное задание объектов и свойств

Базовым классом для объектов является Object.

```
/* создать новый пустой объект базового класса Object */
var objRef = new Object();
```

Объекты можно заполнять данными, не объявляя набор свойств заранее. В этом примере обратите внимание на способ доступа к свойству – через точку, если имя известно, либо через квадратные скобки, когда имя свойства хранится в переменной.

```
objRef.testNumber = 5; // не требуется объявление свойства

var propertyName = "testNumber";
objRef[propertyName] = 5; // можно и так, по строке

delete objRef["testNumber"]; // удалить свойство
```

Можно объявить объект как ассоциативный массив.

```
var objRef = {
        property_1:value_1,
        property_2:value_2,
        ...,
        property_n:value_n
}
```

Получать элементы объекта можно по ключу, а цикл по ключам записывается через конструкцию вида for(key in obj).

```
for (property in objRef) {
        alert(objRef[property]); // обработать свойство
}
```

Почти все сущности, которыми оперирует яваскрипт, являются объектами.
String, Array, Image – список можно продолжать…

Отдельное место в этом списке занимают объекты типа Function, т.е функции.
В языке яваскрипт функция – точно такой же объект, как и другие, но дополнительно обладает рядом важных возможностей.

Например, для ее задания можно использовать разный синтаксис.

```
/* создать новую функцию с аргументом a */
var CF = new Function('a', 'alert(a)');

/* запустить функцию (выведет 123) */
CF(123);

/* присвоить свойству CF.p1 значение, как и любому объекту */
```

```
CF.p1 = 5;

/* специальный синтаксис для функции */
var CF = function(a) { alert(a) }

/* и еще специальный синтаксис */
function CF(a) {
        alert(a);
}
```

Кроме того, функции можно использовать как конструкторы для создания объектов.

7.1.2 Создание объектов при помощи конструктора

Объявим конструктор или, если угодно, класс Dog. Он будет создавать объекты и устанавливать им свойство name.

```
/* объявление класса путем задания конструктора */
var Dog = function(name) {
   var wow = 'WOW!';

   this.name = name; /* name – свойство объекта, который создает функция */
   this.bark = function() { alert(wow); }
}

/* создание объекта класса */
var  pet = new Dog("Bobik");

alert (pet instanceof Dog); // верно! Действительно, объект класса Dog

alert(pet.name);  // Bobik!
pet.bark(); // WOW!
alert(pet.wow); // undefined. Переменная wow имеет область видимости – функцию Dog.
```

Как видно, при создании можно назначать свойства и методы класса. Запуск оператора new сделает новый объект и запустит конструктор Dog в его контексте (т.е this является ссылкой на новый объект).

Все свойства и методы являются общедоступными (public). Т.е, создать закрытые (private), вообще говоря, нельзя.

7.1.2.1 Как (попытаться) объявить свойство private?

Можно эмулировать закрытость свойства, объявив его через var, как это сделано с wow.

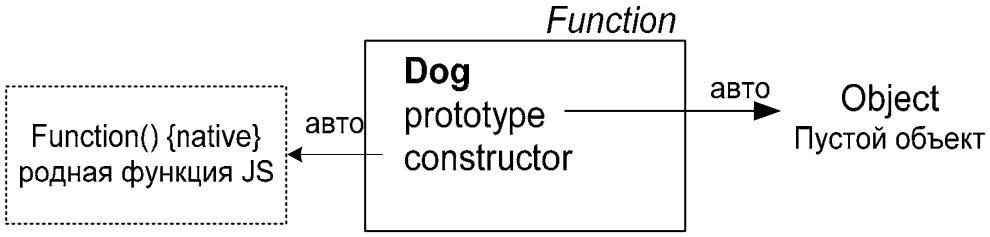
Тогда действительно, лишь функция Dog и методы, объявленные внутри нее, имеют доступ к его значению.

Однако, механизм (замыкания), через который это работает, ничего общего с ООП не имеет, и при другом способе объявлении метода может не работать.

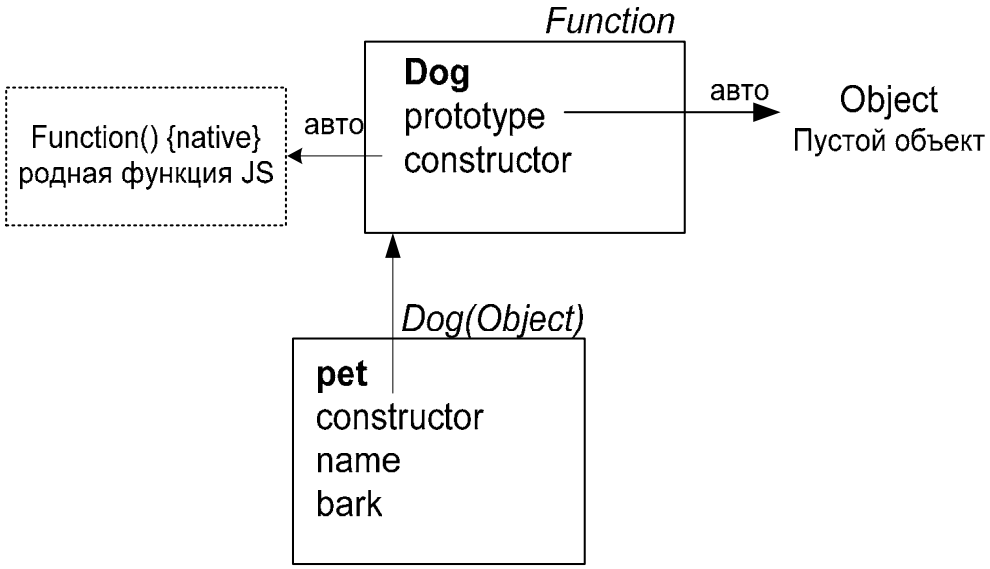
Если это не очевидно сейчас, то станет понятно после ознакомления с замыканиями и задании свойств через прототип.

При создании любого объекта автоматически создается ссылка constructor – на функцию, которая его создала.
А для Function, вдобавок – ссылка prototype на «прототип», которая является основой наследования.

При объявлении функции **Dog** получается такая конструкция:



После создания объекта **pet**:



У pet нет свойства prototype, т.к этот объект не является Function и не наследует от него!

7.1.3 Пример наследования

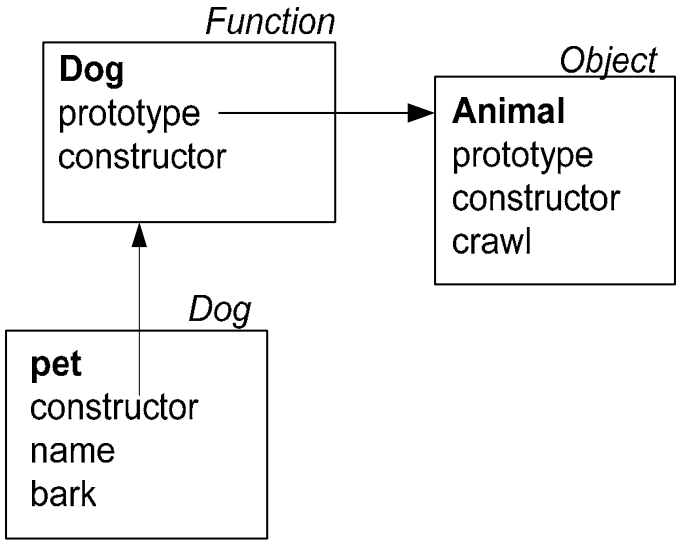
Унаследуем Dog от Animal через прототип. Для этого сначала объявим Animal, а потом укажем, что Animal является прототипом Dog.

```
var Animal = function(name) {
    // конструктор устанавливает свойство isNamed, если указано имя
    if (name) {
       this.isNamed = true;
    }
    this.crawl = function() {
       alert("I'm crawling!");
    }
}
```

```
Dog.prototype = new Animal();

/* если создать объект до установки прототипа, то структура будет не такая, как на рисунке! */
var pet = new Dog("Sharik");

pet.crawl(); // работает!
```



Если JS не может найти свойство в объекте, то он идет по цепочке constructor->prototype. Так что, просто устанавливая прототип и модифицируя его свойства, можно получить поведение, аналогичное наследованию в языках Java, C++.

7.1.4 Порядок вызова конструкторов

По правилам ООП, перед конструктором потомка Dog должен обязательно быть вызван конструктор Animal с теми же параметрами.

А в примере выше конструктор Animal вызывается лишь один раз – в
`Dog.prototype = new Animal();`

При создании объектов new Dog() вызывается функция Dog, при этом isNamed, не ставится

```
var pet = new Dog("Sharik");
alert(pet.isNamed); // undefined
```

Таким образом, наследование через установку прототипа обеспечивает лишь поиск свойств, а порядок вызова конструкторов мы должны обеспечить дополнительно, например, так:

```
/* добавляем вызов родителя */
var Dog = function(name) {
    // вызвать конструктор Animal в контексте текущего объекта с текущими аргументами
    Animal.apply(this, arguments);
    …
}
```

Теперь мы имеем один лишний вызов конструктора Animal – при указании Dog.prototype. Можно

- предусмотреть этот "лишний" запуск конструктора в родительском классе,

- отделить инициализацию от конструктора,
- просто забыть о нем

Наконец, осталось одно несоответствие – Dog.prototype.constructor должен указывать на Dog (это верно для любого объекта), а сейчас указывает на Animal.

```
Dog.prototype.constructor = Dog; // исправить неправильную ссылку
```

7.1.5 Полный пример наследования

1. Предусмотреть вызов конструктора на этапе присвоения прототипа

```
var Person = function(name, age) {
        if (!arguments.length) return; // сработает при указании прототипа (см. шаг 3)

        this.name = name;
        this.age = age;
        // private переменная будет недоступна в подклассе
        var message = "Hi, man";

        this.sayHi = function () {
                document.writeln(this.name+": " + message+"<br>");
        }
}
```

2. В конструкторе подкласса необходимо вызвать конструктор базового класса

```
var Student = function(name, age, course) {
        /* вызвать конструктор базового класса с нужными аргументами */
        Person.call(this, name, age);
        this.course = course;
}
```

3. Для правильной цепочки поиска свойств - установить общий прототип для всех объектов класса Student. Как правило, вызывается без аргументов.

```
Student.prototype = new Person(); // вызовется конструктор Person без аргументов
```

Возможность указания аргументов при указании прототипа дает дополнительную гибкость, но, как правило, не используется.

4. Для правильной работы интерпретатора JS исправляем сломанное предыдущим присваиванием свойство prototype.constructor.
   Оно должно указывать на функцию, которая создала объект.

```
Student.prototype.constructor = Student;
```

Проверка:

```
aStudent = new Student("Саша", "22", "Программирование");
aStudent.sayHi();
```

Получили наследование, которое может быть продолжено далее по цепочке.

7.1.6 Сравнение наследования в JavaScript с “обычным” наследованием

На классах(Java)	На прототипах(JavaScript)
Классы и объекты – разные сущности	**Все – объекты**
Класс задается при помощи описания, объект – при помощи функции-конструктора	**Объект задается и создается при помощи функции-конструктора, кроме того свойства задаются через прототип**
Новые объекты создаются про помощи new	**То же самое**
Иерархия создается добавлением имени класса-родителя в описание	**Двухступенчатое наследование:** **1. Задать прототип** **2. Вызвать констроктор родителя**
Свойства наследуются по цепочке классов	**Свойства наследуются по цепочке конструктор-прототип**
Класс задает свойства всех объектов данного класса. Нельзя динамически добавить свойство/метод	**Функция-конструктор и прототип задают начальные свойства/методы. Что угодно можно поменять в run-time**

Большинство отличий связаны с самими основами JavaScript

7.1.7 “Это должен знать каждый”

➔ Все, что создается – объекты первого уровня, не требующие для существования “надобъектов" (как функции в Java)

```
var a = 5; // примитивный тип
var clazz = function(arg) { this.property = arg; }
var obj = new clazz(a);
obj.method = clazz; // присвоили новый метод объекту
```

➔ Любой объект полностью динамичен, он может изменяться как угодно и когда угодно. Это относится и к свойству prototype.

```
aStudent.sayHi = function() { alert(“Howdy man!”); }
aStudent.sayHi();
```

➔ Свойства объектов доступны через точку или []

```
aStudent.sayHi === aStudent[“sayHi”]
```

➔ Оператор **new** создает объект, в контексте которого затем выполняется функция-конструктор, устанавливая private переменные через var, public – через this.

➔ Функции всегда являются замыканиями (вложенные функции/классы, и т.п.)

➔ Ключевое слово **this** относится к контексту выполнения , а НЕ к контексту объявления функции.

```
my.object.method(); // внутри method слово this будет указывать на my.object
method.apply(anObject, arguments); // выполнит method в контексте объекта anObject
```

7.2 Замыкания в JavaScript

Каждая функция выполняется в определенном контексте. Рассмотрим код:

```
// глобальный контекст
var outer = function(x) {

	// контекст outer
	var inner = function() {

		// контекст inner
		var myvar = x;

		return myvar;
	}

	var x = 10;

	return inner;
}

var inner1 = outer(1);

alert( inner1() ); // 10 !
```

Переменная outer является функцией, которая при вызове динамически создает и возвращает новую функцию inner.
Контексты выполнения функций вложены друг в друга: [[inner]] -> [[outer]] -> [[global]].
При выполнении var myvar = x; JavaScript-интерпретатор ищет переменную x последовательно в этих контекстах. В нашем случае он находит ее в [[outer]]. Там где нашел, ту и присвоил.

Если к динамически созданной функции inner привязывается какой-то внешний объект, то она уже не исчезает просто так. В примере функция inner возвращается и продолжает существовать уже после того, как outer закончит свою работу. Для того, чтобы ее запуск был успешным, JavaScript полностью сохраняет всю цепочку контекстов [[inner]]->[[outer]]->[[global]] , вместе со всеми переменными.
При вызове inner1() JavaScript будет искать переменную x, найдет ее в контексте [[outer]], она равна 10 (т.к outer уже закончила выполнение) и вернет 10.

Если хочется, чтобы переменная x была сохранена такая, какой была на момент создания функции inner, то есть как минимум два пути. Первый – создать объект и запомнить переменную в его свойстве. Второй – создать промежуточный контекст между [[outer]] и [[inner]], в котором у x будет нужное значение:

```
// глобальный контекст
var outer = function(x) {

	// контекст outer
	var inner = function(x) {
		// создаем функцию и тут же запускаем ее,
		// так что внутри значение x - текущее!

		return function() {

			// контекст inner -> промежуточная функция->outer
			var myvar = x;
```

```
                return myvar;
            }
        }(x);

        var x = 10;

        return inner;
}
```

При запуске inner переменная x будет искаться сначала в [[inner]], а потом в контексте промежуточной анонимной функции, где оно как раз то, что нужно.

Понимание замыканий играет важную роль при написании кода и управления памятью в сложных JavaScript-приложениях.

7.3 Фреймворк dojo

➔ http://dojotoolkit.org

7.3.1 Пакеты

Подключение пакетов происходит в compile-time и runtime-time директивами

```
dojo.require("some.package.class");
dojo.provide("some.package.class");
```

Компиляция производится собственной системы сборки с учетом зависимостей на уровне пакетов. Файл получается больше, чем сборка на уровне функций, но зависимость по пакетам существенно более удобна при разработке крупных проектов.

7.3.2 События

7.3.2.1 dojo.event.connect

Позволяет вешать обработчики как события DOM-узлов (onclick, onmouseover…), так и на вызовы функций.

- ➔ Поддерживает несколько обработчиков
- ➔ Работает кросс-браузерно
- ➔ Автоматическая защита от утечек памяти

Вызов функции doFoo вешается на событие fooNode.onclick

```
var fooNode = document.getElementById("foo"); // DOM node
dojo.event.connect(fooNode, "onclick", doFoo);
```

Функция handler.quux будет вызвана сразу после bar.baz

```
var bar = { baz: function() { alert("baz!");  } };
var handler = { quux: function() {  alert("quux!"); } };
```

```
dojo.event.connect(bar, “baz”, handler, “quux”);
// теперь после bar.baz() будет вызываться handler.quux()
```

7.3.2.2 dojo.event.topic

dojo.event.connect замечательно работает, но ее соединения неявные. Часто гораздо удобнее и правильнее организовывать события в схему publish-subscribe.

```
// подписать оба объекта на событие “/foo”
dojo.event.topic.subscribe(“/foo”, bar, “baz”);
dojo.event.topic.subscribe(“/foo”, xyzzy, “quux”);

// вызовет оба метода с аргументами arg1, arg2
dojo.event.topic.publish(“/foo”, “arg1”, “arg2”);
```

7.3.3 Аспектно-ориентированное программирование (АОП)[3]

7.3.3.1 Краткое введение

Обычное процедурно- или объектно-ориентированное программирование разделяет “*зоны ответственности*” на отдельные сущности. Процедуры, пакеты, классы помогают отделить их друг от друга и инкапсулировать. Однако, далеко не любая функциональность поддается легкой декомпозиции.

Классический пример - **логирование**, так как производится через вставку вызовов во многие части системы. Так что логирование *пересекает* многие методы и процедуры.

Аспектно-ориентированное программирование, как правило, старается инкапсулировать такие пересекающиеся зоны через введение *аспектов(aspect)*. Аспект может менять поведение базового кода через *вплетение(weaving)* в *точки сочленения(join points, pointcuts)*: вызовы методов а, в некоторых реализациях, и в операции над свойствами объектов.

Метод, который производит логирование, выносится отдельно и, при помощи соответствующих методов, привязывается в нужным методам (*вплетается)*.

Например, вплетение *перед вызовом(before)* вызовет логирование перед каждым вызовом метода, с передачей аргументов. При этом сам метод не только не производит вызова, но даже не знает о нем.

Также возможно вплетение *после вызова(after)* и *вместо вызова(around)*. В последнем случае вплетение приведет к вызову нового *вплетенного* метода вместо исходного. При этом вплетенному методу передаются аргументы для вызова исходного, если это вдруг понадобится.

Таким образом, можно устроить логирование всех вызовов методов БЕЗ какой-либо модификации вызывающего кода.

---

[3] К сожалению, устоявшегося перевода некоторых терминов не существует, поэтому используемая терминология будет сопровождаться названиями на английском языке

Еще один распространенный пример применения АОП – транзакции. Закончить, начать или продолжить текущую транзакцию решает не сам метод (который не знает о ней), а вплетенный код.

Таким образом,

1. улучшается разделение кода
2. можно модифицировать поведение методов изменения соответствующих библиотек

Более подробно об АОП можно почитать, например, на

- ➔ http://en.wikipedia.org/wiki/Aspect-oriented_programming
- ➔ http://www.javable.com/columns/aop/workshop/02/
  http://www2.parc.com/csl/projects/aop/
- ➔ http://www.google.ru/search?q=aspect-oriented+programming

7.3.3.2 Реализация в dojo

Что же касается нашей главной темы – dojo, то в ней АОП реализуется через динамический перехват методов. Т.к методы являются лишь свойствами объектов (включая глобальный объект window), то метод можно переименовать на новый, динамически созданный, а старый вызвать в любом удобном месте.
Происходит это, приблизительно, так:

```
var global = this;

function naiveConnect(funcName1, funcName2){
    global["__prefix"+funcName1] = funcName1;
    global[funcName1] = function(){
        global["__prefix"+funcName1].apply(global, arguments);
        global[funcName2].apply(global, arguments);
    }
}
```

Вплетение осуществляет метод dojo.event.connect. Например, вплетение вместо:

```
function foo(arg1, arg2){
    // ...базовый код
}

function aroundFoo(invocation){
    // код для вплетения вместо, invocation - массив аргументов для вызываемой функции
    if(invocation.args.length < 2){
        // можно поменять аргументы
        invocation.args.push("default for arg2");
    }

    // вызвать исходную функцию
    var result = invocation.proceed();
    // а тут – и поменять результат, если будет необходимо
    return result;
}

// произвести вплетение
dojo.event.connect("around", "foo", "aroundFoo");
```

i. Коммуникация с сервером: AJAX и т.п.

Пример: получить текст

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.txt",
    load: function(type, data, evt){ /* обработать данные */ },
    mimetype: "text/plain"
});
```

Пример: отправка формы

```
dojo.io.bind({
    url: your_url,
    formNode: dojo.byId("your_form"),
    method: "POST",
    load: function(type, value, evt) { dojo.byId("your_div").innerHTML = value; },
    error: function(type, error) { alert("Error: " + type + "n" + error); },
});
```

Пример: добавить AJAX в вызов функции clicked текущего объекта.

```
this.clicked = function() {
    dojo.io.bind({
        url: this.actionURL,
        method: 'get',
        content: {
            id: this.rowId,
            field: this.dbField,
            value: this.checked
        },
        // вызвать функцию в контексте текущего объекта
        load: dojo.lang.hitch(this, this.clickSuccess)
    });
}
```

1. Back/Forward

Добавляются параметры backButton/forwardButton – функции для вызова при нажатии соответствующих кнопок.

```
var sampleFormNode = document.getElementById("formToSubmit");

dojo.io.bind({
    url: "http://foo.bar.com/service.php",

    formNode: sampleFormNode, // отослать форму на адрес

    mimetype: "text/json", // полученные данные автоматом eval'ятся

    load: function(type, evaldObj){
        // спрятать форму при успешной загрузке
        sampleFormNode.style.display = "none";
    },


    backButton: function(){
        // ...когда посетитель жмет back – показать форму
        sampleFormNode.style.display = "";
```

```
    },
    forwardButton: function(){
        // если посетитель нажмет forward до следующего запроса
        sampleFormNode.style.display = "none"; // не отсылать форму, а просто спрятать
    }
});
```

2. Закладки

Добавляется параметр changeURL, который копируется в хэш-часть URL.

```
dojo.io.bind({
    url: "http://foo.bar.com/sampleData.txt",
    load: function(type, data, evt){ /* обработать данные */ },
    changeURL: "foo,bar,baz",
    mimetype: "text/javascript"
});
```

```
URL до функции: http://foo.bar.com/howdy.php
URL после функции: http://foo.bar.com/howdy.php#foo,bar,baz
```

К сожалению, в настоящий момент полной синхронизации смены URL с получением информации нет.

3. Компоненты интерфейса (виджеты, Widgets)

API для виджетов, шаблонная система,
Готовые компоненты.
Пример:

Описание класса SlideShow.js

```
// указать пакет для системы зависимостей
dojo.provide("dojo.widget.SlideShow");

// загрузить зависимости
dojo.require("dojo.widget.*");

// определить класс
dojo.widget.SlideShow = function(){
    // наследование
    dojo.widget.HtmlWidget.call(this);

    this.templatePath = dojo.uri.dojoUri("src/widget/templates/HtmlSlideShow.html");
    this.templateCSSPath = dojo.uri.dojoUri("src/widget/templates/HtmlSlideShow.css");

    this.widgetType = "SlideShow";

    // добавить свойства
    this.imgUrls = [];
    this.delay = 5000;
    this.imgHeight = this.imgWidth = 400;
    this.img1 = null;
    this.img2 = null;
}
// закончить наследование
```

```
dojo.inherits(dojo.widget.SlideShow, dojo.widget.HtmlWidget);
// добавить тег для вставки в страницу
dojo.widget.tags.addParseTreeHandler("dojo:slideshow");
```

Шаблон HtmlSlideShow.html

```
<div style="position: relative; width: 800px; height: 600px;" dojoAttachPoint="imagesContainer">
   <img dojoAttachPoint="img1" border="0" class="slideShowImg" style="z-index: 1;" />
   <img dojoAttachPoint="img2" border="0" class="slideShowImg" style="z-index: 0;" />
</div>
```

Свойства imagesContainer, img1, img2 будут автоматически прочитаны парсером в соответствующие свойства объекта.

Код страницы SlideShow.html

```
…
<img dojoType="SlideShow"
   imgUrls="images/1.jpg;images/2.jpg;images/3.jpg;images/4.jpg"
   transitionInterval="700"
   imgWidth="800"
   imgHeight="600"
   delay="7000"
   src="images/1.jpg" />
…
```

Свойства из тега <img> будут также автоматически прочитаны парсером, которые покажет на месте него заполненный и проинициализированный обработчиками шаблон.

Ссылки:

- http://dojotoolkit.org/docs/fast_widget_authoring.html

2. Коллекция проблем и решений. “А что у ней внутре?”

a. Кнопки back-forward, закладки

Одна из проблем при работе с AJAX – обработка back-forward кнопок, которые должны возвращать приложение на предыдущий-следующий момент работы, но этого не происходит.
Справедливости ради, заметим, что и в классических веб-приложениях эти кнопки не всегда работают так, как нужно, особенно это касается IE.
Но в AJAX эти кнопки обычно работают вообще не так как нужно, back производит выход с адреса приложения, вместо изменения состояния на предыдущее, и т.п.

Причина проста – почти все транспорты никак не влияют на history. IFrame может на нее повлиять, но при обычном сабмите внутрь ифрейма такое влияние может только запутать или даже повредить работе в приложении. При описании IFrame-транспорта было рассмотрено, как избавиться от этого – достаточно

1. создавать ифрейм динамически
2. загружать новый URL через iframe.contentWindow.document.location.replace (или аналоги для Mozilla/Safari)

При этом также пропадает “кликающий звук”, и IFrame-транспорт становится приблизительно аналогичным другим транспортам (кроме индикатора прогресса, которого нет для XMLHttpRequest).

Как сделать, чтобы back-forward работали надлежащим образом? Внешний интерфейс простой – снабжать функцию AJAX-запроса необязательным параметром – функцией, которая будет выполняться при нажатии кнопки back.
Эта функция может восстанавливать предыдущее состояние приложения, отменять результаты работы и т.п.

Чтобы такая функция работала, нужно переходы back-forward отлавливать и обрабатывать. Для этого в документ на стадии загрузки добавляется специальный IFrame.

```
// Именно так, до загрузки(onload) документа
document.write("
        <iframe name='history' id='history'  src=”ihistory.html' + "'></iframe>"
);
```

Браузеры, как правило, поддерживают два внутренних списка history. Один из них – для посещенных страниц внешнего уровня, и он доступен в выпадающем списке у кнопки back (или вроде этого). А другой – внутренний, который включает в себя навигацию по ифреймам, и именно он используется для переходов back-forward.

ihistory.html – пустой документ, который принимает ряд параметров в URL (например, domain для установки document.domain: ihistory.html?domain=example.com).
Мы вольны менять его location как угодно через frames[history].location = src, а такая смена локации приведет к

1. попаданию нового элемента в history
2. вызову нашей функции изнутри iframe’а, когда тот загрузится.

Каждый раз, когда мы хотим создать новый элемент в history –

1. Добавляем нужный вызов(объект со спец. функцией) в стек вызовов historyStack

2. Ставим флажок moveForward (просто JS-переменная)

3. Вызываем смену location с новым URL’ом, в который входит хэш (т.е #что-то) со специальной меткой.
   Эта метка либо передается вместе с элементом (**становятся возможны закладки** “add to bookmark”), либо генерируется случайным образом (например, текущее время в миллисекундах).

4. При загрузке ифрейма если флажок установлен – снимаем его и ничего не делаем больше.

Каждый раз, когда пользователь нажимает back/forward – ифрейм ihistory переходит на соответствующий URL, вызывает функцию загрузки (т.к нет флажка moveForward), которая выполняет действия, исходя из содержимого стека.
При вызове добавления нового элемента в хистори после нажатия back несколько раз браузер сам удалит часть стека, которая “впереди” текущего момента, нам придется это сделать самостоятельно.

Такая обработка никак не влияет на обычные back-forward. Просто добавляются специальные history-метки, обработка которых, за счет загрузки ihistory, осуществляется нашими средствами.

Данный способ работает в IE/Mozilla/Opera 9. Есть проблемы с forward в Safari, не работает с Opera 8.

Вообще, хэш-часть URL может быть изменена без перезагрузки страницы, так что закладки можно организовывать и без управления back-forward, если оно вдруг не нужно.

b. Диагностика и борьба с утечками памяти

Утечка памяти(memory leaks) происходит, если объект создается, но не удаляется должным образом, так что память, которая была под него выделена, остается занятой.

Симптомы проблемы: при работе в браузере количество потребляемой памяти все время растет, а само приложение работает все медленнее и медленнее, по мере того как браузер съедает ресурсы.

Утечка бывает

- одностраничной – память освобождается при перезагрузке/уходе со страницы
- многостраничной - только закрытие браузера и зависимых окон очищает память.

Утечек памяти в браузерах много, и их причины – самые разные. Более подробно – см. официальные багрепорты.
Одинаковые причины, в зависимости от браузеров, могут давать разные типы утечек или не давать утечки вовсе.

Здесь будет рассмотрен основной паттерн утечек, которые создают наибольшие проблемы при разработке приложений.

А именно - утечки при круговых ссылках между разными типами объектов: DOM(COM), JavaScript, ActiveX/XMLHttpRequest. Т.к эти типы объектов имеют разные сборщики мусора, то в ряде случаев круговые ссылки не разрушаются, и все связанные ими объекты остаются в памяти.
“Чемпион” по видам и масштабу утечек – безусловно, IE, хотя и другие браузеры тоже им подвержены.

Явный пример круговой ссылки

```
<html>
<head>
	<script type="text/javascript">
		var object;

		function SetupLeak() {
			// Ссылка JS -> DOM
			object=document.getElementById("LeakedDiv");

			// Ссылка DOM -> JS
			document.getElementById("LeakedDiv").property = object;
		}
	</script>
</head>
```

```
<body onload="SetupLeak()">
        <div id="LeakedDiv"></div>
</body>
</html>
```

При перезагрузке такой страницы память под объекты DOM/JS не освобождается.
В таком простейшем случае утечка почти не заметна, но когда используются замыкания, то функция сохраняет с собой всю цепочку контекстов выполнения, так что масштаб проблем увеличивается на много порядков (см о замыканиях в соответствующем разделе).

i. Утечка с замыканием

```
<html>
<head>
<script type="text/javascript">
        function makeLeak() {
                var div = document.getElementById('myDiv');
                div.onclick = function() {
                        // что-нибудь сделать, не важно что
                };

                var someVar = new Array(1000).join(new Array(2000).join("XXXXX"));
        }
</script>
</head>
<body onload="makeLeak()">
<div id="myDiv"></div>
</body>
</html>
```

Обработчик события onclick сохраняет всю цепочку контекстов, вместе с переменной someVar

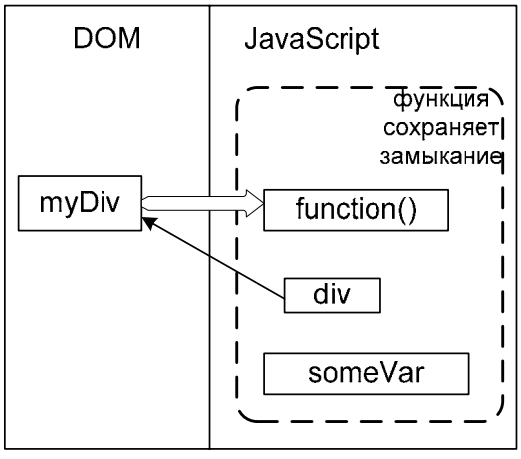


Рисунок 1 Общая картина утечки

c. Автоматическое уничтожение круговых ссылок

Техника – как избежать утечек при событиях и других связях DOM <-> JS.

1. Функция добавления события/ссылки регистрирует свойство в специальном “реестре” – особом массиве-свойстве дом-узла: node.__clobberAttrs__.push(“onload”) и добавляет сам узел в общий массив clobberNodes.
2. На window.unload навешивается функция, которая все узлы из clobberNodes очищает от атрибутов, ссылающихся на JS

d. JScript delete (IE)

При удалении свойства объекта в IE оно помечается специальным флагом и остается жить в памяти, пока объект существует. Это происходит из-за внутренних особенностей реализации, см. например http://blogs.msdn.com/ericlippert/archive/2004/10/07/239289.aspx

```
// объект, с которым будем работать
var obj = {};

// функция для вставки-удаления свойства
function addAndRemove() {
  var key;
  for (var i = 0; i < 10000; i++) {
    // произвольный уникальный ключ
    key = Math.random();

    // добавить свойство
    obj[key] = key; // значение не важно

    // удалить свойство
    delete obj[key];

    // на самом деле значение осталось в памяти
    // и будет там висеть, пока сборщик мусора не удалит obj !
  }
}

// для теста
window.setInterval("addAndRemove()", 50);
```

Проблему можно решить периодическим копированием свойств объекта во временный объект tmp, а затем obj = tmp. Так что весь объект будет собран и уничтожен сборщиком мусора, вместе со свойствами.

e. JavaScript / ActiveX

Простейший пример утечки:

```
function getUrl(url) {
        var xmlhttp = getXmlHttp();  // получить XMLHttpRequest-объект для запроса
        xmlhttp.open("GET", url+'?r='+Math.random(););
        xmlhttp.onreadystatechange = function() {
                if (xmlhttp.readyState == 4) { /* вызвать коллбэк  */ }
        }
        xmlhttp.send(null);
}
```

К XMLHttpRequest цепляется JS-обработчик измененного состояния, который должен иметь доступ к самому запросу для проверки состояния/обработки результата… Или

ссылаться на функцию, которая имеет к нему доступ – без разницы, образуется циклическая JavaScript <-> ActiveX ссылка, так что весь запрос (с результатом) и обработчик (с замыканием) останутся в памяти до закрытия браузера.
Уничтожить такую ссылку ни через delete ни через установку в null не удается.

Безопасный вариант работы – отслеживать xmlhttp.readyState регулярными проверками по setTimeout, так что выполняется принцип: ActiveX не ссылается ни на один JS объект.

i. Инструменты диагностики многостраничных утечек

Для диагностики утечек в IE – drip
http://www.outofhanwell.com/ieleak/
Разъяснения основных паттернов ликов в IE разработчиками Microsoft:
http://msdn.microsoft.com/library/en-us/ietechcol/dnwebgen/ie_leak_patterns.asp
http://blogs.msdn.com/ericlippert/archive/2003/09/17/53028.aspx

f. Одностраничные утечки

Сложные AJAX-приложения, как правило, долго работают в едином пространстве объектов. Если при этом создается много новых объектов, могут возникнуть проблемы. Один момент был описан в пункте про delete. Другой – копирование JavaScript кода.

Рассмотрим, например, следующее описание объекта

```
function someClass() {
	this.method1 = function() { /* какой-то код */ }
	this.method2 = function() { /* какой-то код */ }
	…
}

var a = new someClass();
```

При создании объекта через такой конструктор, будет создаваться новая функция, код которой копируется в новый объект.
Таким образом, если описание класса достаточно большое, то каждый новый объект будет отъедать уже существенное количество памяти. Создание 1000 объектов тестового класса размером 30k на IE отъедает ~1M памяти. Само по себе может быть и не страшно, но, умножаясь на другие утечки, может создать реальные проблемы.

не обладают такой проблемой

```
// явное присваивание прототипа
someClass.prototype.method1 = function() { .. /* код */ }

// копирование в прототип пачки методов через специальную функцию
myLib.extend(someClass,
	{ method1: function() {…} … }
);
```

3. Инструментарий

a. Редакторы

- JSEclipse
- Komodo
- jEdit
- Vi(m)
- …

b. Отладчики

- Venkman (Mozilla plugin)
- Miscrosoft Script Debugger (IE, включить в свойствах браузера JS-отладку)

c. Помощники

Многие инструменты пересекаются по возможностям, но ни одна из них не перекрывает другой полностью.

- JavaScript console (Opera, Safari, Mozilla)
  - классический инструмент для просмотра сообщений об ошибках JS/CSS

- FireBug (Mozilla plugin)
  - просмотр сообщений об ошибках
  - просмотр основных заголовов HTTP-запросов, тела XMLHttpRequest
  - выбор и полная информация о любом элементе DOM

- DOM inspector (Mozilla, при Custom установке Firefox поставить галочку) / IEDocMon (IE plugin)
  - Подробная информация о DOM документа, свойствах элементов и внешних загруженных объектах, удобный поиск объектов
  - Аналогичное расширение плюс набор своих фишек - Internet Explorer Developer Toolbar (IE plugin)

- Web Developer (Mozilla plugin)
  - полоска с кучей удобных инструментов

- LiveHttpHeaders (Mozilla plugin) / ieHttpHeaders (IE plugin)
  - показывает полные заголовки HTTP-запросов

- Show selection source (Mozilla, контекстное меню) / Fullsource (IE plugin)
  - дает возможность смотреть результирующий HTML, который показывает браузер в настоящий момент с учетом сработавших скриптов

- Drip (IE)
  - диагностика и локализация утечек памяти

- Apache2 + bw_mod + mod_proxy

- Ограничение скорости соединения к сайту, для чего ставится медленный локальный прокси. Полезно для проработки/тестирования деталей UI, относящихся к сетевым задержкам и ошибкам.
  Пример конфигурации :

```
<VirtualHost 127.0.0.1:80>
    ServerName быстрый.сайт.ru      # в hosts прописать
    BandWidthModule On
    BandWidth all 10240     # ограничение скорости

    ProxyPass / http:///IP.БЫСТРОГО.САЙТА/
    ProxyPassReverse / http:///IP.БЫСТРОГО.САЙТА/
</VirtualHost>
```

- VirtualPC/VMWare/Wine
  - отладка в браузерах, ориентированных на работу в другой ОС.

d. Сжатие JavaScript

---

i. Gzip в жизни браузеров

Современные вебсерверы позволяют применять GZIP-сжатие для HTML, равно как и JS/CSS файлов.

Однако, на текущий момент в браузерах (в основном, в IE) много ошибок, касающихся правильной декомпрессии.
Например, страница передается в плагины в неразжатом виде, теряется часть страницы, декомпрессия происходит не тогда, когда надо(читай, не происходит) и т.п…

Поэтому, если Вы хотите, чтобы сайты показывались у посетителей максимально стабильно, лучше декомпрессии избежать вообще.

Бывает, что страницы содержат много HTML-кода: плохая верстка, стили не вынесены в CSS, или (что бывает куда реже) просто много информации.
В таких случаях gzip может быть оправдан (больше хитов ценой стабильности) для HTML.
С другой стороны, JS/CSS гораздо менее динамичен, кэшируется в браузере, декомпрессия глючит сильнее, чем HTML, лучше его вообще не GZIP’ать.

---

ii. Сжатие без архивации

Способов сжать яваскрипт достаточно много

1. Удалить комментарии
2. Убрать лишние пробелы
3. Заменить имена переменных на более короткие

Пункты 1 и 2, в общем, очевидны. Внешнее API сжатых скриптов должно оставаться тем же, поэтому пункт 3 реализовать достаточно сложно.

Наилучший, известный автору, вариант – патч к Rhino от Alex Russel. Сам Rhino представляет собой реализацию JavaScript на Java, включая парсер.
Патченный Rhino можно использовать для безопасной замены всех локальных переменных короткими именами.
При этом имена функций, константы и стандартные свойства сохраняются, так что внешнее API остается неизменным.

Также есть несколько дополнительных полезных опций и, разумеется, можно добавлять новые – проект Open Source.

Компрессор можно скачать здесь
http://dojotoolkit.org/svn/dojo/trunk/buildscripts/lib/custom_rhino.jar
Соответствующий патч для обычного Rhino
http://dojotoolkit.org/svn/dojo/trunk/buildscripts/lib/custom_rhino.diff
Использовать так:

```
java -jar custom_rhino.jar -c infile.js > outfile.js 2>&1
```

4. Классические ошибки в разработке

Что-то из этого списка учесть легко, что-то – сложно, что-то – невыгодно при разработке простых приложений… Но иметь в виду стоит, в любом случае.

a. Веб-интерфейс

- Отсутствие видимой индикации инициированных действий.

В GMail, например, когда человек нажимает ссылку – справа появляется надпись “Loading...”, которая показывает, что инициировано какое-то действие и оно ещё не закончилось.

- Изобретение собственных странных интерфейсов. Типично для AJAX-приложения – “Кликните на эту неочевидную финтифлюшку, чтобы добиться ещё более неочевидного результата”.

- Неочевидная разбивка операций. Во время привычной работы пользователя со страницей происходят странные действия, к которым он не привык. Например, автопроверка формы.

- Прокрутка страницы и сбивание фокуса пользователя. Если приложение пытается само видоизменять страницу, перестраивая или прокручивая её, то читать такую страницу практически невозможно.

- Нераспространение локальных изменений на остальную часть страницы.
  При “прочитывании” сообщения должны изменяться все существующие на странице счётчики “прочитанного”, которых это касается.
  Другими словами, нужно соблюдать целостность интерфейса.

- Неожиданное мигание и видоизменение страниц. Чрезмерное увлечение “асинхронностью” AJAX приводит к тому, что страница начинает жить собственной жизнью.

b. Реализация

- Неправильно работающие кнопки Back/Forward.

- Невозможность передать ссылку друзьям или в закладки. AJAX-приложения, всегда имеющие один и тот же URL, и при этом не настолько личные, как почта.

- Замедление браузера из-за объёмов кода. Больше кода — медленнее работает. JavaScript — не самый быстрый язык.

- При асинхронных запросах НЕ гарантирован порядок выполнения. В результате многочисленных запросов к серверу может получиться, что пользователь в итоге не сможет сказать, чем всё закончилось и в каком состоянии находится система.

- Согласно спецификации HTTP, предназначение GET - извлечения информации, этот метод должен быть *безопасным* и *идемпотентным*, в то время как для модификации используется POST.

  В данном контексте *безопасный* означает, что эта операция предназначены для извлечения, а не модификации информации. Другими словами, у запроса GET обычно не должно быть побочного действия. *Идемпотентный* означает, что несколько запросов к одному и тому же унифицированному локатору (URL) ресурса должны выдавать одинаковый результат. Приведенные определения менее строгие, чем они могут показаться. По существу, их смысл состоит в том, что если пользователь обращается по ссылке, он должен быть уверен, что с его точки зрения он не изменяет ресурс.

  Обеспечение оптимальной производительности браузеров и промежуточного программного обеспечения (прокси, брандмауэров и решений по доставке контента а-ля Akamai) достигается зависит от возможности различать запросы.
  В частности, некоторые корпоративные прокси кэшируют(на небольшое время) любой GET-запрос, вне зависимости от заголовков.

  Смотрите сами, насколько такое поведение безопасно для вашей задачи.

5. Когда использовать AJAX ?

a. Плюсы

- Быстрота ответа приложения
- Новые возможности проектирования

b. Минусы

- Обязателен JavaScript на клиенте
- Легко ошибиться в проектировании, оценках разработки

c. Сложности

- Усложненное (автоматизированное) тестирование и отладка
- Кросс-браузерные несовместимости (для Rich Client), модификации браузеров

d. Так когда же?

- Приложение разрабатывается для основных браузеров

- Нужно реализовать вещи, которые недоступны в классической архитектуре запрос-обновление страницы

- **Просто когда это улучшит жизнь пользователям!**