AJAX/COMET

Лекция №5.1 (веб версия: http://goo.gl/b6KVNO)

- Введение
  - AJAX
  - Где можно использовать AJAX?
  - COMET
- Транспорты AJAX
  - IMG
  - IFRAME
  - SCRIPT
  - XHR
- XMLHttpRequest
  - Отправка запроса
  - Обработка ответов
  - Синхронные/асинхронные зарпросы
  - Подготовка/обработка данных
    - Определение формата полученных данных
    - Подготовка данных к отправке
      - URL-encoded данные
      - JSON данные
      - XML данные
- Кросс-доменные запросы
  - XMLHttpRequest
  - JSONP

Введение

AJAX

Это подход к созданию веб-приложений, подразумевающий их функционирование без перезагрузки страницы.

Впервые был использован человеком по имени Джесси Джеймс Гаррет в 2005м году. А возможность использовать данных подход появилась впервые в браузере Microsoft IE5. Распространение же подход получил после того, как свет увидел сервис Google Mail.

Расшифровывается аббревиатура, как Asynchronous JavaScript and XML. Несмотря на то, что в названии используется XML, это нас ни к чему не обязывает, т.е. с помощью AJAX мы можем передовать информацию в любом текстовом формате будь то обычный

текст, JSON, XML и даже бинарные данные, но эта возможность предусмотрена лишь в спецификации XMLHttpRequest Level 2 и на данный момент поддерживается не всеми браузерами.

Где можно использовать AJAX?

Очевидно, что данная технология идеально подходит для элементарных действий, таких как комментирование, голосования, выставление рейтингов. Все это отправка минимальных объемов данных на сервер. В данном случае загружать весть документ целиком заново будет настоящим расточительством, когда нужно отправить всего лишь 1 запрос и после ответа обновить небольшую часть страницы.

Кроме того, можно использовать AJAX для динамической загрузки содержимого. Для ускорения загрузки страницы, или когда изначально просто неизвестно что именно показывать пользователю. Самым ярким примером тут может стать «живой поиск», автодополнения, постепенная загрузка изображений товаров в интернет магазинах.

Самое важное, что AJAX открыл возможности для построения в вебе полноценных приложений (Rich Internet Applications, Single Page Web Applications), которые своим поведением походят на нативные, но при этом доступны везде, где есть интернет, кросплатформенны, нет пролем с обновлением и прочие плюшки..

COMET

Комет — способ асинхронного взаимодействия, когда не клиент спрашивает информацию, а сервер пушит ее клиенту.

Самый понятный и простой пример COMET-приложения — чат. N клиентов отправляют сообщения одному серверу, он — пушит эти сообщения всем остальным клиентам.

Подробно рассматриваться COMET в текущей лекции не будет.

Транспорты AJAX

Осуществить асинхронное взаимодействие между клиентом и сервером без перезагрузки страницы можно целым рядом способов, в каждом из которых для передачи информации используется различный «транспорт».

IMG

Самый простой и ограниченный вид AJAX транспорта. Суть реализации взаимодействия заключается в том, чтобы динамически создать тег <img>. В тот момент, когда скрипт задаст значение src браузер инициирует GET запрос к указанному URL. Данные для передачи нужно «зашить» прямо в URL.

Минусов у данного транспорта 2:

- сервер в любом случае должен вернуть изображение в ответ, это может быть, например, однопиксельная прозрачная картинка
- связь получается *только в одну сторону*

IFRAME

Сам подход практически идентичен предыдущему, создаем тег <iframe>, устанавливаем ему src в нужное значение. Сервер отдает ответ в виде содержимого документа, загружаемого в iframe. Скрипт с родительской страницы, парсит загруженный в iframe документ и забирает данные, отданные сервером.

На заметку:
Вы не сможете получить доступ к содержимому iframe, если он был загружен с домена, отличного от того, где находится родительская страница.

SCRIPT

Главным отличием этого вида транспорта от предыдущих является то, что он не попадает под same-origin политику безопасности. Именно на нем базируется подход JSONP, который мы рассмотрим чуть позже.

Доступ к возвращенному контенту, родительская страница получает автоматически, т.к. полученный код будет выполнен браузером сразу после загрузки.

XHR

XMLHttpRequest — самый удобный способ взаимодействия с сервером из браузера, никаких дополнительных DOM узлов и прочего непотребства. Данный объект предоставляет API, способный возвращать ответы в виде обычного текса или Document object.

Как и в ситуации с AJAX, название XMLHttpRequest не очень удачное. Т.к. формат не обязательно должен быть XML, а кроме http, XMLHttpRequest замечательно умеет слать запросы и по https протоколу.

На заметку:
Нет возможности использовать этот вид транспорта при работе в локальной файловой системе, т.е. страница открытая из file://... не сможет отравить AJAX запрос с использованием XMLHttpRequest.

XMLHttpRequest

На текущий момент API, предоставляемое объектом XMLHttpRequest поддерживается всеми, даже не очень современными браузерами. А в некоторых из них даже реализован стандарт XMLHttpRequest Level 2, который расширяет возможности представленные в первой версии.

Отправка запроса

Первым делом нам необходимо создать объект XMLHttpRequest, т.к. IE 6-7 больше не существуют, то сделать это удивительно просто

```
var xhr = new XMLHttpRequest();
```

После того, как объект создан необходимо вызвать его метод .open() для того, чтобы задать 2 обязательных параметра запроса

- метод запроса (GET, POST, DELETE, HEAD, OPTIONS, PUT)
- URL к которому мы будем обращаться

Задаются эти два параметра первым и вторым аргументом соответственно, в качестве 3-его аргумента метод .open() может принимать булевое значение, которое задаст синхронно или асинхронно будет отправлен запрос и логин/пароль для HTTP авторизации, если они необходимы.

При указании целевого URL нужно помнить о том, чтобы он соответствовал Cross-Origin политике безопасности браузера, т.е. о том, отправить запрос с помощью XMLHttpRequest можно только на тот же самый домен, на котором располагается документ который его инициирует.

XMLHttpRequest Level 2 позволяет нам отправлять даже кросс-доменные запросы, но в таком случае отвечающая сторона должна принять кое-какие меры, рассмотрим это чуть позже.

```
function sendRequest(msg){

	var xhr = new XMLHttpRequest();

	xhr.open("POST", "someData.json");
}
```

После того, как мы зададим метод запроса и целевой URL нам может понадобиться задать так же и какие-то HTTP-заголовки для этого XMLHttpRequest предоставляет метод .setRequestHeaser(), который первым аргументом принимает название заголовка, который нужно установить, а вторым — его значение.

Нужно иметь в виду, что не для всех заголовков можно задать значения, ниже приведен список из тех, для которых этого не удастся сделать с помощью .setRequestHeader()

Accept-Charset	**Content-Transfer-Encoding**	**TE**
Accept-Encoding	**Date**	**Trailer**
Connection	**Expect**	**Transfer-Encoding**
Content-Length	**Host**	**Upgrade**
Cookie	**Keep-Alive**	**User-Agent**
Cookie2	**Referer**	**Via**

После того, как установлены заголовки самое время отправить запрос, для этого у нас есть метод .send(). Который в качестве единственного аргумента принимает тело отплравляемого сообщения, для запросов которые не могду иметь тела, например GET или HEAD в качестве аргумента в .send() нужно передать null или пустую строку.

Таким образом функция, которая отправит некоторое сообщение на сервер будет выглядеть следующим образом:

```
function sendRequest(msg){

	var xhr = new XMLHttpRequest();

	xhr.open("POST", "someData.json");
	xhr.setRequestHeader("Content-Type", "text/plain;charset=UTF-8");

	xhr.send(msg);
}
```

Обработка ответов

Каждый HTTP ответ состоит из 3 частей:

- Статусная строка
  - код ответа
  - статусное сообщение
- Заголовки ответа
- Тело ответа

Для того, чтобы получить каждую из частей ответа в объекте XMLHttpRequest присутствует свойство или метод.

Для статуса это свойство status и statusText, для статусного сообщения. Заголовки ответа можно получить либо по одному с помощью .getResponseHeader(), либо все сразу с помощью .getAllResponseHeasers(). Тело ответа можно получить через совойства responseXML или responseText, в зависимости от формата содержимого ответа.

Описанные выше свойства и методы не будут неопределены/не вернут корректного значения до момента получения ответа.

O_O *как же узнать когда пришел ответ от сервера?!*

Каждый экземпляр объекта XMLHttpRequest имеет несколько этапо на своем жизненном пути, которые отражаются на значении такого его свойтва, как .reasyState. Принимать оно может следующие значения:

Константа	Значение	Текстовый эквивалент
UNSENT	**0**	.open() не был вызван
OPENED	**1**	.open() был вызван
HEADERS_RECEIVED	**2**	получены заголовки ответа
LOADING	**3**	идет процесс получения ответа
DONE	**4**	ответ полностью получен

В соответствии с таблицей значение .readyState можно сравнивать либо с целочисленным значением, либо с соответствующей ему константой, например XMLHttpRequest.DONE, но константы данные определены только в современных браузерах и IE > 8.

По стандарту каждый раз при смене значения .readyState должно генерироваться соответствующее событие readystatechange, но это происходит в каждом браузере по-разному. Например, довольно часто событие не генерируется при изменении значения с 0 на 1, а со значением 3 readystatechange может быть сгененрировано несколько раз, для того, чтобы можно было отследить и визуализировать процесс загрузки данных.

Задать обработчик для этого события можно как и спомощь привычных методов, attachEvent() или addEventListener(), так и через свойство .onreadystatechange, т.к. нам вполне достаточно одного обработчика на это событие, то не вижу никаких проблем не использовать .onreadystatechange наша функция для отправки запросов после усовершенстования будет выглядеть следующим образом

```
function sendRequest(msg){

    var xhr = new XMLHttpRequest();

    xhr.open("POST", "someData.json");
    xhr.setRequestHeader("Content-Type", "text/plain;charset=UTF-8");
```

```
    xhr.onreadystatechange = function(){
        if(xhr.readyState === 4 && xhr.status === 200){
            alert(xhr.responseText);
        }
    }

    xhr.send(msg);
}
```

Синхронные/асинхронные зарпросы

Выше в метод .send() мы передали только 2 аргумента, а еще выше узнали, что с помощью третьего аргумента можно задать синхронно или асинхронно будет отправлет запрос. По умолчанию запрос будет отправлен асинхронно.

Разница между этими 2мя вариантами в том, что при синхронной отправке ваш браузер «залипнет» на время ожидания ответа от сервера. И если это ожидание будет долгим, пользователю вряд ли понравится то, что интерфейс приложения вообще никак не откликается. Соответственно при асинхронной отправке этого эффекта наблюдаться не будет, и после отправки запроса браузер сможет без проблем обрабатывать реакции пользователя, до того момента, пока не придет ответ.

Отправка синхронного запроса будет выглядеть так

```
function sendRequest(msg){

    var xhr = new XMLHttpRequest();

    xhr.open("GET", "someData.json", false);
    //...
    xhr.send(null);
}
```

Подготовка/обработка данных

В зависимости от того, какой Content-Type мы укажем при обращении и каким методом отравим запрос, сервер будет ожидать от нас данные разного формата. Соответственно в зависимости от того что нам отправляет сервер мы вынуждены по-разному обратывать полученную информацию да и доступ к ней тоже нужно будет получать разными способами.

Определение формата полученных данных

Для того, чтобы определить, что нам прислала сервер необходимо прочитать заголовок ответа Content-Type.

```
function sendRequest(msg, callback){
```

```
    var xhr = new XMLHttpRequest();

    xhr.open("POST", "someData.json");
    xhr.setRequestHeader("Content-Type", "text/plain;charset=UTF-8");
    xhr.onreadystatechange = function(){
        if(xhr.readyState === 4 && xhr.status === 200){
            var type = xhr.getResponseHeader("Content-Type");

            if(type.indexOf("xml") !== -1){
                callback(xhr.responseXML);
            }
            else if(type === "application/json"){
                callback(JSON.parse(xhr.responseText))
            }
            else{
                callback(xhr.responseText);
            }
        }
    }

    xhr.send(msg);
}
```

В приведенном выше фрагменте кода, мы усовершенствовали нашу функцию для отправки запросов таким образом, что теперь она в состоянии корректно принять и подготовить различные типы данных, пришедшие от сервера и передать их для обработки в другую, полученную в качестве второго аргумента функцию.

В некоторых случаях сервер может отдавать заголовки, которые не соответствуют формату получаемых данных. Чтобы подстраховаться на этот случай XHR предоставляет нам метод .overrideMimeType(), если выпереданите туда какое-либо значение, то XHR проигнорирует заголовок отправленный сервером и будет ориентироваться только на то, значение, которое вы установили самостоятельно.

Подготовка данных к отправке

URL-encoded данные

При отправке данных методом GET они должны быть упакованы прямо в URL в следующем виде:

```
key1=val1&key2=val2&key3=val3&keyN=valN
```

Аналогичным же образом (пары ключ-значение связаны через =, сами пары между собой соединены через &, а все спец символы заменены их 16-ми кодами) данные пакуются, если вы отправляете форму, засабмитив ее.

Такому способу компановки данных соотвествует значение заголовка Content-Type “application/x-www-form-urlencoded”. Проблема наша в том, что после того, как мы скриптом соберем данные из формы, или получим их из какого-либо другого источника они скорее всего будут представлять из себя обыкновенный javascript объект.

```
{
    key1 : "val1",
    key2 : "val2",
    ...
    keyN : "valN"
}
```

Так что нам придется сделать функцию, которая эти данные приведет в пригодный для отправки вид. Выглядеть она будет примерно так.

```
function encodeData(data){

    var data  = data || {},
        pairs = [];

    for(var key in data){

        var value = data[name].toString();

        name  = encodeURIComponent(name).replace("%20", "+");
        value = encodeURIComponent(value).replace("%20", "+");

        pairs.push(name + "=" + value);
    }

    return pairs.join("&");
}
```

Соответственно функция для отправки запроса теперь может выглядеть так

```
function sendRequest(data, callback){

    var xhr = new XMLHttpRequest();

    xhr.open("POST", "someData.json");
    xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
    xhr.onreadystatechange = function(){
        // ...
    }
```

```
    xhr.send(encodeData(data));
}
```

JSON данные

Для того, чтобы отправить JSON данные, особых телодвижений нам не нужно, достаточно установить правильный заголовок запроса и упаковать данные с помощью метода JSON.stringfy()

```
function sendJSONRequest(data, callback){

    var xhr = new XMLHttpRequest();

    xhr.open("POST", "someData.json");
    xhr.setRequestHeader("Content-Type", "application/json");
    xhr.onreadystatechange = function(){
        // ...
    }

    xhr.send(JOSN.stringify(data));
}
```

XML данные

С XML все немного сложнее, т.к. придется создать XML документ, примерно такого вида:

```
<query>
    <find entity="students" grade="4" course="front-end">
</query>
```

Зато есть и хорошие новости: объект XHR самостоятельно выставит корректный заголовок, как только поймет, что в качестве аргумента .send() получил XML документ.

Кросс-доменные запросы

В целях безопасности нельзя организовать кросс-доменные запросы большинством из доступных AJAX транспортов.

XMLHttpRequest

С помощью XHR можно отравить cross-origin запрос, если со стороны сервера будут предприняты меры, состоящии в том, чтобы среди заголовков ответа находился “Access-Control-Allow-Origin”, со значением домена, с которого отправлен запрос.

Данная возможность поддерживается во всех современных браузерах и даже в IE 8,9 но там для отправки cross-origin запроса вместо XHR нужно использовать объект XDomainRequest.

JSONP

Это замечательный механизм, который позволяет отправить запрос на любой домен. Доступна эта возможность потому, что в качестве транспорта используется тэг <script>, на загружаемые с помощью которого ресурсы не распространяется cross-origin политика безопасности.

На заметку:
Из этого следует, что обращаться с помощью JSONP нужно *только к серверам, которым вы доверяете*.

Суть метода очень проста:

- на странице динамически создается тэг <script>
- в URL указанно в качестве значения атрибута src добавляется параметр callback, в который помещается имя функции, которую мы хотим вызвать на клиенте, после того, как получим ответ
- ответ сервер оформляет в виде вызова этой самой функции и передает нужные данные в нее, как аргументы
- как только содрежимое тэга <scirpt> загружено на страницу — оно сразу выполняется, следовательно вызывается наша функция

Пример функции, которая отправляет кросс-доменный запрос с помощью JSONP

```
function sendJSONPRequest(URL, callback){

    var cbNum  = "cb" + sendJSONPRequest.counter++,
        cbName = "jsonpCallback" + cbNum,
        script = document.createElement("script");

    if(URL.indexOf("?") === -1)
        URL += ("?callback=" + cbName);
    else
        URL += ("&callback=" + cbName);


    jsonpCallback[cbNum] = function(responseData){

        try{
            callback(responseData);
        }
        finally{
            delete jsonpCallback[cbNum];
            script.parentNode.removeChild(script);
        }
    }
```

```
    script.src = URL;
    document.body.appendChild(script);
}
```