Основы XMLHttpRequest

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

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

Использовать его очень просто.

Пример использования

Как правило, `XMLHttpRequest` используют для загрузки данных.

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

```
// 1. Создаём новый объект XMLHttpRequest
var xhr = new XMLHttpRequest();

// 2. Конфигурируем его: GET-запрос на URL 'phones.json'
xhr.open('GET', 'phones.json', false);

// 3. Отсылаем запрос
xhr.send();

// 4. Если код ответа сервера не 200, то это ошибка
if (xhr.status != 200) {
  // обработать ошибку
  alert( xhr.status + ': ' + xhr.statusText ); // пример вывода: 404: Not
Found
} else {
  // вывести результат
  alert( xhr.responseText ); // responseText -- текст ответа.
}
```

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

Настроить OPEN

Синтаксис:

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

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

Задаёт основные параметры запроса:

- `method` — HTTP-метод. Как правило, используется GET либо POST, хотя доступны и более экзотические, вроде TRACE/DELETE/PUT и т.п.

- `URL` — адрес запроса. Можно использовать не только http/https, но и другие протоколы, например `ftp://` и `file://`.

  При этом есть ограничения безопасности, называемые «Same Origin Policy»: запрос со страницы можно отправлять только на тот же `протокол://домен:порт`, с которого она пришла. В следующих главах мы рассмотрим, как их можно обойти.

- `async` — если установлено в `false`, то запрос производится синхронно, если `true` — асинхронно.

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

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

- `user`, `password` — логин и пароль для HTTP-авторизации, если нужны.

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

Отослать данные: send()

Синтаксис:

```
xhr.send([body])
```

Именно этод метод открывает соединение и отправляет запрос на сервер.

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

Отмена: abort()

Вызов `xhr.abort()` прерывает выполнение запроса.

Основные свойства, содержащие ответ сервера:

Ответ сервера: status, statusText, responseText

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

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

`responseText`
: Текст ответа сервера.

Есть и ещё одно свойство, которое используется гораздо реже:

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

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

Синхронные и асинхронные запросы

Если в методе `open` установить параметр `async` равным `false` или просто забыть его указать, то запрос будет синхронным.

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

```
                                    // Синхронный запрос
xhr.open('GET', 'phones.json', false);

// Отсылаем его
xhr.send();
// ...весь JavaScript "подвиснет", пока запрос не завершится
```

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

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

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

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

Изменённый JS-код:

```
var xhr = new XMLHttpRequest();

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

xhr.send(); // (1)

xhr.onreadystatechange = function() { // (3)
  if (xhr.readyState != 4) return;
```

```
  button.innerHTML = 'Готово!';

  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText);
  } else {
    alert(xhr.responseText);
  }

}

button.innerHTML = 'Загружаю...'; // (2)
button.disabled = true;
```

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

Событие readystatechange

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

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

Все состояния, по спецификации:

```
const unsigned short UNSENT = 0; // начальное состояние
const unsigned short OPENED = 1; // вызван open
const unsigned short HEADERS_RECEIVED = 2; // получены заголовки
const unsigned short LOADING = 3; // загружается тело (получен очередной
пакет данных)
const unsigned short DONE = 4; // запрос завершён
```

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

Пример ниже демонстрирует переключение между состояниями. В нём сервер отвечает на запрос `digits`, пересылая по строке из 1000 цифр раз в секунду.

Точка разрыва пакетов не гарантирована!!!

При состоянии `readyState=3` (получен очередной пакет) мы можем посмотреть текущие данные в `responseText` и, казалось бы, могли бы работать с этими данными как с «ответом на текущий момент».

Однако, технически мы не управляем разрывами между сетевыми пакетами. Если протестировать пример выше в локальной сети, то в большинстве браузеров разрывы будут каждые 1000 символов, но в реальности пакет может прерваться на любом байте.

Чем это опасно? Хотя бы тем, что символы русского языка в кодировке UTF-8 кодируются двумя байтами каждый — и разрыв может возникнуть *между ними*.

Получится, что при очередном `readyState` в конце `responseText` будет байт-полсимвола, то есть он не будет корректной строкой — частью ответа! Если в скрипте как-то по-особому это не обработать, то неизбежны проблемы.

HTTP-заголовки

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

Для работы с HTTP-заголовками есть 3 метода:

`setRequestHeader(name, value)`

Устанавливает заголовок `name` запроса со значением `value`.

Например:

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

Ограничения на заголовки

Нельзя установить заголовки, которые контролирует браузер, например `Referer` или `Host` и ряд других (полный список тут).

Это ограничение существует в целях безопасности и для контроля корректности запроса.

Поставленный заголовок нельзя снять
Особенностью `XMLHttpRequest` является то, что отменить `setRequestHeader` невозможно.

Повторные вызовы лишь добавляют информацию к заголовку, например:

```
xhr.setRequestHeader('X-Auth', '123');
xhr.setRequestHeader('X-Auth', '456');

// в результате будет заголовок:
// X-Auth: 123, 456
```

`getResponseHeader(name)`

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

Например:

```
xhr.getResponseHeader('Content-Type')
```

`getAllResponseHeaders()`

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

Заголовки возвращаются в виде единой строки, например:

```
Cache-Control: max-age=31536000
Content-Length: 4260
Content-Type: image/png
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

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

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

Тайм-аут

Максимальную продолжительность асинхронного запроса можно задать свойством `timeout`:

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

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

```
xhr.ontimeout = function() {
  alert( 'Извините, запрос превысил максимальное время' );
}
```

Полный список событий

Современная спецификация предусматривает следующие события по ходу обработки запроса:

- `loadstart` — запрос начат.
- `progress` — браузер получил очередной пакет данных, можно прочитать текущие полученные данные в `responseText`.
- `abort` — запрос был отменён вызовом `xhr.abort()`.
- `error` — произошла ошибка.
- `load` — запрос был успешно (без ошибок) завершён.
- `timeout` — запрос был прекращён по таймауту.
- `loadend` — запрос был завершён (успешно или неуспешно)

Используя эти события можно более удобно отслеживать загрузку (`onload`) и ошибку (`onerror`), а также количество загруженных данных (`onprogress`).

Ранее мы видели ещё одно событие — `readystatechange`. Оно появилось гораздо раньше, ещё до появления текущего стандарта.

В современных браузерах от него можно отказаться в пользу других, необходимо лишь, как мы увидим далее, учесть особенности IE8-9.

IE8,9: XDomainRequest

В IE8 и IE9 поддержка `XMLHttpRequest` ограничена:

- Не поддерживаются события, кроме `onreadystatechange`.

- Некорректно поддерживается состояние `readyState = 3`: браузер может сгенерировать его только один раз во время запроса, а не при каждом пакете данных. Кроме того, он не даёт доступ к ответу `responseText` до того, как он будет до конца получен.

Дело в том, что, когда создавались эти браузеры, спецификации были не до конца проработаны. Поэтому разработчики браузера решили добавить свой объект `XDomainRequest`, который реализовывал часть возможностей современного стандарта.

А обычный `XMLHttpRequest` решили не трогать, чтобы ненароком не сломать существующий код. Мы подробнее поговорим про `XDomainRequest` в другой главе.

PRIM (Её на сайте НЕТ. Смотреть в разделах моих локальных носителей: ДЛЯmysite/Полезное/Веб-мастерам/Ajax - Основы `XMLHttpRequest`.doc)**

Пока лишь заметим, что для того, чтобы получить некоторые из современных возможностей в IE8,9 — вместо **`new XMLHttpRequest()`** нужно использовать **`new XDomainRequest`.**

**Кросс-браузерно:

```
var XHR = ("onload" in new XMLHttpRequest()) ? XMLHttpRequest :
XDomainRequest;
var xhr = new XHR();
```

Теперь в IE8,9 поддерживаются события `onload`, `onerror` и `onprogress`. Это именно для IE8,9. Для IE10 обычный `XMLHttpRequest` уже является полноценным.

IE9 и кеширование

Обычно ответы на запросы `XMLHttpRequest` кешируются, как и обычные страницы.

Но IE9- по умолчанию кеширует все ответы, не снабжённые антикеш-заголовком. Другие браузеры этого не делают. Чтобы этого избежать, сервер должен добавить в ответ соответствующие антикеш-заголовки, например `Cache-Control: no-cache`.

Впрочем, использовать заголовки типа `Expires`, `Last-Modified` и `Cache-Control` рекомендуется в любом случае, чтобы дать понять браузеру (не обязательно IE), что ему следует делать.

Альтернативный вариант — добавить в URL запроса случайный параметр, предотвращающий кеширование.

Например, вместо `xhr.open('GET', 'service', false)` написать:

```
xhr.open('GET', ''service?r=' +
Math.random(), false);
```

По историческим причинам такой способ предотвращения кеширования можно увидеть много где, так как старые браузеры плохо обрабатывали кеширующие заголовки. Сейчас серверные заголовки поддерживаются хорошо.

Типовой код для GET-запроса при помощи `XMLHttpRequest`:

ИТОГ:

```
var xhr = new XMLHttpRequest();

xhr.open('GET', '/my/url', true);

xhr.send();

xhr.onreadystatechange = function() {
  if (this.readyState != 4) return;

  // по окончании запроса доступны:
  // status, statusText
  // responseText, responseXML (при content-type: text/xml)

  if (this.status != 200) {
    // обработать ошибку
    alert( 'ошибка: ' + (this.status ? this.statusText || 'запрос не удался')
);
    return;
  }

  // получить результат из this.responseText или this.responseXML
}
```

Мы разобрали следующие методы `XMLHttpRequest`:

- `open(method, url, async, user, password)`
- `send(body)`
- `abort()`
- `setRequestHeader(name, value)`
- `getResponseHeader(name)`
- `getAllResponseHeaders()`

Свойства `XMLHttpRequest`:

- `timeout`
- `responseText`
- `responseXML`
- `status`
- `statusText`

События:

- `onreadystatechange`
- `ontimeout`
- `onerror`
- `onload`
- `onprogress`
- `onabort`
- `onloadstart`
- `onloadend`