

Конспект по теме 3.1 «Асинхронные запросы»

Содержание конспекта

1. Синхронность и асинхронность в JavaScript
2. Протокол передачи гипертекста HTTP
3. Объект XMLHttpRequest
4. Отправка данных с помощью XMLHttpRequest и FormData

1. Синхронность и асинхронность в JavaScript

Синхронность определяет последовательное выполнение кода на JS.

Синхронность кода означает, что описанные нами команды выполняются последовательно, одна за одной. Такой способ удобно использовать в математических операциях. Но важно помнить, что при синхронном запросе страница блокируется, и пользователь не может взаимодействовать с ней: использовать ссылки, кнопки и подобное.

Асинхронность подразумевает возможность выполнять одну группу команд во время ожидания выполнения другой.

JavaScript часто описывается как **асинхронный язык**, что означает, что вызовы функций и операции не блокируют основной поток во время их выполнения. Это верно в некоторых ситуациях, например, реакция на действие пользователя или запрос на сервер, но JavaScript **выполняется в одном потоке** и поэтому имеет **множество синхронных** компонентов.

Пример синхронного JS:

```
let x = 10;
console.log(x);
console.log(x + 1);

// 10
// 11
```

Важно: помните, что при синхронном запросе страница блокируется, посетитель не сможет нажимать ссылки, кнопки и тому подобное.

Пример асинхронного JS:

```
console.log('1')
setTimeout(function afterTwoSeconds() {
  console.log('2')
}, 2000)
console.log('3')
// будет выведена последовательность 1, 3, 2
```

В этом примере callback будет вызван через 2 секунды. Приложение при этом не остановится, ожидая, пока истекнут эти две секунды.

2. Протокол передачи гипертекста HTTP

HTTP — широко распространённый протокол передачи данных, изначально предназначенный для передачи гипертекстовых документов.

Протокол HTTP предполагает использование клиент-серверной структуры передачи данных:

1. Клиентское приложение формирует запрос;
2. Отправляет его на сервер;
3. Сервер формирует ответ;
4. Передаёт его обратно клиенту.

Обычно HTTP запрос содержит:

- **строку запроса**, в которой указывается версия HTTP-протокола, HTTP-метод запроса и запрашиваемый адрес;
- **заголовки**, в которых передаются другие HTTP-параметры для успешного HTTP-соединения;
- **пустую строку**, чтобы отделить служебную информацию от тела сообщения;
- **необязательное** тело сообщения.

Пример HTTP-запроса:

```
GET /index.php HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (X11; U; Linux i686; ru; rv:1.9b5) Gecko/2008050509 Firefox/3.0b5
Accept: text/html
Connection: close
```

Первая строка — это строка запроса, остальные — заголовки.

Пример ответа на запрос:

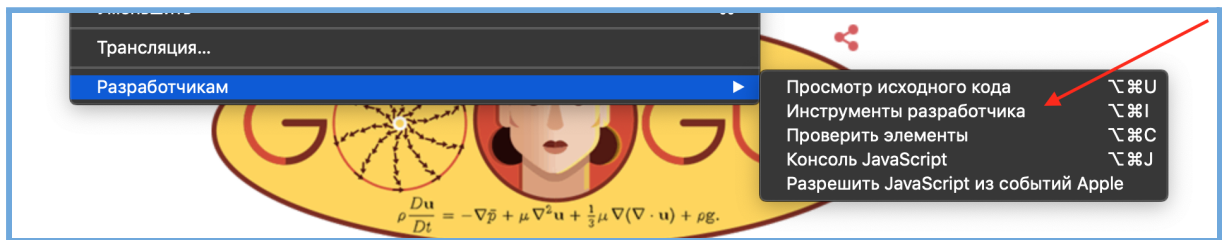
```
HTTP/1.0 200 OK
Server: nginx/0.6.31
Content-Language: ru
Content-Type: text/html; charset=utf-8
Content-Length: 1234
Connection: close
```

Как посмотреть запрос

Посмотреть пример такого запроса можно в браузере: Google Chrome или в любом другом.

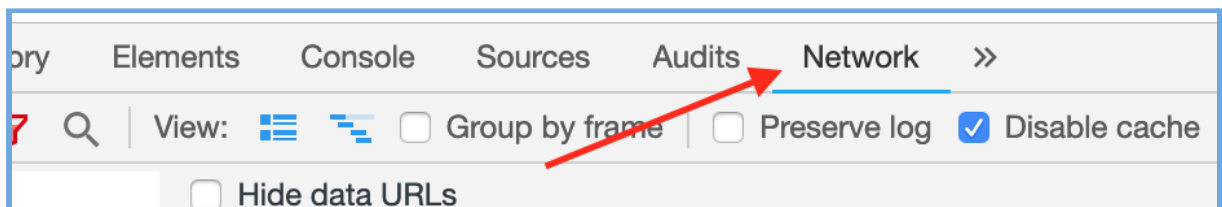
Для этого проходим по следующему пути:

— Посмотреть — > Разработчикам — > Инструменты разработчика — > Network:

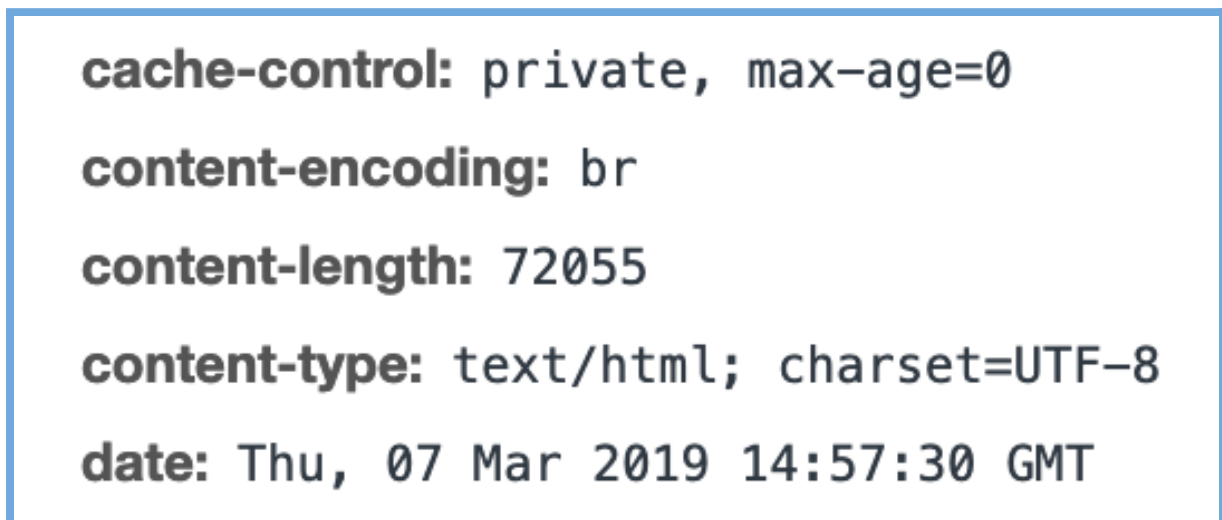


Посмотрим на работу протокола на примере посещения страницы google.com:

1. Браузер устанавливает соединение с сервером и отправляет запрос:



2. Получаем код, список заголовков и тело ответа:



HTTP-заголовки запроса

Заголовки запроса HTTP имеют стандартную для заголовка HTTP структуру: не зависящая от регистра строка, завершаемая двоеточием, и значение, структура которого определяется заголовком. Весь заголовок — одна строка.

Заголовки запроса можно разделить на несколько групп:

- Основные заголовки — General headers;
- Заголовки запроса — Request headers;
- Заголовки сущности — Content-Length.

HTTP-коды состояния

Код состояния HTTP — часть первой строки ответа сервера при запросах по протоколу HTTP.

Это **целое число** из трёх десятичных цифр. Первая цифра указывает на класс состояния.

За кодом ответа обычно следует отделённая пробелом **поясняющая фраза** на английском языке, которая разъясняет человеку причину именно такого ответа.

Код ответа (состояния) HTTP показывает, был ли успешно выполнен определённый HTTP-запрос.

Коды сгруппированы в 5 классов:

- **2xx** — обработка запроса завершилась **успешно**;
- **3xx** — в результате обработки запроса сервер **перенаправляет** нас по другому адресу;
- **4xx** — **ошибка** в самом запросе: указан неверный путь на сервере, нет доступа к запрашиваемому ресурсу;
- **5xx** — произошла **ошибка** на сервере в процессе обработки запроса: ошибка в приложении, обрабатывающем запрос, привела к его падению.

Протокол HTTP предназначен для создания требуемого высокоскоростного обмена данными между пользователем, сервером и поставщиком информации. После программной обработки сервер даёт ответ пользователю в виде предоставленной ответной информации.

3. Объект XMLHttpRequest

Объект **XMLHttpRequest** — это встроенный в браузер объект, с помощью которого можно не только посылать HTTP-запросы к серверу, но и делать это без перезагрузки страниц, что является особенно актуальным для современных веб-приложений и сайтов.

```
// 1. URL запроса;  
// 2. Создаём новый объект XMLHttpRequest;  
// 3. Конфигурируем его: GET-запрос;  
// 4. Отправляем запрос на сервер.
```

Чтобы начать работать с XMLHttpRequest, выполните этот код:


```
let xhr = new XMLHttpRequest();  
// экземпляр объекта XMLHttpRequest  
console.log(xhr);
```

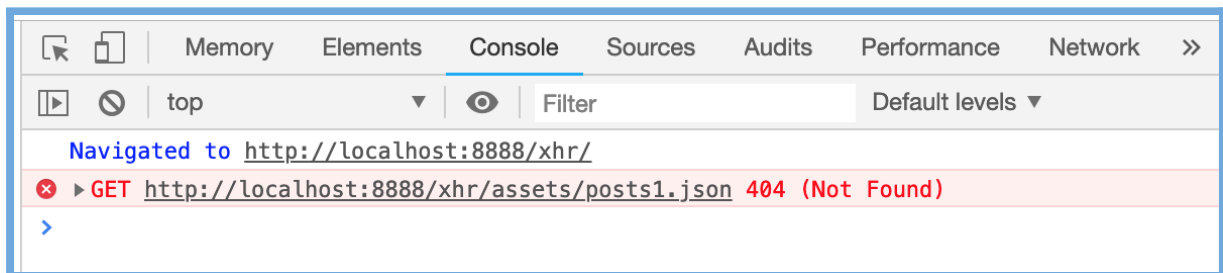
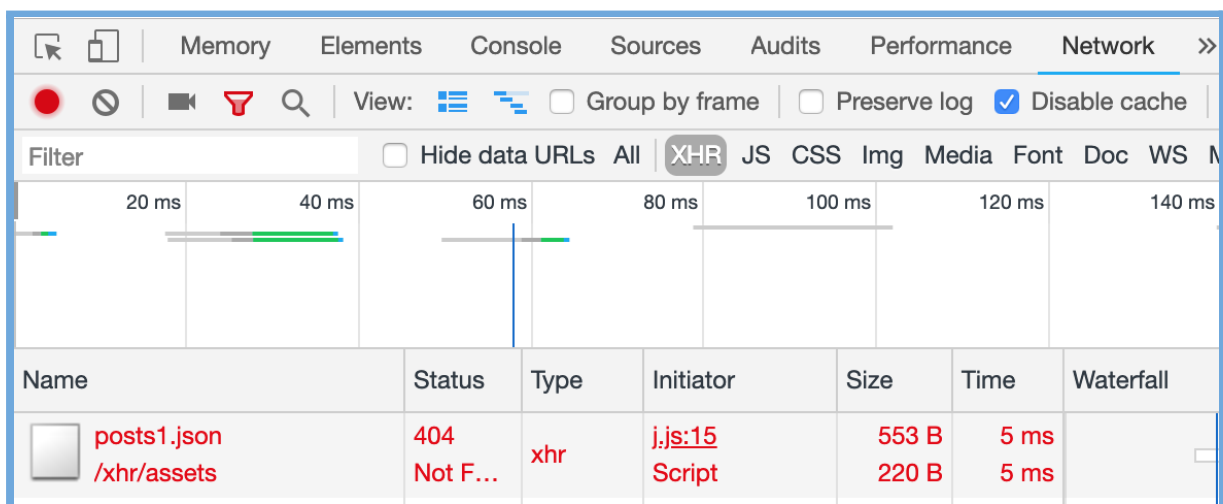
```
XMLHttpRequest {onreadystatechange: null, ready  
▼ State: 0, timeout: 0, withCredentials: false, u  
  pload: XMLHttpRequestUpload, ...} ⓘ  
  onabort: null  
  onerror: null  
  onload: null  
  onloadend: null  
  onloadstart: null  
  onprogress: null  
  onreadystatechange: null  
  ontimeout: null  
  readyState: 0  
  response: ""  
  responseText: ""  
  responseType: ""  
  responseURL: ""
```

Как правило, XMLHttpRequest используют для загрузки данных. Различают два использования XMLHttpRequest — синхронное и асинхронное.


Способы отладки XMLHttpRequest

Для того, чтобы проиграть всевозможные варианты, достаточно снова обратиться к консоли разработчика, которая сообщит, если возникнут ошибки, как на примере ниже:

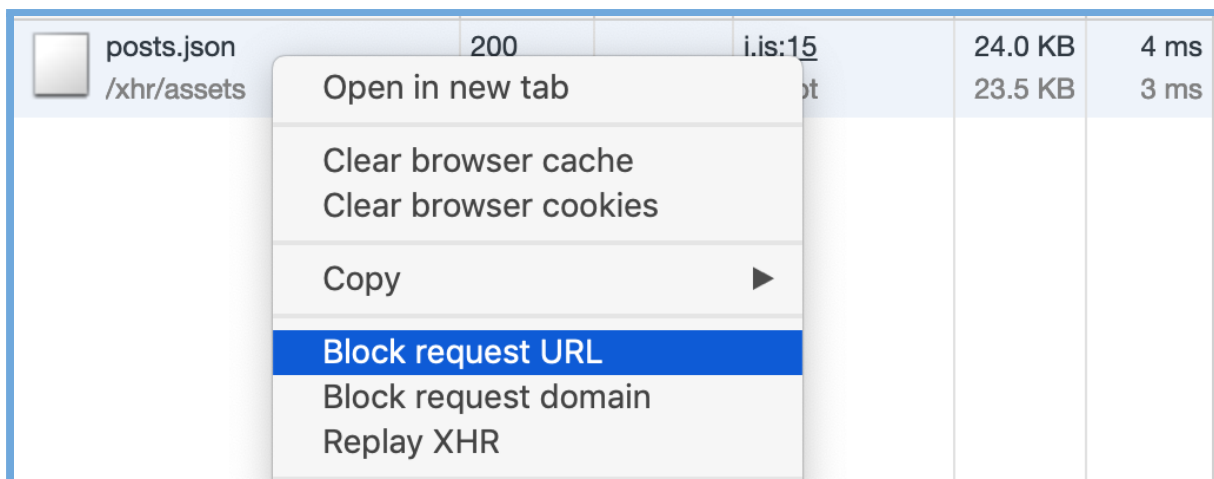
Name	Status	Type	Initiator
 posts.json /xhr/assets	(blocked:devtools)	xhr	j.js:15 Script

The screenshot shows the Chrome DevTools Network tab. At the top, there's a waterfall chart showing the timing of network requests. Below it, a table lists the network requests. The first request is 'posts1.json /xhr/assets' with a status of '404 Not Found', type 'xhr', and initiator 'j.js:15 Script'. The table also shows the size of the request (553 B) and the time taken (5 ms).

Name	Status	Type	Initiator	Size	Time	Waterfall
 posts1.json /xhr/assets	404 Not F...	xhr	j.js:15 Script	553 B 220 B	5 ms 5 ms	

Но также можно симулировать ситуации, например, в случае успешного выполнения запроса и получения требуемого ответа:



Мы можем заблокировать URL запроса, и посмотреть, что в таком случае отобразит браузер, или например заблокировать доступ к запрашиваемому домену. Таким образом вы всегда будете готовы к разным исходам при выполнении запроса.

Основные методы для отправки запросов **XMLHttpRequest**:

- `open()`,
- `send()`.

Создание запроса

Чтобы создать запрос, необходимо описать новый экземпляр класса **XMLHttpRequest** и вызвать для него метод **open** с аргументами **method**, **url** и **async**.

```
let xhr = new XMLHttpRequest( ); // экземпляр объекта XMLHttpRequest
xhr.open(method, URL, async); // создаём запрос
```

- **method** — HTTP-метод создания запроса. Как правило, используется **GET** либо **POST**;
- **URL** — адрес запроса. Можно использовать не только HTTP/HTTPS;
- **async** — позволяет указать способ создания запроса. Если установлено значение `false`, то запрос производится синхронно, а если `true` — асинхронно.

Метод **open** может быть вызван как с тремя аргументами `method`, `url` и `async`, так и только с двумя аргументами `method` и `url`. При отсутствии аргумента `async` метод `open` создаёт запрос асинхронным по умолчанию.

Варианты вызова:

- `open(method, URL);`
- `open(method, URL, async);`

```
xhr.open('GET', url); // асинхронно
xhr.open('GET', url, false); // синхронно
/*
```


При синхронном запросе весь JavaScript подвиснет,
пока запрос не завершится
*/

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

Вывод: метод `open` настраивает запрос на открытие соединения.

Отправка запроса

Отправка запроса происходит благодаря вызову метода `send` для экземпляра объекта `XMLHttpRequest`. Главным требованием для успешной отправки запроса является необходимость предварительного вызова метода `open`. Метод `send` позволяет открыть соединение и отправить запрос. Если запрос асинхронный, каким он является по умолчанию, то возврат из этого метода происходит сразу после отправки запроса:

```
let xhr = new XMLHttpRequest(); // экземпляр объекта XMLHttpRequest
xhr.open('GET', '/xhr/data.txt'); // создаём асинхронный запрос
xhr.send(); // отправляем запрос
```

Посылая серверу запрос, мы хотим выполнить некоторые действия на основе ответа. В этом случае нам необходимо обрабатывать событие, которое как-либо реагировало бы на изменение состояния нашего запроса.

Вывод: именно этот метод открывает соединение и отправляет запрос на сервер.

Метод `setRequestHeader`(name, value) устанавливает значение HTTP-заголовка name равным value:

```
xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')
Устанавливается после метода open(), но до send():
```

```
let xhr = new XMLHttpRequest(); // экземпляр объекта XMLHttpRequest
xhr.open('GET', '/xhr/data.txt'); // создаём асинхронный запрос
xhr.setRequestHeader('Content-Type', 'application/json');
xhr.send(); // отправляем запрос
```

Другие методы для работы с запросами

Также существуют такие методы:

- `abort()` — предотвращает уже отправленный запрос;
- `getResponseHeader()` — возвращает строку, содержащую текст определённого хэдера (header).

Ознакомиться с особенностями методов объекта `XMLHttpRequest` можно на странице [документации](#).

Свойства XMLHttpRequest

Объект **XMLHttpRequest** имеет ряд свойств, которые позволяют проконтролировать выполнение запроса.

Рассмотрим основные свойства, содержащие ответ сервера:

- **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.

Свойства **onreadystatechange** и **readyState**

Когда серверу посылается запрос, мы хотим выполнить некоторые действия на основе ответа.

Событие **onreadystatechange** происходит каждый раз, когда свойство **readyState** (состояние готовности) изменяется.

Свойство **readyState** содержит состояние запроса XMLHttpRequest и принимает значения от 0 до 4:

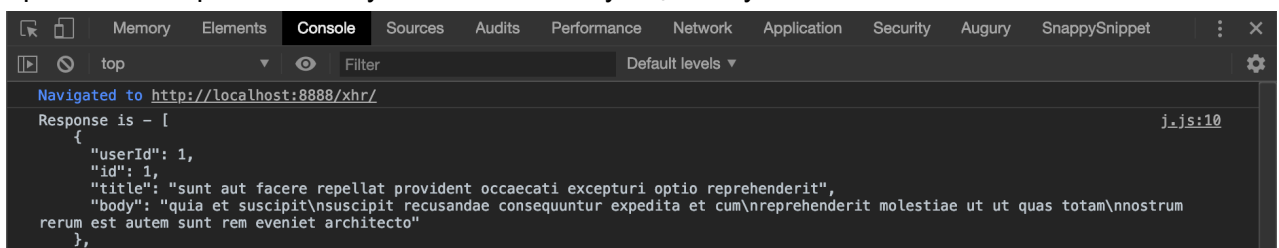
- 0 — запрос не инициализирован,
- 1 — установлено соединение с сервером,
- 2 — запрос получен,
- 3 — обработка запроса,
- 4 — запрос завершён, и ответ готов.

Пример

Рассмотрим пример уже частично известного нам запроса:

```
let xhr = new XMLHttpRequest();
xhr.open('GET', '/assets/posts.json');
xhr.send();
xhr.onreadystatechange = function () {
  if(xhr.readyState === 4) {
    console.log(xhr.responseText);
  }
};
```

При таком запросе мы получаем соответствующий ему ответ:



Свойство **XMLHttpRequest.readyState** возвращает текущее состояние объекта **XMLHttpRequest**.

Объект **XHR** может иметь следующие состояния:

- UNSENT — объект был создан. Метод **open()** ещё не вызывался.
- OPENED — метод **open()** был вызван.
- HEADERS_RECEIVED — метод **send()** был вызван, доступны заголовки (headers) и статус.
- LOADING — загрузка, **responseText** содержит частичные данные;
- DONE — операция полностью завершена.

Пример добавления константы для удобства **xhr.DONE**:

```
xhr.addEventListener('readystatechange', () {  
  if(xhr.readyState === xhr.DONE) console.log(xhr.responseText)  
});
```

Свойство **status** и **statusText**

Свойство **status** содержит статусный код ответа HTTP, который пришёл от сервера:

```
if (xhr.readyState === xhr.DONE && xhr.status === 200) {  
  // инструкция  
}
```

Свойство **statusText** содержит текстовую расшифровку статуса HTTP-ответа.

С помощью статусного кода можно судить об успешности запроса или об ошибках, которые могли бы возникнуть при его выполнении.

Статусы бывают такими:

- Status : 200, **statusText** : «ОК (всё хорошо)»
- Status : 400, **statusText** : «Страница не найдена»

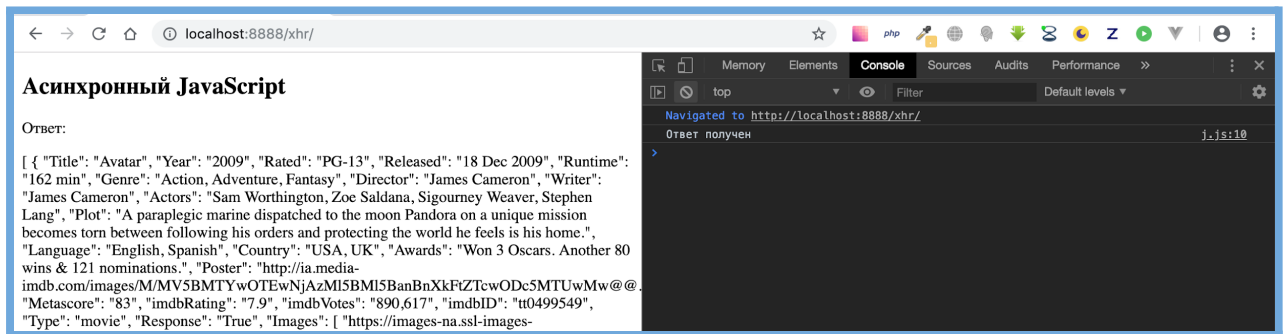
Полный перечень кодов ответов можно найти [здесь](#).

Свойство **responseText** и **responseType**

Только для чтения **XMLHttpRequest** свойство **responseText** возвращает текст ответа. Если ответ сервера не является XML-документом, следует использовать свойство **responseText**:

```
document.getElementById("response").innerHTML=xhr.responseText;
```

Свойство `responseText` возвращает ответ в виде строки, и вы можете далее использовать эту строку соответствующим образом:



Свойство **responseType** является перечислимым значением, которое возвращает тип ответа.

Он также позволяет автору изменять тип ответа.

xhr.responseType

Прежде, чем отправить запрос, необходимо задать для свойства **xhr.responseType** значение **text**, **arraybuffer**, **blob** или **document**.

Обратите внимание: если установить значение **xhr.responseType = ""** или опустить его, по умолчанию выбирается формат **text**. Другие форматы данных можно найти на странице [документации](#).

"arraybuffer"	ArrayBuffer
"blob"	Blob
"document"	Document
"json"	JSON
"text"	DOMString

Свойство **withCredentials** — это Boolean, который определяет, должны ли создаваться кросс-доменные Access-Control запросы с использованием таких идентификационных данных как cookie, авторизационные заголовки или TLS-сертификаты.

Важно: настройка **withCredentials** бесполезна при запросах на тот же домен.

Это часть контроля безопасности, благодаря которому кросс-доменные запросы находятся под пристальным вниманием, и злоумышленникам не получится передать вредоносную информацию.

Важно: по умолчанию браузер не передаёт с запросом куки и авторизующие заголовки.

Чтобы браузер передал вместе с запросом куки и HTTP-авторизацию, нужно поставить запросу `xhr.withCredentials = true`:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', 'http://example.com/', true);
xhr.withCredentials = true;
xhr.send(null);
```

Если бы свойства **withCredentials** не было, то передавать можно было бы без контроля любые данные.

Также используется для определения, будут ли проигнорированы переданные в ответе куки. Значение по умолчанию — `false`. Если оставить его, браузер не пошлёт никаких идентификационных данных.

XMLHttpRequest: индикация прогресса

Запрос XMLHttpRequest состоит из двух фаз:

1. Стадия **закачки** —upload. На ней данные загружаются на сервер.
2. Стадия **скачивания** — download. После того, как данные загружены, браузер скачивает ответ с сервера. На этой стадии используется обработчик **xhr.onprogress**.

Стадии закачки upload

На стадии закачки для того, чтобы получить информацию, используем объект **xhr.upload**. У этого объекта нет методов, он только генерирует события в процессе закачки.

Полный список событий:

- **loadstart**,
- **progress**,
- **abort**,
- **error**,
- **load**,
- **timeout**,
- **loadend**.

```
xhr.upload.onprogress = function() {  
    alert( 'Загружено на сервер' );  
}  
  
xhr.upload.onload = function() {  
    alert( 'Данные полностью загружены на сервер!' );  
}  
  
xhr.upload.onerror = function() {  
    alert( 'Произошла ошибка при загрузке данных на сервер!' );  
}
```

Таким образом, объект **XMLHttpRequest**, или, как его кратко называют, XHR, даёт возможность из JavaScript делать HTTP-запросы к серверу без перезагрузки страницы.

Общий план работы с объектом **XMLHttpRequest** можно представить следующим образом:

1. Создаём экземпляр объекта **XMLHttpRequest**.
2. Открываем соединение с сервером методом **open**.
3. Отправляем запрос методом **send**.

Итоги

Основные методы для отправки запросов **XMLHttpRequest**:

- **open** — Method, Url, async;
- **send** —data;

- `onreadystatechange`.

Ответ сервера находится в:

- `responseText`,
- `responseXML`,
- `status`,
- `statusText`.

4. Отправка данных с помощью XMLHttpRequest и FormData

XMLHttpRequest 2 добавляет поддержку для нового интерфейса **FormData**. Объекты **FormData** позволяют вам легко представлять поля формы и отправлять их с помощью метода **send()**.

Объект **FormData** предназначен для кодирования данных, которые необходимо отправить на сервер посредством **технологии AJAX** (XMLHttpRequest).

Для кодирования данных метод **FormData** использует формат **"multipart/form-data"**. Он позволяет подготовить для отправки по AJAX не только текстовые данные, но и файлы **input** с **type**, равным **file**.

Экземпляр **new FormData([form])** вызывается либо без аргументов, либо с **DOM**-элементом формы.

Предположим, у нас есть некоторый код HTML-формы, хранящей в себе несколько полей ввода, информация с которых должна быть обработана и передана на сервер.

Исходный код формы:

```
<form name="person">
  <input name="name" value="Alex">
  <input name="surname" value="Javascript">
</form>
```

Создание объекта формы:

```
<script>
  // создать объект для формы
  var formData = new FormData(document.forms.person);
</script>
```

В таком случае перед отправкой данных нам необходимо сохранить всю форму в экземпляр класса **FormData**, причем экземпляр класса может быть вызван либо без аргументов, либо с **DOM**-элементом формы.

Структура объекта FormData

Представить себе объект **FormData** можно как набор пар **ключ-значение** — как некоторую коллекцию элементов, в которой каждый представлен в виде ключа и значения (массива значений).

Работа с объектом FormData

Работа с объектом **FormData** начинается с его создания:


```
var formData = new FormData();
```

После создания объекта `FormData` вы можете использовать его различные методы.

Метод `append`

Один из наиболее используемых методов — `append`. Этот метод добавляет в объект `FormData` новую порцию данных (ключ-значение):

```
formData.append(name, value);  
formData.append(name, value, filename).
```

Принцип работы метода заключается в его вызове для объекта `FormData` с двумя аргументами `name` и `value` или же в случае отправки файла с тремя аргументами `name`, `value` и `filename`:

- `name` — HTTP-метод создания запроса,
- `value` — адрес запроса,
- `filename` — способ создания запроса.

Пример использования метода `append`

```
formData.append('key','value1'); //{"key":"value1"}
```

Если указанный ключ уже есть у объекта `FormData`, то этот метод запишет его значение в качестве следующего значения этого ключа:

```
formData.append('key','value2'); //{"key":["value1", "value2"]}
```

Метод `delete`

Для удаления данных из объекта `FormData` предназначен метод `delete`. Он убирает элемент из объекта `FormData` по имени ключа:

```
formData.delete('key');
```

Использование `FormData` для кодирования данных формы

На простом `XMLHttpRequest` примере разберём, как применять объект `FormData` для кодирования данных формы.

Рассмотрим пример простой отправки данных. Для начала создаём экземпляр объекта `FormData`:

```
let formData = new FormData();
```

Далее добавим пару (ключ-значение) с данными, которые собираемся отправить с помощью метода `append()`:

```
formData.append("name", "Alex");
formData.append("age", 25);
```

Далее мы создадим асинхронный запрос с помощью XMLHttpRequest:

```
let xhr = new XMLHttpRequest();
xhr.open("POST", "/handler.php", true);
xhr.send(formData);
/*
    отправим запрос, передав в него экземпляр FormData,
    созданный ранее
*/
```

Реальный пример отправки данных с помощью XMLHttpRequest и FormData

Этот пример будет выполнять следующие основные действия:

- отправлять HTML-форму на сервер;
- обрабатывать данные формы на сервере;
- получать ответ от сервера;
- выводить ответ, обрабатывая посредством JavaScript.

Разработку этого примера начнём с создания HTML-формы и контейнера для вывода результата:

```
<form id="message">
  <label for="name">Имя:</label>
  <input type="text" class="form-control" name="name">

  <label for="name">Сообщение:</label>
  <textarea class="form-control" rows="3" name="message"></textarea>

  <button id="send-message" class="btn btn-primary">Отправить сообщение</button>
</form>
```

Сценарий на JavaScript, который будет кодировать данные HTML-формы (FormData), отправлять её на сервер (XMLHttpRequest), получать ответ с сервера и отображать его на странице в виде маркированного списка:

```
<script>
// получим форму с id = "message"
var message = document.getElementById('message');
message.addEventListener('submit', (e) => {

  var formData = new FormData(message);
  var request = new XMLHttpRequest();
  request.open('POST', 'process.php');
  request.addEventListener('readystatechange', function() {
    if (this.readyState == request.DONE && this.status == 200) {

      var data = JSON.parse(this.responseText);
      var output = '<ul>';
      for (var key in data) {
        output += '<li><b>' + key + "</b>: " + data[key] + '</li>';
```

```
    }  
    output += '</ul>';  
    document.getElementById('result').innerHTML = output;  
  }  
});  
request.send(formData);  
e.preventDefault();  
});  
</script>
```

Материалы, использованные при подготовке:

- [XMLHttpRequest](#)
- [Основы XMLHttpRequest](#)
- [XMLHttpRequest](#)
- [FormData](#)
- [XMLHttpRequest POST, формы и кодировка](#)
- [HTTP](#)