

# Введение в AJAX и COMET



# Что такое AJAX?

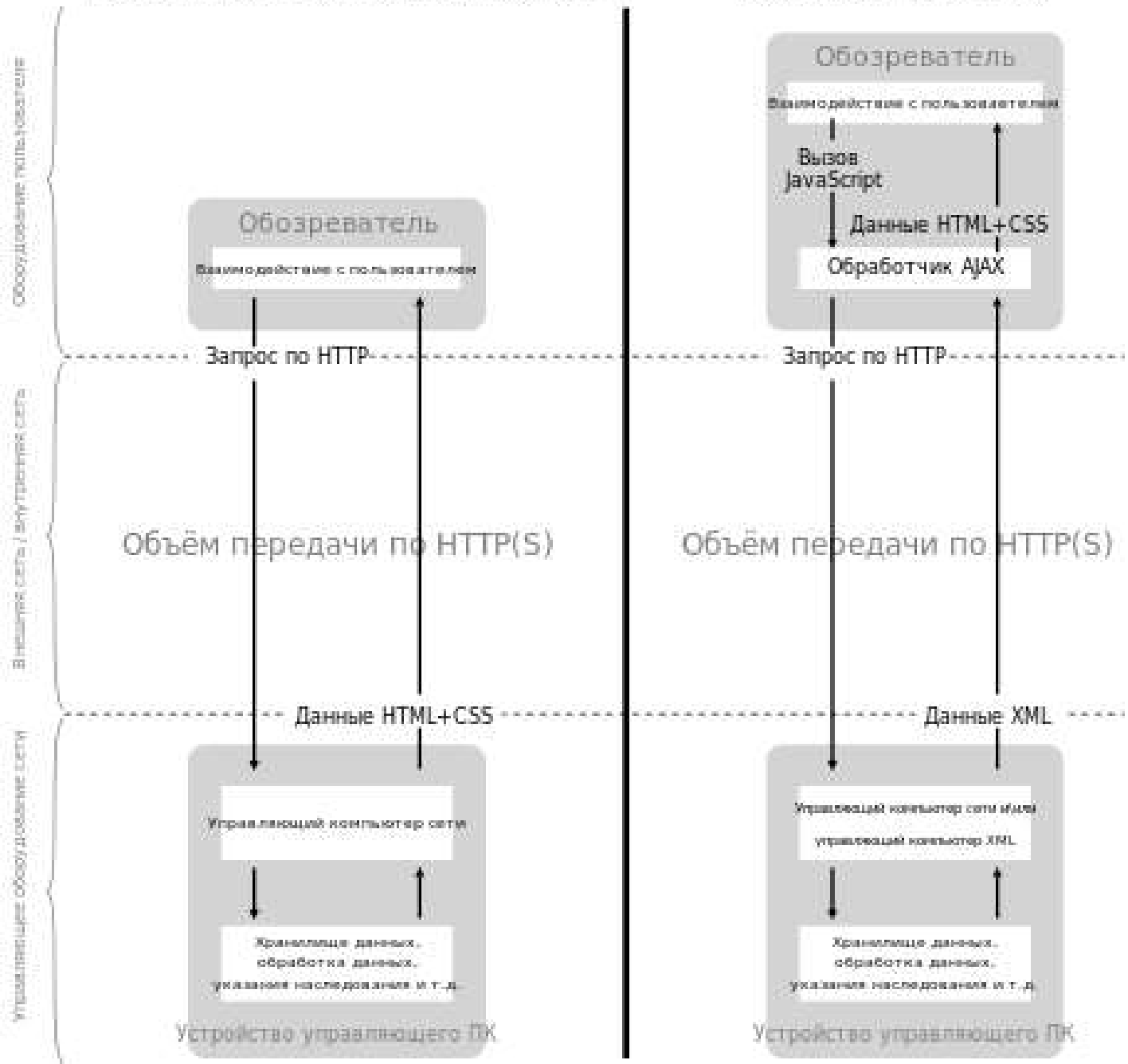
AJAX (аббр. от «**A**synchronous **J**avascript **A**nd **X**ml») – технология обращения к серверу без перезагрузки страницы.

## **В классической модели веб-приложения:**

- Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
- Браузер формирует и отправляет запрос серверу.
- В ответ сервер генерирует совершенно новую веб-страницу и отправляет её браузеру и т. д., после чего браузер полностью перезагружает всю страницу.

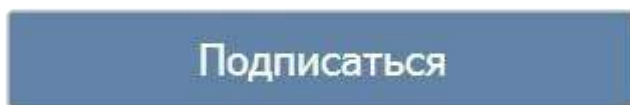
## При использовании AJAX:

- Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
- Скрипт (на языке JavaScript) определяет, какая информация необходима для обновления страницы.
- Браузер отправляет соответствующий запрос на сервер.
- Сервер возвращает только ту часть документа, на которую пришёл запрос.
- Скрипт вносит изменения с учётом полученной информации (без полной перезагрузки страницы).

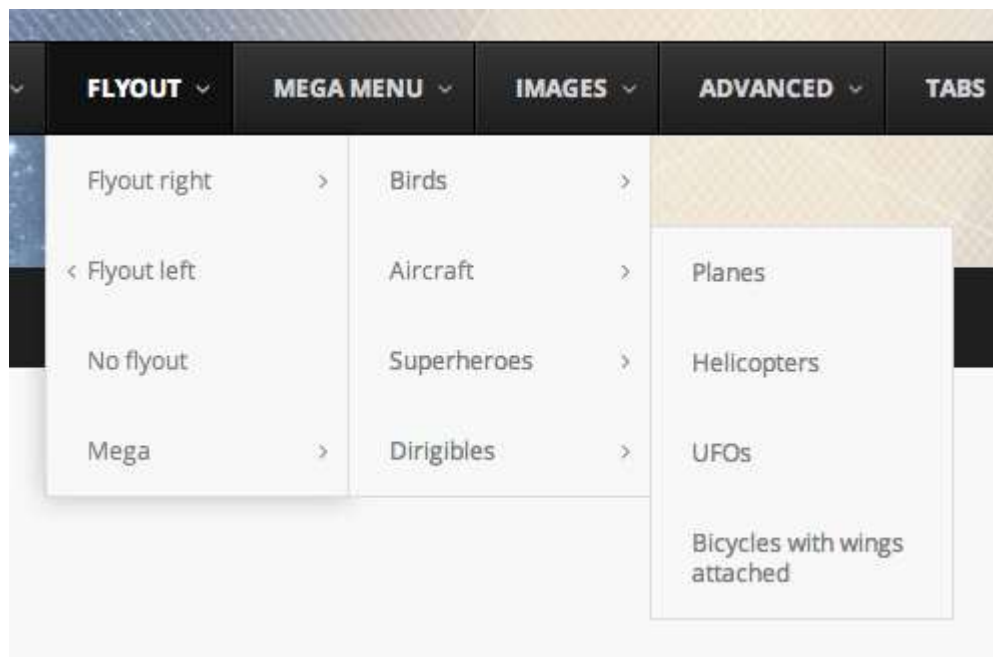


# Что можно сделать с помощью AJAX?

- Элементы интерфейса



- Динамическая подгрузка данных



- **ЖИВОЙ ПОИСК**



wha|

- whale wars**
- what is my ip**
- whats my ip**
- what does my name mean**
- whataburger**
- what is my ip address**
- what time is it**
- what is love**
- what not to wear**
- what the font**

Google Search I'm Feeling Lucky

**Технически, с помощью AJAX можно обмениваться любыми данными с сервером.**

Обычно используются форматы:

- JSON
- XML
- HTML/текст
- Бинарные данные, файлы



# Что такое COMET?

**COMET** – общий термин, описывающий различные техники получения данных по инициативе сервера.

Можно сказать, что AJAX – это «отправил запрос – получил результат», а COMET – это «непрерывный канал, по которому приходят данные».

Существуют библиотеки и фреймворки, добавляющие удобства, например [Socket.io](http://socket.io/) (<http://socket.io/>), [CometD](https://cometd.org/) (<https://cometd.org/>) и другие.

## Примеры COMET-приложений:

- Чат
- Аукцион
- Интерфейс редактирования

На текущий момент технология COMET удобно реализуется во всех браузерах.

# Node.JS для решения задач

Задачи на Ajax требуют взаимодействия с сервером.



Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, [npm](#), is the largest ecosystem of open source libraries in the world.

Download for Windows (x64)

**v6.9.1 LTS**

Recommended For Most Users

**v7.1.0 Current**

Latest Features

[Other Downloads](#) | [Changelog](#) | [API Docs](#) [Other Downloads](#) | [Changelog](#) | [API Docs](#)

# Установка

Для настройки окружения будет достаточно сделать два шага:

1. Установить сервер Node.JS <http://nodejs.org>.
2. Выбрать директорию, в которой будут решаться задачи. Запустить в ней:

**`npm install node-static`**

Это установит в текущую директорию модуль [node-static](#), который станет автоматически доступным для скриптов из поддиректорий.

# Проверка

Создайте поддиректорию и в ней файл `server.js` с содержимым:

```
1 var http = require('http');
2 var static = require('node-static');
3 var file = new static.Server('.');
4
5 http.createServer(function(req, res) {
6   file.serve(req, res);
7 }).listen(8080);
8
9 console.log('Server running on port 8080');
```

- Запустите его: **node server.js**

Должно вывести:

***Server running on port 8080***

⚠ Нельзя запустить больше одного сервера одновременно!

При попытке запуска двух серверов (например, в разных консолях) – будет конфликт портов и ошибка.

- Откройте в браузере  
<http://127.0.0.1:8080/server.js>.



# Примеры

```
1 var http = require('http');
2 var url = require('url');
3 var querystring = require('querystring');
4
5 function accept(req, res) {
6
7   res.writeHead(200, {
8     'Content-Type': 'text/plain',
9     'Cache-Control': 'no-cache'
10  });
11
12   res.end("OK");
13 }
14
15 http.createServer(accept).listen(8080);
```

Для  
краткости

```
1 res.writeHead(200, {
2   'Content-Type': 'text/plain',
3   'Cache-Control': 'no-cache'
4 });
```

# Основные методы

В функции **ассерт** используются два объекта:

**req** – объект запроса («request»), то есть то, что прислал клиент (обычно браузер), из него читаем данные.

**res** – объект ответа («response»), в него пишем данные в ответ клиенту.

- вызов **res.writeHead(HTTP-код, [строка статуса], {заголовки})** пишет заголовки.
- вызов **res.write(txt)** пишет текст в ответ.
- вызов **res.end(txt)** – завершает запрос ответом.



# Примеры

Голосовать!

После нажатия

Ваш голос принят: Sun Nov 13 2016 21:00:40 GMT+0300 (MSK)

`<body>`

```
<button onclick="vote()" id="button">Голосовать!</button>
```

```
<script>
```

```
function vote() {  
    button.innerHTML = ' ... ';
```

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

```
    xhr.open('GET', 'vote', true);
```

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

```
        if (xhr.status != 200) {  
            // обработать ошибку  
            alert('Ошибка ' + xhr.status + ': ' + xhr.statusText);  
            return;  
        }
```

```
        // обработать результат  
        button.innerHTML = xhr.responseText;  
    }
```

```
    xhr.send(null);
```

```
}
```

```
</script>
```

`</body>`

```
function accept(req, res) {
```

```
    // если URL запроса /vote, то...
```

```
    if (req.url == '/vote') {  
        // через 1.5 секунды ответить сообщением
```

```
        setTimeout(function() {  
            res.end('Ваш голос принят: ' + new Date());  
        }, 1500);
```

```
    } else {  
        // иначе считаем это запросом к обычному файлу и выводим его  
        file.serve(req, res); // (если он есть)
```

```
    }
```

```
}
```

# XMLHttpRequest

Объект **XMLHttpRequest** («XHR») дает возможность из JavaScript делать HTTP-запросы к серверу без перезагрузки страницы.

Он может работать с любыми данными, а не только с XML.

# Пример использования

Как правило, XMLHttpRequest используют для загрузки данных.

```
1 // 1. Создаём новый объект XMLHttpRequest
2 var xhr = new XMLHttpRequest();
3
4 // 2. Конфигурируем его: GET-запрос на URL 'phones.json'
5 xhr.open('GET', 'phones.json', false);
6
7 // 3. Отсылаем запрос
8 xhr.send();
9
10 // 4. Если код ответа сервера не 200, то это ошибка
11 if (xhr.status != 200) {
12     // обработать ошибку
13     alert( xhr.status + ': ' + xhr.statusText ); // пример вывода: 404: Not Found
14 } else {
15     // вывести результат
16     alert( xhr.responseText ); // responseText -- текст ответа.
17 }
```

# Результат

Загрузить phones.json!



Подтвердите действие на learn.javascript.ru:

```
[
  {
    "age": 0,
    "id": "motorola-xoom-with-wi-fi",
    "imageUrl": "img/phones/motorola-xoom-with-wi-fi.0.jpg",
    "name": "Motorola XOOM\u2122 with Wi-Fi",
    "snippet": "The Next, Next Generation\r\n\r\nExperience the future with Motorola XOOM with Wi-Fi, the world's first tablet powered by Android 3.0 (Honeycomb).",
  },
  {
    "age": 1,
    "id": "motorola-xoom",
    "imageUrl": "img/phones/motorola-xoom.0.jpg",
    "name": "MOTOROLA XOOM\u2122",
    "snippet": "The Next, Next Generation\r\n\r\nExperience the future with MOTOROLA XOOM, the world's first tablet powered by Android 3.0 (Honeycomb).",
  },
  {
    "age": 2,
    "carrier": "AT&T",
    "id": "motorola-atrix-4g",
    "imageUrl": "img/phones/motorola-atrix-4g.0.jpg",
    "name": "MOTOROLA ATRIX\u2122 4G",
  }
]
```

☐ Предотвратить создание дополнительных диалоговых окон на этой странице.

OK

# Настроить: open

Синтаксис:

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

**method** – HTTP-метод (GET/POST/TRACE/DELETE/PUT и т.п.)

**URL** – адрес запроса (http/https/ftp/file).

**async** – синхронность или асинхронность запросов (false/true).

**user, password** – логин и пароль для HTTP-авторизации, если нужны.

# Отослать данные: send

Синтаксис:

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

В body находится *тело* запроса.

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

# Ответ: **status**, **statusText**, **responseText**

Основные свойства, содержащие ответ сервера:

## **status**

HTTP-код ответа: 200, 404, 403 и др.

## **statusText**

Текстовое описание статуса от сервера: OK, Not Found, Forbidden и др.

## **responseText**

Текст ответа сервера.

## **responseXML**

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)`*.

# Событие readystatechange

Можно посмотреть «текущее состояние запроса» в свойстве xhr.readyState.

Все состояния, по спецификации:

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

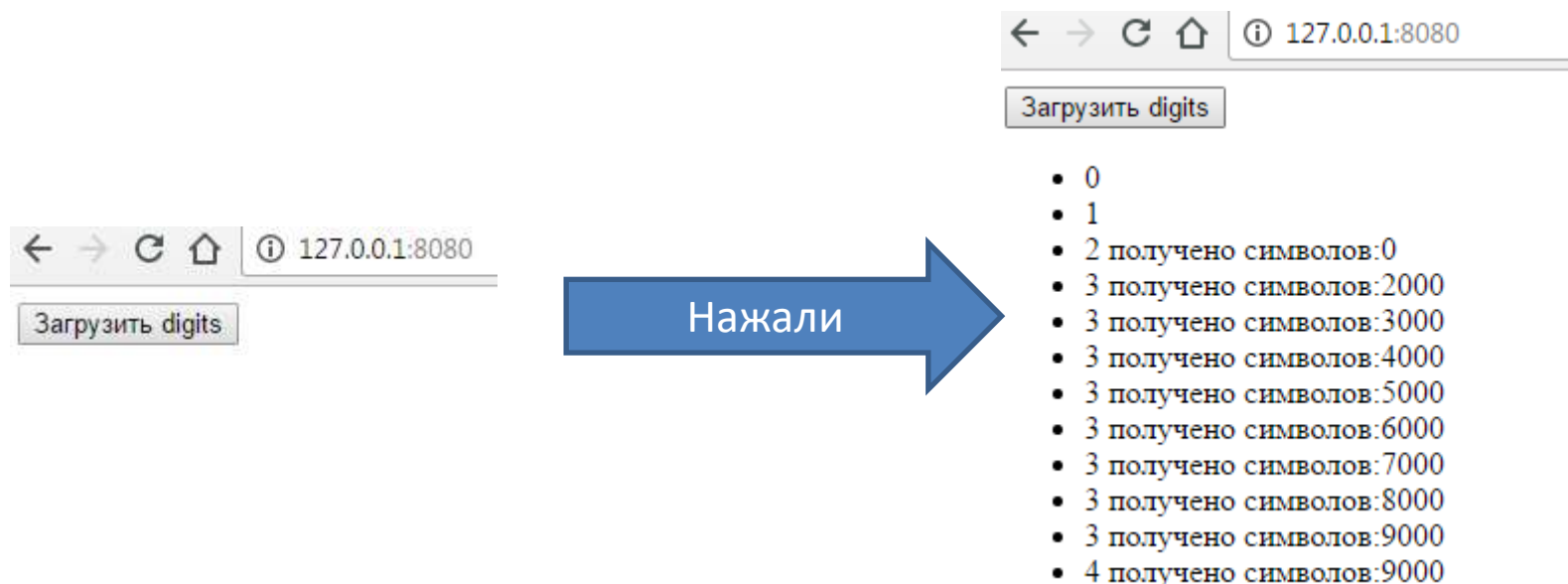
Запрос проходит их в порядке

$0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow \dots \rightarrow 3 \rightarrow 4$ ,

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.



# server.js

```
if (req.url == '/digits') {  
  
    res.writeHead(200, {  
        'Content-Type': 'text/plain',  
        'Cache-Control': 'no-cache'  
    });  
  
    var i = 0;  
  
    var timer = setInterval(write, 1000);  
    write();  
  
    function write() {  
        res.write(new Array(1000).join(++i + ' '));  
        if (i == 9) {  
            clearInterval(timer);  
            res.end();  
        }  
    }  
  
} else {  
    file.serve(req, res);  
}
```

# index.html

```
<button onclick="run()">Загрузить digits</button>

<ul id="log"></ul>

<script>
  function run() {

    var xhr = new XMLHttpRequest();
    write(xhr.readyState);

    xhr.open('GET', 'digits', true);
    write(xhr.readyState);

    xhr.onreadystatechange = function() {
      write(xhr.readyState + " получено символов:" + xhr.responseText.length);
    };

    xhr.send();
  }

  function write(text) {
    var li = log.appendChild(document.createElement('li'));
    li.innerHTML = text;
  }
</script>
```

# HTTP-заголовки

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

Для работы с HTTP-заголовками есть 3 метода:

## 1. **setRequestHeader(name, value)**

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

Например:

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

# Поставленный заголовок нельзя снять

Особенностью XMLHttpRequest является то, что отменить `setRequestHeader` невозможно.

Повторные вызовы лишь добавляют информацию к заголовку, например:

```
1 xhr.setRequestHeader('X-Auth', '123');  
2 xhr.setRequestHeader('X-Auth', '456');  
3  
4 // в результате будет заголовок:  
5 // X-Auth: 123, 456
```

## 2. `getResponseHeader(name)`

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

Например:

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

### 3. 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
```

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

# Свойство `timeout`

Максимальную продолжительность  
асинхронного запроса:

```
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 прочесть 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)

# Задача

Создайте код, который загрузит файл phones.json из текущей директории и выведет все названия телефонов из него в виде списка.

Загрузить phones.json!

Нажали

- Motorola XOOM™ with Wi-Fi
- MOTOROLA XOOM™
- MOTOROLA ATRIX™ 4G
- Dell Streak 7
- Samsung Gem™
- Dell Venue
- Nexus S
- LG Axis
- Samsung Galaxy Tab™
- Samsung Showcase™ a Galaxy S™ phone
- DROID™ 2 Global by Motorola
- DROID™ Pro by Motorola
- MOTOROLA BRAVO™ with MOTOBLUR™
- Motorola DEFY™ with MOTOBLUR™
- T-Mobile myTouch 4G
- Samsung Mesmerize™ a Galaxy S™ phone
- SANYO ZIO
- Samsung Transform™
- T-Mobile G2
- Motorola CHARM™ with MOTOBLUR™

# Кодировка

Во время обычной отправки формы `<form>` браузер собирает значения её полей, делает из них строку и составляет тело GET/POST-запроса для отправки на сервер.

При отправке данных через XMLHttpRequest, это нужно делать самим, в JS-коде.

Большинство проблем и вопросов здесь связано с непониманием, где и какое кодирование нужно осуществлять.

# Кодировка urlencoded

Основной способ кодировки запросов – это *urlencoded*, то есть – стандартное кодирование URL.

Например:

```
1 <form action="/submit" method="GET">
2   <input name="name" value="Ivan">
3   <input name="surname" value="Ivanov">
4 </form>
```

Так как метод GET, итоговый запрос выглядит как

`/submit?name=Ivan&surname=Ivanov`

Все символы, кроме английских букв, цифр и - \_ . ! ~ \* ' ( ) заменяются на их цифровой код в UTF-8 со знаком %.

Например:

```
1 <form action="/submit" method="GET">
2   <input name="name" value="Виктор">
3   <input name="surname" value="Цой">
4 </form>
```

/submit?name=%D0%92%D0%B8%D0%BA%D1%82%D0%BE%D1%80&surname=%D0%A6%D0%BE%D0%B9.

# encodeURIComponent

В JavaScript есть функция **encodeURIComponent** для получения такой кодировки «вручную»:

```
1 alert( encodeURIComponent(' ') ); // %20
2 alert( encodeURIComponent('/') ); // %2F
3 alert( encodeURIComponent('В') ); // %D0%92
4 alert( encodeURIComponent('Виктор') ); // %D0%92%D0%B8%D0%BA%D1%82%D0%BE%D1%80
```

# GET-запрос

Формируя XMLHttpRequest, мы должны формировать запрос «руками», кодируя поля функцией encodeURIComponent.

```
1 // Передаём name и surname в параметрах запроса
2
3 var xhr = new XMLHttpRequest();
4
5 var params = 'name=' + encodeURIComponent(name) +
6             '&surname=' + encodeURIComponent(surname);
7
8 xhr.open("GET", '/submit?' + params, true);
9
10 xhr.onreadystatechange = ...;
11
12 xhr.send();
```

# Сообщаем про AJAX

Запрос, отправленный кодом выше через XMLHttpRequest, никак не отличается от обычной отправки формы. Сервер не в состоянии их отличить.

Поэтому в некоторых фреймворках, чтобы сказать серверу, что это AJAX, добавляют специальный заголовок, например такой:

```
xhr.setRequestHeader("X-Requested-With", "XMLHttpRequest");
```



# POST с urlencoded

В стандартных HTTP-формах для метода POST доступны три кодировки, задаваемые через атрибут `enctype`:

- `application/x-www-form-urlencoded`
- `multipart/form-data`
- `text/plain`

В зависимости от `enctype` браузер кодирует данные соответствующим способом перед отправкой на сервер.

Для примера отправим запрос в кодировке application/x-www-form-urlencoded:

```
1 var xhr = new XMLHttpRequest();
2
3 var body = 'name=' + encodeURIComponent(name) +
4   '&surname=' + encodeURIComponent(surname);
5
6 xhr.open("POST", '/submit', true)
7 xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded')
8
9 xhr.onreadystatechange = ...;
10
11 xhr.send(body);
```

# Кодировка multipart/form-data

Кодировка urlencoded за счёт замены символов на %код может сильно «раздуть» общий объём пересылаемых данных. Поэтому для пересылки файлов используется кодировка: **multipart/form-data**.

В этой кодировке поля пересылаются одно за другим, через строку-разделитель.

Чтобы использовать этот способ, нужно указать его в атрибуте enctype и метод должен быть POST:

```
1 <form action="/submit" method="POST" enctype="multipart/form-data">
2   <input name="name" value="Виктор">
3   <input name="surname" value="Цой">
4 </form>
```

Форма при такой кодировке будет выглядеть примерно так:

```
1  ...Заголовки...
2  Content-Type: multipart/form-data; boundary=RaNdOmDeLiMiTeR
3
4  --RaNdOmDeLiMiTeR
5  Content-Disposition: form-data; name="name"
6
7  Виктор
8  --RaNdOmDeLiMiTeR
9  Content-Disposition: form-data; name="surname"
10
11  Цой
12  --RaNdOmDeLiMiTeR--
```

# POST с multipart/form-data

Сделать POST-запрос в кодировке multipart/form-data можно и через XMLHttpRequest.

Достаточно указать в заголовке Content-Type кодировку и границу, и далее сформировать тело запроса, удовлетворяющее требованиям кодировки.



```
1 var data = {
2     name: 'Виктор',
3     surname: 'Цой'
4 };
5
6 var boundary = String(Math.random()).slice(2);
7 var boundaryMiddle = '--' + boundary + '\r\n';
8 var boundaryLast = '--' + boundary + '--\r\n'
9
10 var body = ['\r\n'];
11 for (var key in data) {
12     // добавление поля
13     body.push('Content-Disposition: form-data; name="' + key + '"\r\n\r\n' + data[key] + '\r\n');
14 }
15
16 body = body.join(boundaryMiddle) + boundaryLast;
17
18 // Тело запроса готово, отправляем
19
20 var xhr = new XMLHttpRequest();
21 xhr.open('POST', '/submit', true);
22
23 xhr.setRequestHeader('Content-Type', 'multipart/form-data; boundary=' + boundary);
24
25 xhr.onreadystatechange = function() {
26     if (this.readyState != 4) return;
27
28     alert( this.responseText );
29 }
30
31 xhr.send(body);
```

# Отправка файла

Можно создать запрос, который сервер воспримет как загрузку файла.

Для добавления файла нужно использовать тот же код, что выше, модифицировав заголовки перед полем, которое является файлом, так:

```
1 Content-Disposition: form-data; name="myfile"; filename="pic.jpg"
2 Content-Type: image/jpeg
3 (пустая строка)
4 содержимое файла
```

# formData

Кодирует формы для отправки на сервер. Это очень удобно. Например:

Этот код отправит на сервер форму с полями name, surname и patronym.

```
1 <form name="person">
2   <input name="name" value="Виктор">
3   <input name="surname" value="Цой">
4 </form>
5
6 <script>
7   // создать объект для формы
8   var formData = new FormData(document.forms.person);
9
10  // добавить к пересылке ещё пару ключ - значение
11  formData.append("patronym", "Робертович");
12
13  // отослать
14  var xhr = new XMLHttpRequest();
15  xhr.open("POST", "/url");
16  xhr.send(formData);
17 </script>
```



Интерфейс:

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

Метод **`formData.append(name, value)`** добавляет данные к форме.

Объект **`formData`** можно сразу отсылать, интеграция `FormData` с `XMLHttpRequest` встроена в браузер. Кодировка при этом будет `multipart/form-data`.

# Другие кодировки

XMLHttpRequest сам по себе не ограничивает кодировку и формат пересылаемых данных.

Поэтому для обмена данными часто используется формат JSON:

```
1 var xhr = new XMLHttpRequest();
2
3 var json = JSON.stringify({
4     name: "Виктор",
5     surname: "Цой"
6 });
7
8 xhr.open("POST", '/submit', true)
9 xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');
10
11 xhr.onreadystatechange = ...;
12
13 // Отсылаем объект в формате JSON и с Content-Type application/json
14 // Сервер должен уметь такой Content-Type принимать и декодировать
15 xhr.send(json);
```

# **XMLHttpRequest:**

## **кросс-доменные запросы**

Обычно запрос XMLHttpRequest может делать запрос только в рамках текущего сайта. При попытке использовать другой домен/порт/протокол – браузер выдаёт ошибку.

# Кросс-доменные запросы

```
1 // (1)
2 var XHR = ("onload" in new XMLHttpRequest()) ? XMLHttpRequest : XDomainRequest;
3
4 var xhr = new XHR();
5
6 // (2) запрос на другой домен :)
7 xhr.open('GET', 'http://anywhere.com/request', true);
8
9 xhr.onload = function() {
10     alert( this.responseText );
11 }
12
13 xhr.onerror = function() {
14     alert( 'Ошибка ' + this.status );
15 }
16
17 xhr.send();
```

1. Мы создаём XMLHttpRequest и проверяем, поддерживает ли он событие onload. Если нет, то это старый XMLHttpRequest, значит это IE8,9, и используем XDomainRequest.
2. Запрос на другой домен отсылается просто указанием соответствующего URL в open. Он обязательно должен быть асинхронным, в остальном – никаких особенностей.

# XMLHttpRequest:

## индикация прогресса

Запрос XMLHttpRequest состоит из двух фаз:

- **Стадия загрузки (upload).** На ней данные загружаются на сервер. Эта фаза может быть долгой для POST-запросов. Для отслеживания прогресса на стадии загрузки существует объект типа **XMLHttpRequestUpload**, доступный как **xhr.upload** и события на нём.
- **Стадия скачивания (download).** После того, как данные загружены, браузер скачивает ответ с сервера. Если он большой, то это может занять существенное время. На этой стадии используется обработчик **xhr.onprogress**.

# Стадия загрузки

На стадии загрузки для получения информации используем объект **xhr.upload**. У этого объекта нет методов, он только генерирует события в процессе загрузки.

Полный список событий:

- loadstart
- progress
- abort
- error
- load
- timeout
- loadend

# Пример установки обработчиков на стадию загрузки

```
1 xhr.upload.onprogress = function(event) {  
2   alert( 'Загружено на сервер ' + event.loaded + ' байт из ' + event.total );  
3 }  
4  
5 xhr.upload.onload = function() {  
6   alert( 'Данные полностью загружены на сервер!' );  
7 }  
8  
9 xhr.upload.onerror = function() {  
10  alert( 'Произошла ошибка при загрузке данных на сервер!' );  
11 }
```

# Стадия скачивания

После того, как загрузка завершена, и сервер соизволит ответить на запрос, XMLHttpRequest начнёт скачивание ответа сервера.

На этой фазе xhr.upload уже не нужен, а в дело вступают обработчики событий на самом объекте xhr. В частности, событие xhr.onprogress содержит информацию о количестве принятых байт ответа.



# Пример обработчика

```
1 xhr.onprogress = function(event) {  
2   alert( 'Получено с сервера ' + event.loaded + ' байт из ' + event.total );  
3 }
```

Все события, возникающие в этих обработчиках, имеют тип **ProgressEvent**, то есть имеют свойства `loaded` – количество уже пересланных данных в байтах и `total` – общее количество данных.

# **Пример: загрузка файла с индикатором прогресса**

Современный XMLHttpRequest позволяет отправить на сервер всё, что угодно. Текст, файл, форму.


# Форма для выбора файла с обработчиком submit:

```
1 <form name="upload">
2   <input type="file" name="myfile">
3   <input type="submit" value="Загрузить">
4 </form>
5
6 <script>
7   document.forms.upload.onsubmit = function() {
8     var input = this.elements.myfile;
9     var file = input.files[0];
10    if (file) {
11      upload(file);
12    }
13    return false;
14  }
15 </script>
```

Получаем файл из формы через свойство `files` элемента `<input>` и передаём его в функцию `upload`:

Этот код отправит файл на сервер и будет сообщать о прогрессе при его загрузке (`xhr.upload.onprogress`), а также об окончании запроса (`xhr.onload`, `xhr.onerror`).

```
1 function upload(file) {  
2  
3     var xhr = new XMLHttpRequest();  
4  
5     // обработчик для загрузки  
6     xhr.upload.onprogress = function(event) {  
7         log(event.loaded + ' / ' + event.total);  
8     }  
9  
10    // обработчики успеха и ошибки  
11    // если status == 200, то это успех, иначе ошибка  
12    xhr.onload = xhr.onerror = function() {  
13        if (this.status == 200) {  
14            log("success");  
15        } else {  
16            log("error " + this.status);  
17        }  
18    };  
19  
20    xhr.open("POST", "upload", true);  
21    xhr.send(file);  
22  
23 }
```

  Файл не выбран  
Прогресс загрузки

09qh8nof2XI.jpg  
Прогресс загрузки



09qh8nof2XI.jpg  
114688 / 140977

09qh8nof2XI.jpg  
success

# Событие onprogress в деталях

Оно представляет собой объект типа **ProgressEvent** со свойствами:

- **Loaded**

Сколько байт уже переслано. Имеется в виду только тело запроса, заголовки не учитываются.

- **lengthComputable**

Если true, то известно полное количество байт для пересылки, и оно хранится в свойстве total.

- **Total**

Общее количество байт для пересылки, если известно.

# А может ли оно быть неизвестно?

1. При загрузке на сервер браузер всегда знает полный размер пересылаемых данных, так что **total** всегда содержит **конкретное количество байт**, а значение **lengthComputable** всегда будет **true**.
2. При скачивании данных – обычно сервер в начале сообщает их общее количество в HTTP-заголовке Content-Length. Но он может и не делать этого, например если сам не знает, сколько данных будет или если генерирует их динамически. Тогда **total** будет равно **0**. А чтобы отличить нулевой размер данных от неизвестного – как раз служит **lengthComputable**, которое в данном случае равно **false**.

**Событие происходит при каждом полученном/отправленном байте, но не чаще чем раз в 50 мс.**

Это обозначено в спецификации progress notifications.



**В процессе получения данных, ещё до их полной передачи, доступен `xhr.responseText`, но он не обязательно содержит корректную строку.**

При пересылке строки в кодировке UTF-8 кириллические символы, как, впрочем, и многие другие, кодируются 2 байтами. Возможно, что в конце одного пакета данных окажется первая половинка символа, а в начале следующего – вторая. Поэтому полагаться на то, что до окончания запроса в `responseText` находится корректная строка нельзя. Она может быть обрезана посередине символа.

*Исключение* – заведомо однобайтные символы, например цифры или латиница.

**Сработавшее событие `xhr.upload.onprogress` не гарантирует, что данные дошли.**

Событие `xhr.upload.onprogress` срабатывает, когда данные отправлены браузером. Но оно не гарантирует, что сервер получил, обработал и записал данные на диск. Он говорит лишь о самом факте отправки.

Поэтому прогресс-индикатор, получаемый при его помощи, носит приблизительный характер.

# Файлы и формы

Выше мы использовали **xhr.send(file)** для передачи файла непосредственно в теле запроса.

При этом посылается только *содержимое* файла. Если нужно дополнительно передать имя файла или что-то ещё — это можно удобно сделать через форму, при помощи объекта **FormData**:

Создадим форму formData и прибавим к ней поле с файлом file и именем "myfile":

```
1 var formData = new FormData();  
2 formData.append("myfile", file);  
3 xhr.send(formData);
```

Данные будут отправлены в кодировке multipart/form-data. Серверный фреймворк увидит это как обычную форму с файлом, практически все серверные технологии имеют их встроенную поддержку. Индикация прогресса реализуется точно так же.

# **XMLHttpRequest:**

## **возобновляемая загрузка**

Современный XMLHttpRequest даёт возможность загружать файл как угодно: во множество потоков, с догрузкой, с подсчётом контрольной суммы и т.п.

Рассмотрим общий подход к организации загрузки, его уже можно расширять, адаптировать к своему фреймворку и так далее.

Поддержка – все браузеры кроме IE9-.

# Неточный `upload.onprogress`

Ранее мы рассматривали загрузку с индикатором прогресса.

**Однако `onprogress` годится лишь для красивого рисования прогресса.**

Для загрузки нужно точно знать количество загруженных байт. Это может сообщить только сервер.

# Алгоритм возобновляемой загрузки

Загрузкой файла будет заведовать объект `Uploader`, его примерный общий вид:

```
1 function Uploader(file, onSuccess, onFail, onProgress) {  
2  
3     var fileId = file.name + '-' + file.size + '-' + +file.lastModifiedDate;  
4  
5     var errorCount = 0;  
6  
7     var MAX_ERROR_COUNT = 6;  
8  
9     function upload() {  
10         ...  
11     }  
12  
13     function pause() {  
14         ...  
15     }  
16  
17     this.upload = upload;  
18     this.pause = pause;  
19 }
```

# Аргументы для new Uploader:

## **file**

Объект File API. Может быть получен из формы, либо как результат Drag'n'Drop.

## **onSuccess, onFail, onProgress**

Функции-коллбэки, которые будут вызываться в процессе (`onProgress`) и при окончании загрузки.



## **fileId**

Уникальный идентификатор файла, генерируется по имени, размеру и дате модификации. По нему мы всегда сможем возобновить загрузку, в том числе и после закрытия и открытия браузера.

## **startByte**

С какого байта загружать. Изначально – с нулевого.

## **errorCount / MAX\_ERROR\_COUNT**

Текущее число ошибок / максимальное число ошибок подряд, после которого загрузка считается проваленной.

# Алгоритм загрузки:

1. Генерируем fileId из названия, размера, даты модификации файла. Можно добавить и идентификатор посетителя.
2. Спрашиваем сервер, есть ли уже такой файл, и если да – сколько байт уже загружено?
3. Отсылаем файл с позиции, которую сказал сервер.

При этом загрузку можно прервать в любой момент, просто оборвав все запросы.

Выберите файл PDFWriter.exe

Загрузить

Пауза

progress 393216 / 1919072



Выберите файл PDFWriter.exe

Загрузить

Пауза

success



# **COMET с XMLHttpRequest: длинные опросы**

Рассмотрим способ организации COMET, то есть непрерывного получения данных с сервера, который очень прост и подходит в 90% реальных случаев.

# Частые опросы

Первое решение, которое приходит в голову для непрерывного получения событий с сервера – это «частые опросы» (polling), т.е периодические запросы на сервер: «я тут, изменилось ли что-нибудь?». Например, раз в 10 секунд.

- В ответ сервер во-первых помечает у себя, что клиент онлайн;
- Во-вторых посылает сообщение, в котором в специальном формате содержится весь пакет событий, накопившихся к данному моменту.

## Недостатки

- Задержки между событием и уведомлением.
- Лишний трафик и запросы на сервер.

## Достоинства

- Простота реализации.

# Длинные опросы

Длинные опросы – отличная альтернатива частым опросам. Они также удобны в реализации, и при этом сообщения доставляются без задержек.

## Схема:

- Отправляется запрос на сервер.
- Соединение не закрывается сервером, пока не появится сообщение.
- Когда сообщение появилось – сервер отвечает на запрос, пересылая данные.
- Браузер тут же делает новый запрос.

Ситуация, когда браузер отправил запрос и держит соединение с сервером, ожидая ответа, является стандартной и прерывается только доставкой сообщений.

Схема коммуникации:



При этом если соединение рвётся само, например, из-за ошибки в сети, то браузер тут же отправляет новый запрос.

## Примерный код клиентской части:

```
1 function subscribe(url) {  
2     var xhr = new XMLHttpRequest();  
3  
4     xhr.onreadystatechange = function() {  
5         if (this.readyState != 4) return;  
6  
7         if (this.status == 200) {  
8             onMessage(this.responseText);  
9         } else {  
10            onError(this);  
11        }  
12  
13        subscribe(url);  
14    }  
15    xhr.open("GET", url, true);  
16    xhr.send();  
17 }
```

Функция `subscribe` делает запрос, при ответе обрабатывает результат, и тут же запускает процесс по новой.



# Пример: чат

Несколько человек при заходе на эту страницу будут получать сообщения друг друга.

Несколько человек при заходе на эту страницу будут получать сообщения друг друга.

Как дела?

# WebSocket

Протокол WebSocket (стандарт RFC 6455) предназначен для решения любых задач и снятия ограничений обмена данными между браузером и сервером.

Он позволяет пересылать любые данные, на любой домен, безопасно и почти без лишнего сетевого трафика.

# Пример браузерного кода

Для открытия соединения достаточно создать объект WebSocket, указав в нём специальный протокол ws.:

```
1 var socket = new WebSocket("ws://javascript.ru/ws");
```

У объекта socket есть четыре коллбэка:

```
1 socket.onopen = function() {  
2   alert("Соединение установлено.");  
3 };  
4  
5 socket.onclose = function(event) {  
6   if (event.wasClean) {  
7     alert('Соединение закрыто чисто');  
8   } else {  
9     alert('Обрыв соединения'); // например, "убит" процесс сервера  
10  }  
11  alert('Код: ' + event.code + ' причина: ' + event.reason);  
12 };  
13  
14 socket.onmessage = function(event) {  
15   alert("Получены данные " + event.data);  
16 };  
17  
18 socket.onerror = function(error) {  
19   alert("Ошибка " + error.message);  
20 };
```

Для отправки данных используется метод **socket.send(data)**. Пересылать можно любые данные.

Например, строку:

```
socket.send("Привет");
```

...Или файл, выбранный в форме:

```
socket.send(form.elements[0].file);
```

Для того, чтобы коммуникация была успешной, сервер должен поддерживать протокол **WebSocket**.

# Установление WebSocket-соединения

Протокол WebSocket работает *над* HTTP.

Это означает, что при соединении браузер отправляет специальные заголовки, спрашивая: «поддерживает ли сервер WebSocket?».

Если сервер в ответных заголовках отвечает «да, поддерживаю», то дальше HTTP прекращается и общение идёт на специальном протоколе WebSocket, который уже не имеет с HTTP ничего общего.

# Установка соединения

Пример запроса от браузера при создании нового объекта

`new WebSocket("ws://server.example.com/chat"):`

```
1 GET /chat HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Origin: http://javascript.ru
6 Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
7 Sec-WebSocket-Version: 13
```

Описания заголовков:

## **GET, Host**

Стандартные HTTP-заголовки из URL запроса

## **Upgrade, Connection**

Указывают, что браузер хочет перейти на websocket.

## **Origin**

Протокол, домен и порт, откуда отправлен запрос.

## **Sec-WebSocket-Key**

Случайный ключ, который генерируется браузером: 16 байт в кодировке [Base64](#).

## **Sec-WebSocket-Version**

Версия протокола. Текущая версия: 13.

Все заголовки, кроме GET и Host, браузер генерирует сам, без возможности вмешательства JavaScript.



**Сервер может проанализировать эти заголовки и решить, разрешает ли он WebSocket с данного домена Origin.**

Ответ сервера, если он понимает и разрешает WebSocket-подключение:

```
1 HTTP/1.1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: hsB1buDTkk24srzE0TBUlZA1C2g=
```

# Протокол JSONP

Если создать тег `<script src>`, то при добавлении в документ запустится процесс загрузки `src`. В ответ сервер может прислать скрипт, содержащий нужные данные.

Таким образом можно запрашивать данные с любого сервера, в любом браузере, без каких-либо разрешений и дополнительных проверок.

Протокол JSONP – это «надстройка» над таким способом коммуникации.

# Запрос

Простейший пример запроса:

```
1 function addScript(src) {  
2   var elem = document.createElement("script");  
3   elem.src = src;  
4   document.head.appendChild(elem);  
5 }  
6  
7 addScript('user?id=123');
```

Такой вызов добавит в <head> документа тег:

```
1 <script src="/user?id=123"></script>
```

При добавлении тега <script> с внешним src в документ браузер тут же начинает его скачивать, а затем – выполняет.

В данном случае браузер запросит скрипт с URL /user?id=123 и выполнит.

# Обработка ответа, JSONP

Допустим, сервер хочет прислать объект с данными.

Конечно, он может присвоить её в переменную, например так:

```
// ответ сервера
```

```
var user = {name: "Вася", age: 25 };
```

...А браузер по script.onload отловит окончание загрузки и прочитает значение user.

Но что, если одновременно делается несколько запросов? Получается, нужно присваивать в разные переменные.

Протокол JSONP как раз и призван облегчить эту задачу.

- Вместе с запросом клиент в специальном, заранее оговорённом, параметре передаёт название функции.

Обычно такой параметр называется `callback`. Например :

**`addScript('user?id=123&callback=onUserData');`**

- Сервер кодирует данные в JSON и оборачивает их в вызов функции, название которой получает из параметра `callback`:

// ответ сервера

```
onUserData({  
    name: "Вася",  
    age: 25  
});
```

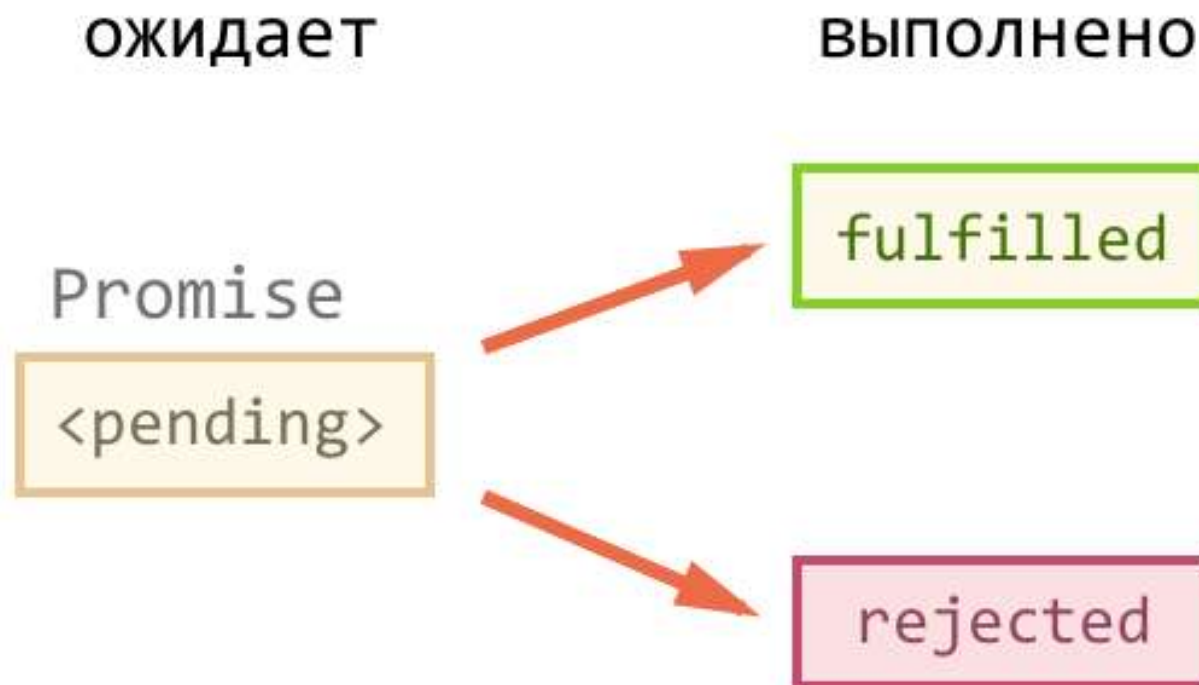
Это и называется JSONP («JSON with Padding»).

# Promise

Удобный способ организации асинхронного кода.

**Promise** – это специальный объект, который содержит своё состояние.

Вначале pending («ожидание»), затем – одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).



На promise можно навешивать коллбэки двух типов:

- onFulfilled – срабатывают, когда promise в состоянии «выполнен успешно».
- onRejected – срабатывают, когда promise в состоянии «выполнен с ошибкой».



Способ использования, в общих чертах, такой:

- Код, которому надо сделать что-то асинхронно, создаёт объект `promise` и возвращает его.
- Внешний код, получив `promise`, навешивает на него обработчики.
- По завершении процесса асинхронный код переводит `promise` в состояние `fulfilled` (с результатом) или `rejected` (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.

# Синтаксис создания Promise:

```
1 var promise = new Promise(function(resolve, reject) {  
2     // Эта функция будет вызвана автоматически  
3  
4     // В ней можно делать любые асинхронные операции,  
5     // А когда они завершатся – нужно вызвать одно из:  
6     // resolve(результат) при успешном выполнении  
7     // reject(ошибка) при ошибке  
8 })
```

Универсальный метод для навешивания обработчиков:

**`promise.then(onFulfilled, onRejected)`**

`onFulfilled` – функция, которая будет вызвана с результатом при `resolve`.

`onRejected` – функция, которая будет вызвана с ошибкой при `reject`.

С его помощью можно назначить как оба обработчика сразу, так и только один:

```
1 // onFulfilled сработает при успешном выполнении
2 promise.then(onFulfilled)
3 // onRejected сработает при ошибке
4 promise.then(null, onRejected)
```

# **.catch**

Для того, чтобы поставить обработчик только на ошибку, вместо

**.then(null, onRejected)**

МОЖНО НАПИСАТЬ

**.catch(onRejected)**

Синхронный throw – то же самое, что reject

Если в функции промиса происходит синхронный throw (или иная ошибка), то вызывается reject:

```
1  'use strict';
2
3  let p = new Promise((resolve, reject) => {
4    // то же что reject(new Error("o_o"))
5    throw new Error("o_o");
6  })
7
8  p.catch(alert); // Error: o_o
```

```
1 'use strict';
2
3 // Создаётся объект promise
4 let promise = new Promise((resolve, reject) => {
5
6     setTimeout(() => {
7         // переводит промис в состояние fulfilled с результатом "result"
8         resolve("result");
9     }, 1000);
10
11 });
12
13 // promise.then навешивает обработчики на успешный результат или ошибку
14 promise
15     .then(
16         result => {
17             // первая функция-обработчик - запустится при вызове resolve
18             alert("Fulfilled: " + result); // result - аргумент resolve
19         },
20         error => {
21             // вторая функция - запустится при вызове reject
22             alert("Rejected: " + error); // error - аргумент reject
23         }
24     );
```

```
1 // Этот promise завершится с ошибкой через 1 секунду
2 var promise = new Promise((resolve, reject) => {
3
4     setTimeout(() => {
5         reject(new Error("время вышло!"));
6     }, 1000);
7
8 });
9
10 promise
11     .then(
12         result => alert("Fulfilled: " + result),
13         error => alert("Rejected: " + error.message) // Rejected: время вышло!
14     );
```



Функции `resolve/reject` принимают ровно один аргумент – результат/ошибку.

Именно он передаётся обработчикам в `.then`

# **Promise после reject/resolve – неизменны**

Заметим, что после вызова resolve/reject промис уже не может «передумать».

Когда промис переходит в состояние «выполнен» – с результатом (resolve) или ошибкой (reject) – это навсегда.

```
1 'use strict';
2
3 let promise = new Promise((resolve, reject) => {
4
5     // через 1 секунду готов результат: result
6     setTimeout(() => resolve("result"), 1000);
7
8     // через 2 секунды – reject с ошибкой, он будет проигнорирован
9     setTimeout(() => reject(new Error("ignored")), 2000);
10
11 });
12
13 promise
14     .then(
15         result => alert("Fulfilled: " + result), // сработает
16         error => alert("Rejected: " + error) // не сработает
17     );
```

# Промисификация

*Промисификация* – это когда берут асинхронный функционал и делают для него обёртку, возвращающую промис.

После промисификации использование функционала зачастую становится гораздо удобнее.

В качестве примера сделаем такую обёртку для запросов при помощи XMLHttpRequest.

```
1 function httpGet(url) {  
2  
3     return new Promise(function(resolve, reject) {  
4  
5         var xhr = new XMLHttpRequest();  
6         xhr.open('GET', url, true);  
7  
8         xhr.onload = function() {  
9             if (this.status == 200) {  
10                 resolve(this.response);  
11             } else {  
12                 var error = new Error(this.statusText);  
13                 error.code = this.status;  
14                 reject(error);  
15             }  
16         };  
17  
18         xhr.onerror = function() {  
19             reject(new Error("Network Error"));  
20         };  
21  
22         xhr.send();  
23     });
```

Функция httpGet(url) будет возвращать промис, который при успешной загрузке данных с url будет переходить в fulfilled с этими данными, а при ошибке – в rejected с информацией об ошибке

Как видно, внутри функции объект XMLHttpRequest создаётся и отправляется как обычно, при onload/onerror вызываются, соответственно, resolve (при статусе 200) или reject.

- Использование:

```
1 httpGet("/article/promise/user.json")
2   .then(
3     response => alert(`Fulfilled: ${response}`),
4     error => alert(`Rejected: ${error}`)
5   );
```

Fulfilled: {  
 "name": "iliakan",  
 "isAdmin": true  
}

OK

# Метод **fetch**

Метод **fetch** – это XMLHttpRequest нового поколения. Он предоставляет улучшенный интерфейс для осуществления запросов к серверу: как по части возможностей и контроля над происходящим, так и по синтаксису, так как построен на промисах.

# Синтаксис метода fetch:

```
let promise = fetch(url[, options]);
```

**url** – URL, на который сделать запрос,

**options** – необязательный объект с настройками запроса.



## Свойства options:

- `method` – метод запроса,
- `headers` – заголовки запроса (объект),
- `body` – тело запроса: `FormData`, `Blob`, строка и т.п.
- `mode` – одно из: «`same-origin`», «`no-cors`», «`cors`», указывает, в каком режиме кросс-доменности предполагается делать запрос.
- `credentials` – одно из: «`omit`», «`same-origin`», «`include`», указывает, пересылать ли куки и заголовки авторизации вместе с запросом.
- `cache` – одно из «`default`», «`no-store`», «`reload`», «`no-cache`», «`force-cache`», «`only-if-cached`», указывает, как кешировать запрос.
- `redirect` – можно поставить «`follow`» для обычного поведения при коде 30х (следовать редиректу) или «`error`» для интерпретации редиректа как ошибки.

# Использование

- При вызове `fetch` возвращает промис, который, когда получен ответ, выполняет коллбэки с объектом `Response` или с ошибкой, если запрос не удался.

```
1 'use strict';
2
3 fetch('/article/fetch/user.json')
4   .then(function(response) {
5     alert(response.headers.get('Content-Type')); // application/json; charset=utf-8
6     alert(response.status); // 200
7
8     return response.json();
9   })
10  .then(function(user) {
11    alert(user.name); // iliakan
12  })
13  .catch( alert );
```

application/json; charset=utf-8

OK

200

☐ Не давать этой странице создавать дополнительные диалоговые окна

OK

iliakan

☐ Не давать этой странице создавать дополнительные диалоговые окна

OK

Объект `response` кроме доступа к заголовкам `headers`, статусу `status` и некоторым другим полям ответа, даёт возможность прочитать его тело, в желаемом формате.

Варианты описаны в спецификации [Body](#), они включают в себя:

- `response.arrayBuffer()`
- `response.blob()`
- `response.formData()`
- `response.json()`
- `response.text()`

Соответствующий вызов возвращает промис, который, когда ответ будет получен, вызовет коллбэк с результатом.

В примере выше мы можем в первом `.then` проанализировать ответ и, если он нас устроит – вернуть промис с нужным форматом. Следующий `.then` уже будет содержать полный ответ сервера.

- <https://github.com/github/fetch>
- <https://fetch.spec.whatwg.org/>
- <https://fetch.spec.whatwg.org/#response>