Протокол WebSocket (стандарт [RFC 6455](http://tools.ietf.org/html/rfc6455)) 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](https://learn.javascript.ru/websockets" \l "primer-brauzernogo-koda)**

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

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

У объекта socket есть четыре колбэка: один при получении данных и три – при изменениях в состоянии соединения:

socket.onopen = function() {

alert("Соединение установлено.");

};

socket.onclose = function(event) {

if (event.wasClean) {

alert('Соединение закрыто чисто');

} else {

alert('Обрыв соединения'); // например, "убит" процесс сервера

}

alert('Код: ' + event.code + ' причина: ' + event.reason);

};

socket.onmessage = function(event) {

alert("Получены данные " + event.data);

};

socket.onerror = function(error) {

alert("Ошибка " + error.message);

};

**Для посылки 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);

Просто, не правда ли? Выбираем, что переслать, и socket.send().

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

Чтобы лучше понимать происходящее – посмотрим, как он устроен.

**[Установление WebSocket-соединения](https://learn.javascript.ru/websockets" \l "ustanovlenie-websocket-soedineniya)**

Протокол WebSocket работает *над* TCP.

Это означает, что при соединении браузер отправляет по HTTP 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.

**[Установление соединения](https://learn.javascript.ru/websockets" \l "ustanovlenie-soedineniya)**

Пример запроса от браузера при создании нового объекта new WebSocket("ws://server.example.com/chat"):

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Origin: http://javascript.ru

Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==

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](http://ru.wikipedia.org/wiki/Base64).

**Sec-WebSocket-Version**

Версия протокола. Текущая версия: 13.

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

**Такой XMLHttpRequest создать нельзя**

Создать подобный XMLHttpRequest-запрос (подделать WebSocket) невозможно, по одной простой причине: указанные выше заголовки запрещены к установке методом setRequestHeader.

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

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

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=

Здесь строка Sec-WebSocket-Accept представляет собой перекодированный по специальному алгоритму ключ Sec-WebSocket-Key. Браузер использует её для проверки, что ответ предназначается именно ему.

Затем данные передаются по специальному протоколу, структура которого («фреймы») изложена далее. И это уже совсем не HTTP.

**[Расширения и подпротоколы](https://learn.javascript.ru/websockets" \l "rasshireniya-i-podprotokoly)**

Также возможны дополнительные заголовки Sec-WebSocket-Extensions и Sec-WebSocket-Protocol, описывающие расширения и подпротоколы (subprotocol), которые поддерживает данный клиент.

Посмотрим разницу между ними на двух примерах:

* Заголовок Sec-WebSocket-Extensions: deflate-frame означает, что браузер поддерживает модификацию протокола, обеспечивающую сжатие данных.

Это говорит не о самих данных, а об улучшении способа их передачи. Браузер сам формирует этот заголовок.

* Заголовок Sec-WebSocket-Protocol: soap, wamp говорит о том, что по WebSocket браузер собирается передавать не просто какие-то данные, а данные в протоколах [SOAP](http://ru.wikipedia.org/wiki/SOAP) или WAMP («The WebSocket Application Messaging Protocol»). Стандартные подпротоколы регистрируются в специальном каталоге [IANA](http://www.iana.org/assignments/websocket/websocket.xml).

Этот заголовок браузер поставит, если указать второй необязательный параметр WebSocket:

var socket = new WebSocket("ws://javascript.ru/ws", ["soap", "wamp"]);

При наличии таких заголовков сервер может выбрать расширения и подпротоколы, которые он поддерживает, и ответить с ними.

Например, запрос:

GET /chat HTTP/1.1

Host: server.example.com

Upgrade: websocket

Connection: Upgrade

Origin: http://javascript.ru

Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==

Sec-WebSocket-Version: 13

Sec-WebSocket-Extensions: deflate-frame

Sec-WebSocket-Protocol: soap, wamp

Ответ:

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-Accept: hsBlbuDTkk24srzEOTBUlZAlC2g=

Sec-WebSocket-Extensions: deflate-frame

Sec-WebSocket-Protocol: soap

В ответе выше сервер указывает, что поддерживает расширение deflate-frame, а из запрошенных подпротоколов – только SOAP.

**[WSS](https://learn.javascript.ru/websockets" \l "wss)**

Соединение WebSocket можно открывать как WS:// или как WSS://. Протокол WSS представляет собой WebSocket над HTTPS.

**Кроме большей безопасности, у WSS есть важное преимущество перед обычным WS – большая вероятность соединения.**

Дело в том, что HTTPS шифрует трафик от клиента к серверу, а HTTP – нет.

Если между клиентом и сервером есть прокси, то в случае с HTTP все WebSocket-заголовки и данные передаются через него. Прокси имеет к ним доступ, ведь они никак не шифруются, и может расценить происходящее как нарушение протокола HTTP, обрезать заголовки или оборвать передачу.

А в случае с WSS весь трафик сразу кодируется и через прокси проходит уже в закодированном виде. Поэтому заголовки гарантированно пройдут, и общая вероятность соединения через WSS выше, чем через WS.

**[Формат данных](https://learn.javascript.ru/websockets" \l "format-dannyh)**

Полное описание протокола содержится в [RFC 6455](http://tools.ietf.org/html/rfc6455).

Здесь представлено частичное описание с комментариями самых важных его частей. Если вы хотите понять стандарт, то рекомендуется сначала прочитать это описание.

**[Описание фрейма](https://learn.javascript.ru/websockets" \l "opisanie-freyma)**

В протоколе WebSocket предусмотрены несколько видов пакетов («фреймов»).

Они делятся на два больших типа: фреймы с данными («data frames») и управляющие («control frames»), предназначенные для проверки связи (PING) и закрытия соединения.

Фрейм, согласно стандарту, выглядит так:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-------+-+-------------+-------------------------------+

|F|R|R|R| опкод |М| Длина тела | Расширенная длина тела |

|I|S|S|S|(4бита)|А| (7бит) | (1 байт) |

|N|V|V|V| |С| |(если длина тела==126 или 127) |

| |1|2|3| |К| | |

| | | | | |А| | |

+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +

| Продолжение расширенной длины тела, если длина тела = 127 |

+ - - - - - - - - - - - - - - - +-------------------------------+

| | Ключ маски, если МАСКА = 1 |

+-------------------------------+-------------------------------+

| Ключ маски (продолжение) | Данные фрейма ("тело") |

+-------------------------------- - - - - - - - - - - - - - - - +

: Данные продолжаются ... :

+ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - +

| Данные продолжаются ... |

+---------------------------------------------------------------+

С виду – не очень понятно, во всяком случае, для большинства людей.

**Позвольте пояснить: читать следует слева-направо, сверху-вниз, каждая горизонтальная полоска это 32 бита.**

То есть, вот первые 32 бита:

0 1 2 3

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1

+-+-+-+-+-------+-+-------------+-------------------------------+

|F|R|R|R| опкод |М| Длина тела | Расширенная длина тела |

|I|S|S|S|(4бита)|А| (7бит) | (1 байт) |

|N|V|V|V| |С| |(если длина тела==126 или 127) |

| |1|2|3| |К| | |

| | | | | |А| | |

+-+-+-+-+-------+-+-------------+ - - - - - - - - - - - - - - - +

Сначала идёт бит FIN (вертикальная надпись на рисунке), затем биты RSV1, RSV2, RSV3 (их смысл раскрыт ниже), затем «опкод», «МАСКА» и, наконец, «Длина тела», которая занимает 7 бит. Затем, если «Длина тела» равна 126 или 127, идёт «Расширенная длина тела», потом (на следующей строке, то есть после первых 32 бит) будет её продолжение, ключ маски, и потом данные.

А теперь – подробное описание частей фрейма, то есть как именно передаются сообщения:

**FIN: 1 бит**

Одно сообщение, если оно очень длинное (вызовом send можно передать хоть целый файл), может состоять из множества фреймов («быть фрагментированным»).

У всех фреймов, кроме последнего, этот фрагмент установлен в 0, у последнего – в 1.

Если сообщение состоит из одного-единственного фрейма, то FIN в нём равен 1.

**RSV1, RSV2, RSV3: 1 бит каждый**

В обычном WebSocket равны 0, предназначены для расширений протокола. Расширение может записать в эти биты свои значения.

**Опкод: 4 бита**

Задаёт тип фрейма, который позволяет интерпретировать находящиеся в нём данные. Возможные значения:

* 0x1 обозначает текстовый фрейм.
* 0x2 обозначает двоичный фрейм.
* 0x3-7 зарезервированы для будущих фреймов с данными.
* 0x8 обозначает закрытие соединения этим фреймом.
* 0x9 обозначает PING.
* 0xA обозначает PONG.
* 0xB-F зарезервированы для будущих управляющих фреймов.
* 0x0 обозначает фрейм-продолжение для фрагментированного сообщения. Он интерпретируется, исходя из ближайшего предыдущего ненулевого типа.

**Маска: 1 бит**

Если этот бит установлен, то данные фрейма маскированы. Более подробно маску и маскирование мы рассмотрим далее.

**Длина тела: 7 битов, 7+16 битов, или 7+64 битов**

Если значение поле «Длина тела» лежит в интервале 0-125, то оно обозначает длину тела (используется далее). Если 126, то следующие 2 байта интерпретируются как 16-битное беззнаковое целое число, содержащее длину тела. Если 127, то следующие 8 байт интерпретируются как 64-битное беззнаковое целое, содержащее длину.

Такая хитрая схема нужна, чтобы минимизировать накладные расходы. Для сообщений длиной 125 байт и меньше хранение длины потребует всего 7 битов, для бóльших (до 65536) – 7 битов + 2 байта, ну а для ещё бóльших – 7 битов и 8 байт. Этого хватит для хранения длины сообщения размером в гигабайт и более.

**Ключ маски: 4 байта.**

Если бит Маска установлен в 0, то этого поля нет. Если в 1 то эти байты содержат маску, которая налагается на тело (см. далее).

**Данные фрейма (тело)**

Состоит из «данных расширений» и «данных приложения», которые идут за ними. Данные расширений определяются конкретными расширениями протокола и по умолчанию отсутствуют. Длина тела должна быть равна указанной в заголовке.

**[Примеры](https://learn.javascript.ru/websockets" \l "primery)**

Некоторые примеры сообщений:

* Нефрагментированное текстовое сообщение Hello без маски:

0x81 0x05 0x48 0x65 0x6c 0x6c 0x6f (содержит "Hello")

В заголовке первый байт содержит FIN=1 и опкод=0x1 (получается 10000001 в двоичной системе, то есть 0x81 – в 16-ричной), далее идёт длина 0x5, далее текст.

* Фрагментированное текстовое сообщение Hello World из трёх частей, без маски, может выглядеть так:
* 0x01 0x05 0x48 0x65 0x6c 0x6c 0x6f (содержит "Hello")
* 0x00 0x01 0x20 (содержит " ")

0x80 0x05 0x57 0x6f 0x72 0x6c 0x64 (содержит "World")

* + У первого фрейма FIN=0 и текстовый опкод 0x1.
  + У второго FIN=0 и опкод 0x0. При фрагментации сообщения, у всех фреймов, кроме первого, опкод пустой (он один на всё сообщение).
  + У третьего, последнего фрейма FIN=1.

А теперь посмотрим на все те замечательные возможности, которые даёт этот формат фрейма.

**[Фрагментация](https://learn.javascript.ru/websockets" \l "fragmentatsiya)**

Позволяет отправлять сообщения в тех случаях, когда на момент начала посылки полный размер ещё неизвестен.

Например, идёт поиск в базе данных и что-то уже найдено, а что-то ещё может быть позже.

* У всех сообщений, кроме последнего, бит FIN=0.
* Опкод указывается только у первого, у остальных он должен быть равен 0x0.

**[PING / PONG](https://learn.javascript.ru/websockets" \l "ping-pong)**

В протокол встроена проверка связи при помощи управляющих фреймов типа PING и PONG.

Тот, кто хочет проверить соединение, отправляет фрейм PING с произвольным телом. Его получатель должен в разумное время ответить фреймом PONG с тем же телом.

Эта функциональность встроена в браузерную реализацию, так что браузер ответит на PING сервера, но управлять ей из JavaScript нельзя.

**Иначе говоря, сервер всегда знает, жив ли посетитель или у него проблема с сетью.**

**[Чистое закрытие](https://learn.javascript.ru/websockets" \l "chistoe-zakrytie)**

При закрытии соединения сторона, желающая это сделать (обе стороны в WebSocket равноправны) отправляет закрывающий фрейм (опкод 0x8), в теле которого указывает причину закрытия.

В браузерной реализации эта причина будет содержаться в свойстве reason события onclose.

**Наличие такого фрейма позволяет отличить «чистое закрытие» от обрыва связи.**

В браузерной реализации событие onclose при чистом закрытии имеет event.wasClean = true.

**[Коды закрытия](https://learn.javascript.ru/websockets" \l "kody-zakrytiya)**

Коды закрытия вебсокета event.code, чтобы не путать их с HTTP-кодами, состоят из 4 цифр:

**1000**

Нормальное закрытие.

**1001**

Удалённая сторона «исчезла». Например, процесс сервера убит или браузер перешёл на другую страницу.

**1002**

Удалённая сторона завершила соединение в связи с ошибкой протокола.

**1003**

Удалённая сторона завершила соединение в связи с тем, что она получила данные, которые не может принять. Например, сторона, которая понимает только текстовые данные, может закрыть соединение с таким кодом, если приняла бинарное сообщение.

**[Атака «отравленный кэш»](https://learn.javascript.ru/websockets" \l "ataka-otravlennyy-kesh)**

В ранних реализациях WebSocket существовала уязвимость, называемая «отравленный кэш» (cache poisoning).

**Она позволяла атаковать кэширующие прокси-сервера, в частности, корпоративные.**

Атака осуществлялась так:

1. Хакер заманивает доверчивого посетителя (далее Жертва) на свою страницу.
2. Страница открывает WebSocket-соединение на сайт хакера. Предполагается, что Жертва сидит через прокси. Собственно, на прокси и направлена эта атака.
3. Страница формирует специального вида WebSocket-запрос, который (и здесь самое главное!) ряд прокси серверов не понимают.

Они пропускают начальный запрос через себя (который содержит Connection: upgrade) и думают, что далее идёт уже следующий HTTP-запрос.

…Но на самом деле там данные, идущие через вебсокет! И обе стороны вебсокета (страница и сервер) контролируются Хакером. Так что хакер может передать в них нечто похожее на GET-запрос к известному ресурсу, например http://code.jquery.com/jquery.js, а сервер ответит «якобы кодом jQuery» с кэширующими заголовками.

Прокси послушно проглотит этот ответ и закэширует «якобы jQuery».

1. В результате при загрузке последующих страниц любой пользователь, использующий тот же прокси, что и Жертва, получит вместо http://code.jquery.com/jquery.js хакерский код.

Поэтому эта атака и называется «отравленный кэш».

**Такая атака возможна не для любых прокси, но при анализе уязвимости было показано, что она не теоретическая, и уязвимые прокси действительно есть.**

Поэтому придумали способ защиты – «маску».

**[Маска для защиты от атаки](https://learn.javascript.ru/websockets" \l "maska-dlya-zaschity-ot-ataki)**

Для того, чтобы защититься от атаки, и придумана маска.

*Ключ маски* – это случайное 32-битное значение, которое варьируется от пакета к пакету. Тело сообщения проходит через XOR ^ с маской, а получатель восстанавливает его повторным XOR с ней (можно легко доказать, что (x ^ a) ^ a == x).

Маска служит двум целям:

1. Она генерируется браузером. Поэтому теперь хакер не сможет управлять реальным содержанием тела сообщения. После накладывания маски оно превратится в бинарную мешанину.
2. Получившийся пакет данных уже точно не может быть воспринят промежуточным прокси как HTTP-запрос.

**Наложение маски требует дополнительных ресурсов, поэтому протокол WebSocket не требует её.**

Если по этому протоколу связываются два клиента (не обязательно браузеры), доверяющие друг другу и посредникам, то можно поставить бит Маска в 0, и тогда ключ маски не указывается.

**[Пример](https://learn.javascript.ru/websockets" \l "primer)**

Рассмотрим прототип чата на WebSocket и Node.JS.

HTML: посетитель отсылает сообщения из формы и принимает в div

<!-- форма для отправки сообщений -->

<form name="publish">

<input type="text" name="message">

<input type="submit" value="Отправить">

</form>

<!-- здесь будут появляться входящие сообщения -->

<div id="subscribe"></div>

Код на клиенте:

// создать подключение

var socket = new WebSocket("ws://localhost:8081");

// отправить сообщение из формы publish

document.forms.publish.onsubmit = function() {

var outgoingMessage = this.message.value;

socket.send(outgoingMessage);

return false;

};

// обработчик входящих сообщений

socket.onmessage = function(event) {

var incomingMessage = event.data;

showMessage(incomingMessage);

};

// показать сообщение в div#subscribe

function showMessage(message) {

var messageElem = document.createElement('div');

messageElem.appendChild(document.createTextNode(message));

document.getElementById('subscribe').appendChild(messageElem);

}

Серверный код можно писать на любой платформе. В нашем случае это будет Node.JS, с использованием модуля [ws](https://github.com/websockets/ws):

var WebSocketServer = new require('ws');

// подключённые клиенты

var clients = {};

// WebSocket-сервер на порту 8081

var webSocketServer = new WebSocketServer.Server({

port: 8081

});

webSocketServer.on('connection', function(ws) {

var id = Math.random();

clients[id] = ws;

console.log("новое соединение " + id);

ws.on('message', function(message) {

console.log('получено сообщение ' + message);

for (var key in clients) {

clients[key].send(message);

}

});

ws.on('close', function() {

console.log('соединение закрыто ' + id);

delete clients[id];

});

});

Рабочий пример можно скачать: [websocket.zip](https://learn.javascript.ru/article/websockets/websocket.zip). Понадобится поставить два модуля: npm install node-static && npm install ws.

**[Итого](https://learn.javascript.ru/websockets" \l "itogo)**

WebSocket – современное средство коммуникации. Кросс-доменное, универсальное, безопасное.

На текущий момент он работает в браузерах IE10+, FF11+, Chrome 16+, Safari 6+, Opera 12.5+. В более старых версиях FF, Chrome, Safari, Opera есть поддержка черновых редакций протокола.

Там, где вебсокеты не работают – обычно используют другие транспорты, например IFRAME. Вы найдёте их в других статьях этого раздела.

Есть и готовые библиотеки, реализующие функциональность COMET с использованием сразу нескольких транспортов, из которых вебсокет имеет приоритет. Как правило, библиотеки состоят из двух частей: клиентской и серверной.

Например, для Node.JS одной из самых известных библиотек является [Socket.IO](http://socket.io/).

К недостаткам библиотек следует отнести то, что некоторые продвинутые возможности WebSocket, такие как двухсторонний обмен бинарными данными, в них недоступны. С другой – в большинстве случаев стандартного текстового обмена вполне достаточно.