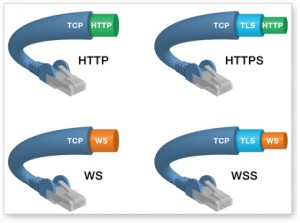
**WEBSOCKET**

**WebSocket** es una tecnología que proporciona un canal de comunicación bidireccional y [full-duplex](https://es.wikipedia.org/wiki/Duplex_(telecomunicaciones)#Full-duplex) sobre un único [socket](https://es.wikipedia.org/wiki/Socket_de_Internet) [TCP](https://es.wikipedia.org/wiki/Transmission_Control_Protocol). Está diseñada para ser implementada en [navegadores](https://es.wikipedia.org/wiki/Navegador_web) y [servidores web](https://es.wikipedia.org/wiki/Servidor_web), pero puede utilizarse por cualquier aplicación cliente/servidor. La API de WebSocket está siendo normalizada por el [W3C](https://es.wikipedia.org/wiki/World_Wide_Web_Consortium), mientras que el protocolo WebSocket ya fue normalizado por la [IETF](https://es.wikipedia.org/wiki/Internet_Engineering_Task_Force) como el [RFC 6455](https://tools.ietf.org/html/rfc6455). Debido a que las conexiones TCP comunes sobre puertos diferentes al 80 son habitualmente bloqueadas por los administradores de redes, el uso de esta tecnología proporcionaría una solución a este tipo de limitaciones proveyendo una funcionalidad similar a la apertura de varias conexiones en distintos puertos, pero [multiplexando](https://es.wikipedia.org/wiki/Multiplexaci%C3%B3n" \o "Multiplexación) diferentes servicios WebSocket sobre un único puerto TCP (a costa de una pequeña sobrecarga del protocolo).

Esto es genial, ya que las posibilidades de magia en un sito web aumentan considerablemente, pero hay que tener en cuenta algunas cosas, ya que son **Web** Sockets

[](http://fernetjs.com/wp-content/uploads/2012/11/cables21.jpg)

Se inicia con un handshake HTTP, por lo que si no hay HTTP no hay WebSockets

Tanto servidor como cliente tienen que soportarlo (para eso tenemos a HTML 5 en cliente y SocketIO en el servicor  )

Sólo podemos transmitir texto/ JSON (a diferencia del TCP convencional en el que podemos transmitir streams de bytes)

La conexión TCP va por el puerto 80 (lo cual está bueno en algunos casos donde tenemos firewalls)

Así como tenemos HTTPS (seguros), podemos tener WSS jeje, pero no todos los navegadores que soportan WS, también soportan WSS

**Negociación del Protocolo Websocket**

Método petición – respuesta: Para establecer una conexión WebSocket, el cliente manda una petición de negociación WebSocket, y el servidor manda una respuesta de negociación WebSocket, como se puede ver en el siguiente ejemplo:

Petición del navegador al servidor:

GET /demo HTTP/1.1

Host: example.com

Connection: Upgrade

Sec-WebSocket-Key2: 12998 5 Y3 1 .P00

Sec-WebSocket-Protocol: sample

Upgrade: WebSocket

Sec-WebSocket-Key1: 4 @1 46546xW%0l 1 5

Origin: http://example.com

^n:ds[4U

Respuesta del servidor:

HTTP/1.1 101 WebSocket Protocol Handshake

Upgrade: WebSocket

Connection: Upgrade

Sec-WebSocket-Origin: http://example.com

Sec-WebSocket-Location: ws://example.com/demo

Sec-WebSocket-Protocol: sample

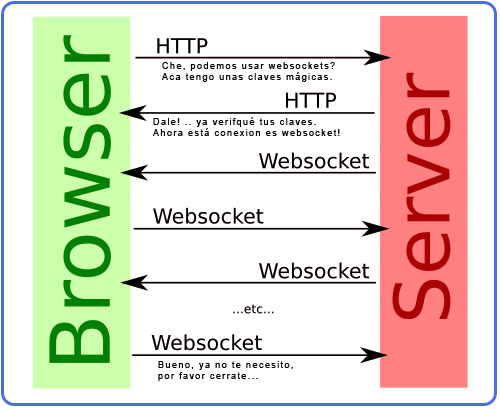
8jKS'y:G\*Co,Wxa-

Los 8 bytes con valores numéricos que acompañan a los campos Sec-WebSocket-Key1 y Sec-WebSocket-Key2 son tokens aleatorios que el servidor utilizará para construir un token de 16 bytes al final de la negociación para confirmar que ha leído correctamente la petición de negociación del cliente.

La negociación o handshake se construye concatenando los números que acompañan al primer campo, y dividiéndolo por el número de espacios en blanco en el propio campo. Esto mismo se repite para el segundo campo. Los dos números resultantes se concatenan entre sí y con los 8 bytes que van después de los campos. El resultado final es una suma [MD5](https://es.wikipedia.org/wiki/MD5) de la cadena concatenada.[3](https://es.wikipedia.org/wiki/WebSocket#cite_note-3)

Esta negociación puede parecerse a la negociación HTTP, pero no es así. Permite al servidor interpretar parte de la petición de negociación como HTTP y entonces cambiar a WebSocket.

Una vez establecida, las tramas WebSocket de datos pueden empezar a enviarse en ambos sentidos entre el cliente y el servidor en modo full-duplex. Las tramas de texto pueden ser enviadas en modo [full-duplex](https://es.wikipedia.org/wiki/Duplex_(telecomunicaciones))también, en ambas direcciones al mismo tiempo. La información se segmenta en tramas de únicamente 2 bytes. Cada trama empieza con un byte 0x00, termina con un byte 0xFF, y contiene datos UTF-8 entre ellos. Tramas de datos binarios no están soportadas todavía en el API. Las tramas WebSocket de texto utilizan un terminador, mientras que las tramas binarias utilizan un prefijo de longitud.



**PASO A TRAVÉS DE SERVIDORES PROXY**

La implementación de cliente del protocolo WebSocket intenta detectar si el [agente de usuario](https://es.wikipedia.org/wiki/Agente_de_usuario) está configurado para utilizar un proxy a la hora de conectar a un host y puerto remoto, y si es así, utiliza el método[HTTP](https://es.wikipedia.org/wiki/HTTP) CONNECT para establecer un túnel persistente.

Aunque el protocolo WebSocket es indiferente a la conexión sobre servidores proxy o cortafuegos, implementa una negociación compatible con HTTP para que los servidores HTTP puedan compartir sus puertos HTTP y HTTPS por defecto (80 y 443) con una pasarela o servidor WebSocket. El protocolo WebSocket define un prefijo ws:// y wss:// para indicar una conexión *WebSocket* y *Websocket Secure*, respectivamente. Ambos esquemas utilizan un [mecanismo *HTTP upgrade*](https://es.wikipedia.org/wiki/Cabecera_HTTP/1.1_Upgrade) para actualizar al protocolo WebSocket. Algunos servidores proxy no interfieren en la conexión y funcionan perfectamente con WebSocket; otros afectan al correcto funcionamiento de WebSocket, provocando que la conexión falle. En algunos casos puede que se requiera configuración adicional en el servidor proxy, y algunos servidores proxy puede que necesiten actualizarse para soportar WebSocket.

Si el tráfico sin cifrar de WebSocket pasa por un proxy explícito o transparente en su camino al servidor WebSocket, entonces, independientemente de que el proxy se comporte como debiera, esta conexión, a día de hoy, muy probablemente fallará (a medida que WebSocket se extienda, los servidores proxy lo tendrán en cuenta). De ahí que las conexiones sin cifrar de WebSocket se deberían utilizar sólo en las topologías más sencillas.

Si se utiliza una conexión WebSocket cifrada, se necesitará utilizar una capa [TLS](https://es.wikipedia.org/wiki/Transport_Layer_Security) en la conexión Websocket Secure para asegurar que un comando HTTP CONNECT se manda cuando el navegador está configurado para utilizar un servidor proxy explícitamente. Esto establece un túnel, que proporciona una comunicación TCP de bajo nivel y punto a punto a través del proxy HTTP, entre el cliente WebSocket Secure y el servidor Websocket. En el caso de servidores proxy trasparentes, el navegador no será consciente del uso de un servidor proxy, así que no mandará una petición HTTP CONNECT. En cualquier caso, como el tráfico está cifrado, los servidores proxy trasparentes intermedios podrían simplemente permitir el tráfico cifrado a través de ellos, de manera que la conexión WebSocket funcionará sin problemas si se utiliza WebSocket Secure. Utilizar cifrado no es gratis en lo que a consumo de recursos se refiere, pero nos asegura la tasa de éxito más alta.

Desafortunadamente, una actualización reciente al borrador (versión 76) rompió completamente la compatibilidad con proxies inversos y pasarelas, ya que los 8 bytes de datos que el cliente debe enviar después de las cabeceras no se anuncian en una cabecera Content-Length, por lo que los intermediarios no reenvían los datos hasta que se completa la negociación. Y como la negociación necesita de esos 8 bytes para completarse, ésta nunca se completa y entra en un punto muerto. En el estado actual de las cosas, no es recomendable modificar estos componentes intermediarios para que soporten este comportamiento HTTP no estándar, porque hacerlo llevería a estos componentes a ser vulnerables a ataques HTTP smuggling, ya que un atacante simplemente tendría que hacer un intento de actualización al protocolo WebSocket en una petición para ser capaz de mandar más datos de los que el servidor HTTP de destino pudiera parsear, saltándose así algunos filtros de seguridad obligatorios. No se sabe si esta problemática se solucionará en un nuevo borrador o no.

**Esquema de URL**

La especificación del protocolo WebSocket define dos nuevos esquemas de [URI](https://es.wikipedia.org/wiki/URI), ws: y wss: para conexiones no cifradas y cifradas. Además del nombre del esquema, el resto de componentes del URI se definen con la sintaxis genérica de URI.

**Navegadores que soportan WebSocket**

* Chrome 4
* Safari 5 (incluye iOS 4.2)
* Mozilla Firefox 8

Actualmente, estos dos navegadores soportan el draft-ietf-hybi-thewebsocketprotocol-00.

**FUNCIONAMIENTO DE LOS WEBSOCKETS**

Históricamente las aplicaciones web se han basado en un modelo cliente-servidor, donde el cliente siempre es el primero en iniciar la comunicación con el servidor. Con la llegada de AJAX, las peticiones podían hacerse de forma asíncrona y el cliente no necesariamente tenia que esperar la respuesta del servidor para poder continuar trabajando con la aplicación web, sin embargo la comunicación tenia que ser iniciada por el cliente y si por algún motivo, el servidor tenia nueva información que debía transferirse al cliente, era el cliente el que tenia que realizar las peticiones contra el servidor para recuperar la nueva información. Evidentemente, el hecho de que el servidor pueda comunicarse con el cliente para informar sobre cualquier “novedad”, es una característica que siempre ha resultado muy útil y pensando en ello, se han diseñado varias alternativas que funcionan sobre el protocolo HTTP, como por ejemplo Comet, Push o con el uso de conexiones HTTP persistentes. Con estas alternativas, el servidor puede enviar datos al cliente sin que el cliente tenga que iniciar una interacción, son técnicas que se utilizan actualmente en muchos servicios en Internet y en su día, GMail también lo hacia, sin embargo, mantener una conexión HTTP durante un largo periodo de tiempo es costoso y no es una muy buena idea en aplicaciones que deben tener una baja latencia. Es aquí donde HTML5 viene “al rescate” con los ***websockets.***

Los websockets son una alternativa muy óptima al problema descrito en líneas anteriores, ya que permite crear conexiones TCP utilizando sockets entre el cliente (navegador) y el servidor. Como con cualquier socket TCP, ambas partes pueden transferir datos en cualquier momento y de esta forma el servidor podrá enviar información al cliente cuando sea necesario.

**USO DE WEBSOCKETS EN HTML5**

Para abrir una conexión entre cliente y servidor utilizando websockets, ahora se utilizan los esquemas “ws://” en lugar de “http://” y “wss://” en lugar de https://. Además, el modelo de programación que debe utilizarse es basado en eventos, dichos eventos alertarán al cliente cuando se realice una conexión con el servidor, cuando ocurra un error, o cuando se reciba un mensaje por parte del servidor. Para crear un websocket, se debe ingresar la dirección del servidor web con el esquema “ws://” o “wss://” y especificar alguno de los subprotocolos soportados en la especificación de websockets.

Después de abrir una conexión con el servidor, el cliente puede enviar mensajes con la función “send” que se encuentra definida en la referencia a la conexión creada anteriormente. Los datos que se pueden enviar pueden ser cadenas de texto o datos binarios representados con los objetos Blob o ArrayBuffer. Un ejemplo del uso de websockets desde el lado del cliente podría ser el siguiente.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | var connection = new WebSocket('<ws://html5rocks.websocket.org/echo>', ['soap', 'xmpp']);  connection.onopen = function () {    connection.send('Hello Server!'); // Send the message 'Ping' to the server  };    connection.onerror = function (error) {    console.log('WebSocket Error ' + error);  };    connection.onmessage = function (e) {    console.log('From Server: ' + e.data);  };    connection.send('Hello cutty server!');    var img = canvas\_context.getImageData(0, 0, 200, 200);  var binary = new Uint8Array(img.data.length);  for (var i = 0; i < img.data.length; i++) {    binary[i] = img.data[i];  }  connection.send(binary.buffer);    var file = document.querySelector('input[type="file"]').files[0];  connection.send(file); |

En el ejemplo anterior se ha creado una conexión con el servidor utilizando la clase WebSocket, la cual se encuentra disponible en la mayoría de navegadores modernos. La instancia creada de dicho objeto es utilizada para declarar una serie de funciones de callback que serán invocadas cuando se produzcan diferentes tipos de eventos posibles.

Por otro lado, aunque se trata de una tecnología muy interesante y que se comienza a utilizar con mayor frecuencia en aplicaciones web, no obstante la principal dificultad a la hora de utilizar websockets, es la imposibilidad de establecer conexiones utilizando servidores proxy por medio y dado que en la mayoría de entornos empresariales, el uso de servidores proxy es bastante común, nos encontramos con una limitación que hay que tener en cuenta cuando se habla de utilizar WebSockets en aplicaciones web. La razón de esto, es que los WebSockets utilizan el valor “upgrade” para la cabecera “Connection” o directamente la cabecera “Upgrade” con el el valor “WebSocket” y dicho valor indica que la conexión que inicialmente se ha establecido utilizando HTTP, debe “actualizarse” para utilizar sockets del tipo TCP. Este tipo de cambios en las conexiones no son soportados por los servidores proxy del tipo HTTP, ya que están diseñados para trabajar con paquetes de datos que utilizan el protocolo HTTP, en este caso, la conexión es automáticamente cortada por el servidor. Para “mitigar” este problema existen varias soluciones, como por ejemplo el uso del proyecto Apache Camel ([http://camel.apache.org](http://camel.apache.org/)) o mi favorita, el uso de “mod\_proxy\_wstunnel” (ver: <http://httpd.apache.org/docs/2.4/mod/mod_proxy_wstunnel.html>) sobre esta extensión de Apache os hablaré en una próxima entrada.

Hasta este punto se ha hablado del lado del cliente, sin embargo en el lado del servidor se requiere un cambio de enfoque muy importante, ya que ahora hablamos de que el servidor debe soportar múltiples conexiones abiertas al mismo tiempo, con el consecuente consumo de recursos que aquello implica, por éste y otros motivos, la mayoría de servidores web modernos soportan modelos “Non-bloquing IO” o lo que es lo mismo, varios grupos de hilos que se ejecutan de forma concurrente y asíncrona para el procesamiento de respuestas. Se trata de un modelo de arquitectura de software ampliamente aceptado y utilizado hoy en día, tanto en servidores como en sistemas operativos. Alguno de los servidores web que soportan estas características (sin ser una lista exhaustiva y basándome únicamente en los que he probado) se mencionan a continuación:

Apache Web Server 2.2 / 2.4 (<http://httpd.apache.org/>)

Jetty (<http://www.eclipse.org/jetty/>)

Apache Tomcat (<http://tomcat.apache.org/>)

JBoss (<http://www.jboss.org/>)

Socket.IO (<http://socket.io/>)

Pywebsocket para Apache web Server. (<http://code.google.com/p/pywebsocket/>)

Tornado (<https://github.com/tornadoweb/tornado>)

**VULNERABILIDADES COMUNES EN EL USO DE WEBSOCKETS**

Ahora que está claro qué son y para qué sirven los websockets, es el momento de hablar sobre las vulnerabilidades que se pueden producir cuando se implementan de forma indebida o con pocos controles sobre los datos intercambiados entre cliente y servidor. Estamos hablando de una tecnología joven, que si bien tiene un potencial enorme a la hora de crear aplicaciones web robustas, esto no es gratis y tiene sus riesgos. Algunos de dichos riesgos se explican a continuación.

**Transporte no securizado y vulnerable a ataques MITM.**

Como se ha mencionado anteriormente, los websockets funcionan utilizando el protocolo TCP, lo que habilita muchas posibilidades a la hora de realizar conexiones contra multitud de servicios, sin embargo si el canal de comunicación no se encuentra debidamente securizado, un ataque del tipo MITM puede comprometer todos los mensajes enviados entre cliente y servidor. Siempre es una buena practica utilizar el contexto “wss://” para establecer conexiones cifradas con TLS.

**Los websockets no soportan autenticación ni autorización.**

El protocolo de websockets no soporta los mecanismos tradicionales de autenticación y autorización. Es un asunto relacionado con la implementación propiamente dicha del protocolo y en el caso de que el cliente y el servidor intercambien información sensible, además de securizar el canal de comunicación utilizando TLS, también es recomendable utilizar mecanismos de autenticación basados en tokens/tickets. Dichos mecanismos son bastante populares en implementaciones REST, donde algunos servicios solamente pueden ser consumidos si el cliente cuenta con un token de autenticación valido y dicho token se vincula con el servidor por medio de una cuenta de usuario. Dicho patrón de autenticación se ha vuelto cada vez más popular y en aplicaciones que utilizan websockets que requieren mecanismos de control sobre los usuarios autenticados, es una excelente forma de mantener un control de acceso a recursos sensibles.

**Validación en los datos de entrada**

Aunque los websockets utilicen TCP para realizar las conexiones entre clientes y servidores, aun nos encontramos en el contexto de una aplicación web y es importante validar los datos de entrada de los usuarios. En el caso de no validar los campos de entrada adecuadamente, se pueden producir vulnerabilidades del tipo XSS aunque para la comunicación se ha un protocolo distinto a HTTP.

**Vulnerabilidades Cross Site Request Foregy**

Tal como se comentaba en un articulo anterior, las políticas de “same origin policy” que aplican cuando se trata de compartir recursos entre distintos dominios, ahora ya no son tan estrictas cuando se utiliza la cabecera HTTP “Origin”. Tal como se mencionaba en dicho articulo, se trata de una característica que está muy bien cuando se trata de compartir recursos con dominios “fiables” y cuando hablamos de relaciones de confianza en el mundo de la seguridad de la información, siempre pueden haber situaciones que le permiten a un atacante abusar de dichas condiciones.

Los websockets ***no están limitados a las restricciones de SOP***, esto significa que un atacante puede iniciar una petición websocket desde su sitio web malicioso contra un endpoint (ws:// o wss://) de una aplicación web aunque no se encuentre en el mismo dominio. Esto tiene unas implicaciones tremendas, ya que el handshake que ejecutará el cliente para iniciar una petición WebSocket es una petición HTTP regular y los navegadores enviarán las cookies y cabeceras HTTP de autenticación aunque se trate de un dominio cruzado, si y sólo si, el servidor web no valida adecuadamente la cabecera “Origin”.  
Para que el lector se haga una idea del problema, supongamos que la víctima ha iniciado sesión en un sitio web que tiene uno o varios endpoints “ws” o “wss” y en una pestaña nueva del navegador, ingresa en una página web maliciosa controlada por el atacante. Dicha página podría realizar una petición WebSocket contra el sitio web en el que el usuario se encuentra identificado y si dicho sitio web, no valida adecuadamente el valor de la cabecera “Origin”, la respuesta a la petición del atacante podrá contener, entre otras, las cookies y cabeceras de autenticación utilizadas por el usuario, sin importar que se trate de un dominio cruzado. Esta situación puede dar lugar a una vulnerabilidad del tipo CSRF.

**EJEMPLO DE FUNCIONAMIENTO:**

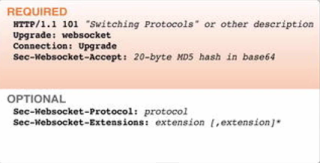
Los WebSockets comunican usando la capa **TCP** la conexión se establece sobre Http y es básicamente un mecanismo **Handshake** entre el cliente y el servidor, pero después del este la conexión es actualizada a TCP. Este funcionaría de la siguiente forma:

1. Se realiza la llamada http, que es iniciada desde el lado del cliente. La cabecera de esta llamada http sería algo similar a esto:

[](http://4.bp.blogspot.com/-vP-y1z4ZE2g/VVtTNm3DAcI/AAAAAAAAAJo/JEHKLTRXkqY/s1600/httpcliente.PNG)

* + Host: Es el nombre del servidor al que estamos llamando.
  + Upgrade: indica que es una llamada de actualización. En este caso *websocket*.
  + Connection: define que es una llamada de actualización.
  + Sec-Websocket-Key: es una clave generada aleatoriamente, que posteriormente es usada para autentificar la respuesta. Es la clave del Handshake.
  + Sec-Websocket-Origin: es otro parámetro, pero en este caso opcional, que muestra donde sea generado la petición. En el lado del servidor es usado para comprobar la autentificación de la petición.

1. Una vez que el servidor comprueba la autentificación va a enviar una respuesta que se ve, así:

[](http://4.bp.blogspot.com/-d2E_ZwIsp44/VVtTMou-wUI/AAAAAAAAAJs/uoGVjqdeAec/s1600/httpserver.PNG)

* + Sec-Websocker-Accept: hay una clave que es comprobada y autentificada con la *key* enviada para ver que las respuestas están viendo al cliente de origen correcto.

1. Una vez que es abierta la conexión el cliente y el servidor pueden enviar los datos de uno a otro, estos datos son enviados en forma de pequeños paquetes usando el protocolo TCP, así que estas llamadas no son http, por tanto no son visibles directamente dentro de la herramienta de desarrolladores en el navegador por ejemplo.

### **EJEMPLOS:**

En este ejemplo de crea un nuevo WebSocket, conectandose al servidor ws://www.example.com/socketserver. El nombre del protocolo "protocolOne"  es el utilizado para la consulta del socket, aunque puede ser omitido.

var exampleSocket = new WebSocket("ws://www.example.com/socketserver", "protocolOne");

Como respuesta, exampleSocket.readyState es CONNECTING. El readyState será  OPEN una ves que la conexion este lista para transferir información.

Si se quiere establecer una conexión que soporte varios protocolos, se puede establecer un array de protolos:

var exampleSocket = new WebSocket("ws://www.example.com/socketserver", ["protocolOne", "protocolTwo"]);

Una vez que la conexión este establecida (readyState estará OPEN), exampleSocket.protocol te dirá qué protocolo ha seleccionado el servidor.

En los ejemplos anteriores ws sustituye http, y de igual manera wss sustituye a https. Al crear un WebSocket se hace uso del mecanismo Upgrade de HTTP, por lo que la petición de actualización del protocolo está implícita cuando accedemos al servidor HTTP con ws://www.example.com o wss://www.example.com.

## **Enviando Información al servidor:**

Una ves la conexión esta abierta, se puede iniciar a enviar datos al servidor. Para hacer esto, simplemente se llama al metodo [send()](https://developer.mozilla.org/en/WebSockets/WebSockets_reference/WebSocket#send()) del objeto WebSocket, cada vez que se desea enviar un mensaje:

exampleSocket.send("Here's some text that the server is urgently awaiting!");

Puedes enviar información como un string, [Blob](https://developer.mozilla.org/es/docs/Web/API/Blob), o en un  [ArrayBuffer](https://developer.mozilla.org/en/JavaScript_typed_arrays/ArrayBuffer).

**Nota:** Antes de la version 11, Firefox sólo soportaba el envío de datos como una cadena.

Como la conexión es asincronica y es propensa a fallar, no hay garantia de poder llamar al metodo send() inmediatamente despúes de haber creado el objeto WebSocket de manera exitosa. Para enviar información se debe estar seguro de que almenos una conexión ya esta abierta, usando el manejador onopen:

exampleSocket.onopen = function (event) {

exampleSocket.send("Here's some text that the server is urgently awaiting!");

};

## **Usando JSON para transferir Objetos**

Una forma en que podes enviar informacion compleja el servidor es utilizar [JSON](https://developer.mozilla.org/en/JSON). Por ejemplo, un programa para chatear puede interactuar con el servidor usando un protocolo que implementa el uso de paquetes de JSON-encapsulated data:

// Envia texto a todos los usuarios através del servidor

function sendText() {

// Se construye un Objeto msg que contiene la información que el servidor necesita procesar de ese cliente.

var msg = {

type: "message",

text: document.getElementById("text").value,

id: clientID,

date: Date.now()

};

// Send the msg object as a JSON-formatted string.

exampleSocket.send(JSON.stringify(msg));

// Blank the text input element, ready to receive the next line of text from the user.

document.getElementById("text").value = "";

}

## **Recibiendo mensajes del servidor**

WebSockets API es un manejador de eventos; cuando el mensaje en recibido, un "message" el evento es pasado el manejador onmessage. Para escuchar la entrada de información, se puede hacer algo como lo siguiente:

exampleSocket.onmessage = function (event) {

console.log(event.data);

## **Recibiendo e interpretando objetos JSON**

Vamos a suponer una aplicación de un chat, donde el cliente usa JSON para transmitir objetos de información. Hay varios tipos de paquetes que el cliente recibirá, como:

* Inicio de sesión handshake
* Mensajes de texto
* Actualización de Lista de usuarios

El código que interpretará los mensajes que entran será así:

exampleSocket.onmessage = function(event) {

var f = document.getElementById("chatbox").contentDocument;

var text = "";

var msg = JSON.parse(event.data);

var time = new Date(msg.date);

var timeStr = time.toLocaleTimeString();

switch(msg.type) {

case "id":

clientID = msg.id;

setUsername();

break;

case "username":

text = "<b>User <em>" + msg.name + "</em> signed in at " + timeStr + "</b><br>";

break;

case "message":

text = "(" + timeStr + ") <b>" + msg.name + "</b>: " + msg.text + "<br>";

break;

case "rejectusername":

text = "<b>Your username has been set to <em>" + msg.name + "</em> because the name you chose is in use.</b><br>"

break;

case "userlist":

var ul = "";

for (i=0; i < msg.users.length; i++) {

ul += msg.users[i] + "<br>";

}

document.getElementById("userlistbox").innerHTML = ul;

break;

}

if (text.length) {

f.write(text);

document.getElementById("chatbox").contentWindow.scrollByPages(1);

}

};

Se usa [JSON.parse()](https://developer.mozilla.org/en/JavaScript/Reference/Global_Objects/JSON/parse" \o "en/JavaScript/Reference/Global Objects/JSON/parse) para convertir el objeto JSON de vuelta al original, luego se examina y se realiza la acción pertinente.

## **Formato de texto de los datos**

El texto recibido a través de la conexión WebSocket está en formato UTF-8.

Antes de Gecko 9.0 (Firefox 9.0 / Thunderbird 9.0 / SeaMonkey 2.6), algunos no-caracteres que siguen siendo texto UTF-8 válido podrían causar el cierre de la conexión. Ahora Gecko permite esos valores.

## **Cerrando la conexión**

Cuando se a terminado de usar la conexipon WebSocket, se llama el metodo [close()](https://developer.mozilla.org/en/WebSockets/WebSockets_reference/WebSocket" \l "close()" \o "en/WebSockets/WebSockets reference/WebSocket#close()) del objeto WebSocket:

exampleSocket.close();

It may be helpful to examine the socket's bufferedAmount attribute before attempting to close the connection to determine if any data has yet to be transmitted on the network.

## **Consideraciones de Seguridad**

Los WebSockets no deben ser usados en ambientes de contenido mezclado; eso es, no debes abrir una conexión no segura de WebSocket desde una página cargada usando HTTPS o viceversa. De hecho, algunos navegadores prohiben esto explícitamente, incluyendo Firefox 8 y posteriores.

## **CREAR UN WEBSOCKET**

Un websocket es una tecnología [HTML5](http://www.manualweb.net/tutorial-html5/) que nos permite abrir sockets para una comunicación bidirecional, full-duplex, con baja latencia y que se basa en el protocolo TCP. Los Websockets están siendo estandarizado por la [W3C](http://www.w3.org/) mediante el [Websocket API](http://dev.w3.org/html5/websockets/).

La versión del protocolo de Websockets más soportada en la actualidad es la versión 76.

Lo primero que tenemos que hacer es definir el Websocket. Para ello utilizamos la clase WebSocket, la cual recibe como parámetro la URI del Websocket.

1. **var** mysocket = **new** WebSocket("ws://echo.websocket.org");

Lo primero que vemos es que la URI del Websocket se caracteriza por el protocolo ws:// o wss:// si fuese una conexión segura.

En nuestro caso utilizamos la URI

ws://echo.websocket.org

que es un Websocket que responde con un eco de cada información enviada al servidor.

Una vez que hemos creado el Websocket lo siguiente que tenemos que hacer es definir las funciones de callback que controlarán el intercambio de información con el Websocket.

Las funciones de callback que hay que implementar son:

* **onopen** - cuándo se abre el Websocket.
* **onmessage** - para cuando recibimos un mensaje por el Websocket.
* **onclose** - cuándo se cierra el Websocket.
* **onerror** - si se produce un error en el Websocket.

Todas las funciones de callback del Websocket tienen la siguiente estructura:

1. mysocket.onxxx = **function** (evt){ ... };

Así, codificaremos las siguientes funciones de callback:

1. mysocket.onopen = **function** (evt){
2. escribir("Websocket abierto");
3. };
5. mysocket.onmessage = **function** (evt){
6. escribir("RECIBIDO: " + evt.data);
7. };
9. mysocket.onclose = **function** (evt){
10. escribir("Websocket cerrado");
11. };
13. mysocket.onerror = **function** (evt) {
14. escribir("ERROR: " + evt.data);
15. }

Vemos que en todos los casos utilizamos la función escribir, la cual vuelca el contenido en una capa. De la siguiente forma:

1. **function** escribir(texto){
2. valor = document.getElementById("caja").value;
3. document.getElementById("caja").value = valor + texto + "**\n**";
4. }

Una vez que tenemos el Websocket abierto y con sus funciones de callback definidas empezamos a enviar mensajes. Para enviar mensajes al Websocket utilizamos el método .send()

1. **function** enviar(texto) {
2. mysocket.send(texto);
3. escribir("ENVIADO: " + texto);
4. }

Para finalizar nuestro cambio de mensajes con el Websocket utilizaremos el método .close() para cerrar el Websocket.

1. **function** desconectar(){
2. mysocket.close();
3. }

Solo nos quedará tener un pequeño layout que nos permita ejecutar las funciones enviar() y desconectar(). Así como tener una capa en la cual se vuelque el contenido.

1. [**<textarea**](http://december.com/html/4/element/textarea.html) id="caja" cols="100" rows="20"**></textarea>**[**<br**](http://december.com/html/4/element/br.html)/**>**
2. [**<input**](http://december.com/html/4/element/input.html) id="mensaje" type="text" size="105"**></input>**
3. [**<button**](http://december.com/html/4/element/button.html) onClick="enviar(document.getElementById('mensaje').value);"**>**Enviar**</button>**
4. [**<button**](http://december.com/html/4/element/button.html) onClick="desconectar()"**>**Desconectar**</button>**

Por último que nos queda por saber es el soporte de los navegadores para los Websockets. En la actualidad los navegadores que soportan los Websockets son:

* Firefox 4
* Google Chrome 4
* Opera 11
* Safari 5

## **EJEMPLO:**

Cliente WebSocket

La aplicación WebSocket del lado del cliente es muy fácil, todo el código consta de unos métodos y eventos como lo muestra el código siguiente

1. websocket = new WebSocket("ws://localhost:5800/server.php");
2. websocket.onopen = function(evt) { /\* escribir algo \*/ };
3. websocket.onclose = function(evt) { /\* escribir algo \*/ };
4. websocket.onmessage = function(evt) { /\* escribir algo\*/ };
5. websocket.onerror = function(evt) { /\* escribir algo \*/ };
6. websocket.send(message);
7. websocket.close();

Para abrir la conexión de socket, simplemente llamamos new WebSocket (ws: / / URL del servidor), ya que WebSocket utiliza un protocolo diferente para las conexiones, utilizamos ws :/ / en lugar de http://, seguido por el anfitrión, número de puerto y servidor:

chat2

Inmediatamente después de abrir la conexión, tenemos que fijar algunos controladores de eventos que nos permiten conocer el estado de la conectividad, los errores y los mensajes entrantes, para sus referencias:

1. WebSocket(wsUri) — crea un nuevo objeto WebSocket.
2. .onopen — El evento se produce cuando se establece la conexión.
3. .onclose — El evento se produce cuando la conexión está cerrada.
4. .onmessage — El evento se produce cuando el cliente recibe los datos del servidor.
5. .onerror — El evento se produce cuando hay un error.
6. .send(message) — Transmite datos a servidor utilizando una conexión abierta.
7. .close() — Termina la conexión existente.

Como se explica en los ejemplos anteriores, se comienza creando un objeto WebSocket, adjuntando los controladores de eventos y luego utilizando el método websocket.send para enviar los datos. Puesto que estamos enviando una colección de valores de chat, como el nombre de usuario y mensaje, convertiremos nuestros datos a formato JSON antes de enviarlos al servidor.

Servidor WebSocket

Ahora tenemos a nuestra página de chat en vivo listo para conectarse al servidor, pero también tenemos que crear un servidor WebSocket que se ejecuta de forma permanente (sin tiempos de espera).

**PHP Socket**

Básicamente nuestro servidor debe hacer lo siguiente:

1. Abrir un socket.
2. Enlazar a una dirección.
3. Escuchar las conexiones entrantes.
4. Aceptar conexiones.
5. WebSocket Handshake.
6. Tramas de datos Desenmascarar / Encode.

**ABRIR UN SOCKET:**

Primero creamos un socket con socket\_create ( int $domain , int $type , int $protocol ) de esta manera:

- $socket = socket\_create (AF\_INET, SOCK\_STREAM, SOL\_TCP);

**AF\_INET:** Protocolo basado en Internet IPv4. TCP y UDP son protocolos comunes de esta familia de protocolos.

**SOCK\_STREAM:** Proporciona flujos de bytes orientados a conexión, secuenciados, fiables y full-duplex.

Si el protocolo deseado es TCP o UDP, también se pueden usar las constantes SOL\_TCP, y SOL\_UDP correspondientes.

**ENLAZAR A UNA DIRECCIÓN:**

Vincular el nombre dado en address al socket descrito por socket socket\_bind ( resource$socket , string$address [, int$port = 0 ] ) , esto se tiene que hacer antes de establecer una conexión mediante socket\_connect () o socket\_listen ().

- socket\_bind ($socket, 'localhost');

**ESCUCHAR LAS CONEXIONES ENTRANTES:**

Una vez creado el socket, configuramos nuestro servidor para que escuche la conexión entrante en ese socket.

- socket\_listen($socket);

**ACEPTAR CONEXIONES:**

Esta función acepta conexiones entrantes.

- socket\_accept ($socket);

Pueden bajar el ejemplo completo desde aquí. Para que el servidor comience a esperar conexiones deben ejecutar desde la consola:

1 $ php server.php