

Socket.io

¿Qué son websockets?

Protocolo de comunicación

- Full-duplex
- Una sola conexión permanente
- Stream de mensajes
- Contenido en tiempo real

¿Qué son websockets?

Es decir...

- El cliente puede enviar y **recibir** datos en tiempo real
- Orientado a “eventos” (mensajes)
- Siempre conectado
- Baja latencia

Websockets y Node.js

Funcionan especialmente bien con Node.js

- El servidor maneja muchas conexiones simultáneas
- Buena integración con JSON
- Eventos

¿Para qué sirven?

Fundamentalmente, para:

- Actividades colaborativas
- Juegos multijugador
- Visores de eventos, logs, análisis en tiempo real
- Acelerar ciertas operaciones
 - Enviar datos
 - Cargar recursos
- En resumen: tiempo real en vez de “a petición”

Implicaciones

El navegador, mediante Javascript, inicia la conexión. Pero:

- Si se recarga o cambia la página, la conexión se corta.
- Por tanto, los clientes que utilizan WebSockets son siempre aplicaciones Single Page
- Para poder mantener el enlace con el servidor permanentemente

Estándar WebSocket

<https://tools.ietf.org/html/rfc6455>

Define la parte servidor y la parte cliente

- Protocolo: **ws://** o **wss://** (seguro)
- Implementado en navegadores modernos
- La adopción en el lado del servidor ha sido más lenta, sobre todo en sistemas multithread (Rails, .NET, etc...)
- En nodejs hay muchas implementaciones, construidas sobre el módulo **net** (sockets)

Estándar WebSocket

Soporte en browsers (Octubre 2015)

Web Sockets 📄 - CR										Global		86.58% + 0.72% = 87.3%
Bidirectional communication technology for web apps										unprefixed:		86.58% + 0.65% = 87.23%
										Spain		93.19% + 0.8% = 93.99%
										unprefixed:		93.19% + 0.73% = 93.91%
<div>Current aligned</div> <div>Usage relative</div> <div>Show all</div>												
IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Chrome for Android			
								4.1				
8		38	31					4.3				
9		39	43					4.4				
10		40	44	8		8.4		4.4.4				
11	12	41	45	9	32	9	8	44	45			
	13	42	46		33							
		43	47		34							
		44	48									

Socket.io

Vamos a usar Socket.io

- Una librería para manipular websockets
 - Nos ofrece a la vez la **parte servidor** y la **parte cliente**
 - Muy popular
 - Fallback para navegadores obsoletos
 - Muy fácil de usar
 - Sin fricciones con express
- ➡ <http://socket.io/>

Socket.io

Socket.io tiene dos partes:

- Servidor (Node.js):

```
var express = require("express"),  
    server = require("http").createServer(),  
    io = require("socket.io").listen(server),  
    app = express();
```

```
server.on("request", app).listen(3000);
```

- Cliente:

```
<script src="/socket.io/socket.io.js"></  
script>
```

Socket.io

Los sockets emiten eventos

- Un evento = un “mensaje”
- Se pueden pasar parámetros
- `socket.on(mensaje, callback)`
- `socket.emit(mensaje, [param1, param2, ...])`

Socket.io

server.js

```
var express = require("express"),
    app = express(),
    server = require("http").createServer(app),
    io = require("socket.io").listen(server);

app.use(express.static(__dirname + "/public"));

io.sockets.on("connection", function(socket) {
  socket.emit("ping");
  socket.on("pong", function() {
    console.log("PONG!");
  });
});

server.listen(3000);
```

Socket.io

server.js

```
var express = require("express"),  
    app = express(),  
    server = require("http").createServer(app),  
    io = require("socket.io").listen(server);
```

```
app.use(express.static(__dirname + "/public"));
```

```
io.sockets.on("connection", function(socket) {  
    socket.emit("ping");  
    socket.on("pong", function() {  
        console.log("PONG!");  
    });  
});
```

```
server.listen(3000);
```

Socket.io

server.js

```
var express = require("express"),  
    app = express(),  
    server = require("http").createServer(app),  
    io = require("socket.io").listen(server);
```

```
app.use(express.static(__dirname + "/public"));
```

```
io.sockets.on("connection", function(socket) {  
    socket.emit("ping");  
    socket.on("pong", function() {  
        console.log("PONG!");  
    });  
});
```

```
server.listen(3000);
```

Socket.io

index.html

```
<html>
  <head>
    <script src="/socket.io/socket.io.js"></script>
    <script type="text/javascript">
      var socket = io.connect("http://localhost:3000");

      socket.on("ping", function() {
        console.log("PING!");
        socket.emit("pong");
      });
    </script>
  </head>
  <body></body>
</html>
```

Socket.io

index.html

```
<html>
  <head>
    <script src="/socket.io/socket.io.js"></script>
    <script type="text/javascript">
      var socket = io.connect("http://localhost:3000");

      socket.on("ping", function() {
        console.log("PING!");
        socket.emit("pong");
      });
    </script>
  </head>
  <body></body>
</html>
```


Socket.io

index.html

```
<html>
  <head>
    <script src="/socket.io/socket.io.js"></script>
    <script type="text/javascript">
      var socket = io.connect("http://localhost:3000");

      socket.on("ping", function() {
        console.log("PING!");
        socket.emit("pong");
      });
    </script>
  </head>
  <body></body>
</html>
```

Socket.io

Eventos reservados (servidor):

- `io.sockets.on("connection", cb)`
- `socket.on("message", cb)`
- `socket.on("disconnect", cb)`

Cliente:

- `socket.on("connect", cb)`
- `socket.on("disconnect", cb)`
- `socket.on("error", cb)`
- `socket.on("message", cb)`

Socket.io

Métodos (servidor)

- `socket.broadcast.emit(msg)`
 - les llega a todos menos el emisor
- `socket.disconnect()`
- `socket.emit(msg) / socket.on(msg)`

Métodos (client)

- `var socket = io.connect(host)`
- `socket.disconnect()`
- `socket.emit(msg) / socket.on(msg)`

Socket.io

El servidor puede llamar a **callbacks en el cliente**, a modo de respuesta a un mensaje

```
socket.emit("givemesomething", 5, function(obj){  
  console.log('Server answered', obj);  
});
```

Servidor

```
socket.on('givemesomething', function(n, cb){  
  cb(n*10);  
})
```

Un Chat! (simple)

Vamos a hacer un chat sencillo:

- Los usuarios se loguean eligiendo un nick
 - Todo el mundo escribe en la misma sala común
 - No tenemos indicador de presencia
 - Tendremos que escribir código de servidor y de cliente (app.js y public/code.js)
-
- Esqueleto en **/tema6/chat**

Un Chat! (simple)

En el cliente:

- `Chat.registerHandler(cb)`: callback cuando el usuario escribe un mensaje
- `Chat.postMessage(user, msg)`: Muestra un mensaje de otro
- `Chat.showMyMsg(user, msg)`: Muestra un mensaje propio

Donde:

- `user: {avatar: <string>, name: <string>}`
- `msg: {text: <string>, date: <timestamp>}`

Canales

Con Socket.io podemos crear canales o *namespaces* para agrupar los receptores

```
var express = require("express"),
    app = express(),
    server = require("http").createServer(app),
    io = require("socket.io").listen(server);

app.use(express.static(__dirname + "/public"));

io.of("/canal").on("connection", function(socket) {
  socket.emit("ping");
});

server.listen(3000);
```

Canales

En el cliente:

```
<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
  var socket = io.connect("http://localhost:3000/canal");

  socket.on("ping", function() {
    console.log("PING!");
  });

</script>
```


Canales

Podemos tener varios canales simultáneos
(multiplexando el mismo websocket)

```
io.of("/canal").on("connection", function(socket) {  
    socket.emit("ping");  
});
```

```
io.of("/otro").on("connection", function(socket) {  
    socket.emit("bang!");  
});
```

Canales

En el cliente:

```
var canal = io.connect("http://localhost:3000/canal"),
    otro = io.connect("http://localhost:3000/otro");

canal.on("ping", function() {
    console.log("PING!");
});

otro.on("bang!", function() {
    console.log("Estoy herido!");
});
```

Rooms (salas)

Además de namespaces, socket.io incluye el concepto de sala

- Sólo el servidor mete y saca sockets de una sala
- `socket.join("room")` / `socket.leave("room")`
- Además se puede enviar (o hacer broadcast) a una sala específica:

```
io.to("room").emit(...)  
io.to("room").broadcast.emit(...)
```

¿Chat multisala?

Utilizando **rooms**, los usuarios podrían:

- Loguearse/registrarse (simpleauth)
- Crear salas
- Unirse y salirse de las salas creadas
- Escribir en la sala en la que estén
- Incluso estar en varias salas a la vez, al estilo IRC

¿Chat multisala?

Consejos:

- Puedes identificar un socket de cliente con **socket.id**
- En el servidor, puedes traer las salas está un cliente con **socket.rooms** (Array)
- En el servidor, puedes emitir a una sala con **socket.to("room").emit()**
- Utiliza la sesión (o req.user) para saber en qué sala está un usuario (clave del objeto de sockets + canal)
- Crea mensajes para:
 - Un usuario ha entrado en la sala
 - Un usuario ha salido de la sala
 - Alguien postea un mensaje

APIs sobre WebSockets

Aunque la mayoría de APIs siguen usando HTTP como transporte (REST APIs)...

Cada vez hay más aplicaciones que gestionan todo mediante WebSockets:

- Create/Read/Update/Delete de entidades
- Notificaciones de estas operaciones

APIs sobre WebSockets

Si estas operaciones se realizan mediante WebSocket, tenemos una oportunidad para notificar al resto de usuarios conectados de que se ha creado/modificado/borrado un nuevo *algo*

Evitamos el problema de que cada usuario vea una copia “congelada” de los datos en el servidor

Ejemplo API REST vs WS

Versión REST

GET /posts - devuelve la lista de Posts (SELECT)

POST /posts - crea un nuevo Post (INSERT)

GET /posts/id - devuelve un Post (SELECT 1)

PUT /posts/id - modifica un Post (UPDATE)

DELETE /posts/id - elimina un Post (DELETE)

Ejemplo API REST vs WS

Versión socket.io, nombres de mensajes

```
socket.emit("posts:list", function(posts){...})
```

```
socket.emit("posts:create", postObj,  
  function(post){...})
```

```
socket.emit("posts:get", postId,  
  function(post){ ... })
```

```
socket.emit("posts:update", postObj,  
  function(post){...})
```

```
socket.emit("posts:delete", function(result)  
  {...})
```

```
//Aviso: nuevo Post!
```

```
socket.on("posts:create", function(post){...})
```