Comenzamos creando una carpeta llamada SocketsIO y la establecemos como espacio de trabajo en Visual Studio Code. Damos por supuesto que tenemos la última versión de NodeJS y npm.

Nota: durante este tutorial emplearemos Emacscript para simplicidad del mismo.

Comenzamos creando el archivo package.json

npm init

Quedando en estos momentos con el contenido:

{

"name": "developodo.socketsio",

"version": "1.0.0",

"description": "Web Chat based on Sockets.IO",

"main": “server.js",

"author": "Developodo",

"license": "ISC"

}

Empleamos la librería express que permite implementar servidores web sobre NodeJS de forma sencilla y eficiente.

npm install --save express

Tal como indica el archivo package.json, el archivo de entrada es server.js (por defecto para index.js). Es hora de crearlo.

var app = require('express')();

var http = require('http').createServer(app);

/\*\*

\* We serve index.html when root page is required

\*/

app.get('/', function(req, res){

res.sendFile(\_\_dirname + '/index.html');

});

/\*\*

\* it starts web server on port 3000

\*/

http.listen(3000, function(){

//this message is shown on server side

console.log('listening on \*:3000');

});

Vemos cómo con pocas líneas tenemos un servidor web montado escuchando por el puerto 3000 y listo para servir la página index.html a los clientes.

Creamos ahora index.html:

<!doctype html>

<html>

<head>

<title>Socket.IO chat</title>

<style>

\* { margin: 0; padding: 0; box-sizing: border-box; }

body { font: 13px Helvetica, Arial; }

form { background: #000; padding: 3px; position: fixed; bottom: 0; width: 100%; }

form input { border: 0; padding: 10px; width: 90%; margin-right: .5%; }

form button { width: 9%; background: rgb(130, 224, 255); border: none; padding: 10px; }

#messages { list-style-type: none; margin: 0; padding: 0; }

#messages li { padding: 5px 10px; }

#messages li:nth-child(odd) { background: #eee; }

</style>

</head>

<body>

<ul id="messages"></ul>

<form action="">

<input id="m" autocomplete="off" /><button>Send</button>

</form>

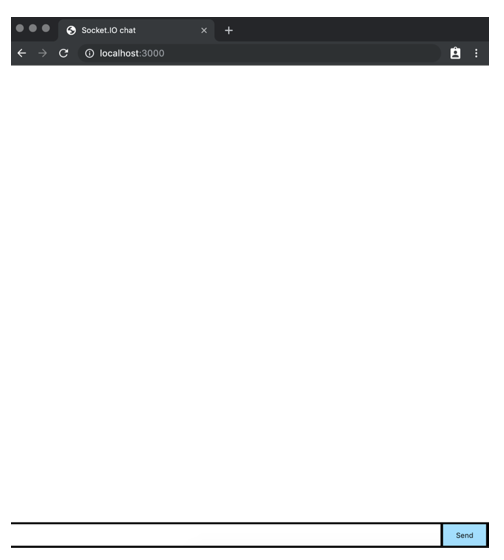
</body>

</html>

Y por último lanzamos el servidor con el comando:

node server

Podemos ver nuestra web visitando <http://localhost:3000>.



Hasta ahora solo hemos implementado un servidor web. Toca hacer uso de los WebSockets y para ello instalamos la librería sockets.io.

npm install --save socket.io

Añadimos a server.js las siguientes líneas:

// We create a socket server listening on http web server socket

var io = require('socket.io')(http);

io.on('connection', (socket)=>{

console.log('a user connected');

socket.on('disconnect',()=>{

console.log('user disconnected');

});

});

Observamos cómo se crea un websocket que recibe como parámetro el propio servidor web (http). Es decir, recibirá las peticiones por ese canal. Y también, se añade un controlador de eventos connection que será lanzado cada vez que alguien se conecte mediante http. Este evento recibe como parámetro el propio socket.

De hecho, se crea otro controlador de eventos dentro de la función flecha sobre el evento disconnect que ejecuta otra función cuando ese mismo usuario se ha desconectado.

Para poder probarlo deberemos desde el cliente (index.html) realizar una conexión de entrada. De este modo, index.html debemos añadir las siguientes líneas (al final de body, pero dentro de éste):

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

<script>

var socket = io();

</script>

Vemos como se importa la librería socket.io.js que se encuentra en node\_modules y que es alcanzable desde la raíz.

Por otra parte creamos un socket (io()) con ninguna url de entrada, por tanto toma como servidor el propio servidor web donde se está alojando la página: es nuestro caso, tanto servidor de sockets como web se encuentran en el mismo servicio.

Si volvemos a ejecutar:

node server

Y visitamos la página <http://localhost:3000>.

Veremos en la consola del servidor:

listening on \*:3000

a user connected

Si recargamos la página, tendremos:

listening on \*:3000

a user connected

user disconnected

a user connected

Es decir, cuando visitamos la página se establece un socket estable y abierto. Que no es cerrado hasta que abandonamos la web por una recarga o porque cerramos la pestaña. En este caso la salida definitiva, una vez cerrado el navegador sería:

listening on \*:3000

a user connected

user disconnected

a user connected

user disconnected

Una de las funcionalidades más potentes de los sockets son la emisión y control de eventos que permiten la comunicación asíncrona entre clientes y servidores.

Comencemos añadiendo la funcionalidad al cliente (index.html) para enviar mensajes al servidor cuando alguien escribe en el cuadro de texto y le da a enviar. Añadimos el siguiente código reemplazando el anterior que creaba la conexión:

<script>

// This anonimous function is executed when page is loaded

(() => {

var socket = io();

document.querySelector('form').onsubmit=(e)=>{

e.preventDefault(); // prevents page reloading

if(document.querySelector('#m').value==="")

return false;

socket.emit('chat message', document.querySelector('#m').value);

document.querySelector('#m').value="";

return false;

};

})();

var socket = io();

</script>

Hemos empleado Javascript puro para asociar un evento onsubmit al formulario en el cual recogeremos el contenido del input text y lo enviaremos al socket asociado a un evento denominado “chat message”.

Para poder controlar estos eventos modificaremos el contenido del server.js de la siguiente forma:

io.on('connection', (socket)=>{

console.log('a user connected');

socket.on('disconnect', ()=>{

console.log('user disconnected');

});

// When a client send a "chat message" event

socket.on('chat message', (msg)=>{

console.log('message: ' + msg);

});

});

Si ejecutamos de nuevo el servidor, visitamos la web, escribimos “Hello Developodo” en el campo de texto y pulsado enter o el botón de enviar, tendremos el siguiente contenido en la consola del servidor:

a user connected

message: Hello Developodo

Una vez que sabemos enviar mensajes desde el cliente al servidor, veamos algunos de los muchos mecanismos que existen para enviar información del servidor al cliente.

Modificamos el contenido de server.js para que contenga el siguiente código:

io.on("connection", socket => {

console.log("a user connected");

socket.on("disconnect", () => {

console.log("user disconnected");

socket.broadcast.emit("chat message", socket["nickname"] + " is gone");

});

// When a client send a "chat message" event

socket.on("chat message", msg => {

console.log("message: " + msg);

//Broadcast message to all sockets

io.emit("chat message", socket["nickname"] + " says: " + msg);

});

socket.on("whoami", nickname => {

// Unicast message to this socket

socket.emit(

"chat message",

"Welcome " + nickname + " to Developodo Chat Room"

);

// Send broadcast to all sockets except sender.

socket.broadcast.emit("chat message", nickname + " is in the room");

socket["nickname"] = nickname;

});

});

La línea:

socket.emit('chat message','Welcome to Developodo Chat Room’);

Envía un mensaje a un socket en concreto.

Vemos como la línea:

socket.broadcast.emit('chat message','A new user is in the room');

Envía un mensaje de broadcast a todos los sockets, excepto al que lanza el evento.

Sin embargo, la línea:

io.emit('chat message', msg);

Envía un mensaje de broadcast a todos los sockets.

Podemos encontrar resumido algunos formatos del comando emit en esta página: <https://socket.io/docs/emit-cheatsheet/>

Además, se ha creado un nuevo evento para enviar el nickname y almacenarlo dentro del objeto socket.

Modificamos el cliente para que sea operativo (index.html):

<script>

// This anonimous function is executed when page is loaded

(() => {

let user=''

while((user=prompt("NickName?"))=='');

let socket = io();

socket.emit("whoami", user);

document.querySelector("form").onsubmit = e => {

e.preventDefault(); // prevents page reloading

if (document.querySelector("#m").value === "") return false;

socket.emit("chat message", document.querySelector("#m").value);

document.querySelector("#m").value = "";

return false;

};

socket.on("chat message", (msg)=> {

let li = document.createElement("li")

let text = document.createTextNode(msg);

li.appendChild(text);

document.querySelector("#messages").append(li);

});

})();

</script>

Cuando realizamos la conexión, enviamos el nickname que hemos recogido por prompt para que sea almacenado en el servidor.

Ejecutar npm install en la carpeta para instalar las dependencias necesarias.

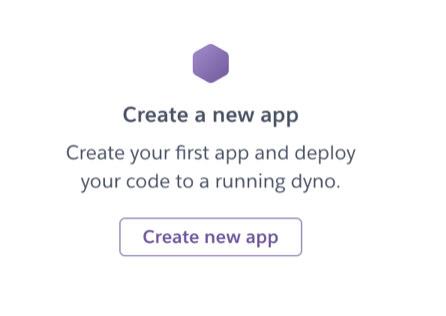
Una vez iniciado el servidor (node server) si lo visualizamos desde dos navegadores obtendremos un comportamiento similar al siguiente (http:// <http://localhost:5000/>):

|  |
| --- |
|  |
|  |

### Despliegue

En esta ocasión vamos a realizar el despliegue del servicio creado en Heroku (<https://www.heroku.com/>) que nos proporciona hosting con compatibilidad para diferentes plataformas, como es NodeJS.

Tras crear la cuenta gratuita y verificar el correo accedemos al panel de control. Desde ahí crearemos una nueva aplicación, asignando país y nombre.



En la siguiente página nos indican los métodos de despliegue. En nuestro caso emplearemos el cliente de Heroku. En primer lugar, lo instalaremos siguiendo los siguientes pasos, en función de nuestro sistema: <https://devcenter.heroku.com/articles/heroku-cli>.

Una vez instalado, procederemos a iniciar sesión ejecutando en la consola (dentro de nuestro proyecto):

heroku login

Una vez hemos iniciado sesión, seguiremos los pasos que nos indica la propia página de creación de aplicación. En nuestro ejemplo serían (dentro de la carpeta del proyecto):

git init

heroku git:remote -a chatdevelopodo

git add .

git commit -am "Let’s deploy"

Como se observa, hacemos uso de la herramienta de control de versiones y repositorio Git (<https://git-scm.com/>)

Antes de realizar el despliegue, algunas consideraciones:

* No podemos definir el puerto por el que escucha Heroku, por tanto, el archivo server.js tiene la siguiente modificación:

/\*\*

it starts web server on port ...

\*/

http.listen(process.env.PORT || 5000, function(){

console.log('listening on Heroku port');

});

* Debemos indicar cuál es el punto de entrada para npm start, para ello modificamos el archivo package.json para que tenga el siguiente contenido adicional:

…

"scripts": {

"start": "node server.js"

},

…

Ahora, desplegamos:

git push heroku master

Automáticamente, subiremos todos nuestros archivos al repositorio y Heroku instala y arranca los servicios necesarios para el despliegue.

Tras unos instantes, la consola finaliza y nos devuelve la url de acceso: <https://chatdevelopodo.herokuapp.com/>