INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

## Contenido

# Socket.IO - Descripción general

Socket.IO es una biblioteca de JavaScript para **aplicaciones web en tiempo real.** Permite la comunicación bidireccional en tiempo real entre servidores y clientes web. Tiene dos partes: una **biblioteca del lado del cliente** que se ejecuta en el navegador y una **biblioteca del lado del servidor** para node.js. Ambos componentes tienen una API idéntica.

## Aplicaciones en tiempo real

Una aplicación en tiempo real (RTA) es una aplicación que funciona dentro de un período que el usuario percibe como inmediato o actual.

Algunos ejemplos de aplicaciones en tiempo real son:

- **Mensajería instantánea** : aplicaciones de chat como Whatsapp, Facebook Messenger, etc. No es necesario que actualice su aplicación / sitio web para recibir mensajes nuevos.
- **Notificaciones** automáticas: cuando alguien te etiqueta en una imagen en Facebook, recibes una notificación al instante.
- **Aplicaciones de colaboración** : aplicaciones como google docs, que permiten que varias personas actualicen los mismos documentos simultáneamente y apliquen cambios a todas las instancias de las personas.
- **Juegos en línea** : juegos como Counter Strike, Call of Duty, etc., también son algunos ejemplos de aplicaciones en tiempo real.

## ¿Por qué Socket.IO?

Escribir una aplicación en tiempo real con pilas de aplicaciones web populares como LAMP (PHP) ha sido tradicionalmente muy difícil. Implica sondear el servidor en busca de cambios, realizar un seguimiento de las marcas de tiempo y es mucho más lento de lo que debería ser.

Los sockets han sido tradicionalmente la solución en torno a la cual se diseñan la mayoría de los sistemas en tiempo real, proporcionando un canal de comunicación bidireccional entre un cliente y un servidor. Esto significa que el servidor puede enviar mensajes a los clientes. Siempre que ocurre un evento,

la idea es que el servidor lo obtenga y lo envíe a los clientes conectados en cuestión.

Socket.IO es muy popular, que es utilizado por **Microsoft Office, Yammer, Zendesk, Trello,** . y muchas otras organizaciones para construir sistemas robustos en tiempo real. Es uno de los **frameworks JavaScript** más poderosos en **GitHub** , y el módulo NPM (Node Package Manager) más dependiente. Socket.IO también tiene una gran comunidad, lo que significa que encontrar ayuda es bastante fácil.

## ExpressJS

Usaremos express para construir el servidor web con el que trabajará Socket.IO. Se puede utilizar cualquier otro marco del lado del servidor de nodo o incluso el servidor HTTP de nodo. Sin embargo, ExpressJS facilita la definición de rutas y otras cosas. Para leer más sobre Express y tener una idea básica al respecto, diríjase al tutorial de ExpressJS.

# Socket.IO - Entorno

Para comenzar a desarrollar con **Socket.IO** , debe tener **instalados Node** y **npm (administrador de paquetes de nodos)** . Si no los tiene, diríjase a la **configuración del nodo** para instalar el nodo en su sistema local. Confirme que node y npm estén instalados ejecutando los siguientes comandos en su terminal.

```
node --version
npm --version
```

Debería obtener una salida similar a:

```
v14.17.0
```

6.14.13 Abra su terminal e ingrese lo siguiente en su terminal para crear una nueva carpeta e ingrese los siguientes comandos:

```
$ mkdir test-project
$ cd test-proect
$ npm init
```

Le hará algunas preguntas; respóndeles de la siguiente manera:

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

```
~$ mkdir test-project
~$ cd test-project/
~/test-project$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (test-project)
version: (1.0.0)
description: A project to learn socket.IO
entry point: (index.js)
test command:
git repository:
keywords:
author: Ayush Gupta
license: (ISC)
About to write to /home/ayushgp/test-project/package.json:

{
  "name": "test-project",
  "version": "1.0.0",
  "description": "A project to learn socket.IO",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Ayush Gupta",
  "license": "ISC"
}


Is this ok? (yes)
~/test-project$
```

Esto creará un archivo de configuración **'package.json node.js'** . Ahora necesitamos instalar **Express** y **Socket.IO.** Para instalarlos y guardarlos en

el **archivo package.json,** ingrese el siguiente comando en su terminal, en el directorio del proyecto:

```
npm install --save express socket.io
```

Una última cosa es que debemos seguir reiniciando el servidor. Cuando hagamos cambios, necesitaremos una herramienta llamada **nodemon.** Para instalar nodemon, abra su terminal e ingrese el siguiente comando:

```
npm install -g nodemon
```

Siempre que necesite iniciar el servidor, en lugar de usar el **nodo app.js** , use **nodemon app.js.** Esto asegurará que no necesite reiniciar el servidor cada vez que cambie un archivo. Acelera el proceso de desarrollo.

Ahora, tenemos nuestro entorno de desarrollo configurado. Comencemos ahora con el desarrollo de aplicaciones en tiempo real con Socket.IO.

# Socket.IO - Hola mundo

En el siguiente capítulo discutiremos el ejemplo básico usando la biblioteca Socket.IO junto con ExpressJS.

## Ejemplo

En primer lugar, cree un archivo llamado **app.js** e ingrese el siguiente código para configurar una aplicación rápida:

```
var app = require('express')();
var http = require('http').Server(app);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');
});

http.listen(3000, function(){
   console.log('listening on *:3000');
});
```

Necesitaremos un archivo **index.html** para servir, cree un nuevo archivo llamado index.html e ingrese el siguiente código en él:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <body>Hello world</body>
</html>
```

Para probar si esto funciona, vaya a la terminal y ejecute esta aplicación usando el siguiente comando:

```
nodemon app.js
```

Esto ejecutará el servidor en localhost: 3000. Vaya al navegador e ingrese localhost: 3000 para verificar esto. Si todo va bien, se imprime en la página un mensaje que dice ***"Hola mundo"*** .

A continuación se muestra otro ejemplo (esto requiere Socket.IO), registrará "Un usuario conectado", cada vez que un usuario vaya a esta página y "Un usuario desconectado", cada vez que alguien navegue o cierre esta página.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
res.sendFile('E:/test/index.html');
});

//Whenever someone connects this gets executed
io.on('connection', function(socket){
   console.log('A user connected');

   //Whenever someone disconnects this piece of code executed
   socket.on('disconnect', function () {
      console.log('A user disconnected');
   });
});
http.listen(3000, function(){
   console.log('listening on *:3000');
});
```

El require ('socket.io') (http) crea una nueva instancia de socket.io adjunta al servidor http. El **controlador de eventos io.on** maneja la conexión, desconexión, etc., eventos en él, usando el objeto socket.

Hemos configurado nuestro servidor para registrar mensajes sobre conexiones y desconexiones. Ahora tenemos que incluir el script del cliente e inicializar el objeto de socket allí, para que los clientes puedan establecer conexiones cuando sea necesario. El script es servido por nuestro servidor io en **'/socket.io/socket.io.js'.**

Después de completar el procedimiento anterior, el archivo index.html tendrá el siguiente aspecto:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
   </script>
   <body>Hello world</body>
</html>
```

Si va a localhost: 3000 ahora (asegúrese de que su servidor esté funcionando), obtendrá **Hello World** impreso en su navegador. Ahora verifique los registros de la consola del servidor, se mostrará el siguiente mensaje:

```
A user connected
```

Si actualiza su navegador, desconectará la conexión del socket y se volverá a crear. Puede ver lo siguiente en los registros de su consola:

```
A user connected
A user disconnected
A user connected
```

## Socket.IO - Manejo de eventos

Los enchufes funcionan según los eventos. Hay algunos eventos reservados, a los que se puede acceder utilizando el objeto socket en el lado del servidor.

Estos son ...

- Connect
- Message
- Disconnect
- Reconnect
- Ping
- Join and
- Leave.

El objeto de socket del lado del cliente también nos proporciona algunos eventos reservados, que son:

- Connect
- Connect_error
- Connect_timeout
- Reconnect, etc.

Ahora, veamos un ejemplo para manejar eventos usando la biblioteca SocketIO.

## Ejemplo 1

```
In the Hello World example, we used the connection and
disconnection events to
log when a user connected and left. Now we will be using the
message event to
pass message from the server to the client. To do this,
modify the io.on
('connection', function(socket)) as shown below -var app =
require('express')();
var http = require('http').Server(app);
```

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

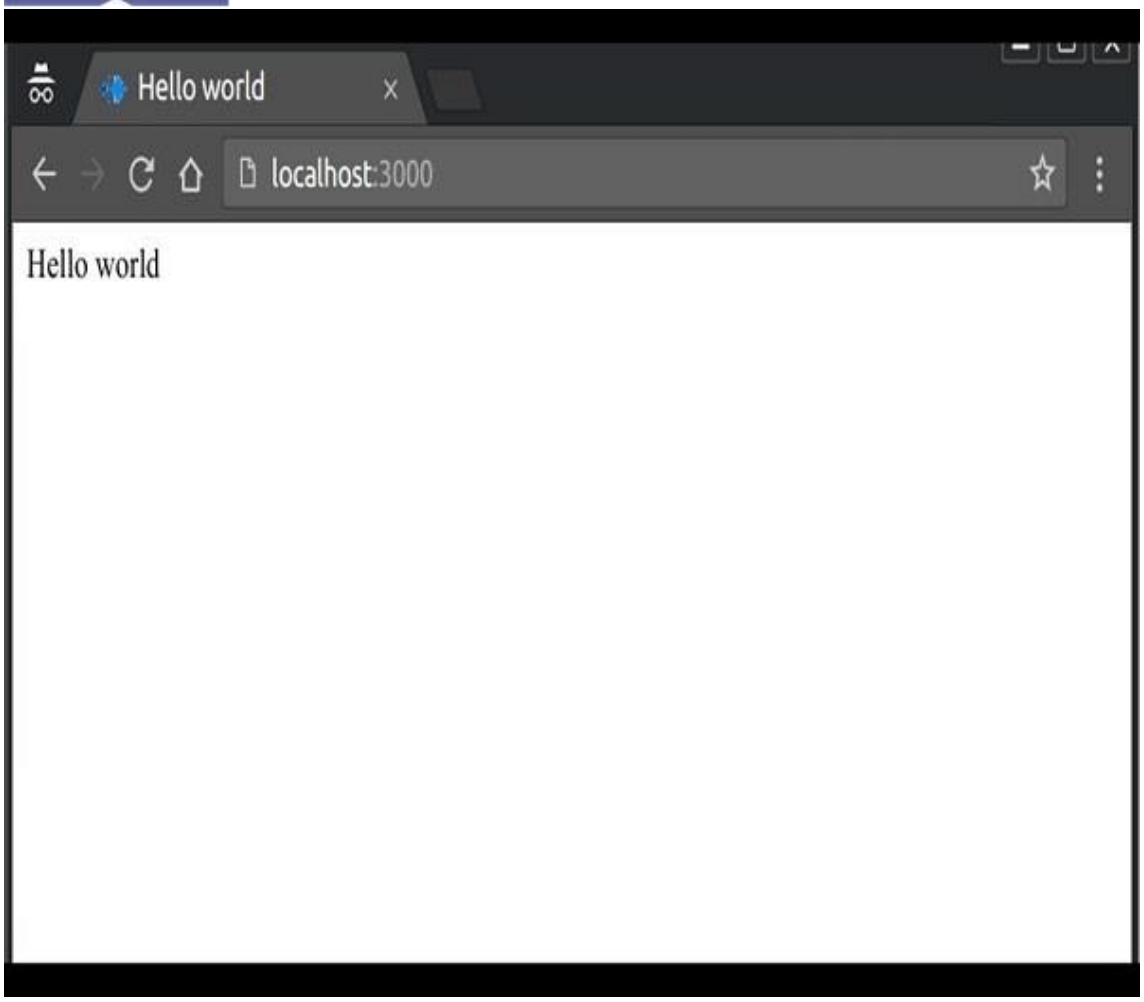
```
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');
});

io.on('connection', function(socket){
   console.log('A user connected');

   // Send a message after a timeout of 4seconds
   setTimeout(function(){
      socket.send('Sent a message 4seconds after
connection!');
   }, 4000);
   socket.on('disconnect', function () {
      console.log('A user disconnected');
   });
});
http.listen(3000, function(){
   console.log('listening on *:3000');
});
```

Esto enviará un evento llamado **message(built in)** a nuestro cliente, cuatro segundos después de que el cliente se conecte. La función de envío en el objeto socket asocia el evento 'mensaje'.

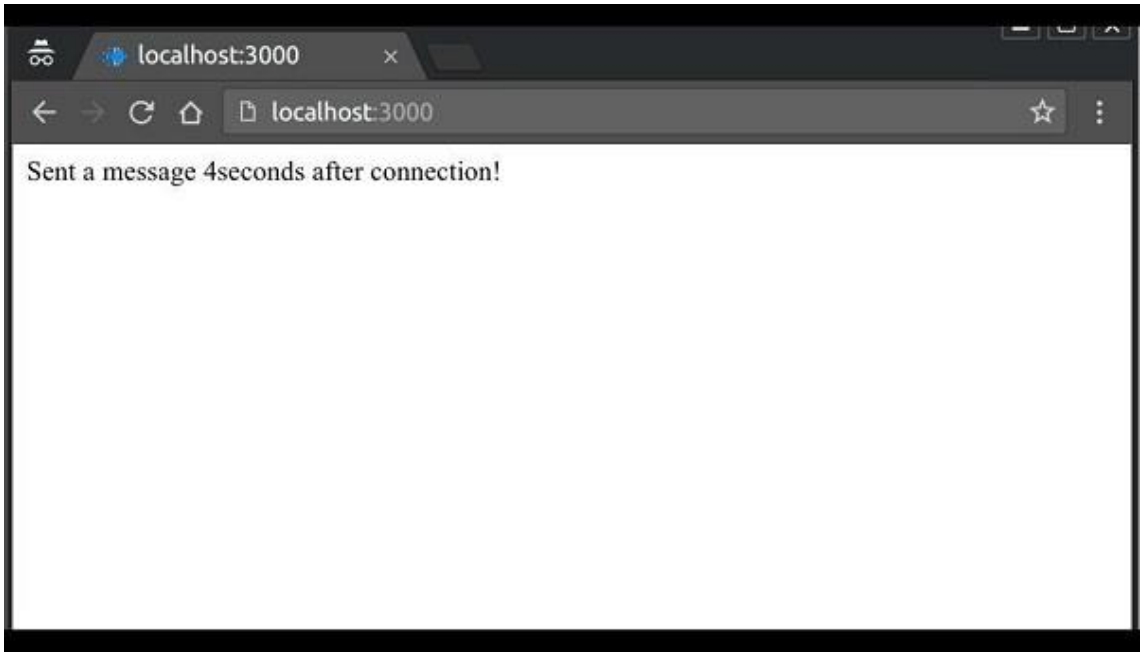
Ahora, necesitamos manejar este evento en nuestro lado del cliente, para hacerlo, reemplace el contenido de la página index.html con lo siguiente:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      socket.on('message',
function(data){document.write(data)});
   </script>
   <body>Hello world</body>
</html>
```

Ahora estamos manejando el evento 'mensaje' en el cliente. Cuando vaya a la página en su navegador ahora, se le presentará la siguiente captura de pantalla.




Después de que pasen 4 segundos y el servidor envíe el evento de mensaje, nuestro cliente lo manejará y producirá el siguiente resultado:



**Nota** : enviamos una cadena de texto aquí; también podemos enviar un objeto en cualquier caso.

El mensaje era un evento integrado proporcionado por la API, pero no es de mucha utilidad en una aplicación real, ya que necesitamos poder diferenciar entre eventos.

Para permitir esto, Socket.IO nos brinda la capacidad de crear **eventos personalizados** . Puede crear y disparar eventos personalizados usando la función **socket.emit** . El siguiente código emite un evento llamado **testerEvent** -

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');
});

io.on('connection', function(socket){
   console.log('A user connected');
   // Send a message when
   setTimeout(function(){
      // Sending an object when emmiting an event
      socket.emit('testerEvent', { description: 'A custom
event named testerEvent!'});
   }, 4000);
   socket.on('disconnect', function () {
      console.log('A user disconnected');
   });
});
http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

Para manejar este evento personalizado en el cliente, necesitamos un oyente que escuche el evento testerEvent. El siguiente código maneja este evento en el cliente:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      socket.on('testerEvent',
function(data){document.write(data.description)});
   </script>
   <body>Hello world</body>
</html>
```

Esto funcionará de la misma manera que en nuestro ejemplo anterior, con el evento testerEvent en este caso. Cuando abra su navegador y vaya a localhost: 3000, será recibido con:

```
Hello world
```

Después de cuatro segundos, este evento se activará y el navegador cambiará el texto a -

```
A custom event named testerEvent!
```

## Ejemplo 2

También podemos emitir eventos desde el cliente. Para emitir un evento desde su cliente, use la función emitir en el objeto socket.

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      socket.emit('clientEvent', 'Sent an event from the
client!');
   </script>
   <body>Hello world</body>
</html>
```

Para manejar estos eventos, use la **función** on en el objeto socket en su servidor.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');
});
io.on('connection', function(socket){
   socket.on('clientEvent', function(data){
      console.log(data);
   });
});

http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

Entonces, ahora si vamos a localhost: 3000, obtendremos un evento personalizado llamado **clientEvent** disparado. Este evento se manejará en el servidor iniciando sesión:

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

Sent an event from the client!

# Socket.IO - Difusión

Difundir significa enviar un mensaje a todos los clientes conectados. La transmisión se puede realizar en múltiples niveles. Podemos enviar el mensaje a todos los clientes conectados, a los clientes en un espacio de nombres y a los clientes en una habitación en particular. Para transmitir un evento a todos los clientes, podemos usar el método **io.sockets.emit** .

**Nota** - Esto emitirá el evento a **TODOS** los clientes conectados (evento al socket que pudo haber disparado este evento).

En este ejemplo, transmitiremos el número de clientes conectados a todos los usuarios. Actualice el archivo **app.js** para incorporar lo siguiente:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');
});

var clients = 0;

io.on('connection', function(socket){
   clients++;
   io.sockets.emit('broadcast',{ description: clients + '
clients connected!'});
   socket.on('disconnect', function () {
      clients--;
      io.sockets.emit('broadcast',{ description: clients + '
clients connected!'});
   });
});

http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

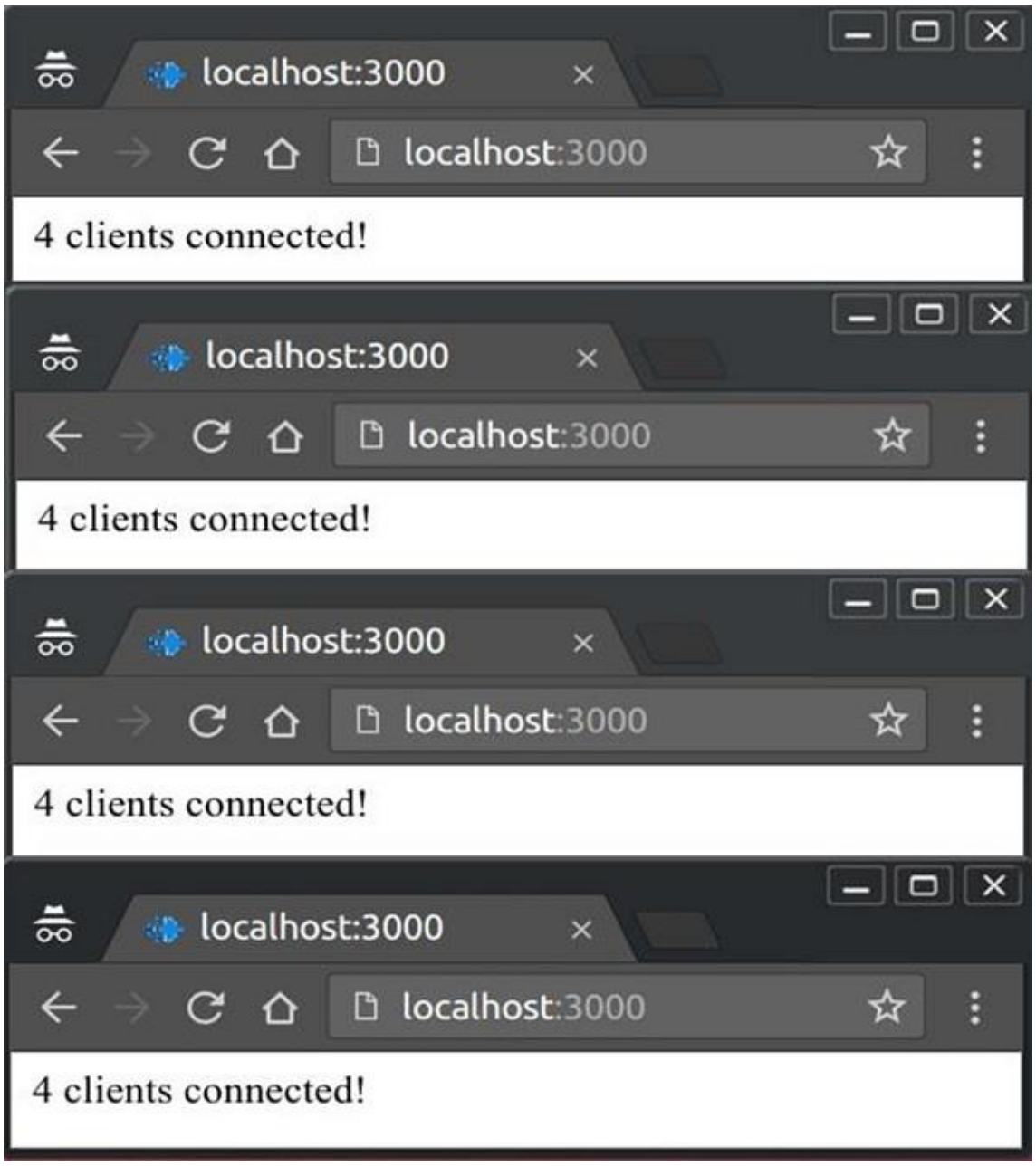
En el lado del cliente, solo necesitamos manejar el evento de transmisión:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      socket.on('broadcast',function(data){
         document.body.innerHTML = '';
         document.write(data.description);
      });
```

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

```
    </script>
    <body>Hello world</body>
</html>
```

Si conecta cuatro clientes, obtendrá el siguiente resultado:




Esto fue para enviar un evento a todos. Ahora, si queremos enviar un evento a todos, menos al cliente que lo causó (en el ejemplo anterior, fue causado por nuevos clientes al conectarse), podemos usar **socket.broadcast.emit.**

Permítanos enviarle al nuevo usuario un mensaje de bienvenida y actualizar a los demás clientes sobre su incorporación. Entonces, en su archivo app.js, al conectar el cliente, envíele un mensaje de bienvenida y transmita el número de cliente conectado a todos los demás.

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');
});
var clients = 0;

io.on('connection', function(socket){
   clients++;
   socket.emit('newclientconnect',{ description: 'Hey,
welcome!'});
   socket.broadcast.emit('newclientconnect',{ description:
clients + ' clients connected!'})
   socket.on('disconnect', function () {
      clients--;
      socket.broadcast.emit('newclientconnect',{ description:
clients + ' clients connected!'})
   });
});
http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

Y tu html para manejar este evento -

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      socket.on('newclientconnect',function(data){
               document.body.innerHTML = '';
               document.write(data.description);
      });
   </script>
   <body>Hello world</body>
</html>
```

Ahora, el cliente más nuevo recibe un mensaje de bienvenida y otros obtienen cuántos clientes están conectados actualmente al servidor.

## Socket.IO - Espacios de nombres

Socket.IO le permite asignar un "espacio de nombres" a sus sockets, lo que esencialmente significa asignar diferentes puntos finales o rutas. Esta es una característica útil para minimizar la cantidad de recursos (conexiones TCP) y, al mismo tiempo, separar las preocupaciones dentro de su aplicación al introducir la separación entre los canales de comunicación. En realidad, varios espacios

de nombres comparten la misma conexión WebSockets, lo que nos ahorra puertos de socket en el servidor.

Los espacios de nombres se crean en el lado del servidor. Sin embargo, los clientes se unen a ellos enviando una solicitud al servidor.

## Espacios de nombres predeterminados

El espacio de nombres raíz '/' es el espacio de nombres predeterminado, al que se unen los clientes si el cliente no especifica un espacio de nombres mientras se conecta al servidor. Todas las conexiones al servidor que utilizan el lado del cliente del objeto de socket se realizan en el espacio de nombres predeterminado. Por ejemplo

```
var socket = io();
```

Esto conectará al cliente al espacio de nombres predeterminado. Todos los eventos en esta conexión de espacio de nombres serán manejados por el **objeto io** en el servidor. Todos los ejemplos anteriores utilizaban espacios de nombres predeterminados para comunicarse con el servidor y viceversa.

## Espacios de nombres personalizados

Podemos crear nuestros propios espacios de nombres personalizados. Para configurar un espacio de nombres personalizado, podemos llamar a la función 'of' en el lado del servidor:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');});

var nsp = io.of('/my-namespace');
nsp.on('connection', function(socket){
   console.log('someone connected');
   nsp.emit('hi', 'Hello everyone!');
});
http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

Ahora, para conectar un cliente a este espacio de nombres, debe proporcionar el espacio de nombres como un argumento para la **llamada** al **constructor io** para crear una conexión y un objeto de socket en el lado del cliente.

Por ejemplo, para conectarse al espacio de nombres anterior, utilice el siguiente HTML:

```
<!DOCTYPE html>
```

```
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io('/my-namespace');
      socket.on('hi',function(data){
         document.body.innerHTML = '';
         document.write(data);
      });
   </script>
   <body></body>
 </html>
```

Cada vez que alguien se conecta a este espacio de nombres, recibirá un evento de 'hola' con el mensaje **"¡Hola a todos!"** .

# Socket.IO - Rooms

Dentro de cada espacio de nombres, también puede definir canales arbitrarios a los que los sockets pueden unirse y salir. Estos canales se denominan rooms. Las habitaciones se utilizan para separar aún más las preocupaciones. Las habitaciones también comparten la misma conexión de socket como los espacios de nombres. Una cosa a tener en cuenta al usar las salas es que solo se pueden unir en el lado del servidor.

## Unirse a las rooms

Puede llamar al método de **join** en el socket para suscribir el socket a un canal / room determinado. Por ejemplo, creemos salas llamadas **'room- <room-number>'** y unamos algunos clientes. Tan pronto como esta sala esté llena, cree otra sala y únase a los clientes allí.

**Nota** : actualmente estamos haciendo esto en el espacio de nombres predeterminado, es decir, '/'. También puede implementar esto en espacios de nombres personalizados de la misma manera.

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);
app.get('/', function(req, res){
   res.sendfile('index.html');
});
var roomno = 1;
io.on('connection', function(socket){
   socket.join("room-"+roomno);
   //Send this event to everyone in the room.
   io.sockets.in("room-"+roomno).emit('connectToRoom', "You
are in room no. "+roomno);
})
http.listen(3000, function(){
```

```
   console.log('listening on localhost:3000');
});
```

Simplemente maneje este evento **connectToRoom** en el cliente.

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      socket.on('connectToRoom',function(data){
         document.body.innerHTML = '';
         document.write(data);
      });
   </script>
   <body></body>
</html>
```

Ahora, si conecta tres clientes, los dos primeros recibirán el siguiente mensaje:

```
You are in room no. 1
```

## Salir de una habitación

Para salir de una habitación, debe llamar a la función de salida del mismo modo que llamó a la función de unión en el socket.

Por ejemplo: para dejar la habitación ' **habitación-1** ',

```
socket.leave("room-"+roomno);
```

# Socket.IO - Manejo de errores

Hemos trabajado en servidores locales hasta ahora, lo que casi nunca nos dará errores relacionados con conexiones, tiempos de espera, etc. Sin embargo, en entornos de producción de la vida real, el manejo de dichos errores es de suma importancia. Por lo tanto, ahora discutiremos cómo podemos manejar los errores de conexión en el lado del cliente.

La API del cliente nos proporciona los siguientes eventos integrados:

- **Connect** : cuando el cliente se conecta correctamente.
- **Connecting** : cuando el cliente está en proceso de conexión.
- **Disconnect**: cuando el cliente está desconectado.
- **Connect_failed** : cuando falla la conexión al servidor.
- **Error** : se envía un evento de error desde el servidor.
- **Message** : cuando el servidor envía un mensaje mediante la función de **envío** .

- **Reconnet** : cuando la reconexión al servidor se realiza correctamente.
- **Reconnecting** : cuando el cliente está en proceso de conexión.
- **Reconnect_failed** : cuando falla el intento de reconexión.

Para manejar errores, podemos manejar estos eventos usando el objeto out-socket que creamos en nuestro cliente.

Por ejemplo - Si tenemos una conexión que falla, podemos usar el siguiente código para conectarnos nuevamente al servidor -

```
socket.on('connect_failed', function() {
   document.write("Sorry, there seems to be an issue with the
connection!");
})
```

# Socket.IO - Registro y depuración

Socket.IO utiliza un módulo de depuración muy famoso desarrollado por el autor principal de ExpresJS, llamado debug. Anteriormente, Socket.IO solía registrar todo en la consola, lo que dificultaba bastante la depuración del problema. Después de la versión v1.0, puede especificar qué desea registrar.

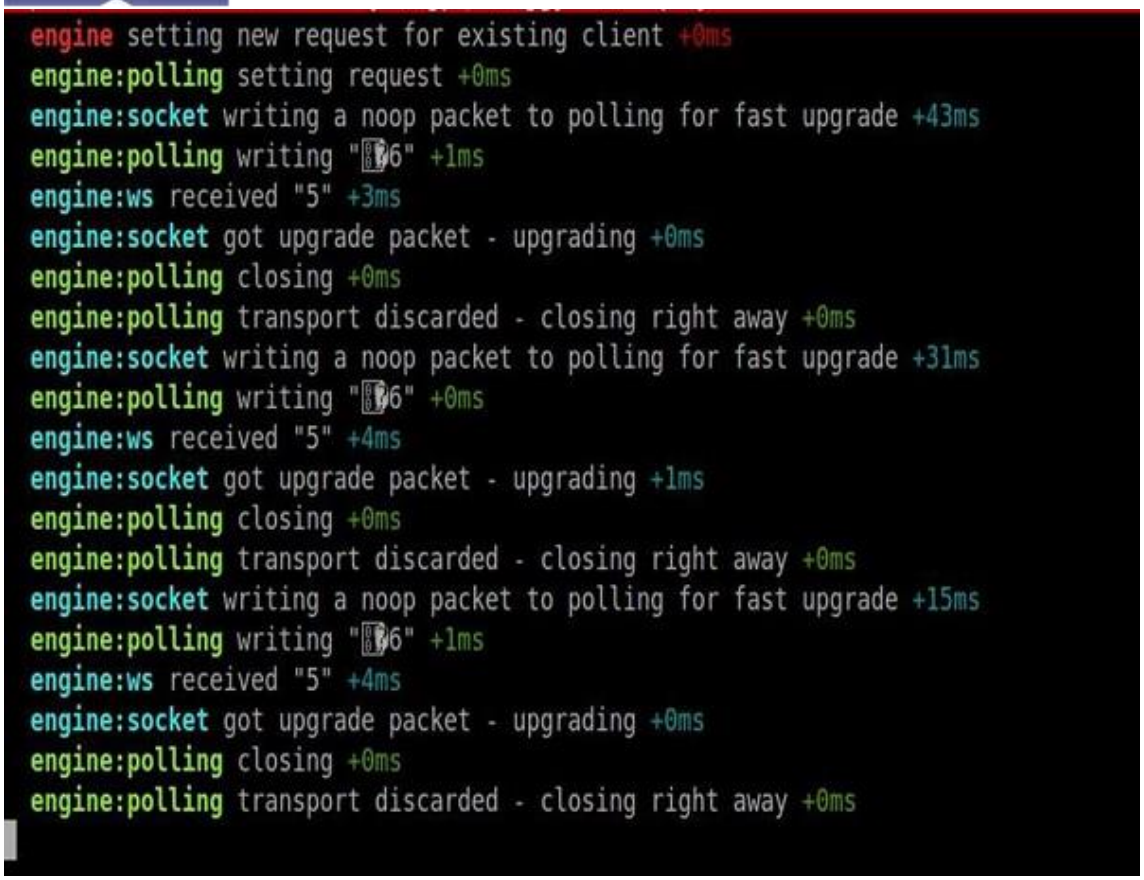
## Lado del servidor

La mejor manera de ver qué información está disponible es usar el * -

```
DEBUG=* node app.js
```

Esto coloreará y dará salida a todo lo que suceda en la consola de su servidor. Por ejemplo, podemos considerar la siguiente captura de pantalla.

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES




## Lado del cliente

Pegue esto en la consola, haga clic en entrar y actualice su página. Esto volverá a generar todo lo relacionado con Socket.io en su consola.

```
localStorage.debug = '*';
```

Puede limitar la salida para obtener la información de depuración con los datos entrantes del socket usando el siguiente comando.

```
localStorage.debug = 'socket.io-client:socket';
```

Puede ver el resultado como en la siguiente captura de pantalla, si usa la segunda declaración para registrar la información:


Hay una muy buena publicación de blog relacionada con la depuración de socket.io aquí.

# Socket.IO - Internos

En este capítulo, discutiremos sobre Fallbacks, Conexión usando Socket.IO, Eventos y Mensajes.

## Fallbacks

Socket.IO tiene muchos mecanismos de transporte subyacentes, que se ocupan de diversas limitaciones que surgen debido a problemas entre navegadores, implementaciones de WebSocket, cortafuegos, bloqueo de puertos, etc.

Aunque W3C tiene una especificación definida para la API de WebSocket, todavía carece de implementación. Socket.IO nos proporciona mecanismos de reserva que pueden solucionar estos problemas. Si desarrollamos aplicaciones utilizando la API nativa, tenemos que implementar las alternativas nosotros mismos. Socket.IO cubre una gran lista de alternativas en el siguiente orden:

- WebSockets
- FlashSocket
- XHR long polling
- XHR multipart streaming
- XHR polling
- JSONP polling
- iframes

## Conexión mediante Socket.IO

La conexión Socket.IO comienza con el protocolo de enlace. Esto hace que el apretón de manos sea una parte especial del protocolo. Aparte del protocolo de

enlace, todos los demás eventos y mensajes del protocolo se transfieren a través del conector.

Socket.IO está diseñado para su uso con aplicaciones web y, por lo tanto, se supone que estas aplicaciones siempre podrán usar HTTP. Es debido a este razonamiento que el protocolo de enlace de Socket.IO se lleva a cabo a través de HTTP mediante una solicitud POST en el URI de protocolo de enlace (pasado al método de conexión).

## Eventos y mensajes

La API nativa de WebSocket solo envía mensajes a través de. Socket.IO proporciona una capa adicional sobre estos mensajes, lo que nos permite crear eventos y nuevamente nos ayuda a desarrollar aplicaciones fácilmente al separar los diferentes tipos de mensajes enviados.

La API nativa envía mensajes solo en texto sin formato. Socket.IO también se encarga de esto. Maneja la serialización y deserialización de datos por nosotros.

Disponemos de una API cliente oficial para la web. Para otros clientes como teléfonos móviles nativos, otros clientes de aplicaciones también podemos usar Socket.IO siguiendo los siguientes pasos.

- **Paso 1** : se debe establecer una conexión utilizando el mismo protocolo de conexión descrito anteriormente.
- **Paso 2** : los mensajes deben estar en el mismo formato especificado por Socket.IO. Este formato permite a Socket.IO determinar el tipo de mensaje y los datos enviados en el mensaje y algunos metadatos útiles para la operación.

El formato del mensaje es:

```
[type] : [id ('+')] : [endpoint] (: [data]
```

Los parámetros del comando anterior se explican a continuación:

- **El type** es un número entero de un solo dígito, que especifica qué tipo de mensaje es.
- **ID** es el ID de mensaje, un número entero incremental que se utiliza para reconocimientos.
- **El endpoint** es el punto final del socket al que se pretende enviar el mensaje ...
- **data** son los datos asociados que se entregarán al socket. En el caso de los mensajes, se trata como texto sin formato, para otros eventos, se trata como JSON.

En el próximo capítulo, escribiremos una aplicación de chat en Socket.IO.

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

Ahora que estamos familiarizados con Socket.IO, escribamos una aplicación de chat, que podemos usar para conversar en diferentes salas de chat. Permitiremos que los usuarios elijan un nombre de usuario y les permitiremos chatear usándolo. Entonces, primero, configuremos nuestro archivo HTML para solicitar un nombre de usuario:

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
   </script>
   <body>
      <input type="text" name="name" value=""
placeholder="Enter your name!">
      <button type="button" name="button">Let me
chat!</button>
   </body>
</html>
```

Ahora que hemos configurado nuestro HTML para solicitar un nombre de usuario, creemos el servidor para aceptar conexiones del cliente. Permitiremos que las personas envíen sus nombres de usuario elegidos mediante el evento **setUsername** . Si existe un usuario, responderemos mediante un evento **userExists** , de lo contrario, **usaremos** un evento **userSet** .

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');});
users = [];
io.on('connection', function(socket){
   console.log('A user connected');
   socket.on('setUsername', function(data){
      if(users.indexOf(data) > -1){
         users.push(data);
         socket.emit('userSet', {username: data});
      } else {
         socket.emit('userExists', data + ' username is
taken! Try some other username.');
      }
   })
});
http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

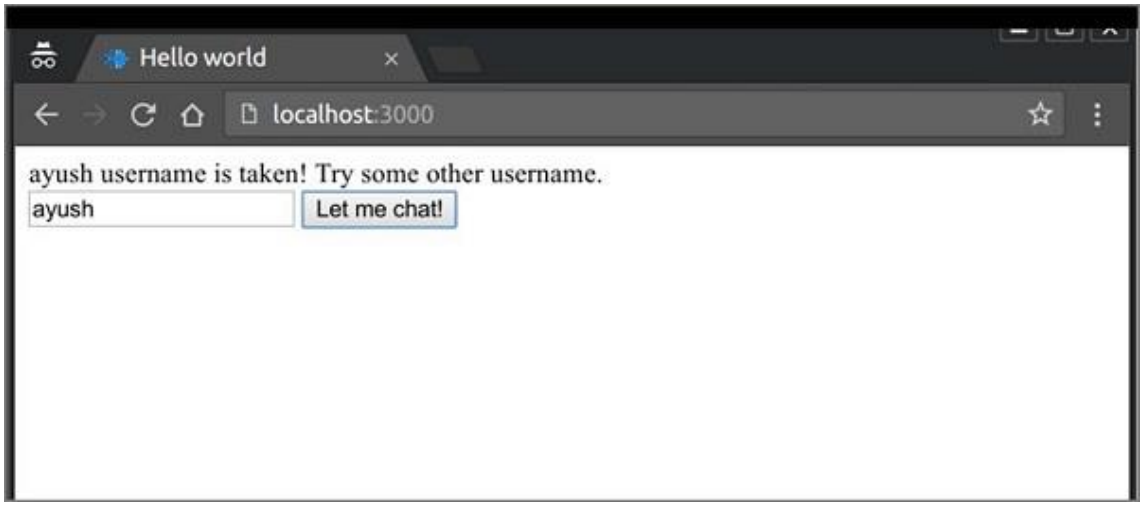
INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

Necesitamos enviar el nombre de usuario al servidor cuando la gente haga clic en el botón. Si el usuario existe, mostramos un mensaje de error; de lo contrario, mostramos una pantalla de mensajes -

```
<!DOCTYPE html>
<html>
   <head><title>Hello world</title></head>
   <script src="/socket.io/socket.io.js"></script>
   <script>
      var socket = io();
      function setUsername(){
         socket.emit('setUsername',
document.getElementById('name').value);
      };
      var user;
      socket.on('userExists', function(data){
         document.getElementById('error-container').innerHTML
= data;
      });
      socket.on('userSet', function(data){
         user = data.username;
         document.body.innerHTML = '<input type="text"
id="message">\
         <button type="button" name="button"
onclick="sendMessage()">Send</button>\
         <div id="message-container"></div>';
      });
      function sendMessage(){
         var msg = document.getElementById('message').value;
         if(msg){
            socket.emit('msg', {message: msg, user: user});
         }
      }
      socket.on('newmsg', function(data){
         if(user){
            document.getElementById('message-
container').innerHTML +='<div><b>' + data.user + '</b>: ' +
data.message + '</div>'
         }
      })
   </script>
   <body>
      <div id="error-container"></div>
      <input id="name" type="text" name="name" value=""
placeholder="Enter your name!">
      <button type="button" name="button"
onclick="setUsername()">Let me chat!</button>
      </body>
   </html>
```

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

Ahora, si conecta dos clientes con el mismo nombre de usuario, le dará un error como se muestra en la captura de pantalla a continuación:




Una vez que haya proporcionado un nombre de usuario aceptable, se le llevará a una pantalla con un cuadro de mensaje y un botón para enviar mensajes. Ahora, tenemos que manejar y dirigir los mensajes al cliente conectado. Para eso, modifique su archivo app.js para incluir los siguientes cambios:

```
var app = require('express')();
var http = require('http').Server(app);
var io = require('socket.io')(http);

app.get('/', function(req, res){
   res.sendFile('E:/test/index.html');});
users = [];
io.on('connection', function(socket){
   console.log('A user connected');
   socket.on('setUsername', function(data){
      console.log(data);
      if(users.indexOf(data) > -1){
         socket.emit('userExists', data + ' username is
taken! Try some other username.');
      } else {
         users.push(data);
         socket.emit('userSet', {username: data});
      }
   });
   socket.on('msg', function(data){
      //Send message to everyone
      io.sockets.emit('newmsg', data);
   })
});
http.listen(3000, function(){
   console.log('listening on localhost:3000');
});
```

INSTITUTO TÉCNICO DE ESTUDIOS PROFESIONALES

¡Ahora conecte cualquier número de clientes a su servidor, proporcione un nombre de usuario y comience a chatear! En el siguiente ejemplo, hemos conectado dos clientes con los nombres Ayush y Harshit y hemos enviado algunos mensajes de ambos clientes: