

# Práctica 4 - Node.js y Socket.io

David Jesús Ruiz de Valdivia Torres

## Parte 1 - Ejemplos

### Ejemplo 1 - helloworld.js

```
1  var http = require("http");
2  var httpServer = http.createServer(
3    function(request, response) {
4      console.log(request.headers);
5      response.writeHead(200, {"Content-Type": "text/plain"});
6      response.write("Hola mundo");
7      response.end();
8    }
9  );
10 httpServer.listen(8080);
11 console.log("Servicio HTTP iniciado");
```

En este ejemplo usamos el módulo http para crear un servidor, se imprime la cabecera de la petición por consola, se indica que la cabecera de la respuesta es texto plano y luego ya en el cuerpo añades el Hola mundo. Finalmente haces que el server escuche el puerto que quieras.

Resultado:



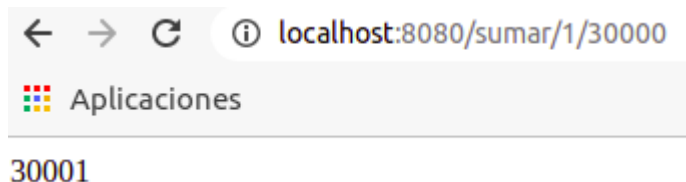
Hola mundo

## Ejemplo 2 - calculadora.js

```
1  var http = require("http");
2  var url = require("url");
3
4  function calcular(operacion, val1, val2) {
5      if (operacion=="sumar") return val1+val2;
6      else if (operacion == "restar") return val1-val2;
7      else if (operacion == "producto") return val1*val2;
8      else if (operacion == "dividir") return val1/val2;
9      else return "Error: Parámetros no válidos";
10 }
11
12 var httpServer = http.createServer(
13     function(request, response) {
14         var uri = url.parse(request.url).pathname;
15         var output = "";
16         while (uri.indexOf('/') == 0) uri = uri.slice(1);
17         var params = uri.split("/");
18         if (params.length >= 3) {
19             var val1 = parseFloat(params[1]);
20             var val2 = parseFloat(params[2]);
21             var result = calcular(params[0], val1, val2);
22             output = result.toString();
23         }
24         else output = "Error: El número de parámetros no es válido";
25
26         response.writeHead(200, {"Content-Type": "text/html"});
27         response.write(output);
28         response.end();
29     }
30 );
31 httpServer.listen(8080);
32 console.log("Servicio HTTP iniciado");
33
```

Se crea un server como antes y se procesa la url, se recoge la url a partir del puerto y se “parte en tres partes”, una será el primer operando, otro será el segundo y otro será la operación. Se pasan como parámetros a la función calcular la cual hace una operación dependiendo del tipo que le pases y se obtiene el resultado. Después se imprime como en el ejemplo anterior y se pone el server a escuchar en el puerto que quieras.

Resultado:



### Ejemplo 3 - calculadora-web.js y calculadora-web.html

```
var httpServer = http.createServer(  
  function(request, response) {  
    var uri = url.parse(request.url).pathname;  
    if (uri=="/") uri = "/calc.html";  
    var fname = path.join(process.cwd(), uri);  
    fs.exists(fname, function(exists) {  
      if (exists) {  
        fs.readFile(fname, function(err, data){  
          if (!err) {  
            var extension = path.extname(fname).split(".")[1];  
            var mimeType = mimeTypes[extension];  
            response.writeHead(200, mimeType);  
            response.write(data);  
            response.end();  
          }  
          else {  
            response.writeHead(200, {"Content-Type": "text/plain"});  
            response.write('Error de lectura en el fichero: '+uri);  
            response.end();  
          }  
        });  
      }  
    });  
  })  
);
```

Aquí básicamente se crea el server, obtenemos la ruta del archivo que queremos abrir (en este caso calc.html) y lo leemos. Si existe, vemos que extensión tiene y se la pasamos a la cabecera de la respuesta y al cuerpo le pasamos los datos del archivo.

```
    }  
    else{  
      while (uri.indexOf('/') == 0) uri = uri.slice(1);  
      var params = uri.split("/");  
      if (params.length >= 3) { //REST Request  
        console.log("Petición REST: "+uri);  
        var val1 = parseFloat(params[1]);  
        var val2 = parseFloat(params[2]);  
        var result = calcular(params[0], val1, val2);  
        response.writeHead(200, {"Content-Type": "text/html"});  
        response.write(result.toString());  
        response.end();  
      }  
      else {  
        console.log("Petición inválida: "+uri);  
        response.writeHead(200, {"Content-Type": "text/plain"});  
        response.write('404 Not Found\n');  
        response.end();  
      }  
    }  
  });  
};  
httpServer.listen(8080);  
console.log("Servicio HTTP iniciado");
```

El resto sería como en el ejemplo anterior.

```

<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Calculadora</title>
  </head>
  <body>
    <form action="javascript:void(0);" onsubmit="javascript:enviar();">
      Valor1: <input type="text" id="val1" /><br />
      Valor2: <input type="text" id="val2" /><br />
      Operación:
      <select id="operacion">
        <option value="sumar">Sumar</option>
        <option value="restar">Restar</option>
        <option value="producto">Producto</option>
        <option value="dividir">Dividir</option>
      </select><br />
      <input type="submit" value="Calcular" />
    </form>
    <span id="resul"></span>
  </body>
  <script type="text/javascript">
    var serviceURL = document.URL;
    function enviar() {
      var val1 = document.getElementById("val1").value;
      var val2 = document.getElementById("val2").value;
      var oper = document.getElementById("operacion").value;

      var url = serviceURL+"/"+oper+"/"+val1+"/"+val2;
      var httpRequest = new XMLHttpRequest();
      httpRequest.onreadystatechange = function() {
        if (httpRequest.readyState === 4){
          var resultado = document.getElementById("resul");
          resultado.innerHTML = httpRequest.responseText;
        }
      };
      httpRequest.open("GET", url, true);
      httpRequest.send(null);
    }
  </script>

```

En el HTML está el formulario que ve el usuario y que prepara los valores para que sean usados en el js como he explicado arriba.

Resultado:

← → ↻ ⓘ localhost:8080

Aplicaciones

Valor1: 250000

Valor2: 5

Operación: Dividir ▼

Calcular

50000

## Ejemplo 4 - connections.js y connections.html

```
var http = require("http");
var url = require("url");
var fs = require("fs");
var path = require("path");
var socketio = require("socket.io");
var mimeTypes = { "html": "text/html", "jpeg": "image/jpeg", "jpg": "image/jpeg", "png": "image/png", "js": "text/javascript", "css": "text/css" };

var httpServer = http.createServer(
  function(request, response) {
    var uri = url.parse(request.url).pathname;
    if (uri=="/") uri = "/connections.html";
    var fname = path.join(process.cwd(), uri);
    fs.exists(fname, function(exists) {
      if (exists) {
        fs.readFile(fname, function(err, data){
          if (!err) {
            var extension = path.extname(fname).split(".")[1];
            var mimeType = mimeTypes[extension];
            response.writeHead(200, mimeType);
            response.write(data);
            response.end();
          } else {
            response.writeHead(200, {"Content-Type": "text/plain"});
            response.write('Error de lectura en el fichero: '+uri);
            response.end();
          }
        });
      } else {
        console.log("Petición inválida: "+uri);
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write('404 Not Found\n');
        response.end();
      }
    });
  }
);
```

Esta primera parte carga el archivo html que vamos a usar tal y como hemos hecho en el ejemplo anterior.

```
httpServer.listen(8080);
var io = socketio(httpServer);

var allClients = new Array();
io.sockets.on('connection',
  function(client) {
    allClients.push({address:client.request.connection.remoteAddress, port:client.request.connection.remotePort});
    console.log('New connection from ' + client.request.connection.remoteAddress + ':' + client.request.connection.remotePort);
    io.sockets.emit('all-connections', allClients);
    client.on('output-evt', function(data) {
      client.emit('output-evt', 'Hola Cliente!');
    });
    client.on('disconnect', function() {
      console.log("El cliente "+client.request.connection.remoteAddress+" se va a desconectar");
      console.log(allClients);

      var index = -1;
      for(var i = 0; i<allClients.length;i++){
        //console.log("Hay "+allClients[i].port);
        if(allClients[i].address == client.request.connection.remoteAddress
          && allClients[i].port == client.request.connection.remotePort){
          index = i;
        }
      }

      if (index != -1) {
        allClients.splice(index, 1);
        io.sockets.emit('all-connections', allClients);
      } else {
        console.log("EL USUARIO NO SE HA ENCONTRADO!")
      }
      console.log('El usuario '+client.request.connection.remoteAddress+' se ha desconectado');
    });
  }
);

console.log("Servicio Socket.io iniciado");
```

En esta segunda parte se está indicando que cada vez que se conecte un cliente, se añade su información a allClients y se imprime un mensaje en la terminal del servidor avisando de esto. Se le envían a todos los clientes este vector y también se envía un saludo a cada uno.

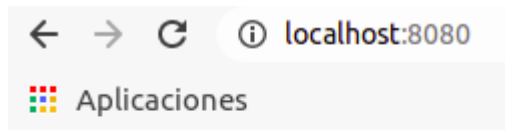
Cuando un cliente se desconecta, lo eliminamos del array de clientes y le notificamos con el nuevo array al resto de clientes.

```
<title>Connections</title>
</head>
<body>
  <span id="mensaje_servicio"></span>
  <div id="lista_usuarios"></div>
</body>
<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
  function mostrar_mensaje(msg){
    var span_msg = document.getElementById('mensaje_servicio');
    span_msg.innerHTML = msg;
  }

  function actualizarLista(usuarios){
    var listContainer = document.getElementById('lista_usuarios');
    listContainer.innerHTML = '';
    var listElement = document.createElement('ul');
    listContainer.appendChild(listElement);
    var num = usuarios.length;
    for(var i=0; i<num; i++) {
      var listItem = document.createElement('li');
      listItem.innerHTML = usuarios[i].address+": "+usuarios[i].port;
      listElement.appendChild(listItem);
    }
  }

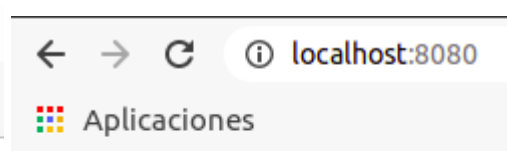
  var serviceURL = document.URL;
  var socket = io.connect(serviceURL);
  socket.on('connect', function(){
    socket.emit('output-evt', 'Hola Servicio!');
  });
  socket.on('output-evt', function(data) {
    mostrar_mensaje('Mensaje de servicio: '+data);
  });
  socket.on('all-connections', function(data) {
    actualizarLista(data);
  });
  socket.on('disconnect', function() {
    mostrar_mensaje('El servicio ha dejado de funcionar!!');
  });
</script>
```

Por parte del HTML, contamos con un método que imprime mensajes en la página y otro que actualiza la lista de usuarios. Después como los clientes están suscritos a varios eventos, como cuando se conectan, que saludan al servicio, 'output-evt' que imprime en la página un mensaje que le envíe el servidor. Un evento 'all-connections' al que están suscritos todos que sirve para recibir la lista de usuarios actualizada y por último el de desconexión del servidor, que muestra un mensaje en la página avisando de que el servidor deja de estar en marcha.



Mensaje de servicio: Hola Cliente!

- ::1:43812
- ::1:43814



El servicio ha dejado de funcionar!!

- ::1:43812

```

David@david-HP-Pavilion-Notebook:~/Documentos/DSB/P4/ejemplos$ nodejs connection
s.js
Servicio Socket.io iniciado
Petición invalida: /favicon.ico
New connection from ::1:43812
Petición invalida: /favicon.ico
New connection from ::1:43814
El cliente ::1 se va a desconectar
[ { address: '::1', port: 43812 },
  { address: '::1', port: 43814 } ]
El usuario ::1 se ha desconectado

```

## Ejemplo 5 - mongo-test.js y mongo-test.html

La primera parte del js es como en los ejemplos anteriores, se carga el html que vayamos a usar. Pasamos directamente a la segunda parte.

```

MongoClient.connect("mongodb://localhost:27017/", { useUnifiedTopology: true }, function(err, db) {
  httpServer.listen(8080);
  var io = socketio(httpServer);

  var dbo = db.db("pruebaBaseDatos");
  dbo.createCollection("test", function(err, collection){
    io.sockets.on('connection', function(client) {

      client.emit('my-address', {host:client.request.connection.remoteAddress, port:client.request.connection.remotePort});
      client.on('poner', function (data) {
        collection.insertOne(data, {safe:true}, function(err, result) {});
      });
      client.on('obtener', function (data) {
        collection.find(data).toArray(function(err, results){
          client.emit('obtener', results);
        });
      });
    });
  });
});
console.log("Servicio MongoDB iniciado");

```

En esta segunda parte lo primero que se hace es crear la conexión con Mongo y usamos como base de datos “pruebaBaseDatos”, creamos una colección llamada test y ponemos el servidor a escuchar la conexión de los cliente para que, cada vez que se conecte uno, el servidor emita el evento ‘my-address’ que recibirá ese cliente y este responderá con el evento ‘poner’ el cual inserta en la colección creada los datos del cliente. Después tendrá que llegar otro evento del cliente ‘obtener’ mediante el cual se buscan sus datos en la colección para enviárselos al cliente (este luego actualizará su lista con esta información).

En esta segunda parte (captura abajo) encontramos que los clientes van a estar escuchando a 3 eventos, “my-address”, “obtener” y “disconnect”.

- En el de “my-address” (ya explicado arriba) se envía información acerca del cliente y su tiempo de conexión al servidor y luego envía un evento “obtener” para más tarde recibir esos datos y poder actualizar su lista.
- En el de “obtener” se llama a la función actualizarLista en la cual se toma una referencia del HTML de la lista y se añade el contenido del vector con información de los clientes recibido del servidor y se añade como elemento de una lista en HTML
- En “disconnect” se actualiza la lista pero no se envían parámetros para quitar al usuario que se ha desconectado.

```

<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
  <title>MongoDB Test</title>
</head>
<body>
  <div id="resultados"></div>
</body>
<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
  function actualizarLista(usuarios){
    var listContainer = document.getElementById('resultados');
    listContainer.innerHTML = '';
    var listElement = document.createElement('ul');
    listContainer.appendChild(listElement);
    var num = usuarios.length;
    for(var i=0; i<num; i++) {
      var listItem = document.createElement('li');
      listItem.innerHTML = JSON.stringify(usuarios[i]);
      listElement.appendChild(listItem);
    }
  }

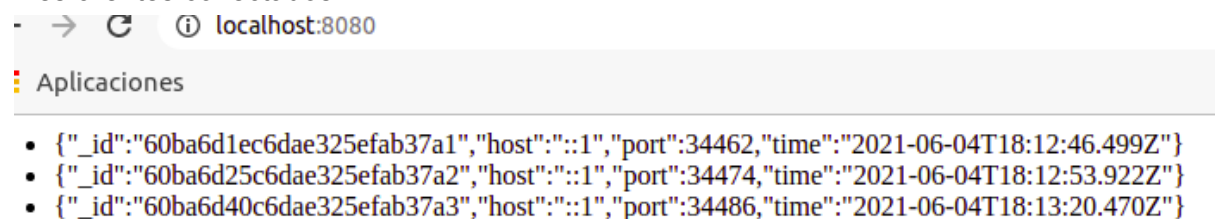
  var serviceURL = document.URL;
  var socket = io.connect(serviceURL);

  socket.on('my-address', function(data) {
    var d = new Date();
    socket.emit('poner', {host:data.host, port:data.port, time:d});
    socket.emit('obtener', {host: data.address});
  });
  socket.on('obtener', function(data) {
    actualizarLista(data);
  });
  socket.on('disconnect', function() {
    actualizarLista({});
  });
</script>
</html>

```

Resultado:

Tres clientes conectados



localhost:8080

Aplicaciones

- {"\_id":"60ba6d1ec6dae325efab37a1","host":"::1","port":34462,"time":"2021-06-04T18:12:46.499Z"}
- {"\_id":"60ba6d25c6dae325efab37a2","host":"::1","port":34474,"time":"2021-06-04T18:12:53.922Z"}
- {"\_id":"60ba6d40c6dae325efab37a3","host":"::1","port":34486,"time":"2021-06-04T18:13:20.470Z"}

## Parte 2 - Ejercicio simulación domótica

La técnica a usar en la comunicación de los nodos y los usuarios ha sido la actualización de datos. Todas las partes del programa avisaban al resto de partes mediante eventos cuando surgía algún cambio. Los métodos de la simulación son los siguientes:



### **Cliente:**

- actualizarHistorial(msg): Obtiene los mensajes que ya hay en el historial de eventos y le añade msg.
- comprobacionAgente(): Esta función se ejecuta cada vez que se ingresan nuevos valores. El agente comprueba que los nuevos datos estén dentro de los umbrales definidos. Y si alguno se sale, se envía un evento “alerta” con el mensaje de alerta a enviar que será añadido en el historial, además también cierra las persianas si se cumplen las condiciones enunciadas para ello (o las abre en caso contrario).
- apagarAire()/encenderAire(): Crea un array donde indica el mensaje de la acción a cometer, el estado actual del aire y un identificador de la incidencia para luego enviarlo al servidor.
- cerrarPersianas()/abrirPersianas(): Método casi idéntico al par de arriba mencionado pero referente a las persianas.
- enviarMedidas(): Obtiene los valores introducidos por el usuario, llama a comprobacionAgente() y los envía al servidor.

El cliente además, escucha los siguiente eventos:

- ‘all-connections’: Este evento actualiza el estado del aire o las persianas y escribe en el historial el mensaje de acuerdo a ello.
- ‘emision-medidas’: Recibe las nuevas medidas, crea un mensaje con ello y actualiza el historial.
- ‘emision-alerta’: Actualiza el historial con el mensaje que le llegue (le llegarán alertas).
- ‘disconnect’: Cuando se pierda conexión con el servidor se actualizará el historial con un mensaje al respecto.
- 

### **Servidor:**

El servidor lo primero que hace es crear una base de datos “domotica” y una colección “test” y luego empieza a escuchar los siguientes eventos:

- ‘nuevos-datos’: Recibe datos del método ya mencionado “enviarMedidas()” del cliente, obtiene la hora exacta y mete estos datos en la colección. Después emite el evento ‘emision-medidas’ a los clientes con las nuevas medidas ya mencionadas.
- ‘estado-aire’ y ‘estado-persianas’ reciben el estado del aire o de las persianas de parte de un cliente, lo imprime por terminal y lo difunde al resto.
- ‘alerta’: Recibe una alerta de un cliente, la imprime por pantalla y la difunde al resto.

**Capturas del funcionamiento más abajo:**  
**Interfaz de usuario.**

**Domótica**

**Sensor aire acondicionado**

Máximo: 40

Mínimo: 5

Temperatura:

Encender aire acondicionado

Apagar aire acondicionado

**Sensor luminosidad**

Máximo: 40

Mínimo: 5

Luminosidad:

Abrir persianas

Cerrar persianas

Enviar datos

**Con los botones podemos interactuar con los actuadores y vemos como se difunde la información.**

**Domótica**

**Sensor aire acondicionado**

Máximo: 40

Mínimo: 5

Temperatura:

Encender aire acondicionado

Apagar aire acondicionado

**Sensor luminosidad**

Máximo: 40

Mínimo: 5

Luminosidad:

Abrir persianas

Cerrar persianas

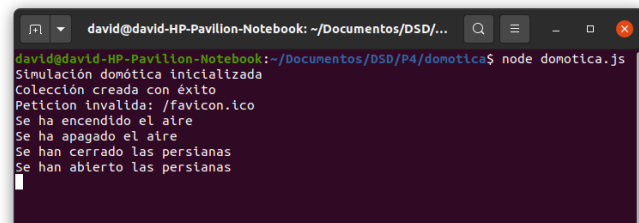
Enviar datos

Se ha encendido el aire

Se ha apagado el aire

Se han cerrado las persianas

Se han abierto las persianas



**Los cambios de valores también se guardan y hasta cierran y abren las persianas.**

**Domótica**

**Sensor aire acondicionado**

Máximo: 40

Mínimo: 5

Temperatura:

Encender aire acondicionado

Apagar aire acondicionado

**Sensor luminosidad**

Máximo: 40

Mínimo: 5

Luminosidad:

Abrir persianas

Cerrar persianas

Enviar datos

Se ha encendido el aire

Se ha apagado el aire

Se han cerrado las persianas

Se han abierto las persianas

El valor de la temperatura esta fuera del umbral

El nuevo valor de la temperatura es 50 y el de la luminosidad es 20

El valor de la temperatura esta fuera del umbral

El valor de la temperatura esta fuera del umbral

Se han cerrado las persianas

El nuevo valor de la temperatura es 50 y el de la luminosidad es 60

