

Práctica 4 DSD

Nombre: Ana López Mohedano Grupo: A2

Ejemplos

Hello world

En este ejemplo, realizamos un servidor HTTP básico utilizando el módulo http que viene integrado en Node.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");
```

Primero de todo, importamos http de Node.js para poder crear y manejar solicitudes HTTP, que es la primera línea de código.

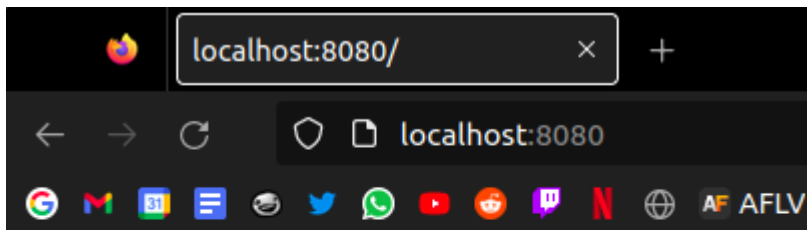
La siguiente parte, la de httpServer, crea el servidor HTTP utilizando el método createServer() que toma como argumento una función que se ejecutará cada vez que se reciba una solicitud HTTP. En este caso, recibe los parámetros request y response, que contienen información sobre la solicitud recibida y la respuesta que se enviará al cliente, respectivamente.

Dentro de la función, se imprime en la consola los headers de la solicitud recibida utilizando console.log(). A continuación, se establece el código de estado en 200 que quiere decir que todo ha ido correctamente y se especifica el tipo de contenido en "text/plain". Después, se escribe en la respuesta 'Hola mundo' y se finaliza la respuesta con response.end().

La línea 10 (httpServer.listen(8080);) inicia el servidor HTTP en el puerto 8080. La última línea imprime por consola el mensaje para saber que se ha iniciado correctamente el servicio HTTP.

Ejecución:

```
ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$ node helloworld.js
Servicio HTTP iniciado
```



Hola mundo

En la terminal:

```
Servicio HTTP iniciado
{
  host: 'localhost:8080',
  'user-agent': 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/113.0',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8',
  'accept-language': 'es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3',
  'accept-encoding': 'gzip, deflate, br',
  connection: 'keep-alive',
  cookie: 'pma_lang=es',
  'upgrade-insecure-requests': '1',
  'sec-fetch-dest': 'document',
  'sec-fetch-mode': 'navigate',
  'sec-fetch-site': 'none',
  'sec-fetch-user': '?1'
}
```

Calculadora

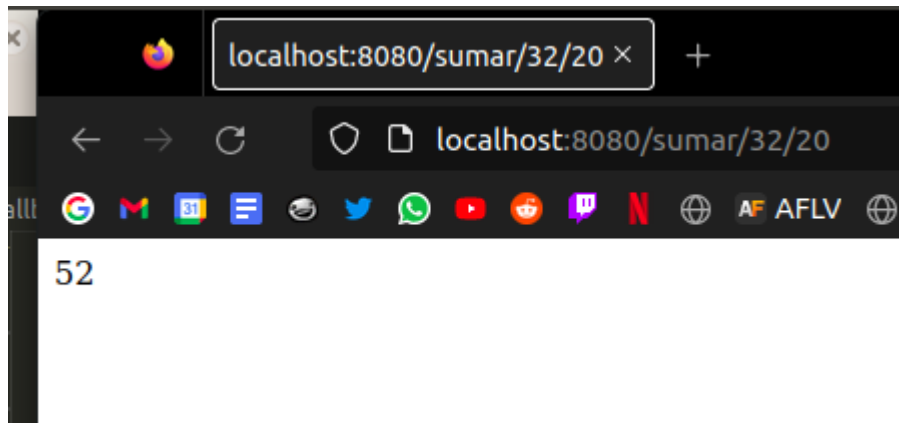
```
1  var http = require("http");
2  var url = require("url");
3
4  function calcular(operacion, val1, val2) {
5      if (operacion=="sumar") return val1+val2;
6      else if (operacion == "restar") return val1-val2;
7      else if (operacion == "producto") return val1*val2;
8      else if (operacion == "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");
```

En este ejemplo tenemos una estructura similar, primeramente importando el módulo http de Node.js, y el módulo 'url' para analizar y manipular URL, permite descomponer una URL en sus diferentes componentes, como el protocolo, el nombre de host, el puerto, la ruta y los parámetros de consulta. Este servidor recibe peticiones HTTP y procesa esa petición para realizar una operación matemática simple. Se define una función 'calcular' que toma una operación (en total 4: sumar, restar, producto o dividir) y dos valores numéricos, y devuelve el resultado de la operación. La petición HTTP es analizada utilizando el método 'url.parse' para obtener la ruta solicitada y la operación matemática a realizar. Luego, se verifica el número de parámetros y se ejecuta la operación utilizando "calcular".

El resultado de la operación se convierte en una cadena de texto y se envía de vuelta a través de la respuesta HTTP. En este caso, el tipo de contenido es 'text/html'.

Ejecución:

```
o ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$ node calculadora.js
Servicio HTTP iniciado
```



Aquí introducimos la dirección localhost:8080 (está escuchando en el puerto 8080), seguido de '/' y la operación que deseamos realizar. Después, escribimos otra '/', el primer operando, '/' y el segundo operando. En este caso, queremos sumar 32 y 20, que efectivamente es 52.

Calculadora-web

Este ejemplo consiste en una calculadora usando también un servidor HTTP que maneje solicitudes RESTful.

La primera línea, como en el primer ejemplo, importa el módulo http de Node.js, que proporciona una API para crear servidores HTTP. Luego se importan los módulos 'url', 'fs' y 'path', que se utilizan para analizar y manipular las rutas de las solicitudes HTTP, leer archivos del sistema de archivos y trabajar con rutas de archivos.

Después se define un objeto 'mimeTypes' que asocia extensiones de archivo con tipos MIME. Este objeto se utiliza más adelante para establecer correctamente los encabezados HTTP al enviar archivos al navegador.

Luego se define una función 'calcular' que toma una operación (en total 4: sumar, restar, producto o dividir) y dos valores numéricos, y devuelve el resultado de la operación. Si se proporcionan valores no numéricos, se devuelve un mensaje de error.

A continuación, se crea un servidor HTTP utilizando el método 'http.createServer()'. Este método toma una función de devolución de llamada que se ejecuta cada vez que se recibe una solicitud HTTP en el servidor. Esta función recibe dos argumentos: 'request' y 'response'. La función de devolución de llamada maneja tanto las solicitudes de archivos estáticos como las solicitudes RESTful.

Primero, la función analiza la ruta de la solicitud HTTP utilizando el módulo 'url'. Si la ruta es la raíz del servidor ("/"), se redirige a "calc.html", que es la página web que contiene la calculadora. A continuación, se crea una ruta completa a partir de la ruta relativa utilizando el método path.join().

Si el archivo existe en el sistema de archivos, se lee y se envía al cliente utilizando el método `fs.readFile()`. Se establece el tipo MIME del archivo utilizando el objeto `mimeTypes` y se escribe en el objeto de respuesta `response`. Si hay un error al leer el archivo, se envía un mensaje de error al cliente.

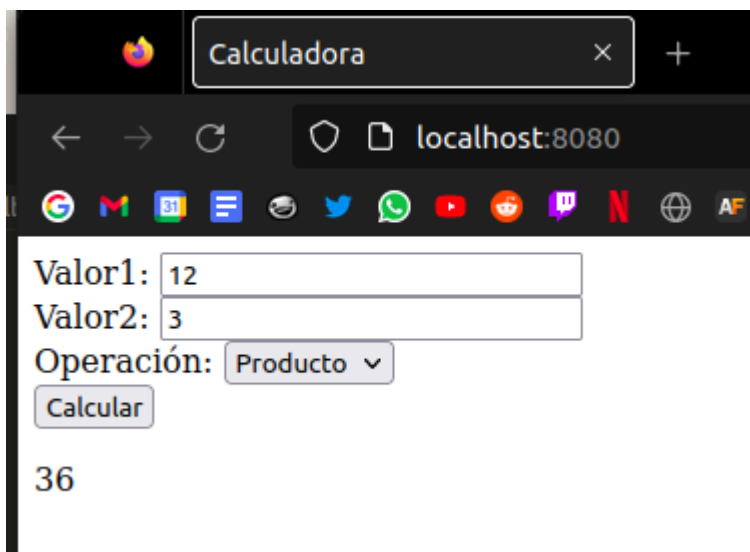
En cambio, si el archivo no existe, se analiza la solicitud como una solicitud RESTful. La ruta de la solicitud debe tener la forma `/operacion/valor1/valor2`, donde “operacion” es una de las operaciones definidas en la función “calcular”, y `valor1` y `valor2` son valores numéricos. Si la solicitud es una solicitud RESTful válida, se calcula el resultado de la operación utilizando la función “calcular” y se envía al cliente como una respuesta HTML.

Si la solicitud no es ni una solicitud de archivo estático ni una solicitud RESTful válida, se envía un mensaje de error “404 Not Found” al cliente.

Finalmente, el servidor HTTP se pone en escucha en el puerto 8080 y se muestra un mensaje en la consola indicando que el servidor HTTP ha sido iniciado.

Ejecución:

```
ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$ node calculadora-web.js
Servicio HTTP iniciado
```



Calculadora

localhost:8080

Valor1: 12

Valor2: 3

Operación: Producto

Calcular

36

Introducimos los valores por texto, seleccionamos la operación que queramos y le damos a calcular, y efectivamente, en este caso, nos hace el producto de 12 y 3, que es 36.

Connections (socket)

En este ejemplo creamos un servidor que utiliza el módulo `http` de Node.js para crear un servidor web en el puerto 8080. También utilizará el módulo `url`, el módulo `fs` (para

interactuar con el sistema de archivos del servidor) y el módulo `socket.io` para habilitar la comunicación bidireccional en tiempo real entre el servidor y el cliente.

La función de creación del servidor HTTP `http.createServer()` maneja las solicitudes de los clientes y responde en consecuencia. Cuando un cliente hace una solicitud, se extrae la ruta de acceso (`pathname`) de la URL de la solicitud utilizando el módulo `url`.

Luego se utiliza el módulo `path` para unir la ruta de acceso con el directorio de trabajo actual (`process.cwd()`) y obtener la ruta completa del archivo solicitado (`fname`).

El servidor verifica si el archivo solicitado existe utilizando el método `fs.exists()`. Si el archivo existe, se lee y se escribe en la respuesta de la solicitud. El tipo MIME del archivo se determina utilizando el objeto `mimeTypes` y se establece en el encabezado de la respuesta.

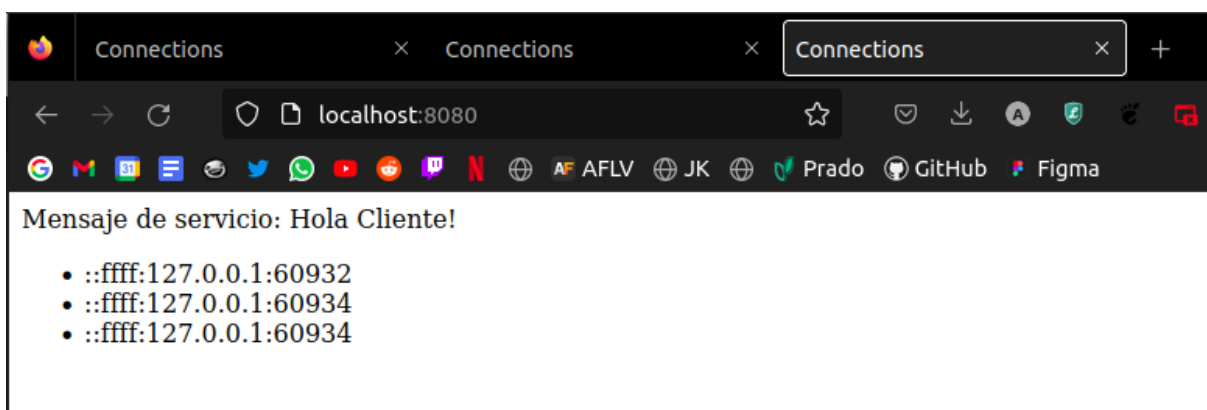
Si el archivo no existe, se escribe un mensaje de error 404 en la respuesta. En ambos casos la respuesta se cierra utilizando el método `response.end()`.

En la siguiente parte del código se utiliza la biblioteca `Socket.io` para manejar conexiones en tiempo real entre clientes y servidor. La función `socketio(httpServer)` crea un objeto de `Socket.IO` que se asocia con el servidor HTTP previamente creado.

Se crea un array llamado `allClients` que almacena los detalles de cada conexión que se realiza con el servidor. Cuando un cliente se conecta, se emite un evento de conexión que agrega el objeto `client` al array `allClients` y envía una lista de todos los clientes conectados a través del evento `'all-connections'` a todos los clientes conectados actualmente.

También hay dos manejadores de eventos para el objeto `client`. El primero es para el evento `'output-evt'`, que simplemente envía una respuesta al cliente conectado. El segundo es para el evento `'disconnect'`, que se activa cuando el cliente se desconecta. Cuando un cliente se desconecta, se busca el objeto `client` en el array `allClients` para eliminarlo. Luego, se emite un evento `'all-connections'` a todos los clientes conectados para actualizar la lista de conexiones.

Ejecución:



Aquí abrimos 3 conexiones y se muestran a continuación en la página, junto con el mensaje de "Hola Cliente!".

Anotación:

En mi caso para ejecutar los ejemplos uso el comando `$node [nombre_archivo]`, pero antes tengo que configurar con `nvm` la versión de node, ya que me da un error con las librerías.

```
ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$ node connections.js
node: /lib/x86_64-linux-gnu/libc.so.6: version `GLIBC_2.28' not found (required by node)
ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$
```

Por lo tanto, instalo la versión 16.15.1 con `$npm install 16.15.1` y ya se usa esa versión para nodejs. Si se me cambia al cambiar de terminal, uso `$nvm use 16.15.1` y funciona correctamente.

```
ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$ nvm use 16.15.1
Now using node v16.15.1 (npm v8.11.0)
ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$ node connections.js
Servicio Socket.io iniciado
```

MongoDB

En esta ocasión creamos un servidor web que utiliza Node.js y MongoDB para almacenar y recuperar datos de una base de datos. Primero importaremos los módulos necesarios, que como en ejemplos anteriores, serán `http`, `url`, `fs`, `path`, y `socket.io`.

```
1 var http = require("http");
2 var url = require("url");
3 var fs = require("fs");
4 var path = require("path");
5 var socketio = require("socket.io");
```

Se importan también los módulos necesarios para conectarse a

MongoDB: `MongoClient` y `MongoServer`. También se define un objeto `'mimeTypes'` que mapea las extensiones de archivos a los tipo MIME correspondientes.

```

11 var httpServer = http.createServer(
12   function(request, response) {
13     var uri = url.parse(request.url).pathname;
14     if (uri=="/") uri = "/mongo-test.html";
15     var fname = path.join(process.cwd(), uri);
16     fs.exists(fname, function(exists) {
17       if (exists) {
18         fs.readFile(fname, function(err, data){
19           if (!err) {
20             var extension = path.extname(fname).split(".")[1];
21             var mimeType = mimeTypes[extension];
22             response.writeHead(200, mimeType);
23             response.write(data);
24             response.end();
25           }
26         } else {
27           response.writeHead(200, {"Content-Type": "text/plain"});
28           response.write('Error de lectura en el fichero: '+uri);
29           response.end();
30         }
31       });
32     } else {
33       console.log("Petición inválida: "+uri);
34       response.writeHead(200, {"Content-Type": "text/plain"});
35       response.write('404 Not Found\n');
36       response.end();
37     }
38   });
39 }
40
41 );

```

Se crea el servidor HTTP utilizando la función 'createServer' del módulo 'http'. La función lee el archivo solicitado y lo envía al cliente como respuesta HTTP. Si el archivo no existe, se envía una respuesta 404.

```

44 MongoClient.connect("mongodb://localhost:27017/", { useUnifiedTopology: true }, function(err, db) {
45   httpServer.listen(8080);
46   var io = socketio(httpServer);
47
48   var dbo = db.db("pruebaBaseDatos");
49   dbo.createCollection("test", function(err, collection){
50     io.sockets.on('connection',
51       function(client) {
52         client.emit('my-address', {host:client.request.connection.remoteAddress, port:client.request.connection.remotePort});
53         client.on('poner', function (data) {
54           collection.insert(data, {safe:true}, function(err, result) {});
55         });
56         client.on('obtener', function (data) {
57           collection.find(data).toArray(function(err, results){
58             client.emit('obtener', results);
59           });
60         });
61       });
62   });
63 });
64
65 console.log("Servicio MongoDB iniciado");

```

Aquí se conecta a la base de datos de MongoDB utilizando la función 'connect' del módulo 'MongoClient'. Se especifica la URL de la base de datos, que en este caso es 'mongodb://localhost:27017/'. También se especifica la opción 'useUnifiedTopology' para utilizar la topología de replicación unificada de MongoDB.

Una vez conectado a la base de datos, se crea un objeto 'dbo' que representa la base de datos 'pruebaBaseDatos' y se crea una colección llamada 'test' utilizando la función 'createCollection'.

Luego se crea un servidor de sockets utilizando la función 'socketio' del módulo 'socketio'. Se especifica que el servidor de sockets debe escuchar en el mismo puerto que el servidor HTTP creado anteriormente (var io = socketio(httpServer)).

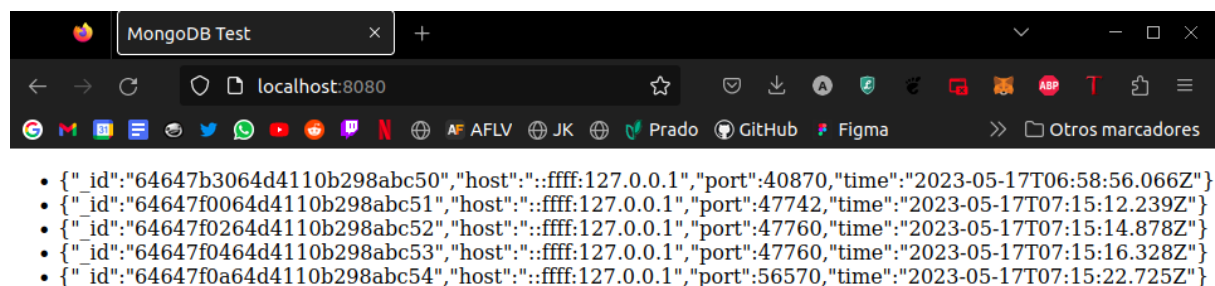
Después se define una función de callback que se ejecutará cada vez que se conecte un cliente al servidor de sockets. Esta función de callback emite un evento 'my-address' al cliente para que el cliente conozca su dirección IP y puerto.
./home/ana/UNI/2022_2023/DSD/Practicas_DSD/Practica4_DSD/calc.html

La función de callback también define dos manejadores de eventos para los eventos 'poner' y 'obtener'. Cuando el cliente emite el evento 'poner', se inserta un nuevo documento en la colección 'test'. Cuando el cliente emite el evento 'obtener', se recuperan todos los documentos de la colección 'test' y se emiten al cliente mediante el evento 'todos'.

Por último, se muestra un mensaje en la consola indicando que el servidor ha iniciado correctamente.

Ejecución:

```
ana@ana:~/UNI/2022_2023/DSD/Practicas_DSD/Documentos/P4/ejemplos$ node mongo-test.js
Servicio MongoDB iniciado
```



En este caso, he recargado unas 4 veces, y se van almacenando las veces que accedo al servidor, mostrando los datos del cliente que se conecta.

Para acceder a los datos guardados, se accede a través de los siguientes comandos:

```
$ mongo
```

```
$ show dbs;
```

```
$ use pruebaBaseDatos; // en nuestro caso se llama pruebaBaseDatos
```

```
$ show collections; // nos saldrá test
```

```
$ db.test.find(); // mostrará los datos guardados en
```

```
test/home/ana/UNI/2022_2023/DSD/Practicas_DSD/Practica4_DSD/calc.html/home/ana/UNI/
/home/ana/UNI/2022_2023/DSD/Practicas_DSD/Practica4_DSD/calc.html/2022_2023/DSD/
Practicas_DSD/Practica4_DSD/calc.html
```

Ejercicio

Nuestro ejercicio se trata de un sistema domótico básico compuesto de dos sensores (luminosidad y temperatura), dos actuadores (motor persiana y sistema de Aire/Acondicionado), un servidor que sirve páginas para mostrar el estado y actuar sobre los elementos de la vivienda. Además dicho servidor incluye un agente capaz de notificar alarmas y tomar decisiones básicas.

Se compone de dos sensores (uno para el aire acondicionado y otro para la luminosidad para las persianas) que difunden información acerca de las medidas tomadas a través del servidor. Esta medidas se proporcionarán mediante un formulario de entrada que proporcionará el servidor para poder incluir las medidas de ambos sensores, también mostrando los mínimos y máximos para poder cambiarlos. Al introducir una nueva medida en el formulario, se publicará el correspondiente evento que incluirá dicha medida. La página mostrará los cambios en el estado de los actuadores, a modo de un histórico de eventos.

Ese histórico de eventos se guardará en una base de datos con la correspondiente marca de tiempo asociada a cada evento.

Cada usuario accederá al estado del sistema a través del servidor mostrando la información en la correspondiente página que este enviará. El usuario podrá abrir o cerrar la persiana, y/o encender y apagar el sistema de aire acondicionado en cualquier momento.

También tendremos un agente que detectará cuando las medidas sobrepasan los umbrales máximos y mínimos y publicará la correspondiente alarma, provocando el cierre de la persiana.

Tendremos dos archivos: servidor.js y servidor.html.

A continuación se explica el código para ejecutar esta tarea:

Servidor.js

```
1 // módulos http, url, fs, path y socketio
2 var http = require("http");
3 var url = require("url");
4 var fs = require("fs");
5 var path = require("path");
6 var socketio = require("socket.io");
7
8 function getTimestamp() {
9   return new Date().toISOString();
10 }
11
12
13 // módulos mongodb y mimeTypes
14 var MongoClient = require('mongodb').MongoClient;
15 var MongoServer = require('mongodb').Server;
16 var mimeTypes = { "html": "text/html", "jpeg": "image/jpeg", "jpg": "image/jpeg", "png": "image/png", "js": "text/javascript", "css": "text/css", "swf": "application/x-shockwave-flash" };
17
18 // Server http
19 var httpServer = http.createServer(
20   function(request, response) {
21     //
```

Al principio, como en los ejemplos, se añadirán los módulos http, url, fs, path y socket.io. Luego se añaden los módulos de mongo para poder usar la base de datos.

Creamos el servidor:

```

18 // Server http
19 var httpServer = http.createServer(
20     function(request, response) {
21         ///
22         var uri = url.parse(request.url).pathname;
23         if (uri=="/") uri = "/servidor.html";
24         ///
25         var fname = path.join(process.cwd(), uri);
26         fs.exists(fname, function(exists) {
27             if (exists) {
28                 fs.readFile(fname, function(err, data){
29                     if (!err) {
30                         var extension = path.extname(fname).split(".")[1];
31                         var mimeType = mimeTypes[extension];
32                         response.writeHead(200, mimeType);
33                         response.write(data);
34                         response.end();
35                     }
36                     else {
37                         response.writeHead(200, {"Content-Type": "text/plain"});
38                         response.write('Error de lectura en el fichero: '+uri);
39                         response.end();
40                     }
41                 });
42             }
43         });
44     }
45 );
46

```

Y luego hacemos la conexión con la base de datos:

```

48 // MongoDB
49 MongoClient.connect("mongodb://localhost:27017/", { useUnifiedTopology: true }, function(err, db) {
50     if(err)
51         throw err;
52
53     httpServer.listen(8081);
54     var io = socketio(httpServer);
55
56     // la base de datos se llamará domotica
57     var dbo = db.db("domotica");
58
59
60     // Creamos las colecciones de persianas y aire acondicionado
61     dbo.createCollection("persianas", function(err, collection){
62         if(!err)
63             console.log("Coleccion creada en mongo: " + collection.collectionName);
64     });
65
66     dbo.createCollection("aire_acondicionado", function(err, collection){
67         if(!err)
68             console.log("Coleccion creada en mongo: " + collection.collectionName);
69     });
70

```

Primero lanzaremos el servidor, en mi caso en el puerto 8081 porque me da error usando el 8080, saltando un aviso de que ya está en uso. Se puede cambiar según convenga para que pueda funcionar correctamente.

Primeramente, lanzamos el socket y usamos la base de datos llamada “domotica”. Crearemos las colecciones de persianas y aire_acondicionado, para almacenar los cambios y estados de cada uno en sus propias colecciones.

```
72 io.sockets.on('connection',function(socket) {
73   console.log("Conectado al socket");
74
75   // Inserción de colección aire_acondicionado //
76   // Insertamos los valores
77   socket.on('enviar-datos-aire', function (datos) {
78     dbo.collection("aire_acondicionado").insert({valor:datos[0], minimo:datos[1], maximo:datos[2]}, {safe:true},
79     function(err, result) {
80       if (!err){
81         console.log("Hemos insertado en A/C: {valor:" + datos[0] + ", minimo:" + datos[1] + ", maximo:" + datos[2] + "}");
82         io.sockets.emit('Registro', getTimestamp() + " - Modificación de A/C: " + datos[0] + " grados");
83       }
84       else
85         console.log("Error insertando datos en aire acondicionado.");
86     });
87   });
88
89   // Introducimos el estado (activar el aire acondicionado)
90   socket.on('activar_ac', function (datos) {
91     dbo.collection("aire_acondicionado").insert({estado:datos}, {safe:true},
92     function(err, result) {
93       if (!err){
94         console.log("Insertado estado nuevo de A/C: {estado:" + datos + "}");
95         io.sockets.emit('Registro', getTimestamp() + " - Aire acondicionado " + datos); // aire acondicionado apagado/encendido
96       }
97       else
98         console.log("Error insertando datos en aire acondicionado.");
99     });
100   });
101 }
```

Seguidamente, establecemos la conexión con el socket, mostrando un mensaje por consola de “Conectado al socket” para confirmar que se ha realizado correctamente la conexión.

Esta función se encarga de recibir datos de aire acondicionado a través de un evento de socket llamado “enviar-datos-aire”, que se encarga de insertar el valor de la temperatura, el mínimo y el máximo, cuyos valores se han enviado desde una función enviar, cogiendo los datos del formulario del html. Se emite un evento llamado “Registro” a todos los sockets conectados con un mensaje con la marca de tiempo (getTimestamp()) y la información modificada del aire acondicionado.

A continuación tenemos otra función parecida que se llama cuando se activa o se desactiva el aire acondicionado, y que registra el nuevo estado del aire acondicionado (encendido o apagado) en la base de datos. Emite un evento llamado “Registro” a todos los sockets conectados con la marca de tiempo y el estado nuevo del aire acondicionado.

```

103 // Inserción de colección Persianas //
104 // Insertamos los valores
105 socket.on('enviar-datos-persianas', function (datos) {
106     dbo.collection("persianas").insert({valor:datos[0], minimo:datos[1], maximo:datos[2]}, {safe:true},
107     function(err, result) {
108         if (!err){
109             console.log("Insertado en Persianas: {valor:" + datos[0] + ", minimo:" + datos[1] + ", maximo:" + datos[2] + "}");
110             io.sockets.emit('Registro', getTimestamp() + " - Modificación de luminosidad: " + datos[0]);
111         }
112         else
113             console.log("Error insertando datos en persianas.");
114     });
115 });
116
117 // Introducimos el estado (activar persiana - abrir persiana)
118 socket.on('activar_pers', function (datos) {
119     dbo.collection("persianas").insert({estado:datos}, {safe:true},
120     function(err, result) {
121         if (!err){
122             console.log("Insertado estado nuevo de las persianas: {estado:" + datos + "}");
123             io.sockets.emit('Registro', getTimestamp() + " - Persiana " + datos);
124         }
125         else
126             console.log("Error insertando datos en persianas.");
127     });
128 });

```

En este caso haremos estas dos últimas funciones pero aplicadas a las persianas, teniendo una función para insertar el valor nuevo de la luminosidad, el mínimo y el máximo, emitiendo también un evento “Registro” donde informa de que la luminosidad ha sido modificada, y otra función para cuando se suben o se bajan las persianas, almacenando el nuevo estado de las persianas en la base de datos y emitiendo un evento “Registro” con el nuevo estado de las persianas (abierta o cerrada).

Por último, tenemos dos eventos: un evento “aviso” que emitirá un evento para todos los sockets conectados con un mensaje de texto que avisa de que se han sobrepasado algún o algunos umbrales mínimos o máximos y que por ende se cierran las persianas automáticamente; y un evento “aviso_ambos”, que hará lo mismo pero con un mensaje de texto de que ambos sensores han sobrepasado los umbrales y que por ende se cierran las persianas.

```

130 // Aviso de que se pasa de los umbrales y se cierran las persianas
131 socket.on ('aviso', function (sensor) {
132     var alarma = "Aviso: " + sensor + " fuera de los umbrales, cerrando persianas";
133     console.log (alarma);
134     io.sockets.emit ('aviso', alarma);
135 });
136
137 // Aviso de que se pasa de los umbrales y se cierran las persianas (en caso de que los dos sensores sobrepasen)
138 socket.on ('aviso_ambos', function () {
139     var alarma = "Aviso: los sensores fuera de los umbrales, cerrando persianas";
140     console.log (alarma);
141     io.sockets.emit ('aviso', alarma);
142 })
143
144 });

```

Servidor.html

En el caso del html, mostramos los formularios y el registro de los eventos (el log).

```

15 <!-- Aire acondicionado -->
16 <div class="actuador">
17   <form class="formulario" id="aire">
18     <h2> Aire acondicionado </h2>
19     <div class="elemento centrado">
20       <label for="valor">Grados: </label>
21       <input name="valor" type="number" value="23" min="5" max="35"/>
22     </div>
23     <div class="elemento centrado">
24       <label for="min">Mínimo: </label>
25       <input name="min" type="number" value="18" min="5" max="35"/>
26     </div>
27     <div class="elemento centrado">
28       <label for="max">Máximo: </label>
29       <input name="max" type="number" value="30" min="5" max="35"/>
30     </div>
31     <div class="main">
32       <input class="centrado" type="submit" name="submit" value="Actualizar temperatura"
33         onclick="enviar('aire');return false;"/>
34       <label class="switch centrado">
35         <input type="checkbox" id="activar_ac" onclick="encender('activar_ac');">
36         <span class="slider round"></span>
37       </label>
38     </div>
39   </form>
40 </div>

```

Primeramente tenemos el actuador del aire acondicionado, que tendrá un formulario con id “aire” para luego poder acceder a sus datos a través del id, y que tiene tres elementos a introducir: los grados actuales, el mínimo y el máximo. El mínimo y el máximo también se podrá modificar en la propia página, con valores entre 5 y 35. Luego tendrá un botón que actualizará la temperatura, llamando a la función enviar, la cual enviará los datos al agente para comprobar si se pasa de los umbrales o no, y en caso afirmativo, enviar un aviso a todos los sockets, cerrar las persianas, y almacenar el nuevo estado de la persiana en la base de datos (“Cerrada”). Escribimos “return false;” para que al pulsarlo no recargue la página y evita que el formulario se envíe de forma predeterminada. Al final tenemos un label de clase switch centrado, que será un interruptor de encendido/apagado, y un campo de entrada “checkbox” que representa ese interruptor. El atributo “id” se utiliza para identificar este campo en el código de javascript y luego poder actuar según si se cambia este valor de temperatura o el valor de luminosidad (que tendrá otro checkbox en su propio actuador). El atributo onclick llamará a la función encender cuando se hace clic en el interruptor. Por último, un elemento que se utiliza para crear el diseño visual del interruptor.

```

42 <!-- Persianas -->
43 <div class="actuador">
44   <form class="formulario" id="persianas">
45     <h2> Persianas </h2>
46     <div class="elemento centrado">
47       <label for="valor">Luminosidad: </label>
48       <input name="valor" type="number" value="5" min="0" max="12"/>
49     </div>
50     <div class="elemento centrado">
51       <label for="min">Mínimo: </label>
52       <input name="min" type="number" value="3" min="0" max="12"/>
53     </div>
54     <div class="elemento centrado">
55       <label for="max">Máximo: </label>
56       <input name="max" type="number" value="7" min="0" max="12"/>
57     </div>
58     <div class="main">
59       <input class="centrado" type="submit" name="submit"
60         value="Actualizar luminosidad" onclick="enviar('persianas');return false;"/>
61       <label class="switch centrado">
62         <input type="checkbox" id="activar_pers" onclick="encender('activar_pers');">
63         <span class="slider round"></span>
64       </label>
65     </div>
66   </form>
67 </div>
68
69 </div>

```

Aquí tenemos el otro actuador para las persianas, que tiene una estructura igual, con un valor de luminosidad, uno de mínimo y otro de máximo, que puede oscilar entre 0 y 12, y que por defecto el mínimo es 3 y el máximo es 7. El resto tiene un funcionamiento igual al anterior actuador.

```

70
71 <div class="centrado log-class actuador">
72   <h1>Histórico de eventos del servidor</h1>
73   <div id="log"></div>
74 </div>
75

```

Aquí tenemos el log (el histórico de eventos) que se irá expandiendo según ocurran eventos.

```

78 <script src="/socket.io/socket.io.js"></script>
79 <script type="text/javascript">
80
81     var serviceURL = document.URL;
82     var socket = io.connect(serviceURL);
83
84
85     // Funcion para enviar los datos al agente y al socket
86     function enviar(id) {
87         // id = aire ó persianas
88         var formulario = document.getElementById(id);
89         var datos = new Array();
90
91         // Valor (grados o luminosidad)
92         datos.push(formulario[0].value);
93         // Mínimo de los valores
94         datos.push(formulario[1].value);
95         // Máximo de los valores
96         datos.push(formulario[2].value);
97
98         // Envío de información al Agente
99         if (id == "aire")
100             sensor = "temperatura";
101         else
102             sensor = "luminosidad";
103
104         agente (sensor, datos); // envío al agente de qué sensor y qué datos
105
106         // Envío de información al servidor
107         socket.emit('enviar-datos-' + id, datos);
108     }
109

```

Tenemos la función enviar, que se encargará de recoger los datos del formulario (que identificará según el id, que en nuestro caso puede ser “aire” o “persianas”) y los meterá en un array “datos”, que luego se emitirá un evento, que según el actuador (“id”), será “enviar-datos-aire” o “enviar-datos-persianas”. Se envían los datos recopilados del formulario como argumento del evento, enviando los datos al servidor a través del socket. Antes de eso enviaremos la información al agente, junto con el sensor (“aire” o “persianas”) y los datos del formulario. Ese agente se encargará de comprobar si los datos sobrepasan los umbrales, lo describiremos más adelante.


```

111 function encender(que_activar){
112     var activador = document.getElementById(que_activar);
113     this.estado;
114
115     if (que_activar=="activar_ac") {
116         if(activador.checked == true){
117             this.estado = "Encendido";
118         }
119         else{
120             this.estado = "Apagado";
121         }
122         socket.emit(que_activar, estado);
123     }
124     else { // activar_pers
125         if(activador.checked == true){
126             this.estado = "Abierta";
127         }
128         else{
129             this.estado = "Cerrada";
130         }
131         socket.emit(que_activar, this.estado);
132     }
133 }
134

```

Aquí tenemos la función encender, que es la que se encarga de activar o desactivar un actuador específico, que se le pasa como argumento "que_activar". En la primera línea obtenemos una referencia al elemento con el id "que_activar", que puede ser "activar_ac" o "activar_pers". Luego comprueba si está marcado como activado o no, y según ese valor, le concede el contrario a la variable this.estado (si está apagado, se enciende, y si está encendido, se apaga). Esta variable la usaremos principalmente en el agente para que cuando haya que cerrar la persiana al pasar los umbrales, comprobemos primero si está abierta para cerrarla, y si ya está cerrada no hacer nada. Luego, se emite un evento a través del socket con el nombre "activar_pers" o "activar_ac" y se pasa el valor de "this.estado" como argumento, para registrarlo en el log y meter el nuevo estado en la base de datos.

```

137 // Actualiza el registro de eventos del servicio
138 function actualizarLog (mensaje, importante) {
139     var div = document.getElementById("log");
140     if (importante==true)
141         var evento = "<p style='color: red;'> " + mensaje + " </p>"
142     else
143         var evento = "<p> " + mensaje + " </p>"
144     div.innerHTML += evento;
145
146     // Actualizar scroll
147     div.scrollTop = div.scrollHeight;
148 }

```

Esta función actualiza el registro de los eventos del servicio, recibiendo un mensaje y una variable booleana que indicará si es importante o no (es importante cuando sobrepasa los umbrales). Si es importante, se añadirá un párrafo al div correspondiente del html con la fuente en color rojo, y si no es importante, se añadirá tal cual, del color predeterminado (negro). La última línea de código actualiza la posición de desplazamiento vertical del elemento "div" para asegurarse de que siempre esté visible el último mensaje agregado. Establecer scrollTop en scrollHeight hace que el elemento se desplace hasta el final, mostrando el contenido más reciente.

```
150 // Recepción de eventos del servidor
151 // registrar cambios
152 socket.on ('Registro', function (mensaje) {
153     actualizarLog (mensaje);
154 });
155 // registrar aviso (sobrepaso de umbrales)
156 socket.on ('aviso', function (mensaje) {
157     actualizarLog (mensaje, true);
158 });
159
```

Aquí tenemos la recepción de eventos del servidor, que serán 'Registro' y 'aviso'. Cuando recibe el evento 'Registro', llama a la función actualizarLog que acabamos de describir y muestra el evento en pantalla. Cuando recibe el evento 'aviso', llama a la función actualizarLog y muestra el texto de aviso (que será el sobrepaso de los umbrales máximos y/o mínimos). En este caso se marca a true la variable importante, por lo que este mensaje se imprimirá en rojo.

Agente

```
161 function agente (sensor, datos) {
162     // cojo los valores de aire y persianas
163     var temp = document.getElementById("aire");
164     var luz = document.getElementById("persianas");
165
166     /// Variables:
167     // temp[0] es el valor de la temperatura
168     // temp[1] el mínimo de temperatura
169     // temp[2] el máximo de temperatura
170
171     // luz[0] es el valor de la luminosidad
172     // luz[1] el mínimo de luminosidad
173     // luz[2] el máximo de luminosidad
174
175     // ambos sobrepasan los máximos
176     if ( (+temp[0].value > +temp[2].value) && (+luz[0].value > +luz[2].value) ){ // max
177         socket.emit ("aviso_ambos"); // ambos sensores
178         if(this.estado == "Abierta"){
179             document.getElementById("activar_pers").checked = false;
180             encender ("activar_pers"); // cerrar persiana
181         }
182     }
183     // ambos sobrepasan los mínimos
184     else if ( (+temp[0].value < +temp[1].value) && (+luz[0].value < +luz[1].value) ){ //min
185         socket.emit ("aviso_ambos"); // ambos sensores
186         if(this.estado == "Abierta"){
187             document.getElementById("activar_pers").checked = false;
188             encender ("activar_pers"); // cerrar persiana
189         }
190     }
191     // el cambio actual ha sobrepasado el mínimo o máximo
192     else if ( (+datos[0] < +datos[1]) || (+datos[0] > +datos[2]) ) { // cambio actual
193         socket.emit ("aviso", sensor); // avisar del sensor que sobrepasa
194         if(this.estado == "Abierta"){
195             document.getElementById("activar_pers").checked = false;
196             encender ("activar_pers"); // cerrar persiana
197         }
198     }
199     // no sobrepasa ningún límite, no hace nada
200     else {}
201 }
202
203
```

Esta función (que se encuentra dentro de servidor.html) recibe dos argumentos, 'sensor' y 'datos'. La variable 'sensor' hará referencia al sensor que ha cambiado y que habrá que comprobar si ese nuevo cambio sobrepasa los umbrales. La variable 'datos' serán los datos del formulario del actuador. Tenemos las variables locales 'temp' y 'luz' que obtendrán la referencia de los elementos con id "aire" y "persianas" (los valores de la temperatura y luminosidad, con sus máximos y mínimos).

Luego se hace una comprobación con una estructura if-else, que comprueba si ambos valores sobrepasan los mínimos, si ambos sobrepasan los máximos o si el último cambio (los 'datos' que ha recibido) ha sobrepasado el mínimo o máximo. En cualquiera de esos casos, se procede a cerrar la persiana en caso de que esté abierta, comprobándolo con la variable "this.estado". Si está abierta, se llama a la función 'encender' con el argumento de "activar-pers", que cerrará las persianas, y desmarcando el checkbox; en caso de estar cerrada no se hace nada. Independientemente de si está abierta o cerrada la persiana, al

detectar si sobrepasa los umbrales, se emitirá un evento 'aviso' (o 'aviso_ambos', si se trata de los dos sensores) a todos los sockets, avisando de que los sensores han sobrepasado los umbrales, y que se cerrarán las persianas.
En caso de que no se detecte que no sobrepasa ningún umbral, no se hace nada.

Así se muestra la página del servicio domótico:

Aire acondicionado

Grados:

Mínimo:

Máximo:

☒

Persianas

Luminosidad:

Mínimo:

Máximo:

☐

Histórico de eventos del servidor

Aviso: los sensores fuera de los umbrales, cerrando persianas

2023-05-25T09:11:38.546Z - Modificación de A/C: 35 grados

2023-05-25T09:11:40.881Z - Persiana Abierta

Aviso: los sensores fuera de los umbrales, cerrando persianas

2023-05-25T09:11:41.957Z - Persiana Cerrada

2023-05-25T09:11:41.958Z - Modificación de luminosidad: 12

Podemos observar los dos actuadores a ambos lados, el de aire acondicionado y el de persianas, y abajo un histórico de los eventos del servidor que va añadiendo mensajes según ocurran los eventos. Podemos editar los valores con las flechas que se muestran al lado de los valores o introduciéndolos por teclado, y luego pulsando los botones de actualizar para enviar el cambio y que el programa haga todo el proceso explicado anteriormente (se llama a "enviar", y "enviar" llama al "agente").