

Universidad de Granada

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y
DE TELECOMUNICACIÓN

PRÁCTICA 4

NODE.JS

Desarrollo de Sistemas Distribuidos

Autor:
Raquel Molina Reche

Junio 2021

Índice

I	Implementación de los ejemplos	1
1.	EJEMPLO 1: helloworld	2
2.	EJEMPLO 2: Calculadora REST	3
3.	EJEMPLO 3: Calculadora REST interfaz	4
4.	Ejemplo 4: Socket.io	6
5.	Ejemplo 5: Uso de MongoDB desde Node.js	9
II	Sistema domótico	12
6.	Planteamiento de la solución	12
7.	Sensores y umbrales máximos y mínimos	13
8.	Actuadores	13
9.	Eventos controlados por el agente:	13
9.1.	Eventos que solo producen alarmas	13
9.2.	Eventos que producen alarmas y cambios de estado	14
10.	Ejemplos de ejecución	14
10.1.	Ejemplo	15

Introducción

Los contenidos de esta práctica comprenden el manejo de mecanismos para la implementación de servicios web mediante la plataforma Node.js haciendo uso del lenguaje de programación JavaScript. Node.js trabaja los servicios con un modelo de comunicación asíncrono y dirigido por eventos.

También se incluye la utilización de diferentes módulos de Node.js:

- Socket.io: implementación de aplicaciones basadas en el protocolo publish-subscribe.
- MongoDB: para el almacenamiento de información.

Esta práctica consta de dos partes: una para la prueba de ejemplos e iniciación a Node.js, Socket.io y MongoDB, y una segunda parte que consiste en la implementación de un sistema domótico compuesto por sensores y actuadores.

Parte I

Implementación de los ejemplos

1. EJEMPLO 1: helloworld

Este es un ejemplo muy básico de helloworld.

Para ello se importa un módulo http ya que se va hacer un servicio web, y una vez importado se llama a la función para crear un servidor.

Para crear el servidor se requiere de una función que será la que se ejecutará cada vez que el cliente realice una petición, esta función requiere dos parámetros: request y response que son la petición y la respuesta que llega. En la cabecera de la respuesta se escribe que el contenido será texto plano y después se escribe el cuerpo “Hola mundo” para finalizar la respuesta.

Probar su funcionamiento:

1. Ejecutando desde el terminal: `$nodejs helloworld.js`
2. Una vez ejecutado el servicio, desde el navegador comprobar si la dirección “`http://localhost:8080/`” muestra el mensaje “HolaMundo”.

```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS/
ejemplos/ejemplo1$ ls
helloworld.js
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS/
ejemplos/ejemplo1$ nodejs helloworld.js
Servicio HTTP iniciado
|
```

Figura 1:

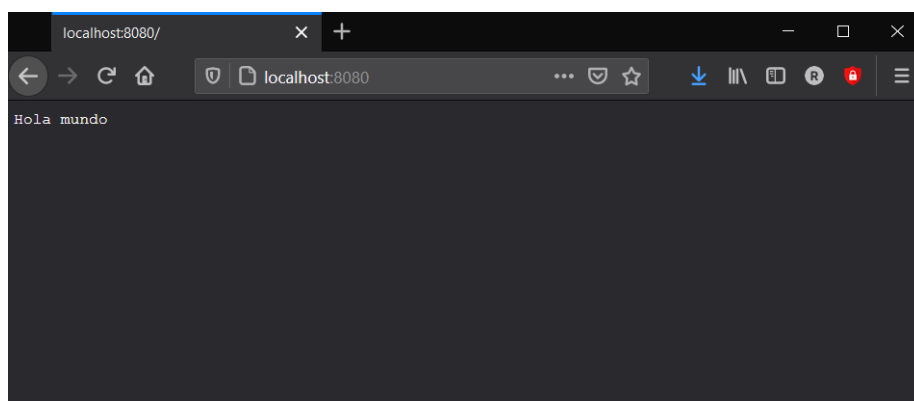


Figura 2:

Además después de comprobar como en la consola aparece la cabecera de la petición pues se hacía un “console.log(request.headers)”:

```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS/
ejemplos/ejemplo1$ nodejs helloworld.js
Servicio HTTP iniciado
{ host: 'localhost:8080',
  'user-agent':
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:88.0) Gecko/20100101 Firefox
/88.0',
  accept:
    'text/html,application/xhtml+xml,application/xml;q=0.9,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',
  connection: 'keep-alive',
  cookie: 'Webstorm-62eba692=7ce3d377-6d61-49b6-bcf9-fb5e4f00f9fe',
  'upgrade-insecure-requests': '1' }
{ host: 'localhost:8080',
  'user-agent':
    'Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:88.0) Gecko/20100101 Firefox
/88.0',
  accept: 'image/webp,*/*',
  'accept-language': 'es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3',
  'accept-encoding': 'gzip, deflate',
  connection: 'keep-alive',
  referer: 'http://localhost:8080/',
  cookie: 'Webstorm-62eba692=7ce3d377-6d61-49b6-bcf9-fb5e4f00f9fe' }
```

Figura 3:

2. EJEMPLO 2: Calculadora REST

Este ejemplo consiste en la realización de una calculadora en base a peticiones orientadas a recursos. Para ello además del módulo http se requiere el módulo url ya que las peticiones a la calculadora se van a manejar mediante la dirección.

Para el cálculo de las operaciones se realiza mediante la función calcular que recibe por parámetros la operación y dos operandos y devuelve el resultado de la misma.

Se crea el servidor con la función que se ejecutará cada vez que el cliente realice una petición, con los parámetros: request y response que son la petición y la respuesta que llega.

En la función se maneja la url dividiendo y adquiriendo la operación y los operandos que requiere el cliente y que se encuentran a partir de la definición del puerto en la dirección. Mediante los métodos slice y split sobre cadenas de caracteres, por ejemplo si la dirección es “http://localhost:8080/sumar/2/3”: la dirección se empieza a procesar desde “/sumar/2/3”, se elimina la primera barra y se separan a partir de las barras obteniendo 3 cadenas, la operación requerida (sumar) y los operandos (2 y 3) que requieren ser convertidos a datos de tipo float. Una vez desglosada la petición se llama a la función calcular y se obtiene el resultado, que para mostrarlo por pantalla se convierte a tipo String.

Si durante el procesamiento se produce algún error se muestra, en caso contrario una vez obtenido el resultado se envía la respuesta como texto plano y como salida el resultado del cálculo.

Probar su funcionamiento:

1. Ejecutando desde el terminal: `$nodejs calculadora.js`
2. Una vez ejecutado el servicio, desde el navegador solicitar la operación en la dirección siguiendo el siguiente patrón: `http://localhost:8080/operacion/operando1/operando2` y observar el resultado mostrado.

```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS/
ejemplos/ejemplo2$ ls
calculadora.js
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS/
ejemplos/ejemplo2$ nodejs calculadora.js
Servicio HTTP iniciado
|
```

Figura 4:

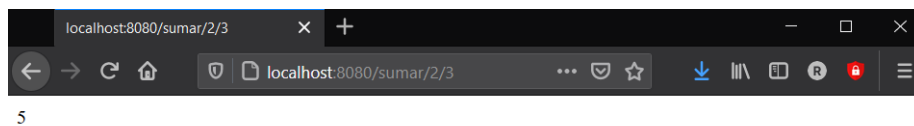


Figura 5:

3. EJEMPLO 3: Calculadora REST interfaz

Este ejemplo consiste en la implementación de una calculadora distribuida usando una interfaz tipo REST.

En el código fuente del servicio `calculadora-web.js` se maneja la posibilidad de que se realice el cálculo mediante la petición por la dirección url, que se realiza de la misma manera que en el ejemplo anterior, y además añade la posibilidad de que se maneje con una interfaz mediante el archivo `calc.html`.

Para ello a la versión del ejemplo anterior se agrega el uso del módulo `fs` para realizar operaciones de entrada/salida sobre el sistema de ficheros en el servidor que se usa para leer la petición y los datos y devolver la respuesta. Y además se agrega el uso del módulo `path` para obtener la

ruta del fichero calc.html que contiene la interfaz para el cliente.

En este caso en el cliente se maneja la petición con un formulario del que el envío se realiza a partir de la función JavaScript enviar() que obtiene los operandos introducidos por el usuario de los inputs y la operación a partir del select que contiene una lista con las operaciones disponibles.

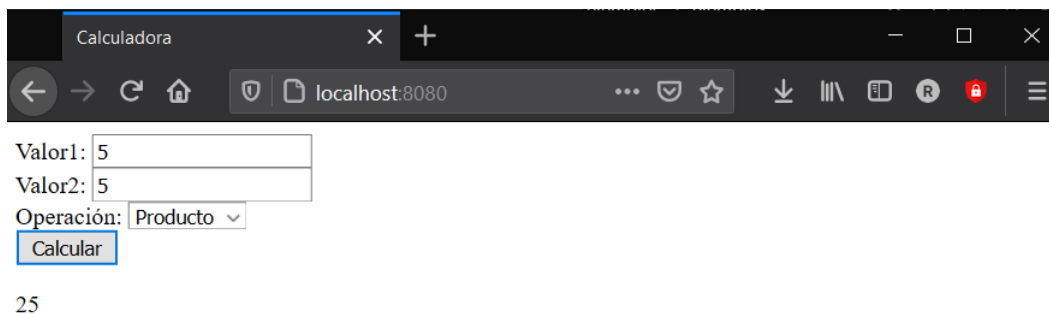
Una vez se obtiene la petición y los datos se crea una petición REST construyendo la url que requiere el servicio y generando la petición asíncrona http correspondiente que es de tipo GET y que cuando el servicio devuelva el resultado incluirá el valor de este en el espacio span para el resultado.

Probar su funcionamiento:

1. Ejecutando desde el terminal: `$nodejs calculadora-web.js`
2. Una vez ejecutado el servicio, desde el navegador `http://localhost:8080` y completar el formulario obteniendo la respuesta del servicio.

```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS
/ejemplos/ejemplo3$ node calculadora-web.js
Servicio HTTP iniciado
|
```

Figura 6:



Calculadora

Valor1: 5

Valor2: 5

Operación: Producto

Calcular

25

Figura 7:

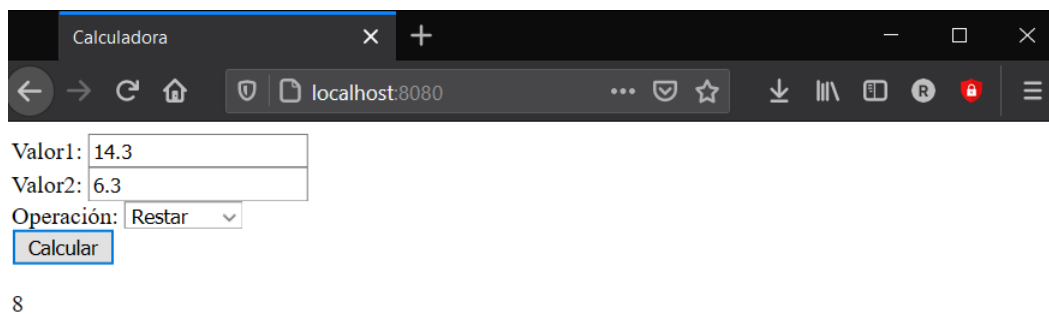


Figura 8:

Además en la consola aparece un registro de las peticiones REST que se han realizado:

```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS
/ejemplos/ejemplo3$ node calculadora-web.js
Servicio HTTP iniciado
Petición REST: producto/5/5
Petición REST: restar/14.3/6.3
|
```

Figura 9:

4. Ejemplo 4: Socket.io

Este ejemplo consiste en la implementación de un servicio que envía notificaciones que contienen las direcciones de los clientes que se encuentran conectados al servicio, de manera que cuando un cliente se conecta este queda suscrito y se notifica a todos los demás usuarios suscritos del nuevo cliente e igualmente cuando un usuario se desconecta. Para ello se hace uso de socket.io que es un módulo de node.js que permite enviar notificaciones a los clientes de un servicio.

En el código fuente del servicio `connections.js` a parte de los módulos de los ejemplos anteriores se añade el módulo `socket.io`. La creación del servidor se realiza de la misma manera que en el ejemplo anterior a diferencia de que el fichero `html` se llama `connections.html`. Después de llamar al servidor para que escuche en el puerto 8080, se maneja la parte de `socketio` conectando este módulo con el servidor esperando a que se produzcan eventos que se administran mediante la función `on('connection, ...)`, es decir a partir de la conexión de un nuevo cliente:

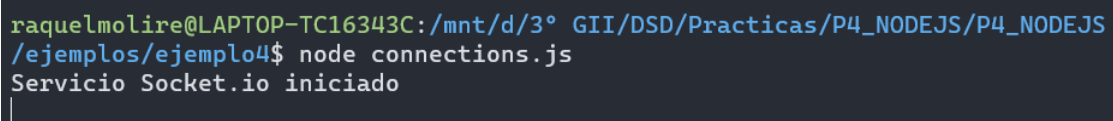
1. Se incluye el nuevo cliente en el array de clientes.
2. Se realiza un `emit('all-connections', ...)` de manera que se envía a todos los clientes que están conectados en el servicio la información del nuevo cliente.

3. Se suscribe el servicio al evento output-evt del cliente para que cuando el cliente emita al servicio una petición a través de este evento se le notifique.
4. Si algún cliente se desconecta: Se elimina el cliente del array de clientes conectados y se emite una notificación al resto de clientes de la desconexión de ese cliente mediante el array actualizado.

En el caso del cliente se emplea un fichero html, este tiene un espacio span para mostrar el mensaje de servicio y un espacio div que contendrá la lista de usuarios conectados. A partir de JavaScript se completará el espacio span y div con la información correspondiente que llegue por parte del servicio mediante la función mostrar_mensaje y actualizarLista . A partir de la URL se conecta al servicio, emitiendo por el evento output-evt el mensaje 'Hola Servicio!' al servicio. Cuando el cliente reciba un evento output-evt del servicio, éste modificará el contenido del span con el mensaje de servicio y cuando el cliente reciba un evento all-connections modificará el contenido del div con el nuevo array de clientes, pues así se ha implementado en el servicio. Cuando el cliente reciba un evento 'disconnect' mostrará el mensaje 'El servicio ha dejado de funcionar!!' y en el momento en el que el servicio vuelva a estar disponible de nuevo se conectará a él y se repiten los pasos anteriores.

Probar su funcionamiento:

1. Ejecutando desde el terminal: `$nodejs connections.js`
Por motivos de instalación y/o sistema operativo, para poder realizar la ejecución de forma correcta se ha ejecutado previamente la siguiente línea:
`$npm i --save socket.io`
2. Una vez ejecutado el servicio, conectar y desconectar diferentes clientes a partir del navegador con `http://localhost:8080`.



```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3º GII/DSD/Practicas/P4_NODEJS/P4_NODEJS
/ejemplos/ejemplo4$ node connections.js
Servicio Socket.io iniciado
|
```

Figura 10: Se ejecuta el servicio

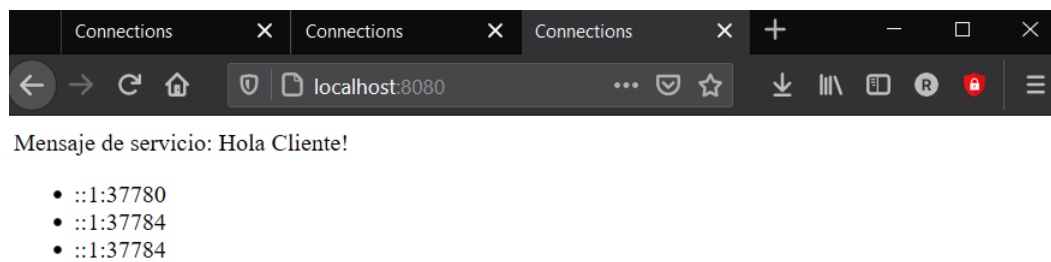


Figura 11: Se conectan 3 clientes

En la consola el servicio lleva un historial de eventos: los usuarios que se conectan y los que se desconectan. En este caso uno de los usuarios anteriores se ha desconectado:

```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS
/ejemplos/ejemplo4$ node connections.js
Servicio Socket.io iniciado
Petición invalida: /favicon.ico
New connection from ::1:37780
New connection from ::1:37784
New connection from ::1:37784
El cliente ::1 se va a desconectar
[ { address: '::1', port: 37780 },
  { address: '::1', port: 37784 },
  { address: '::1', port: 37784 } ]
El usuario ::1 se ha desconectado
|
```

Figura 12:

También se actualiza la lista de usuarios conectados en el resto de clientes:

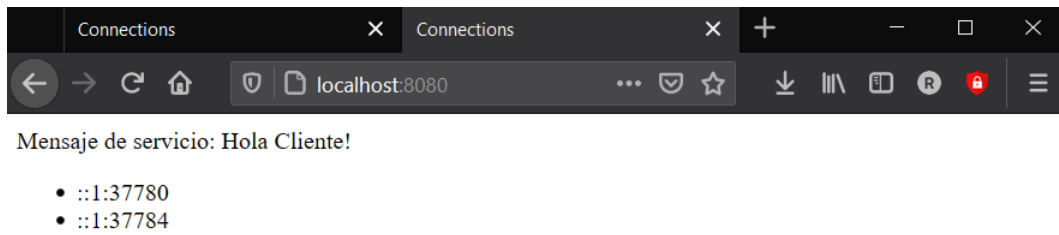


Figura 13:

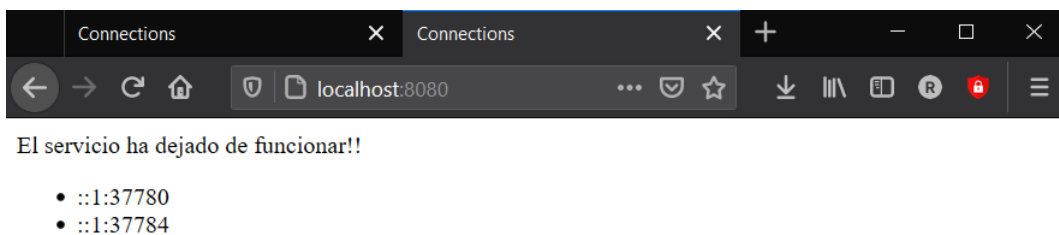


Figura 14:

5. Ejemplo 5: Uso de MongoDB desde Node.js

Este ejemplo es similar al anterior pero añadiendo el uso de MongoDB para almacenar información de los clientes y obtenerla según las peticiones que los clientes realicen al servicio.

En el código fuente del servicio `mongo-test.js` a parte de los módulos del ejemplo anterior se añade el módulo `mongodb` y se obtienen a partir de este módulo el cliente y el servidor de `mongodb`. La creación del servidor se realiza de la misma manera que en el ejemplo anterior a diferencia de que el fichero `html` se llama `mongo-test.html`.

Para la nueva funcionalidad el cliente de mongodb se conecta a la base de datos indicando dirección IP y puerto y de la misma que por defecto es localhost y 27017 respectivamente. Se llama al servidor para que escuche en el puerto 8080 y se maneja la parte de socketio conectándolo al servidor.

A partir de este momento se manejan los eventos y la base de datos, para ello se inicia la base de datos "pruebaBaseDatos" se crea una colección "test" que será con la que se trabajará. Una vez creada la colección se manejan los eventos de manera que cuando se conecte un cliente se envía a partir del evento "my-address" se envía la información del cliente (su dirección y puerto por el que se conecta). En el evento "poner" se inserta en la colección los datos que devuelva el cliente y en el evento "obtener" se busca en la colección la petición del cliente y se devuelve el resultado mediante un array.

En el caso del cliente se emplea un fichero html, este tiene un espacio div que contendrá el resultado de la petición al servicio. A partir de JavaScript se completará el espacio div con la información correspondiente que llegue por parte del servicio mediante la función actualizarLista. A partir de la URL se conecta al servicio a través de socketio, y se escucha el evento "miaddress" que recibe del servicio la dirección y el puerto desde donde se conecta y envía la información correspondiente al mismo. En el evento "obener" se recibe del servicio la lista de todos los clientes conectados y en el caso de que el evento producido sea la desconexión del servidor se actualiza la lista a ningún usuario.

Probar su funcionamiento:

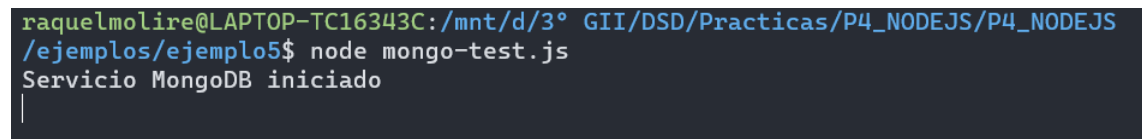
1. Ejecutando desde el terminal: `$nodejs connections.js`

Por motivos de instalación y/o sistema operativo, para poder realizar la ejecución de forma correcta se ha ejecutado previamente la siguiente línea:
`$npm i --save socket.io`

Además hay que tener en cuenta que el servicio mongodb esté activado, en mi caso se activa a partir de la siguiente línea:
`$ sudo service mongodb start`

También es importante considerar que no se pueden crear colecciones que ya estén creadas anteriormente.

2. Una vez ejecutado el servicio, conectar diferentes clientes a partir del navegador con `http://localhost:8080` y desconectar el servicio:



```
raquelmolire@LAPTOP-TC16343C:/mnt/d/3° GII/DSD/Practicas/P4_NODEJS/P4_NODEJS
/ejemplos/ejemplo5$ node mongo-test.js
Servicio MongoDB iniciado
|
```

Figura 15: Se ejecuta el servicio

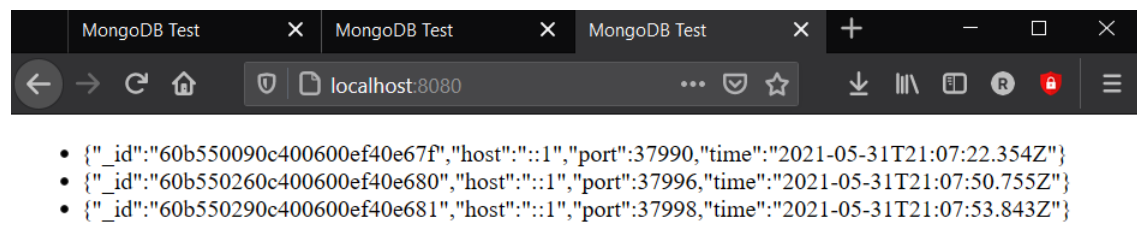


Figura 16: Se conectan 3 clientes y se muestra el contenido de la entrada de la base de datos

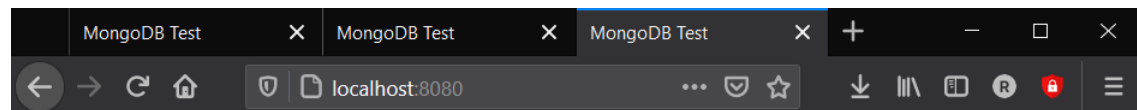


Figura 17: Se desconecta el servicio y la lista de usuarios se actualiza a ningún usuario

Parte II

Sistema domótico

Esta parte consiste en desarrollar, con el uso de Node.js, Socket.io y MongoDB un sistema domótico compuesto por sensores y actuadores. Los actores del sistema a implementar son cuatro: Los usuarios y sensores de la estancia que serán clientes del sistema, el servidor del mismo y un agente que controla los eventos que se producen.

El servidor proporciona las páginas necesarias para la conexión de los usuarios y los sensores y mantiene la comunicación directa con el agente.

El comportamiento del sistema es el siguiente:

- Los sensores difunden el valor de las medidas de los mismos y mostrarán el estado actual de los actuadores mediante el acceso al sistema través del servidor.
- Los usuarios acceden al estado del sistema a través del servidor y muestran el estado actual del mismo: los valores que proporcionan los sensores y los botones para poder modificar el estado de los actuadores.
- El servidor proporcionará las páginas necesarias para los clientes, que serán los sensores y los usuarios, se encargará del manejo del sistema, así como de la emisión y recepción de los eventos necesarios para el control del mismo.
- El agente mantiene una comunicación directa con el servidor controlando el valor de las medidas actuales y realizará cambios en el sistema rigiéndose por distintos valores umbrales, además gestiona el envío de alarmas.

Los requisitos a tener en cuenta engloban:

- El servidor guardará un histórico de los eventos producidos en el sistema en una base de datos con la correspondiente marca de tiempo asociada a cada evento.
- El servidor debe permitir la conexión de múltiples usuarios y sensores.

6. Planteamiento de la solución

Para la realización de esta aplicación se ha englobado a los usuarios y los sensores como clientes del servidor, pues ambos requieren del conocimiento del estado del mismo.

Cuando un cliente se conecta, este se almacena en el servidor como cliente conectado y podrá suscribirse a los eventos que quiera y que sean proporcionados por el servidor. Una vez se ha conectado el servidor le emite su información y los valores actuales del sistema que engloba el valor de los sensores y el estado de los actuadores, además si ya estuvieran registradas algunas alarmas también le serán notificadas al cliente al realizar esa conexión.

Cuando los clientes quieran realizar cambios en el estado del sistema, bien sea por un cambio del valor de un sensor en el caso de los sensores o por el cambio en el estado de un actuador en

el caso de los usuarios, notifican al servidor para que este controle los nuevos valores a partir del agente y notifique a todos los clientes del sistema de los nuevos cambios. De manera que la técnica para que los nodos del sistema cooperen entre sí se realiza mediante **actualización de datos**.

Cabe destacar que se ha extendido la funcionalidad de modo que se han incluido nuevos sensores y actuadores, la interfaz de usuario es muy completa y se maneja la detección de eventos complejos.

Breve inciso a que se han intentado probar otros módulos de Node.js para por ejemplo twittear cambios en los actuadores, pero no ha sido posible llevarlo a cabo por diversos fallos o bien de la desactualización de los módulos o incompatibilidades con el SO. En concreto se ha probado con *node-tweet-stream* y *twitter-lite*

7. Sensores y umbrales máximos y mínimos

- Temperatura : °C. (32 Max / 5 Min)
- Luminosidad: Lux . (200 Max / 100 Min)
- Humedad: Tanto por ciento %. (60 Max / 40 Min)
- Calidad del aire: ICA. (200 Max / 0 Min)
- Movimiento: Verdadero o falso si se ha detectado o no movimiento.

8. Actuadores

- Aire acondicionado
- Calefacción.
- Persiana
- Humidificador
- Humidificador
- Purificador..

9. Eventos controlados por el agente:

9.1. Eventos que solo producen alarmas

- La temperatura actual sobrepasa el umbral máximo (32).
- La temperatura actual sobrepasa el umbral mínimo(5).
- La luminosidad actual sobrepasa el umbral máximo (200).
- La luminosidad actual sobrepasa el umbral mínimo(100).

- La humedad actual sobrepasa el umbral máximo (60).
- La humedad actual sobrepasa el umbral mínimo(40).
- La calidad del aire actual sobrepasa el umbral máximo (200).
- La calidad del aire actual sobrepasa el umbral mínimo(0).
- Se enciende la luz pero la luminosidad actual es alta (150)
- Si se enciende el aire acondicionado con una temperatura baja (¡15) se enciende la calefacción con una temperatura alta (¿15)

9.2. Eventos que producen alarmas y cambios de estado

- Se nota un movimiento, la luminosidad es baja (120) y las luces están apagadas : Se encienden las luces.
- La temperatura y la luminosidad sobrepasan los umbrales máximos: Se cierra la persiana
- La humedad sobrepasa el umbral máximo y el humidificador está activado: Se apaga el humidificador.
- La humedad sobrepasa el umbral mínimo y el humidificador está desactivado: Se enciende el humidificador.
- La calidad del aire sobrepasa el umbral máximo y el purificador de aire está activado: Se apaga el purificador de aire.
- La calidad del aire sobrepasa el umbral mínimo y el purificador de aire está desactivado: Se enciende el purificador de aire.
- Si el aire acondicionado está activado y la calefaccion tambien:
- Si la temperatura actual es mayor que 15 : Se apaga la calefacción y se deja encendido el aire.
- Si la temperatura actual es menor que 15: Se apaga el aire se deja encendida la calefacción.

10. Ejemplos de ejecución

- Iniciar el servidor de MongoDB: `$sudo service mongod start`
- `$node servidor.s` Por motivos de instalación y/o sistema operativo, para poder realizar la ejecución de forma correcta se ha ejecutado previamente la siguiente línea:
`$npm i --save socket.io`

Además hay que tener en cuenta que el servicio mongod esté activado, en mi caso se activa a partir de la siguiente línea:
\$ sudo service mongod start

También es importante considerar que no se pueden crear colecciones que ya estén creadas anteriormente.

10.1. Ejemplo

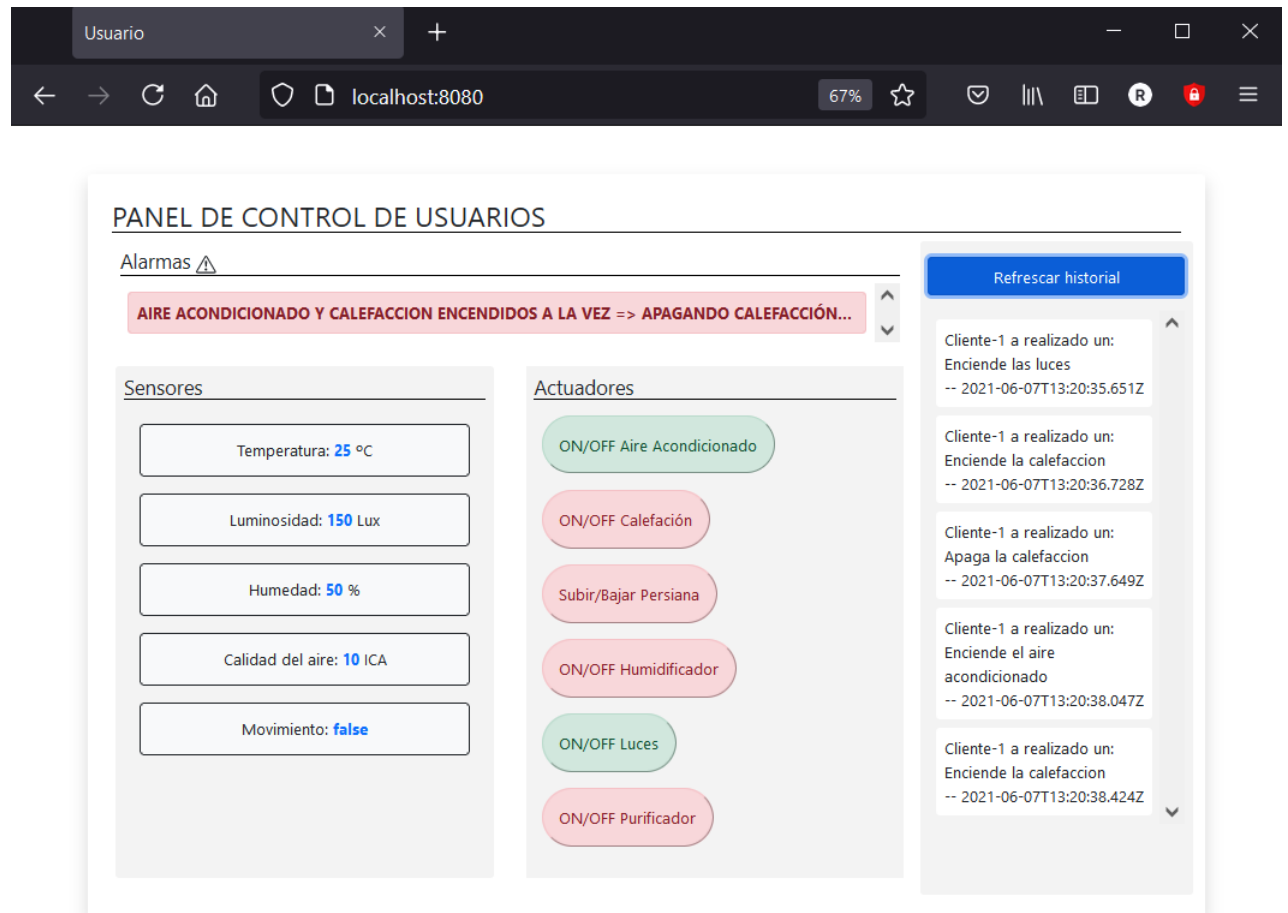


Figura 18: Usuarios

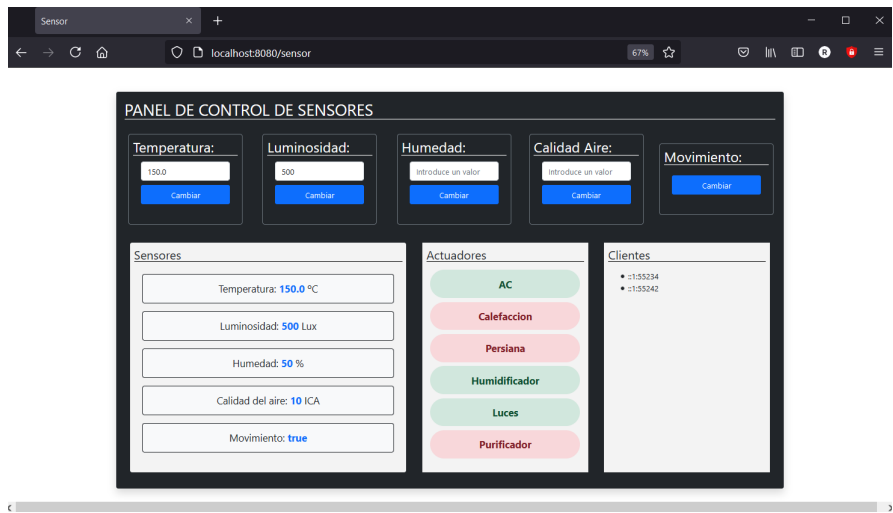


Figura 19: Sensores

```

raquel@lirio:APTOD-TC16393C:/mnt/d/3* G11/DSD/Practicas/P4_NODEJS/sist_domotico$ node servidor.js
Servicio MongoDB iniciado
New connection from ::1:55264
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
Se modifica el estado de la calefaccion
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
Se modifica el estado de la persiana
New connection from ::1:55266
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
Se modifica el estado del humidificador
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
Se modifica el estado de las Luces
Se modifica el estado de los sensores
  Agente: se sobrepasa el limite máximo de temperatura!
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
Se modifica el estado de los sensores
  Agente: se sobrepasa el limite máximo de temperatura!
  Agente: se sobrepasa el limite máximo de luminosidad!
  Agente: se cierra la persiana porque se sobre pasan los limites máximos de luz y temperatura!
  Agente: se detecta que se ha encendido la luz pero la luminosidad actual es alta!
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
Se modifica el estado de los sensores
  Agente: se sobrepasa el limite máximo de temperatura!
  Agente: se sobrepasa el limite máximo de luminosidad!
  Agente: se sobrepasa el limite máximo de humedad!
  Agente: se sobrepasa el limite máximo de temperatura!
  Agente: se detecta que se ha encendido la luz pero la luminosidad actual es alta!
  Agente: se detecta que esta encendida la calefaccion con una temperatura alta!
  Agente: se sobrepasa el limite máximo de temperatura!

```

Figura 20: Servidor