

Node.js

PRÁCTICA 4

CARMEN CHUNYIN FERNÁNDEZ NÚÑEZ

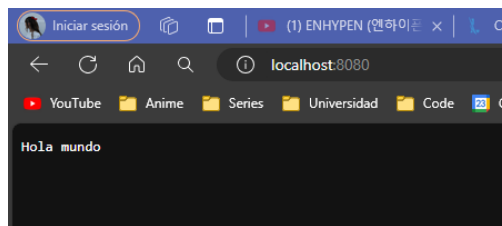
1 INTRODUCCIÓN

En esta práctica se pide probar unos ejemplos de prueba y comentarlos, e implementar un sistema domótico básico. Para ello, es necesario tener instalado Node.js y MongoDB. También es necesario instalar npm para poder instalar los módulos de Socket.io y el de MongoDB.

2 PRIMERA PARTE. EJEMPLOS

2.1 HELLOWORLD.JS

Este ejemplo lo único que realiza es mostrar por pantalla en el navegador el mensaje **Hola mundo**. Adicionalmente, muestra por la terminal del servidor el JSON con toda la información de la petición del cliente.



```
alissea@ALISSEA:/mnt/c/Users/carne/OneDrive/Documentos/Universidad/3IngInf/SEGUNDO_CUATRI/DSD/Prácticas/DSD_UGR/P4/codigoEjemplos$ node ./helloworld.js
Servicio HTTP iniciado
{
  host: 'localhost:8080',
  connection: 'keep-alive',
  'cache-control': 'max-age=0',
  'sec-ch-ua': '"Microsoft Edge";v="125", "Chromium";v="125", "Not.A/Brand";v="24"',
  'sec-ch-ua-mobile': '70',
  'sec-ch-ua-platform': '"Windows"',
  dnt: '1',
  'upgrade-insecure-requests': '1',
  'user-agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/537.36 Edg/125.0.0.0',
  accept: 'text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7',
  'sec-fetch-site': 'none',
  'sec-fetch-mode': 'navigate',
  'sec-fetch-user': '?1',
  'sec-fetch-dest': 'document',
  'accept-encoding': 'gzip, deflate, br, zstd',
  'accept-language': 'es,es-ES;q=0.9,en;q=0.8,en-GB;q=0.7,en-US;q=0.6',
  cookie: 'pma_lang=es; PHPSESSID=f3e492bf986172cadcee7ff2d561e356'
}
```

Para implementarlo, lo primero que hay que hacer es montar el servidor HTTP. Para ello, se importa el módulo de Node.js **http**.

Una vez importado, se crea el servidor con el método **createServer** cuyo argumento es una función que se ejecuta cada vez que se realiza una petición HTTP. Para que el servidor muestre la información de la petición, es necesario usar **request.headers** y pasarlo al log de la consola de JavaScript.

Para que el cliente reciba el mensaje, es necesario escribir una cabecera básica con el código 200, indicando que es correcto y el tipo MIME, que en este caso será texto plano. Luego con el método **write**, se escribe el mensaje en el cliente. Por último, se finaliza con **end** para que se mande el mensaje.

```
import http from 'node:http';

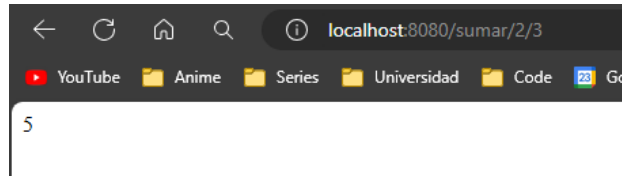
http.createServer((request, response) => {
  console.log(request.headers);
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.write('Hola mundo');
  response.end();
})
.listen(8080);

console.log('Servicio HTTP iniciado');
```

2.2 CALCULADORA.JS

Este ejemplo implementa una calculadora básica utilizando Node.js. A la calculadora se le pasa el operador y los operandos en la URL del navegador. En caso de ser incorrectos o no tener suficientes, muestra un error. Las operaciones implementadas son: sumar, restar, producto, dividir.

La sintaxis de la URL sería la siguiente: <http://localhost:8080/<operador>/<operando-1>/<operando-2>>



La parte de creación del servidor es igual que en el ejemplo anterior. Además, se ha creado una función **calcular** que permite realizar las funciones de la calculadora. Para obtener los parámetros de la operación se realiza lo siguiente:

1. **let {url} = request;** : Extrae la propiedad url del objeto request y la asigna a la variable url.
2. **url = url.slice(1);** : Elimina el primer carácter de la URL (posiblemente un carácter / adicional).
3. **const params = url.split('/');** : Divide la URL en un array de subcadenas, utilizando / como delimitador.
4. **let output='';** : Inicializa una variable output vacía que se utilizará para almacenar el resultado del cálculo.
5. Comprueba si el array params tiene al menos 3 elementos. Si es así, se realizará el cálculo; de lo contrario, no se realizará ninguna operación.
6. Convierte el segundo y tercer elemento del array params en un número de punto flotante y lo asigna a la variable val1 y val 2 respectivamente.
7. **const result = calcular(params[0], val1, val2);** : Llama a la función calcular. El resultado de la función se almacena en la variable result.
8. **output = result.toString();** : Convierte el resultado en una cadena de texto y lo asigna a la variable output.

Por último, se manda de manera idéntica a la explicada en el ejemplo anterior

```
import http from 'node:http';

function calcular(operacion, val1, val2) {
  if (operacion == 'sumar') return val1+val2;
  else if (operacion == 'restar') return val1-val2;
  else if (operacion == 'producto') return val1*val2;
  else if (operacion == 'dividir') return val1/val2;
  else return 'Error: Parámetros no válidos';
}

http.createServer((request, response) => {
  let {url} = request;
  url = url.slice(1);
  const params = url.split('/');
  let output='';
  if (params.length >= 3) {
    const val1 = parseFloat(params[1]);
    const val2 = parseFloat(params[2]);
    const result = calcular(params[0], val1, val2);
    output = result.toString();
  }
  else output = 'Error: El número de parámetros no es válido';

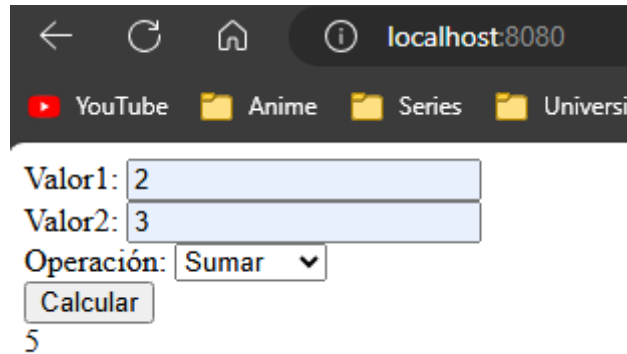
  response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
  response.write(output);
  response.end();
})
.listen(8080);

console.log('Servicio HTTP iniciado');
```

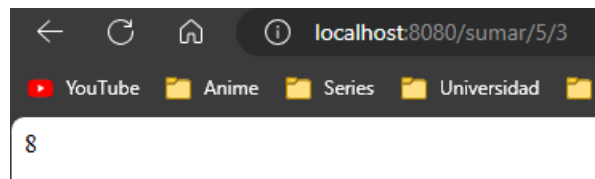
2.3 CALCULADORA-WEB.JS

Este ejemplo implementa lo mismo que **calculadora.js**, pero teniendo una interfaz web que simplifica la introducción de los valores. Hace uso de los módulos adicionales **path** y **fs** para buscar el archivo HTML con la ruta, para comprobar la existencia de archivos en el sistema y para ofrecerlos al cliente.

Solicitudes a la raíz (/): el servidor devuelve el contenido del archivo **calc.html** ubicado en el mismo directorio que el archivo JavaScript. El archivo se lee utilizando **readFile** y se envía la respuesta HTTP con un código de estado 200 (OK) y un tipo de contenido HTML. Si ocurre un error al leer el archivo, se envía una respuesta con un código de estado 500 (Error interno del servidor) y un mensaje de error.



Solicitudes a cualquier otra ruta: el servidor calcula el resultado de una operación matemática básica (suma, resta, producto o división) entre dos valores numéricos proporcionados como parámetros en la URL. Los parámetros deben estar en el siguiente formato: **<operacion>/<valor1>/<valor2>**. Por ejemplo, una solicitud a **/sumar/5/3** devolverá el resultado de la suma de 5 y 3, es decir, 8. Es decir, como el ejemplo anterior.

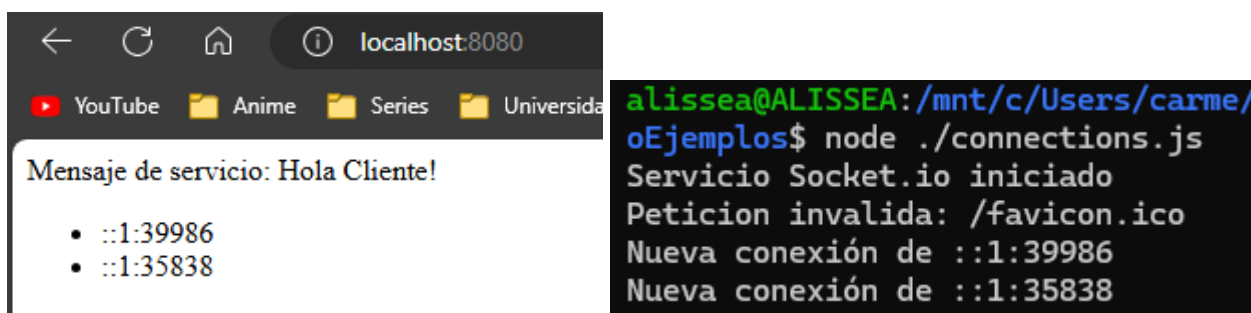


2.4 CONNECTIONS.JS

Este ejemplo hace uso del módulo **Socket.io** para poder actualizar todos los clientes de manera asíncrona y realizar una comunicación entre el cliente y el servidor para que todos tengan la información más actualizada.

Cuando un cliente se conecta, este aparece en la pantalla del navegador con su dirección IP y el puerto usado. Además, si se abren nuevas pestañas conectadas también, se observa como aparece la dirección y el puerto del nuevo cliente junto a los que ya estaban conectados. Esto mismo pasa con las desconexiones, los clientes se actualizan en función de las conexiones o desconexiones.

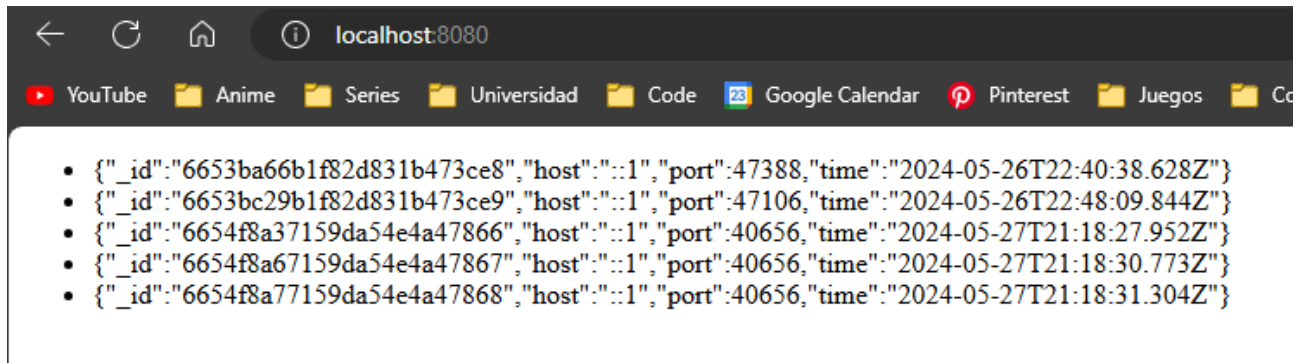
En la pantalla del servidor aparecen los clientes conectados, las nuevas conexiones y cuando un usuario se va a desconectar. Además, cuando el cliente se desconecta del servidor, aparece un mensaje informándolo.



Cada vez que un cliente se conecte al servidor se ejecutará el callback pasado a **on(connection)**. Cuando un cliente se conecta al servidor Socket.IO, se almacena su dirección IP y puerto en el array **allClients**. Se emite un evento **all-connections** a todos los clientes conectados, incluyendo el cliente que se acaba de conectar. Cuando un cliente se desconecta, se elimina su dirección IP y puerto del array **allClients** y se emite de nuevo el evento **all-connections** a todos los clientes conectados.

2.5 MONGO-TEST.JS

Este ejemplo muestra al cliente todas las conexiones a la base de datos NoSQL, mostrando en un JSON el identificador, la dirección IP donde se encuentra el cliente, el puerto y la hora a la que se ha realizado la conexión. Cuando se refresca la página o se abre una pestaña nueva del navegador, aparecen nuevas líneas de conexiones a la base de datos.



3 SEGUNDA PARTE. SISTEMA DOMÓTICO

3.1 SERVIDOR.JS

El servidor tendrá varias funciones. Por un lado, llevará el recuento de los clientes conectados, al igual que el ejemplo de connections, aunque esta funcionalidad no es utilizada. Por otro lado, almacenará en una base de datos los cambios hechos a los sensores, de forma similar al ejemplo mongo-test. Y, además, hará de intermediario entre las distintas vistas (clientes, sensores, agente), respondiendo a sus eventos.

```
.createServer((request, response) => {
  let {url} = request;
  if(url == '/'){
    url = '/cliente.html';
    const filename = join(process.cwd(), url);

    readFile(filename, (err, data) => {
      if(!err){
        response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
        response.write(data);
        response.end();
      }
      else{
        response.writeHead(200, {'Content-Type': 'text/plain'});
        response.write('Error de lectura en el fichero: '+ url);
        response.end();
      }
    });
  }
  else{
    const filename = join(process.cwd(), url + '.html');

    readFile(filename, (err, data) => {
      if(!err){
        response.writeHead(200, {'Content-Type': 'text/html; charset=utf-8'});
        response.write(data);
        response.end();
      }
      else{
        response.writeHead(200, {'Content-Type': 'text/plain'});
        response.write('Error de lectura en el fichero: '+ url);
        response.end();
      }
    });
  }
});
```

Por ejemplo, cuando se accede a la página de los sensores (localhost:8080/sensores) y se rellena el formulario y envían los nuevos valores, estos se guardan en la base de datos y se envían a los clientes.

```
client.on('actualizar-sensores', (data) =>{
  console.log('Actualizando sensores');
  sensores.insertOne(data, {safe:true}).then((result) => {
    io.sockets.emit('actualizar-sensores',{
      temperatura: data.temperatura,
      luminosidad: data.luminosidad,
      humedad: data.humedad,
      time:data.time
    });
    console.log(data);
  });
});
```

Además, he creado una función que obtiene el último valor medido en la base de datos, para así poder inicializar los clientes a su último estado.

```
client.on('obtener', () => {
  sensores.find().sort({_id:-1}).limit(1).forEach(function(result){
    client.emit('obtener',{
      temperatura: result.temperatura,
      luminosidad: result.luminosidad,
      humedad: result.humedad,
      time:result.time
    });
  });
});
```

Y para poder mostrar una lista con todas las modificaciones, se crea una función que devuelve el array de la base de datos

```
client.on('historico', () => {
  sensores.find().toArray().then((results) => {
    client.emit('historico', results);
  });
});
```

Para gestionar las alertas, he creado una variable **alertas** en el servidor y la editamos a través de tres funciones, una para añadir alertas, otra para poder mostrarlas y una última para eliminar todas las alertas del array.

```
client.on('alerta', (data) =>{
  alertas.push(data);
  io.sockets.emit('alerta', data);
});

client.on('mostrar-alertas',()=>{
  io.sockets.emit('mostrar-alertas', alertas);
});

client.on('borrar-alertas', () =>{
  alertas = [];
  io.sockets.emit('mostrar-alertas', alertas);
});
```

Para todos y cada uno de los actuadores existen dos funciones, una que obtiene el valor del actuador, los cuales se guardan en variables dentro del servidor. Y otra que lo modifica.

```
// ACTUADORES
client.on('obtener_estado_persiana', () =>{
  client.emit('persiana', estado_persiana);
});

client.on('persiana', () =>{
  if(estado_persiana == 'abierta')
    estado_persiana = 'cerrada';
  else
    estado_persiana = 'abierta';

  io.sockets.emit('persiana', estado_persiana);
  console.log("La persiana está: " + estado_persiana);
});
```

Por último, he creado un método para poder resetear la base de datos

```
client.on('resetear', () => {
  console.log('Se ha reseteado la base de datos');
  dbo.collection("sensores").drop();
  io.sockets.emit('resetear', () => {});
});
```

En la terminal se muestran mensajes como: los datos introducidos a la BD, usuarios conectados o desconectados, mensajes de actuadores...

```
alissea@ALISSEA:/mnt/c/Users/carne/OneDrive/Documentos$ node servidor.js
Servicio Socket.io iniciado
Nueva conexión de ::1:41962
Nueva conexión de ::1:52166
Actualizando sensores
{
  temperatura: '20',
  luminosidad: '200',
  humedad: '30',
  time: '2024-05-28T20:35:15.741Z',
  _id: new ObjectId('66564003ac2cd8a35a1896d7')
}
El usuario ::1:41962 se va a desconectar
El usuario ::1:41962 se ha desconectado
Nueva conexión de ::1:38046
La persiana está: abierta
La persiana está: cerrada
```

3.2 CLIENTE.HTML

La vista del cliente simplemente recogerá los datos adquiridos de los sensores y los mostrará por pantalla, al igual que el estado de los actuadores y un historial de las modificaciones en los sensores. Además, desde el usuario podremos encender o apagar todos los actuadores mediante botones, al igual que resetear la base de datos. Le he añadido la hora a tiempo real.

Sensores:

Temperatura: 20°C

Luminosidad: 200 lumens

Humedad: 30%

Hora: 22:35:19

Actuadores:

Estado de la persiana: cerrada

Estado del A/C: apagado

Estado del radiador: apagado

Estado del humidificador: apagado

Estado del deshumidificador: apagado

Estado de las luces: apagadas

Historial

- temperatura: 20, luminosidad: 200, humedad: 30, fecha: 2024-05-28T20:35:15.741Z

Nada más conectar un cliente, se obtienen todos los estados de los actuadores además de los datos más recientes de los sensores, para poder iniciar el cliente.

```
socket.on('connect', () => {
  socket.emit('obtener');
  socket.emit('historico');
  socket.emit('obtener_estado_persiana');
  socket.emit('obtener_estado_aire');
  socket.emit('obtener_estado_radiador');
  socket.emit('obtener_estado_humidificador');
  socket.emit('obtener_estado_deshumidificador');
  socket.emit('obtener_estado_luces');
  socket.emit('mostrar-alertas');
});
```

Cuando se actualizan los sensores, se ven reflejados los cambios en el cliente

```
socket.on('actualizar-sensores', (data) => {
  var temp = document.getElementById('temp');
  var lum = document.getElementById('lum');
  var hum = document.getElementById('hum');

  temp.innerHTML = data.temperatura;
  lum.innerHTML = data.luminosidad;
  hum.innerHTML = data.humedad;
  socket.emit('historico');
});
```


Los botones, al hacerles click cambian el estado del actuador asociado.

```
var botonReseteo = document.getElementById('resetear');
botonReseteo.addEventListener('click', (event) => {
    socket.emit('resetear');
    socket.emit('borrar-alertas');
});
```

Cuando se resetea, se eliminan todas las alertas además del historial

```
var botonReseteo = document.getElementById('resetear');
botonReseteo.addEventListener('click', (event) => {
    socket.emit('resetear');
});

socket.on('resetear', () =>{
    socket.emit('historico');
    socket.emit('borrar-alertas');
});
```

Mostrar alertas, muestra todas las alertas que hubiese en el array de alertas del servidor

```
socket.on('mostrar-alertas', (alertas) => {
    const alerta = document.getElementById('alerta');
    alerta.innerHTML = '';
    alertas.forEach((alertaIndividual) => {
        const p = document.createElement('p');
        p.innerHTML = alertaIndividual;
        alerta.appendChild(p);
    });
});
```

3.3 SENSORES.HTML

La página de los sensores consiste en un formulario que lo único que realiza es enviar los datos al servidor para que éste los guarde en la base de datos y se los envíe al cliente. Además, vacía el array de alertas



← ↻ 🏠 🔍 ⓘ localhost:8080/sensores

YouTube 📁 Anime 📁 Series 📁 Universidad 📁 Code 23 Goo

Temperatura:

Luminosidad:

Humedad:

```
function enviar_sensores() {
    var temp = document.getElementById("temperatura").value;
    var lum = document.getElementById("luminosidad").value;
    var hum = document.getElementById("humedad").value;

    var serviceURL = 'localhost:8080';
    var socket = io.connect(serviceURL);
    var d = new Date();
    socket.emit('borrar-alertas');
    socket.emit('actualizar-sensores', { temperatura: temp, luminosidad: lum, humedad: hum, time: d});
}
```

3.4 AGENTE.HTML

El agente recibe los datos de los sensores como cualquier otro cliente, además de almacenar el valor de los actuadores. Solo que, si algún valor de los sensores sobrepasa un mínimo o máximo, apagará o encenderá determinado actuador si es que no lo estaba antes. Además, mandará una alerta al cliente avisándole de que ha encendido o apagado el actuador. Como funcionalidad nueva le he añadido que encienda o apague las luces a una determinada hora automáticamente, al igual que subir y bajar las persianas

```
socket.on('actualizar-sensores', (data) =>{

    temp_actual = data.temperatura;
    lum_actual = data.luminosidad;
    hum_actual = data.humedad;

    if(temp_actual > temp_maxima && estado_ac == 'apagado'){
        socket.emit('aire');
        socket.emit('alerta', "ALERTA: Mucha Calor!! Encendiendo el aire");
    }
    else if(temp_actual < temp_min && estado_ac == 'encendido'){
        socket.emit('aire');
        socket.emit('alerta', "ALERTA: Mucho Frio!! Apagando el aire");
    }
}
```

```
const controlluces = () => {
    const ahora = new Date();
    const hora = ahora.getHours();

    console.log(estado_luces)
    if((hora >= hora_encender || hora < hora_apagar) && estado_luces == 'apagadas'){
        socket.emit('luces')
        socket.emit('alerta', "Se ha hecho de noche, encendiendo las luces");
    }
    else if(hora >= hora_apagar && hora < hora_encender && estado_luces == 'encendidas'){
        socket.emit('luces')
        socket.emit('alerta', "Es de día, apagando las luces");
    }
}

if((hora >= hora_encender || hora < hora_apagar) && estado_persiana == 'abierta'){
    socket.emit('persiana');
    socket.emit('alerta', "Se ha hecho de noche, cerrando la persiana");
}
else if(hora >= hora_apagar && hora < hora_encender && estado_persiana == 'cerrada'){
    socket.emit('persiana');
    socket.emit('alerta', "Es de día, sube las persianas");
}

socket.emit('mostrar-alertas');
}

setInterval(controlluces, 5000);
controlluces();

socket.on('disconnect', function(params) {
    document.getElementById("estado_agente").innerText = "Inactivo";
});
```

3.5 RESUMEN Y EJEMPLO DE USO

Los sensores almacenan valores de temperatura, luminosidad y humedad.

Los actuadores son:

- Aire acondicionado
- Persianas
- Radiador
- Humidificador
- Deshumidificador
- Luces

Todos ellos se pueden encender y apagar manualmente, además, si está el agente activo, se encargará de apagarlos o encenderlos cuando sea necesario.ç

r

Sensores:

Temperatura: 32°C

Luminosidad: 200 lumens

Humedad: 10%

Hora: 22:50:27

Actuadores:

Estado de la persiana: cerrada

Estado del A/C: encendido

Estado del radiador: apagado

Estado del humidificador: encendido

Estado del deshumidificador: apagado

Estado de las luces: encendidas

Persiana

Aire Acondicionado

Radiador

Humidificador

Deshumidificador

Luces

ALERTA: Mucha Calor!! Encendiendo el aire

ALERTA: Mucha Humedad!! Encendiendo el humidificador

Historial

- temperatura: 20, luminosidad: 200, humedad: 30, fecha: 2024-05-28T20:35:15.741Z
- temperatura: 32, luminosidad: 200, humedad: 10, fecha: 2024-05-28T20:50:23.332Z

Resetear BD

RESETEO

✓

Sensores:

Temperatura: 32°C

Luminosidad: 200 lumens

Humedad: 10%

Hora: 22:50:46

Actuadores:

Estado de la persiana: cerrada

Estado del A/C: encendido

Estado del radiador: apagado

Estado del humidificador: encendido

Estado del deshumidificador: apagado

Estado de las luces: encendidas

Persiana

Aire Acondicionado

Radiador

Humidificador

Deshumidificador

Luces

Historial

Resetear BD