Memoria Práctica 4: Node.js

Paso previo: Instalación del software

Para poder llevar a cabo la práctica ha sido necesario instalar los siguientes componentes:

- Node.js junto con su herramienta npm
  ```
  sudo apt install nodejs
  node -v
  npm -v
  ```

- La herramienta socket.io desde npm (en la carpeta del proyecto)
  ```
  sudo npm install socket.io
  ```

- La base de datos mongodb y su herramienta para poder acceder con node.js
  ```
  sudo apt-get install mongodb
  sudo npm install mongodb
  ```

Con los comandos especificados anteriormente he podido instalar correctamente el software en mi sistema (Ubuntu 20.04).

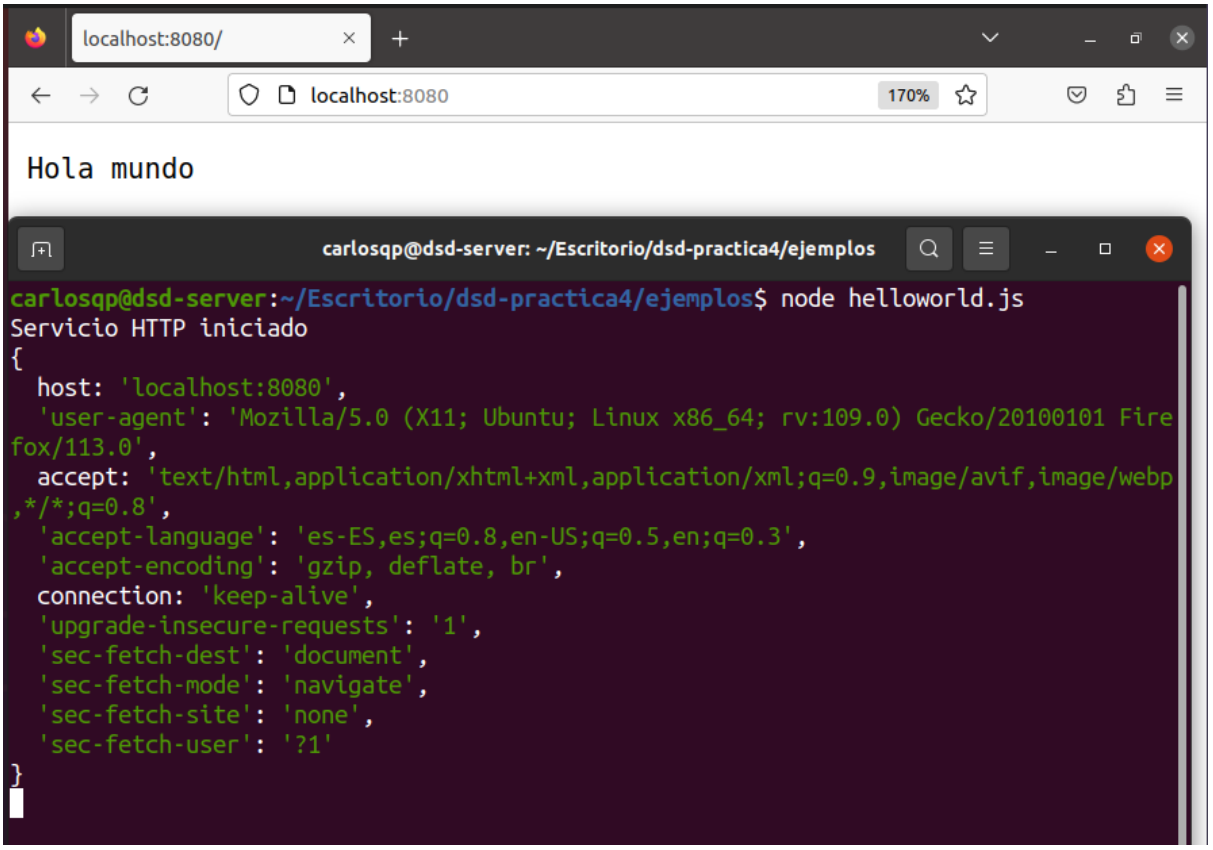
Primera parte: Ejecución de ejemplos

Pese a que se proporcionan estos ficheros en PRADO, he incluido en la carpeta `./ejemplos` del zip entregado los ficheros (en algunos incluyen otros comentarios, y en el caso específico de mongodb, se ha realizado una modificación del código).

Ejemplo 1: helloworld.js

Este es un ejemplo muy básico, en el que tras lanzar el script se carga una página que dice Hola mundo. Para ejecutarlo (desde la carpeta ejemplos):

```
node helloworld.js
```

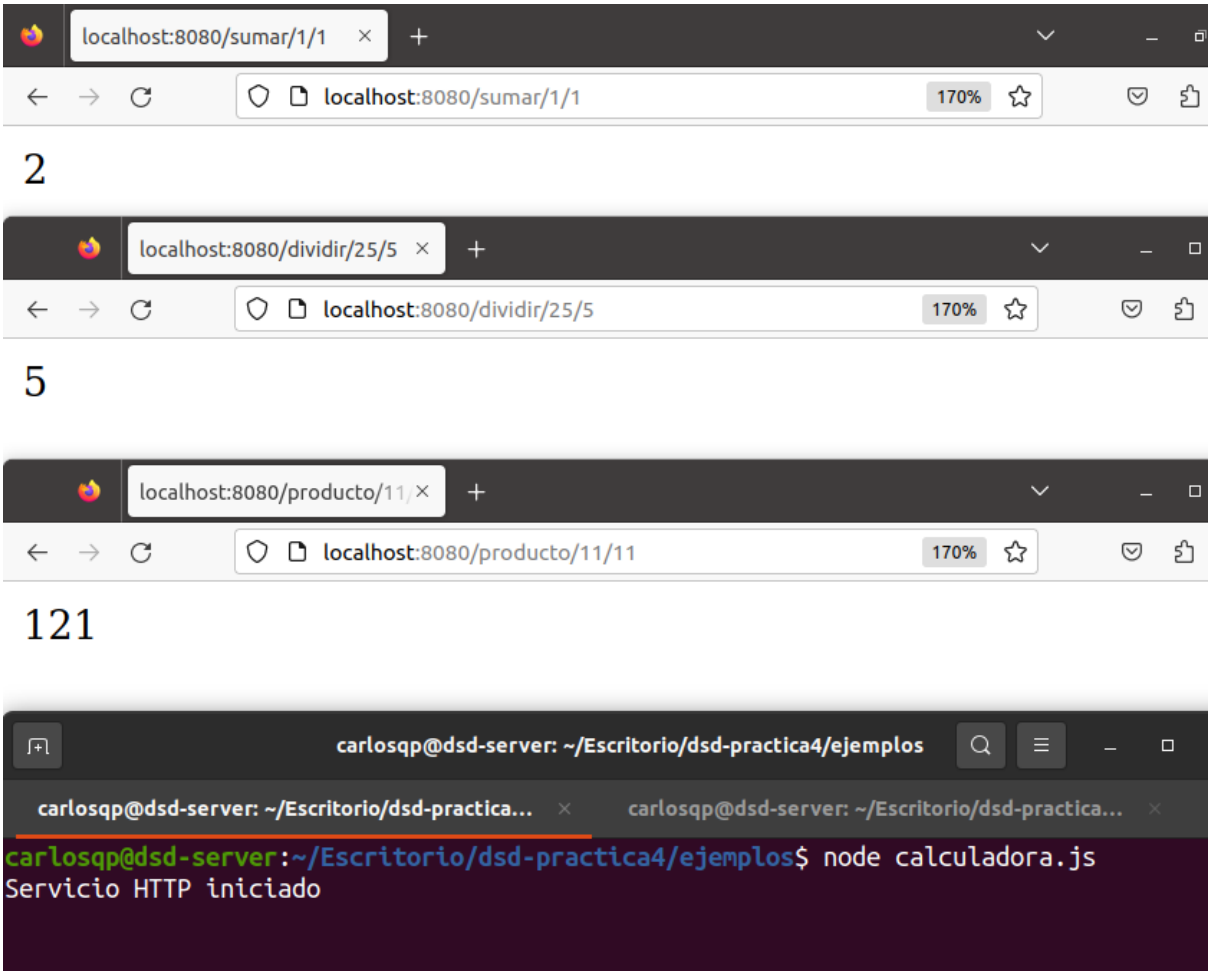


Ejemplo 2: calculadora.js

El siguiente ejemplo implementa una calculadora. Se introducen los parámetros por la URL. Cuando el servidor recibe una petición, analiza la URL, separa los parámetros (que deben de introducirse con el formato http://localhost:8080/operacion/operador1/operador2 , llama a la función calcular y posteriormente sirve una página que muestra el resultado de la operación solicitada.

Para ejecutarlo, desde la carpeta donde se encuentra el archivo:

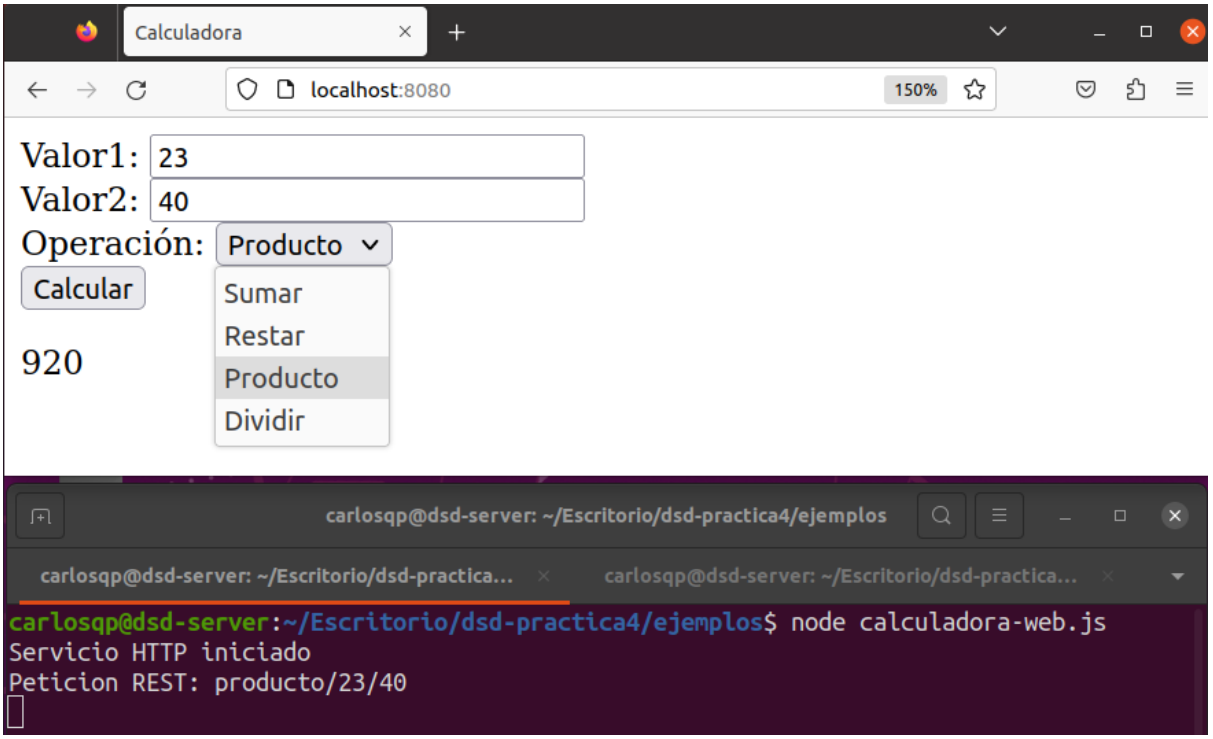
```
node calculadora.js
```



Ejemplo 3: calculadora-web.js

Este ejemplo mejora el ejemplo anterior, sirviendo esta vez una plantilla html. Esta proporciona una interfaz de usuario para realizar los cálculos en lugar de tener que pasarlos por parámetro en la URL. Lanzamos desde la terminal el script con node y abrimos en el navegador la siguiente dirección http://localhost:8080

```
node calculadora-web.js
```



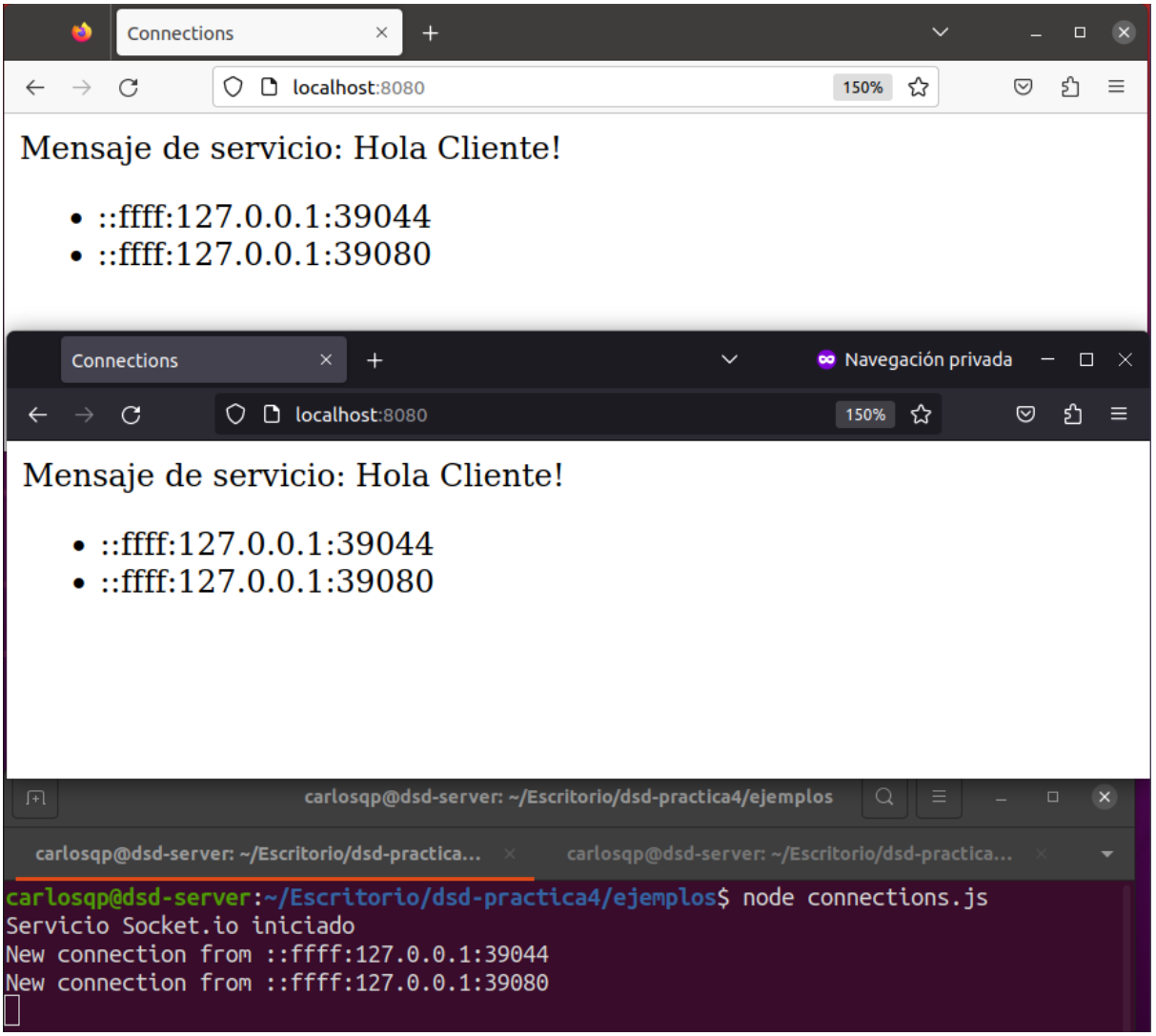
Ejemplo 4: connections.js

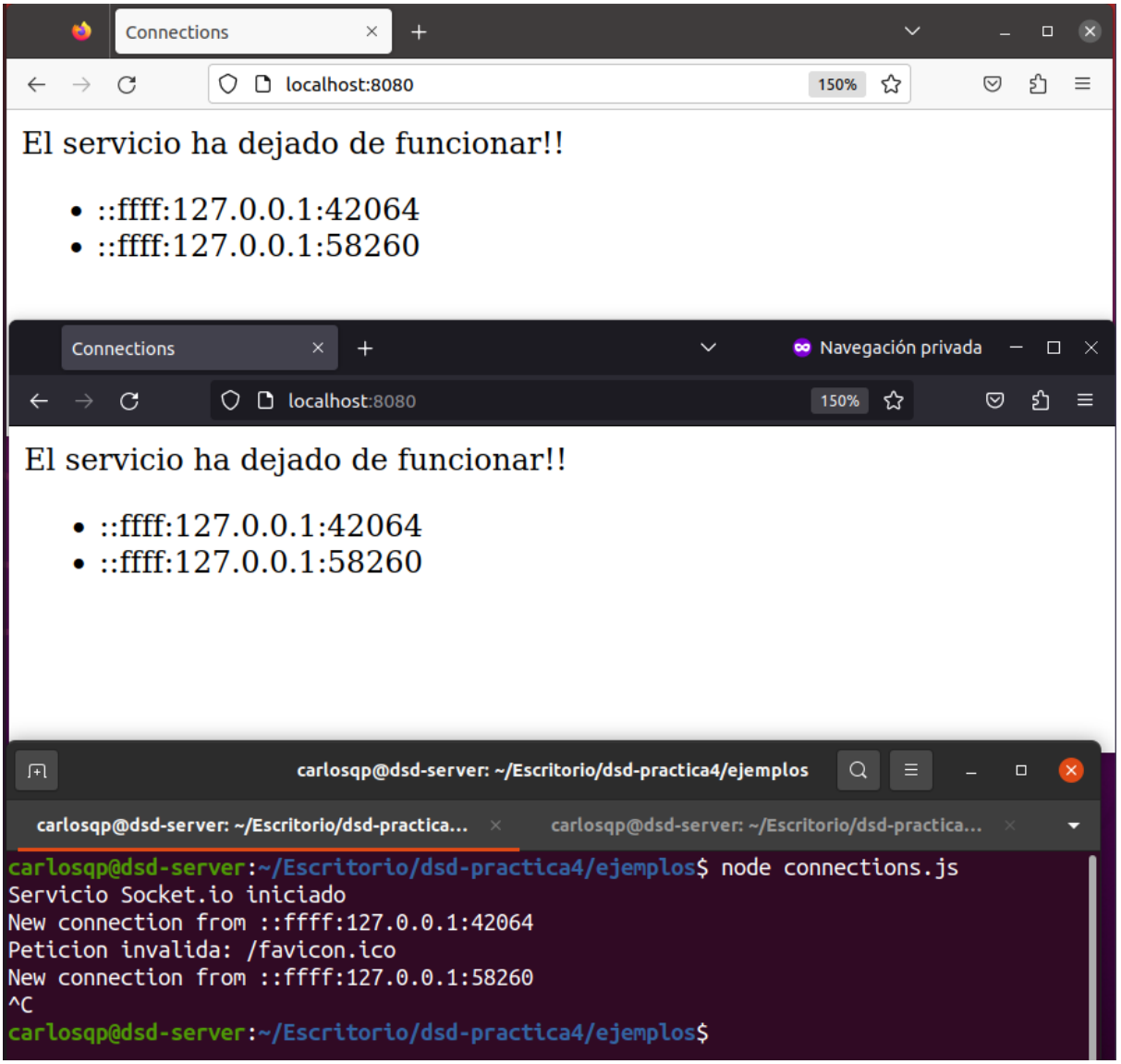
Este ejemplo es algo más complejo que los anteriores. Esta vez se utiliza socket.io, para manejar diferentes eventos asociados a los clientes que se conectan al servicio web. En específico se trata el evento de conexión y desconexión de los clientes, así como el de desconexión del servidor (ver segunda imagen), actualizando la lista de clientes conectados en tiempo real de cada uno de los clientes.

Para ejecutarlo, al igual que el resto, desde la ubicación donde se encuentra el archivo:

```
node connections.js
```

Y para probar su funcionamiento es conveniente abrir en varias pestañas el servicio web y observar como la lista de clientes conectados se actualiza en tiempo real.


Connections
localhost:8080
150%
Mensaje de servicio: Hola Cliente!
::ffff:127.0.0.1:39044
::ffff:127.0.0.1:39080
Connections
Navegación privada
localhost:8080
150%
Mensaje de servicio: Hola Cliente!
::ffff:127.0.0.1:39044
::ffff:127.0.0.1:39080
carlosqp@dsd-server: ~/Escritorio/dsd-practica4/ejemplos
carlosqp@dsd-server: ~/Escritorio/dsd-practica...
carlosqp@dsd-server: ~/Escritorio/dsd-practica...
carlosqp@dsd-server:~/Escritorio/dsd-practica4/ejemplos$ node connections.js
Servicio Socket.io iniciado
New connection from ::ffff:127.0.0.1:39044
New connection from ::ffff:127.0.0.1:39080



Ejemplo 5: mongo-test.js

Este último ejemplo es el más complejo de todos ellos. Inicialmente no podía ejecutar correctamente el código proporcionado. Pero tras mirar en el foro de dudas de la práctica en PRADO y la aclaración en clase del profesor pude detectar y solucionar el problema: mi versión del driver mongodb no soportaba el código proporcionado por lo cual tuve que realizar modificaciones (ver código del archivo entregado *`mongo-test.js`*.

Además de esto, la primera vez que ejecutaba esta nueva versión del programa, funcionaba correctamente pero después de esta el resto de ejecuciones fallaban. Esto se debe a la siguiente función:

```
dbo.createCollection("test").then((collection) => {…}).catch((err)
=> {...});
```

Esto se debe a que la primera vez que ejecuté el ejemplo no tenía creada la colección *test*, por lo que el programa lo creaba y realizaba toda la funcionalidad correctamente. Sin

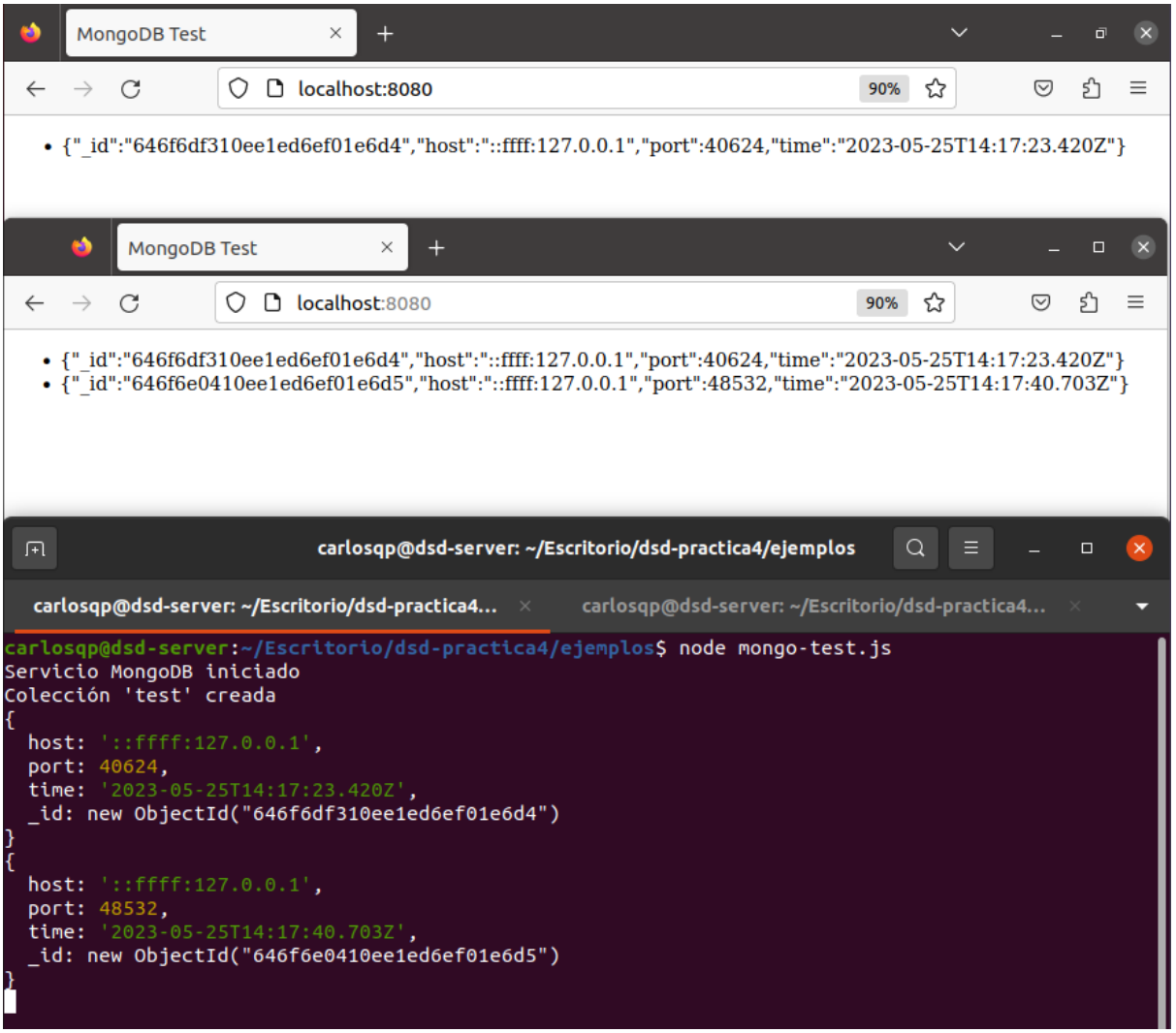
embargo en las posteriores ejecuciones, como la colección ya existía, el programa daba error. Hay varias soluciones para este problema, la más lógica es comprobar si dicha colección existe (y trabajar con ella sin volver a crearla) o no (en tal caso, crearla como ya se hizo en el código original). A continuación se muestran en negrita algunos de los cambios más relevantes en el código.

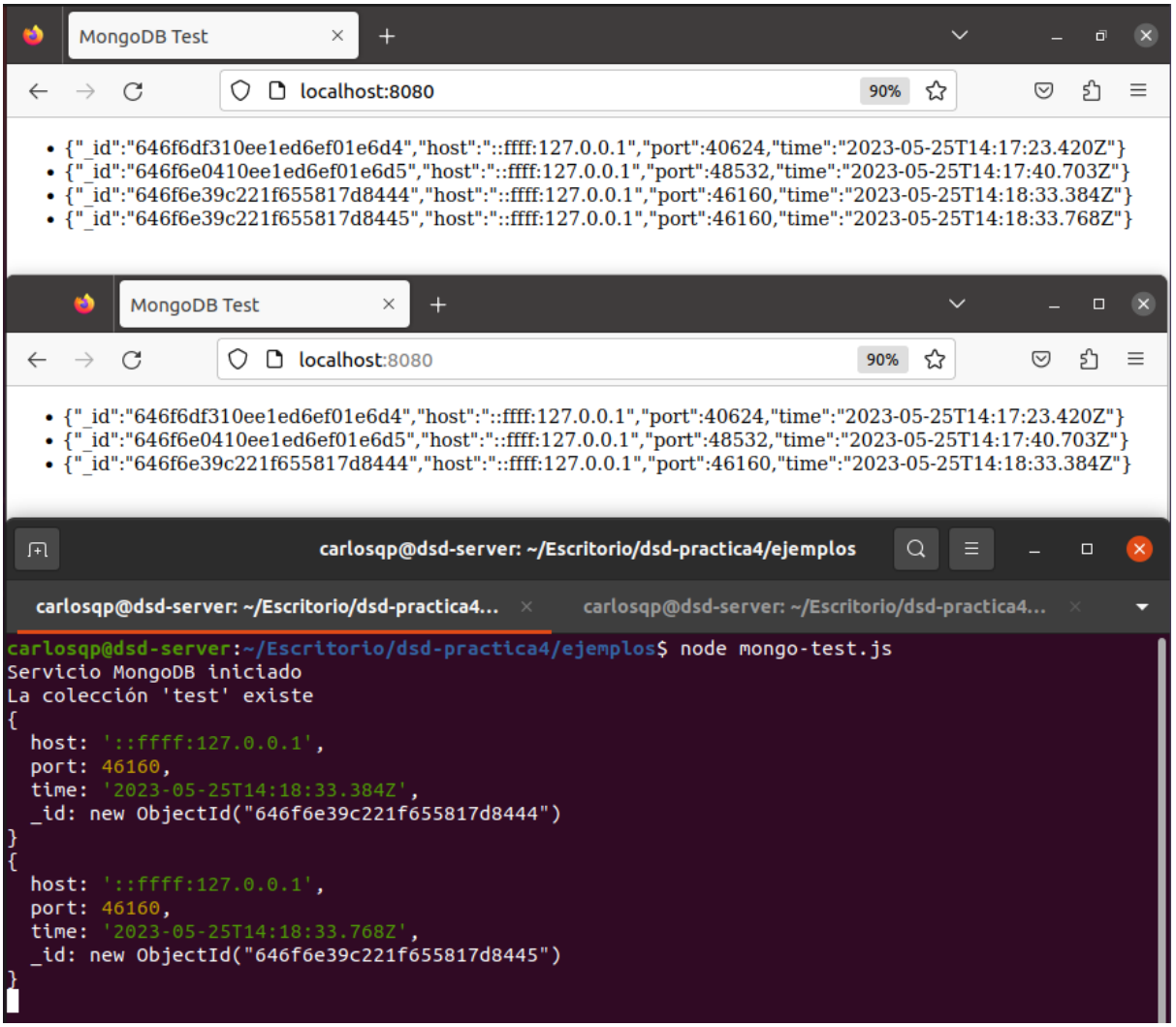
```
MongoClient.connect("mongodb://127.0.0.1:27017",                    {
useUnifiedTopology: true }, function(err, db) {
}).then((db) => {
      httpServer.listen(8080);
      var io = socketio(httpServer);

      var dbo = db.db("pruebaBaseDatos");
      // Comprueba si existe la coleccion en la base de datos
      dbo.listCollections({name:'test'})
      .next()
      .then((info) => {
            if(info) {
                  // La colección ya existe
                  console.log("La colección 'test' existe");
            } else {
                  // Si no existe, la crea
                  dbo.createCollection('test')
                  .then((collection) => {
                        console.log("Colección 'test' creada");
                  }).catch((err)=> {
                        console.log(err);
                  })
            }
      }).catch((err)=> {
            console.log(err);
      })

      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) {
                        dbo.collection('test').insertOne(data,
{safe:true}).then((result) => {
                              console.log(data)
                        }).catch((err) => {
                              console.error(err)
                        });
                  });
```

Tras arreglar todos estos problemas, lo que hace el programa es simplemente insertar en la tabla test un registro con un id único, la dirección IP del cliente, su puerto, y la hora de la conexión. El servidor envía al cliente todos los registros de la colección test para mostrarlos en una lista.

A continuación se muestran dos ejemplos de ejecución; el primero con la colección *test* sin que aún exista (el servidor la crea), y otra con la colección *test* ya creada. Para ejecutarlo, al igual que con todos se ejecuta el siguiente comando:

```
node mongo-test.js
```

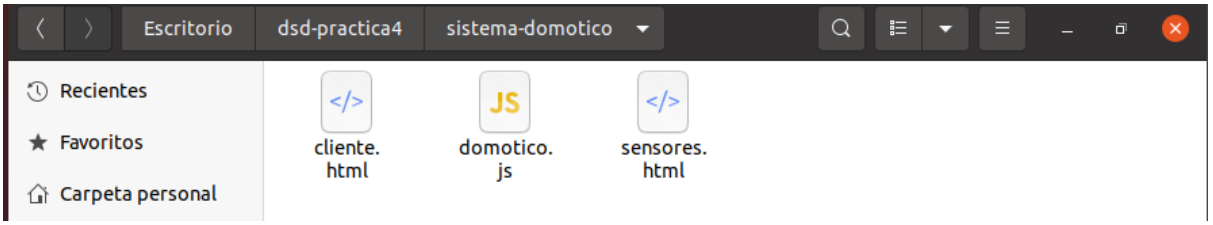




Segunda parte: Sistema domótico IoT

La última parte de la práctica pedía implementar un sistema domótico inteligente. Dicho sistema está compuesto por dos sensores (luminosidad y temperatura) y dos motores/actuadores (aire acondicionado y persiana) que modifican los valores de sendos sensores en un valor constante cada vez que se realiza una acción sobre ellos.

En cuanto a la implementación, el proyecto consiste en dos archivos html (*cliente.html* y *sensores.html*) y un archivo JavaScript (*domotico.js*). En este último es donde se implementan tanto el servidor que sirve las dos páginas html, como el agente (una función que realiza la lógica requerida dentro del servidor).



El código de todos los archivos viene comentado y detallado, especificando los detalles más relevantes. No obstante a continuación se explicará más en profundidad la función de cada uno de los archivos.

Para explicar el código del servidor *domotico.js*, se puede dividir el archivo en dos mitades. Una primera mitad que crea el servicio web, analizando y sirviendo el archivo que se solicite (*cliente* o *sensores*). Esta parte es muy básica, puesto que es similar a las de los ejemplos previamente explicados.

Sin embargo, es en la segunda mitad del código donde se implementa toda la lógica y procesamiento de eventos. El servidor guarda de forma local los valores de los umbrales de los sensores, así como los valores actuales.

```
// 15° <= temperatura <= 35°
const minTemperatura = 15;
const maxTemperatura = 35;
var temperatura = 25;

// 50 lux <= luminosidad <= 250 lux
const minLuminosidad = 50;
const maxLuminosidad = 250;
var luminosidad = 100;
```

Tras esto, el servidor se conecta al servicio de base de datos, inicia el servidor y establece la base de datos “pruebaBaseDatos” para utilizarla. Luego, al igual que en el ejemplo 5, comprueba si existe la colección “domótico” (y la crea si es necesario).

```
MongoClient.connect("mongodb://127.0.0.1:27017",                    {
useUnifiedTopology: true }, function(err, db) {
}).then((db) => {
      httpServer.listen(8080);
      var io = socketio(httpServer);

      var dbo = db.db("pruebaBaseDatos");
      // Comprueba si la coleccion domotico existe, y si no la crea
      dbo.listCollections({name:'domotico'})
      .next()
      .then((info) => {
            if(info) {
                  // La colección ya existe
                  console.log("La colección 'domotico' existe");
            } else {
                  // Si no existe, la crea
                  dbo.createCollection('test')
                  .then((collection) => {
                        console.log("Colección 'domotico' creada");
                  }).catch((err)=> {
                        console.log(err);
```

```
                })
            }
        }).catch((err)=> {
            console.log(err);
        });
```

A continuación, se definen varias funciones que se utilizarán a lo largo del programa. Todas ellas vienen comentadas, no obstante cabe destacar dos de ellas:

- insertarRegistro(): guarda los cambios de los sensores insertando un registro en la BD y envía el aviso de actualizar historial. Siempre que se actualiza alguno de los sensores es llamada.

```
// Funcion que utiliza la base de datos;
// Siempre que inserta nueva informacion en la bd envia un aviso de actualizar historial
// Para ello debe obtener información de todos los registros de la coleccion domotico
function insertarRegistro() {

    // Inserta el nuevo registro
    var data = {fecha:new Date(),temperatura:temperatura, luminosidad:luminosidad};
    dbo.collection('domotico').insertOne(data, {safe:true}).then((result) => {
        // console.log(data);
        console.log("Registrado nuevo cambio en los sensores: " + JSON.stringify(data));

    }).catch((err)=> {
        console.error(err);
    });

    // Vuelve a enviar todos los registros de la coleccion (actualizada)
    dbo.collection('domotico').find().toArray().then((results) => {
        io.sockets.emit('actualizarHistorial',results);
    }).catch((err) => {
        console.error(err)
    });
}
```

- agente(): implementa la lógica del agente básico que se pedía en la práctica. Analiza los valores de los sensores, emitiendo un aviso sobre su estado (OK o ALERTA). Además, si el sensor de luminosidad no está en los umbrales adecuados, actúa subiendo/bajando la persiana tantas veces como haga falta hasta establecer el sensor en un valor adecuado.

```
// Función que implementa la lógica del agente:
// - Comprueba los umbrales de los sensores
// - Crea y emite el aviso
//  - Modifica los sensores subiendo/bajando el motor(A/C o persiana) las veces necesarias hasta que quede en un umbral aceptable
function agente() {

	// Crea el aviso
	var aviso = "[Temperatura: ";
	if(minTemperatura<=temperatura&&temperatura<=maxTemperatura) {
		aviso += "OK, ";
	} else {
		aviso += "ALERTA, ";
	}

	aviso += "Luminosidad: ";
	if(minLuminosidad<=luminosidad&&luminosidad<=maxLuminosidad) {
		aviso += "OK]";
	} else {
		aviso += "ALERTA]";
	}

	// Enviar el aviso del agente
	io.sockets.emit('avisosAgente', aviso);

	// El agente ejecuta su funcion (si es necesario)
	var agente_actua = false;

	// Para la luminosidad, el agente actua si es necesario
	if(luminosidad<minLuminosidad) {
		agente_actua = true;
		console.log("Agente dice: ¡Demasiada oscuridad! Debemos subir la persiana");
		while(luminosidad<minLuminosidad) {
			console.log("Agente dice: He subido la persiana");
			subirPersiana();
		}
	} else if (luminosidad>maxLuminosidad) {
		agente_actua = true;
		console.log("Agente dice: ¡Demasiada luz! Debemos bajar la persiana");
		while(luminosidad<minLuminosidad) {
			console.log("Agente dice: He bajado la persiana");
			bajarPersiana();
		}
	}
```

```
	// Para la temperatura, el agente no realiza ninguna accion (lo pone en el guión de practicas)
	// Si se quisiera ejecutar alguna accion, se deberia hacer lo mismo que con la luminosidad


	// Si el agente ha actuado, entonces reenvia los datos actualizados
	if(agente_actua) {
		// Sólo envía el cambio final, no los cambios intermedios
		io.sockets.emit('actualizarSensores',{temperatura: temperatura, luminosidad: luminosidad});
		// Vuelve a llamar a la funcion para generar nuevamente el mensaje y volver a emitirlo
		agente();
	}
}
```

Por último, se definen los eventos en los que el servidor realiza alguna acción:

- *connection*: evento básico de conexión de un cliente. Implementa la funcionalidad del servidor para cada uno de los clientes.
- *disconnect*: evento básico de desconexión.
- *actualizarSensores*: evento utilizado para enviar la información actual de los sensores a los clientes conectados.

```
io.sockets.emit('actualizarSensores',{temperatura: temperatura, luminosidad: luminosidad});
```

- *actualizarHistorial*: evento por el cual el servidor envía los datos del historial a los clientes.
- *avisosAgente*: evento por el cual el servidor envía el aviso del agente a los clientes conectados.

```
io.sockets.emit('avisosAgente', aviso);
```

- Los demás eventos *cambiarTemperatura*, *cambiarLuminosidad*, *subirAC*, *bajarAC*, *subirPersiana*, *bajarPersiana* son solicitudes por parte de los clientes del servicio para modificar los valores de los sensores de una forma u otra. Estos eventos se procesan todos de la misma manera:
    1. Se realiza el cambio en el sensor
    2. Se inserta el cambio en la base de datos (*insertarRegistro()*, que a su vez emite el aviso de *actualizarHistorial*)
    3. Se envían los valores actualizados de los sensores a los clientes con el evento *actualizarSensores.*
    4. Se llama al agente para que procese los nuevos valores y aplique su lógica.

```
client.on('cambiarTemperatura', function(data) {
	temperatura = parseInt(data.temperatura);
```

```
      // Envía la nueva temperatura
      io.sockets.emit('actualizarSensores', {temperatura:
temperatura, luminosidad: luminosidad});
      // Inserta el cambio en la BD
      insertarRegistro();
      // Llama al agente para que cree el aviso y modifique los
valores si es necesario
      agente();
});
```

Para ejecutar el servidor, lanzar desde una terminal:

```
node domotico.js
```

Continuando con los demás archivos, *cliente.html* es una página web que se divide en tres secciones: una sección para mostrar los valores actuales de los sensores, otra sección con botones para poder subir/bajar los motores, y otra sección para mostrar los avisos del agente.

En cuanto a la lógica del cliente, crea un cliente socket y se conecta al servicio del servidor. Define las funciones con las cuales actualizará su página en tiempo real (por ejemplo, `actualizarAviso`, `actualizarSensores`…) así como las funciones para actuar sobre los motores (`subirAC()`, `bajarPersiana()` …) . Estas últimas emitirán un evento al servidor para que este active el motor correspondiente.

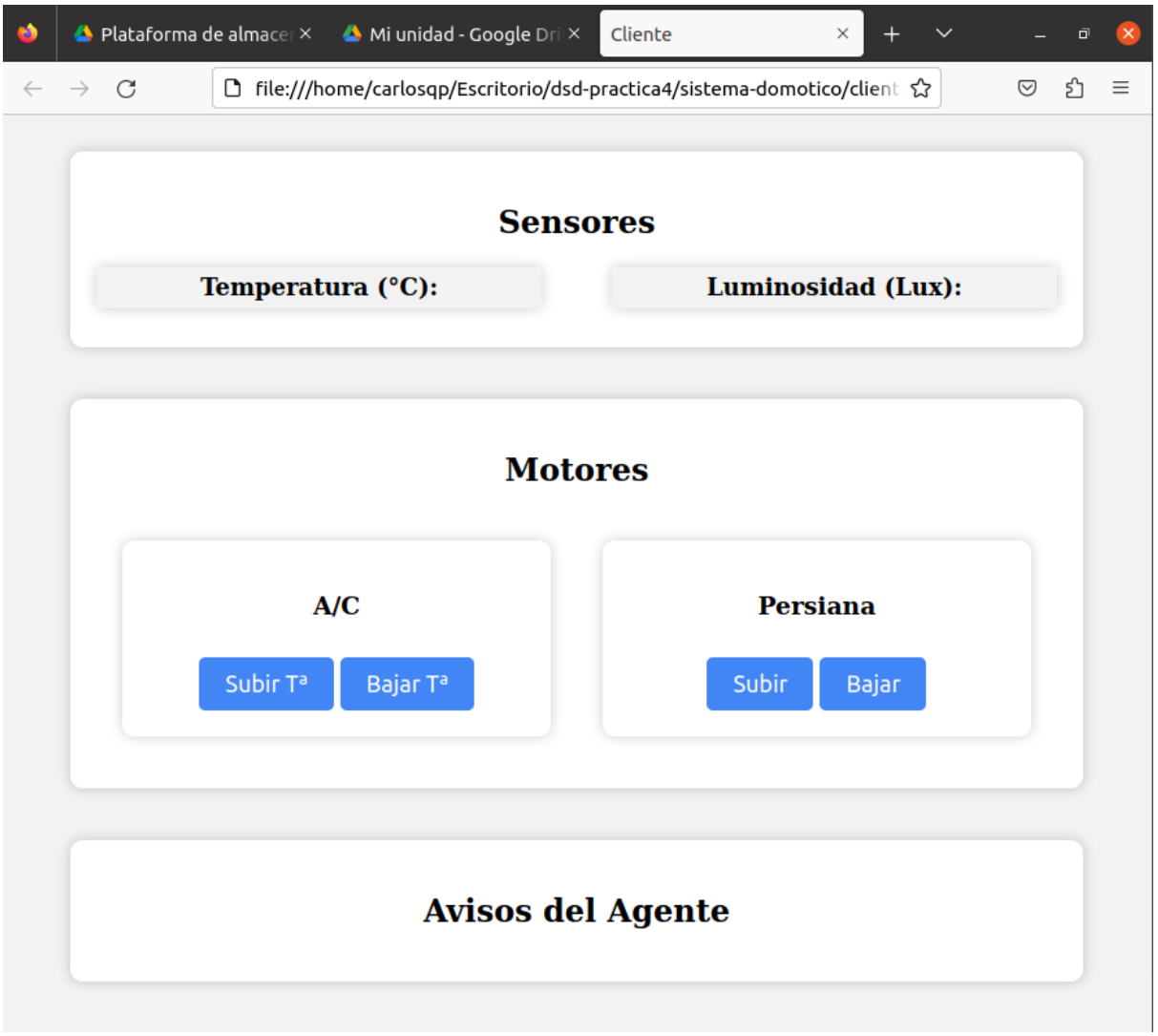
```
// Función para subir A/C -> envía un evento "subirAC" al servidor
    function subirAC() {
          if (confirm("Subir la temperatura del A/C implicará una
subida de la temperatura de la habitación.\n\n¿Estás seguro de que
quieres hacerlo?")) {
             // Si el usuario hace clic en "Sí", se ejecutará este
bloque de código
            socket.emit("subirAC", {});
            console.log("enviada peticion subirAC");
        }
    }
```

Una vez definidas dichas funciones, el cliente se suscribe/espera a recibir los siguientes eventos:

```
// Tratamiento de eventos con socket.io
// Al conectarse, muestra por consola que se ha conectado al
servidor
socket.on('connect', function(){
    console.log("Conectado al servicio " + serviceURL);
});
```

```
// Cada vez que se actualicen los sensores, el cliente actualiza su valor
socket.on('actualizarSensores', function(data) {
     // Llamamos la método actualizarSensores que es el encargado de modificar el HTML
    actualizarSensores(data);
});

// Para recibir mensajes de aviso del agente
socket.on("avisosAgente", function(data) {
    actualizarAviso(data);
});

// Al desconectarse, muestra por consola que se ha desconectado
socket.on('disconnect', function(){
    console.log("Desconectado del servicio " + serviceURL);
});
```

A este recurso se accede por medio de la URL http://localhost:8080/cliente



En cuanto a sensores.html, esta página simula la funcionalidad de un administrador. A diferencia de la página del cliente, aquí se puede modificar y establecer directamente el valor de los sensores. Se divide también en secciones: una para cambiar los valores de los sensores, otra para mostrar los avisos del agente, y otra para mostrar el histórico de cambios en los sensores.

En cuanto a la lógica, es muy similar a la del cliente. Crea un cliente socket y se conecta al servicio del servidor. Define las funciones con las cuales actualizará su página en tiempo real (por ejemplo, `actualizarAviso`, `actualizarSensores…`), y, a diferencia del cliente, esta vez se crean event listenners para los envíos de formularios (esto es debido a que cuando se envía un formulario, por defecto carga de nuevo la página, y no queremos eso), que realizarán el envío del evento correspondiente (`'cambiarTemperatura'`, `'cambiarLuminosidad'`) al servidor para que lo procese.

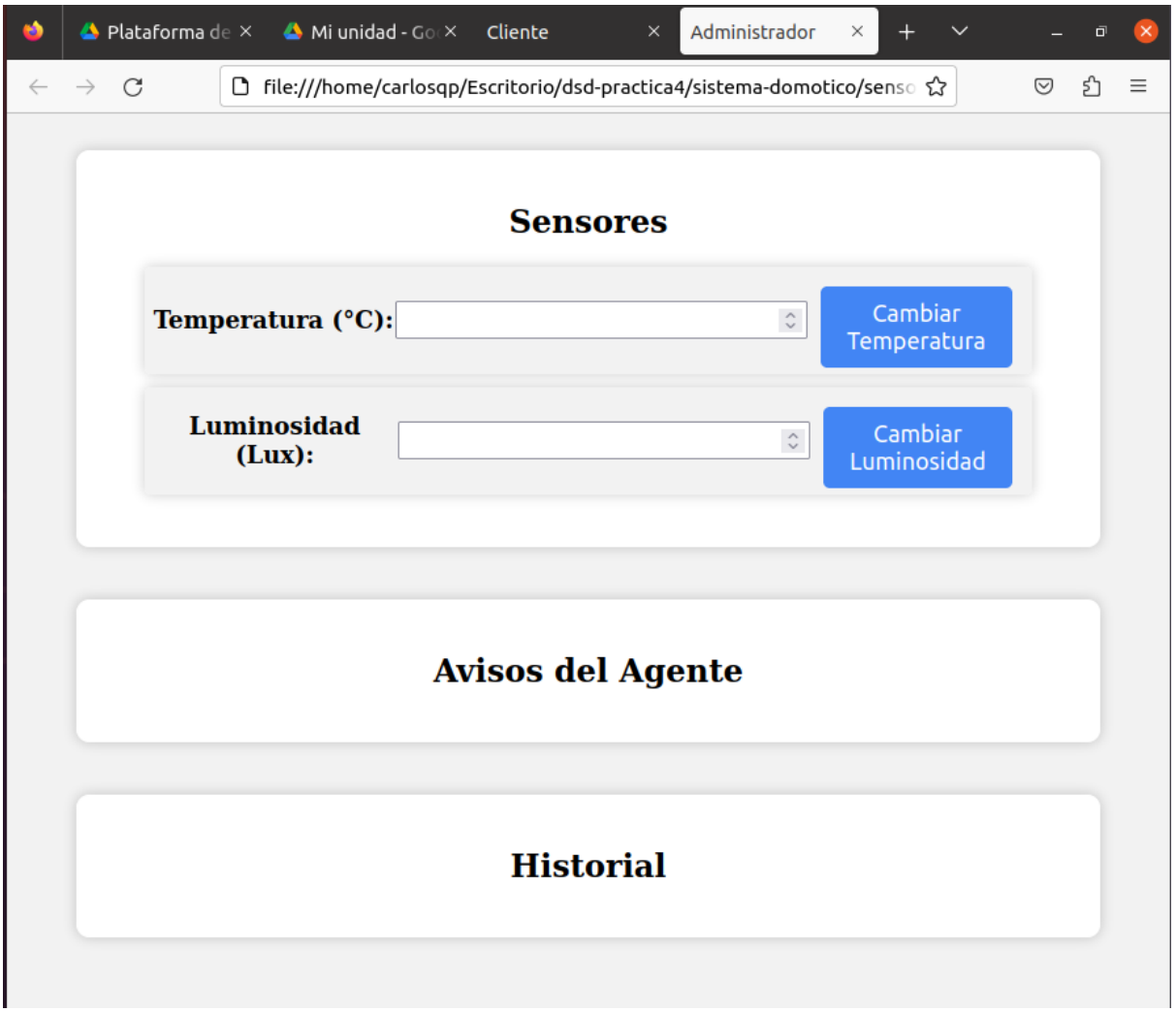
```
document.getElementById('formTemperatura').addEventListener('submit'
, function(event) {
            event.preventDefault(); // Evita la recarga de la página
para recibir los avisos
                    if (confirm("¿Seguro que desea cambiar la
temperatura?")) {
                  // Si el usuario hace clic en "Sí", se ejecutará
este bloque de código
                                                    var input =
document.getElementById('temperatura');
                var valorIntroducido = input.value;
                    socket.emit("cambiarTemperatura", {temperatura:
valorIntroducido});
                     console.log("enviada peticion de cambiar la
temperatura ");
            }
        });
```

Una vez definidas dichas funciones y listenners, el cliente se suscribe/espera a recibir los siguientes eventos:

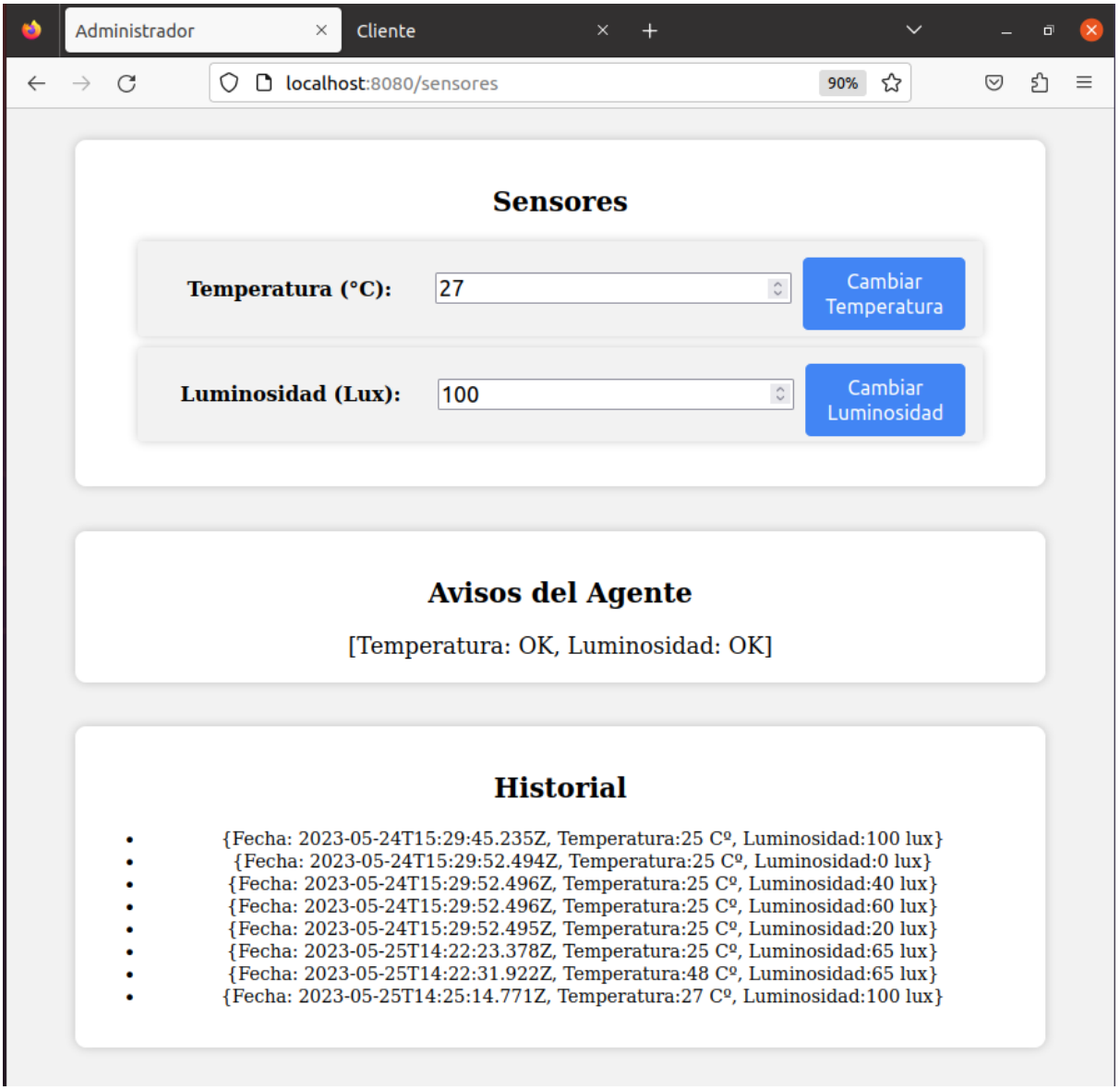
```
/// Tratamiento de eventos con socket.io
// Al conectarse, muestra por consola que se ha conectado al
servidor
socket.on('connect', function(){
    console.log("Conectado al servicio " + serviceURL);
});

// Cada vez que se actualicen los sensores, el cliente actualiza
su valor
socket.on('actualizarSensores', function(data) {
     // Llamamos la método actualizarSensores que es el encargado
de modificar el HTML
```

```
    actualizarSensores(data);
});

// Para recibir mensajes de aviso del agente
socket.on("avisosAgente", function(data) {
    actualizarAviso(data);
});

// Para recibir informacion del historial
socket.on('actualizarHistorial', function(data) {
    actualizarHistorial(data);
})

// Al desconectarse, muestra por consola que se ha desconectado
socket.on('disconnect', function(){
    console.log("Desconectado del servicio " + serviceURL);
});
```

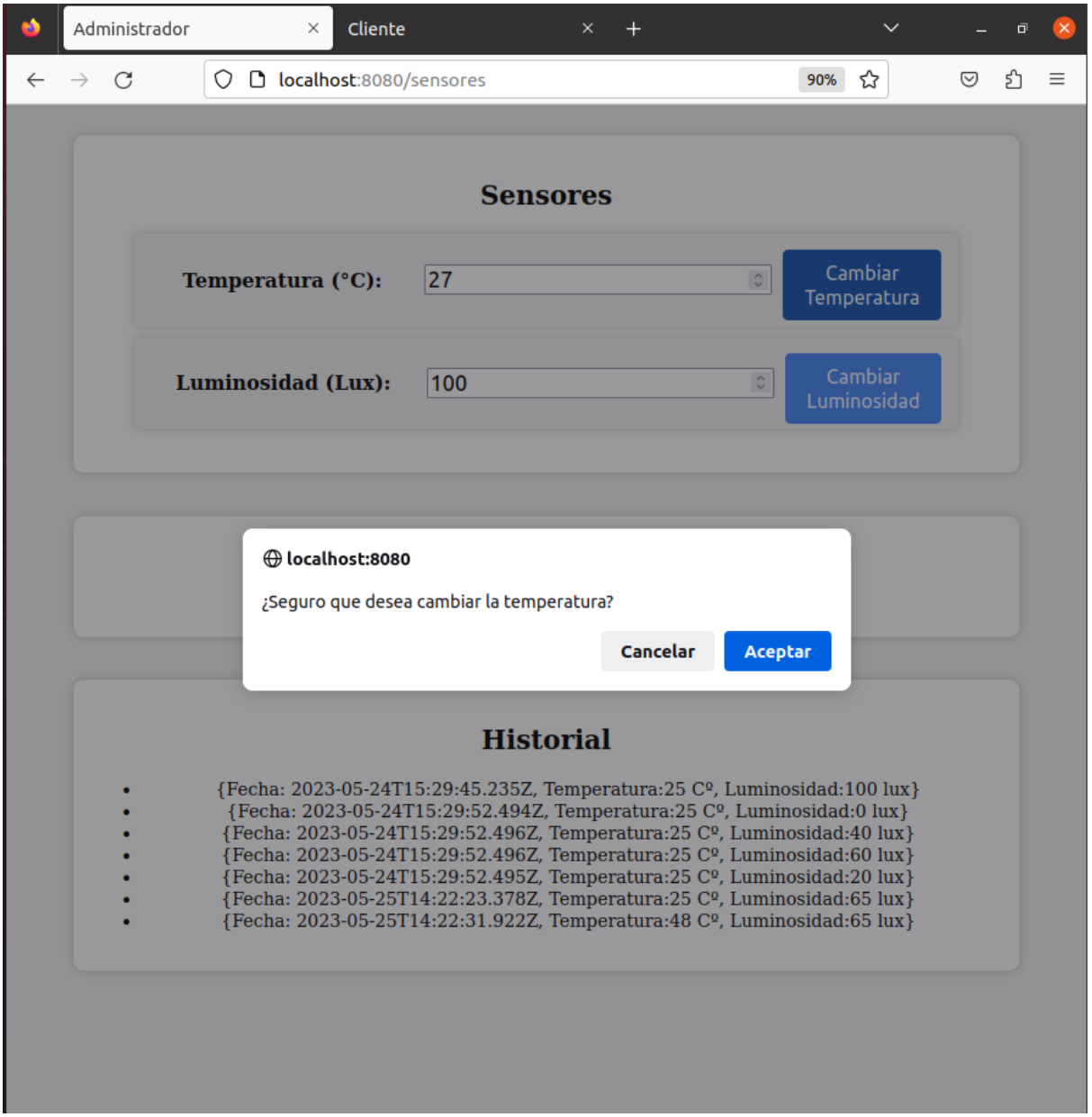
A este recurso se accede por medio de la URL http://localhost:8080/sensores

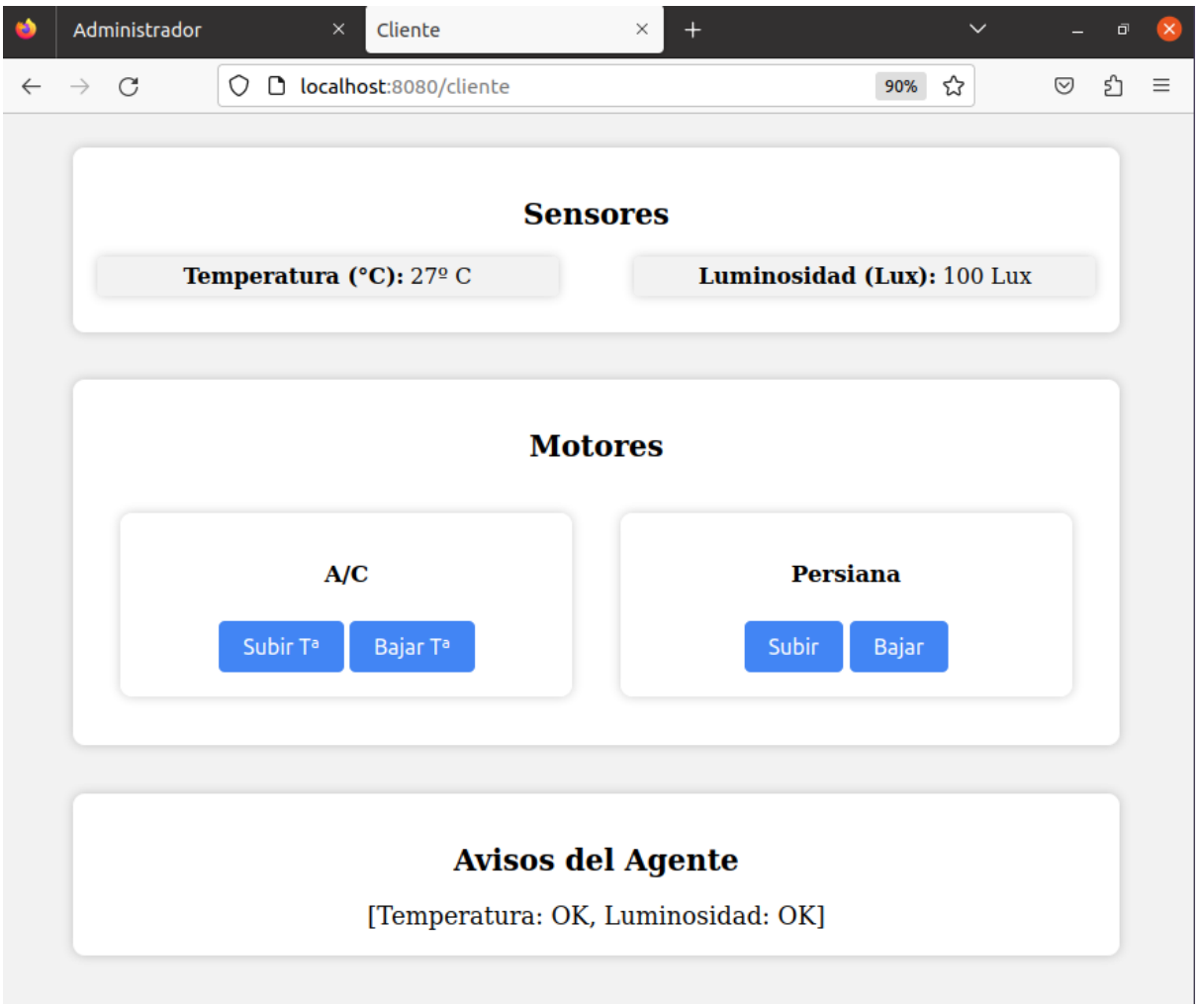


A continuación se muestran algunas capturas de pantalla del programa en ejecución:

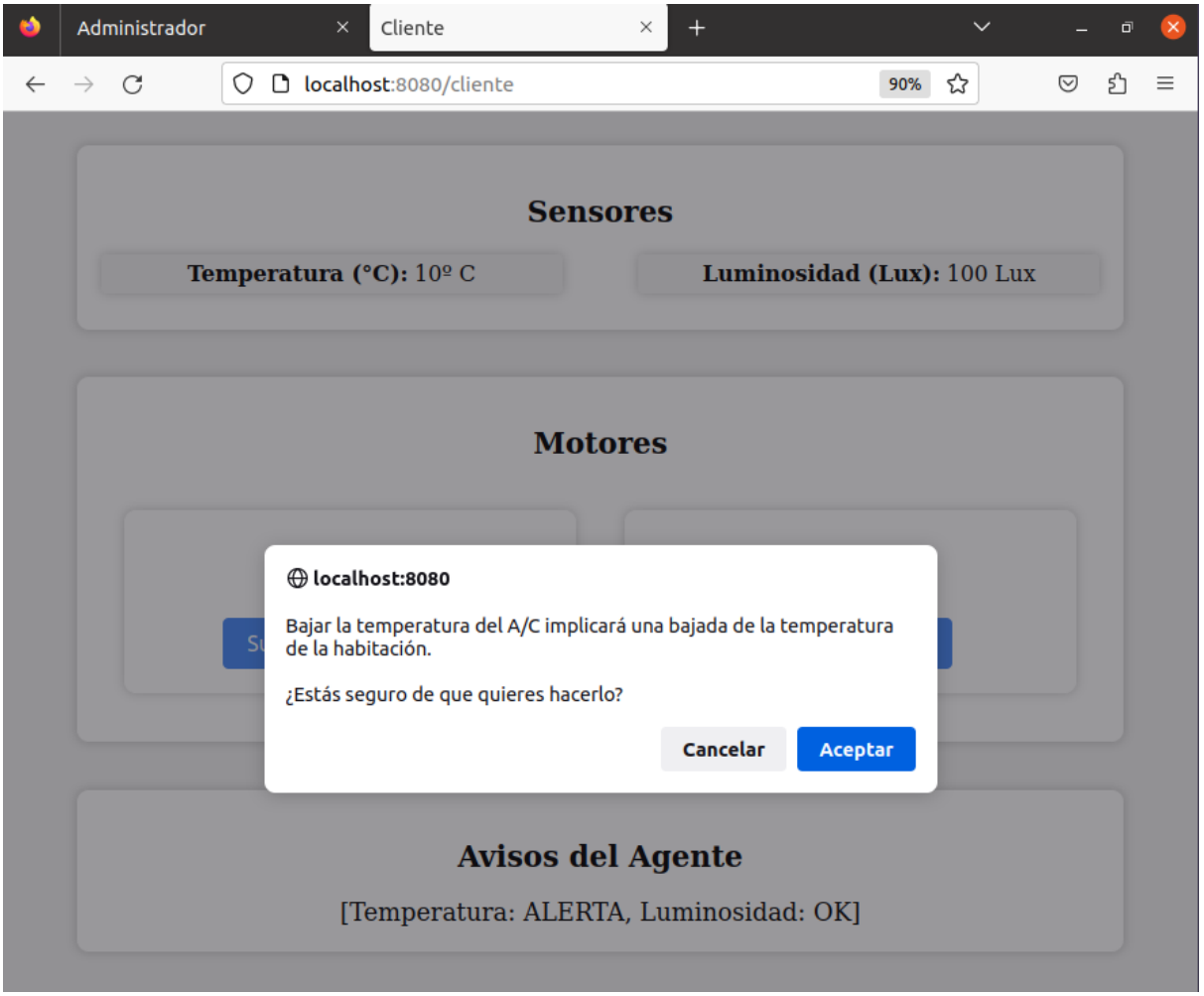


Interfaz del administrador(sensores.html)


Administrador
Cliente
localhost:8080/sensores
90%
Sensores
Temperatura (°C):
27
Cambiar Temperatura
Luminosidad (Lux):
100
Cambiar Luminosidad
localhost:8080
¿Seguro que desea cambiar la temperatura?
Cancelar
Aceptar
Historial
{Fecha: 2023-05-24T15:29:45.235Z, Temperatura:25 Cº, Luminosidad:100 lux}
{Fecha: 2023-05-24T15:29:52.494Z, Temperatura:25 Cº, Luminosidad:0 lux}
{Fecha: 2023-05-24T15:29:52.496Z, Temperatura:25 Cº, Luminosidad:40 lux}
{Fecha: 2023-05-24T15:29:52.496Z, Temperatura:25 Cº, Luminosidad:60 lux}
{Fecha: 2023-05-24T15:29:52.495Z, Temperatura:25 Cº, Luminosidad:20 lux}
{Fecha: 2023-05-25T14:22:23.378Z, Temperatura:25 Cº, Luminosidad:65 lux}
{Fecha: 2023-05-25T14:22:31.922Z, Temperatura:48 Cº, Luminosidad:65 lux}



Interfaz del cliente (cliente.html)




```
carlosqp@dsd-server:~/Escritorio/dsd-practica4/sistema-domotico$ node domotico.js
Servicio Web + MongoDB + Socket.io iniciado
La colección 'domotico' existe
Peticion invalida: /favicon.ico
El usuario ::ffff:127.0.0.1 se ha conectado
El usuario ::ffff:127.0.0.1 se ha conectado
Registrado nuevo cambio en los sensores: {"fecha":"2023-05-25T14:25:14.771Z","temperatu
ra":27,"luminosidad":100,"_id":"646f6fca365c1c1061cc7b2b"}
Registrado nuevo cambio en los sensores: {"fecha":"2023-05-25T14:26:22.064Z","temperatu
ra":10,"luminosidad":100,"_id":"646f700e365c1c1061cc7b2c"}
Registrado nuevo cambio en los sensores: {"fecha":"2023-05-25T14:26:48.157Z","temperatu
ra":8,"luminosidad":100,"_id":"646f7028365c1c1061cc7b2d"}
Agente dice: ¡Demasiada oscuridad! Debemos subir la persiana
Agente dice: He subido la persiana
Registrado nuevo cambio en los sensores: {"fecha":"2023-05-25T14:27:59.847Z","temperatu
ra":8,"luminosidad":40,"_id":"646f706f365c1c1061cc7b2e"}
Registrado nuevo cambio en los sensores: {"fecha":"2023-05-25T14:27:59.849Z","temperatu
ra":8,"luminosidad":60,"_id":"646f706f365c1c1061cc7b2f"}
```

Salida del servidor+agente