

Práctica 4: Node.JS

28 de mayo del 2021

Introducción

Esta memoria se dividirá en dos partes, en la primera implementaremos los ejemplos proporcionados en el guión, probaremos y explicaremos su funcionamiento además de comentar las diferencias existentes entre estos.

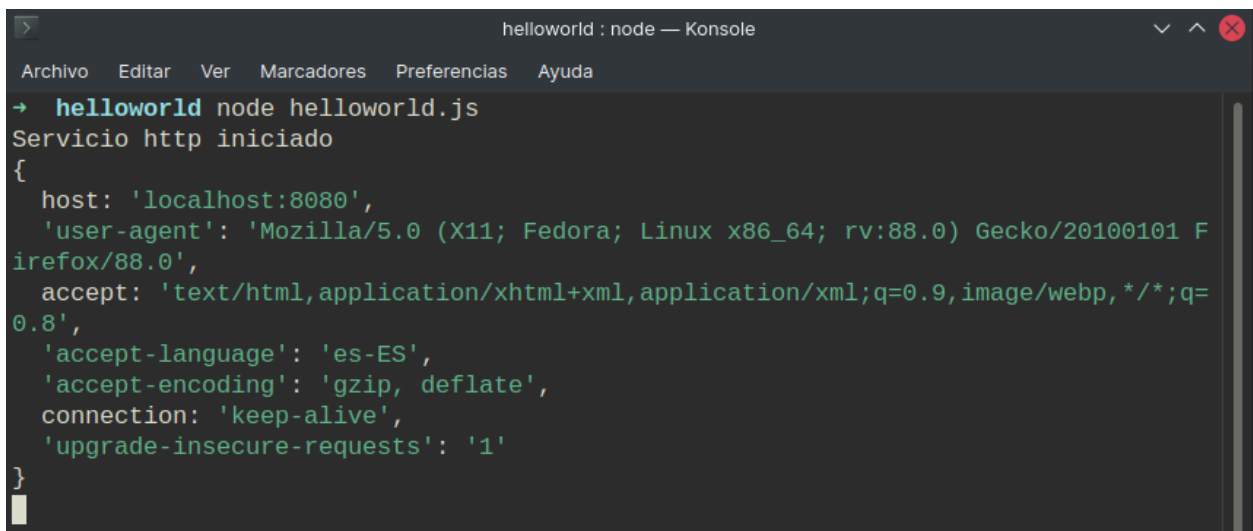
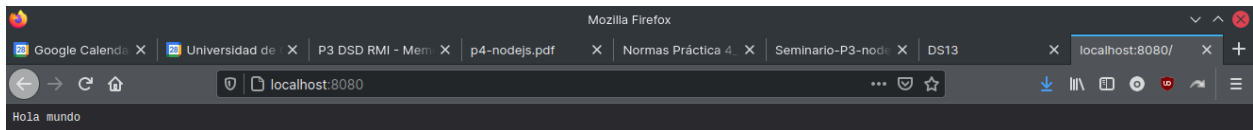
En la segunda parte implementaremos un servidor domótico con distintos componentes como sensores, actuadores y un agente que simulará lo que podría ser el comportamiento de una casa inteligente con dispositivos IoT.

Introducción	1
1ª Parte: Implementación de ejemplos	3
Ejemplo 1: Hello World!	3
Ejemplo 2: Calculadora básica	3
Ejemplo 3: Calculadora con interfaz	4
Ejemplo 4: Connections con Socket.io	6
Ejemplo 5: Registro de accesos con MongoDB	9
Problemas encontrados	10
2ª Parte: Sistema Domótico	11
Descripción	11
Arquitectura de la aplicación	11
Implementación y problemas encontrados	12
Servidor	12
Introducción	13
Implementación	13
Problemas encontrados	16
Sensores	16
Introducción	17
Implementación de la API de OpenWeatherMap	17
Métodos utilizados	18
Problemas encontrados	19
Por suerte, se nos permite acceder a la geolocalización accediendo al servidor desde localhost.	20
Agente	20
Implementación	21
Panel de usuario	22
Implementación	23
Diseño	25
Instrucciones de uso	25
Prueba de uso	27

1ª Parte: Implementación de ejemplos

Ejemplo 1: Hello World!

Empezamos con un sencillo Hola mundo, para ello, utilizamos el código proporcionado en el guión, ejecutamos “*node helloworld.js*” en la terminal y accedemos a localhost:8080.



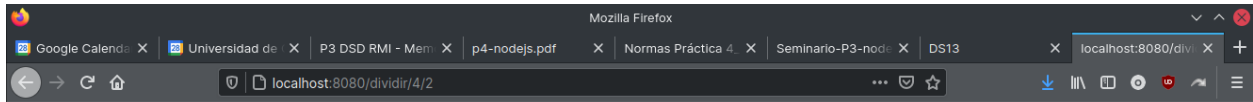
El código de este ejemplo simplemente crea un servidor http y le asigna una función a realizar cuando reciba una petición. Esta función muestra por consola la cabecera de la petición recibida y genera una respuesta para el cliente con el texto plano “Hola mundo”.

Por último, se pone a escuchar al servidor en el puerto 8080 con la orden “*httpServer.listen(8080)*”.

Como hemos podido comprobar, la ejecución del servidor no termina al llegar al final del código. Este seguirá sirviendo peticiones mientras existan puertos del sistema operativo en uso.

Ejemplo 2: Calculadora básica

En este ejemplo se implementa una calculadora básica distribuida que usa una interfaz tipo REST. Esta calculadora tomará los parámetros de la URL que solicite el cliente, es decir, para solicitar el cálculo de 4/2, el cliente tendrá que acceder a <http://localhost:8080/dividir/4/2>.



2

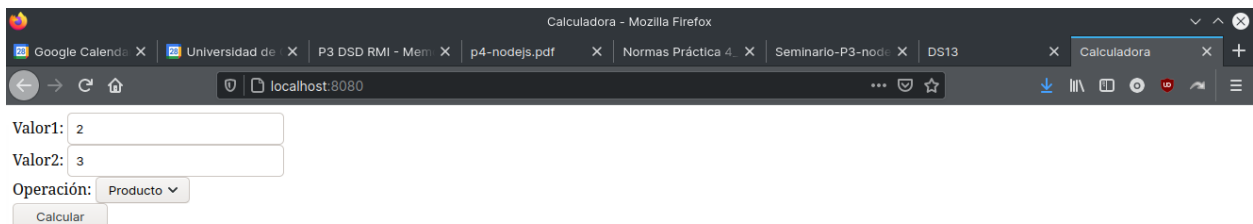
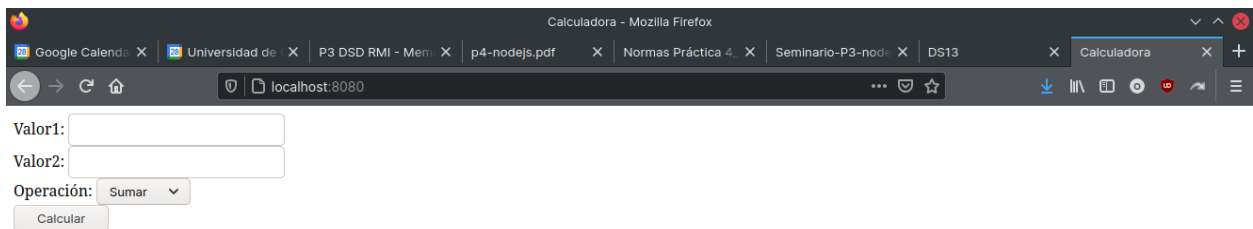
Para la implementación, se crea una función simple “*calcular(operación, val1, val2)*” para realizar los cálculos (suma, resta multiplicación y división) y se vuelve a crear un servidor http.

Este servidor ha de ser capaz de obtener los tres parámetros de la URL, procesar el cálculo mediante la función anterior y escribir la solución en la respuesta al cliente.

Ejemplo 3: Calculadora con interfaz

En este ejemplo se realiza una mejora sobre el anterior. Mejoraremos la experiencia del usuario gracias a la inclusión de una interfaz web implementada en html para la petición de los cálculos al servidor.

Podemos observar su funcionamiento a continuación:



6

```
→ calculadora node calculadora-web.js
Servicio HTTP iniciado
Petición REST: producto/2/3
```

El código de este ejemplo incluye algunas novedades que comentaremos a continuación.

En primer lugar, definimos una serie de MIME types para indicarle al cliente el formato con el que el servidor le va a responder. En este ejemplo utilizaremos el tipo “text/html” para generar una respuesta en formato html.

En este ejemplo se hace uso del módulo “fs” para poder cargar el .html que contiene la interfaz de la calculadora y servirlo al cliente.

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

Si el .html de la interfaz no está disponible, se pasa al ejemplo del funcionamiento anterior, esperando obtener los parámetros para el cálculo de la URL de la primera petición.

El .html que se sirve al cliente contiene un formulario con los parámetros para el cálculo y una función de javascript encargada de enviar los datos introducidos al servidor para su cálculo.

```
<script type="text/javascript">
  var serviceURL = document.URL;
  function enviar() {
    var val1 = document.getElementById("val1").value;
    var val2 = document.getElementById("val2").value;
    var oper = document.getElementById("operacion").value;

    var url = serviceURL+"/"+oper+"/"+val1+"/"+val2;
    var httpRequest = new XMLHttpRequest();
    httpRequest.onreadystatechange = function() {
      if (httpRequest.readyState === XMLHttpRequest.DONE){
        var resultado = document.getElementById("resul1");
        resultado.innerHTML = httpRequest.responseText;
      }
    };
    httpRequest.open("GET", url, true);
    httpRequest.send(null);
  }
</script>
```

En esta función podemos ver cómo se obtienen dichos parámetros del formulario, para proceder a generar una petición al servidor con XMLHttpRequest().

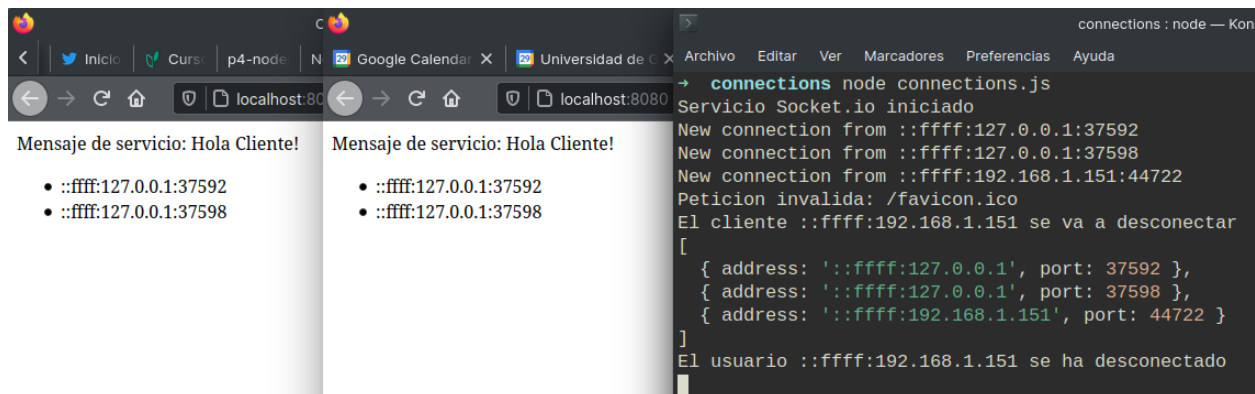
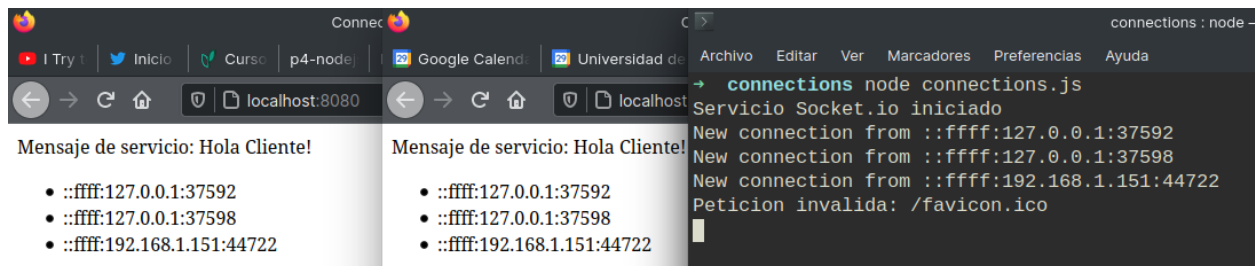
Esto lo consigue obteniendo los parámetros para el cálculo del formulario rellenado por el usuario y generando una petición a la URL del servidor como sucedía en el primer ejemplo de calculadora, es decir, accediendo a dirección/operación/val1/val2.

Por último, con el `if (httpRequest.readyState === XMLHttpRequest.DONE)` establecemos que cuando se reciba una respuesta a la petición, se actualice el contenedor html destinado al resultado del cálculo con la respuesta del servidor.

Ejemplo 4: Connections con Socket.io

En este ejemplo haremos uso de los sockets para implementar un modelo publish-subscribe en el que sea posible notificar a un cliente de todos los clientes conectados actualmente sin que éste lo haya solicitado. Para ello, definiremos una serie de eventos a los cuales el servidor o el cliente estarán suscritos mediante la librería Socket.io.

Probamos el ejemplo conectando dos ventanas del navegador al servidor además de un dispositivo móvil que probaremos a desconectar.



El servidor ("connections.js") comienza sirviendo un .html al cliente como en los ejemplos anteriores y a continuación, comienza a poner en marcha los sockets.

```
42 var allClients = new Array();
43 io.sockets.on('connection',
44   function(client) {
45     allClients.push({address:client.request.connection.remoteAddress, port:client.request.connection.remotePort});
46     console.log('New connection from ' + client.request.connection.remoteAddress + ':' + client.request.connection.remotePort);
47     io.sockets.emit('all-connections', allClients);
48     client.on('output-evt', function (data) {
49       client.emit('output-evt', 'Hola Cliente!');
50     });
51     client.on('disconnect', function() {
52       console.log("El cliente "+client.request.connection.remoteAddress+" se va a desconectar");
53       console.log(allClients);
54       var index = -1;
55       for(var i = 0; i<allClients.length;i++){
56         //console.log("Hay "+allClients[i].port);
57         if(allClients[i].address == client.request.connection.remoteAddress
58           && allClients[i].port == client.request.connection.remotePort){
59           index = i;
60         }
61       }
62     }
63     if (index != -1) {
64       allClients.splice(index, 1);
65       io.sockets.emit('all-connections', allClients);
66     }else{
67       console.log("EL USUARIO NO SE HA ENCONTRADO!")
68     }
69     console.log('El usuario '+client.request.connection.remoteAddress+' se ha desconectado!');
70   });
71 }
72 );
73
```

Podemos ver que el servidor se suscribe al evento 'connection' y define una función para cuando esto suceda, es decir, para cuando un cliente se conecte al servidor. En esta función se añade la dirección y el puerto del cliente al array allClients, se muestra por terminal dicha información del cliente y se envía el array a todos los clientes suscritos al evento 'all-connections' mediante

```
io.sockets.emit('all-connections', allClients);
```

A continuación, el servidor se suscribe al evento 'output-evt'. Al recibirlo, se le comunicará al cliente remitente un evento del mismo tipo con el mensaje 'Hola Cliente!'.

Por último, el servidor se suscribe al evento 'disconnect' que se produce cuando un cliente se desconecta del servidor. Cuando esto ocurre, será necesario eliminar al cliente del array de allClients y comunicar al resto de clientes esta desconexión.

En el lado del cliente (connections.html) se definen una serie de eventos a los que suscribirse:

- **connect:** Al establecer conexión con el servidor envía a este un mensaje con el contenido 'Hola Servicio!'.
- **output-evt:** Al recibir un mensaje de este tipo del servidor, muestra su contenido por pantalla.
- **all-connections:** Este es el evento asociado al listado con todos los clientes conectados al servicio. Muestra por pantalla este listado.
- **disconnect:** Cuando se produzca una desconexión del servidor, notifica al cliente mostrando por pantalla el mensaje 'El servicio ha dejado de funcionar!!'.

Ejemplo 5: Registro de accesos con MongoDB

En este ejemplo crearemos un registro de accesos al servidor cuyas entradas serán almacenadas en una base de datos en MongoDB. También se utilizará socket.io para la suscripción de eventos entre cliente y servidor.

Probemos el ejemplo accediendo varias veces al servidor desde una pestaña del navegador y desde un dispositivo móvil.

The screenshot shows a web browser window on the left and a terminal window on the right. The browser window displays a list of JSON objects representing access logs. The terminal window shows the MongoDB command prompt with a query to find all documents in the 'pruebaBaseDatos' database.

```

{"_id":"60b21be140d3490bd3ace5fc","host":"::ffff:127.0.0.1","port":37950,"time":"2021-05-29T10:48:01.673Z"}
{"_id":"60b21be440d3490bd3ace5fd","host":"::ffff:127.0.0.1","port":37950,"time":"2021-05-29T10:48:04.648Z"}
{"_id":"60b21be640d3490bd3ace5fe","host":"::ffff:127.0.0.1","port":37950,"time":"2021-05-29T10:48:06.331Z"}
{"_id":"60b21beb40d3490bd3ace5ff","host":"::ffff:192.168.1.151","port":44810,"time":"2021-05-29T10:48:09.998Z"}
{"_id":"60b21bf240d3490bd3ace600","host":"::ffff:192.168.1.151","port":44796,"time":"2021-05-29T10:48:16.786Z"}
{"_id":"60b21bf540d3490bd3ace601","host":"::ffff:192.168.1.151","port":44794,"time":"2021-05-29T10:48:19.757Z"}
{"_id":"60b21bf840d3490bd3ace602","host":"::ffff:127.0.0.1","port":37960,"time":"2021-05-29T10:48:22.729Z"}

```

```

> use pruebaBaseDatos
switched to db pruebaBaseDatos
> db.test.find().pretty()
{
  "_id" : ObjectId("60b21be140d3490bd3ace5fc"),
  "host" : "::ffff:127.0.0.1",
  "port" : 37950,
  "time" : "2021-05-29T10:48:01.673Z"
}
{
  "_id" : ObjectId("60b21be440d3490bd3ace5fd"),
  "host" : "::ffff:127.0.0.1",
  "port" : 37950,
  "time" : "2021-05-29T10:48:04.648Z"
}
{
  "_id" : ObjectId("60b21be640d3490bd3ace5fe"),
  "host" : "::ffff:127.0.0.1",
  "port" : 37950,
  "time" : "2021-05-29T10:48:06.331Z"
}
{
  "_id" : ObjectId("60b21beb40d3490bd3ace5ff"),
  "host" : "::ffff:192.168.1.151",
  "port" : 44810,
  "time" : "2021-05-29T10:48:09.998Z"
}
{
  "_id" : ObjectId("60b21bf240d3490bd3ace600"),
  "host" : "::ffff:192.168.1.151",
  "port" : 44796,
  "time" : "2021-05-29T10:48:16.786Z"
}
{
  "_id" : ObjectId("60b21bf540d3490bd3ace601"),
  "host" : "::ffff:192.168.1.151",
  "port" : 44794,
  "time" : "2021-05-29T10:48:19.757Z"
}
{
  "_id" : ObjectId("60b21bf840d3490bd3ace602"),
  "host" : "::ffff:127.0.0.1",
  "port" : 37960,
  "time" : "2021-05-29T10:48:22.729Z"
}

```

En el código de este ejemplo se hace uso de los eventos 'connection', 'my-address', 'poner', 'obtener' y 'disconnect'.

Empezando por el servidor (mongo-test.js), se inicia el servidor http y a continuación se empieza a trabajar con Mongo y socket.io.

```
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.port});
53         client.on('poner', function (data) {
54           collection.insertOne(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 });
```

Vemos que se obtiene una referencia a la BD de mongo de nombre “pruebaBaseDatos” y se crea una colección test dentro de esta. A continuación, se define una función a realizar cuando se conecte un cliente. En esta función se emite al cliente recién conectado su dirección y puerto y el servidor se suscribe a dos eventos del cliente:

- **poner:** Este es el evento que comunica el cliente cuando desea almacenar un dato en la BD del servidor, así que se insertará en la colección de ‘test’ la información enviada por el cliente.
- **obtener:** Este es el evento que comunica el cliente cuando desea acceder a algún dato de la BD del servidor. Al recibirlo se buscará en la colección la dirección aportada por el cliente y le será emitido un evento ‘obtener’ con los resultados de la consulta.

El cliente (mongo-test.html) actuará ante los siguientes eventos:

- **my-address:** Se emitirá al servidor el host, puerto y timestamp del acceso a este pidiendo que sea almacenado en la BD con el evento ‘poner’. A continuación, intentaremos obtener el acceso que acaba de ser almacenado mediante un evento ‘obtener’.
- **obtener:** Si el servidor responde emitiendo un evento ‘obtener’, el cliente actualizará su lista de accesos en pantalla con el listado recibido.
- **disconnect:** Cuando el servicio caiga, el listado de accesos del cliente será vaciado.

Problemas encontrados

El código de este ejemplo genera una excepción cuando la colección que se utiliza, 'test' ya ha sido creada anteriormente. Para replicar el problema es tan sencillo como iniciar el ejemplo por primera vez, cerrar el servidor e intentar volverlo a abrir. Al intentar establecer una conexión aparece lo siguiente

```
+ mongo node mongo-test.js
Servicio MongoDB iniciado
/home/sergiogarcia/Universidad/Tercero/SegundoCuatri/DSD/Prácticas/P4/mongo
/mongo-test.js:54
                                collection.insertOne(data, {safe:true}, fun
ction(err, result) {});
                                ^
TypeError: Cannot read property 'insertOne' of undefined
    at Socket.<anonymous> (/home/sergiogarcia/Universidad/Tercero/SegundoCu
atri/DSD/Prácticas/P4/mongo/mongo-test.js:54:16)
    at Socket.emit (events.js:376:20)
    at Socket.emitUntyped (/home/sergiogarcia/node_modules/socket.io/dist/t
yped-events.js:69:22)
    at /home/sergiogarcia/node_modules/socket.io/dist/socket.js:428:39
    at processTicksAndRejections (internal/process/task_queues.js:77:11)
```

Parece ser, que al existir ya dicha colección, la variable collection de `dbo.createCollection("test", function(err, collection)` se queda undefined. El servidor vuelve a funcionar al realizar `"db.test.drop()"` en el shell de mongo.

Para solucionar esto, será necesario comprobar `"err"` para ver si la colección ya existe y en ese caso, utilizar la ya existente en lugar de crear una nueva.

2ª Parte: Sistema Domótico

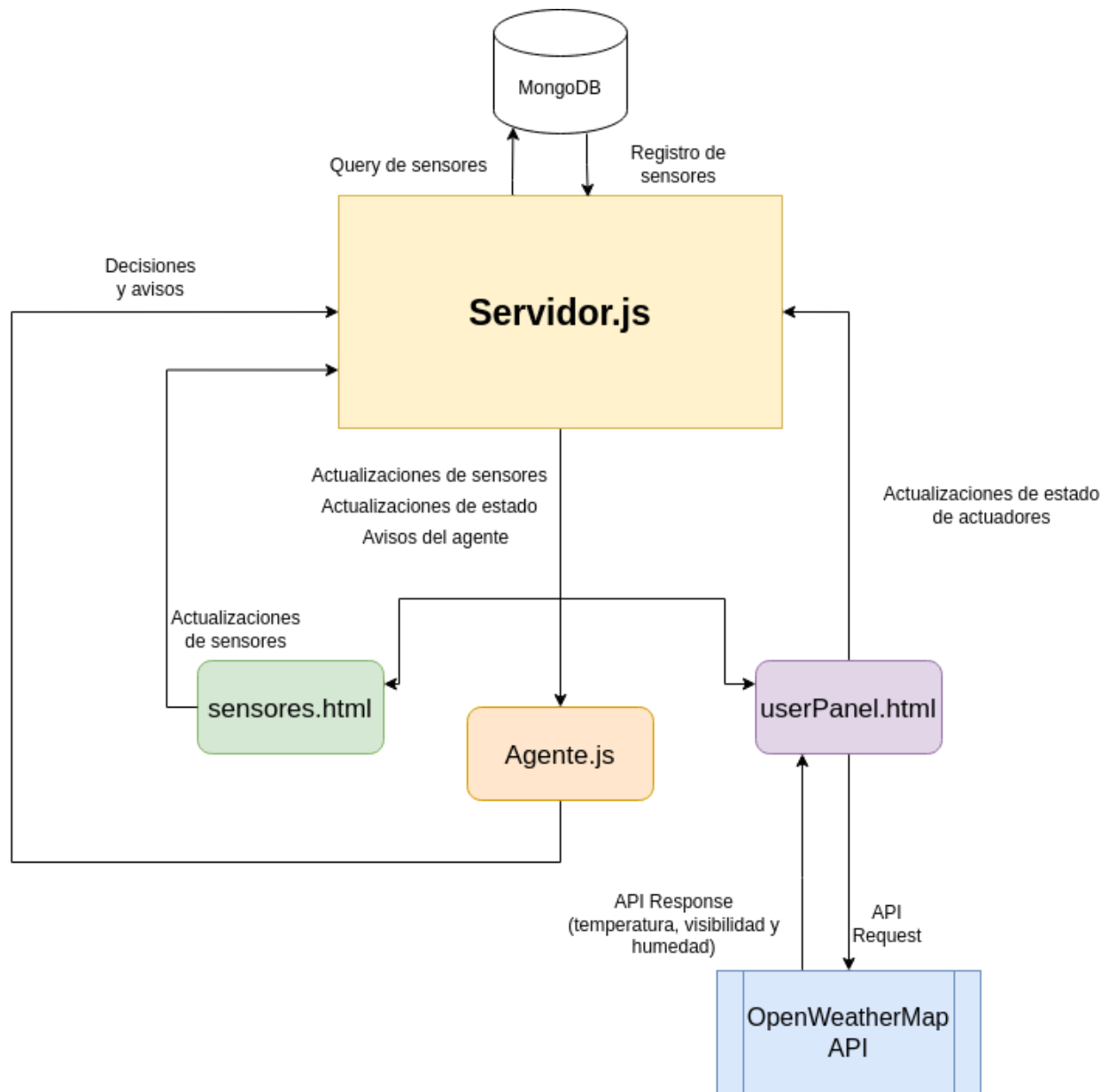
Descripción

Para esta parte de la práctica se ha implementado un sistema domótico con la funcionalidad especificada en el guión de la práctica, haciendo uso de Node.JS, Socket.IO y MongoDB.

Además, la funcionalidad del sistema ha sido extendida con mejoras como la posibilidad de los sensores de medir la humedad en el ambiente, el uso de la API de OpenWeatherMap en conjunto con la API de geolocalización de HTML para obtener lecturas meteorológicas en tiempo real de la ubicación del usuario, eventos complejos e incluso la creación de un diseño atractivo para la interfaz de las páginas del servicio.

Arquitectura de la aplicación

Podemos ver cómo se organiza la aplicación a grandes rasgos en este diagrama:



Implementación y problemas encontrados

Servidor

Introducción

El servidor se encargará de servir los .html que le sean solicitados (formulario de sensores y panel de control de usuario) además de gestionar las suscripciones a eventos y almacenar tanto el estado actual del sistema con respecto a sus dispositivos IoT como un histórico de las mediciones generadas por los sensores. Esto último se realizará en una base de datos en MongoDB.

Implementación

Antes de empezar, es necesario hablar de los módulos utilizados. Necesitaremos los siguientes:

- **http:** Para crear el servidor http.
- **url:** Para obtener la url y la ruta a la que está accediendo el cliente.
- **fs:** Para realizar operaciones con el sistema de archivos. En este caso, se quiere obtener de disco el .html a servir al cliente.
- **path:** Nos ayudará con la gestión de rutas del sistema de archivos y se usará en conjunto con fs.
- **socket.io:** Este módulo es el que nos permite toda la gestión de eventos y suscripciones del sistema. Gracias a esto, servidor, agente y usuarios podrán comunicarse entre sí y hacer posible este sistema.
- **mongodb:** Es el driver de mongodb para NodeJS. Nos permitirá tener acceso a una base de datos en la que almacenar los registros de las mediciones de los sensores y además, que estas no se pierdan ante una posible caída del servidor.

En servidor.js, se empieza creando el servidor http. El proceso para esto es el mismo que el observado en los ejemplos anteriores de la primera mitad de la práctica, con la única diferencia de que para facilitar el acceso al panel de control del usuario, se ha establecido que esto se pueda hacer accediendo a /userpanel con `else if (uri=="/userpanel") uri = "/userPanel.html";`

A continuación, se define un estado por defecto para los dispositivos del hogar. En este estado, la persiana estará bajada y el aire acondicionado estará apagado y configurado para funcionar en modo frío.

Como hemos mencionado anteriormente, el servidor necesitará establecer conexión con la base de datos, esto lo conseguimos con:

```

43 var estado = {persiana:false, ac:{on:false, modo:"Frío"}};
44 var agente;
45
46 var db_url = "mongodb://localhost:27017/";
47 MongoClient.connect(db_url, { useUnifiedTopology: true }, function(err, db) {
48     if (err) throw err;
49     httpServer.listen(8080);
50     var io = socketio(httpServer);
51
52     var dbo = db.db("P4");
53     dbo.createCollection("sensores", function(err, res){
54         var collection = res;
55         if (err){
56             collection = dbo.collection("sensores");
57             if (collection === undefined) throw err;
58         }
59     });
60 }

```

Podemos ver que en primer lugar se establece la conexión con el servidor de mongodb, se pone al servidor http creado anteriormente a escuchar en el puerto 8080 y se inicia el sistema de sockets para dicho servidor.

Si la conexión con mongodb ha sido exitosa, se procederá a usar/crear una base de datos de nombre "P4" para poder hacer uso de una colección de nombre "sensores" dentro de esta base de datos. Primero se intenta crear la colección con dicho nombre, si la función retorna un error, asumiremos que la colección ya existía e intentaremos consultarla a la base de datos. En el caso de que el error fuese otro y siguiéramos sin tener acceso a la colección, mostramos el error por pantalla.

A partir de este punto, comienza la funcionalidad que depende de Socket.IO. Con

```
io.sockets.on('connection', function(client) {
```

establecemos que cuando un cliente cualquiera se conecte al servidor se realizará lo que contenga la función que se le pasa como segundo parámetro. En este caso, el servidor querrá suscribirse a una serie de eventos que puede producir dicho cliente. Estos eventos son los siguientes:

- **sensores:** Este es el evento asociado al envío de nueva información por parte de los sensores. Cuando los sensores generan este evento, el servidor se encargará de generar una timestamp (esto se realiza en el servidor con el fin de disponer de un reloj centralizado para los participantes del sistema ante posibles discordancias de relojes al tratarse de un sistema distribuido) para a continuación, emitir dicha actualización de los sensores a todos los clientes del sistema y almacenar el registro en la colección de la base de datos.

```
client.on('sensores', function(data){
    data.timestamp = new Date();
    io.sockets.emit('sensor-update', data);
    collection.insertOne(data);
});
```

- **persiana:** Este es un evento que puede generar un cliente para poner en marcha el actuador de la persiana y cambiar su estado. Será necesario comunicar de este cambio al resto de clientes para que todos estén actualizados.

```
client.on('persiana', function(data){
    estado.persiana = data;
    io.sockets.emit('emitir-estado', estado);
});
```

- **ac:** Es el equivalente al evento anterior pero para el aire acondicionado.
- **obtener-estado:** Este será el evento que genere un cliente cuando desee saber el estado actual de los dispositivos del hogar. Se le responderá a dicho cliente mediante un evento “emitir-estado” que incluirá el estado actual del aire acondicionado y la persiana.

```
client.on('obtener-estado', function(){
    client.emit('emitir-estado', estado);
});
```

- **sensores-limite:** Este será el evento generado por el agente cuando suceda la situación crítica descrita en el guión, es decir, que tanto el sensor de temperatura como el de luminosidad superen unos valores máximos (30°C y 80 respectivamente). Cuando esto suceda, el servidor se encargará de emitir el aviso generado por el agente al resto de clientes con el evento genérico “agent-msg”.
- **heat-warning, brightness-warning, general-warning:** Los tres realizan la misma función de reenviar un warning generado por el agente al resto de clientes. El motivo por el que se han especificado eventos distintos para el calor y la temperatura además del genérico es por si en un futuro se quisiera realizar acciones específicas a dichos eventos más complejas.

```
client.on('heat-warning', function (data) {
  io.sockets.emit('agent-msg', data);
});

client.on('brightness-warning', function (data) {
  io.sockets.emit('agent-msg', data);
});

client.on('general-warning', function (data) {
  io.sockets.emit('agent-msg', data);
});
```

- **obtener-registro:** Los clientes que accedan al panel de control del usuario generarán este evento al comienzo para solicitar el listado completo de mediciones de los sensores y poder mostrar el histórico al usuario.

```
client.on('obtener-registro', function (){
  collection.find().toArray(function(err, results){
    client.emit('emitir-registro', results);
  });
});
```

Se usará `collection.find()` para obtener el contenido de la colección al completo y `toArray` para generar un array con dicho contenido. Cuando se haya completado la query a la base de datos se emitirá un evento “emitir-registro” al cliente que ha realizado la petición y se incluirán los datos consultados.

Problemas encontrados

- **Conexión a la base de datos** cuando la colección ya existía previamente: Este problema ya ha sido tratado anteriormente.
- **Operaciones asíncronas** en Node.JS: Al intentar obtener información de la base de datos, como la query se realiza de forma asíncrona, al intentar utilizar el resultado de dicha query en la línea siguiente nos encontramos que la variable está vacía. Esto se puede solucionar pasando como parámetro de la query la función que va a hacer uso del resultado de esta, para asegurarnos de que solo se utiliza una vez los datos están disponibles para su uso.

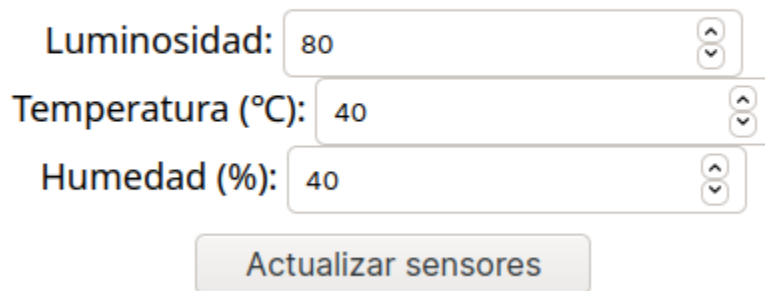
Sensores

Introducción

Este componente se encargará de tomar las medidas de temperatura, luminosidad y humedad que utilizará el servidor. Para ello se dispone de dos opciones:

- **Un formulario manual:** Será posible generar una nueva lectura de los sensores rellenando un formulario con tres campos, temperatura, luminosidad y humedad.

Formulario de sensores



Formulario de sensores con tres campos de entrada y un botón de actualización:

- Luminosidad: 80
- Temperatura (°C): 40
- Humedad (%): 40
- Botón: Actualizar sensores

- **OpenWeatherMap API:** Gracias al uso de la API gratuita de OpenWeatherMap, podemos obtener mediciones meteorológicas en tiempo real de las tres variables mencionadas anteriormente dada una ubicación. Veremos cómo funciona más adelante.

Para poder alternar entre métodos de medición, la página de los sensores dispone de un interruptor que permite activar el uso de la API meteorológica.

Utilizar API de OpenWeatherMap



Implementación de la API de OpenWeatherMap

Antes de poder hacer uso de la API que ofrecen, es necesario registrarse en su [web](#) para obtener la clave que usaremos para las peticiones. Para esta práctica usaremos su plan gratuito, el cual tiene un número limitado de peticiones pero es más que suficiente para el uso que se le va a dar aquí.

Una vez tengamos la clave, ya podemos empezar a realizar llamadas a la API. Cuando se active esta funcionalidad desde la página de sensores se llamará al método

“enableOpenWeatherMap()”, el cual se encargará de activar un temporizador para realizar consultas meteorológicas automáticas cada tres segundos mediante la función setInterval de javascript.

Para realizar las llamadas se utilizará el método fetch, al que le pasaremos la url que proporciona OWM parametrizada de forma que incluya nuestra key y la ciudad sobre la que realizamos la consulta.

Se ha hecho uso de la API de geolocalización de HTML para obtener las coordenadas del usuario y poder así, darle a esta información basada en su ubicación. Debido a que el uso de esta API presenta problemas al ser usada en HTTP (se comentará más adelante), se establecerá la ubicación a Granada en caso de que no se consigan obtener los datos de posicionamiento.

Una vez con la url definida, se realizará la petición a la API y cuando se reciba respuesta, esta se convertirá a JSON y se extraerán las tres variables de las que hace uso la aplicación, temperatura (es necesario realizar la conversión a celsius), luminosidad (dividimos entre 100 ya que los valores recibidos van de 0 a 10000 y los nuestros de 0 a 100) y humedad.

Una vez conseguidas las tres variables, se emitirá un evento ‘sensores’ para comunicar al servidor de las nuevas lecturas.

```
function enableOpenWeatherMap(){
  owm_interval = setInterval(function(){
    var api_url = 'https://api.openweathermap.org/data/2.5/weather?q=Granada,ES&appid=' + key;
    if (coord)
      api_url = 'https://api.openweathermap.org/data/2.5/weather?lat=' + coord.coords.latitude + '&lon=' + coord.coords.longitude + '&appid=' + key;

    fetch(api_url)
      .then(function(resp) { return resp.json(); })
      .then(function(data) {
        var grados_celsius = Math.round(parseFloat(data.main.temp)-273.15);
        var luminosidad = Math.round(data.visibility/100);
        var humedad = data.main.humidity;
        socket.emit('sensores', {luminosidad:luminosidad, temperatura:grados_celsius, humedad:humedad});
      });
  }, 3000);
}
```

Métodos utilizados

Además del método descrito anteriormente, hay una serie de métodos y eventos que se utilizan:

- **enviar():** Este método se encarga de extraer los valores introducidos en el formulario y de emitir un evento “sensores” para informar al servidor de la nueva lectura. Se llamará a enviar() cuando se pulse en el botón de enviar formulario.

```
function enviar() {
  var luminosidad = document.getElementById("luminosidad").value;
  var temperatura = document.getElementById("temperatura").value;
  var humedad = document.getElementById("humedad").value;

  socket.emit('sensores', {luminosidad:luminosidad, temperatura:temperatura, humedad:humedad});
}
```

- **toggleOpenWeatherMap():** Este método activa y desactiva el uso de la API de OWM, es decir, pondrá en marcha o detendrá el intervalo que gestiona las llamadas a la API.

```
function toggleOpenWeatherMap(){
    var estado_owm = document.getElementById("toggle_owm_actuador").checked;

    if (estado_owm)
        enableOpenWeatherMap();
    else{
        if (owm_interval !== undefined){
            clearInterval(owm_interval);
            owm_interval = undefined;
        }
    }
}
```

- **getLocation():** Este método hace uso de la API de geolocalización de HTML para obtener las coordenadas actuales del usuario. Una vez las obtiene, las muestra por pantalla en sus correspondientes contenedores.

Y los siguientes eventos:

- **obtener-estado:** Al iniciar los sensores, se emitirá este evento para solicitar al servidor el estado actual de sus dispositivos y comenzar así con un estado actualizado.
- **emitir-estado:** Los sensores se suscriben a este evento para actualizar la información que se muestra por pantalla sobre el estado de los dispositivos/actuadores.

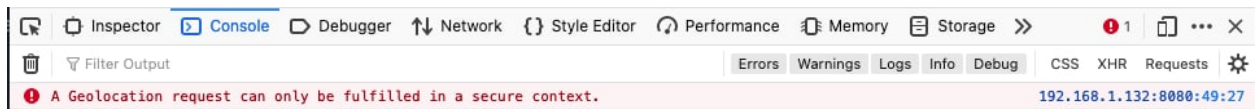
```
socket.on('emitir-estado', function(data){
    estado = data;
    document.getElementById("ac").innerHTML = estado.ac.on ? 'Encendido' : 'Apagado';
    document.getElementById("modo_ac").innerHTML = estado.ac.modo;
    document.getElementById("persiana").innerHTML = estado.persiana ? 'Subida' : 'Bajada';
});
```

Problemas encontrados

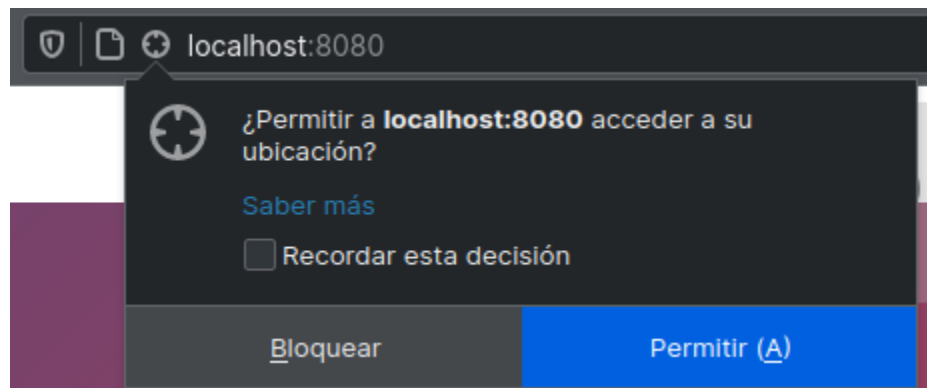
- Problemas para usar la **API de geolocalización** del usuario: Ya que es posible utilizar la información meteorológica proporcionada por OpenWeatherMaps, se intentó utilizar la geolocalización del usuario para obtener su ubicación actual y realizar las peticiones a la API con la ciudad en la que este se encontrase. Para ello, se implementó esta función

```
function getLocation() {
  if (navigator.geolocation) {
    navigator.geolocation.getCurrentPosition(function(position){
      coord = position;
      document.getElementById("coordenadas").style.display = "block";
      document.getElementById("lat").innerHTML = coord.coords.latitude
      document.getElementById("lon").innerHTML = coord.coords.longitude;
    });
  } else {
    console.log("Geolocalización no disponible.");
  }
}
```

pero por desgracia, por motivos de seguridad solo es posible obtener la geolocalización del usuario si se trata de una conexión HTTPS.



Por suerte, se nos permite acceder a la geolocalización accediendo al servidor desde localhost.



Agente

Este componente del sistema se encarga de actuar ante ciertas situaciones que se produzcan en el sistema relacionadas con las mediciones de los sensores y con el estado de los dispositivos como la persiana y el aire acondicionado.

Implementación

El agente (agente.js) comienza obteniendo dos parámetros de la línea de comandos, el host y puerto del servidor principal del sistema. Estos serán necesarios para conectarse e interactuar con el servidor domótico.

```
var addr = process.argv.slice(2);
if (addr.length != 2){
    console.log("Uso: node[js] agente.js host port");
    process.exit(-1);
}
var socket = io.connect('http://' + addr[0] + ':' + addr[1]);
```

A continuación, se utilizan una serie de eventos:

- **connect_error:** Este es un evento predefinido de Socket.IO que se produce cuando hay un error de conexión. El agente se suscribe a este para poder informar de si se ha producido un error al intentar comunicarse con el servidor principal.
- **connect:** Evento predefinido de Socket.IO que se produce al establecer la conexión con el servidor. El agente se suscribe a este para aportar feedback sobre el estado de la conexión.
- **sensor-update:** Este es el evento más importante al que estará suscrito el agente. Cuando el servidor emita una actualización de las mediciones de los sensores, el agente comprobará que no se da ninguna condición indeseada. En el caso de que se diera, emitiría un evento al servidor para avisar de ello y en ocasiones, intentar arreglarlo. Las condiciones a comprobar son las siguientes:
 - **Límite de sensores de temperatura y luminosidad:** Si se sobrepasan los 30°C y 80 de luminosidad se emitirá un evento “sensores-limite” al servidor avisando de esto además de emitir otro evento para encender el aire acondicionado en modo frío.

```
if (temp >= 30 && luminosidad >= 80){
    socket.emit("sensores-limite", "Se han sobrepasado los umbrales " +
        "de los sensores, cerrando persianas y encendiendo el aire...");
    socket.emit('persiana', false);
    socket.emit('ac', {on:true, modo:"Frío"});
}
```

- **Temperatura excesivamente alta:** Se aplica el mismo procedimiento anterior cuando se alcanzan los 30°C pero con un mensaje de aviso distinto de acuerdo a esta situación contenido dentro de un evento “heat-warning”.

- **Temperatura excesivamente baja:** Este caso se da cuando la temperatura baja de los 15°C. De nuevo, el agente emitirá un aviso descriptivo y emitirá un evento “ac” pero esta vez con el modo caliente.
- **Humedad excesivamente alta:** Cuando la humedad sobrepasa el 50%, se emitirá un evento para activar el aire acondicionado en modo deshumidificador además de el evento de aviso.
- **Luminosidad excesivamente alta:** Si la luminosidad excede 80, avisar de esto al servidor con un evento “brightness-warning”.
- **emitir-estado:** El agente se suscribe a este evento con el fin de detectar posibles casos de malgasto energético como puede ser tener el aire acondicionado encendido mientras la persiana/ventana está abierta. Para ello, se comprueba si en el estado de los dispositivos comunicado por el servidor el AC está funcionando y la persiana está abierta. En caso de estarlo se emite un evento “general-warning” (destinado a mensajes de aviso del agente genéricos) que avisará de lo ocurrido.

```
socket.on('emitir-estado', function (data) {  
  if (data.ac.on && data.persiana)  
    socket.emit("general-warning", "Posible malgasto energético: AC encendido y persiana subida");  
});
```

Panel de usuario

Este será el punto de acceso que tenga el usuario para consultar información sobre el estado del sistema domótico además de poder interactuar sobre este con acciones como subir o bajar la persiana y encender o apagar el aire.

Este panel de control presenta principalmente cuatro elementos:

- **Panel de luminosidad:** En este panel se mostrará la última lectura de los sensores sobre luminosidad, el estado de la persiana y un botón para que el usuario pueda interactuar con esta.
- **Panel de temperatura:** Este panel contiene la última lectura de los sensores sobre la temperatura, el estado y modo actual del aire acondicionado y por último, un botón para interactuar con el AC.
- **Panel de humedad:** Este panel muestra la última lectura de los sensores sobre la humedad actual.
- **Registro de sensores:** En este panel se muestra el histórico de lecturas de los sensores. Este se va actualizando conforme surgen nuevas lecturas.

Implementación

Al inicializar el panel de control del usuario, se realiza lo siguiente:

```
var estado = {persiana:false, ac:{on:false, modo:"Frío"}};
var serviceURL = document.URL;
serviceURL = serviceURL.substring(0, serviceURL.lastIndexOf('/'));
var socket = io.connect(serviceURL);

socket.emit('obtener-estado', {});
socket.emit('obtener-registro', {});
```

En primer lugar, se define un estado por defecto para los actuadores al igual que se hizo en el servidor, a continuación se recorta la url para quedarnos solo con el host:puerto que es lo que necesita el método connect de Socket.IO para establecer la conexión con el servidor.

Una vez realizada la conexión, se emiten dos eventos “obtener-estado” y “obtener-registro” cuyo objetivo es solicitar al servidor el estado de los dispositivos y el registro completo de medidas de los sensores para poder partir con el estado del servidor actualizado.

El panel del usuario se suscribe a los siguientes eventos:

- **emitir-estado:** El panel de usuario se suscribe a este evento para mantener la vista que ofrece al usuario sobre el estado de los dispositivos actualizada. Para ello, se consulta el estado recibido y se modifican los contenedores html correspondientes.

```
socket.on('emitir-estado', function(data){
    estado = data;
    document.getElementById("ac").innerHTML = estado.ac.on ? 'Encendido' : 'Apagado';
    document.getElementById("modo_ac").innerHTML = estado.ac.modo;
    document.getElementById("persiana").innerHTML = estado.persiana ? 'Subida' : 'Bajada';
    updateBotones();
});
```

- **sensor-update:** Sigue la misma filosofía que el evento anterior pero esta vez tratándose de las mediciones de los sensores. Al final, se añadirá al registro para que esta medición aparezca también en el panel de registro de sensores.

```
socket.on('sensor-update', function(data){
    document.getElementById("luminosidad").innerHTML=data.luminosidad;
    document.getElementById("temperatura").innerHTML=data.temperatura;
    document.getElementById("humedad").innerHTML=data.humedad;
    addRegistro(data);
});
```

- **agent-msg:** El objetivo de esto es mostrar por pantalla al usuario los avisos que ha generado el agente, modificando el contenedor html correspondiente.
- **emitir-registro:** El servidor emite este evento como respuesta a la solicitud realizada al inicio del panel de usuario para comunicar el registro de mediciones de los sensores al completo.

Además, hace uso de los siguientes métodos:

- **set_ac():** Es el método que se invoca cuando el usuario interactúa con el botón del aire acondicionado. Su objetivo es modificar el estado de este y emitir un evento “ac” informando del nuevo estado.

```
function set_ac(){
    estado.ac.on = !estado.ac.on;
    document.getElementById("ac").innerHTML = estado.ac.on ? 'Encendido' : 'Apagado';
    updateBotones();
    socket.emit('ac', {on:estado.ac.on, modo:estado.ac.modo});
}
```

- **set_persiana():** Igual al anterior pero con el actuador de la persiana.
- **update_botones():** Se encarga de actualizar el texto de los botones para la persiana y el AC de acuerdo a su estado actual.
- **addRegistro():** Genera los elementos html necesarios junto con sus atributos para insertar un nuevo registro de los sensores en el panel de registros.

```
function addRegistro(element){
    var contenedor = document.getElementById("box");
    var cadena = "Luminosidad: " + element.luminosidad +
    ", temperatura: " + element.temperatura + ", humedad: " + element.humedad + ", timestamp: " +
    element.timestamp;
    var contenedor_elemento = document.createElement('div');
    var clase = document.createAttribute("class");
    clase.value = "registro";
    contenedor_elemento.setAttributeNode(clase);
    contenedor_elemento.innerHTML = cadena;
    contenedor.appendChild(contenedor_elemento);
}
```

Diseño

Para mejorar la estética de la aplicación he hecho uso de un diseño web que realicé para las prácticas de la asignatura de Sistemas de Información Basados en Web. Este diseño resulta bastante adecuado para esta aplicación y mantiene una correcta visualización en distintos tamaños de ventana.

Esta es una imagen de la plantilla vacía:



Todo el estilo de la aplicación se ha realizado mediante CSS en el archivo “./styles/style.css”.

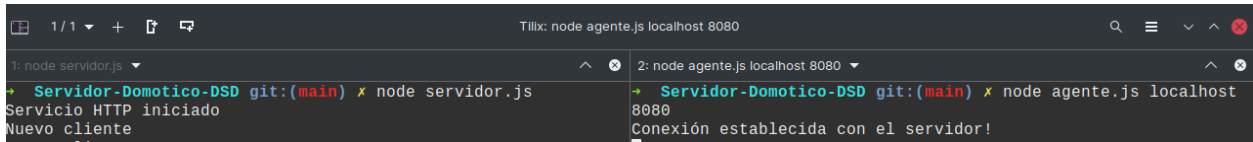
Instrucciones de uso

Para utilizar la aplicación, será necesario disponer de varios módulos de NodeJS. Estos son:

- MongoDB
- Socket.IO (servidor y socket.io-client)

Una vez se disponga de dichos módulos, será necesario ubicarse en el directorio del código, abrir una terminal y ejecutar “*node[js] servidor.js*”

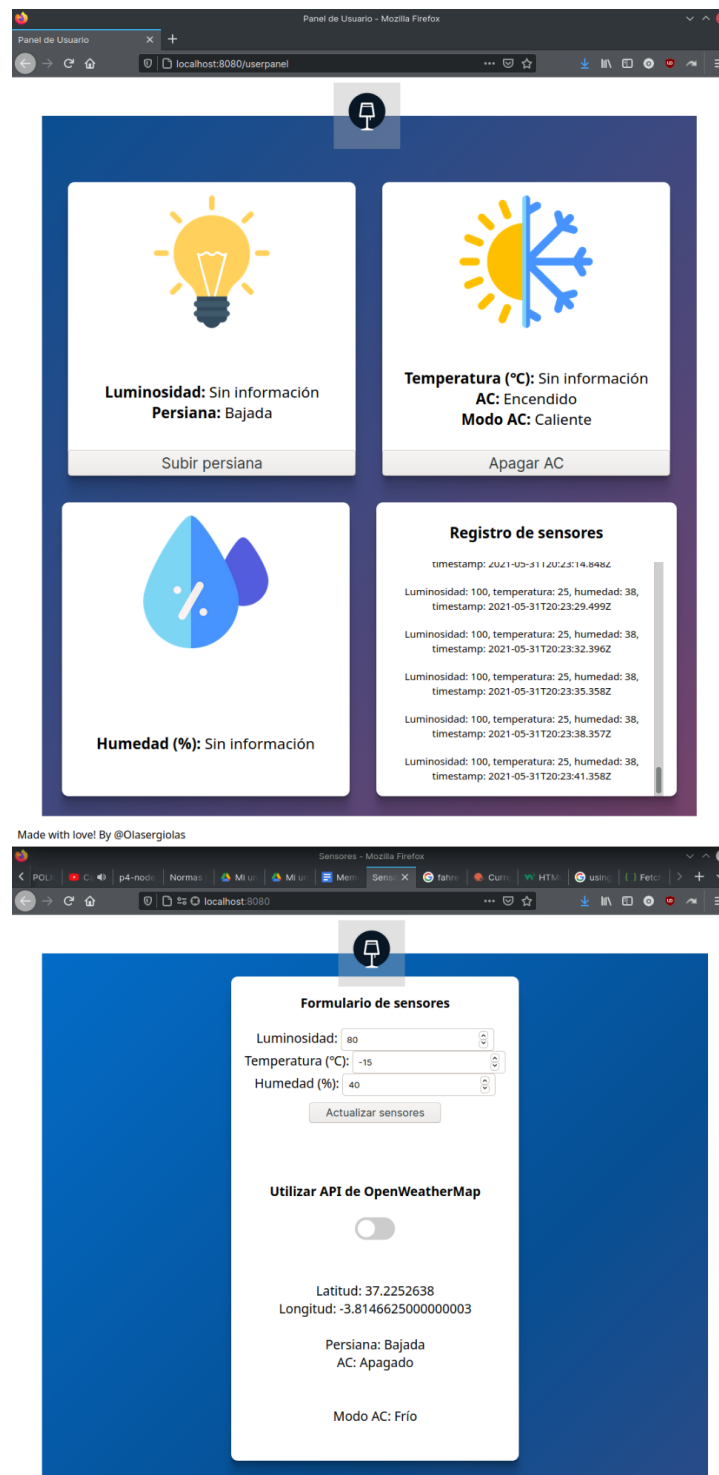
Tras esto, podemos abrir otra terminal y poner en marcha el agente con “*node[js] agente.js host puerto*”. Para estas pruebas utilizaremos “*node agente.js localhost 8080*”.



```
1: node servidor.js
+ Servidor-Domotico-DSD git:(main) * node servidor.js
Servicio HTTP iniciado
Nuevo cliente

2: node agente.js localhost 8080
+ Servidor-Domotico-DSD git:(main) * node agente.js localhost 8080
Conexión establecida con el servidor!
```

Ahora solo será necesario acceder en una ventana del navegador a “*localhost:8080*” y en otra a “*localhost:8080/userpanel*”



Prueba de uso

Para poner a prueba la aplicación y simular el funcionamiento del sistema domótico se ha grabado el siguiente [vídeo](#). En él, podemos ver como tanto el panel de control como el formulario de los sensores se mantiene actualizado en todo momento y cómo el agente toma decisiones ante ciertas situaciones. Además, podemos ver en acción el uso de la API de OpenWeatherMap para tomar la información de los sensores.