# Práctica 4 - Desarrollo de servicios web con Node.js, Socket.io y MongoDB

## Parte 1: Ejemplos

**Jose Luis Gallego Peña - Desarrollo de Sistemas Distribuidos (DSD2)** Para ejecutar los ejemplos ha sido necesario instalar nodejs, socketio y mongodb. Para iniciar los servicios web se introduce el siguiente comando en la terminal:

```
node servicio.js
```

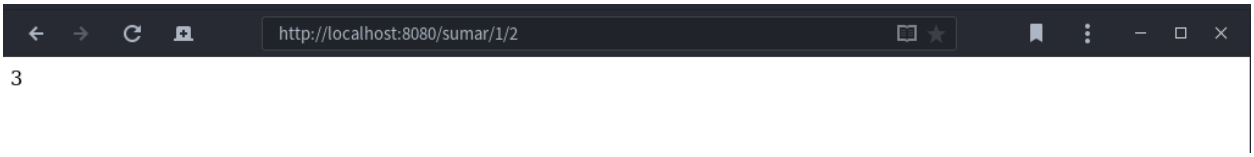
Para ver el servicio web iniciamos el navegador y escribimos la dirección en la que esté alojado, en este caso todos estarán en **localhost:8080**. Es decir, en el puerto 8080 de nuestra máquina.

### 1. helloworld.js

Este primer ejemplo simplemente crea un servidor HTTP y como respuesta escribe un "Hola mundo" en texto plano en el navegador.


### 2. calculadora.js

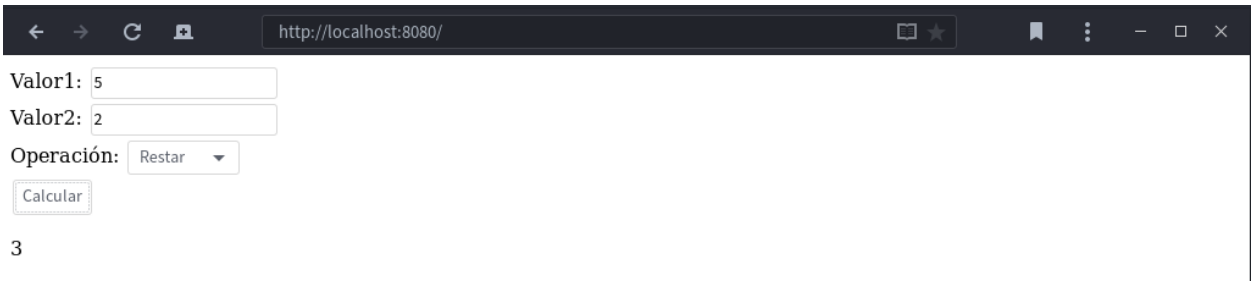
Este servicio recibe peticiones REST y es una calculadora. Si pones la dirección normal, te dice un mensaje por defecto, pero si quieres usar la calculadora entonces habrá que pasar por la uri del servicio el tipo de operacion (sumar, restar, multiplicar, dividir) y luego los operandos, todo ello dividido por /.


Crea un servidor HTTP y obtiene la uri que ha introducido el cliente, dividiendo los 3 parametros y pasandoselos a una función para que calcule según esos parámetros. Finalmente escribe como respuesta el resultado usando el objeto response, al igual que con el hola mundo.

### 3. calculadora-web.js

Este servicio es como el anterior, recibe peticiones REST y es una calculadora. Sin embargo, ahora usa una pequeña interfaz de usuario hecha en html en la que no es necesario introducir los datos en la uri de la web (aunque sigue estando la posibilidad), si no que se usa un formulario.




Tenemos por un lado el servicio **calculadora-web.js** y por otro lado el cliente web **calc.html**.

El servicio es parecido al anterior, sin embargo antes de todo comprueba que la uri tiene parámetros (como en el ejemplo anterior) o no. Si tiene parámetros, funciona como antes, si no, comprueba si existe ese fichero html y lo muestra.

```
fs.exists(fname, function(exists) {
            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 existe el fichero, escribe con el objeto respuesta los datos de ese html y no hace nada más, el resto lo hace el html.

El cliente web html tiene varios bloques con el formulario, con las cajas de texto para introducir los datos y un botón submit para enviar estos datos. Cuando se pulsa ese botón submit se llama a un script de javascript definido en este fichero

```
<form action="javascript:void(0);" onsubmit="javascript:enviar();">
```

Este fichero hace es una petición AJAX al servicio usando los parámetros escritos por url, obtiene el resultado y lo incrusta en el html.

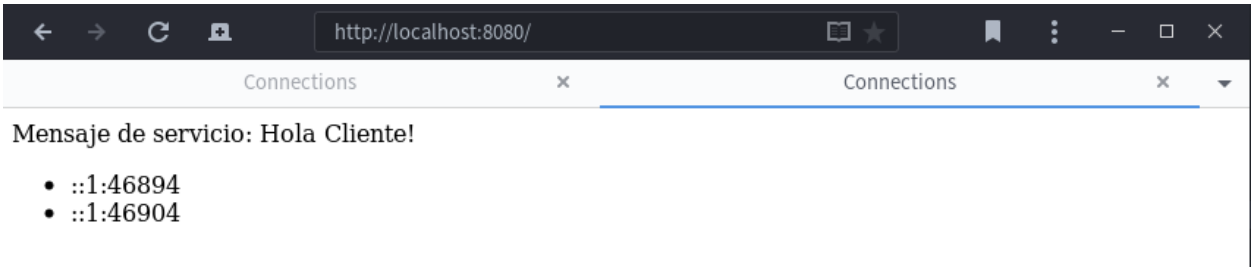
```
var serviceURL = document.URL;
        function enviar() {
            var val1 = document.getElementById("val1").value;
            var val2 = document.getElementById("val2").value;
            var oper = document.getElementById("operacion").value;

            var url = serviceURL+"/"+oper+"/"+val1+"/"+val2;
            var httpRequest = new XMLHttpRequest();
            httpRequest.onreadystatechange = function() {
                if (httpRequest.readyState === 4){
                    var resultado = document.getElementById("resul");
                    resultado.innerHTML = httpRequest.responseText;
```

```
            }
        };
        httpRequest.open("GET", url, true);
        httpRequest.send(null);
    }
```

La XMLHttpRequest es la petición AJAX (asíncrona, es decir, espera a recibir los datos) de tipo GET a la url que se ha formado en un string usando los parámetros del formulario.

## 4. connections.js

Este servicio usa socket.io para actualizar lo que muestra una página web sin tener que recargarla. En este caso, lo que hace es mostrar la dirección de un usuario y el puerto desde el que se conecta al servicio. La dirección es la misma puesto que para probarlo he usado el mismo ordenador y dos pestañas distintas. La información se actualiza en tiempo real en ambas pestañas. Si se cerrase una pestaña, se quitaría uno de los dos elementos de la lista y solo quedaria uno.




Tenemos por un lado el servicio **connections.js** y por otro lado el cliente web **connections.html**.

- El servicio crea un servidor HTTP de la misma manera que el ejemplo anterior, mostrando el archivo html. Una vez creado este servidor y dicho en qué puerto escucha, el resto de los ejemplos terminaba ahí porque no tenía que hacer más, sin embargo en este caso es ahora cuando se crea el servicio de socket.io, que escuchará en el servidor ya creado.

  ```
  httpServer.listen(8080);
  var io = socketio.listen(httpServer);
  ```

  Se crea un array de clientes, en el que se almacenarán a modo de base de datos volatil los clientes que se conectarán con el servidor. Cada vez que socket.io detecte una conexión (mediante el evento "connection"), llamará a una función que obtiene los datos de ese cliente (dirección y puerto) y los añade al array. Seguido de esto, envía un mensaje usando la función **emit** al resto de clientes web con el array. El por qué de esto se explicará después. Luego, usando la función **on** espera a recibir un evento del tipo **output-evt** que es un mensaje que le habrá enviado el cliente a modo de saludo, indicando que la conexión es correcta, y entonces el servicio le enviará al cliente un mensaje para que lo muestre.

  Se indica también una escucha al evento "disconnect". En el caso de que un cliente conectado se desconecte, se elimina del array de clientes y se manda un mensaje a todos los clientes con el nuevo array de clientes.

- El cliente web html al igual que antes define varios bloques en los que mostrará los datos obtenidos. Contiene un script javascript que, con socketio, se conecta al servidor que hemos definido anteriormente, y a continuación define varias escuchar a distintos eventos:

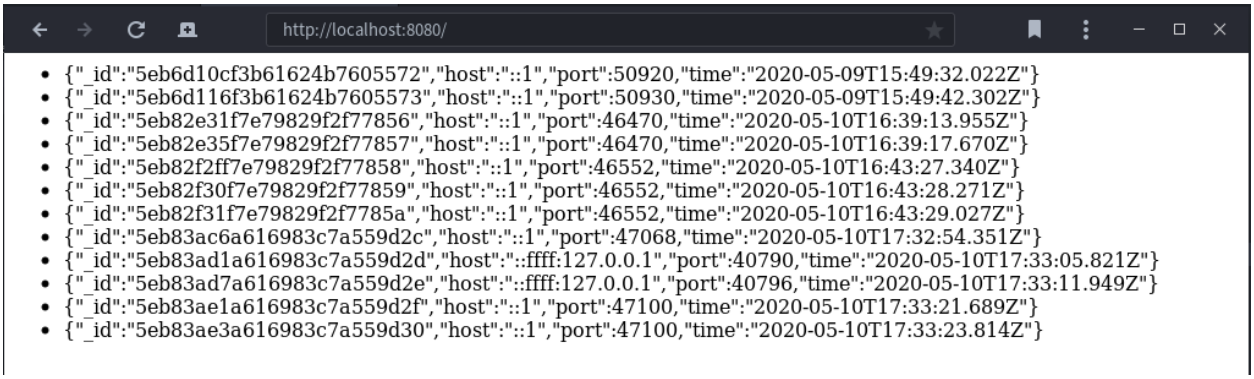
  Escucha en **connect**, que como hemos dicho antes es cuando se conecta, que lo que hace es llamar al evento de saludo que dijimos antes para que el servidor le vuelva a mandar un mensaje y el cliente web lo escuche en **output-evt**, para mostrarlo en la web. También muestra un mensaje en el evento **disconnect** en el caso en el que el servidor se caiga.

  Escucha en **all-connections** aquellos mensajes que se han dicho antes que se mandan a todos los clientes, cuyo argumento es la lista de todos los clientes. Llama a una función eliminando la lista que

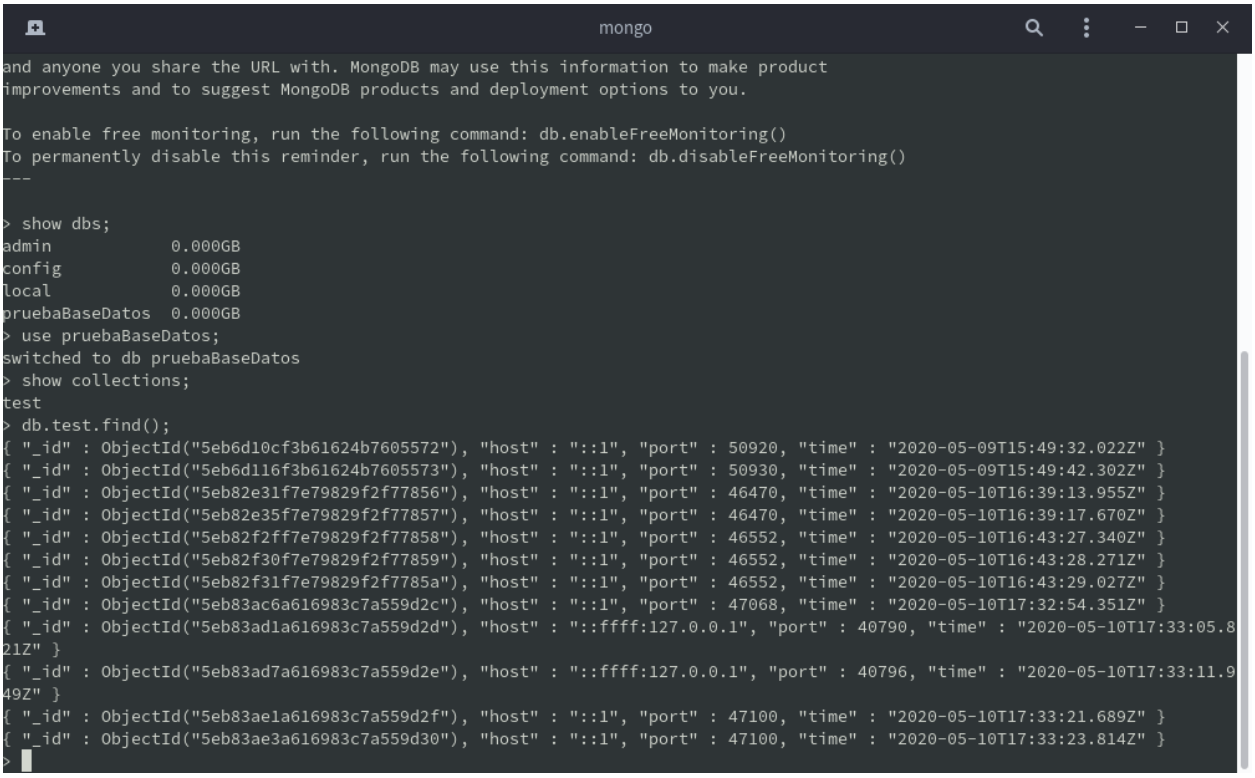
ya mostraba la web y mostrando la nueva actualizada.

## 5. mongo-test.js

Este servicio usa socket.io y mongodb para actualizar la página sin tener que recargarla y almacenar y sacar datos de una base de datos NoSQL. Al igual que en el ejemplo anterior, muestra la dirección de un usuario y el puerto desde el que se conecta al servicio. La información se actualiza en tiempo real en ambas pestañas, sin embargo ahora si se cerrase una pestaña no se quitaría un elemento de la lista, se quedan siempre guardados en la base de datos, a modo de historial de todas las conexiones desde el principio.




Además, podemos ver cómo en la base de datos tenemos los datos que vemos en la web




Tenemos por un lado el servicio **mongo-test.js** y por otro lado el cliente web **mongo-test.html**.

- El servicio crea un servidor HTTP de la misma manera que el ejemplo anterior, mostrando el archivo html. Una vez creado este servidor y dicho en qué puerto escucha, se sigue como antes pero en este caso usando una función de mongodb para conectarse a la base de datos.

  Cuando se conecte a la base de datos, se llama a una función callback que al igual que antes usa socketio

para escuchar el servidor http y entonces se crea una colección en la base de datos (como una tabla en las relacionales).

A continuación se mandan un mensajes a un evento que escuchará el cliente web. El evento **my-address** envía los datos del cliente como antes. Y luego, escuchará a dos eventos esperando a que el cliente le mande los mensajes:

- **poner**, en el cual recibe los datos del cliente y los almacena en la BD.
- **obtener**, en el cual recibe una petición del cliente y realiza una consulta en la BD, para a continuación enviarle un mensaje al cliente con esos datos.

Es decir, cada vez que un usuario se conecte, se hace una conexion a la BD y se escucha para obtener datos o introducir datos segun mensajes del cliente.

- El cliente web al igual que antes define varios bloques en los que mostrará los datos obtenidos. Contiene un script javascript que, con socketio, se conecta al servidor que hemos definido anteriormente, y a continuación define varias escuchar a distintos eventos:

  **my-address**, en el cual recibe los datos del cliente que se ha conectado, crea una variable con la hora y envia todos esos datos a **poner** del servicio, para que se le introduzcan esos datos, y al **obtener** del servicio, para volver a obtener esos datos introducidos y al escuchar en ese mismo evento actualizar la lista como antes (con todos los clientes desde el principio).

```
socket.on('my-address', function(data) {
          var d = new Date();
          socket.emit('poner', {host:data.host, port:data.port, time:d});
          socket.emit('obtener', {host: data.address});
      });
      socket.on('obtener', function(data) {
          actualizarLista(data);
      });
```