

# DSD-P4

Raúl Durán Racero

Mayo 2022

## 1. Introducción

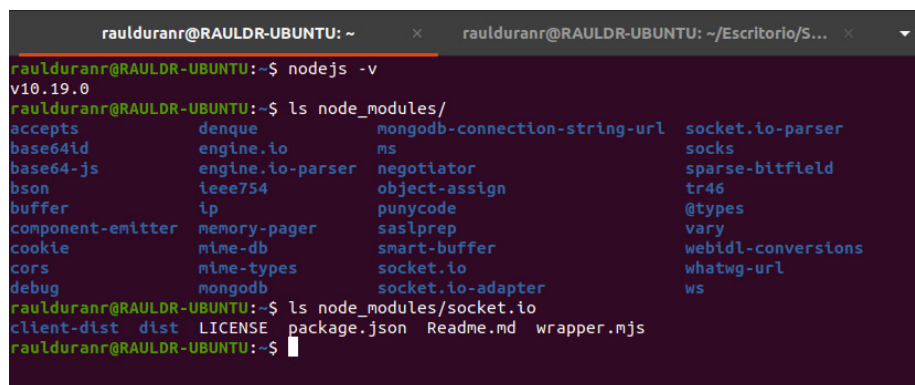
En esta practica, se nos propone como ejercicio desarrollar con Node.js y Socket.IO el diseño e implementación del sistema de una casa domótica, el cual está formado por dos sensores de luminosidad y temperatura, dos actuadores que serían una persiana y el sistema de aire acondicionado, un servidor que proporciona páginas para poder comprobar el estado y actuar sobre los elementos de la vivienda, y un agente capaz de notificar alarmas y tomar decisiones básicas.

Antes de comenzar con el desarrollo del ejercicio, se nos propone una serie de ejemplos que tenemos que comprender y comprobar que funcionen correctamente en nuestro sistema.

## 2. Ejemplos

Antes de poder ejecutar los ejemplos, tenemos que comprobar que tenemos correctamente instalado:

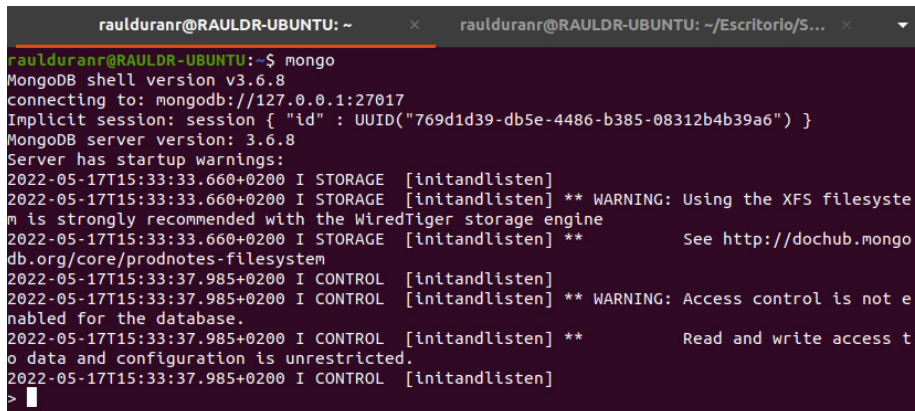
- Node.js y Socket.IO:



```
raulduranr@RAULDR-UBUNTU: ~  
raulduranr@RAULDR-UBUNTU:~$ nodejs -v  
v10.19.0  
raulduranr@RAULDR-UBUNTU:~$ ls node_modules/  
accepts      denque      mongodb-connection-string-url  socket.io-parser  
base64id     engine.io  ms                        socks  
base64-js    engine.io-parser  negotiator      sparse-bitfield  
bson         ieee754    object-assign     tr46  
buffer       ip         punycode          @types  
component-emitter  memory-pager  saslprep        vary  
cookie       mime-db    smart-buffer      webidl-conversions  
cors         mime-types socket.io          whatwg-url  
debug        mongodb    socket.io-adapter ws  
raulduranr@RAULDR-UBUNTU:~$ ls node_modules/socket.io  
client-dist  dist  LICENSE  package.json  Readme.md  wrapper.mjs  
raulduranr@RAULDR-UBUNTU:~$
```

Figura 1: Comprobamos que ambos están instalados correctamente

## ■ MongoDB



```
rauldurant@RAULDR-UBUNTU: ~$ mongo
MongoDB shell version v3.6.8
connecting to: mongodb://127.0.0.1:27017
Implicit session: session { "id" : UUID("769d1d39-db5e-4486-b385-08312b4b39a6") }
MongoDB server version: 3.6.8
Server has startup warnings:
2022-05-17T15:33:33.660+0200 I STORAGE [initandlisten]
2022-05-17T15:33:33.660+0200 I STORAGE [initandlisten] ** WARNING: Using the XFS filesystem
is strongly recommended with the WiredTiger storage engine
2022-05-17T15:33:33.660+0200 I STORAGE [initandlisten] ** See http://dochub.mongo
db.org/core/prodnotes-filesystem
2022-05-17T15:33:37.985+0200 I CONTROL [initandlisten]
2022-05-17T15:33:37.985+0200 I CONTROL [initandlisten] ** WARNING: Access control is not e
nabled for the database.
2022-05-17T15:33:37.985+0200 I CONTROL [initandlisten] ** Read and write access t
o data and configuration is unrestricted.
2022-05-17T15:33:37.985+0200 I CONTROL [initandlisten]
```

Figura 2: Comprobamos que podemos acceder a mongodb

Una vez hemos comprobado que tenemos todo lo necesario, podemos pasar a comprobar que los ejemplos dados funcionan correctamente:

## 2.1. Hola Mundo

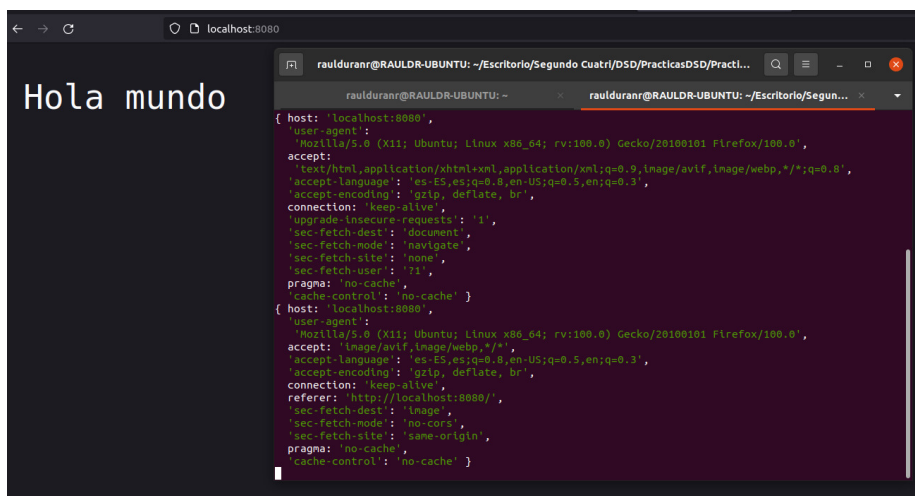


Figura 3: Hello World

## 2.2. Calculadora

En este ejemplo de una calculadora, el operador y los operandos se pasan como argumentos en la dirección, es decir, tras poner localhost:8080. Puede verse en las siguientes imágenes lo que ocurre si no le pasas nada (entras sólo en localhost:8080) y cuando le pasas los argumentos necesarios:

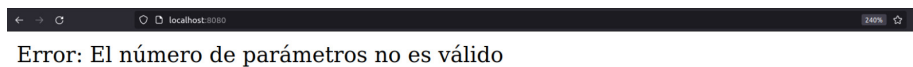


Figura 4: Error si no le pasas los argumentos

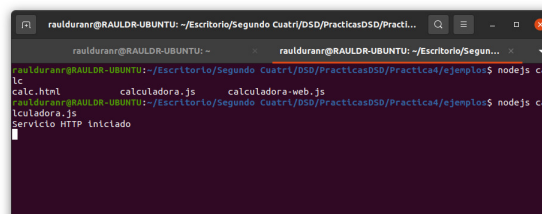


Figura 5: Calculadora por parametros

### 2.3. Calculadora-web

En este ejemplo tenemos de nueva una calculadora, pero esta vez se nos ofrece una interfaz para que sea más interactiva. Aún así, podemos ver cómo se realiza el cálculo en la terminal.

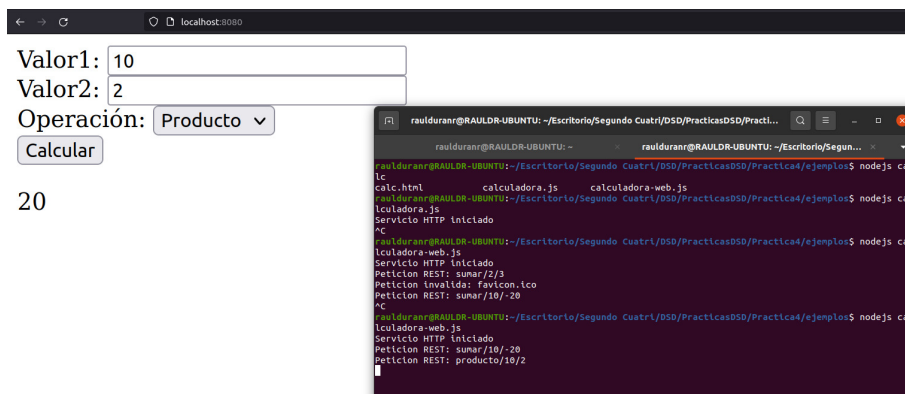


Figura 6: Calculadora con interfaz

## 2.4. Connection

En este ejemplo vemos la implementación de un servicio que envía una notificación que contiene las direcciones de todos los clientes conectados al servicio sobre Socket.IO. Para enviar la notificación a todos los clientes suscritos cada vez que se conecta un nuevo cliente o se desconecta, se usa la función *emit* sobre el array de clientes conectados.

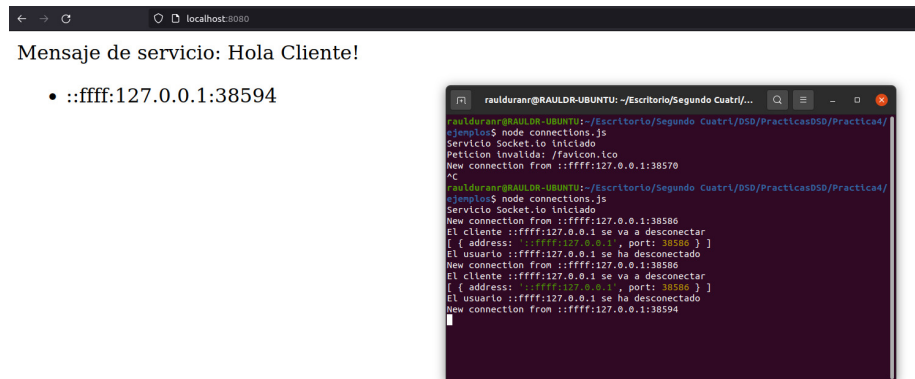


Figura 7: Ejemplo de conexión con Socket.IO

## 2.5. Test MongoDB

Como base de datos para nuestro ejercicio usaremos MongoDB, una base de datos de tipo NoSQL, la cual nos permite guardar información no estructurada. Así, cada entrada de una base de datos MongoDB podrá tener un número variable de parejas claves-valor asociados mediante colecciones. En el ejemplo que se nos ha dado tenemos un servicio que recibe dos tipos de notificaciones mediante Socket.io: "ponerz" y "obtener". El primer tipo de notificación lo introduce el servicio en la base de datos "pruebaBaseDatos", dentro de la colección de claves-valor "test". Cuando se recibe el segundo tipo de notificación, el servicio hace una consulta en base al contenido de dicha notificación, y devuelve el resultado al cliente. Cuando un cliente se conecta, el servicio le devuelve su dirección de conexión.

Lo que hace el cliente en este ejemplo es introducir en la base de datos sus datos: host, puerto y momento de conexión con el servicio. Tras esto, intenta recuperar de la base de datos todo el historial de conexiones realizadas desde el host del cliente.

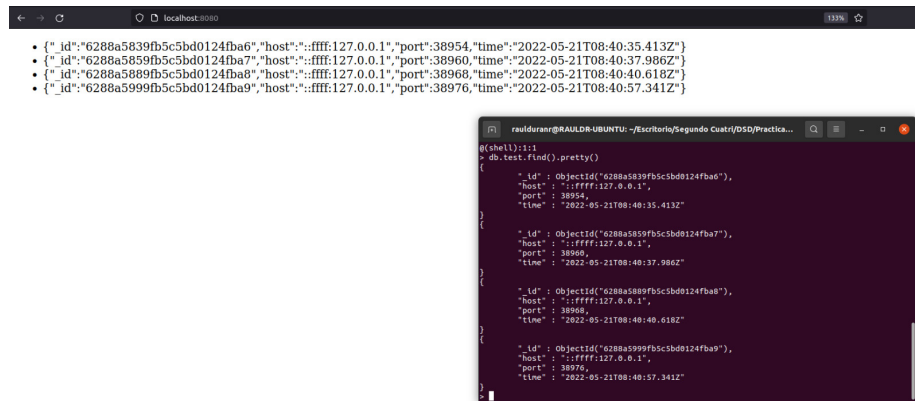


Figura 8: Ejemplo de mongodb

Como se ve en la imagen, cada entrada se corresponde con las veces que el cliente web accede, es decir, cada vez que recarguemos la página y el cliente vuelva a conectarse, se añadirá una entrada en la base de datos. Hay que añadir que para el correcto funcionamiento de MongoDB con Node.js, la versión de Node.js debe ser igual o superior a la v.16, por lo que, en mi caso, he tenido que actualizar Node.js, usando a partir de ahora *node* en lugar de *nodejs*:

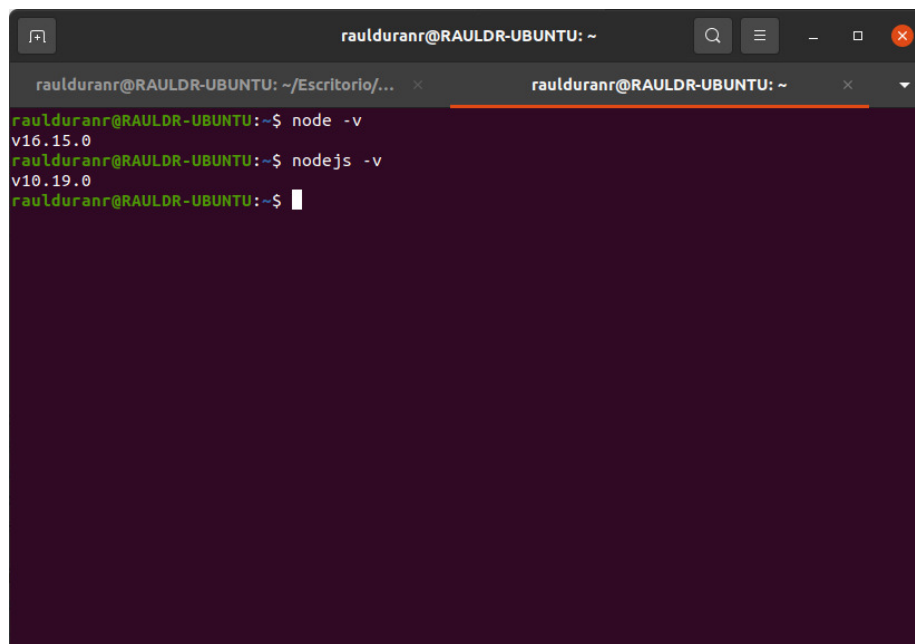


Figura 9: Version de node y nodejs

### 3. Ejercicio: Casa Domótica

El ejercicio que se nos propone consiste en desarrollar un sistema de una casa domótica, es decir, un sistema con sensores y actuadores que funcionen respecto a los datos de un servidor, y con un agente que notifique las alarmas y sea capaz de tomar decisiones básicas.

Ahora es el momento de demostrar que hemos entendido el funcionamiento de los ejemplos.

Para trabajar de manera más cómoda e intuitiva, lo primero que he hecho ha sido diseñar la interfaz de la aplicación de la casa domótica:

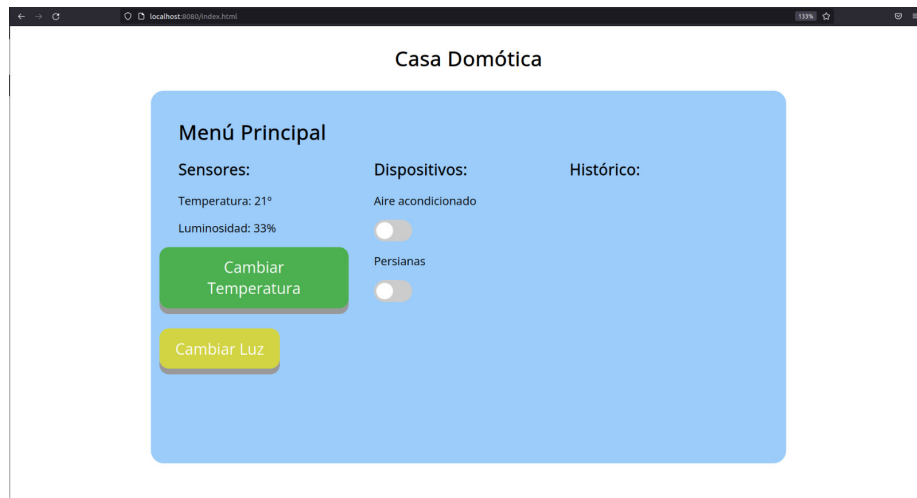


Figura 10: Primera interfaz

Con la interfaz ya diseñada, resulta más sencillo entender cuáles son los elementos que necesitamos. En primer lugar, hay que hacer que los clientes y los servidores se comuniquen correctamente, y que el servidor acepte y actualice la información a los clientes sin problemas.

Para comprobar esto, pedimos al servidor que se cambie la temperatura o la luz, dependiendo del botón pulsado, de manera aleatoria, actualizando el dato que se muestra en el apartado de *Sensores*. Esto lo conseguimos haciendo uso de Socket.io:

```

//Creamos una instancia de un objeto Socket.io
var io = socketIO(httpServer);
//Escuchamos el evento de conexion
io.on("connection", (socket) => {
  console.log("Usuario conectado");
  //Cambia la temperatura cuando recibe:
  socket.on("change temp", (temp) => {
    console.log("Nueva temperatura: " + temp);
    io.emit("change temp", temp);
  });
  //Cambia la luz cuando recibe:
  socket.on("change luz", (luz) => {
    console.log("Nueva luz: " + luz);
    io.emit("change luz", luz);
  });
  //Evento de desconexion
  socket.on("disconnect", () => {
    console.log("Usuario desconectado");
  })
})
})

```

Figura 11: Código de la parte del Socket del archivo JavaScript

```

<script src="/socket.io/socket.io.js"></script>
<script type="text/javascript">
  var socket = io();

  var divTemp = document.getElementById("dato-temp");
  var divLuz = document.getElementById("dato-luz");
  var botonT = document.getElementById("temp-btn");
  var botonL = document.getElementById("luz-btn");

  //Cuando se pulse el botón, cambiará la temperatura aleatoriamente entre 10 y 40
  botonT.addEventListener("click", function(e){
    e.preventDefault();
    var temp = (Math.floor(Math.random()*30)+10) + "°";
    socket.emit("change temp", temp);
  });

  //Cuando se pulse el boton, cambiará la luz entre 10 y 90 %
  botonL.addEventListener("click", function(e){
    e.preventDefault();
    var luz = (Math.floor(Math.random()*90)+10) + "%";
    socket.emit("change luz", luz);
  });

  //Cuando recibamos el change en el cliente, habrá que actualizar las entradas de la pagina
  socket.on("change temp", function(temp){
    var span = document.getElementById("spanTemp");
    span.textContent = temp;
    //divTemp.appendChild(p);
  });
  socket.on("change luz", function(luz){
    var span = document.getElementById("spanLuz");
    span.textContent = luz;
  });
</script>

```

Figura 12: Código de la parte del Socket del archivo HTML

Una vez conseguida esta comunicación, hacemos que se modifiquen también los *toggle buttons* para activar y desactivar los dispositivos. Una vez tenemos la interfaz preparada, funcionando correctamente con Socket.IO, es hora de utilizar MongoDB para almacenar los datos de los sensores en el histórico. Por comodidad, he utilizado **MongoDB Compass** para trabajar fácilmente mediante una interfaz con las bases de datos de MongoDB:

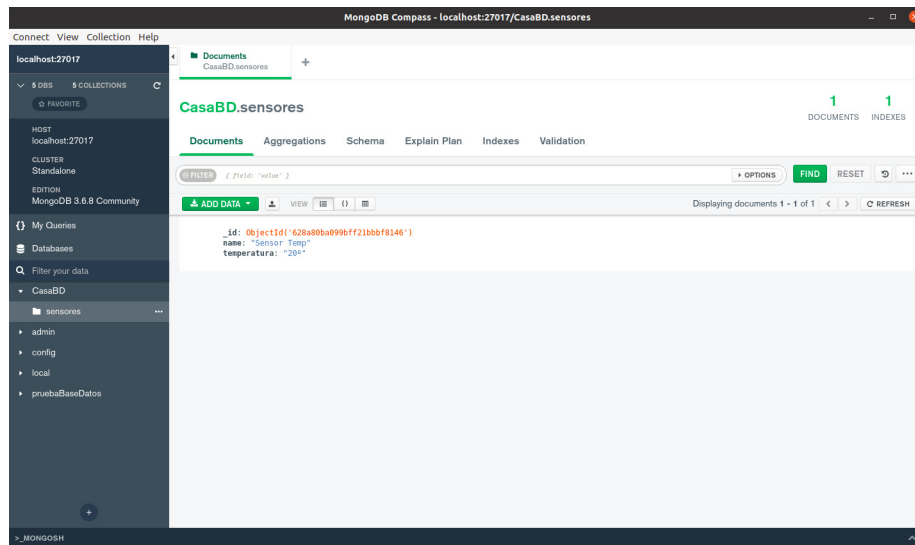


Figura 13: MongoDB Compass

Para el correcto funcionamiento de MongoDB, hay que crear un cliente de Mongo, el cual se conectará a nuestra dirección. Es dentro de esta conexión donde declaramos nuestra instancia de un objeto Socket.IO y ponemos a escuchar al servidor en el puerto deseado, por ejemplo el 8081.

Con el cliente de Mongo conectado, crearemos nuestra base de datos y las colecciones. Algo importante a tener en cuenta es que cuando vamos a insertar un documento en una de las colecciones, dicha colección se creará si no existía, lo cual nos ahorra tener que crear colecciones y tener que comprobar si ya existen antes de crearlas.

Dentro de esta conexión, tendremos que incluir las llamadas socket que vimos anteriormente, añadiendo las nuevas comunicaciones para gestionar el Histórico de eventos, en el cual se mostrará cada vez que se active/desactive un dispositivo, indicando si el que lo hace es el agente al superar un umbral definido.

En cuanto al agente, lo he implementado en el cliente, donde estará escuchando el mismo mensaje que usamos para cambiar los valores de temperatura y luminosidad. Si el valor que recibe el agente supera ciertos umbrales, es entonces cuando actúa, informando al servidor de que va a activar/desactivar un dispositivo, guardándose en el histórico.



Para ver esto más claro, dejo una imagen de ejemplo:

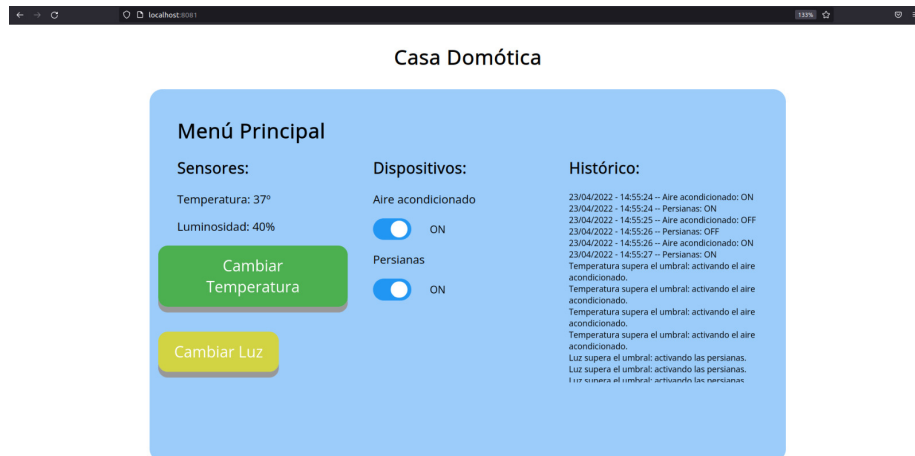


Figura 14: Interfaz de un cliente con histórico

Y podemos ver que la colección se ha creado y actualizado correctamente:

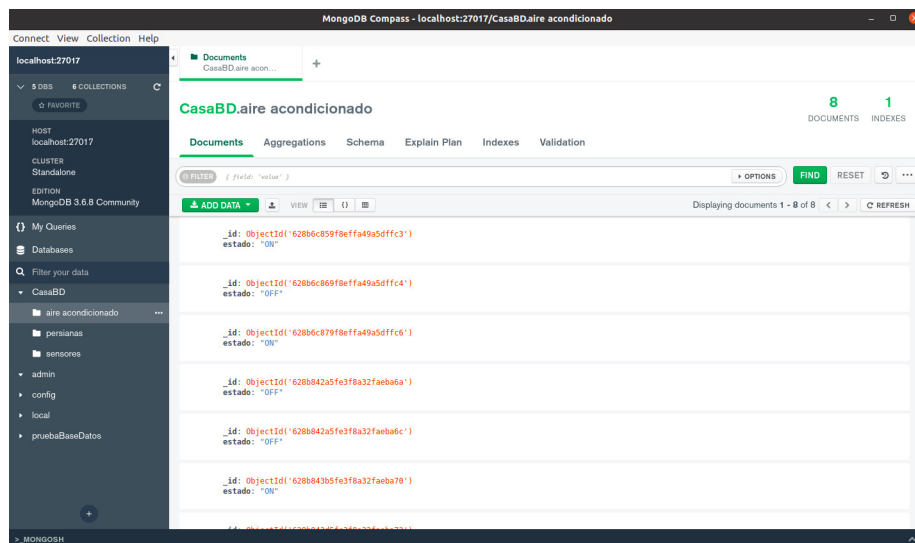


Figura 15: colección Aire acondicionado en Compass