Memoria práctica 4: NodeJS

Alumno: Ahmed El Moukhtari Koubaa

Asignatura: Desarrollo de sistemas distribuidos

Subgrupo: DSD3

En esta memoria se trata de abordar la solución que hemos planteado a esta práctica de manera breve y concisa, así como facilitar la tarea de corrección al corrector. A continuación, se elabora un pequeño esquema conceptual sobre cómo se organiza esta memoria y en qué partes fundamentales está dividida nuestra práctica, cada sección está mucho más detallada:

1º Ejemplos implementados y explicados
   - Hello world
   - Calculadora
   - Connections
   - Mongo

2º Solución al ejercicio: versión básica
   - Servidor
   - Sensores
   - Actuadores
   - Agente
   - Usuarios

3º Solución al ejercicio: versión extendida
   - Servidor
   - Sensores
   - Actuadores
   - Agente
   - Usuarios
   - Nuevos sensores
   - Frontend mejorado
   - Simulación de sensores mediante APIs
   - Notificación a usuario mediante aplicaciones de servicio de mensajería
   - Comunicación y control del sistema mediante mensajes desde otra app

4º Conclusión

1º Ejemplos implementados y explicados

Se nos facilita el código de un conjunto de ejemplos sobre el funcionamiento básico de NodeJS así como de algunas bibliotecas que se importan y usan (socket.io, mongodb, etc). Debemos realizar la implementación de dicho código y explicar su funcionamiento. Todos los ejemplos han sido implementados, comentados y algunos ligeramente modificados (para solventar errores previos, experimentar, mejorar la eficiencia, etc). Los principales ejemplos son 4 y se detallan a continuación.

- Hello world

  Este es el ejemplo más básico de todos. Es un programa mínimo sobre el funcionamiento de nodejs. En este ejemplo vemos cómo se importa la biblioteca http, con ella podemos crear un servicio definiendo su funcionalidad. La función que se le pasa a este servicio recibe peticiones, las muestra por pantalla y elabora un mensaje de respuesta con el texto “Hola mundo”.

- Calculadora

  Hay varios ficheros relacionados con la calculadora. La principal función de la calculadora es mostrarnos a procesar la URL que nos llega con la petición que nos realizan. Así pues, establecemos un formato de URL consistente en <dirección del servicio>/<acción>/<arg1>/<arg2>, así podemos recibir la operación a realizar con los argumentos de la misma y respondemos con un mensaje http. Una extensión es la calculadora-web, igual que el caso anterior, pero esta vez el servicio abre un fichero html y lo devuelve por mensaje, así el usuario tiene una vista gráfica en la que puede introducir la acción que desea y enviarla con un submit button sin necesidad de entender el formato URL. Internamente el html genera la URL y realiza una petición usando XMLHttpRequest.

- Connections

  Se nos muestra aquí a mantener una conexión con el cliente, de manera que seamos capaces de enviarle mensajes sin tener que esperar a una petición directa, es decir, el servicio conocerá al cliente y cuando lo desee la enviará mensajes. Aprendemos con este ejemplo a mantener conexiones y a definir eventos y qué hacer cuando recibimos dicho evento además de cómo emitirlo. Podríamos decir que se basa en el patrón publish-suscriber. Usamos la biblioteca socket.io.

- Mongo

  Con este ejemplo aprendemos a realizar una conexión a la base de datos mongoDB. Además, aprendemos a usar colecciones, si estás existen se usan si no se crean. Un HTML acompaña este ejemplo. Lo que hacemos en él es guardar en la base de datos la dirección IP del cliente que se nos conectó y mostrar el contenido de dicha colección a los clientes que se nos conectan. Así pues, cuando un cliente se conecta recibe el contenido de la colección en la que se encuentra el registro de clientes. Usamos socket.io para mantener las conexiones y procesarlas.

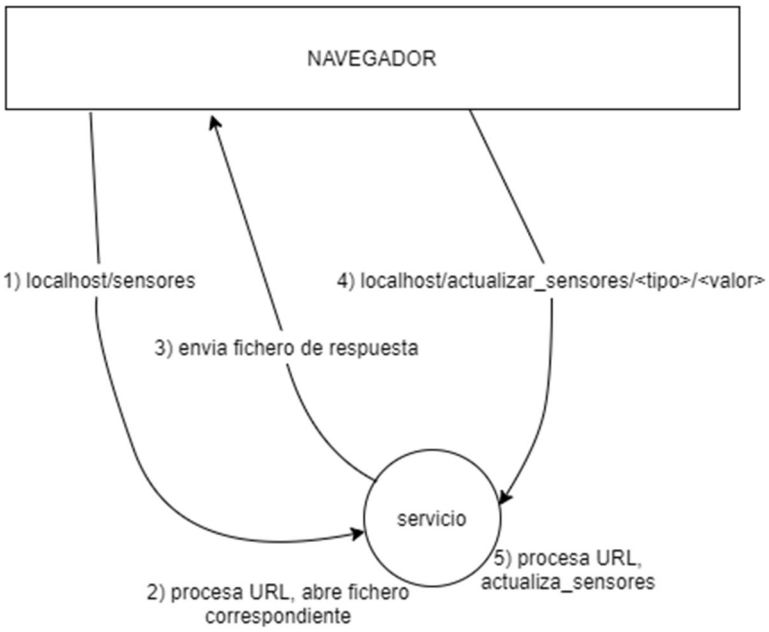
2º Solución al ejercicio: versión básica

En la versión básica se responde a lo que se nos pide en el enunciado del ejercicio propuesto. Tenemos así 5 entidades, que son:

- Servidor: recibe peticiones y las procesa, es el núcleo del sistema.
- Usuario: desde un panel revisa el estado del sistema (sensores, actuadores).
- Sensores: envían mediciones al servidor.
- Actuadores: controlan elementos en casa (usados más por usuario)
- Agente: módulo lógico que aporta inteligencia al sistema.

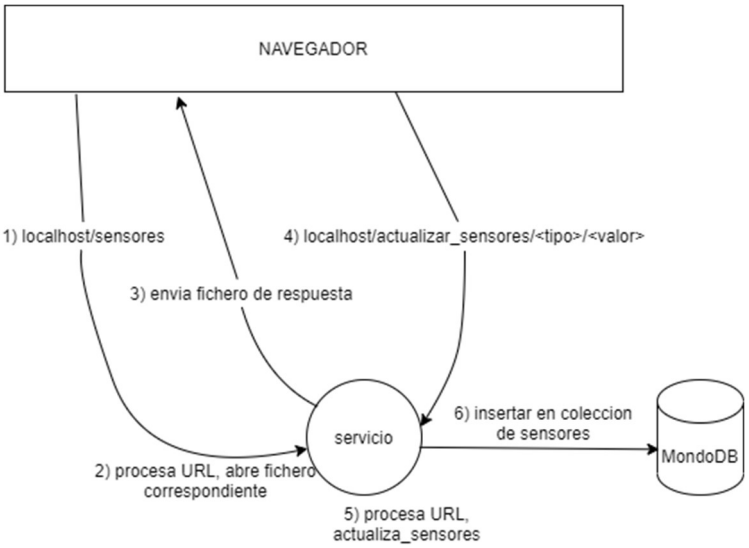
Nuestra solución comienza en servidor.js donde se crea el servicio y en este se define el procesamiento de peticiones mediante URLs con métodos y parámetros. El servicio en cuestión es capaz de abrir ficheros del sistema y elaborar una respuesta al cliente enviando dicho fichero (HTML).

Las peticiones por URL parametrizadas son realizadas principalmente por los sensores, para ello. Los sensores son un fichero HTML en donde se muestra un formulario para introducir los valores de dichos sensores (en la realidad este formulario no existe como tal, sino que son los sensores quienes envían sus propias mediciones). Para usar los sensores se accede a http://localhost:8080/sensores y desde ahí se genera la URL parametrizada mediante el formulario. La primera palabra entre "/" debe ser la acción, actualmente solo responde a la acción "actualizar_sensores" pero está programado de esta manera para que si en un futuro necesitamos desarrollar una API sea mucho más fácil e intuitivo de hacer.



Por servidor y servicio nos referimos a lo mismo, usamos la palabra servidor por ser la terminología que nos propuso el ejercicio a resolver y decimos servicio porque es lo que realmente estamos usando.

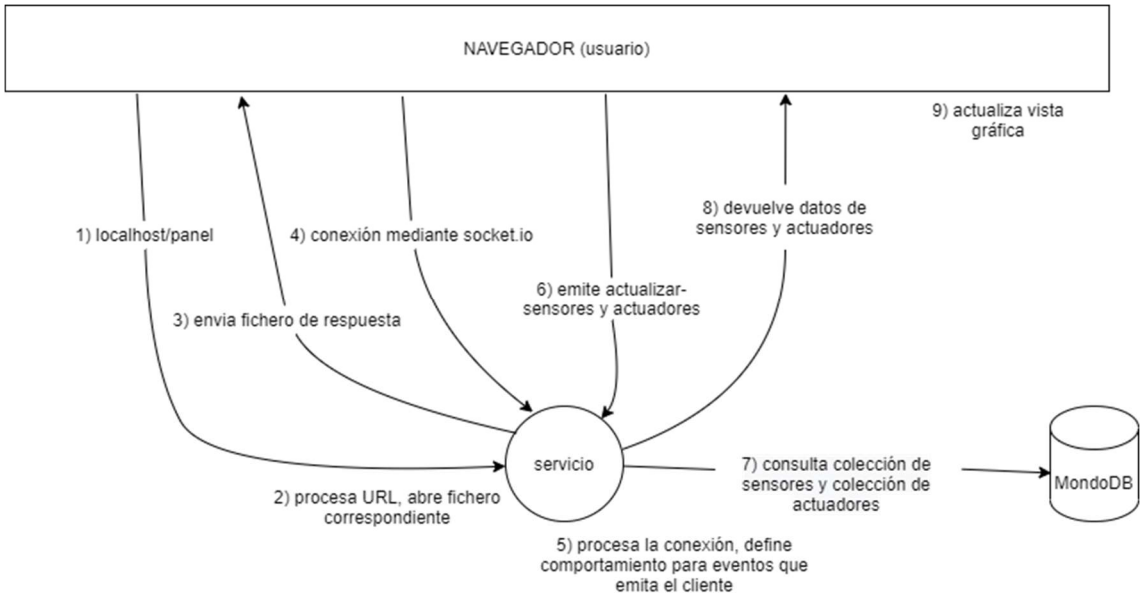
Una vez tenemos los sensores funcionando y el servicio procesando las mediciones que estos nos dan, lo que añadimos al sistema es una base de datos. Ahora el servicio se conecta a una base de datos mongodb y crea la colección de sensores, donde se guardan los sensores con dos atributos fundamentales, tipo (hacer referencia al tipo de sensor que es) y valor (medición que nos manda dicho sensor), además de un atributo time para registrar en qué momento fue tomada dicha medición. Así pues, cuando recibimos una petición de actualizar sensores, los actualizamos y los registramos en la base de datos.



Así es como queda el diagrama de funcionamiento de nuestro sistema tras conectar la base de datos. Cuando se incluya el usuario al diagrama, este recibirá un mensaje cada vez que se actualicen los sensores.

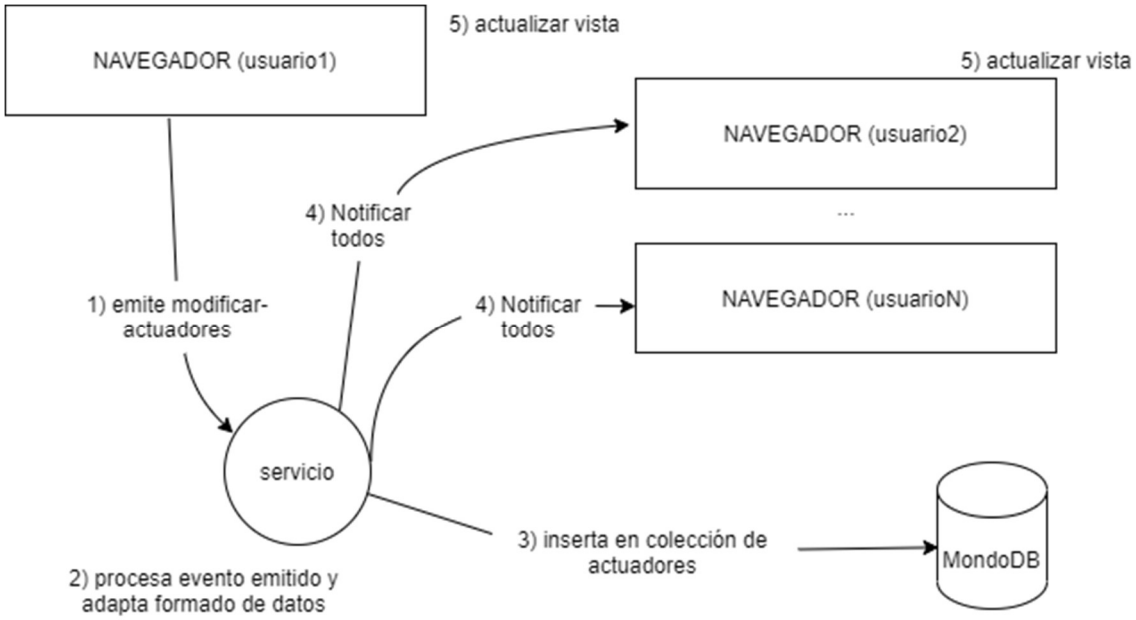
Llegados a este punto ya podemos ofrecerle información de valor al usuario. El usuario deberá conectarse a nuestro sistema para revisar algunos datos y mandarnos otros. Para esto creamos un panel que le facilite dicha tarea al usuario. Usamos el fichero panel.html para dotar de funcionalidad al usuario.

Como el usuario debe mantener una conexión activa con el servidor y estar al tanto de los cambios que se produzcan en el sistema, usamos socket.io. En la parte del servidor, definimos el comportamiento al recibir la conexión de un cliente (cualquiera que se conecte a nuestro servicio) y dentro de esta definición definimos el comportamiento ante eventos que nos mande un cliente, principalmente “actualizar-sensores” y “actualizar-actuadores”. Cuando un cliente emite un evento actualizar-<algo> lo entendemos y lo tratamos como un GET, una petición de lectura y lo que hacemos es una consulta a la base de datos buscando el último registro que añadimos y devolviéndoselo al usuario mediante el panel.HTML. En panel.html también se hace uso de socket.io y se define el comportamiento ante ciertos eventos, como “actualizar-sensores” entonces la interfaz gráfica se actualiza mostrando los datos pasados al evento. Para entender mejor los eventos y los comportamientos recomiendo leer el código y los comentarios que hay.



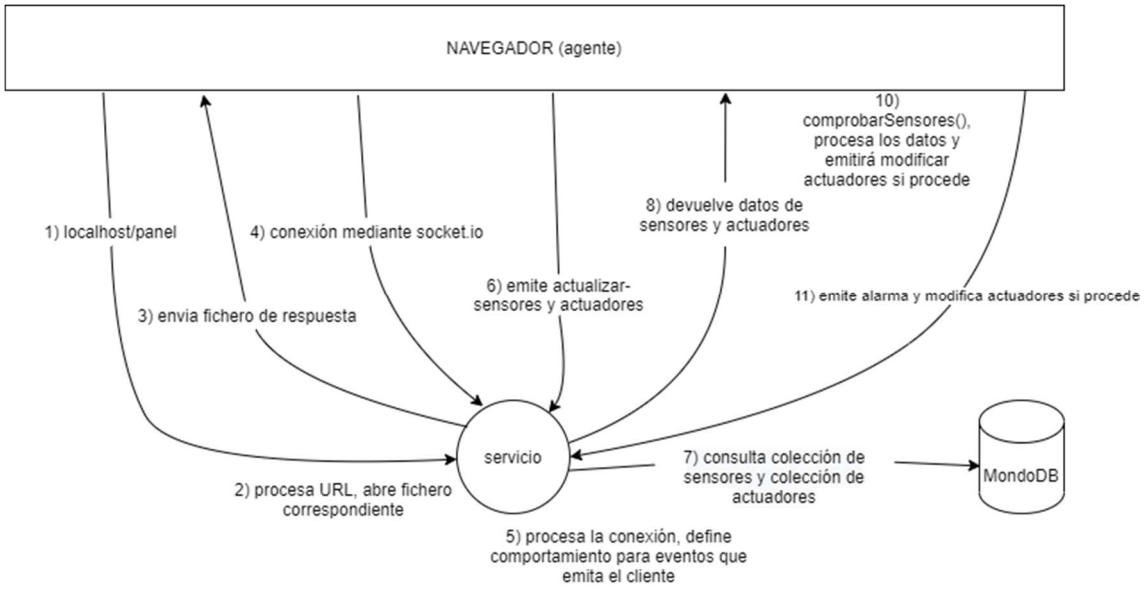
En la imagen de arriba podemos ver el proceso que ocurre cuando un usuario se conecta a nuestro servicio para ver el panel de usuario.

Como nos proponía el enunciado del ejercicio propuesto, además de ver el estado de los sensores y actuadores, el usuario podría también modificar los actuadores del sistema. Recordamos que nuestro sistema está diseñado para gestionar un hogar y los diferentes elementos inteligentes que en él se encuentren, por lo que es muy necesario el uso de actuadores, por ejemplo, para subir la temperatura del aire acondicionado, encender o apagar las luces, poner el horno a calentar, subir la temperatura del frigorífico, usar las cámaras de vigilancia, etc. Nuestro sistema usa dos actuadores básicos, uno para persianas y otro para aire acondicionado. Para modificar el valor de los actuadores, definimos el evento modificar-actuadores. Dicho evento es emitido por el usuario desde el panel y debe ser transmitido por el servidor a todos los demás usuarios.

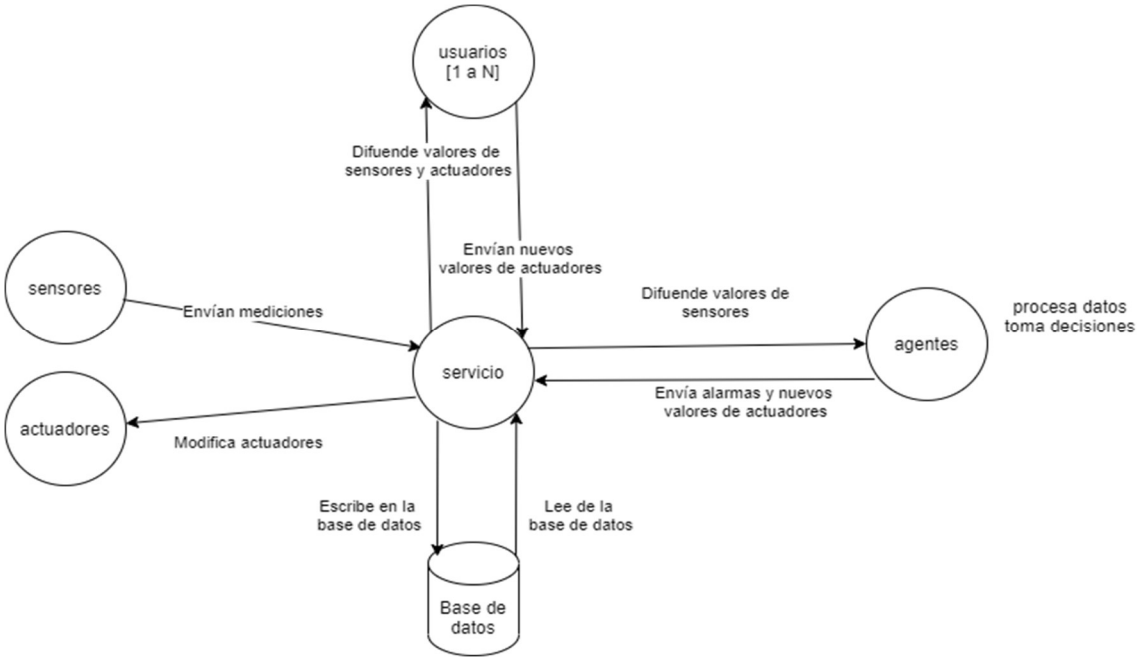


Cuando se modifican los actuadores se registran los nuevos valores en la colección correspondiente, esto no lo solicita el ejercicio a resolver, pero lo hacemos debido a que en el futuro nos podría interesar para aprender de nuestros usuarios y adaptarnos a ellos, por ejemplo, todos los días a las 3 de la tarde se bajan las persianas, tal vez el sistema debería hacerlo autónomamente. Por otro lado, al modificar los actuadores notificamos a todos los usuarios que esto ha ocurrido para que se actualicen las vistas ya que más de un usuario podría estar revisando el panel y pensando en modificar los actuadores, si uno lo hace y el otro no lo sabe volvería a modificarlos sin saber el valor real de estos.

Llegados a este punto nuestro sistema está casi completo, solo nos falta el agente. El agente se percibe como un módulo para dotar de inteligencia a nuestro sistema. El agente será capaz de procesar los datos de los sensores y actuar en base a estos enviando alarmas al servidor, modificando los actuadores, etc. A nivel de implementación dicho agente es como un usuario más, solo que no accede usando el panel sino su propia interfaz agente.HTML. Al igual que cualquier otro usuario, recibe un mensaje cuando se actualizan los sensores y es capaz de modificar los actuadores usando una función que le permite determinar si procede o no, dicha función es *comprobarSensores*.



Con esto nuestro sistema ya estaría completo, ahora podemos unir todos los diagramas con todas las entidades para obtener una vista general del funcionamiento del sistema.



En la imagen de arriba podemos ver de manera resumida cuál es la arquitectura de nuestro sistema, esto es, cómo se organizan los diferentes componentes y cómo se comunican entre sí. Hemos detallado más arriba cómo se realiza cada comunicación y cómo se lleva a cabo cada función, pero para entenderlo de la mejor manera posible se recomienda leer el código y los comentarios que lo acompañan.

3º Solución al ejercicio: versión extendida

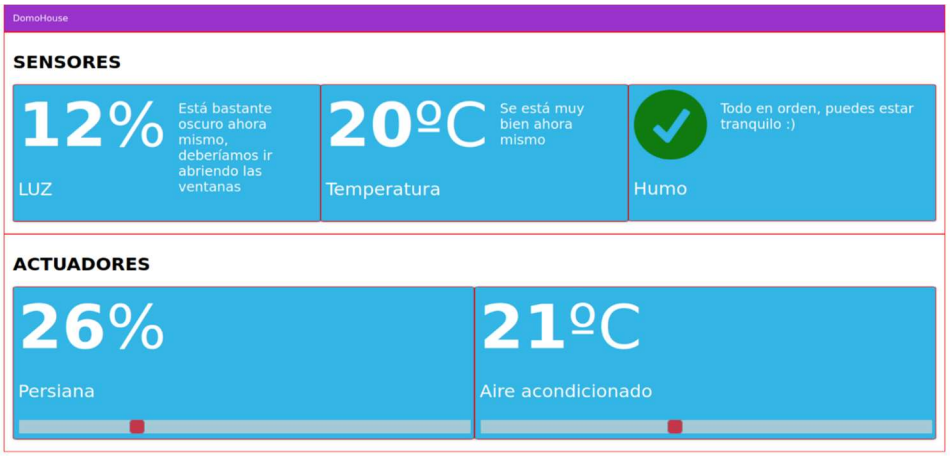
A partir de ahora llamaremos a nuestro sistema DomoHouse. La versión extendida se basa en la versión básica, mantiene toda su arquitectura y modelo de funcionamiento y añade algunas funcionalidades que se detallarán a continuación. Internamente se ha programado el sistema de manera un tanto diferente para que sea más fácil extender su funcionalidad en el futuro, ahora los sensores no son fijos si no que pueden variar en tiempo de ejecución, se han desacoplado todavía más los componentes del sistema y se han creado funciones para funcionalidades comunes.

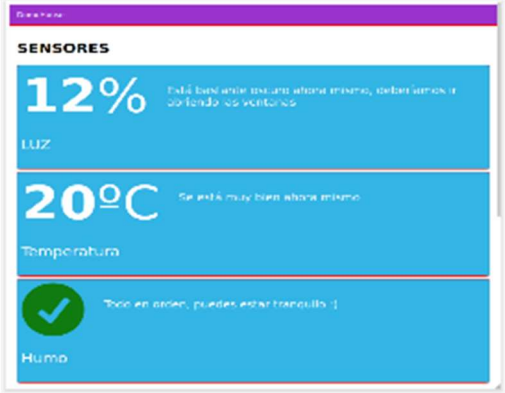
- Nuevos sensores

  Se ha preparado el sistema para que añadir un nuevo sensor no suponga un desafío y se ha añadido un nuevo sensor “detector de humo”. Este sensor es muy básico, pero muy interesante. Puede tomar como valor 0 o 1. Si en algún momento este sensor no indica 0, entonces el agente emite una alarma por fuego (realmente es por humo, pero al usuario se le notifica que hay fuego en casa).

- Frontend mejorado

  Se ha mejorado la interfaz gráfica del panel. Este panel es realmente la única interfaz necesaria en el sistema ya que es la única que será vista y accedida por el usuario. Se ha añadido para eso una hoja de estilos css y se ha modificado el fichero panel.HTML. Se ha dotado a esta página de un diseño responsive que actualmente se ve bien en diferentes móviles. Las representaciones de los sensores se han agrupado en una fila superior, cada icono tiene nombre del sensor, valor y un mensaje en base al valor actual. Los actuadores se encuentran en la fila inferior, ahora son sliders, por lo que se introduce el valor deslizando, es mucho más cómodo que hacerlo mediante un formulario.

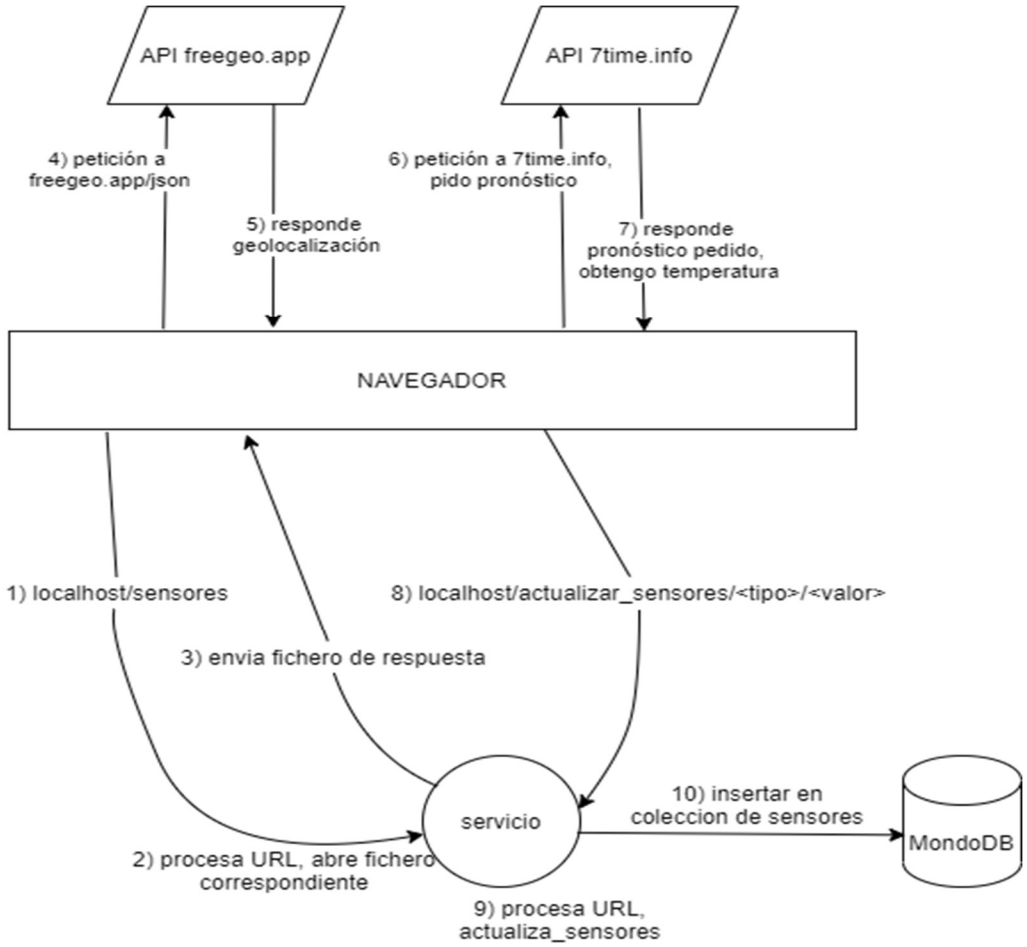




- Simulación de sensores mediante APIs

  Hemos añadido un módulo de simulación a sensores.HTML. Dicha simulación se puede activar o desactivar y consiste en dar unos valores aproximados de la temperatura que hay en casa actualmente, es decir, mejoramos la implementación de los sensores haciendo DomoHouse más realista. Para llevar a cabo esta simulación hacemos uso de varias APIs. La primera de ellas es freegeo.app/json, esta API nos devuelve información geográfica basándose en nuestra IP. Los datos que nos interesa obtener, son la longitud y la latitud, por lo que después de hacerle una petición y obtener la respuesta en formato JSON, extraemos dichos datos.

  Una vez obtenida la longitud y la latitud, hacemos una llamada a otra API, 7time.info. Esta API nos devuelve la predicción del tiempo basándose en nuestras coordenadas geográficas, la predicción devuelta es a 3 días divida en vectores que se corresponden con franjas de tiempo (de 3 horas aproximadamente). Como nos interesa la temperatura actual, cogemos la primera franja según nuestra zona horaria. Una vez obtenida la temperatura, actualizamos la vista y pausamos el proceso durante un intervalo de tiempo (usamos promesas y mecanismos de sincronización básicos de NodeJS), al final del intervalo repetimos el proceso si no se desactivo la simulación.

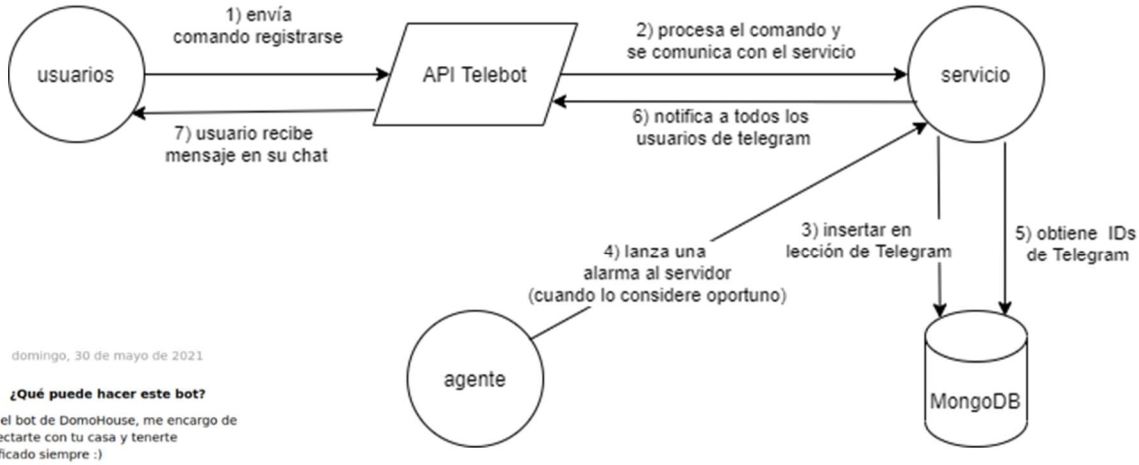


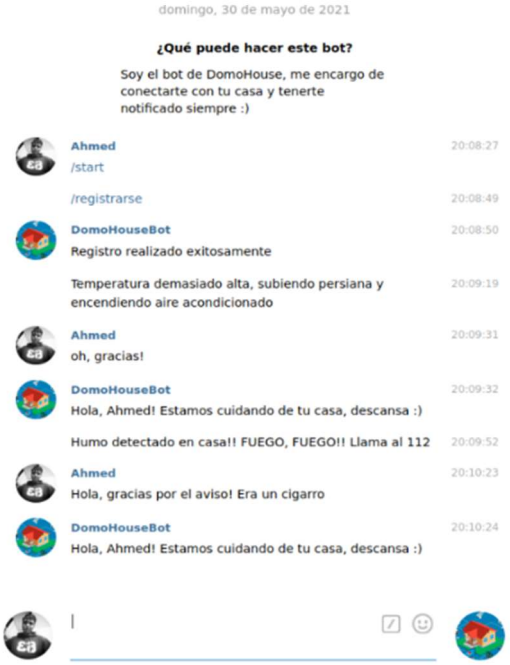
- Notificación a usuarios, mediante servicios de mensajería instantánea

Cuando algo pasa en nuestra casa la mejor manera de enterarnos no es mediante una notificación en una web, ni teniendo que abrir una app para poder actualizar el contenido, no, la mejor manera es recibir una notificación mediante un servicio que usemos a menudo. Entre estos servicios, podrían estar el servicio de correo electrónico, una red social como twitter o un servicio de mensajería instantánea como Whatsapp o Telegram.

En nuestro caso hemos añadido un módulo para recibir notificaciones vía Telegram. Hemos usado para esto una de las diversas APIs que ofrece Telegram para la creación y el control de bots. La API que hemos usado se llama Telebot. Dicha API requiere ser instalada y luego importada como cualquier otra biblioteca (ver el código para más detalles). Para usarla solo necesitamos crear un bot y obtener su API token, tras eso podremos usar la API para mandar y recibir mensajes.

Definimos el comportamiento de nuestro bot cuando recibamos ciertos eventos. El primer evento que necesitamos, es el comando /registrarse, cuando se emite dicho evento guardamos el id del emisor en la base de datos y cuando el agente detecte una alarma (de cualquier tipo, temperatura, luz, humo, etc) se le envía un mensaje a todos los usuarios que se registraron mediante Telegram.





Nuestro bot se llama domohouse_dsd_bot en Telegram. Nos registramos, hacemos que salten la alarmas y vemos que nos ha notificado perfectamente.

- Comunicación y control mediante otras aplicaciones

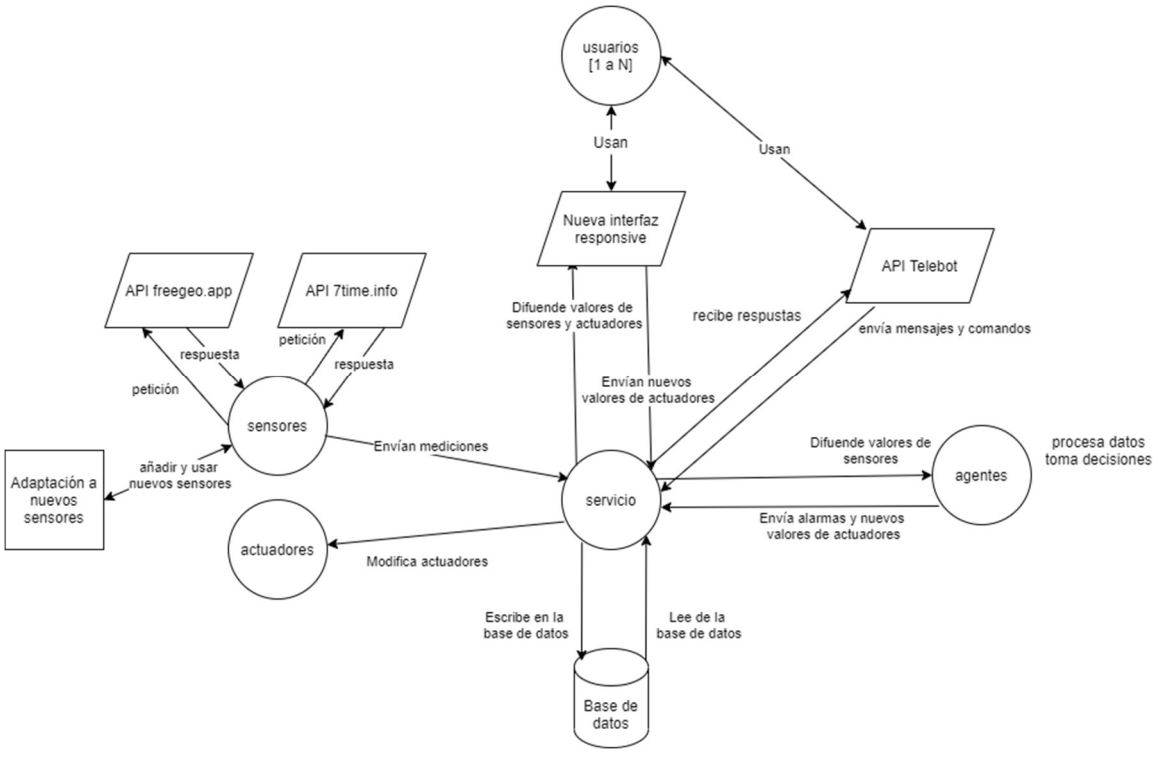
  Volvemos a usar nuestro bot de Telegram y extendemos su funcionalidad, no solo se podrá obtener información de él, sino que también la podremos introducir. Definimos algunos comandos más como subir temperatura, bajarla, subir el aire, etc. Un nuevo comando “info” nos aclara qué es lo que hace exactamente el bot y un comando estado nos resume el estado del sistema (sensores, actuadores, alarmas, etc).

Con esto conseguimos una nueva manera de interaccionar con nuestro sistema, ya no es necesario usar directamente el panel. Gracias a esto tenemos mayor desacoplamiento, reusabilidad del código del servicio y compatibilidad con otros servicios. Por ahora los comandos que escucha y entiende el bot son los siguientes:

- Info
- Estado
- Registrarse
- Subir_persiana
- Bajar_persiana
- Subir_aire
- Bajar_aire

Para ver cómo se tratan dichos comandos recomiendo ver el código de servidor.js, al final de este están documentados los eventos que el bot recibe y cómo actúa internamente.

El diagrama final del sistema con todas las entidades relacionadas es el siguiente.



4º Conclusión

Tras haber realizado el ejercicio propuesto y demás tareas que se nos pedían en el enunciado, podemos afirmar que hemos aprendido cuál es el funcionamiento básico de NodeJS.

Entre otras cosas hemos aprendido a realizar peticiones http a servicios, no solo a los nuestros sino también a otras APIS. Mientras buscábamos APIs para usar, nos hemos encontrado con que grandes empresas ya ofrecen una al público, el problema está en que la mayoría son de pago, esto nos ha limitado un tanto. Tras una búsqueda en profundidad hemos encontrado APIs open source muy útiles. Me atrevería a decir que comprendo bastante mejor el funcionamiento de una API y sé cómo usarlas, además de que me queda más claro la filosofía de los servicios y microservicios.

Hemos manejado conexiones con usuarios mediante socket.io, mantener la conexión, emitir eventos, definir comportamientos ante eventos externos que se nos emitan, etc. También hemos mejorado nuestra compresión sobre bases de datos no relacionales.

Como resultado de aprender sobre socket.io y las APIs hemos podido manejar una API más compleja como es la API Telebot. Hemos mantenido conexiones con los usuarios que se conectaban mediante Telegram, hemos procesado sus mensajes actuando en consecuencia y les hemos ofrecido una respuesta.

Como aspectos negativos:

- Los registros de sensores se actualizan uno a uno en la base de datos, en la realidad podríamos recibir montones de mediciones por segundo, esto podría llegar a suponer un cuello de botella en la base de datos. Lo ideal sería tener un servicio a parte que recibiese los mensajes y cuando tuviese una cantidad los enviase al servidor para introducirlos de una vez.

- Las APIs que usamos para simulación devuelven más datos de los que necesitamos, deberíamos intentar hacer dichas consultas sólo cuando desconozcamos los datos que vamos a pedir, para evitar saturar al sistema.

- Añadir un sensor por parte del servidor es fácil, pero requiere de un cambio en el agente y en el panel. Deberíamos crear un módulo común a todas las partes interesadas para que se evite repetir código y se optimice la compatibilidad.

- Los nuevos sensores o actuadores que se intenten añadir al sistema deberían ser “ilimitados”, es decir, todos aquellos que cumplan con el formato de nuestra API (que realicen peticiones como lo esperamos). Los sensores podrían emitir una acción de registro a nuestro sistema pasándonos un controlador de agente para añadirlo al módulo de agentes, así conseguiríamos mantener la funcionalidad y maximizar la compatibilidad de todo el sistema.