Doge Power

Collar de Perro Inteligente

Introducción

Cada vez son más las personas que tienen un animal de compañía; en España, el porcentaje alcanza el 40% (según la Asociación Madrileña de Veterinarios de Animales de Compañía), de los cuales el 60% son de perros. A pesar de un porcentaje tan alto, el mundo de las mascotas y la tecnología se encuentran actualmente muy alejados, lo que deja un amplio campo de ideas para facilitar el cuidado de estos por parte de sus dueños. Es por esto por lo que hemos decidido desarrollar un collar que facilite el cuidado de estos gracias a funciones como localizador GPS, sistema anti-escape y modo amaestramiento entre otros.

Para su desarrollo, hemos planteado mecanismos que a priori suponemos que serán factibles, tanto a nivel de conocimientos como económico, por lo que es posible que conforme avance el proyecto vayamos añadiendo y/o eliminando funcionalidades.

El producto va destinado principalmente a los dueños de perros de todas las razas, pero su uso también es adecuado para otras mascotas como gatos, hurones, caballos, etc., pero no descartamos que alguien quiera usarlo para, por ejemplo, una iguana.

Funcionalidades

- Sistema anti-escape

  Una de las principales funciones que trataremos de implementar es un sistema por el cual, si se detecta que el perro entra en una zona cercana a una puerta abierta, esta se cerrará. Para ello, una posible implementación sería un sensor en el collar y otro en la puerta, que, al estar a cierta distancia, activa un motor que la cierra. Un detalle para tener en cuenta es que la puerta puede golpear al perro, por ello deberemos cerrar la puerta con poca fuerza o implementar un sensor parecido al de las puertas de cochera, que detecta si hay un objeto en medio para detener el motor.

  Además, si el perro consigue escapar, se notificará al dueño de esto. Usaremos un sensor que detecta si el collar se encuentra dentro de cierta zona, y para notificar usaremos email o vía aplicación.

- Localizador

  Mediante un módulo GPS, podremos obtener la ubicación de la mascota en tiempo real. El usuario mandará una petición al servidor, y este obtendrá la ubicación aproximada.

Alcántara Laguna, Alejandro
Chamber González de Quevedo, Enrique

- Luz

  El usuario podrá enviar una petición para encender un led y así poder encontrar a su perro en situaciones de poca visibilidad

- Alerta por alarma y streaming

  El collar implementará un micrófono que, en caso de detectar un ruido fuerte, enviará una notificación al usuario, que podrá comenzar una conexión para escuchar lo que ocurre. El objetivo es conocer cuando el perro detecta un peligro, como por ejemplo un intruso, ya que este comenzará a ladrar.

- Sensor ahorcamiento

  En caso de que el collar quede enganchado y exista peligro de asfixia, se enviará una notificación al usuario. Se colocará un sensor de presión en el cuello del perro para detectarlo.

- Chip NFC

  En caso de perderse, se podrá escanear un chip NFC con datos importantes para que pueda ser llevado con su dueño, ya que así no es necesario llevarlo a un veterinario para que lea el típico chip que llevan.

- Modo amaestramiento

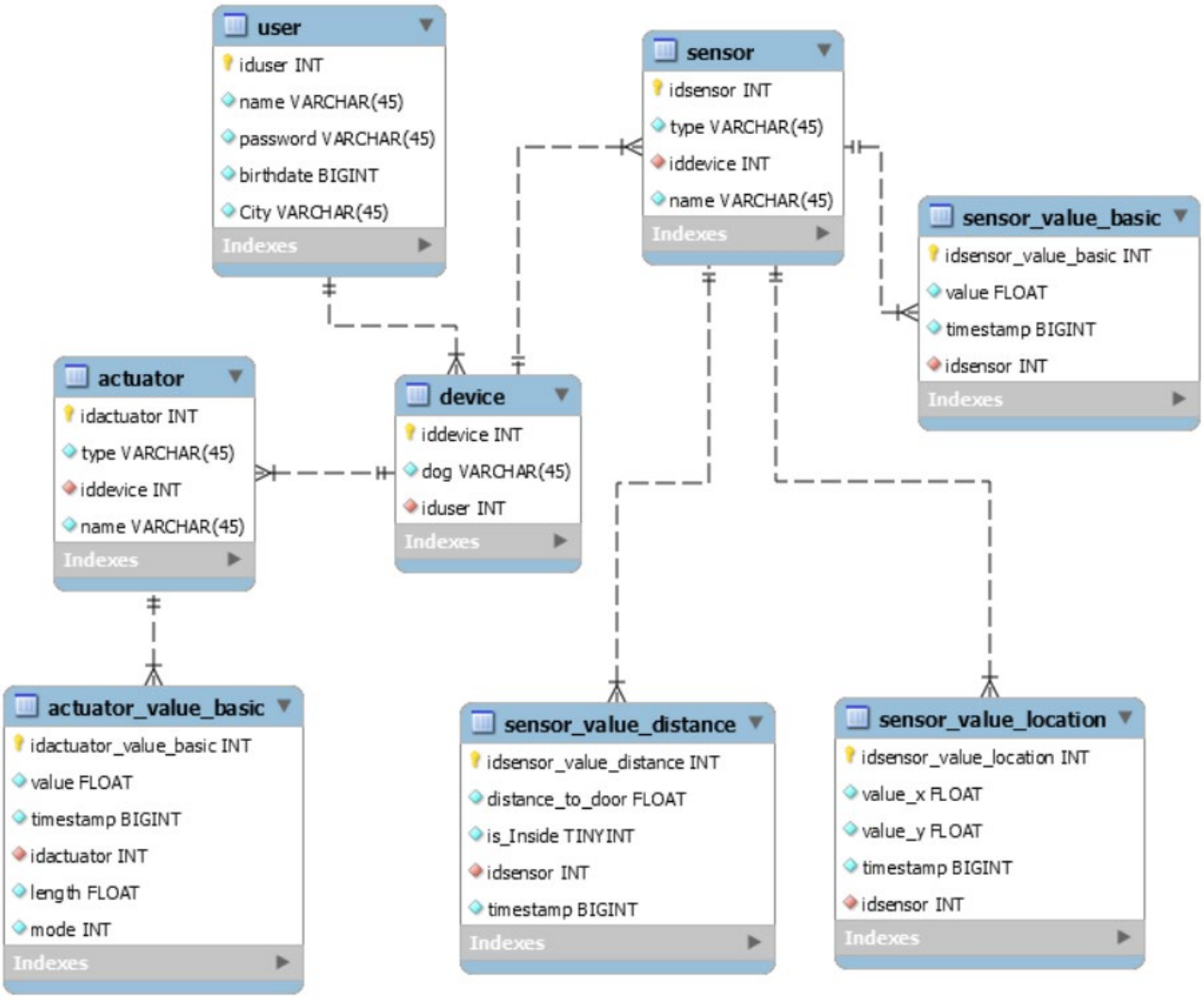
  El collar tendrá implementado un motor de vibración para funcionar como los collares de amaestramiento. Simplemente el usuario mandará una señal para que el collar vibre con distinta intensidad según la necesidad (limitado para no herir al perro) y así amaestrarlo.

- App (Opcional)

  En caso de disponer del suficiente tiempo, se tratará de implementar una app para dispositivos móviles para el uso del collar y de las notificaciones, en lugar de una web y notificaciones por email.

Base de datos

La base de datos diseñada para nuestro proyecto es relativamente simple, puesto que la única información que almacenaremos será la de los usuarios, dispositivos, sensores y actuadores. Para ello, hemos creado una tabla para cada uno, excepto para los dos últimos, que presentan tablas para la información del sensor/actuador, y por separado tablas para los valores. Estos, por regla general, solo precisan del id del sensor, el valor y el timestamp, pero luego existen otros distintos, como es por ejemplo la ubicación, que precisa de dos coordenadas, además de otras extras. El resultado es el siguiente diagrama UML:



Para casi todas las tablas hemos usado solo atributos básicos con el objetivo de obtener una base de datos simple y que no nos lleve una gran cantidad de tiempo trabajar con ella. Como hemos indicado anteriormente, para sensor y actuator tenemos atributos tales como el dispositivo al que pertenecen, el nombre del sensor/actuador (pressure, led, etc.) y el tipo. Con el tipo nos referimos a si es básico (sensor/actuador, valor y timestamp) o no, que en nuestro caso son el tipo Location, ya que estos sensores recogen el valor de las coordenadas en x e y, y Distance, que además de recoger el valor

de la distancia, también recoge si el dispositivo se encuentra dentro o fuera de la zona, que suele ser una casa, de ahí el nombre del atributo “distance_to_door”.

API REST

En el diseño de la API REST principalmente hemos creado funciones básicas que nos conecten con la base de datos, ya que, tras analizar varias veces el proyecto, no hemos visto necesaria la creación de ninguna función especial. En resumen, nuestra API REST está formada por:

- Métodos GET:

  Hemos creado un método GET para cada tabla. Para evitar un código excesivo, hemos unificado la función para obtención de los valores de los sensores (también para los actuadores) independientemente del tipo de sensor, puesto que, al haber tres tipos distintos, necesitamos tres consultas distintas, pero con un simple switch-case según el tipo, se ha solucionado. Además, tanto para los valores de sensores como para los de actuadores hemos añadido la opción de filtrar mediante timestamp, lo cual es realizable mediante la misma función, ya que esta comprueba si existe el parámetro de timestamp, y dependiendo del caso, la consulta difiere. También hemos creado una función que nos devuelve todos los dispositivos pertenecientes a un usuario dado.

- Métodos POST:

  Para insertar nuevos elementos hemos usado los métodos POST. Uno para cada tabla, excepto para device, actuator y sensor, ya que, creando un dispositivo, creamos automáticamente sus sensores y actuadores, ya que no tiene sentido que este uno se genere sin los otros.

- Métodos PUT:

  Hemos usado los métodos PUT para actualizar, por lo que solo han sido necesarios para actualizar los datos de usuario y del dispositivo.

- Métodos DELETE:

  Al igual que los PUT, solo son necesarios para eliminar usuarios y dispositivos, puesto que no tiene sentido eliminar sensores o actuadores solo, y el historial de valores no puede modificarse.

Para estas peticiones hemos usado URLs intuitivas y cortas. La nomenclatura es “/api/(tipo de elemento)/: (id)” independientemente del tipo de petición, puesto que esto no supone un problema, pero para algunas peticiones es distinta:

- Para obtener los dispositivos de un usuario, en '(tipo de elemento)' colocamos "devicesOf".
- Cuando tratamos con los valores de los sensores, la nomenclatura es: "/api/(sensor o actuator)/values/: (id)".
- Para los nuevos usuarios o dispositivos, en lugar de ': (id)' usamos "new", ya que los parámetros se los pasaremos todos en el cuerpo ("new" no sería necesario, puesto que sin él no tendríamos problemas tampoco, pero lo mantenemos para hacerlo más intuitivo).

Por último, en cuanto al cuerpo de las peticiones para los métodos POST y PUT, simplemente usamos un JSON para las columnas en la base de datos, aunque es necesario aclarar que el cuerpo debe contener todos los atributos, de lo contrario no será efectiva la petición. Esto provoca que debamos introducir también el id en el cuerpo, el cual será ignorado puesto que, en los POST, la BBDD es quien lo asigna, y en los PUT se toma el dato pasado por la URL.

MQTT

El uso de MQTT en nuestro proyecto es corto pero importante, ya que su uso es necesario para un sistema de actuadores eficiente. En nuestro caso, solo nos será necesario el uso de dos canales, uno por actuador, además siendo los dos similares en atributos, por lo que se simplifica aún más.

Para MQTT tendremos dos clases: la primera el servidor, cuya estructura es genérica para todos los proyectos en la totalidad del código prácticamente. La estructura de la que hablamos se trata de una clase con los métodos para iniciar e inicializar el servidor y para las acciones como clientes de conectarnos, desconectarnos, suscribirnos y darnos de baja. El único punto del que debemos fijarnos es en los canales, pues es aquí donde debemos especificar de qué canales haremos uso; en nuestro caso, estos son led y vibración.

El cliente actualmente es un modelo que autogenera valores para pruebas, pues el modelo que debemos usar dependerá de los actuadores. Aún sí, nos sirve para poder observar su funcionamiento. Es preciso inicializar las opciones del cliente, pues existen varios parámetros que, según un valor u otro, nuestro cliente trabajará de distinta forma; entre estos parámetros están el Timeout, usuario y contraseña, id del cliente, etc. Como ya hemos dicho, nuestro código es un prototipo hasta que podamos avanzar en la parte hardware, por lo que hemos implementado clientes automáticos que se suscriben a los canales y cada poco segundos publican información en los canales; que en estos casos se trata de objetos de tipo led y vibración, es decir, nuestros actuadores.