Proyecto “AulaCare”

Desarrollo de Aplicaciones Distribuidas

Javier Blasco Vázquez
Enrique Piedra Venegas
Jose Antonio Valdivieso Casado

Descripción del proyecto

En la universidad, así como en el mundo laboral, es importante que las instalaciones donde se realiza la actividad productiva mantengan unas buenas condiciones para que no causen molestias o distracciones que puedan interferir en el rendimiento. El objetivo de este trabajo será crear una máquina capaz de medir ciertas condiciones del ambiente de las aulas, con la finalidad de detectar condiciones que no sean correctas para estudiar o impartir clase.

Dicha máquina estará formada por un microcontrolador (**ESP8266**), una pantalla que mostrará los valores de las condiciones y varios sensores, que medirán lo siguiente:

- **Ruido ambiente.**
- **Temperatura.**
- **Humedad.**

Para medir la temperatura y humedad, se ha optado por un sensor **DHT-22** y para el ruido un **KY-038**, que nos indica si el ruido sobrepasa un umbral determinado.

Tras hacer dichas mediciones, si los valores son correctos se mostrará por la pantalla los valores leídos. Si algo se sale de los valores recomendados, se mostrará un mensaje de advertencia indicando qué está fallando. Para relacionar la temperatura y la humedad se ha empleado el **índice de bochorno** (o “heatindex” en inglés). Así mismo, para que los alumnos y los profesores puedan ver desde cualquier parte los datos de cada aula, se realizará una aplicación web y para smartphones.

Además, los datos son almacenados con el objetivo de detectar patrones que puedan ayudar a dar con problemas que causen que las condiciones no sean las correctas.

La aplicación web cuenta con un panel de administración para que usuarios con permisos puedan ver todas las mediciones o borrarlas y hacer todos los cambios que quieran de la base de datos. Dichos cambios se realizarán mediante el uso de una API.

Las condiciones consideradas como buenas son:

- Temperatura entre 18 y 25º.
- Índice de bochorno entre 18 y 25º.
- Sonido por debajo de los valores recomendados.

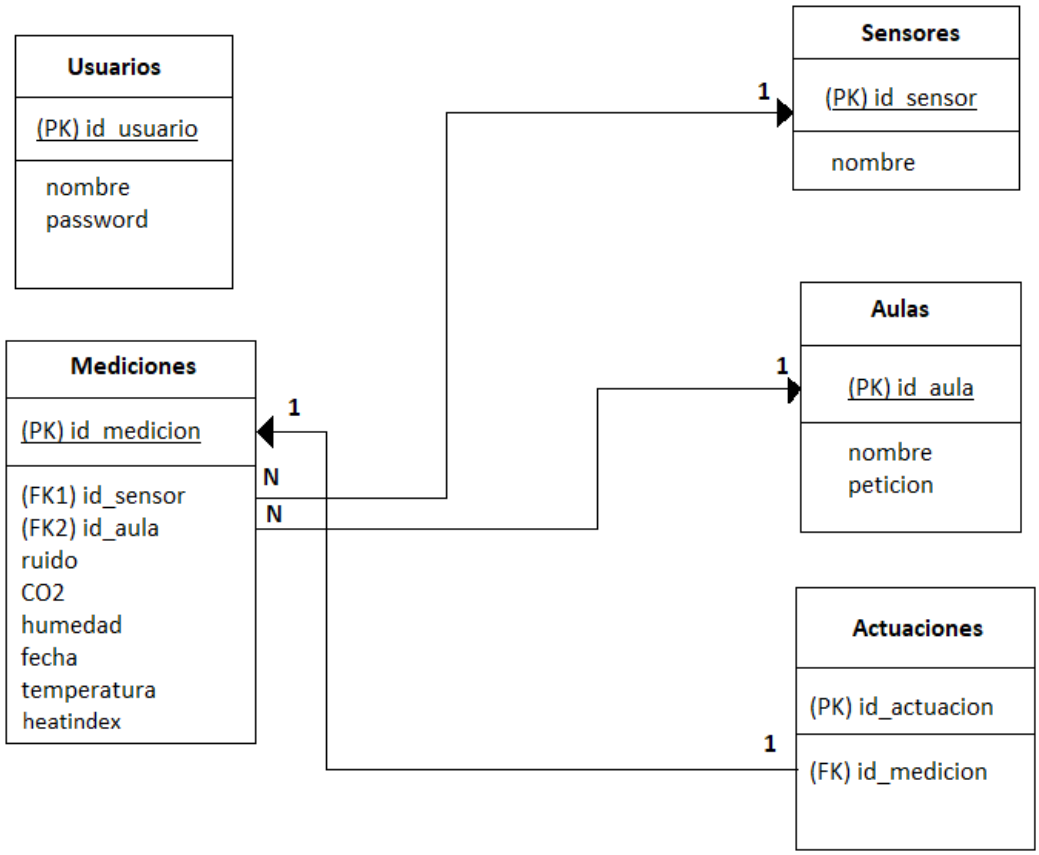
Todas las mediciones que se salen de ese rango lanzarán una actuación, es decir, un mensaje de advertencia en una pantalla LCD que lleva incorporado el dispositivo y una nueva entrada en la tabla “actuaciones” de la base de datos, de forma que podamos tener identificadas de forma clara cuáles son las malas mediciones.

Base de datos

Para nuestra base de datos se ha hecho uso de la **tecnología MySQL** apoyándonos en **MySQL Workbench** como UI para insertar datos de pruebas y ver el estado de la base de datos a lo largo del proceso de desarrollo.

Las tablas son las siguientes:

- **Usuarios**. Se usa para identificar a los administradores de la red. Les permite acceder a todo el historial de mediciones, así como a la administración del resto de tablas del proyecto. El resto de usuarios acceden a las mediciones de forma anónima y solo verán la última medición de cada aula.
- **Sensores**. Usada para identificar a los sensores.
- **Aulas.** Se emplea para identificar a cada aula.
- **Mediciones**. Usada para guardar el aula, el sensor y todas las mediciones recogidas, así como la fecha de la misma.
- **Actuaciones.** Sirve como un registro de las mediciones que han dado como resultado condiciones negativas, resultando en una actuación sobre la pantalla de información (y el aviso en la aplicación de que las condiciones no son idóneas).



Programación del Servidor

Para la programación del servidor se ha hecho uso de la solución **Vertx**, que proporciona tanto la API como la página web. Así, el proyecto se encuentra dividido en tres vértices:

- **Databaseverticle** → Se encarga de proporcionar la API y de la mayoría de conexiones y cambios que se hacen en la base de datos.
- **Webverticle** → Proporciona todos los archivos de la página web, encargándose de hacer uso de la API facilitada por el Databaseverticle con esa finalidad.
- **Main** → Solo tiene una función, que es desplegar los dos otros vértices. Así, en las configuraciones de arranque del proyecto solo tendremos que pasarle como argumento el parámetro "run Main".

Databaseverticle

A continuación se detalla todas las rutas permitidas por el vértice de la base de datos, que está escuchando siempre al puerto 8090:

- **IP:8090/usuario/:id_usuario** → *GET*. Dada una id de usuario devuelve toda la información del mismo.
- **IP:8090/usuario/nuevo** → *POST*. Recibe un objeto JSON con los campos "nombre" y "password" de tipo String y añade un usuario a la base de datos.
- **IP:8090/usuario/todos/** → *GET*. Devuelve un objeto JSON con una lista con la información de todos los usuarios.
- **IP:8090/usuario/edita-admin/:id_usuario** → *POST*. POST. Recibe un objeto JSON con los campos "id_usuario" de tipo Integer y "nombre" y "password" de tipo String y realiza los cambios sobre el usuario con el id indicado.
- **IP:8090/usuario/borra/:id_usuario** → *POST*. Recibe un objeto JSON con el campo "id_usuario" (Integer) y lo borra de la base de datos.
- **IP:8090/aulas** → *GET*. Devuelve un objeto JSON son todas las aulas.
- **IP:8090/aula/:id_aula** → *GET*. Devuelve un objeto JSON con el aula del id que recibe como parámetro.
- **IP:8090/aula/nueva** → *POST.* Recibe un objeto JSON con el campo "nombre" (String) y añade a la base de datos un aula con ese nombre.
- **IP:8090/aula/borra/:id_aula** → *POST.* Recibe un objeto JSON con el campo "id_aula" (Integer) y la borra de la base de datos.
- **IP:8090/aula/edita/:id_aula** → *POST.* Recibe un objeto JSON con los campos "id_aula"(Integer) y "peticion" (boolean) y actualiza el valor de la columna petición de dicha aula en la base de datos.
- **IP:8090/aula/edita-admin/:id_aula** → *POST. POST.* Recibe un objeto JSON con los campos "id_aula"(Integer), "nombre" (String) y "peticion" (boolean) y actualiza los valores de las columnas "nombre" y "petición" de dicha aula en la base de datos.
- **IP:8090/sensor/:id_sensor** → *GET*. Devuelve un JSON con la información del sensor cuya id recibe como parámetro.
- **IP:8090/sensor/todos/** → *GET.* Devuelve un JSON con la lista de todos los sensores.
- **IP:8090/sensor/nuevo/** → *POST.* Recibe un objeto JSON con el campo "nombre" (String) y añade a la base de datos un sensor con ese nombre.
- **IP:8090/sensor/edita-admin/:id_sensor** → *POST*. Recive un JSPON con los campos "id-sensor" (Integer) y "nombre" (String) y actualiza la columna nombre del sensor indicado en la base de datos.

- **IP:8090/sensor/borra/:id_sensor** → *POST*. Recibe un JSON con el campo “id_sensor” (Integer) y borra el sensor correspondiente de la base de datos.
- **IP:8090/medicion/:id_medicion** → *GET.* Recibe una id de una medición por GET y devuelve un JSON con todos los datos de dicha medición.
- **IP:8090/medicion/todas/:id_aula** → *GET.* Recibe un id de un aula y devuelve un objeto JSON con una lista con todas las mediciones de un aula.
- **IP:8090/medicion/borra/:id_medicion** → *POST*. Recibe un objeto JSON con el campo “id_medicion” y la borra de la base de datos.
- **IP:8090/medicion** → *PUT.* Recibe un objeto JSON con los campos “id_sensor” (Integer), “id_aula” (Integer), “ruido” (boolean), “CO2” (double), “humedad”(double), “temperatura” (double), “fecha” (long) y “heatindex” (double) y añade una nueva medición con dichos valores a la base de datos. Además, si sobrepasa los valores predeterminados, también añade una actuación con dicha medición a la base de datos.
- **IP:8090/medicion/sensor/:id_sensor** → *GET.* Recibe la id de un sensor y devuelve las mediciones de dicho sensor en una lista dentro de un objeto JSON.
- **IP:8090/medicion/actual/:id_aula** → *DELETE.* Borra la medición actual (la más reciente) de un aula.
- **IP:8090/actuacion/todas** → *GET.* Devuelve un objeto JSON con una lista con todas las actuaciones.
- **IP:8090/actuacion/:id_actuacion** → *GET.* Devuelve un objeto JSON con un la actuación correspondiente a la id que recibe como parámetro.
- **IP:8090/actuacion** → *PUT.* Recibe un objeto JSON con el campo “id_medicion” y añade una actuación con dicha medición a la base de datos.
- **IP:8090/actuacion/borra/:id_actuacion** → *POST.* Recibe un objeto JSON con el campo “id_actuacion”(Integer) y borra dicha actuación de la base de datos.

En todos los casos, si falla cualquier acceso a la base de datos se devuelve un objeto JSON con el código de estado “401” y con la causa del error. Si el resultado es el correcto en la cabecera llevará el código de estado “200”. Para permitir el cruzado de datos entre la página web y la API, al estar ejecutándose en puertos diferentes en todas las funciones que son necesarias se ha introducido la cabecera “Access-Control-Allow-Origin” desde el valor “http://IP:8080”.

Webverticle

A continuación se detalla todas las rutas permitidas por el vértice de la página web, que está escuchando siempre al puerto 8080:

- **IP:8080/web/** → Devuelve la página web principal, lo que vendría a ser el index. En ella, todos los usuarios (anónimos y registrados) pueden ver la última medición de todas las aulas acompañados por el estado mediante un intuitivo código de colores, siguiendo el símil del semáforo: color **rojo**, malas condiciones, color **amarillo**, condiciones un poco por encima o por debajo de lo recomendado, color **verde**, todas las mediciones dentro de los valores correctos.
- **IP:8080/web/info** → Devuelve una página con información sobre AulaCare.
- **IP:8080/web/login** → Devuelve la pantalla de login, que contiene dos campos donde se debe introducir el usuario y la contraseña.
- **IP:8080/web/panelAdmin** → Se llega a través del login, y recibe como parámetros por POST el usuario y la contraseña que se haya introducido. Si es correcto, devuelve la página del panel de administración, en la que se pueden borrar mediciones y actuaciones, así como añadir, editar y borrar aulas, sensores y usuarios. Si es incorrecto, devuelve una página con un mensaje de error.

- **IP:8080/css/style.css** →Devuelve el archivo de style que se necesitan en todos los apartados de la web.
- **IP:8080/css/administracion.css** →Devuelve el archivo de estilo necesario solo en el panel de administración.
- **IP:8080/css/login.css** → Devuelve el archivo de estilo necesario solo en la página de login.
- **IP:8080/js/administracion.js**→ Devuelve el archivo javascript necesario en el panel de administración.
- **IP:8080/js/index.js**→ Devuelve el archivo javascript necesario en la página de inicio.
- **IP:8080/img/fondoVerde** → Devuelve la imagen con un cuadrado verde usada en el index.
- **IP:8080/img/fondoAmarillo** → Devuelve la imagen con un cuadrado amarillo usada en el index.
- **IP:8080/img/fondoRojo** → Devuelve la imagen con un cuadrado rojo usada en el index.
- **IP:8080/img/favicon** → Devuelve la imagen con el icono usado como favicon en todas las páginas de la web.

Además de Vertx, se han empleado otros dos frameworks para el desarrollo de la web:

- **Bootstrap**. Se trata de un framework CSS que facilita el desarrollo de las webs y, sobretodo, que sean fácilmente responsivas, por lo que se pueden ver de forma correcta independientemente del navegador o dispositivo con el que estemos accediendo. Nos hemos decantado por Bootstrap por su facilidad de uso y por su soporte (originalmente es un desarrollo de Twitter).
- **jQuery**. Se trata de un framework para javascript que facilida muchos procedimientos, como las llamadas a ciertas funciones o las peticiones de tipo get y post.

Ambos se han incluído mediante CDN.

Programación del hardware

Para la programación del ESP8266 y los demás dispositivos se ha usado el entorno de desarrollo Arduino IDE, principalmente por comodidad. En el código se hacen uso de distintas librerías:

- **LiquidCrystal_I2C**, librería de Adafruit (versión 1.0.7) → Para la pantalla LCD1602, pantalla de dos líneas y dieciséis caracteres cada una.
- **DHT**, librería de Adafruit (versión 1.3.10) → Para el sensor DHT-22, de humedad y temperatura.
- **Math.h**, librería para cálculos matemáticos.
- **ESP8266WiFi.h** y **ESP8266HTTPClient.h**, usados para conectar el microcontrolador a la red WiFi y hacer las conexiones con la API.

En el código se hace uso de algunas variables globales, siendo las más importantes:

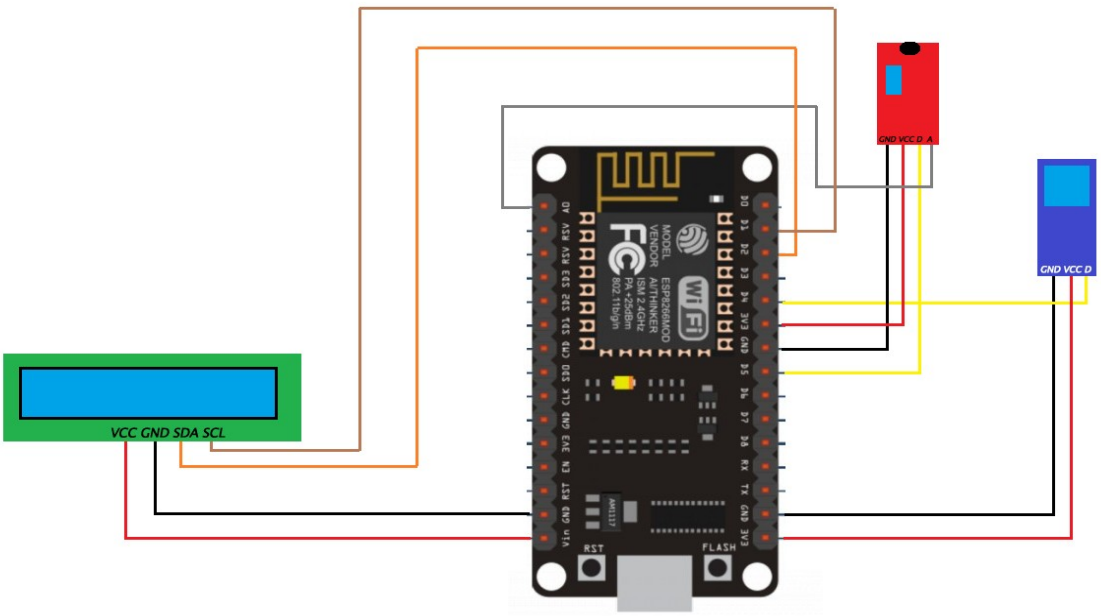
- **DHTPIN** para indicar a qué pin está conectado el sensor DHT-22.
- **DHTTYPE** para indicar que nuestro modelo es el 22.
- **dht**, que inicializa el objeto DHT.
- **lcd**, que inicializa el objeto de la pantalla.
- **id_sensor** para indicar qué sensor es en la base de datos.
- **id_aula** para indicar a qué aula corresponden las mediciones.
- **SSID** y **PASS** para las credenciales del WiFi.
- **MICA** y **MIC** para la lectura del sensor KY-038 (tanto analógico, que devuelve un valor de 0 a 1023 que indica el valor umbral, como digital, que nos dice si dicho valor umbral se ha sobrepasado).

- **client** que inicializa el WiFiClient.
- **SERVER_IP** y **SERVER_PORT** para las conexiones con la API.

Las funciones que incluye el código son:

- **sendPostRequest(double ruido, double humedad, double temperatura, double hic)** → Recibe el ruido, la humedad, la temperatura y el heatindex y envía los datos en un JSON a la API para que esta añada dichos valores como una nueva medición en la Base de Datos. Esto solo se realiza si seguimos conectados al WiFi.
- **lectura()** → Hace una medición de todos los valores, calcula el heatindex y coloca en pantalla o bien un mensaje de advertencia (si los valores están fuera de los recomendados) o bien imprime todos los valores leídos (si los valores están todos correctos). Para el sensor de ruido realiza cincuenta mediciones cada cien milisegundos, debido a su baja precisión esto es necesario para evitar falsos negativos. Si uno de los valores sale alto, entonces hay ruido ambiente alto. Hace una llamada al final a "sendPostRequest", en la que le pasa los valores leídos y calculados para que se envíen los datos al servidor.
- **checkPetition()** → Comprueba mediante una llamada a la API si hay una petición de nueva medición en la base de datos.
- **sendLeido()** → Una vez realizada una medición pedida, cambia el valor de petición a "false" en la base de datos (mediante una llamada a la API).
- **setup()** → Se encarga principalmente de inicialización de variables: inicializa el puerto serie (9600 de baudio), la variable del sensor DHT, de la pantalla lcd y del contador de tiempos. Además, realiza la conexión con la red wifi y una primera lectura.
- **loop()** → El bucle principal realiza: cada cinco minutos una lectura y la puesta de la variable de tiempos a cero, y cada treinta segundos comprueba si hay alguna petición de lectura, si es así realiza una lectura y avisa de que ya la ha hecho. Después, aumenta el contador de tiempos. Entre iteración e iteración hay un delay de 30000 milisegundos (30 segundos), que se realiza hasta diez veces (30 segundos x 10 = 300 segundos, que son cinco minutos).

Finalmente, el esquema de las conexiones sería el siguiente (siendo el dispositivo rojo el KY-038 y el azul el DHT-22):



Programación de la app

Para la aplicación de smartphones se ha hecho uso del framework **Flutter**, programado en el lenguaje **Dart**. La elección de Flutter se ha hecho principalmente por el soporte (se trata de un proyecto de Google) y porque además es multiplataforma. Las aplicaciones programadas en Flutter funcionan tanto en Android como en iOS, aunque nosotros solo hemos podido comprobar su funcionamiento en Android al no disponer de dispositivos de Apple.

El entorno de desarrollo ha sido el recomendado por Google, que es el propio **Android Studio**.

La app incluye:

- **Clase Aula** → No es la misma que la clase Aula disponible en la base de datos. Para mayor facilidad y comodidad, además del nombre y la petición las aulas llevan todos los datos de sus últimas mediciones: temperatura, humedad, índice de calor, fecha y ruido.
- **Clase MyApp**, que extiende de **StatelessWidget** → Función que construye las páginas de la aplicación.
- **Clase MyHomePage**, que extiende de **StatefulWidget** → Función con todos los datos necesarios para la página “home” de la apliación, la única de la que dispondrá. Además de los widgets de presentación (que son un encabezado, dos pestañas, texto, una “listtile”, un mensaje y botones, todo dentro de la **función build**), incluye algunas otras funciones de apoyo, como **getAulas**, que devuelve una lista con las Aulas en un objeto Future (ya que se trata de una petición asíncrona), **enviaPeticion(int id_aula)**, que recibe una id de un aula mediante un botón y hace una llamada a la API para que ponga el valor de la columna “petición” a true del aula indicada, **stringSubtitle(Aula a)**, que recibe un objeto Aula y devuelve un string con el formato de la presentación adecuado y **colorMedicion(Aula a)**, dada un aula devuelve el color que debe tener el aula.
- **Método Main** → Indica qué clase debe arrancarse al abrir la aplicación.

Así, la app tendría las mismas funciones para el usuario anónimo que la web. No así para los administradores, ya que para acceder al panel de administración sí es necesario acceder vía web.