# UNIVERSIDAD DE ALCALÁ
# GRADO EN INGENIERÍA INFORMÁTICA

# COMPUTACIÓN UBICUA

# ESTACIÓN METEOROLÓGICA INTELIGENTE

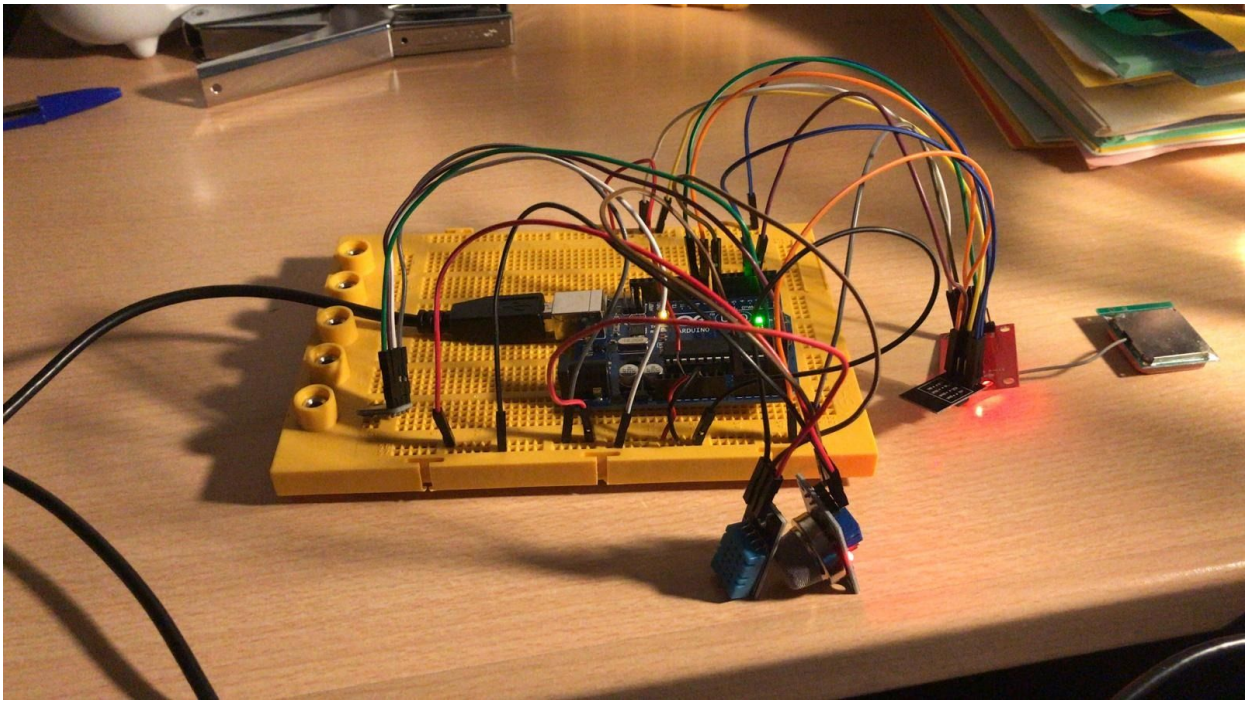
# CURSO 2019/20

**Profesores: Javier Albert Segui, Ana Castillo Martínez**

**Álvaro Pérez Álamo 06026427J**
**David Moreno López 09104275R**
**Javier García Jiménez 09099503J**
**Isabel Martínez Gómez 06027983M**

# ÍNDICE





# 1. Montaje de Arduino

### Prototipo de estación meteorológica con Arduino

Para montar el prototipo de la estación meteorológica finalmente hemos usado una placa Arduino Uno, un sensor de calidad del aire MQ-135 para recoger el nivel de CO2, un sensor de temperatura y humedad DHT11 para tanto el nivel de temperatura como de humedad, un sensor de presión BMP180 para calcular las atmósferas, un módulo Wi-Fi ESP8266 para las comunicaciones entre el Arduino y el servidor, un módulo GPS GY-NEO6MV2 para recoger la latitud y longitud del prototipo y saber su localización exacta y por último un anemómetro casero con el que recogeremos la velocidad del viento.

### Anemómetro

Se ha desarrollado un anemómetro casero a partir de vasos de papel, un tubo de Pringles, palillos, Super Glue, cinta aislante y un motor de una disquetera antigua.
Este anemómetro se conecta con la placa Arduino a través de los pines GND y el pin analógico A1. La salida del anemómetro es en voltios, por lo que hicimos una serie de pruebas en un coche, sacando el anemómetro por la ventana y obteniendo los siguientes datos:

- A 10 km/h la media de voltios obtenidos fueron 4
- A 15 km/h la media de voltios obtenidos fueron 7
- A 20 km/h la media de voltios obtenidos fueron 10
- A 30 km/h la media de voltios obtenidos fueron 15
- A 40 km/h la media de voltios obtenidos fueron 19

Por tanto, hemos considerado que la velocidad del aire en km/h es siempre aproximadamente el doble que los voltios obtenidos. Esto es:

***Velocidad = Voltios * 2***

## 2. Código Arduino

Al principio del código definimos las variables globales estableciendo los pines digitales del Arduino para aquellos módulos en los que sean necesarios como el módulo Wi-Fi, GPS o los analógicos, como el sensor de calidad del aire.

```
#define DHTPIN 7
#define DHTTYPE DHT11
#define PIN_MQ135 A0
#define co2Zero 55
char ssid[] = "Javier";
char pass[] = "holachicos";
TinyGPS gps;
AltSoftSerial serialgps(8,9);
MQ135 sensor = MQ135(PIN_MQ135);

WiFiEspClient wifi;
PubSubClient client (wifi);
SoftwareSerial soft(2, 3);
int status = WL_IDLE_STATUS;
int SMOKEA0=A0;

SFE_BMP180 bmp180;

DHT dht(DHTPIN, DHTTYPE);
```

En la función ***setup()*** inicializamos:

- El puerto Serial y el GPS a 9600 baudios
- El cliente que se conectará con el servidor
- El módulo Wi-Fi
- Los sensores de temperatura y humedad, presión y calidad del aire

```
void setup() {

   Serial.begin(9600);
   serialgps.begin(9600);
   client.setServer("192.168.1.38",1883);
   dht.begin();
   bmp180.begin();
   pinMode(SMOKEA0, INPUT);
  InitWiFi();

}
```

En la función ***loop()*** inicializamos variables y arrays que nos serán útiles en el código. Por otra parte, calculamos el nivel de CO2 en el ambiente haciendo 10 lecturas y calculando la estimación.

```
double T,P;
int co2now[10];
int co2raw=0;
int co2comp=0;
int co2ppm=0;
int zzz=0;
int grafX=0;
char temp[10];
char longi[10];
char lat[10];
char hum[10];
char pres[10];
char smok[10];
for(int x=0; x<10;x++){
  co2now[x]=analogRead(A0);
  delay(200);
}
for(int x=0; x<10;x++){
  zzz=zzz+co2now[x];
}
co2raw=zzz/10;
co2comp=co2raw-co2Zero;
co2ppm=map(co2comp,0,1023,400,5000);
```

La siguiente imagen muestra la **lectura de los datos de la temperatura, humedad y presión** recogidas por los sensores.

```
float temperature = dht.readTemperature();
float humidity = dht.readHumidity();
char buffer1[20];
 client.connect("Arduino");
 char status;
dtostrf(co2ppm,0, 0, smok);
dtostrf(humidity,0, 0, hum);
dtostrf(temperature,0, 0, temp);
status = bmp180.startPressure(3); //Inicio lectura de presión
  if (status != 0)
  {
    delay(status); //Pausa para que finalice la lectura
    status = bmp180.getPressure(P,T); //Obtenemos la presión
    if (status != 0)
    {
          dtostrf(P,0, 0, pres);
    }
  }
```

Con esta línea de código leemos el voltaje recibido por el anemómetro y lo convertimos en kilómetros por hora.

```
float v = (analogRead(1)*2);
```

Esta parte del código corresponde a la parte del **GPS** en la que sacamos la **latitud y longitud** en la que se encuentra la estación meteorológica. Por otra parte, **enviamos al servidor todos los datos recogidos** hasta el momento.

```
float latitude, longitude;
while(serialgps.available()>0)
{
  int c = serialgps.read();

  if(gps.encode(c))
  {

    gps.f_get_position(&latitude, &longitude);
    Serial.print(F("Latitud/Longitud: "));
    Serial.print(latitude,5);
    Serial.print(F(", "));
    Serial.println(longitude,5);
  }
}
dtostrf(latitude,7, 5, lat);
dtostrf(longitude,7, 5, longi);
sprintf(bufferl,"%s %s %s %s %s %s %s",temp,hum,pres,smok,lat,longi,velocidad);
Serial.print(bufferl);
     client.publish("temperatura", bufferl);
```

Esta es la última parte de la función ***loop()*** y en la que se comprueba si el módulo Wi-Fi se ha desconectado del WPA/WPA2 o si ha tenido algún error para reconectarse.

```
status = WiFi.status();
if ( status != WL_CONNECTED) {
  while ( status != WL_CONNECTED) {
    Serial.print("Attempting to connect to WPA SSID: ");
    Serial.println(ssid);
    // Connect to WPA/WPA2 network
    status = WiFi.begin(ssid, pass);
    delay(500);
  }
  Serial.println("Connected to AP");
}
```

```
void InitWiFi()
{
  // initialize serial for ESP module
  soft.begin(9600);
  // initialize ESP module
  WiFi.init(&soft);
  // check for the presence of the shield
  if (WiFi.status() == WL_NO_SHIELD) {
    Serial.println(F("WiFi shield not present"));
    // don't continue
    while (true);
  }

  Serial.println(F("Connecting to AP ..."));
  // attempt to connect to WiFi network
  while ( status != WL_CONNECTED) {
    Serial.print(F("Attempting to connect to WPA SSID: "));
    Serial.println(ssid);
    // Connect to WPA/WPA2 network
    status = WiFi.begin(ssid,pass);
    delay(500);
  }
  Serial.println(F("Connected to AP"));
}
```

## 3. Servidor

Hemos creado un servidor que comunica la estación meteorológica con la base de datos y la base de datos con la aplicación móvil a través del protocolo MQTT.
La función del servidor es recoger los datos obtenidos por la estación meteorológica y enviárselos a la base de datos para que ésta los almacene y de esta manera, enviarlos a la aplicación para que los muestre.
El servidor está desarrollado en Java, y es la parte que une todos los componentes del sistema.
El servidor se encargará por tanto de atender todas las necesidades de la aplicación, así como recoger los datos enviados por los sensores a través del Arduino y después enviarlos a la base de datos para esto, el servidor contiene varios hilos y clases para facilitar la realización de todas estas tareas de manera simultánea.
Para comenzar, comentaremos la clase **SubscriberArduino**, la cual es la conexión entre el servidor y el Arduino. Esta clase se encarga de conectar el servidor al broker de mosquitto. Para ello, crea un nuevo cliente MQTT, con una clase relacionada llamada **MqttCallBackArduino** (que después explicaremos), la cual se conectará al broker utilizando la función *subscribe* del protocolo MQTT. Este cliente por tanto, recibirá toda la información publicada del *topic* "temperatura".
Siguiendo con esto, debemos hablar ahora de la clase **MqttCallBackArduino**. Esta clase posee un método clave en el desarrollo del sistema: *messageArrived()*. Esta función se lanza cada vez

que el Arduino ha publicado datos en el *topic*. Cuando esta función es llamada, se activa la maquinaria del servidor para realizar todas las tareas que tiene encomendadas:
Primero, se crea un *ArrayList* de *String* cuya función es guardar todos los datos que envía el arduino de forma que se pueda conocer cada uno de ellos individualmente. Esto se hace ya que el Arduino está programado para enviar un solo mensaje a través de MQTT, dicho mensaje contiene todos los datos recogidos por los sensores separados por espacios. Por tanto, cuando un mensaje llega al servidor, en la función *messageArrived()* se almacenan los datos en un *ArrayList* de manera individual.
Tras esto, el servidor se conecta a la base de datos del sistema, y a través de la clase **IntroData**, se insertan los datos recogidos por los sensores en la base de datos. La clase **introData**, como se puede suponer, es una clase bastante simple cuyo cometido es recoger los datos almacenados en el ya comentado *ArrayList*, y enviarlos a la base de datos.
Tras esto, se lanzan varios hilos cuya característica en común es que son *publishers*. Esto quiere decir que cada hilo se va a encargar de realizar el método *publish()* del protocolo MQTT, cada uno con diferentes datos.
Esto está realizado así porque la aplicación móvil tiene varios apartados los cuales reciben datos diferentes, por tanto, para cada apartado de la aplicación, el servidor lanza un hilo que se encargará de publicar los datos necesarios.
El primer hilo que se lanza es el hilo **Publisher.** Este hilo se encarga de los datos relacionados con los apartados de la aplicación *"Home"* y *"Info"*. Los datos que este hilo publica son: temperatura, humedad, presión, CO2 y velocidad del viento. En este caso, este hilo atiende a dos apartados de la aplicación móvil porque ambas comparten la temperatura y la humedad en sus respectivas funciones, debido a esto, se ha decidido que este hilo que manda todos esos datos atendería a los dos, y ya que en el apartado *"Info"*, se usan la temperatura y la humedad, las otras variables se ignoran, y así se ahorra un hilo por parte del servidor.
En cuanto al siguiente hilo lanzado, vamos a hablar sobre el **PublisherStats**. Este hilo se encarga de los datos relacionados con el apartado *"Estadísticas*". En este apartado de la aplicación se necesitan las medias y las modas de la temperatura y la humedad en diferentes intervalos de tiempo. Para ello, esta clase utiliza la clase **Estadistica** y su función *calcularEstadisticas()*, la cual devuelve un *String* con todas las medias y modas necesarias.
Este *String* sigue un patrón tal que así:

*Mediat,modat,mediah,modah;....*

Esto es así para que desde la aplicación se pueda diferenciar cada uno de los datos y para enviar solo un mensaje al *topic*.
Ahora pasaremos a hablar del hilo **PublisherGraficos**. Se puede deducir de su propio nombre, que se encarga del apartado *"Gráficos"* de la aplicación. Para ello, como el anterior, hace uso de la clase **Estadistica** aunque esta vez, aprovecha la función *mediasSemana()*.
Esta función devuelve, igual que hemos comentado anteriormente, los datos necesarios en un *String*, aunque en *mediasSemana()*, los datos que se calculan son las medias de la temperatura y la humedad en cada uno de los días de la última semana, es decir, media del primer día, media del segundo… así hasta 7 días antes de la fecha actual. El *String* que devuelve esta función tiene un patrón parecido al anterior, con el mismo objetivo que se ha comentado anteriormente.

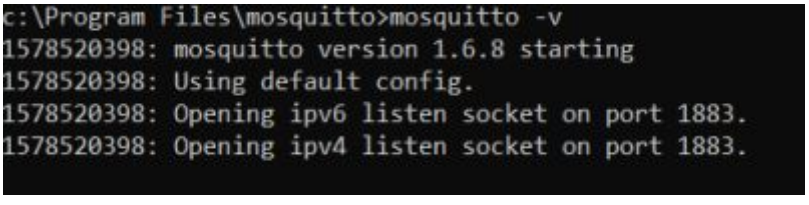
Es importante destacar que la clase **Estadística** posee una conexión con la base de datos para poder calcular todos estos datos estadísticos.
Para terminar con los hilos que manejan la información del servidor hablaremos del último hilo, este hilo se llama **PublisherUbicacion** y es el que se encarga del apartado *"Ubicación"* de la aplicación.
Cuando se reciben los datos de la estación meteorológica, este hilo hace uso de las coordenadas de latitud y longitud enviadas y recogidas por la clase **MqttCallBackArduino** para publicarlas y que la aplicación pueda acceder a ellas con el objetivo de conocer la ubicación de nuestra estación.
Todos estos hilos son los encargados de que la información que debe manejar el servidor fluya de manera correcta entre todos los elementos del sistema ya sea entre el Arduino y el servidor a través del broker MQTT, desde el servidor a la aplicación a través de los distintos publisher y el broker de MQTT, o entre el servidor y la base de datos para almacenar todo lo necesario en la misma.

# 4. Base de datos

La base de datos está creada sobre MySQL y sirve para almacenar los datos que le envía el servidor recibidos por los sensores. Estos datos son temperatura, humedad, presión, calidad del aire y velocidad del viento. Además, hemos añadido un atributo fecha que registra el día y hora en el que se ha recogido cada muestra, esto nos sirve posteriormente para hacer las gráficas y estadísticas de las últimas 24 horas, última semana, último mes y último año.

# 5. Protocolo MQTT.

Además de lo mencionado en el apartado del servidor sobre el tema de publishers y subscribers del protocolo MQTT es importante mencionar también el papel del *broker*.
El broker del protocolo MQTT es donde se realiza la comunicación entre los *publishers* y los *subscribers* de nuestro sistema.
Por tanto, el broker elegido para el sistema ha sido mosquitto. Mosquitto es un programa libre que se instala en el ordenador y que abre un broker para el protocolo MQTT por defecto en el puerto 1883. Para realizar esto, simplemente hay que realizar un cd en la consola de Windows (en nuestro caso Windows que es donde se ha desarrollado), y moverse al directorio en el que está instalado el mosquitto. Una vez allí, con el comando mosquitto -v, se inicia el broker y nuestros elementos del sistema están listos para comunicarse.
En la siguiente captura se puede observar cómo se inicia el broker de mosquitto.



# 6. Aplicación

Por último pero no por ello menos importante, pasemos a hablar de la aplicación.
Como fue mencionado en las primeras explicaciones del proyecto, es una aplicación para móviles Android, cuyo objetivo es acercar al usuario los datos recogidos por el sensor ya sea en tiempo real o a lo largo del tiempo gracias a la información recogida en la base de datos.
Gracias a la aplicación también, el usuario puede llevar un control sobre las variables meteorológicas en el lugar en el que tenga la estación colocada.
Para acceder a las diferentes funcionalidades de la estación hay que pulsar la pantalla en la primera imagen que sale, la cual contiene nuestro logo y un mensaje de "BIENVENIDO".

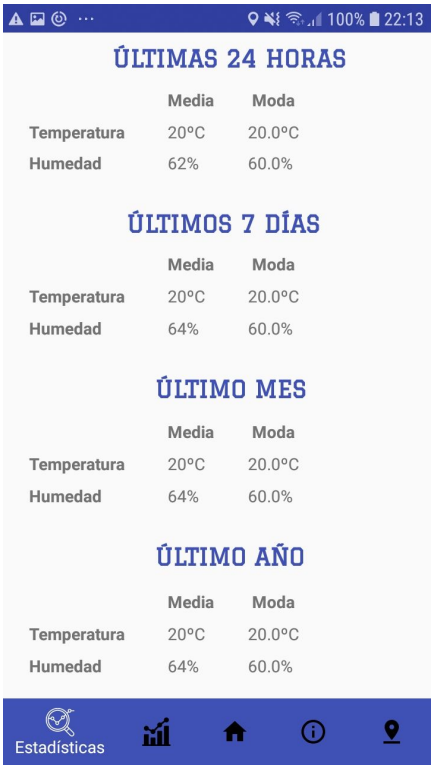

Pasemos a hablar más técnicamente de la aplicación.
Como se puede ver gráficamente, a través de una barra en la parte inferior de la pantalla, la aplicación cuenta con 5 apartados. En lo que sería el proyecto de desarrollo (en Android Studio), cada uno de estos apartados se corresponde con una "Activity":

- "Home": se corresponde con **InicioActivity**
- "Estadísticas": se corresponde con **StatActivity**
- "Gráficos": se corresponde con **GraficosActivity**
- "Info": se corresponde con **InfoActivity**
- "Ubicación": se corresponde con **UbicacionActivity**

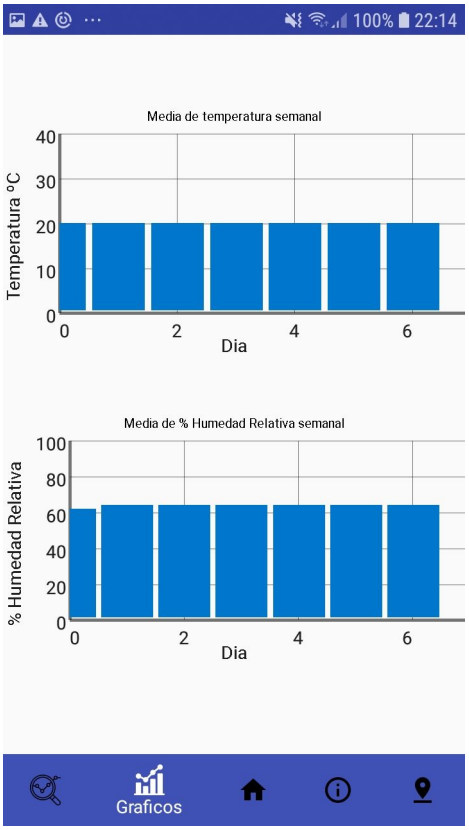
Cada una de estas actividades se encargan de ofrecer al usuario diferentes datos que le van a dar al usuario la utilidad y la información que necesita.

En el apartado "Home" se muestran los datos recogidos por los sensores a tiempo real. Estos datos son: temperatura, humedad relativa, CO2 en el aire y velocidad del viento. También se muestra la sensación térmica del momento actual calculada según los valores de la velocidad del viento y la temperatura.


En el apartado “Estadísticas” se recogen las medias y las modas de la temperatura y de la humedad en diferentes intervalos de tiempo. Estos intervalos de tiempo son: en las últimas 24 horas, en la última semana, en el último mes y en el último año.
Por tanto, en este apartado el usuario puede ver cómo han ido variando las temperaturas y las humedades del lugar donde esté colocada la estación a lo largo del tiempo.

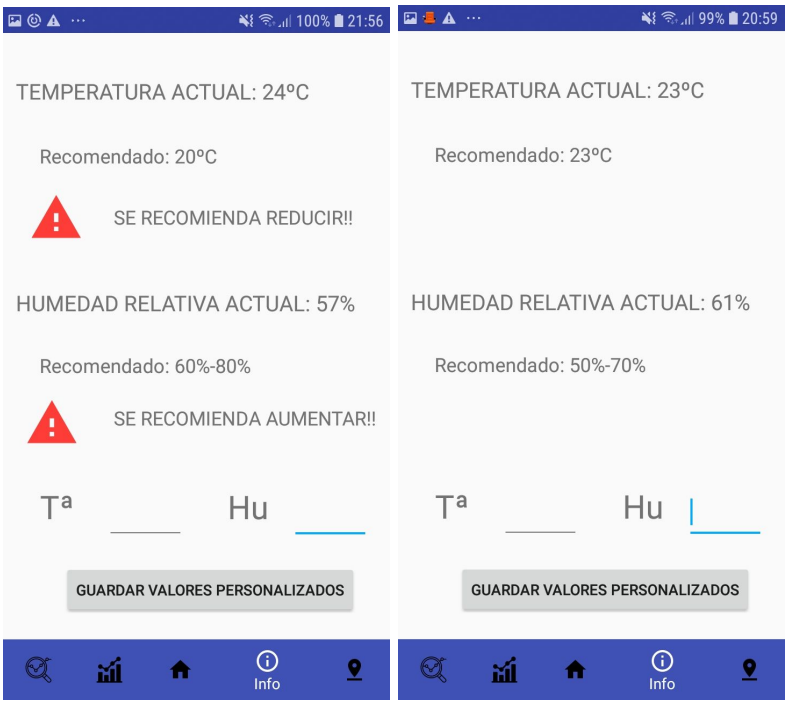


En el apartado “Gráficos” se le proporciona al usuario una visión gráfica acerca de las medias de la temperatura y la humedad en cada uno de los días de la última semana. Esto le permite al usuario controlar si las temperaturas son constantes en el lugar donde esté la estación, o en caso de que no lo sean, será fácil identificar si las temperaturas/humedades están siendo muy inconstantes en los últimos días.




Continuemos con el apartado “Info”. En este apartado el usuario puede observar una comparación entre una temperatura ideal y la temperatura actual, y entre una humedad ideal y la actual.
Esta temperatura y humedad ideales están configuradas por defecto a, en el caso de la temperatura, 21 ºC y en el caso de la humedad, en un rango entre 40 y 60 por ciento, pero puedes introducir la temperatura ideal que tu quieras por pantalla y lo mismo con la humedad, salvo que al introducir la humedad se cogerá un rango de un 10 por ciento mayor y menor que el tanto por ciento introducido. Esto es así ya que si se aumenta en un interior la temperatura o la humedad, estaría gastando electricidad sin necesitarlo. De forma contraria, si estos valores disminuyen, podría hacer que el usuario no esté cómodo teniendo frío.
En este apartado por tanto, se observan dichas variables ideales o recomendadas y las que existen en el momento actual. En caso de que la temperatura actual sea 2 grados inferior o superior a la recomendada, el sistema mostrará una visualización gráfica con un triángulo rojo que indica si se recomienda bajar o subir la temperatura, además, para notificar al usuario sobre esto, el móvil vibrará cada vez que el sistema recoja datos y estos no sean los recomendados por el usuario . En cuanto a la humedad, si el valor de la humedad se sale del rango, se muestra el mismo aviso para subir o bajar la misma.

Además, se ha añadido la funcionalidad que permite que el usuario personalice los valores recomendados de temperatura y de humedad.
Gracias a esta opción, el usuario puede poner como recomendados los valores que desee, y cuando vuelva a abrir la aplicación, estos valores seguirán guardados.




Por último llegamos al apartado de la "Ubicación". Este apartado muestra la ubicación de nuestra estación a tiempo real. Cada vez que el sensor manda datos al servidor, se recoge la ubicación (a través de la latitud y la longitud) y se actualiza en el mapa.
El mapa está implementado con Google Maps, e indica la posición de la estación gracias a un marcador. En la siguiente captura se puede observar el mapa resultante al conectar la estación en el parking de la Escuela Politécnica Superior de Alcalá de Henares.

Ahora vamos a pasar a comentar, como cada una de estas actividades puede recibir la información del servidor.
Como se ha comentado anteriormente, el protocolo utilizado para la comunicación es el MQTT, por tanto, para recibir la información es necesario que cada una de las actividades llame a un método *subscribe()* de dicho protocolo para poder conocer la información publicada por el servidor en los diferentes *topics*.
Por tanto, cada una de las actividades está asociada con una clase diferente de *Subscriber*, y además, cada una de las clases *Subscriber* hace uso de una clase diferente de **MqttCallBack.**
Por tanto, las clases *Subscriber* se encargan de subscribirse a un *topic* y generar la clase **MqttCallBack** asociada a dicho *subscriber*, la cual se encarga de recoger el mensaje que se publica en su *topic* y llamar a un método de la actividad con la que está relacionada para actualizar la información existente en pantalla.

Como conclusión, cada una de las clases de las actividades, lanzan una clase *subscriber* que se suscribe a un *topic* (cada una al que le interese) que a su vez lanzan una clase **MqttCallBack** que se encarga de recibir los mensajes y enviarlos a la actividad correspondiente.
Algo que hay que remarcar para el correcto uso de la aplicación es que los datos de los distintos apartados se pierden si se pulsa el botón de atrás de los dispositivos android, por tanto, para mantener los datos de los apartados y no "reiniciar ese apartado", lo correcto es no pulsar el botón de atrás.