Memoria Proyecto DAD
Sistema de Riego Automático

Por Pablo Cano Navajas y Alberto Jiménez Medel
Redactada por Pablo Cano Navajas

Índice

1. *Introducción*

Nuestro proyecto trata de la implementación de un pequeño sistema de riego automático basado en la medición de valores de humedad y temperatura de las distintas plantas que contenga nuestro huerto o plantación.

En el presente documento se abordarán cuales son los objetivos que hemos tratado de cubrir, como hemos implementado el sistema final y cuales han sido los fallos que hemos tenido a lo largo del proceso.

Así mismo, este proyecto es una mera aproximación a lo que sería un sistema funcional en la vida real. Ya que por ejemplo nuestras placas reciben la energía a través del mismo cable USB-C que utilizamos para subir el código, en un entorno real habría que idear un sistema de alimentación para nuestro controlador, con unas baterías apropiadas y un estudio del consumo de energía de estas; o bien, un sistema de alimentación dependiente de la red eléctrica, para el que habría que incluir transformadores de tensión para nuestras placas, etc. A su vez, la elección de la lógica de nuestro actuador también habría que afinarla dependiendo del entorno donde fuéramos a instalar nuestro sistema de riego, no es lo mismo el verano de Asturias que el de Sevilla (por desgracia).

Por no hablar de temas de eficiencia con el servicio REST, pero todo ello lo iremos tratando a lo largo del documento.

2. *Objetivos*

Por tanto, los objetivos de nuestro proyecto son sencillos:

- Saber cuando hay que regar una planta
- Saber como podemos regar la planta
- Saber como podemos escalar nuestro sistema

Por norma general podemos saber cuando hay que regar una planta por dos cosas, o bien la humedad del suelo de la planta, o bien la temperatura alrededor de la planta.

Si una planta tiene la tierra poco húmeda o prácticamente seca necesita ser regada, ya que de lo contrario esta se deshidratará conduciendo a su muerte. A su vez, si la tierra se encuentra muy seca reduce su capacidad de absorber agua, esto es algo que nos interesa evitar ya que podría fastidiar nuestro sistema de riego; mientras la tierra está muy seca todo el agua que le demos tiende a quedarse “encima” de la tierra pudiendo hacer que desborde del macetero. Esto también es interesante de cara a un huerto que no use macetas, ya que si bien no desbordaría supone un gasto extra de agua.

Si el entorno de la planta es demasiado cálido, esto aumenta la tasa de evaporación del agua, cosa que es mala por lo anteriormente explicado. Si bien la temperatura nos puede servir de indicador, es algo que habría que medir bien en un ámbito de aplicación práctica, ya que no por regar mucho la temperatura de alrededor va a bajar. Si no lo medimos bien, puede resultar que nuestro sistema de riego automático se acabe convirtiendo en un sistema de despilfarro automático de agua.

Ya sabemos cuando hay que regar la planta, ahora nos preguntamos ¿Cómo lo hacemos? Pues bien, para este cometido necesitaremos un sensor que nos mida la temperatura, un sensor que nos mida la humedad y un actuador que nos permita encender y apagar el riego.

¿Como escalamos el sistema? Necesitamos una estructura que sea fácilmente replicable para poder utilizarla en cada una de las plantas o macetas que tengamos.

Vistos los objetivos, pasemos con la implementación.

3. *Implementación*

3.1. *El Hardware*

Lo primero a ver es que “cacharros” vamos a poner en cada una de nuestras plantas. Necesitamos un microcontrolador que no sea muy aparatoso y unos sensores/actuadores de tamaño reducido. Nosotros hemos optado por:

- Una ESP32 NodeMCU como microcontrolador, con un área de 5x2.5 cm^2 cumple con nuestros requerimientos de espacio. Con un total de 24 pines nos es más que suficiente para los requerimientos del proyecto. Además esta placa cuenta con conectividad wifi, lo cual es esencial (veremos más adelante por qué).
- Un DHT11 como sensor, antes dijimos que íbamos a necesitar un sensor que nos midiera la temperatura y otro que nos midiera la humedad, pues bien, este es un todo en uno. También cuenta con un tamaño reducido (2.5x1.5 cm^2).
- Un Relé como actuador, quizá es el componente que más pesa y que más ocupa de los tres (aproximadamente 7 cm^2). Este irá conectado a nuestro riego y se encargará de encenderlo o apagarlo

Sumado a esto vamos a necesitar un cable USB-C por placa para suministrarle energía y poder programarla, además de los cables para interconectar los componentes.

3.2. *El Software*

Vamos a utilizar una estructura cliente-servidor, con un servidor REST que irá conectado a una base de datos SQL y MQTT para manejar los mensajes entre nuestro servidor y nuestra placa.

3.2.1. *Tipos*

Crearemos dos tipos en Java, uno para el sensor y otro para el actuador. Cada uno de ellos tendrá un id de dispositivo (idSensor/idActuador) y un id de grupo (idGroup). El sensor tendrá por un lado los atributos de humedad y temperatura de tipo Double. El actuador por otro lado solo tendrá un atributo de tipo Boolean “activo”, para saber si está encendido o no.

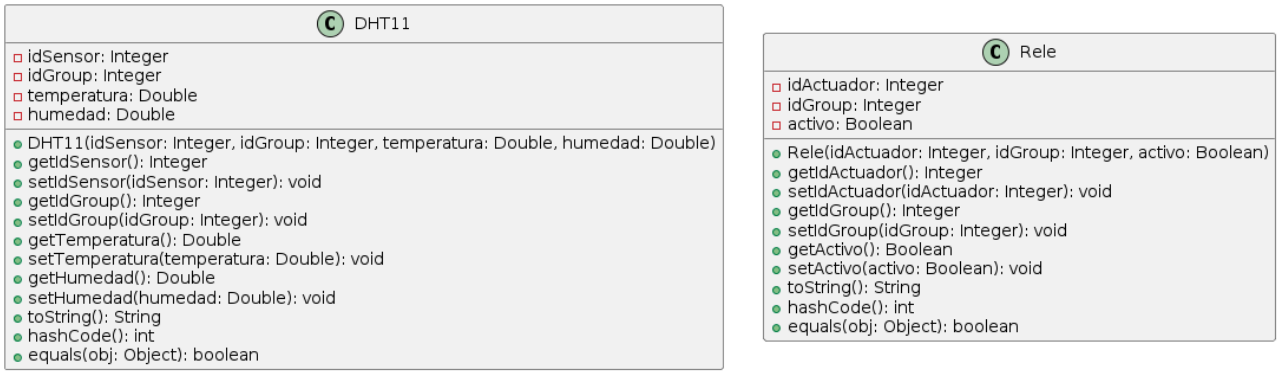


Figura 1: Diagrama UML de las Clases implementadas

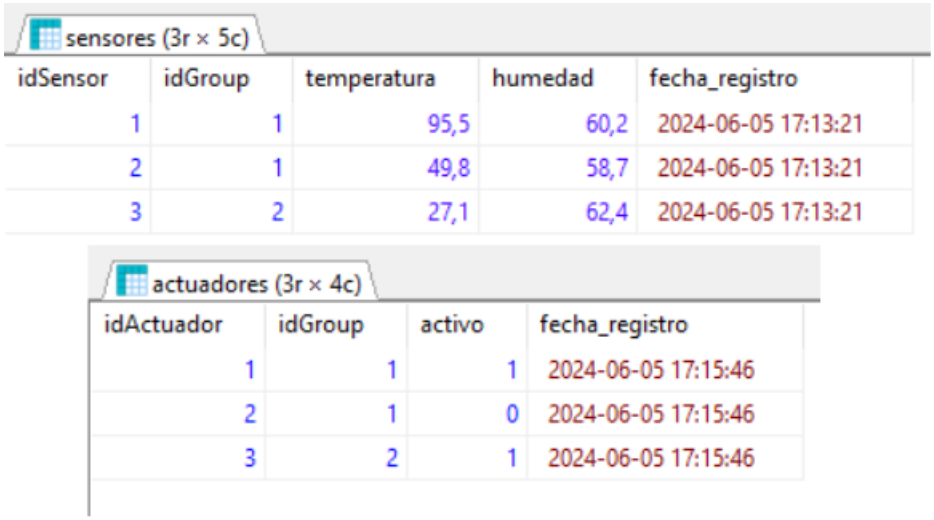
Una pregunta que se nos puede plantear es ¿El idGroup para que sirve? Pues bien, cada uno de nuestros tríos de placa, sensor y actuador que vamos a tener por cada planta forman un grupo. Para la planta 1 grupo 1, para la planta 2 grupo 2, etc. Este atributo es muy útil ya que al final necesitamos saber que planta es la que necesita ser regada. Si no pudiéramos identificar el conjunto de componentes que rodean a esta sería imposible llevar acabo este cometido.

3.2.2. *Base de datos*

Hemos optado por utilizar MariaDB con HeidiSQL como base de datos, ya que la habíamos usado anteriormente en otras asignaturas. Vamos a contar únicamente con dos tablas, una para sensores y otra para actuadores.

Para la tabla de sensores sus atributos serán exactamente los mismos a los que poseía nuestra clase DHT11. Unicamente se le suma el atributo “fecha_registro” que se trata del timestamp que deja cada entrada al ser registrada, este atributo lo rellena automáticamente MariaDB cuando se añade una entrada, con lo cual no hay que preocuparse de rellenarlo nosotros.

Para la tabla de actuadores pasa exactamente lo mismo.



sensores (3r × 5c)

idSensor	idGroup	temperatura	humedad	fecha_registro
1	1	95,5	60,2	2024-06-05 17:13:21
2	1	49,8	58,7	2024-06-05 17:13:21
3	2	27,1	62,4	2024-06-05 17:13:21

actuadores (3r × 4c)

idActuador	idGroup	activo	fecha_registro
1	1	1	2024-06-05 17:15:46
2	1	0	2024-06-05 17:15:46
3	2	1	2024-06-05 17:15:46

Figura 2: Nuestras tablas con algunos valores de ejemplo

Estas tablas serán rellenadas por nuestras placas en el momento de su puesta en marcha como veremos más adelante.

Nuestras tablas no contienen claves primarias para permitir los valores repetidos.

3.2.3. Endpoints

Nuestras ESP32 no se van a conectar directamente a la base de datos para publicar los valores de los sensores y actuadores, en cambio lo van a hacer a un servidor REST que gestione esas peticiones por HTTP.

Para ello hemos implementado los siguientes Endpoints:

```
router.route("/api/sensores*").handler(BodyHandler.create());
router.get("/api/sensores").handler(this::getAllSensores);
router.get("/api/sensores/:sensorid").handler(this::getLastIdSensor);
router.get("/api/sensores/grupo/:groupid").handler(this::getLastIdGroupSensores);
router.post("/api/sensores").handler(this::postOneSensor);
router.delete("/api/sensores/:sensorid").handler(this::deleteOneSensor);
router.put("/api/sensores").handler(this::putOneSensor);

router.route("/api/actuadores*").handler(BodyHandler.create());
router.get("/api/actuadores").handler(this::getAllActuadores);
router.get("/api/actuadores/:actuadorid").handler(this::getLastIdActuador);
router.get("/api/actuadores/grupo/:groupid").handler(this::getLastIdGroupActuadores);
router.post("/api/actuadores").handler(this::postOneActuador);
router.delete("/api/actuadores/:actuadorid").handler(this::deleteOneActuador);
router.put("/api/actuadores").handler(this::putOneActuador);
```

Figura 3: Endpoints implementados en nuestro servidor REST

Como puede apreciarse tenemos los mismos endpoints tanto para sensores como para actuadores, por lo que de aquí en adelante cuando se quiera referenciar a “actuadorid” o a “sensorid” lo haré como “dispositivoid”, pero es simplemente para facilitarme la explicación.

- GET
  - getAllSensores/getAllActuadores: Nos devuelve todos los datos que estén contenidos en la tabla correspondiente.
  - getLastIdSensor/getLastIdActuador: Nos devuelve el último elemento que ha sido introducido en la tabla correspondiente dado un “dispositivoid”. Para esto es para lo que sirve el atributo “fecha_registro”.
  - getLastIdGroupSensores/getLastIdGroupActuadores: Hace exactamente lo mismo que el endpoint anterior pero esta vez filtra por “groupid”.
- POST
  - postOneSensor/postOneActuador: Añade un dispositivo a su tabla correspondiente.
- PUT
  - putOneSensor/putOneActuador: Actualiza un dispositivo que ya esté registrado en su tabla correspondiente.
- DELETE
  - deleteOneSensor/deleteOneActuador: Borra un dispositivo de la tabla dado su “dispositivoid”

Los endpoints más útiles para nuestro proyecto serán por un lado, los de obtener el último valor dado el groupId, el post, y el método put. Todo esto pensando en nuestros actuadores, los subiremos una vez con el post y a partir de ese momento simplemente iremos actualizando sus valores para no inundar la base de datos, ya que no están cambiando constantemente de valor como si lo hacen nuestros dht11.

3.2.4.MQTT

En este apartado hablaremos del cliente mqtt que nuestro servidor estará ejecutando, ya que la función del broker la hemos suplido con [mosquitto](). .

Los datos de los dht11 se irán publicando en el topic *datos/sensor*, así que lo primero que hará nuestro cliente será suscribirse a ese topic.

Nuestra lógica para el actuador será la siguiente: Si el valor de humedad recibido del sensor es inferior al 50% o la temperatura es superior a 24 grados, entonces mandaremos la orden de que se active nuestro relé. Esto lo haremos publicando un mensaje en el topic *instrucciones/rele.*

Como dijimos en la introducción, estos valores son elegidos a modo de ejemplo, ya que habría que hacer un estudio más exhaustivo de cuales serían los valores más óptimos para nuestro cultivo en concreto (Como pongas 24º en verano no vas a querer ver la factura del agua).

La idea y lo que acabó siendo

```
mqttClient.publishHandler(handler -> {
    System.out.println("Mensaje recibido:");
    System.out.println("    Topic: " + handler.topicName().toString());
    System.out.println("    Id del mensaje: " + handler.messageId());
    System.out.println("    Contenido: " + handler.payload().toString());

    // Parsear el JSON recibido
    String payload = handler.payload().toString();
    DHT11 sensorData = gson.fromJson(payload, DHT11.class);


    if (sensorData.getHumedad() < 50 || sensorData.getTemperatura() > 24 ) {
        Rele estadoActuador = getReleIdGroup(miGestorRest, sensorData.getIdGroup());
        if (!estadoActuador.getActivo()) { //Si ya está encendido, para que lo voy a estar encendiendo todo el rato
            String response = gson.toJson(new Rele(estadoActuador.getIdActuador(), estadoActuador.getIdGroup(), true));
            mqttClient.publish("instrucciones/rele", Buffer.buffer(response), MqttQoS.AT_LEAST_ONCE, false,
                    false);
        }
            String response = gson.toJson(new Rele(sensorData.getIdGroup(), sensorData.getIdGroup(), true));
            mqttClient.publish("instrucciones/rele", Buffer.buffer(response), MqttQoS.AT_LEAST_ONCE, false,
                    false);
    } else {
        Rele estadoActuador = getReleIdGroup(miGestorRest, sensorData.getIdGroup());
        if (estadoActuador.getActivo()) { //Si no esta encendido para que voy a estar apagandolo todo el rato
            String response = gson.toJson(new Rele(estadoActuador.getIdActuador(), estadoActuador.getIdGroup(), true));
            mqttClient.publish("instrucciones/rele", Buffer.buffer(response), MqttQoS.AT_LEAST_ONCE, false,
                    false);
        }
        String response = gson.toJson(new Rele(sensorData.getIdGroup(), sensorData.getIdGroup(), false));
        mqttClient.publish("instrucciones/rele", Buffer.buffer(response), MqttQoS.AT_LEAST_ONCE, false,
                false);
    }
```

Figura 4: Código correspondiente al recibimiento del mensaje y la lógica del actuador

Nuestra idea a la hora de la lógica del actuador era la siguiente: el cambio en las condiciones físicas no va a ser inmediato, luego el sensor seguirá percibiendo valores que disparan la lógica mientras ya se ha activado el relé ¿Para qué voy a estar diciendo que este se encienda si ya está encendido?

Para ello creamos la clase **ApiService**, que simplemente implementa las llamadas a nuestros endpoints. Así podemos hacer un get del último elemento filtrado por el idGroup, esto es muy bueno ya que a nosotros nos llega los datos del sensor pero entre esos datos *no está el idActuador* del relé asociado a ese sensor. Cómo el idGroup es el mismo para ambos, caso cerrado.

La idea por un lado era no saturar de actualizaciones sin motivo a la base de datos y a nuestra API REST, aunque las peticiones GET no nos las quitaba nadie.

El único problema es que a la hora de hacer la demostración del funcionamiento de nuestro sistema teníamos que publicar instrucciones artificiales en *instrucciones/rele* ya que no podemos cambiar las condiciones físicas del recinto donde realizamos la prueba, al menos no fácilmente. Dado que las placas son bastante más rápidas mandado mensajes que nosotros, al final el último valor registrado en la base de datos nunca correspondía con el ultimo valor enviado mediante mqtt, luego esto daba fallo.

Quizá el costo de estar actualizando constantemente la tabla de los actuadores con valores repetidos es menor que los fallos producidos por cruce de valores desincronizados entre mqtt y la base de datos, habría que estudiarlo.

Sin embargo, la clase **ApiService** nos sigue siendo útil para conocer el idActuador.

3.3. El Firmware

Nuestras ESP32 tienen implementados los métodos para hacer las consultas a la API REST y para gestionar los mensajes MQTT.

Si bien nuestro servidor era el que se suscribía al topic *datos/sensor* y publicaba mensajes en *instrucciones/rele* aquí es justo al contrario.

```
12  const int DEVICE_ID = 1;
13  const int ID_SENSOR = 1;
14  const int ID_ACTUADOR = 1;
15  const int ID_GROUP = 1;
16
17  const int relayPin = 15;
18  const int dht11Pin = 13;
```

Figura 5: Parte del código dedicada a la configuración de la placa

Al principio de nuestro código tendremos las siguientes constantes, tendremos que modificarlas de acuerdo a la placa que estemos configurando. Será esta la que publicará los datos en la base de datos, por ello es necesario definirlo. *DEVICE_ID* sirve unicamente para darle nombre a la placa en mqtt, que se definirá como *"ArduinoClient_"+DEVICE_ID*.

Los pines se explican por si solos, deberemos indicar en que pin digital están conectados ambos dispositivos.

3.3.1.Lógica de actuador desde el lado del Firmware

```
void OnMqttReceived(char *topic, byte *payload, unsigned int length)
{
  Serial.print("Received on ");
  Serial.print(topic);
  Serial.print(": ");
  String content = "";
  for (size_t i = 0; i < length; i++) {
    content.concat((char)payload[i]);
  }
  Serial.print(content);
  Serial.println();

  // Parsear el mensaje recibido
  DynamicJsonDocument doc(2048);
  deserializeJson(doc, content);
  int idActuador = doc["idActuador"];
  int idGroup = doc["idGroup"];
  bool activo = doc["activo"];

  if (activo) {
    digitalWrite(relayPin, HIGH);
    putOneActuador(serializeActuatorStatusBody(idActuador,idGroup,activo)); //Actualizamos el valor en la base de datos con un put, para no inundar de registros
  } else {
    digitalWrite(relayPin, LOW);
    putOneActuador(serializeActuatorStatusBody(idActuador,idGroup,activo));
  }
}
```

Figura 6: Código para la parte de la lógica del actuador

Como comentamos en el punto 3.2.4, la clase **ApiService** nos seguía haciendo falta para saber el idActuador, esto es porque nuestro endpoint de tipo PUT recibe el objeto JSON completo con todos sus parámetros. Quizá hubiera sido más sencillo complementar la parte del idActuador con el propio parámetro definido al principio del código, pero también es cierto que de esta manera el mensaje MQTT queda mucho más claro si lo revisamos desde MQTT-Explorer.

Por lo demás, si activo = true encendemos el relé y si no lo apagamos seguido de su actualización en la BD. Aquí también la idea que se comentaba en 3.2.4, si llegaban menos mensajes a *instrucciones/rele* menos actualizaciones por parte de la placa.

3.3.2.Inicialización del Actuador y el Sensor

```
  pinMode(relayPin,OUTPUT); //Salida digital del relé
  pinMode(dht11Pin,INPUT);  //Entrada digital del dht11
  dht.begin();
  postOneSensor(serializeSensorValueBody(ID_SENSOR,ID_GROUP,55.0,25.0));
  postOneActuador(serializeActuatorStatusBody(ID_ACTUADOR,ID_GROUP,false));
}
```

Figura 7: Extracto del código de setup()

Decíamos que iban a ser nuestras placas las que rellenaran desde el primer momento las entradas en las tablas de la BD, pues bien es aquí donde se hace.

Para el sensor se eligen esos valores con el único objetivo de que sean unos que no disparen la lógica del actuador dado que en cuanto empiece a funcionar la ESP32 va a añadir los nuevos valores y los podemos filtrar fácilmente gracias a nuestros Endpoints.

Además, en esta sección del código también se definen los pines como entrada o como salida.

3.3.3.Funcionamiento del loop()

```
HandleMqtt();

double humedad = dht.readHumidity();
double temperatura = dht.readTemperature();

if (isnan(humedad) || isnan(temperatura)) {
  Serial.println(F("Failed to read from DHT sensor!"));
  return;
}

// Publicar datos del sensor
DynamicJsonDocument doc(2048);
doc["idSensor"] = ID_SENSOR;
doc["idGroup"] = ID_GROUP;
doc["temperatura"] = temperatura;
doc["humedad"] = humedad;

String output;
serializeJson(doc, output);
client.publish("datos/sensor", output.c_str());
postOneSensor(output); //Subo también los datos del sensor a la base de datos.

delay(3000);
```

Figura 8: Extracto del código contenido en loop()

Para esta parte simplemente comentar que iremos mandando los mensajes y publicando en la BD cada 3 segundos. Hemos observado que si se pone un tiempo demasiado elevado en el delay la comunicación con MQTT empieza a fallar, habría que determinar cual sería el valor más optimo de espera para que no sature de peticiones y a su vez no comprometa la comunicación.

4. Montaje

En esta sección simplemente me gustaría aclarar de forma breve por qué hemos montado los sensores y actuadores de la manera que lo hemos hecho para la prueba de concepto.

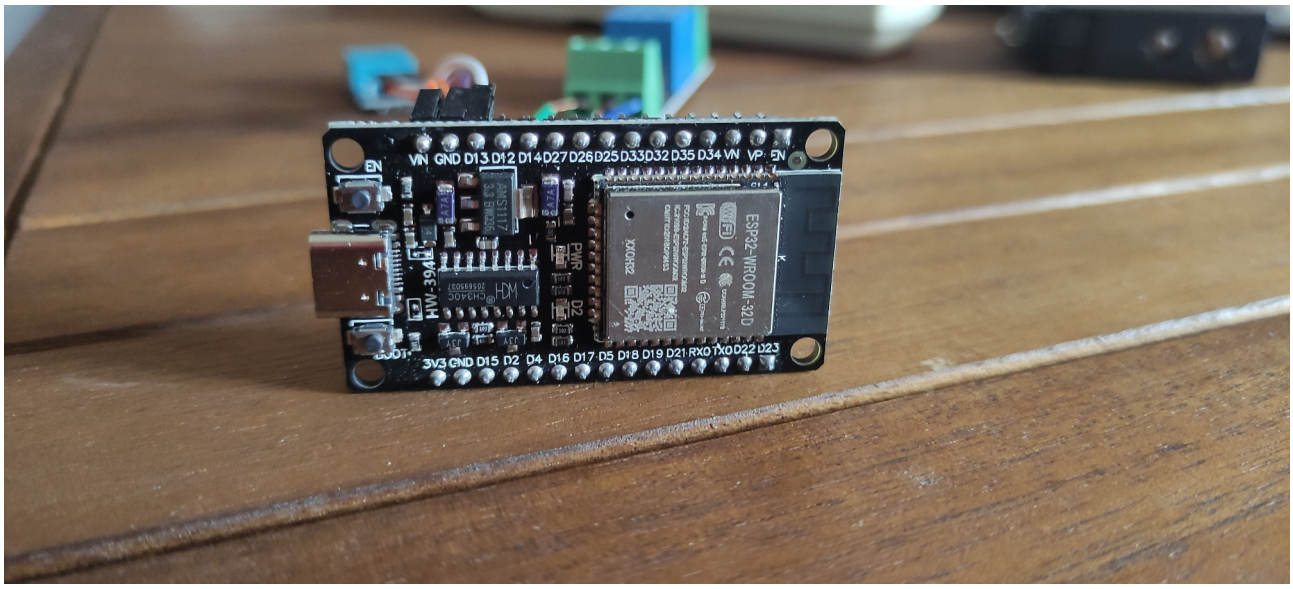


Figura 9: Vista de nuestra ESP32 y sus pines

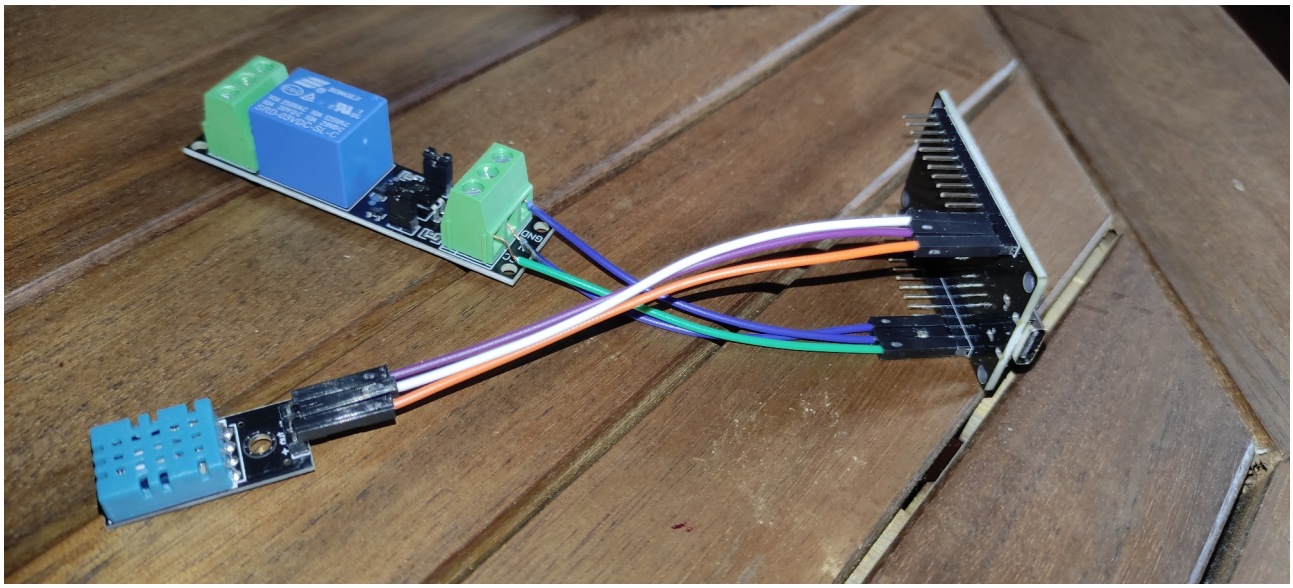


Figura 10: Montaje del Relé y el DHT11

Resulta que la ESP32 tiene unicamente dos entradas de tierra (GND) como podemos ver en la Figura 9, tanto el DHT11 como el Relé necesitan una entrada a tierra luego hemos puesto el Relé en la zona baja y el DHT11 en la zona alta. Lo hemos hecho así ya que nuestro relé es de 3.3v, y únicamente en la toma de la zona baja es donde se especifica que sean exactamente 3.3v.

Por otro lado en la Figura 10 se puede ver como queda el montaje, hemos utilizado cables EasyConnect para facilitarnos el trabajo, aunque para el relé hemos tenido que pelar el extremo del cable que se une a este.

5. Conclusión y Troubleshooting

Ha sido un proyecto interesante a la par que en algunas ocasiones desesperante, ya que sin lugar a dudas donde más tiempo se pierde es en arreglar errores de código, dependencias de librerías y conexión al wifi. Aquí algunas de las cosas que han hecho replantearme por que he elegido esta carrera:

- Para el entregable 2 y 3 las dependencias de las librerías, ya que dependiendo de como crearas el proyecto la sintaxis de las consultas cambiaba sin razón aparente. Lo solucioné usando uno de los proyectos de ejemplo como base.
- Mandar las peticiones a la api REST, ya que empecé utilizando una ruta que no era por confusión a como se hacían las consultas en el primer entregable. Estuve fácilmente 3 o 4 días sin saber que pasaba, y cuando te das cuenta de la tremenda tontería que era lo que estaba fallando no sabes si quieres reír o llorar.
- Para el entregable 3 (de nuevo) no era capaz de insertar valores en la base de datos, esto era porque estaba intentando leer el contenido del routingContext antes de establecer la conexión con la base de datos. Pues bien, otra semana para darme cuenta de esto.
- Para la parte del firmware la placa no se conectaba al internet del móvil, esto es un problema que también le pasó a otros compañeros. Resulta que tenemos el mismo modelo de móvil y por alguna razón que desconozco la zona wifi que este crea es incompatible con la ESP32. Pero esto no se queda aquí, en mi casa, cree una segunda red wifi en mi router para ir haciendo las pruebas ya que no quería dejar puesta la clave de mi red wifi principal en el código del firmware; pues bien, exactamente el mismo problema. Finalmente utilicé mi red wifi principal y solucionado. Este fallo por suerte lo solucioné en una tarde pero te puede dar verdaderos dolores de cabeza si tienes la mala suerte de que la placa tampoco decide conectarse a tu wifi.

Dejando los errores a un lado, cuando vi que el relé funcionaba ciertamente me llené de ilusión y fui corriendo a la ferretería de mi barrio para comprar una pila, unos cables y una bombilla ya que quería ver como funcionaba y no conformarme solo con el pilotito rojo que a este se le enciende.

Disfruté bastante de la última parte del proyecto hasta tal punto que la redacción de esta memoria ha supuesto más un hobby que un trabajo pesado, ya que ciertamente tenía ganas de documentar de la mejor manera posible todo el proceso. Además me ha despertado esas ganas de seguir trasteando con proyectos del estilo por mi cuenta.

Esto todo, espero haberlo plasmado de la mejor manera posible.

Gracias.