

Diseño de Aplicaciones Distribuidas

“Control de Emisiones Vehiculares”

MEMORIA

Hecho por:

RAFEL ANTONIO CUADRADO SOLA

ISAAC LUQUE MESA

PABLO PARRILLA CHÉRCOLES

FRANCISCO JOSÉ RODRÍGUEZ LUNA

INTRODUCCIÓN

Los vehículos son una de las principales fuentes de contaminación en las ciudades, cada vez que ponemos en funcionamiento nuestro coche este emite gases como dióxido de carbono (CO_2), monóxido de carbono (CO), partículas en suspensión entre otros que contribuyen a la contaminación del aire.

La preocupación por el medio ambiente cada vez va creciendo más en nuestra sociedad, pero la mayoría de los conductores desconoce cuánto contamina su vehículo. A pesar de que el fabricante proporciona estos datos sobre las emisiones en la ficha técnica, las cifras se basan en pruebas estandarizadas que no reflejan los valores reales a la hora de usar nuestro vehículo personal.

Solución propuesta

Para resolver esto hemos desarrollado un dispositivo con el que podrás monitorizar en tiempo real las emisiones de tu propio coche. Esto servirá tanto como para controlar los gases contaminantes en caso de tener que hacer una reparación, evitando así tener que hacer varios intentos para pasar la ITV, por ejemplo, como para concienciar a las personas sobre cuánto contaminan en comparación al resto o al transporte público.

Componentes

ESP32

Microcontrolador que usaremos para trabajar con los datos que los sensores y actuadores enviarán y recibirán respectivamente (se usarán 2 ESP32 una para los actuadores y otra para los sensores).

Sensor MQ9

Sensor que utilizaremos para detectar la presencia de algunos gases como monóxido de carbono (CO), gas licuado del petróleo (GLP) y metano (CH₄).

Sensor MAX30105

Sensor óptico que se utiliza para medir partículas suspendidas en el aire y presencia de humo.

Pantalla OLED

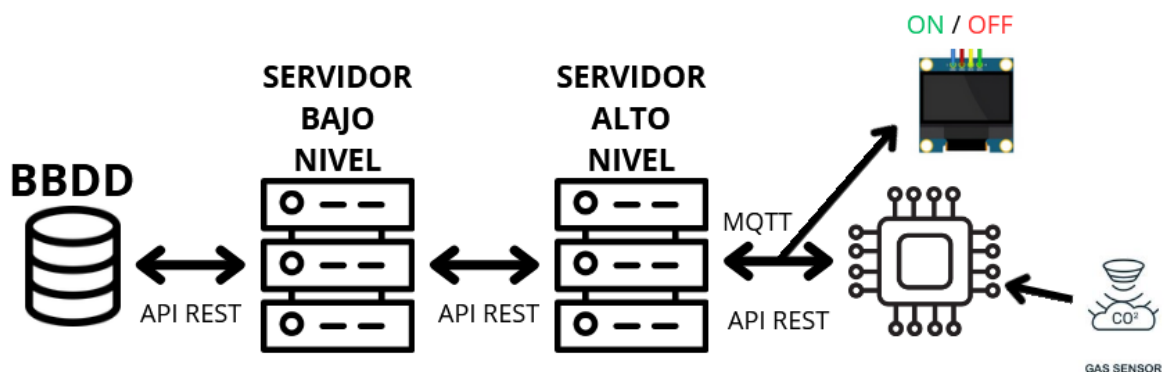
Pantalla que se encargará de mostrar los datos que reciba una esp32.

Bocina

Dispositivo que nos avisará con un sonido en el caso de que los datos recibidos de los gases superen un umbral.

FLUJO DEL PROYECTO

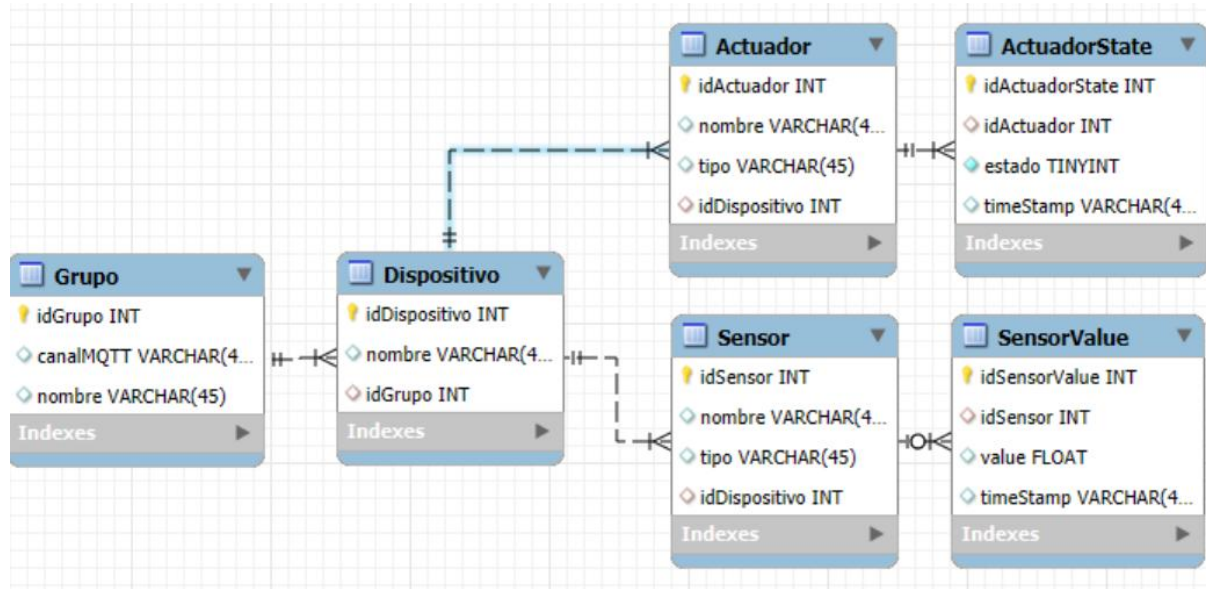
Este proyecto consiste en un sistema donde diferentes partes trabajan juntas para controlar y monitorear sensores y dispositivos. Los datos empiezan desde distintos sensores que detecta gases o partículas conectados a un microcontrolador. Esta placa se comunica mediante REST API con un **servidor de alto nivel**, que es quien se encarga de manipular los actuadores a través de una conexión MQTT con los datos que recibe y reenviar esos mismos datos al **servidor de bajo nivel**. El servidor de bajo nivel se encarga de hablar con la base de datos mediante unas funciones que veremos más adelante para guardar o recuperar la información.



Gracias a esto la lógica del sistema se mantiene desacoplada del almacenamiento, dividiendo el trabajo entre los servidores y favoreciendo una arquitectura más limpia y sostenible. Este enfoque modular facilita la escalabilidad, ya que permite añadir nuevos sensores o actuadores sin necesidad de rediseñar el sistema completo. Además de mejorar el mantenimiento del código al separar claramente las responsabilidades entre procesamiento, comunicación y persistencia de datos.

BASE DE DATOS

Para un mejor manejo de los datos que usamos, hemos creado una base de datos usando MySQL y MySQLWorkbench como nuestro IDE con la siguiente forma:



Esta consta de varias tablas que se relacionan entre sí, para que podamos obtener una organización cohesionada y bien encapsulada. La configuración que hemos usado ha sido la siguiente:

Connection Name: Conexión_Eclipse

Connection Remote Management System Profile

Connection Method: Standard (TCP/IP) Method to use to connect to the RDBMS

Parameters SSL Advanced

Hostname: 127.0.0.1 Port: 3306 Name or IP address of the server host - and TCP/IP port.

Username: IoTAmaso Name of the user to connect with.

Password: Store in Vault ... Clear The user's password. Will be requested later if it's not set.

Default Schema: The schema to use as default schema. Leave blank to select it later.

La tabla *Grupo* incluye un nombre del grupo y el topic a los que estarán suscritos los dispositivos asociados al grupo.

```
CREATE TABLE `grupo` (
  `idGrupo` int NOT NULL AUTO_INCREMENT,
  `canalMQTT` varchar(45) DEFAULT NULL,
  `nombre` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`idGrupo`),
  UNIQUE KEY `idGrupo_UNIQUE` (`idGrupo`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Luego, la tabla *Dispositivo* está formada por las placas que usamos en el proyecto, estas están asociadas un grupo específico dependiendo de la función de la que se encargue.

```
CREATE TABLE `dispositivo` (
  `idDispositivo` int NOT NULL AUTO_INCREMENT,
  `nombre` varchar(45) DEFAULT NULL,
  `idGrupo` int DEFAULT NULL,
  PRIMARY KEY (`idDispositivo`),
  UNIQUE KEY `idDispositivo_UNIQUE` (`idDispositivo`),
  KEY `fk_idGrupo_idx` (`idGrupo`),
  CONSTRAINT `fk_idGrupo` FOREIGN KEY (`idGrupo`) REFERENCES `grupo` (`idGrupo`)
) ENGINE=InnoDB AUTO_INCREMENT=7 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

La tabla *Actuador* recoge el nombre y tipo de cada actuador usado, está asociada a la tabla *Grupo* y tiene también relación con la tabla *ActuadorState* pues esta contiene como clave ajena el identificador de un actuador y se encarga de guardar el estado de estos.

```
DROP TABLE IF EXISTS `actuador`;
/*!40101 SET @saved_cs_client = @@character_set_client */;
/*!50503 SET character_set_client = utf8mb4 */;
CREATE TABLE `actuador` (
  `idActuador` int NOT NULL AUTO_INCREMENT,
  `nombre` varchar(45) DEFAULT NULL,
  `tipo` varchar(45) DEFAULT NULL,
  `idDispositivoAct` int DEFAULT NULL,
  PRIMARY KEY (`idActuador`),
  UNIQUE KEY `idActuador_UNIQUE` (`idActuador`),
  KEY `fk_idDispositivo_idx` (`idDispositivoAct`),
  CONSTRAINT `fk_idDispositivoAct` FOREIGN KEY (`idDispositivoAct`) REFERENCES `dispositivo` (`idDispositivo`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

CREATE TABLE `actuadorstate` (
  `idActuadorState` int NOT NULL AUTO_INCREMENT,
  `idActuador` int DEFAULT NULL,
  `estado` tinyint NOT NULL,
  `timeStamp` varchar(45) DEFAULT NULL,
  `valor` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`idActuadorState`),
  UNIQUE KEY `idActuadorState_UNIQUE` (`idActuadorState`),
  KEY `fk_idActuador_idx` (`idActuador`),
  CONSTRAINT `fk_idActuador` FOREIGN KEY (`idActuador`) REFERENCES `actuador` (`idActuador`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

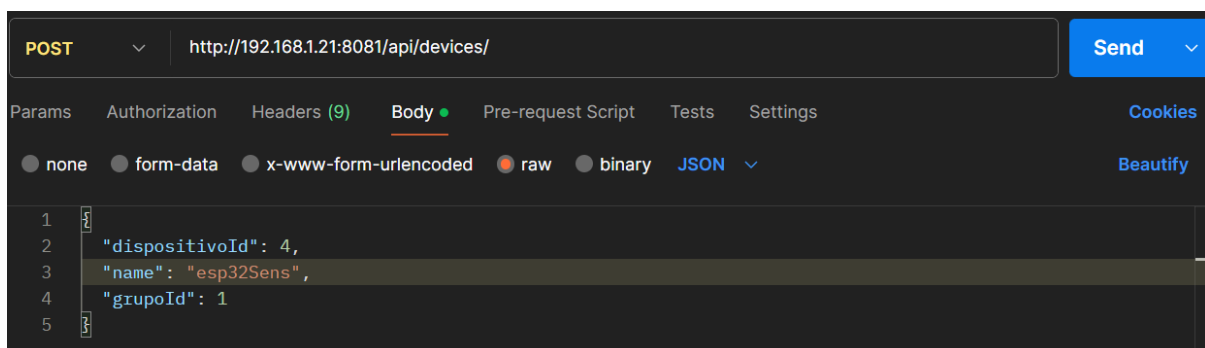
Por último, la tabla *Sensor*, asociada a la tabla *Dispositivo*, guarda el nombre e identificador de los sensores, tiene asociada la tabla *SensorValue* que relaciona cada sensor con los valores que va produciendo.

```
CREATE TABLE `sensor` (
  `idSensor` int NOT NULL AUTO_INCREMENT,
  `nombre` varchar(45) DEFAULT NULL,
  `tipo` varchar(45) DEFAULT NULL,
  `idDispositivo` int DEFAULT NULL,
  PRIMARY KEY (`idSensor`),
  UNIQUE KEY `idSensor_UNIQUE` (`idSensor`),
  KEY `fk_idDispositivo_idx` (`idDispositivo`),
  CONSTRAINT `fk_idDispositivo` FOREIGN KEY (`idDispositivo`) REFERENCES `dispositivo` (`idDispositivo`)
) ENGINE=InnoDB AUTO_INCREMENT=6 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;

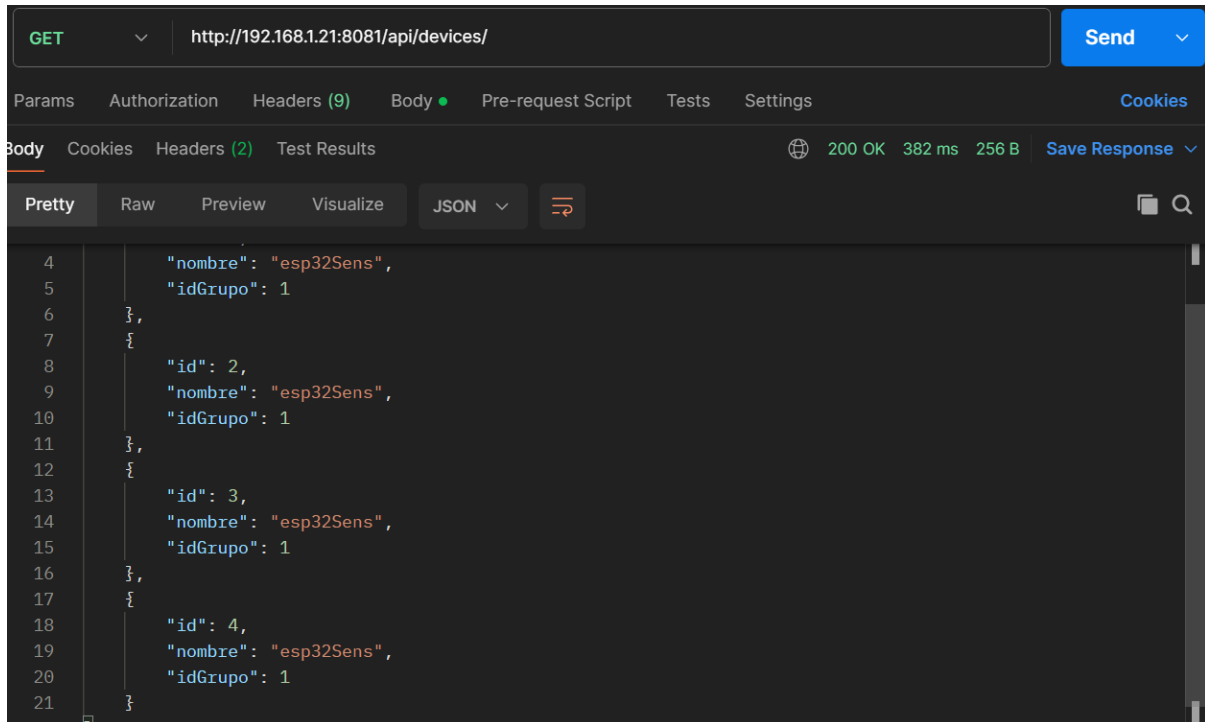
CREATE TABLE `sensorvalue` (
  `idSensorValue` int NOT NULL AUTO_INCREMENT,
  `idSensor` int DEFAULT NULL,
  `value` float DEFAULT NULL,
  `timeStamp` varchar(45) DEFAULT NULL,
  PRIMARY KEY (`idSensorValue`),
  UNIQUE KEY `idSensorValue_UNIQUE` (`idSensorValue`),
  KEY `fk_idSensor_idx` (`idSensor`),
  CONSTRAINT `fk_idSensor` FOREIGN KEY (`idSensor`) REFERENCES `sensor` (`idSensor`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci;
```

Estas relaciones entre tablas que hemos formado nos permiten hacer un buen uso de los datos y trabajar solo con la información que nos interese en el momento, sin tener que filtrar entre multitud de datos para encontrar el que necesitamos.

Para probar que funciona la base de datos, hemos estado haciendo pruebas mediante Postman, tratando de comprobar que las funciones que hemos ido implementando a lo largo del desarrollo del proyecto. Como podemos ver en las imágenes de ejemplo, estamos tratando de hacer Post de un nuevo dispositivo “esp32Sens”.



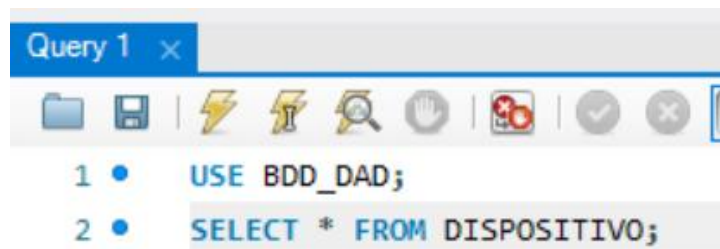
Luego para comprobar que el Post se ha hecho correctamente, hemos intentado hacer un GET para ver los dispositivos que hay actualmente en la base de datos y comprobamos que efectivamente el Post se ha hecho correctamente.



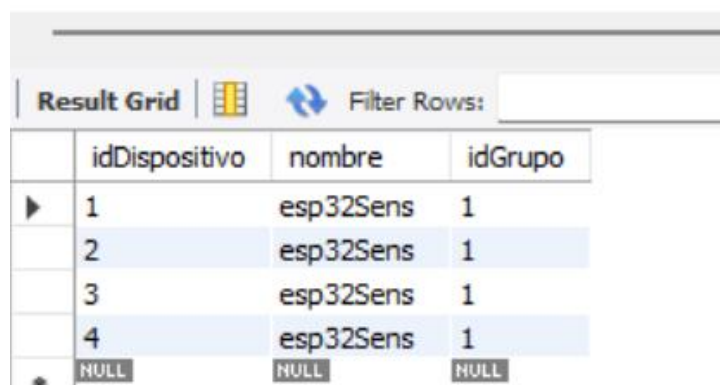
```
GET http://192.168.1.21:8081/api/devices/ 200 OK 382 ms 256 B Save Response

[{"id": 1, "nombre": "esp32Sens", "idGrupo": 1}, {"id": 2, "nombre": "esp32Sens", "idGrupo": 1}, {"id": 3, "nombre": "esp32Sens", "idGrupo": 1}, {"id": 4, "nombre": "esp32Sens", "idGrupo": 1}]
```

Como última comprobación, aunque realmente con el GET ya bastaría, vamos a comprobar que el resultado de esta última función se ha mostrado correctamente viendo en MySQL Workbench el contenido de la tabla, que como podemos ver, coinciden.



```
Query 1 x
1 • USE BDD_DAD;
2 • SELECT * FROM DISPOSITIVO;
```



	idDispositivo	nombre	idGrupo
▶	1	esp32Sens	1
	2	esp32Sens	1
	3	esp32Sens	1
	4	esp32Sens	1
•	NULL	NULL	NULL

SERVIDOR BAJO NIVEL

El **servidor de bajo nivel** se encarga de gestionar directamente la información que se guarda en la base de datos. Sus funciones principales son recibir, almacenar, actualizar y eliminar datos relacionados con sensores, actuadores, dispositivos, etc., a través de una API REST con puerto 8081. Actúa como intermediario entre la base de datos y el servidor de alto nivel, asegurando que los datos estén organizados y accesibles cuando se necesiten. No toma decisiones complejas, solo realiza operaciones básicas con los datos.

Endpoints

Para cada entidad creada en nuestro proyecto, hemos creado una serie de endpoints que nos dan acceso a algunas funciones útiles.

Las entidades que hemos creado son:

- **Sensor:** Describe características del sensor como el tipo o a que dispositivo está conectado.
- **SensorValue:** Es el valor que nos proporciona el sensor asociado mediante una id.
- **Actuador:** Describe características del actuador como el tipo o a que dispositivo está conectado.
- **ActuadorState:** Es el estado del actuador al que está asociado mediante una id.
- **Dispositivo:** Representa la placa o microcontrolador, ya que usaremos varias.
- **Grupo:** Representa el canal MQTT o topic a los que se asociará un dispositivo.

Estos serían nuestros endpoints:

- **Sensor:** Estos endpoints se encargan de gestionar los sensores conectados.
 - `http://IP...:8081/api/sensors`
 - GET: Obtenemos todos los sensores
 - POST: Añadimos un nuevo sensor. En este caso, hay que darle un cuerpo.
 - `http://IP...:8081/api/sensors/:id`
 - GET: Obtenemos el sensor con la id indicada.
 - DELETE: Eliminamos el sensor con la id indicada.
 - PUT: Modificamos el sensor con la id indicada. En este caso, también hay que darle un cuerpo.
- **SensorValue:** Estos endpoints se encargan de gestionar los valores que los sensores nos proporcionan.
 - `http://IP...:8081/api/values`

- POST: Añadimos un nuevo valor de algún sensor. En este caso, lo hace automáticamente la esp32.
 - `http://IP...:8081/api/values/:id_sensor`
 - GET: Obtenemos todos los valores que nos ha proporcionado el sensor cuya id es “id_sensor”.
- **Actuador:** Estos endpoints se encargan de gestionar los actuadores conectados.
 - `http://IP...:8081/api/actuators`
 - GET: Obtenemos todos los actuadores.
 - POST: Añadimos un nuevo actuador. En este caso, hay que darle un cuerpo.
 - `http://IP...:8081/api/actuators/:id`
 - GET: Obtenemos el actuador con la id indicada.
 - DELETE: Eliminamos el actuador con la id indicada.
 - PUT: Modificamos el actuador con la id indicada. En este caso, también hay que darle un cuerpo.
- **ActuadorState:** Estos endpoints se encargan de gestionar los actuadores conectados.
 - `http://IP...:8081/api/states`
 - POST: Añadimos un nuevo estado de algún actuador. En este caso, lo hace automáticamente la esp32.
 - `http://IP...:8081/api/states/:id_actuador`
 - GET: Obtenemos todos los estados que nos ha proporcionado el actuador cuya id es “id_actuador”.
- **Dispositivo:** Estos endpoints se encargan de gestionar los dispositivos conectados.
 - `http://IP...:8081/api/devices`
 - GET: Obtenemos todos los dispositivos.
 - POST: Añadimos un nuevo dispositivo. En este caso, hay que darle un cuerpo.
 - `http://IP...:8081/api/devices/:id`
 - GET: Obtenemos el dispositivo con la id indicada.
 - DELETE: Eliminamos el dispositivo con la id indicada.
 - PUT: Modificamos el dispositivo con la id indicada. En este caso, también hay que darle un cuerpo.
- **Grupo:** Estos endpoints se encargan de gestionar los grupos conectados.
 - `http://IP...:8081/api/groups`
 - GET: Obtenemos todos los grupos.
 - POST: Añadimos un nuevo grupo. En este caso, hay que darle un cuerpo.
 - `http://IP...:8081/api/groups/:id`

- GET: Obtenemos el grupo con la id indicada.
- DELETE: Eliminamos el grupo con la id indicada.
- PUT: Modificamos el grupo con la id indicada. En este caso, también hay que darle un cuerpo.

Funciones Base de Datos

Son funciones muy útiles que sirven para conectarnos directamente con la base de datos y que se gestione automáticamente cuando recibe una petición de REST API.

¿Cómo funcionan?

Cuando hacemos algún tipo de petición del estilo POST, GET, PUT, ETC.... Se ejecuta automáticamente x función asociada mediante un handler.

```
router.post("/api/sensors").handler(this::addOneSensor);
```

Cuando se hace post en el puerto 8081, se llama a esa función:

```
private void addOneSensor(RoutingContext routingContext) {
    final Sensor aux = gson.fromJson(routingContext.getBodyAsString(),
Sensor.class);
    Integer idSensor = aux.getIdentificador();
    Integer idPlaca = aux.getDeviceID();
    String tipoSensor = aux.getTipo();
    String nombre = aux.getNombre();

    if (idSensor == null || idPlaca == null || tipoSensor == null || nombre
== null) {
        routingContext.response().putHeader("content-type",
"application/json; charset=utf-8")
        .setStatusCode(400)
        .end("Campos requeridos (NOT NULL): idSensor, idPlaca,
tipoSensor, nombre");
        return;
    }

    mySQLClient.getConnection(connection -> {
        if (connection.succeeded()) {
            SqlConnection conn = connection.result();

            // Comprobar si ya existe
            conn.preparedQuery("SELECT * FROM SENSOR WHERE idSensor = ?")
            .execute(Tuple.of(idSensor), selectRes -> {
                if (selectRes.succeeded()) {
                    if (selectRes.result().rowCount() > 0) {
                        routingContext.response().putHeader("content-
type", "application/json; charset=utf-8")
                        .setStatusCode(409) // Conflict
                        .end("Ya existe un sensor con idSensor = "
+ idSensor);
                        conn.close();
                    } else {
```

```

        conn.preparedQuery("INSERT INTO
SENSOR(idSensor, nombre, tipo, idDispositivo) VALUES (?, ?, ?, ?)")
        .execute(Tuple.of(idSensor, nombre,
tipoSensor, idPlaca), insertRes -> {
            if (insertRes.succeeded()) {
                routingContext.response().putHeader("content-type", "application/json;
charset=utf-8")
                .setStatusCode(201)
                .end("Sensor añadido");
            } else {
                routingContext.response().putHeader("content-type", "application/json;
charset=utf-8")
                .setStatusCode(500)
                .end("Error al añadir: " +
insertRes.cause().getMessage());
            }
            conn.close();
        });
    } else {
        routingContext.response().putHeader("content-type",
"application/json; charset=utf-8")
        .setStatusCode(500)
        .end("Error al comprobar existencia: " +
selectRes.cause().getMessage());
        conn.close();
    }
});
} else {
    routingContext.response().putHeader("content-type",
"application/json; charset=utf-8")
    .setStatusCode(500)
    .end("Error al conectar con la base de datos: " +
connection.cause().getMessage());
}
});
}
}

```

Esta función, comprueba si el sensor que se va a añadir existe en la base de datos. Si existe, lo rechaza y te salta un mensaje diciendo que ya hay un sensor con id x. Si no existe, lo añade a la base de datos mediante un INSERT. Como se puede ver, estas funciones hacen uso de SQL para ahorrarnos la gestión manual de los datos, lo cual es muy útil.

SERVIDOR ALTO NIVEL

El **servidor de alto nivel** es el que se encarga de la comunicación entre los dispositivos físicos (placas, sensores y actuadores) y el sistema. Al recibir datos, este servidor se encarga de reenviarlos al servidor de bajo nivel para su almacenamiento y, en paralelo, publica mensajes MQTT que permiten activar, desactivar o actualizar los actuadores según los valores de los sensores.

REST API

Al igual que hemos hecho en el servidor de bajo nivel, aquí haremos uso de endpoints. Sin embargo, para el servidor de alto nivel usaremos el puerto 8080 y las funciones que llamarán serán para reenviar los datos al servidor de bajo nivel mediante un post al 8081.

```
Router router = Router.router.vertx();
vertx.createHttpServer().requestHandler(router::handle).listen(8080, result -> {
    if (result.succeeded()) {
        startFuture.complete();
    } else {
        startFuture.fail(result.cause());
    }
});
router.route("/api/groups*").handler(BodyHandler.create());
router.route("/api/devices*").handler(BodyHandler.create());
router.route("/api/sensors*").handler(BodyHandler.create());
router.route("/api/actuators*").handler(BodyHandler.create());
router.route("/api/values*").handler(BodyHandler.create());
router.route("/api/states*").handler(BodyHandler.create());

router.post("/api/groups").handler(this::accionGroupPost);
router.post("/api/devices").handler(this::accionDevicePost);
router.post("/api/sensors").handler(this::accionSensorPost);
router.post("/api/actuators").handler(this::accionActuatorPost);
router.post("/api/values").handler(this::accionValuePost);
router.post("/api/states").handler(this::accionStatePost);
```

```
private void accionGroupPost(RoutingContext routingContext) {
    try {
        Grupo group = gson.fromJson(routingContext.getBodyAsString(), Grupo.class);
        System.out.println("Grupo: " + group);
        webClient.post(8081, IP, "/api/groups")
            .sendBuffer(Buffer.buffer(gson.toJson(group)), res -> {
                if (res.succeeded()) {
                    routingContext.response().setStatusCode(201).end("Dato recibido y re");
                } else {
                    routingContext.response().setStatusCode(500).end("Error reenviando a");
                }
            });
    } catch (Exception e) {
        routingContext.response().setStatusCode(400).end("JSON malformado");
    }
}
```

Estas funciones tienen un contenido corto

MQTT

El protocolo MQTT es un elemento fundamental en la arquitectura del proyecto, ya que permite la comunicación eficiente y en tiempo real entre el servidor de alto nivel y los actuadores y sensores físicos conectados a las placas ESP32.

En nuestro proyecto, el servidor de alto nivel implementa la lógica para el control de los actuadores y sensores a través de la publicación de mensajes en distintos topics MQTT. Cuando la placa ESP32 de los sensores envía datos en formato JSON, el servidor convierte ese mensaje en un tipo `SensorValue`.

```
SensorValue value = gson.fromJson(routingContext.getBodyAsString(), SensorValue.class);
System.out.println("Valor del sensor: " + value);
```

Posteriormente guardamos el ID del mensaje y lo comparamos para ver del sensor que proviene para publicar un topic en un canal con el valor del sensor que nos ha llegado, además si ese valor supera o no un umbral dependiendo del sensor asociado se publicará en otro canal un topic con la bocina a ON u OFF

```
if (ID.equals(mq9CH4)) {
    mq9CH4Value = value.getValue();
    oledMessage = ID + "CH4: " + mq9CH4Value;
}
if (ID.equals(mq9C0)) {
    mq9C0Value = value.getValue();
    oledMessage = ID + "CO: " + mq9C0Value;
}
if (ID.equals(mq9GLP)) {
    mq9GLPValue = value.getValue();
    oledMessage = ID + "GLP: " + mq9GLPValue;
}
if (ID.equals(MAX)) {
    maxValue = value.getValue();
    oledMessage = ID + "Particulas: " + maxValue;
}
```

```
mqttClient.publish(act1, Buffer.buffer(oledMessage), MqttQoS.AT_LEAST_ONCE, false, false);
```

```
if(ID.equals(mq9CH4) && value.getValue()>2000){
    mqttClient.publish(act2, Buffer.buffer("ON"),
        MqttQoS.AT_LEAST_ONCE, false, false);
    System.out.println("Bocina ON");
}
else if(ID.equals(mq9C0) && value.getValue() > 450){
    mqttClient.publish(act2, Buffer.buffer("ON"),
        MqttQoS.AT_LEAST_ONCE, false, false);
    System.out.println("Bocina ON");
}
else if(ID.equals(mq9GLP) && value.getValue() > 450){
    mqttClient.publish(act2, Buffer.buffer("ON"),
        MqttQoS.AT_LEAST_ONCE, false, false);
    System.out.println("Bocina ON");
}
else if(ID.equals(MAX) && value.getValue() > 10000){
    mqttClient.publish(act2, Buffer.buffer("ON"),
        MqttQoS.AT_LEAST_ONCE, false, false);
    System.out.println("Bocina ON");
}
else {
    mqttClient.publish(act2, Buffer.buffer("OFF"),
        MqttQoS.AT_LEAST_ONCE, false, false);
    System.out.println("Bocina OFF");
}
```

Esto ocurre cada vez que se ejecuta un post en el servidor de alto nivel

DESARROLLO DEL FIRMWARE

Esta sección del proyecto fue desarrollada utilizando PlatformIO dentro del entorno Visual Studio Code, una extensión de esta plataforma robusta y flexible para programar microcontroladores. Las placas están programadas para interactuar directamente con el servidor mediante peticiones POST, enviando los valores recolectados por los sensores y reportando los estados actuales de los actuadores. Además, desde el firmware también se modifican los actuadores en respuesta a los mensajes MQTT que recibe mediante la suscripción a un canal, lo que permite un control dinámico del sistema. Como parte de su funcionalidad clave, las placas también crean los datos iniciales y esenciales del sistema al iniciar (grupos, sensores, actuadores, dispositivos, etc...), asegurando que la infraestructura básica esté siempre disponible y operativa.

Estas es la configuración que hemos usado para el proyecto, además de las librerías que hemos necesitado para el correcto funcionamiento.

```
[env:node32s]
platform = espressif32
board = node32s
framework = arduino
lib_deps =
  mikalhart/TinyGPSPlus@^1.0.2
  bblanchon/ArduinoJson@^6.17.3
  knolleary/PubSubClient@^2.8
  robbie-remote/RESTClient@^1.0.0
  arduino-libraries/NTPClient@^3.2.1
  abcdaaaaaaaa/MQSpaceData@^3.1.0
  adafruit/Adafruit SSD1306@^2.5.14
  adafruit/Adafruit GFX Library@^1.12.1
  arduino-libraries/Ethernet@^2.0.2
  miguel5612/MQUnifiedsensor@^3.0.0
  sparkfun/SparkFun MAX3010x Pulse and Proximity Sensor Library@^1.1.2
```

Librerías para pantalla OLED

Librería MQ9

Librería MAX

En nuestro caso, hemos hecho dos pequeños proyectos, uno para la placa con los actuadores y otro para placa con los sensores. Además, en nuestro caso la inserción manual de datos la hacen automáticamente las placas, pero también le hemos subido otro SQL con el código para la inserción manual.

```
String jsonGrupo1= serializeGroup(1,"esp32/actuador/#","actuador");
String jsonDispositivo2=serializeDeviceBody(1,"actuadores",1);
String jsonActuador1=serializeActuatorBody(0,"OLED","OLED",1);
String jsonActuador2=serializeActuatorBody(1,"BOCINA","BOCINA",1);

POST_grupos(jsonGrupo1);
POST_device(jsonDispositivo2);
POST_actuador(jsonActuador1);
POST_actuador(jsonActuador2);
```

Así es como hacemos post:

```
void POST_sensores(String JSON)
{
  String actuator_states_body = JSON;
  describe((char*)"Post estado sensor");
  String serverPath = serverName + "/api/values";
  http.begin(serverPath.c_str());
  test_response(http.POST(actuator_states_body));
  http.end();
  delay(1000);
}
```

```
String serverName = "http://192.168.66.18:8080/";
```

La variable serverName nos lleva al servidor de alto nivel.

Así es la serialización de un valor:

```
String serializeSensorValueBody(int idSensor, long timestamp, float value)
{
    DynamicJsonDocument doc(2048);

    doc["sensorValueId"]=idSensorValue;
    doc["sensorId"] = idSensor;
    doc["timestamp"] = timestamp;
    doc["value"] = value;

    idSensorValue++;

    String output;
    serializeJson(doc, output);
    Serial.println(output);

    return output;
}
```

En el loop ejecutamos constantemente los posts con el respectivo valor.

```
void loop()
{
    leerMQ9();
    leerMAX();
    String valorMAX= serializeSensorValueBody(3,millis(),ir);
    String valorCOMQ9= serializeSensorValueBody(0,millis(),CO);
    String valorCH4Q9= serializeSensorValueBody(1,millis(),CH4);
    String valorLPGMQ9= serializeSensorValueBody(2,millis(),LPG);
    POST_sensores(valorCOMQ9);
    POST_sensores(valorCH4Q9);
    POST_sensores(valorLPGMQ9);
    POST_sensores(valorMAX);
    Serial.println("Valor co");
    Serial.println(CO);
    Serial.println("Valor lpg");
    Serial.println(LPG);
    Serial.println("Valor ch4");
    Serial.println(CH4);
}
```

¿Y cómo hacemos la impresión en la pantalla?

```
if (top == "esp32/actuador/OLED") {  
  if (content != "connected") {  
  
    Serial.println("Mensaje recibido en OLED");  
  
    display.setTextSize(1);  
    display.setTextColor(SSD1306_WHITE);  
    if (id == '0') {  
      display.setCursor(0, 0);  
      display.fillRect(0, 0, 128, 8, SSD1306_BLACK);  
      datoCH4 = dato;  
      display.print(dato);  
    } else if (id == '1') {  
      display.setCursor(0, 8);  
      display.fillRect(0, 8, 128, 8, SSD1306_BLACK);  
      datoCo = dato;  
      display.print(dato);  
    } else if (id == '2') {  
      display.setCursor(0, 16);
```

```
id = content.charAt(0); // E  
dato = content.substring(1);
```

Los mensajes en MQTT son:

1CH4: Valor

De tal manera que con la id que es el primer carácter, comprobamos que sensor es. Y con el resto imprimimos el dato. Según que sensor, irá en una línea de la pantalla oled u otra.

En cambio, la bocina es mucho más sencillo. Simplemente revisa el mensaje en MQTT y si el mensaje es "ON", se activa.

```
else if (top == "esp32/actuador/BOCINA") {  
  if (content != "connected") {  
    if (content == "ON") {  
      Serial.println("Encendida");  
      bocinaState = true;  
      digitalWrite(12, HIGH);  
      delay(2000);  
      digitalWrite(12, LOW);  
    } else {  
      digitalWrite(12, LOW);  
      bocinaState = false;  
    }  
  }  
  Serial.println("Mensaje recibido en BOCINA");  
}
```

Para la obtención de los valores de los sensores, hemos usado los ejemplos que nos proporcionaban las bibliotecas del respectivo sensor. No obstante, hay que mencionar que sensores como el MQ-9 nos ha dado problemas a la hora de calibrarlos. Además, ese mismo sensor nos proporcionaba tres datos distintos. Para solucionarlo hemos creado un sensor más para cada dato y le hemos asociado una id. Por lo tanto, aunque en la realidad tengamos solo dos sensores, en la práctica tenemos cuatro (Tres del MQ-9 más el MAX).

CONCLUSIÓN

A lo largo del desarrollo de este proyecto, hemos conseguido diseñar e implementar un sistema completo y funcional para la detección de emisiones contaminantes en vehículos. Combinando conocimientos de electrónica, programación y arquitectura de sistemas distribuidos, hemos integrado sensores específicos con microcontroladores ESP32 capaces de captar y transmitir datos en tiempo real.

Hemos diseñado una base de datos relacional eficiente que nos ha permitido almacenar de forma estructurada los registros de sensores, actuadores, dispositivos y sus respectivos estados.

Finalmente, el desarrollo del firmware nos ha permitido cerrar el ciclo de funcionamiento del sistema, asegurando una comunicación fluida entre las placas y los servidores, así como una respuesta inmediata ante situaciones críticas, como la detección de niveles peligrosos de contaminantes.

En conjunto, consideramos que el proyecto ha alcanzado con éxito sus objetivos, y hemos logrado una solución tecnológica útil no solo desde el punto de vista técnico, sino también como herramienta de concienciación ambiental. Este trabajo nos ha permitido aplicar de forma práctica muchos de los conocimientos adquiridos durante nuestra formación, además durante el todo el proceso hemos encontrado numerosos imprevistos que hemos tenido que aprender a solucionar, mejorando así nuestra capacidad de hacer troubleshooting. Los problemas que nos han surgido han sido, por ejemplo, el que hemos tenido al calibrar el sensor MQ9 por el funcionamiento interno del propio sensor y las librerías que usa, además, cuando intentamos conectar la base de datos con el servidor de bajo nivel, nos surgieron problemas que no vimos en el aula, y tuvimos que buscar herramientas como StackOverflow para solucionarlos, para finalizar, la última dificultad que tuvimos fue que nos costó entender de forma teórica el funcionamiento de MQTT.