Segundo Entregable

PROYECTO COVIDBUS

Javier Roldán Marín

Iván Romero Pastor

• Introducción:

Hemos decidido realizar el proyecto de una compañía de autobuses con el nombre de CovidBus, para recalcar la comprometida situación en la que nos encontramos actualmente y el aprovechamiento de esto para poder introducir nuevas mejoras en estos, para aportar mayor seguridad y confianza a los consumidores, efectuando así un mayor y mejor uso de los transportes públicos para las personas.

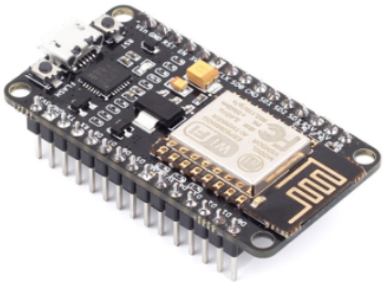
Por esto hemos desarrollado un dispositivo que vaya conectado al autobús que nos indique al usuario la proximidad del autobús a nosotros, con esto estimaremos el tiempo de espera en la parada. Podríamos consultar la temperatura dentro del autobús, así como de la humedad que hay en él.

Además, como idea para el cliente,queremos añadir un pulsador (*botón. en Java de tipo boolean)* mostrado en la aplicación, y que con esto podamos indicar al dispositivo que queremos que el autobús se detenga en la siguiente parada.

Todo ello con el fin de proporcionar al usuario información acerca del autobús, y así poder decidir si es más seguro coger un autobús u otro en estos tiempos que corren.

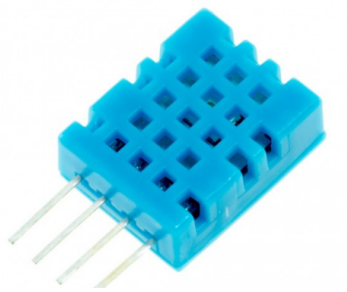
• HARDWARE EMPLEADO:

- ESP8266:



El ESP8266 es un chip Wi-Fi de bajo coste con pila TCP/IP completa y capacidad de MCU. Este pequeño módulo permite a los microcontroladores conectarse a una red Wi-Fi y realizar conexiones TCP/IP.

- DHT11:



El DHT11 es un sensor digital de temperatura y humedad. Muestra los datos mediante una señal digital en el pin de datos (no posee salida analógica) y solo es necesario conectar el pin VCC de alimentación a 3-5V, el pin GND a Tierra (0V) y el pin de datos a un pin digital en nuestro ESP8266.

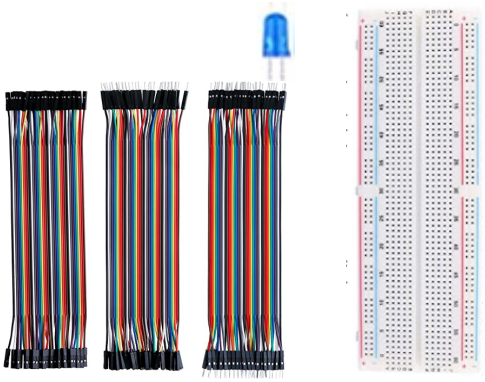
- MQ2:

El MQ-2 Sensor de Gas tiene una sensibilidad especial para medir concentraciones de gas en el aire. Incluye una salida digital que se calibra con un potenciómetro en el módulo en conjunto con un Led indicador. La resistencia del sensor cambia de acuerdo a la concentración del gas en el aire.

- GPS GY-NEO6MV2 NEO-6M:

Módulo GPS 3v-5v, con antena de cerámica. El cual es utilizado para el cálculo de coordenadas que transmite dicho sensor.

- Componentes varios:



Componentes utilizados para interconectar el resto de sensores a la placa, así mismo del uso del led para el uso interacción del usuario con el dispositivo del autobús al requerir una acción.

• EXPLICACIÓN PARTE HARDWARE:

- Librerías empleadas hasta el momento:

Librerías básicas:
#include <Arduino.h>
#include <ESP8266WiFi.h>
#include <ArduinoJson.h>

Librerías cliente HTTP Y MQTT:
#include <ArduinoHttpClient.h>
#include <PubSubClient.h>

Librerías de sensores:
Sensor DHT-11:
#include "DHT.h"
#include <Adafruit_Sensor.h>
#include <DHT_U.h>
Sensor MQ-2: No necesaria
Sensor NEO-6M: (no válidas por el momento)
#include <TinyGPS++.h>
#include <SoftwareSerial.h>

Librería para el Ticker:
#include <Ticker.h>

Librería para el Tiempo: (no válidas por ahora)
#include <Time.h>
#include "TimeLib.h"
#include <DS1307RTC.h>
#include <Wire.h>

- ## Organización del Código:

  - **Includes.**

  - **Declaración e inicialización de variables generales: (línea 22)**

    - Sensor DHT-11→ línea 25
    - Sensor MQ-2 → línea 31
    - Sensor NEO-M6 → línea 34
    - Variables para la librería TIME → línea 42
    - Variables para la conexión http y mqtt → línea 49

  - **Funciones Auxiliares:** → línea 73

    - Conexión Wifi → línea 78
      - void setup_wifi()

    - Conexión MQTT → línea 92
      - void callback(char* topic, byte* payload, unsigned int length()
      - void mqtt_reconnect()
      - void mqtt_setup()
      - mqtt_loop()

    - Funciones del TIMER → línea 159
      (no válidas por el momento)

    - Deserialize → línea 199 (no usadas por el momento)

- Serialize → línea 313
    - String serialize_Sensor_Info(int idsensor, String tipo, String nombre, float last_value1, float last_value2, int iddispositivo)
    - String serialize_Sensor_Data(String timestamp,float valor1, float valor2, int idsensor)

- Funciones de métodos REST de prueba → línea 346 (dejadas como ejemplos REST)

- Funciones de sensores → línea 432
    - void sensor_DHT11() → línea 435
    - void sensor_MQ_2() → línea 505
    - void displayInfo() → para el GPS

- **SETUP()** → línea 647

- **LOOP()** → línea 726

- Descripción de la función SETUP ( )

0) Antes de la función setup(), inicializamos los Ticker del DHT11 y MQ-2:

```
Ticker timer_dht11(sensor_DHT11, 7000);
Ticker timer_mq2(sensor_MQ_2, 10000);
```

1) Llamamos a setup_wifi() que inicia la conexión wifi del ESP

2) Llamamos a setup_mqtt() que inicia la conexión MQTT del ESP8266 con el broker de mosquitto

3) Hacemos un POST con URL → /api/Post_Info_Sensor/ para inicializar el sensor DHT-10 en la base de datos
- Imprimimos el código de estado y cuerpo del mensaje enviado

4) Hacemos un POST con URL → /api/Post_Info_Sensor/ para inicializar el sensor MQ-2 en la base de datos
- Imprimimos el código de estado y cuerpo del mensaje enviado

5) Lanzamos los Tickers con la función .start()

- Descripción de la función LOOP ( )

1) Llamamos constantemente a la función mqtt_loop()

2) Usamos constantemente la llamada a la función .update() para ambos sensores disponibles
(función update() de la librería de Ticker.h)

- Descripción de funciones DHT-11 y MQ-2

1) Leemos los datos recibidos por los pines del ESP8266
- Imprimimos los datos por consola

2) Hacemos un PUT con URL → /api/PutInfoSensor/+ id_sensor
- Imprimimos el código de estado y cuerpo del mensaje enviado

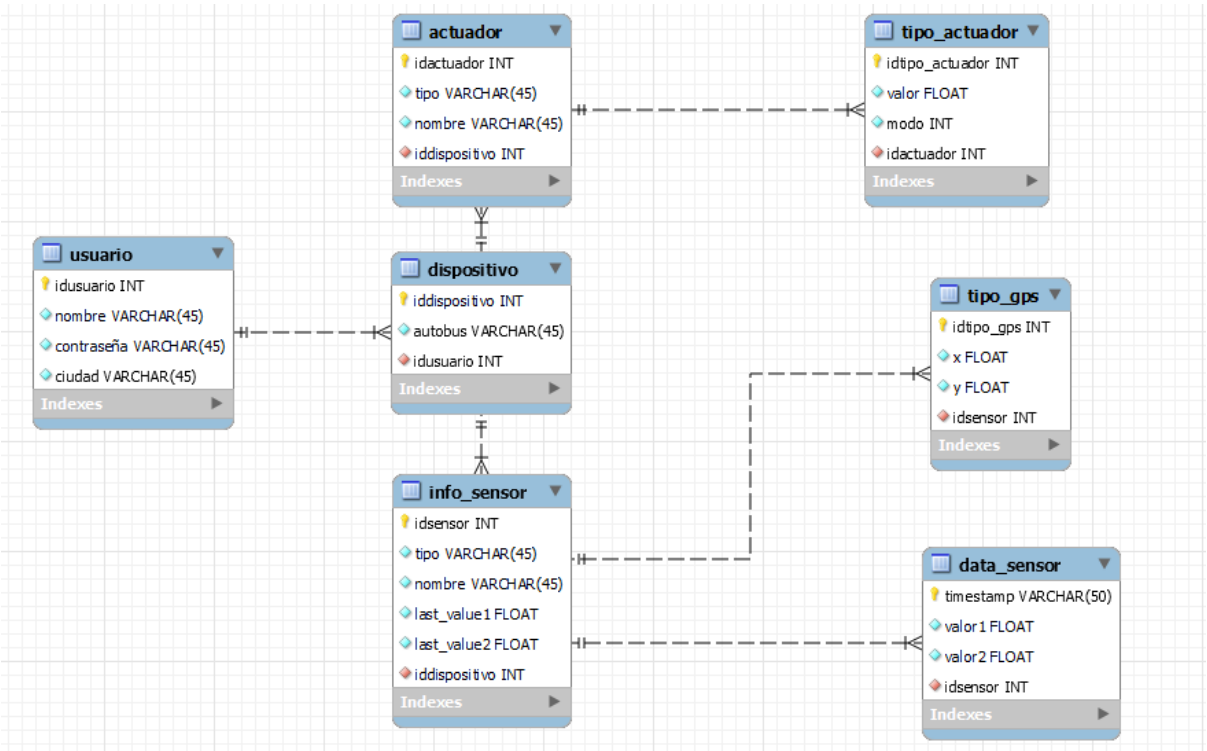
3) Hacemos un POST con URL → /api/Post_Data_Sensor/
- Imprimimos el código de estado y cuerpo del mensaje enviado

• Base de datos:

Nuestra base de datos, se trata de una inicialmente simplificada con el fin de que no sea muy laborioso trabajar con ella. Por lo que, hemos creado las siguientes tablas:

- ➔ **Usuario:** Donde almacenaremos la información correspondiente de cada usuario de la aplicación.
- ➔ **Dispositivo:** Este irá colocado en cada autobús.
- ➔ **Info_Sensor:** Donde se fijará el tipo de sensor que es y a que dispositivo está conectado, así mismo de los dos últimos valores producidos por dicho sensor.
- ➔ **Data_Sensor y Tipo GPS:** Almacenan la información extraída de los sensores cada cierto tiempo(En este caso sensore tipo gps, humedad, temperatura..).
- ➔ **Actuador:** Donde se fijará el tipo de actuador que es y a que dispositivo está conectado.
- ➔ **Tipo_Actuador:** Almacenan la información extraída de los actuadores(Como actuadores tipo led, sonido).

Como resultado de todas estas tablas, obtenemos el siguiente diagrama UML de la base de datos:



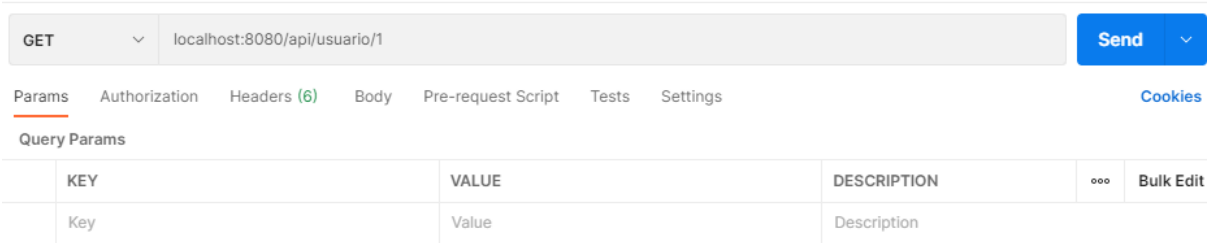
• API Rest:

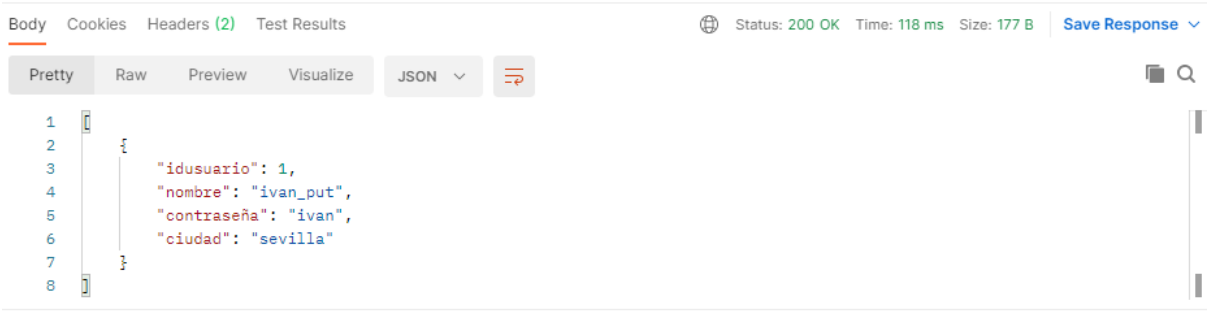
A continuación explicamos los diferentes métodos o servicios REST que tenemos añadidos en el proyecto por ahora, para compartir recursos e información entre los usuarios y el servidor:

- **GET :** Usados para obtener información de la base de datos, los métodos get no usan cuerpo, usan de la URL para solicitar la información deseada.

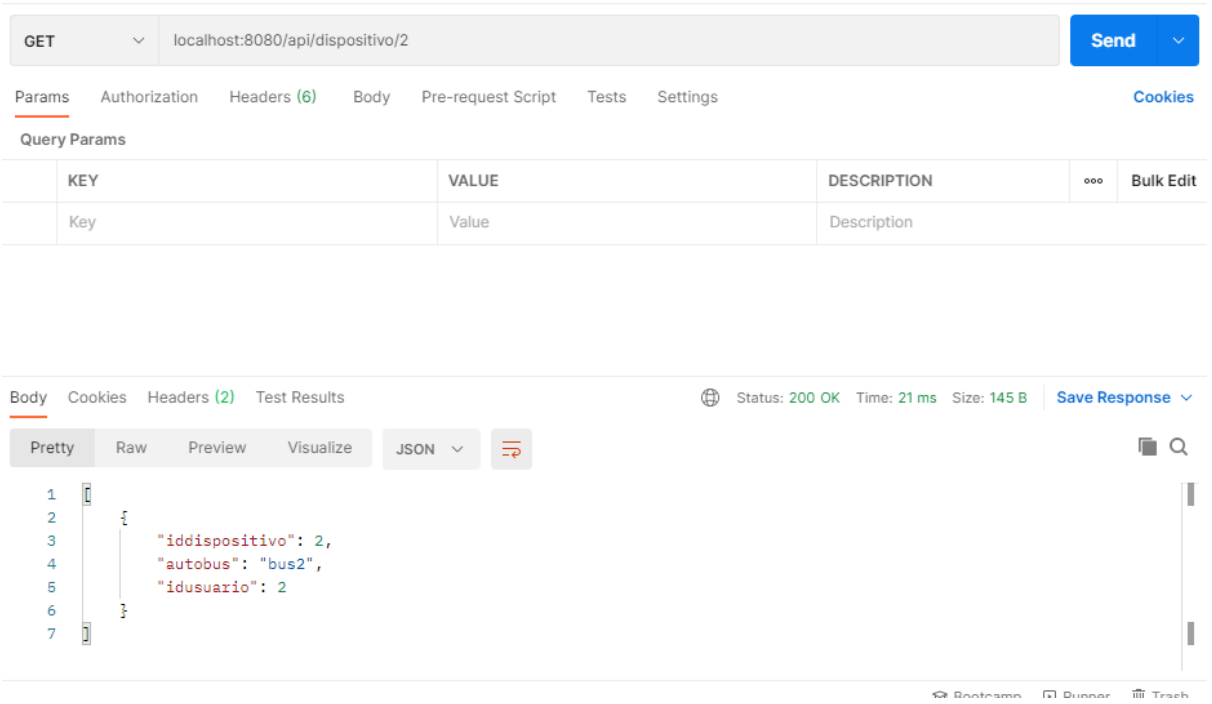
  Hemos introducido este método para todas las tablas.

  - this::getUsuario: mediante el identificador único de usuario, obtenemos la información de un usuario específico.
    URL introducida: "/api/usuario/:idusuario"

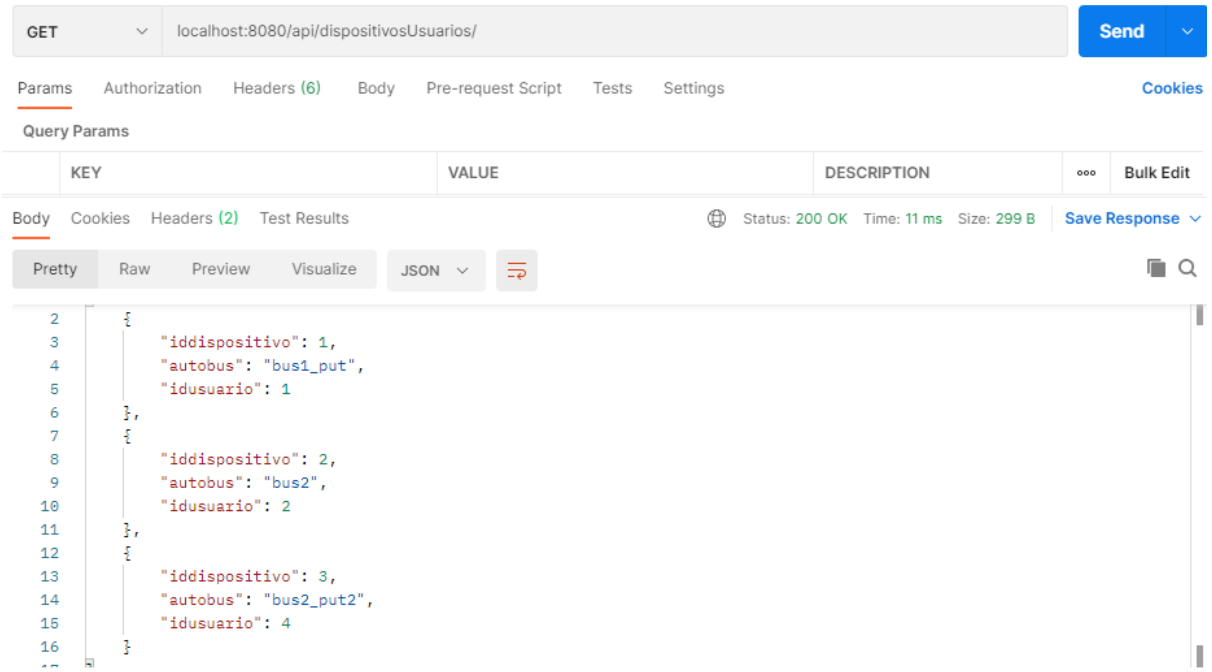




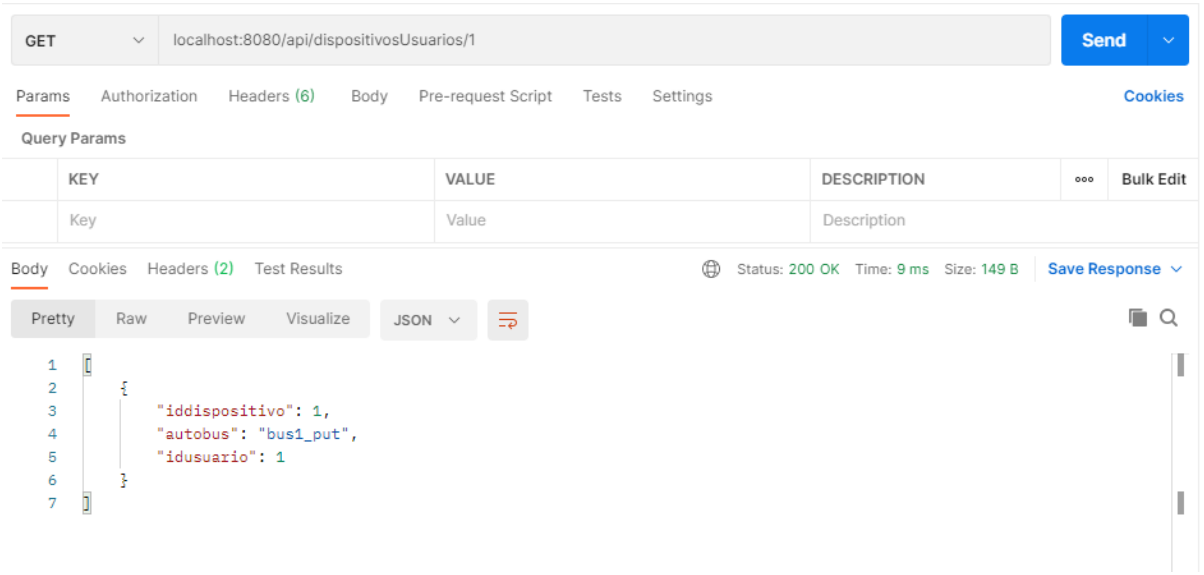
- this::getDipositivo: mediante el id de un dispositivo, obtenemos toda su información asociada.
  URL introducida: "/api/dispositivo/:iddispositivo"



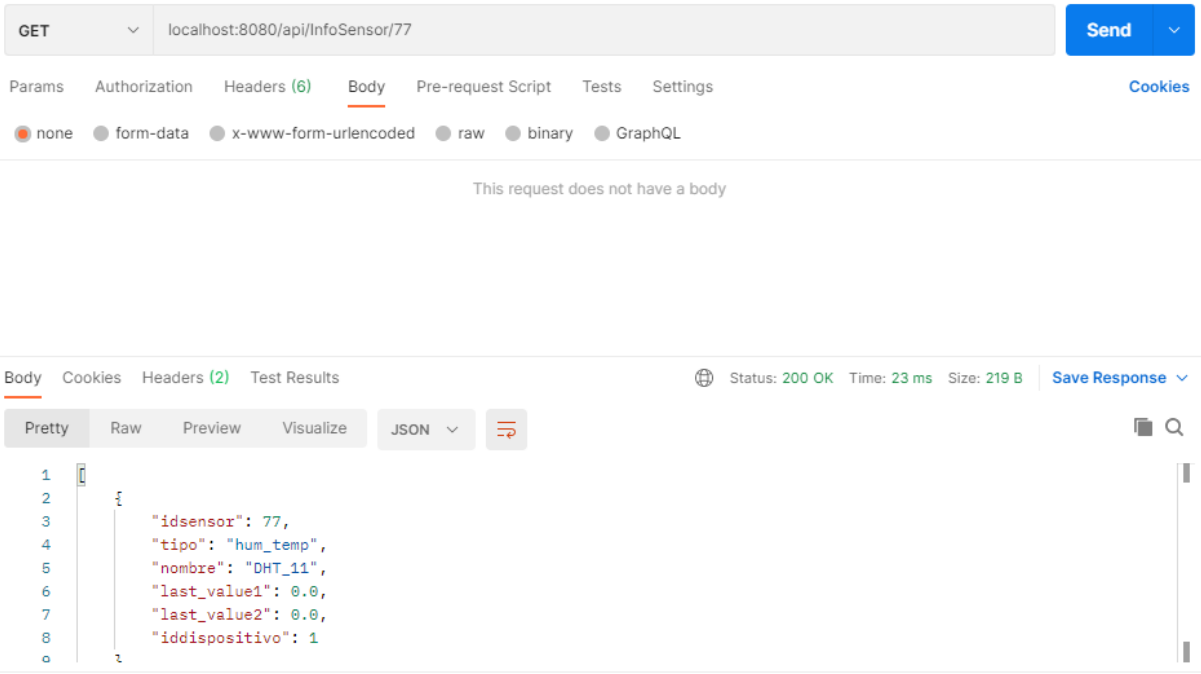
- this::getDipositivosUsuarios: este método nos devuelve todos los dispositivos registrados.
  URL introducida: "/api/dispositivosUsuarios/"



- this::getDipositivoUsuario: método que dado un id de usuario, te devuelve en que bús(dispositivo) está montado.
  URL introducida: "/api/dispositivosUsuarios/:idusuario"

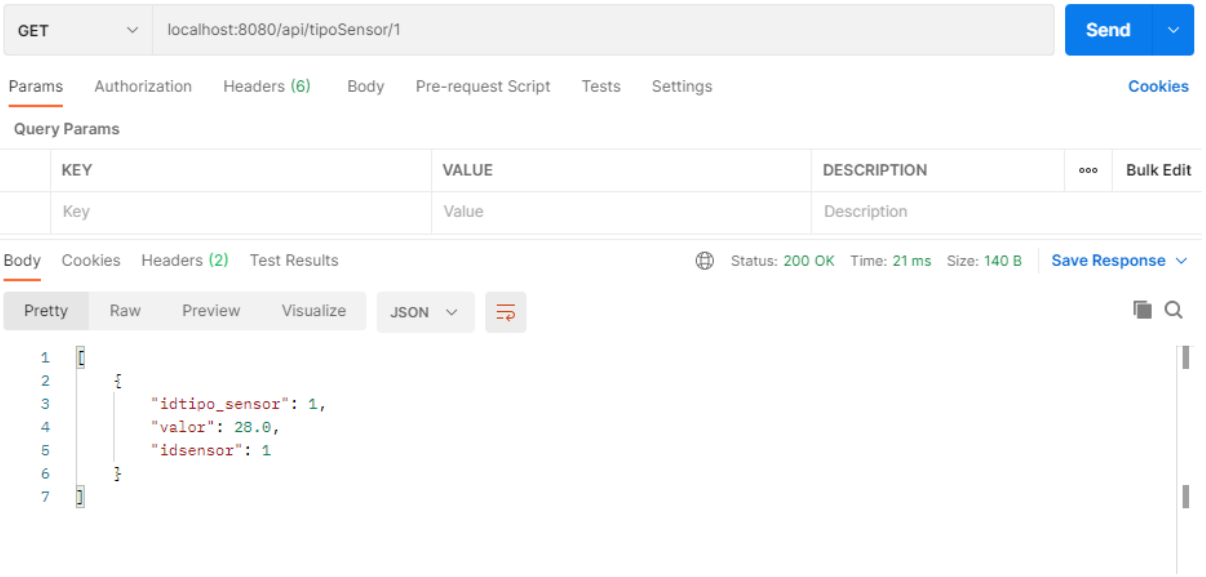


- this::getInfoSensor: este método te devuelve el tipo de sensor estamos dándole como parámetro.
  URL introducida: "/api/InfoSensor/:idsensor"

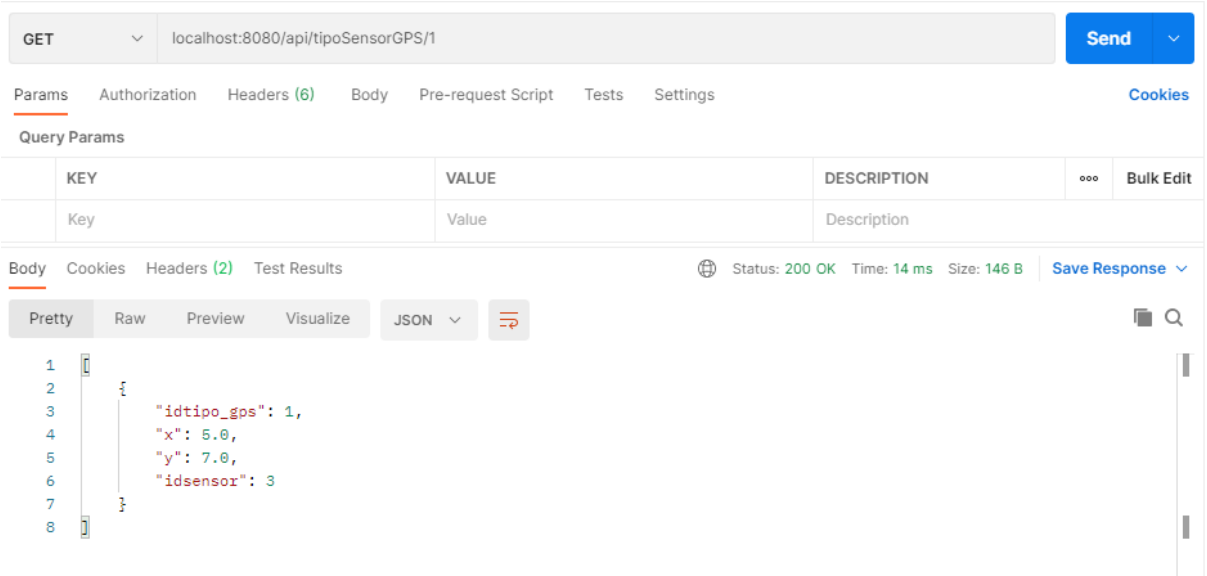


-

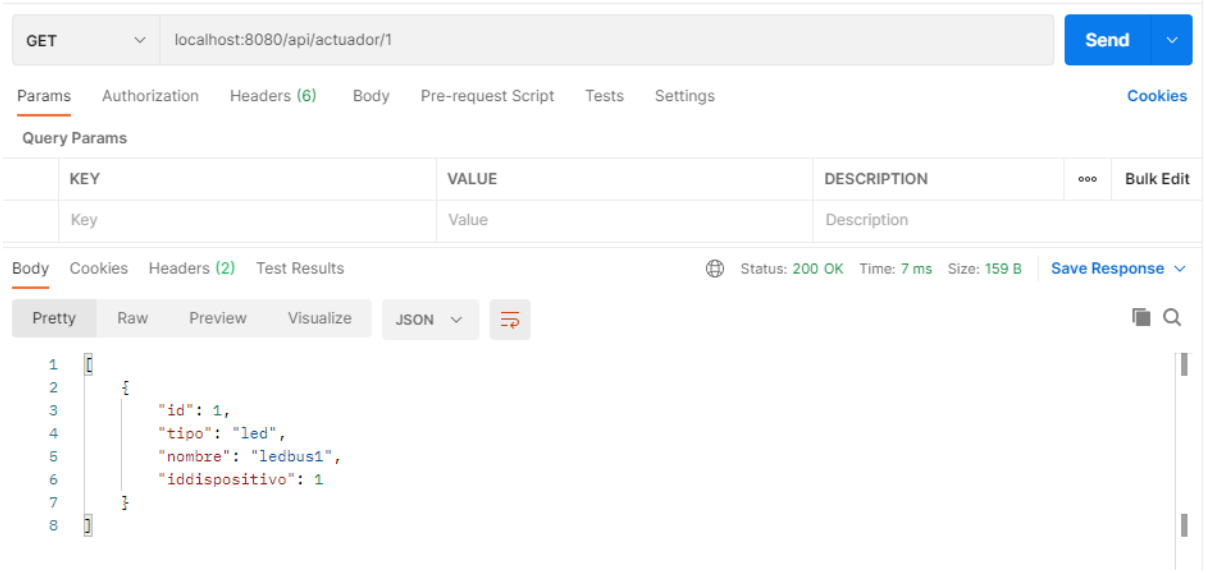
- this::getDataSensor: dado un tiempo determinado devuelve su información actual asociada al intervalo de tiempo.
  URL introducida: "/api/DataSensor/:timestamp"



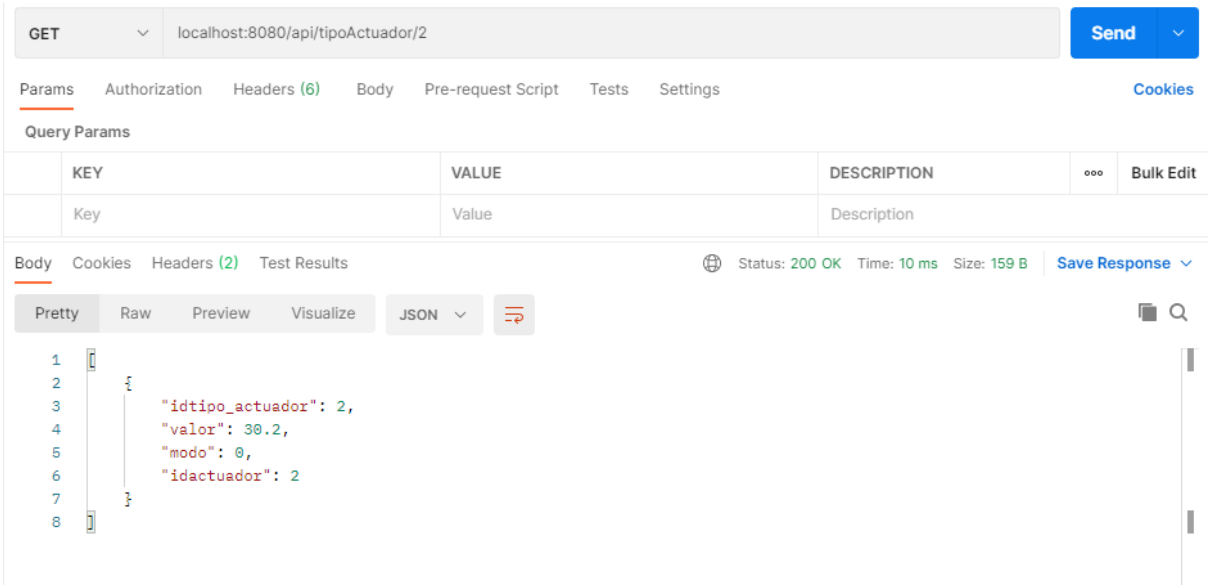
- this::getSensorGPS este método nos devolverá las coordenadas GPS actuales de nuestros dispositivos
  URL introducida: "/api/tipoSensorGPS/:idtipo_gps"



- this::getActuador información respecto a los diferentes actuadores que puedan haber en el proyecto.
  URL introducida: "/api/actuador/:idactuador"



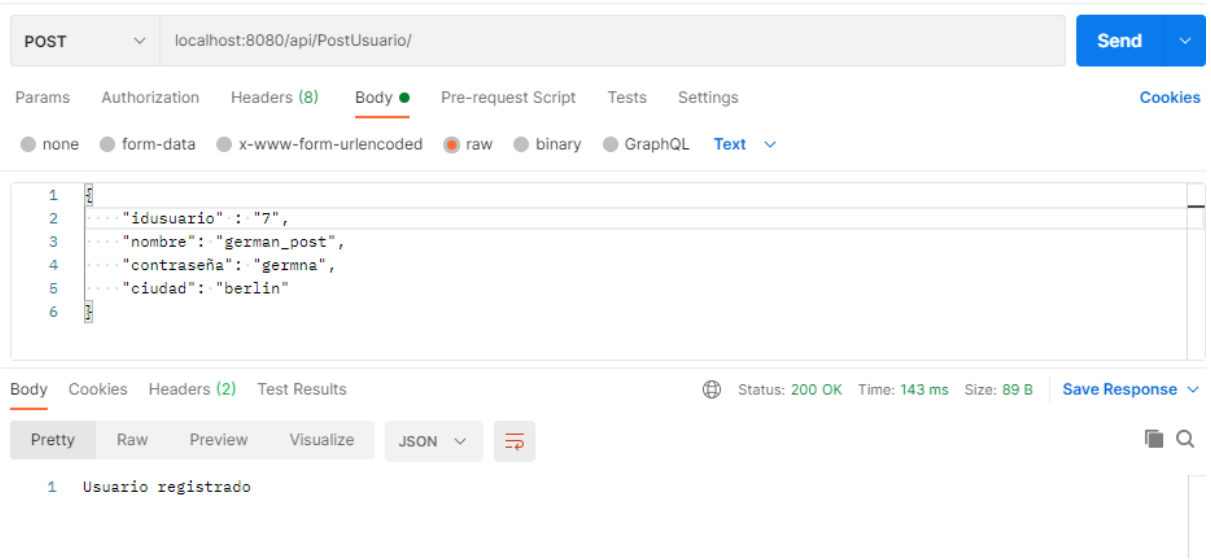
- this::getTipoActuador: información asociada a cada dispositivo actuador.
  URL introducida: "/api/tipoActuador/:idtipo_actuador"



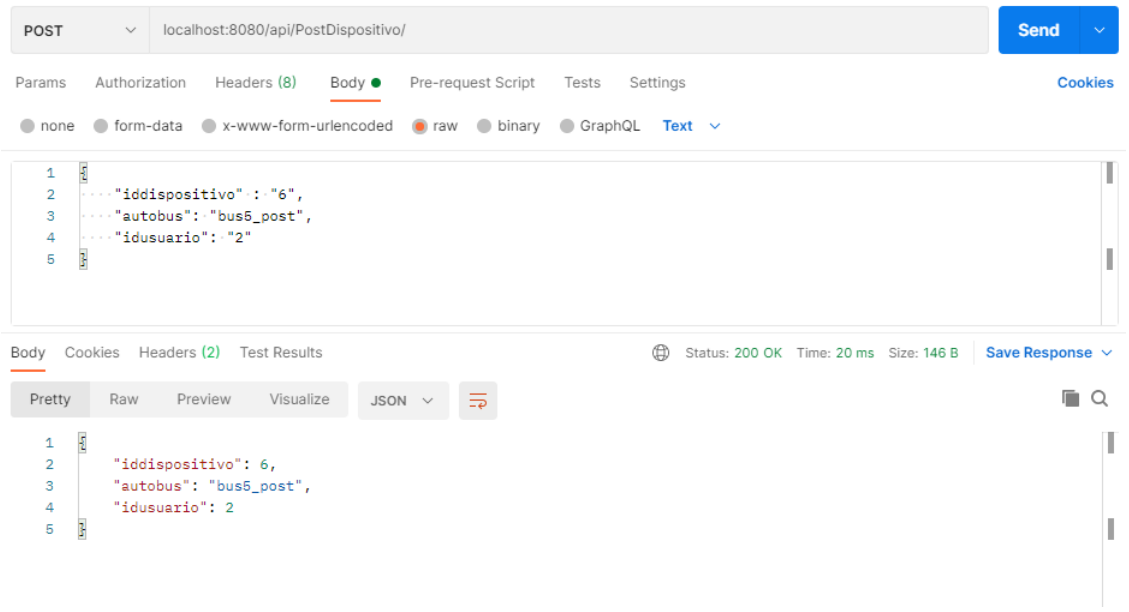
- **POST** : Usados para incluir nuevas entidades a la base de datos.

  Tenemos varios métodos POST, uno para incluir a los usuarios, otro para incluir a los dispositivos (un dispositivo por autobús) , uno para incluir información de un sensor, otro para incluir información del sensor a cada cierto tiempo y para incluir un actuador.

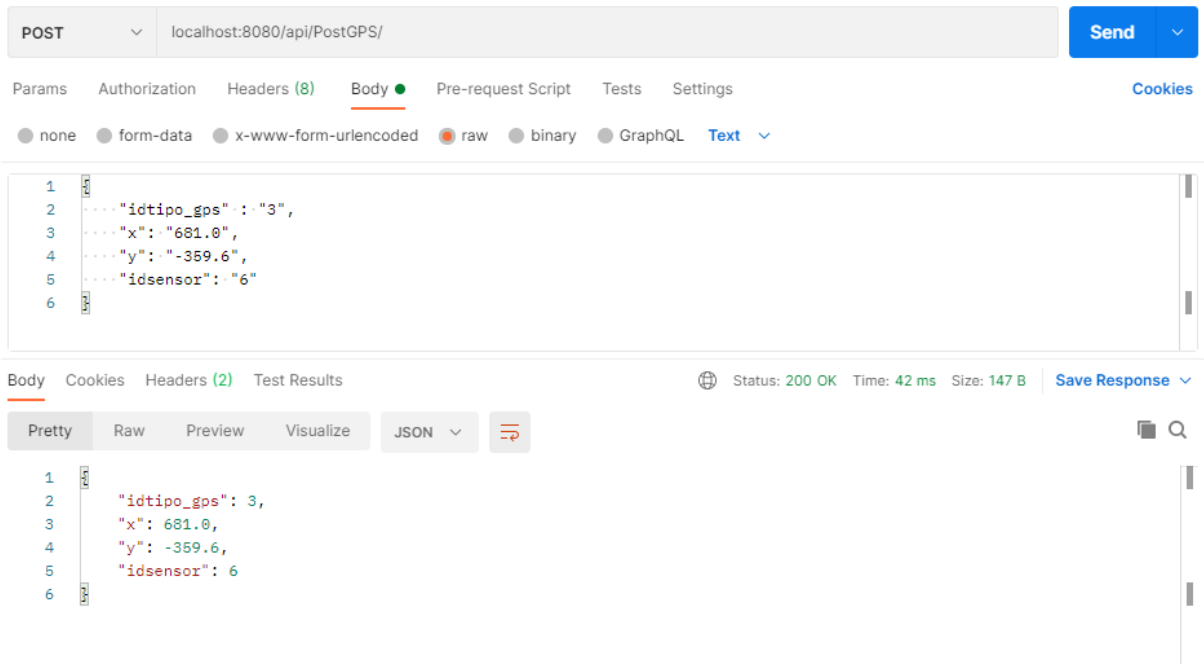
  - this::postUsuario
    URL introducida: "/api/PostUsuario/"



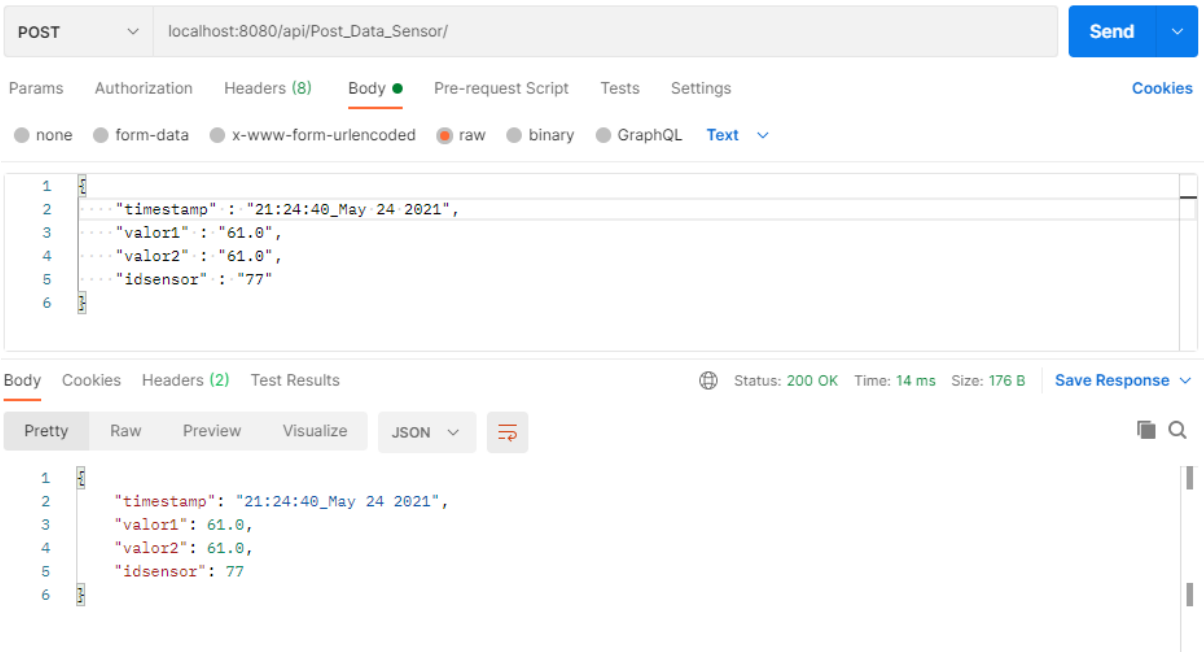
  - this::postDispositivo
    URL introducida: "/api/PostDispositivo/"



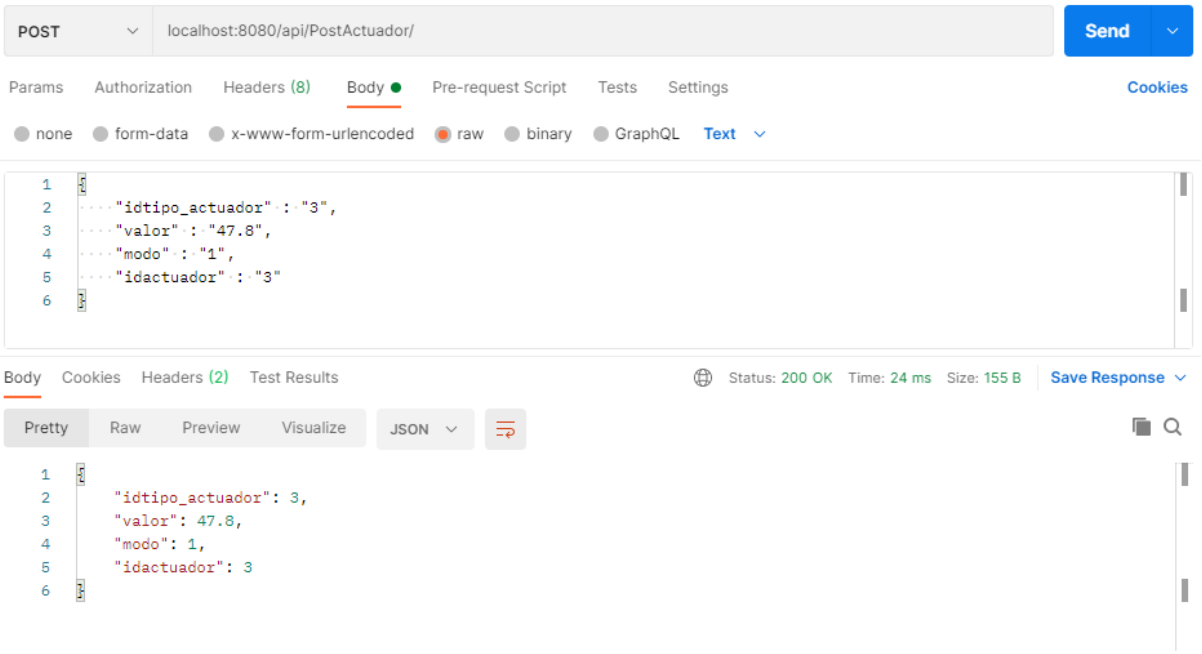
- this::postTipo_GPS
  URL introducida: "/api/PostGPS/"



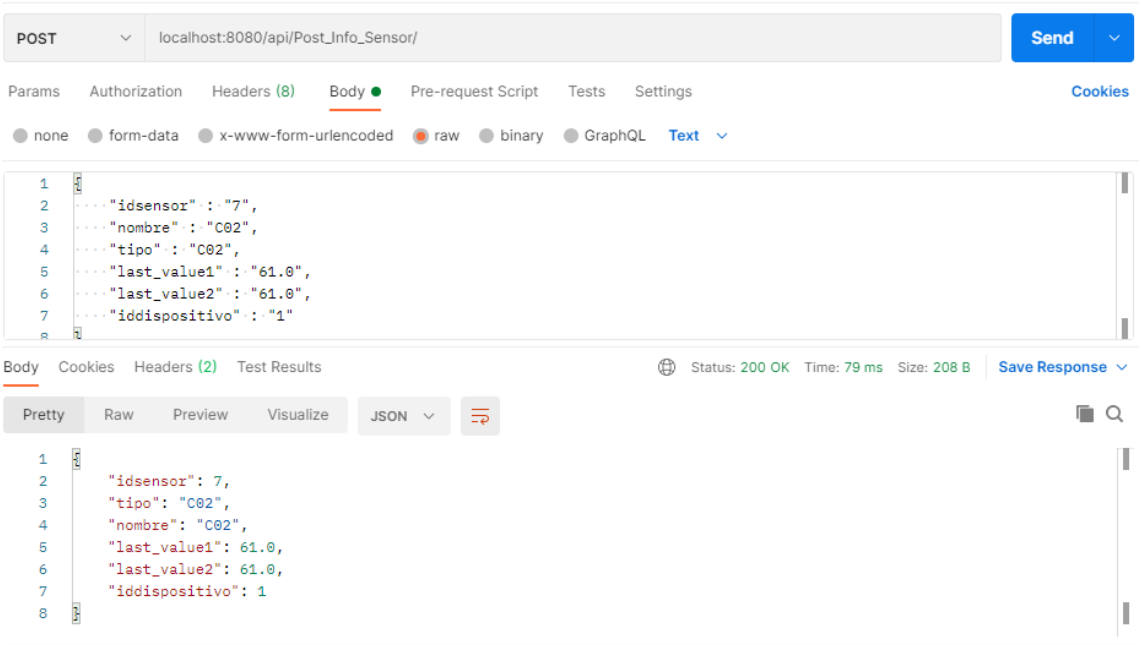
- this::postData_Sensor
  URL introducida: "/api/Post_Data_Sensor/"



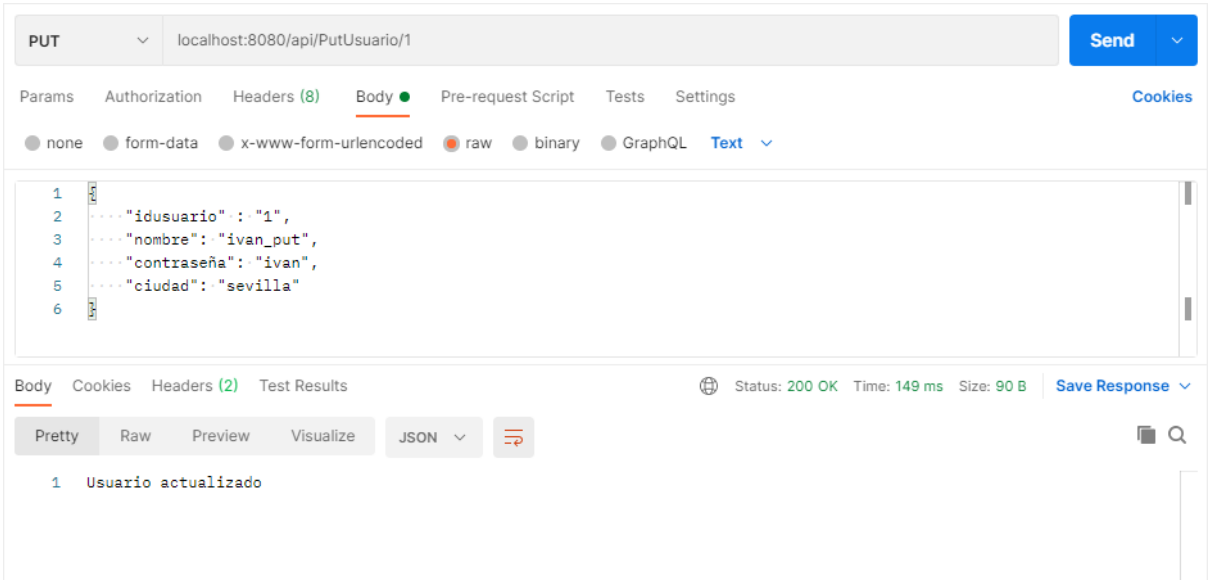
- this::postTipo_Actuador
  URL introducida: "/api/PostActuador/"



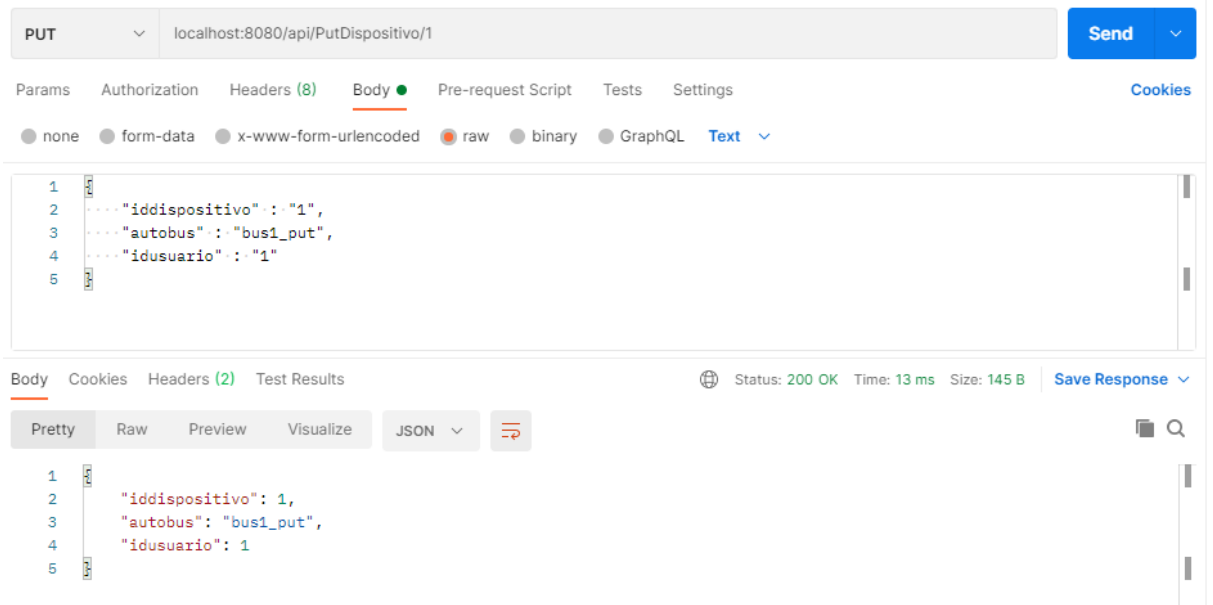
- this::postInfo_Sensor
  URL introducida: "/api/Post_Info_Sensor/"



- **PUT :** Método usado para actualizar un recurso en el servidor, de igual manera al POST, actualización en caso de dispositivo o usuario y así mismo de la información que producen los sensores .
    - this::PutUsuario
      URL introducida: "/api/PutUsuario/:idusuario"

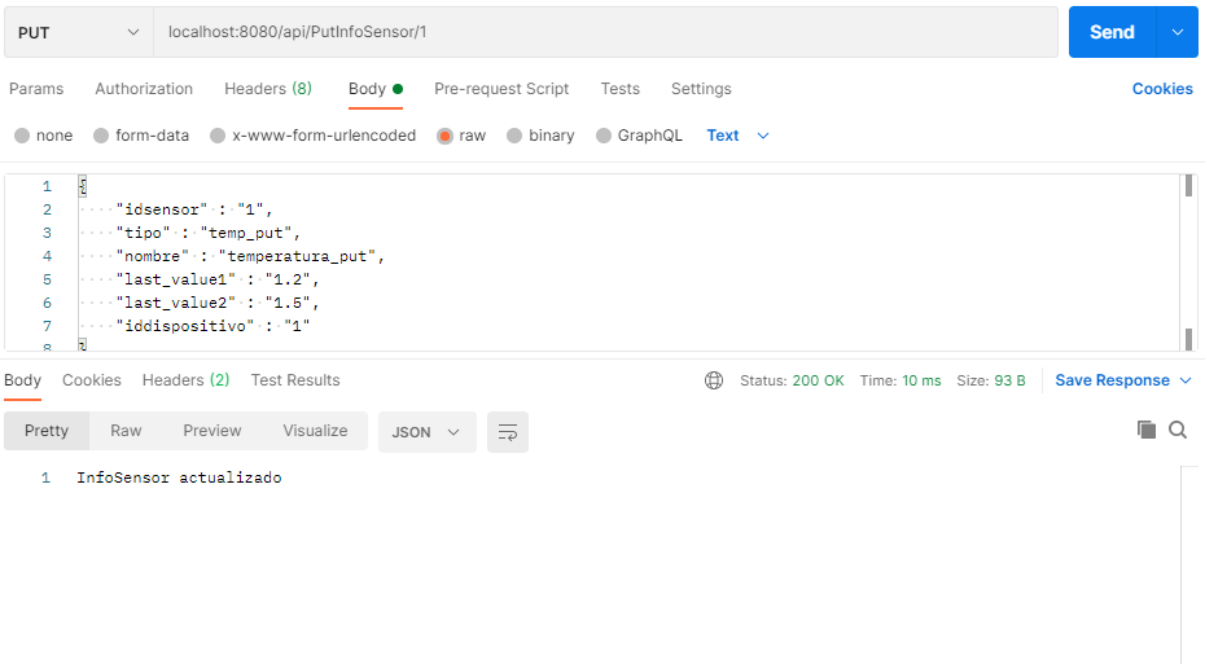


    - this::PutDispositivo
      URL introducida: "/api/PutDispositivo/:iddispositivo"

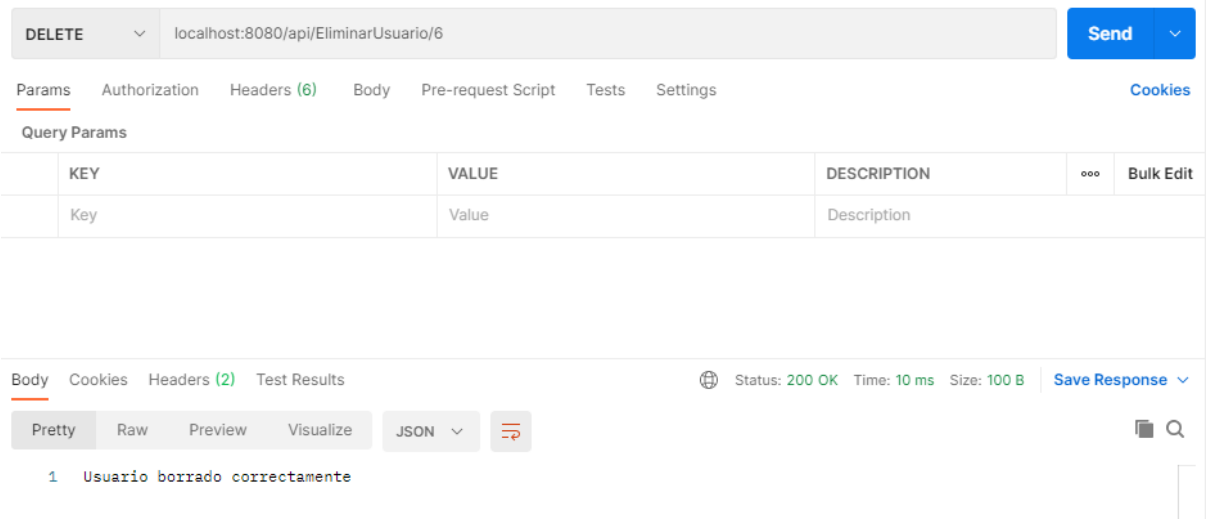


- this::PutInfoSensor

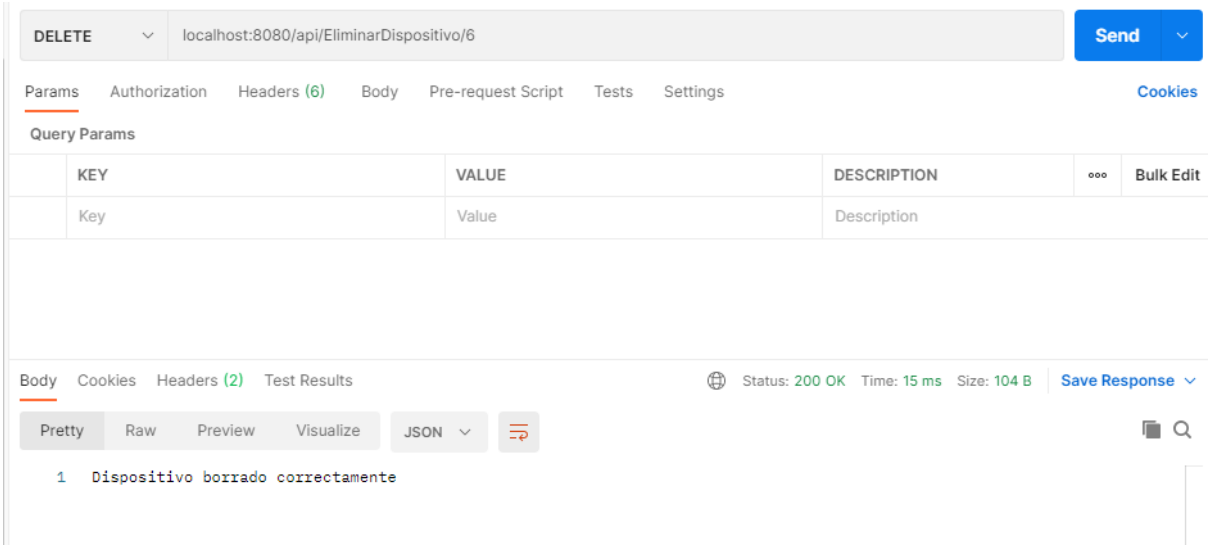
  URL introducida: "/api/PutInfoSensor/:idsensor"



- **DELETE :** Método DELETE para eliminar a una entidad o recurso, de igual forma, hemos introducido este método para los usuarios y los dispositivos.
  - this::DeleteUsuario
    URL introducida: "/api/EliminarUsuario/:idusuario"



- this::DeleteDispositivo
  URL introducida: "/api/PutDispositivo/:iddispositivo"



- Código Eclipse:

- Clase main:

Desde nuestro Verticle main, llamamos con el método vertx.DeployVerticle() a una clase llamada ApiRest donde creamos y aplicamos los métodos Rest anteriormente explicados.

```
package vertx;

import io.vertx.core.AbstractVerticle;

import io.vertx.core.Future;

public class Verticle extends AbstractVerticle{

	@Override

	public void start(Future<Void> startFuture) {

		vertx.createHttpServer().requestHandler(

				request ->{

					request.response().end("hola colega");
// gestiona una peticion, enviando un codigo en este caso no es nada

		}).listen(8082, result->{

			if(result.succeeded()) {

				System.out.println("Todo correcto");

			}else {

				System.out.println(result.cause());

			}

		});

		vertx.deployVerticle(ApiRest.class.getName());

	}

}
```

- Clase ApiRest:

A continuación mostraremos la clase Start que usamos para conectamos con vertx a través del parámetro Promise<Void>

- Conexión base de datos:

```
MySQLConnectOptions mySQLConnectOptions = new
MySQLConnectOptions().setPort(3306).setHost("localhost")


.setDatabase("covidbus").setUser("root").setPassword("ivan1998");

		PoolOptions poolOptions = new PoolOptions().setMaxSize(5);// numero
maximo de conexiones

		mySqlClient = MySQLPool.pool(vertx, mySQLConnectOptions,
poolOptions);



		Router router = Router.router(vertx); // Permite canalizar las peticiones

		router.route().handler(BodyHandler.create());

		//Creacion de un servidor http, recibe por parametro el puerto, el
resultado

		vertx.createHttpServer().requestHandler(router::handle).listen(8080,
result -> {

			if (result.succeeded()) {

				startPromise.complete();

			}else {

				startPromise.fail(result.cause());

			}

		});
```

- **Llamada a los métodos Post anteriormente mencionados:**

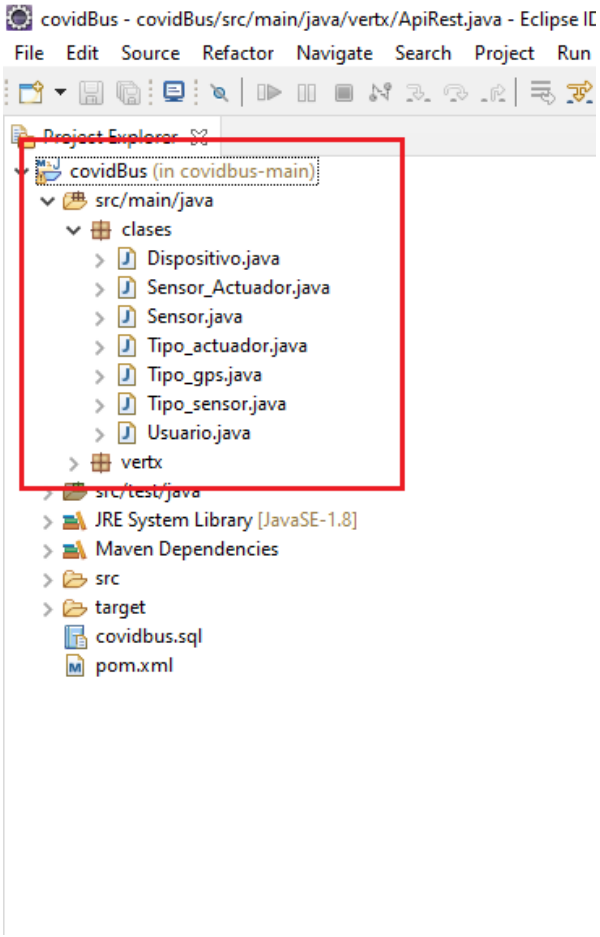
```
router.get("/api/usuario/:idusuario").handler(this::getUsuario);

router.put("/api/PutUsuario/:idusuario").handler(this::PutUsuario);

router.delete("/api/EliminarUsuario/:idusuario").handler(this::DeleteUsuario);

router.post("/api/PostUsuario/").handler(this::postUsuario);

router.get("/api/dispositivo/:iddispositivo").handler(this::getDipositivo);

router.get("/api/dispositivosUsuarios/").handler(this::getDipositivosUsuarios);

router.get("/api/dispositivosUsuarios/:idusuario").handler(this::getDipositivoUsuario);

router.put("/api/PutDispositivo/:iddispositivo").handler(this::PutDispositivo);

router.delete("/api/EliminarDispositivo/:iddispositivo").handler(this::DeleteDispositivo)

router.post("/api/PostDispositivo/").handler(this::postDispositivo);

router.get("/api/sensor/:idsensor").handler(this::getSensor);


router.get("/api/tipoSensor/:idtipo_sensor").handler(this::getTipoSensor);


router.get("/api/tipoSensorGPS/:idtipo_gps").handler(this::getSensorGPS);

router.post("/api/PostGPS/").handler(this::postTipo_GPS);

router.post("/api/PostSensor/").handler(this::postTipo_Sensor);

router.get("/api/actuador/:idactuador").handler(this::getActuador);

router.get("/api/tipoActuador/:idtipo_actuador").handler(this::getTipoActuador);

router.post("/api/PostActuador/").handler(this::postTipo_Actuador);
```

- Ejemplo de llamada a this::getUsuario:

(que definimos en la misma clase ApiRest)

```
private void getUsuario(RoutingContext routingContext) {

// routing da un contenido en formato string por lo que hay que parsearlo

Integer idusuario=Integer.parseInt(routingContext.request().getParam("idusuario"));


mySqlClient.query("SELECT * FROM covidbus.usuario WHERE idusuario = '" +
idusuario + "'", res -> {

        if (res.succeeded()) {

                RowSet<Row> resultSet = res.result();

                JsonArray result = new JsonArray();

                for (Row elem : resultSet) {

                result.add(JsonObject.mapFrom(new
Usuario(elem.getInteger("idusuario"),

                elem.getString("nombre"),

                elem.getString("contraseña"),

                elem.getString("ciudad"))));

                }

routingContext.response().putHeader("content-type",
"application/json").setStatusCode(200).end(result.encodePrettily());

System.out.println(result.encodePrettily());

}else {

routingContext.response().putHeader("content-type",
"application/json").setStatusCode(401)
.end((JsonObject.mapFrom(res.cause()).encodePrettily()));
System.out.println("Error"+res.cause().getLocalizedMessage());

                        }

                });
```

- **Clases creadas para conectar con el servidor, contenidas en el paquete <u>clases.</u>**



No pegamos el código para no hacer más emborrosa la entrega, se encuentran en el zip en **src → main → java → clases**

- Descripción de los mensajes MQTT intercambiados entre los clientes

Hasta el momento tenemos incorporado un único topic llamado **parpadea_led.**

Este lo usaremos en la posterior entrega, en la página web, donde los usuarios podrán pulsar un botón virtual, y este le indicará al ESP8266 que alguien desea bajarse del autobús.

Funciona correctamente la transmisión bidireccional de datos con el broker, para la posterior entrega nos gustaría añadir actuadores relacionados con los parámetros de los sensores