Documentación del Sprint 0

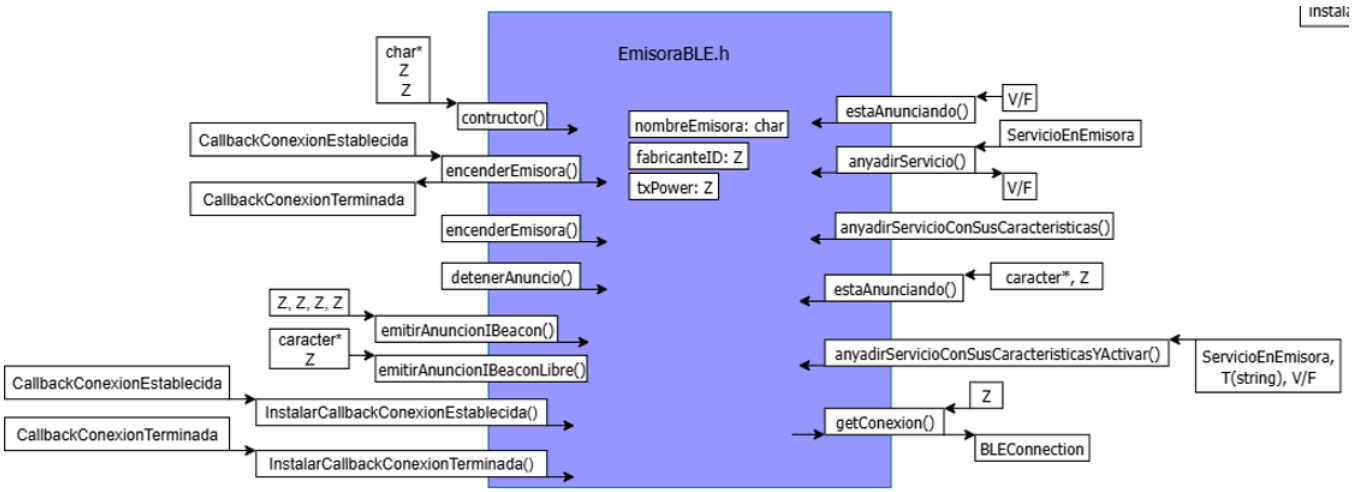
Índice

Realizado por: Marc Vilagrosa Caturla

Documentación Arduino

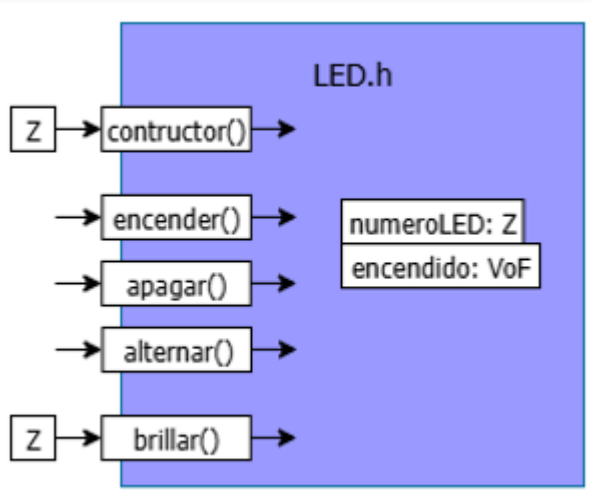
1. Emisorable.h

EmisoraBLE es una clase ligera que envuelve la pila Bluefruit para gestionar la publicidad BLE: inicia o detiene la radio (no se enciende en el constructor, se hace con encenderEmisora), comprueba si está anunciando y publica en formato iBeacon (UUID, major, minor, rssi/txPower) o con una carga libre de 21 bytes.



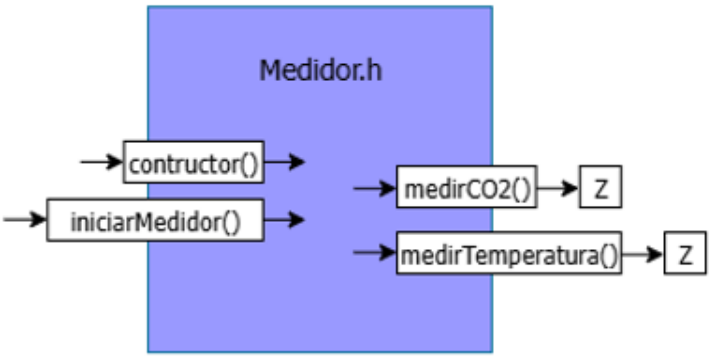
2. LED.h

LED encapsula el control de un LED en un pin digital: el constructor recibe el número de pin, lo configura como salida y deja el LED apagado. Ofrece métodos para encender, apagar y alternar el estado, manteniendo una bandera interna encendido.



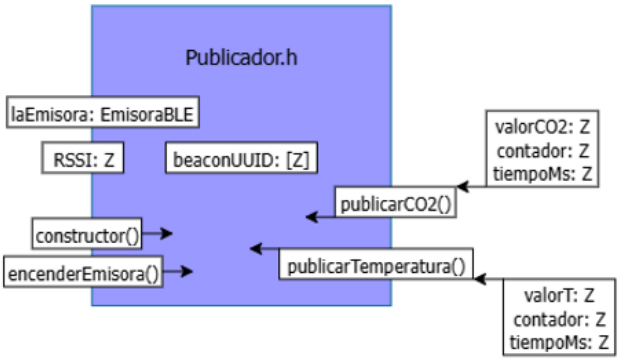
3. Medidor.h

Medidor es una clase mínima que abstrae la lectura de sensores: expone medirCO2() y medirTemperatura(), que devuelven enteros (por ahora valores de prueba) como punto de integración con el sensor real.



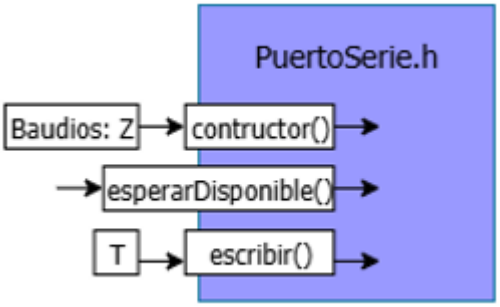
4. Publicador.h

Publicador orquesta la emisión de medidas vía iBeacon. Mantiene el beaconUUID y una EmisoraBLE (nombre, fabricante, txPower), y ofrece encenderEmisora(). Sus métodos publicarCO2(...) y publicarTemperatura(...) codifican el tipo de medida y un contador en major, el valor en minor, inician la publicidad el tiempo indicado y luego la detienen.



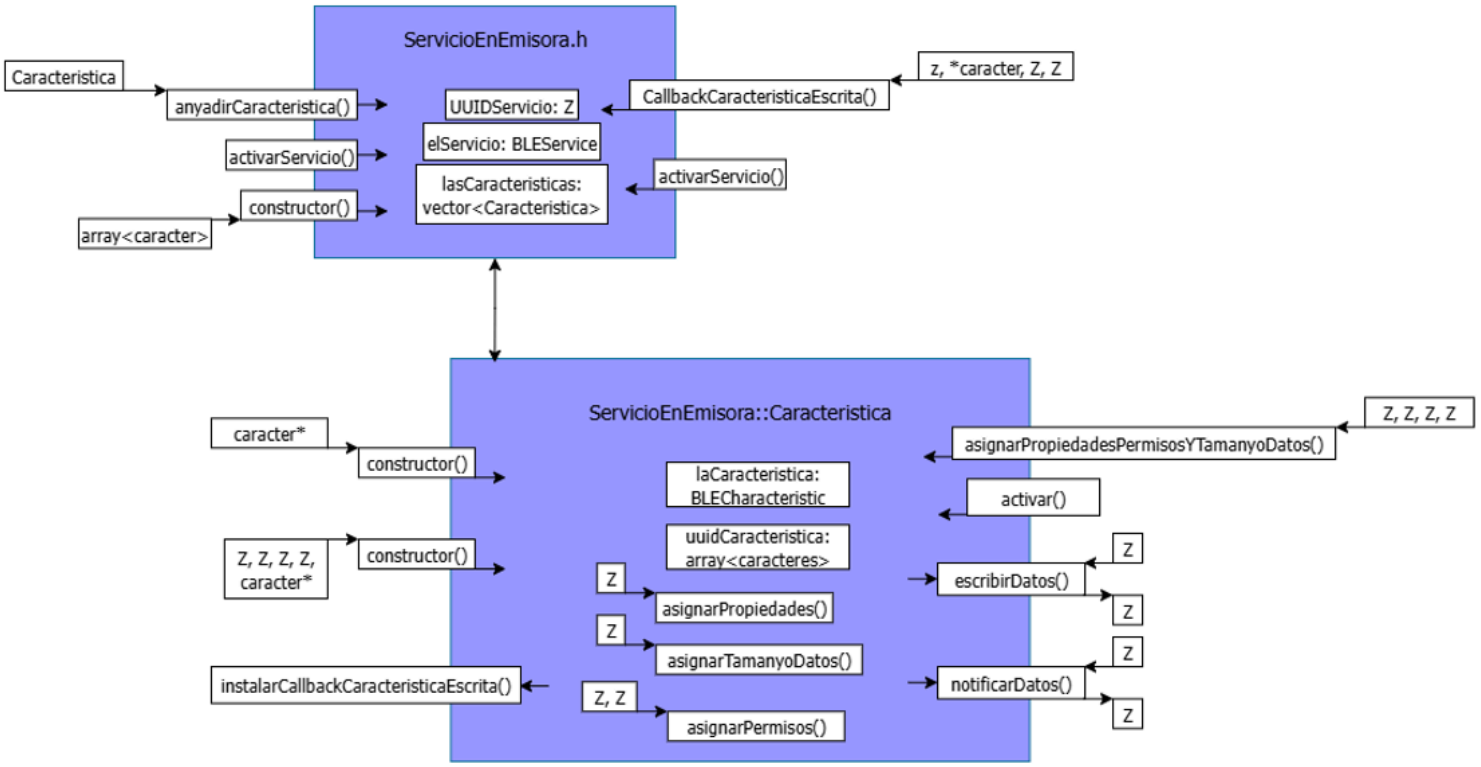
5. PuertoSerie.h

PuertoSerie: se inicializa con los baudios en el constructor, ofrece esperarDisponible() para bloquear hasta que el puerto esté listo y un método genérico escribir(mensaje) que imprime cualquier tipo compatible con Serial.print.



6. ServicioEnEmisora.h

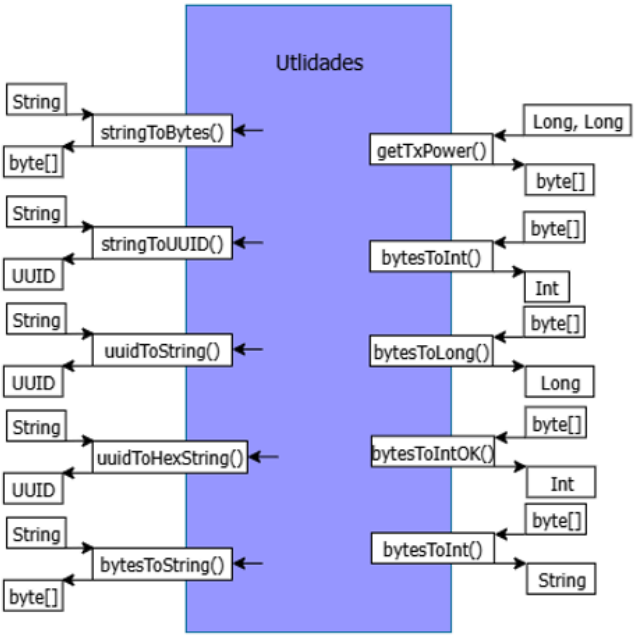
ServicioEnEmisora crea un servicio BLE (GATT) desde un UUID y permite añadirle características. Puedes activarlo, enviar datos para que el móvil los lea o notificar cambios. Además, puedes definir una función que se ejecuta cuando el móvil escribe en una característica concreta (por ejemplo, si la app cambia un ajuste). Se integra directo en la emisora.



Documentación Android

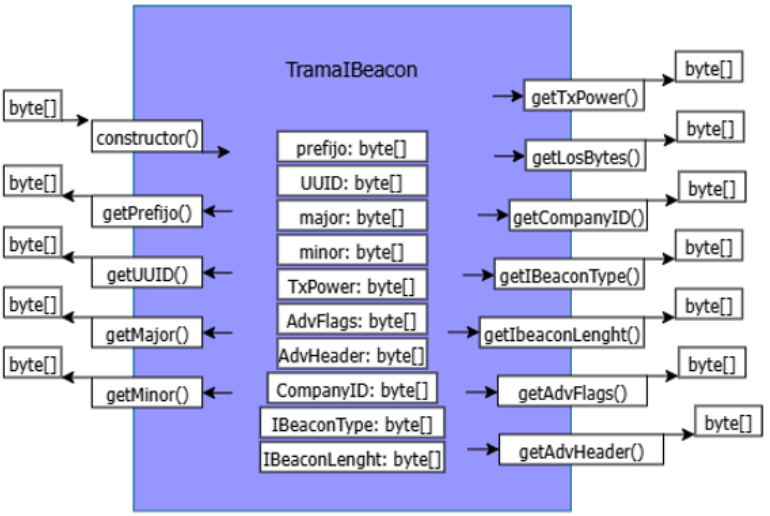
1. Utilidades

Utilidades.java es una caja de herramientas para manejar datos de Bluetooth: convierte entre String, byte[], int, long y UUID, formatea UUID a texto y ofrece funciones para pasar arrays de bytes a números cuidando orden y signo. Incluye ayudas para representar/extraer el txPower. Se usa en el parser de tramas y en la lógica para mostrar, construir y procesar valores de iBeacon de forma consistente.



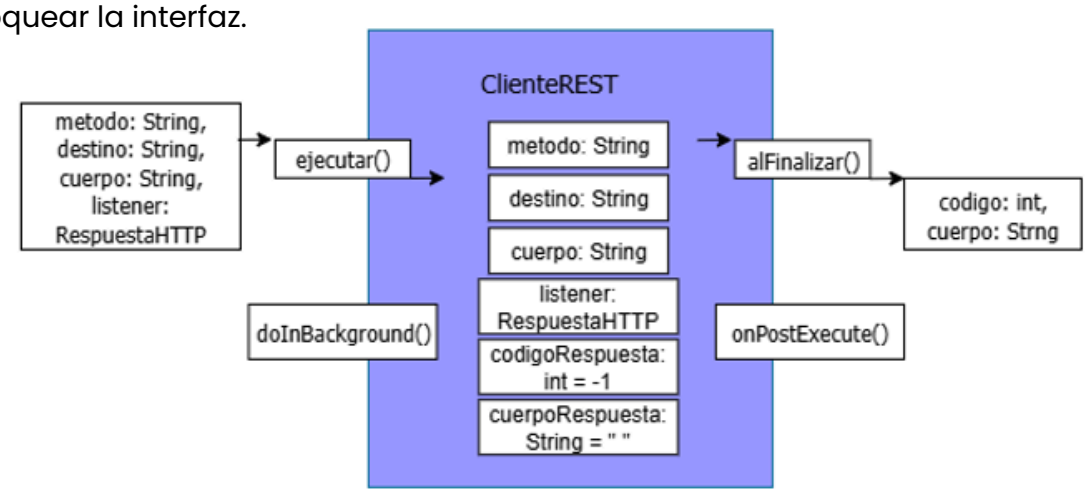
2. TramaIBeacon

TramaIBeacon recibe el byte[] de advertising y lo despieza en los campos de iBeacon: prefijo (flags, header, companyID, tipo y longitud), UUID (16 B), major, minor y txPower. Se usa tras el escaneo para comprobar la trama y extraer la información del beacon de forma segura y consistente.



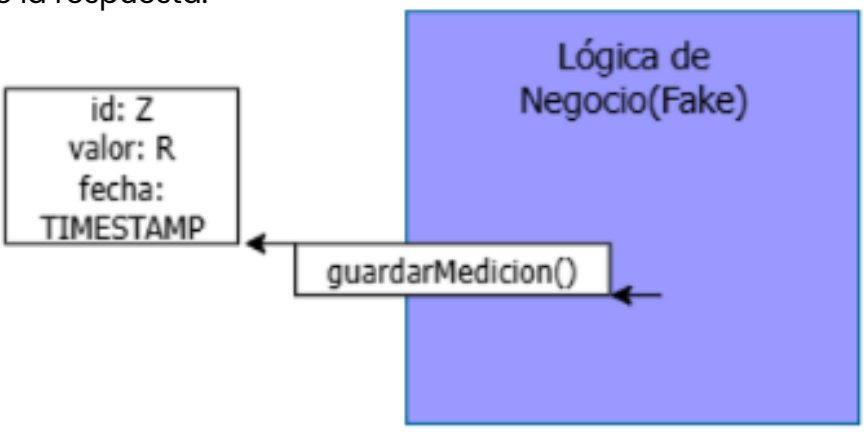
3. ClienteREST

ClienteREST lanza peticiones HTTP simples en segundo plano. Con ejecutar(método, destino, cuerpo, listener) configuras el verbo (GET/POST…), la URL y, opcionalmente, un JSON. La clase abre la conexión, envía el payload si procede, lee el código HTTP y el cuerpo de respuesta, y entrega ambos al hilo principal mediante el callback RespuestaHTTP.alFinalizar(código, cuerpo). Ante errores de red devuelve código -1 y cuerpo vacío. Ideal para hablar con la API sin bloquear la interfaz.



4. LogicaFake

LogicaFake es la capa mínima de negocio que publica una medición en el servidor: ofrece guardarMedicion(int/double), construye el JSON {"valor":<num>} y lo envía por POST a la URL configurada (URL_SERVIDOR), delegando la petición en ClienteREST para ejecutarla en segundo plano. Solo se manda el campo valor; el backend genera el id y la fecha. Al completar, registra en Logcat el código y el cuerpo de la respuesta.

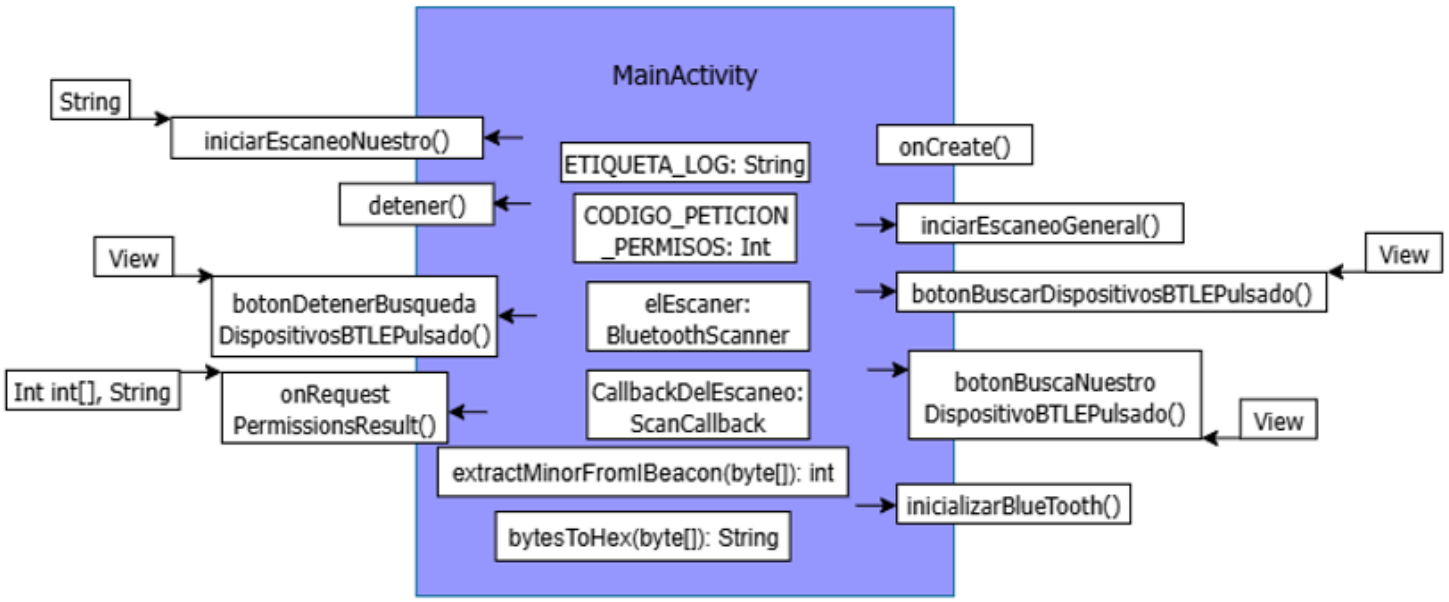


5. MainActivity

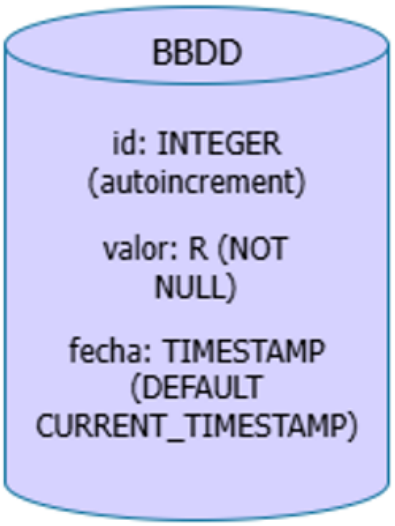
MainActivity coordina el escaneo BLE y el envío automático de la medición. Al iniciar, prepara Bluetooth y permisos; ofrece tres acciones:

- Escaneo general (para depurar).
- Escaneo “nuestro” (filtra por nombre "Grupo4")
- Detener.

En el modo “nuestro” valida que la trama sea iBeacon Apple (4C 00 02 15), extrae el MINOR en big-endian, lo interpreta como int16 con signo y lo manda a LogicaFake.guardarMedicion(), deteniendo el escaneo tras la primera lectura para evitar duplicados (bandera enviado). Usa ScanSettings en baja latencia, gestiona permisos (BT_SCAN/CONNECT o FINE_LOCATION según versión) y proporciona utilidades de log y bytesToHex.



Documentación BBDD



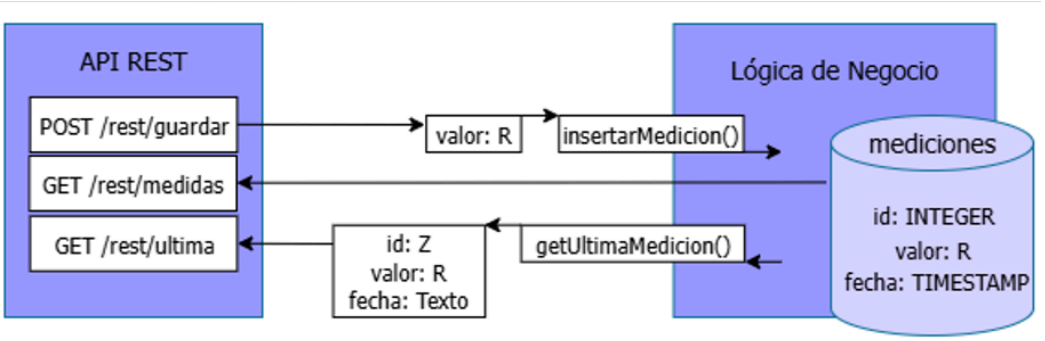
La BBDD almacena las mediciones en una única tabla con tres campos: **id** (entero autoincremental que actúa como clave primaria), **valor** (número real obligatorio que representa la medición) y **fecha** (timestamp automático que se asigna en el momento de insertar el registro). No guarda más información: cada fila es una medición con su identificador y la hora a la que se registró.

Flujo API-Lógica-BBDD

El flujo es directo y en dos sentidos:

Alta: el cliente hace POST /api/v1/mediciones con {"valor": n}; la API valida el cuerpo y delega en insertar_medicion(valor), que escribe la fila en la tabla (el id se autoincrementa y la fecha se sella con CURRENT_TIMESTAMP). Después la API llama a get_ultima_medicion() para recuperar lo insertado y devuelve {id, valor, fecha} (fecha normalizada a UTC).

Consulta: para leer, GET /api/v1/mediciones/ultima usa get_ultima_medicion() y GET /api/v1/mediciones?limit=k usa get_mediciones(limit) la cual; ambos métodos acceden a la BD y la API transforma las filas a JSON. Así, la capa HTTP no toca SQL: solo orquesta entrada/salida, mientras la lógica realiza las operaciones sobre la BBDD.



Al llamar GET /api/v1/mediciones?limit=k, la API toma k (50 por defecto) y pide a la lógica las filas más recientes. Luego convierte esas filas a JSON, normaliza la fecha a UTC y responde una lista con {id, valor, fecha}.