Tower Control

Torre de control automática

Felipe Patricio Artengo Aguado, Francisco Javier Díaz Ventura

Índice

Introducción

Año 1977, Tenerife, Aeropuerto de Los Rodeos. La colisión ocurrió cuando el avión de KLM inició su carrera de despegue mientras el avión de Pan Am, envuelto en la niebla, todavía estaba en la pista y a punto de salir a la calle de rodaje. Al descubrirlo, el avión de KLM intentó elevarse para sobrepasar al avión de Pan Am y casi lo consiguió, pero acabó embistiéndolo. Del choque resultante fallecieron todas las personas a bordo del KLM 4805 y a la mayoría de los ocupantes del Pan Am 1736, con solo 61 sobrevivientes en la parte delantera del avión. Después de muchas investigaciones por parte de las autoridades españolas se descubrió que el error se dio debido a una falta de entendimiento entre piloto - torre de control. El piloto que estaba a punto de despegar entendió mal la señal que le dieron desde la torre de control y ocurrió el desastre.

Con nuestro trabajo queremos poder avisar tanto a pilotos como controladores aéreos para facilitarles su trabajo para que todo este automatizado y minimizar el error humano.

El producto que diseñamos va destinado principalmente a los controladores de torres de control y a los pilotos. Pero también podríamos aplicarlo a la vida cotidiana como pudiesen ser dos coches que vayan por control autónomo para poder guiarlos sin que se coquen con los distintos obstáculos.

Funcionalidades

- **Localizador**

  Mediante un módulo GPS, podremos obtener la latitud, longitud, dirección que esta tomando el avión (0º - 359º), velocidad, altitud y la fecha de la consulta. Esta información se enviará constantemente al servidor.

- **Luz**

  La base de datos cuando reciba un dato enviará una petición para encender el led amarillo cuando este cerca de otro avión (4 km). En el caso del led rojo-blanco si se encendiese, estaríamos hablando que el avión esta a una altura menor que 500 metros.

Base de datos

La base de datos diseñada para nuestro proyecto es relativamente simple, puesto que la única información que vamos a almacenar será la de la posición del avión, la posición de los distintos aeropuertos y el itinerario de cada vuelo. Para ello hemos creado una tabla para cada uno. El resultado es el siguiente diagrama UML:

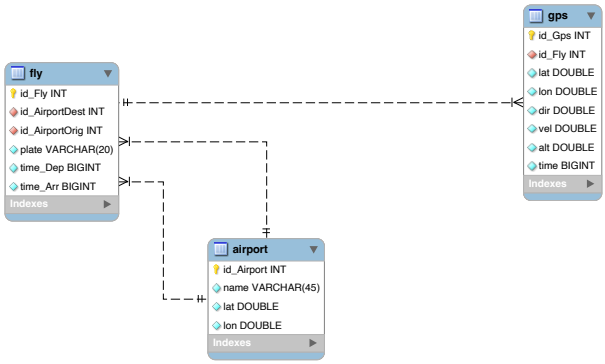


Diagrama UML

Para todas las tablas hemos usado los atributos básicos con el objetivo de obtener una base de datos simple y que no nos lleve una gran cantidad de tiempo trabajar con ella.

API REST

Hemos creado funciones básicas que nos conecten con la base de datos. Para ver si nuestras consultas se realizan correctamente vamos a tener un código 200(todo ha ido bien) y el error 400 (algo ha dio mal).Nuestra API REST está formada por:

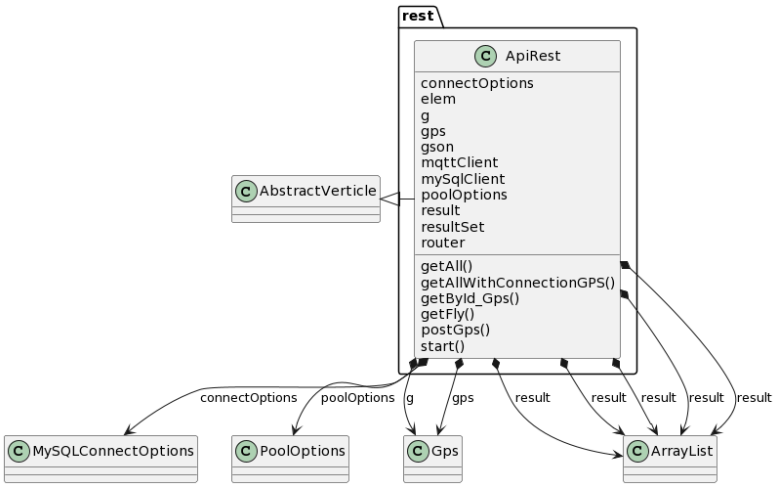


Diagrama de clases de la API

- **Métodos GET:**

  A la hora de hacer los *getAllConnection* lo hemos limitado a 5 entradas para facilitar la lectura. Hemos considerado añadir seis GET's y son los siguientes:

  El método *getAllWithConnectionGPS* realizará una consulta a la base de datos, devolviendo todas las entradas de la tabla *Gps.*

  El método *getById_Gps* va a realizar una consulta a la base de datos a la tabla *Gps*, devolviendo la entrada que concuerde con el id que le digamos.

  El método *getAllWithConnectionFly* realizará una consulta a la base de datos a la tabla *Fly*, devolviendo todas las entradas de la tabla *Fly.*

  El método *getById_Fly* va a realizar una consulta a la base de datos a la tabla *Fly*, devolviendo la entrada que concuerde con el id que le digamos.

  El método *getAllWithConnectionAirport* realizará una consulta a la base de datos, devolviendo todas las entradas de la tabla *Airport.*

  El método *getById_Airport* va a realizar una consulta a la base de datos a la tabla *Airport*, devolviendo la entrada que concuerde con el id que le digamos.

- **Método POST:**

  Para insertar los nuevos elementos hemos usado los métodos POST. El método *postGps* que va a realizar la inserción en la base de datos. En este caso vamos a leer los valores registrados por nuestro sensor GPS y los va a introducir en la base de datos. A su vez se va a realizar una consulta en la cual vamos a comprobar si los aviones en cuestión van a estar cerca el uno del otro. Si lo estuviera va a realizar un publish en el topic del broker MQTT con un 1 y si se alejasen se publicará un 0.

- **Métodos PUT:**

  Hemos considerado oportuno añadir dos métodos PUT:

  putFly y *putAirport:* los vamos a utilizar para actualizar los datos de ambas tablas *Fly y Airport*. Por ejemplo si hubiese retraso en un vuelo o un cambio de aeropuerto por alguna circunstancias podríamos modificar los datos.

Para estas peticiones hemos usado URL cortas y bastante intuitivas. Siguen la siguiente nomenclatura "/api/(tipo de elemento)/: (id)" dependiendo para que petición se va a necesitar el parámetro del id o no. En cuanto al cuerpo de las peticiones en los métodos POST y PUT, usamos un JSON para las columnas en la base de datos, es necesario que a la hora de introducir esos datos, en el cuerpo es necesario introducir todos los atributos, de lo contrario no será efectiva la petición y nos dará error.

MQTT

El uso de MQTT en nuestro proyecto es importante, ya que su uso es necesario para un sistema de actuadores eficiente. La estructura del servidor MQTT, es genérica para todos los proyectos en la totalidad del código prácticamente. La estructura de la que hablamos se trata de una clase con los métodos para iniciar e inicializar el servidor y para las acciones como clientes de conectarnos, desconectarnos, suscribirnos y darnos de baja. El único punto donde puede diferir nuestro MQTT es donde especificamos los canales, en nuestro caso los leds.

Para la implementación del cliente desde el que publicaremos, lo que hemos hecho es crear dentro de la API REST un cliente MQTT que se ha suscrito a los dos canales. Cada vez que hagamos un POST de cualquiera de los actuadores haremos un publish, por lo que guardaremos los datos en la BBDD y se actualizará en la placa.

ESP8266

- **Setup:** Inicializamos la conexión wifi y el MQTT. Obtendremos también los IDs de los sensores y los topics a los que se suscribe.

- **Loop:** En caso de no estar conectados al servidor MQTT nos volveremos a conectar y luego mediante la función *loop()* de la librería de MQTT vamos comprobando si existen nuevos datos para los sensores y actualizar sus valores.

- **Funciones MQTT:** Tenemos la función *callback,* que recibe un mensaje, este mensaje lo troceará y nos quedaremos con los datos que nos interesan. La función *reconnect,* que se conecta al servidor y suscribe a los topics.

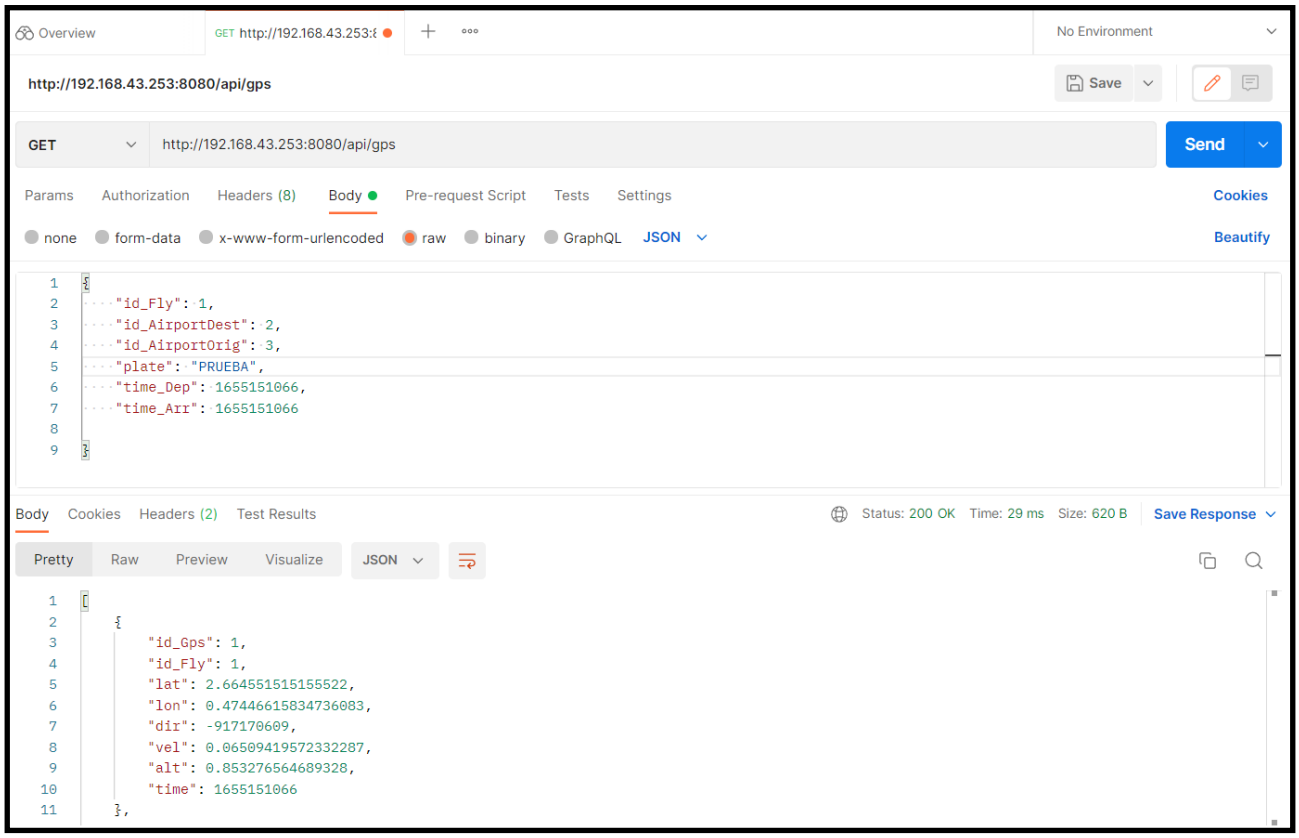
- **Funcion Actuador:** Recibe un 1 o un 0 cada vez que la placa realiza un *POST* . En caso de que la condición cambie por alguna circunstancia va a dejar el led encendido/apagado por 10 segundos. Vamos a comprobar también el dato de altura que recibe el GPS.

- **Funcion sensores:** tenemos una función la cual introduce un JSON la lectura del GPS y se lo envía la API REST a través de un POST para que este la guarde en la base de datos.

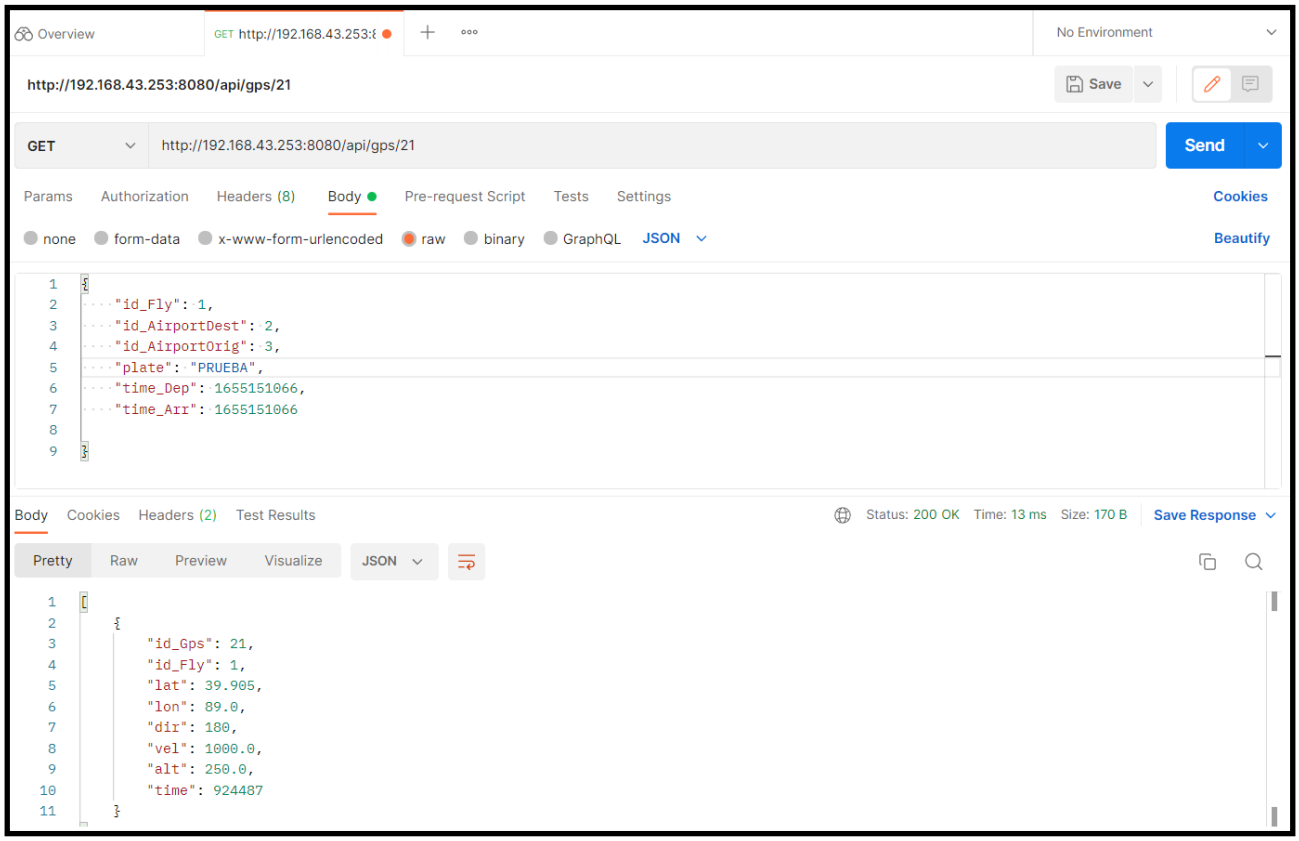
Capturas Postman

Para hacer saber si nuestras peticiones a la base de datos se están realizando correctamente hemos utilizado la aplicación Postman para probarlas:

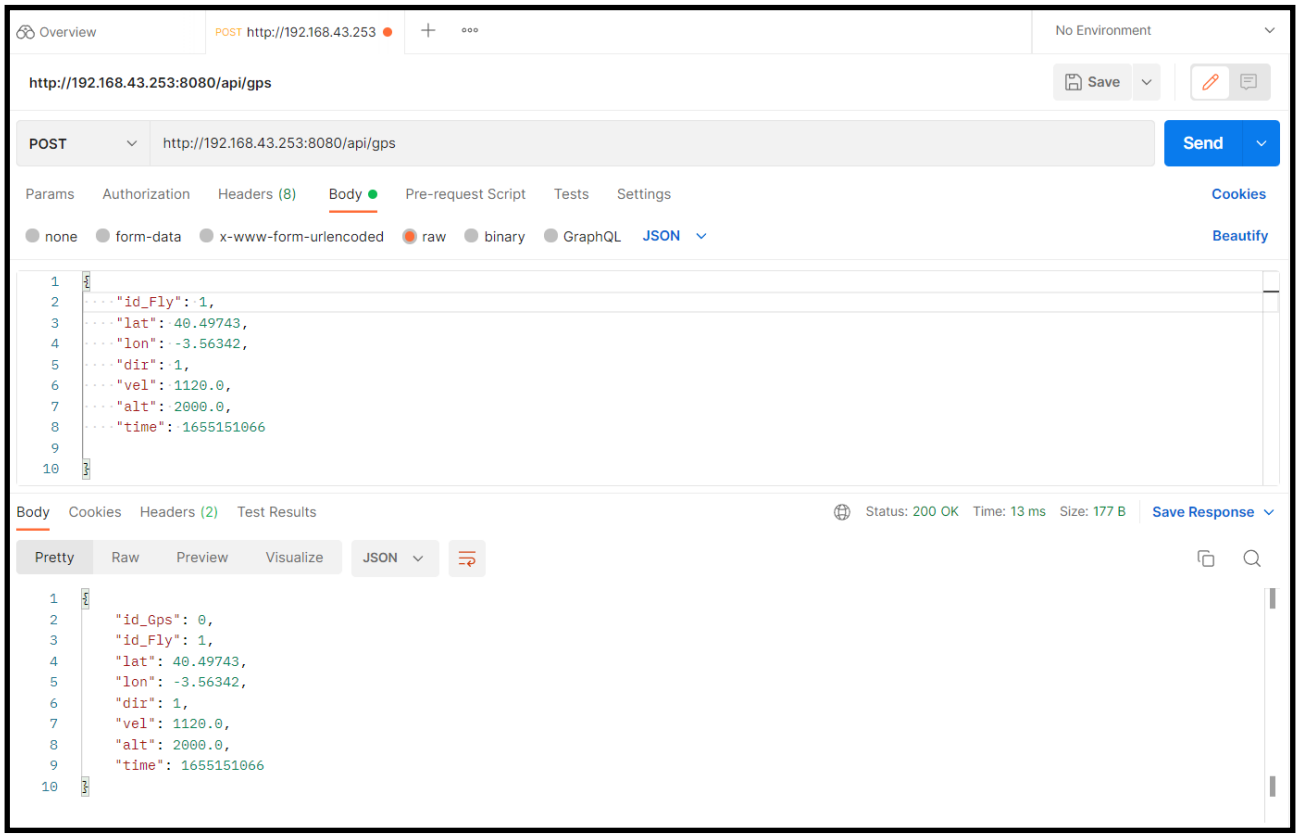
- **Peticiones GPS**



Get *getAllWithConnectionGPS*

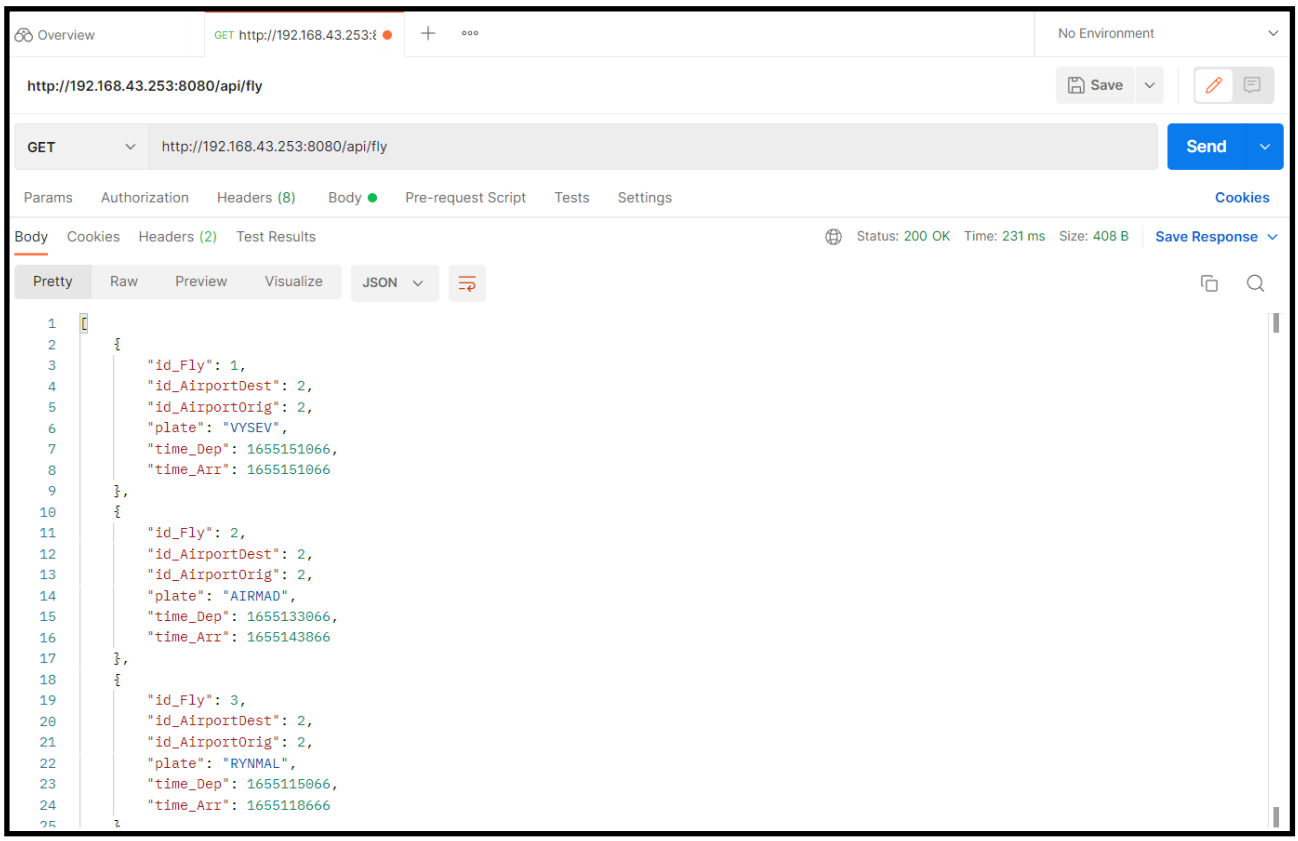


Get *getById_Gps*

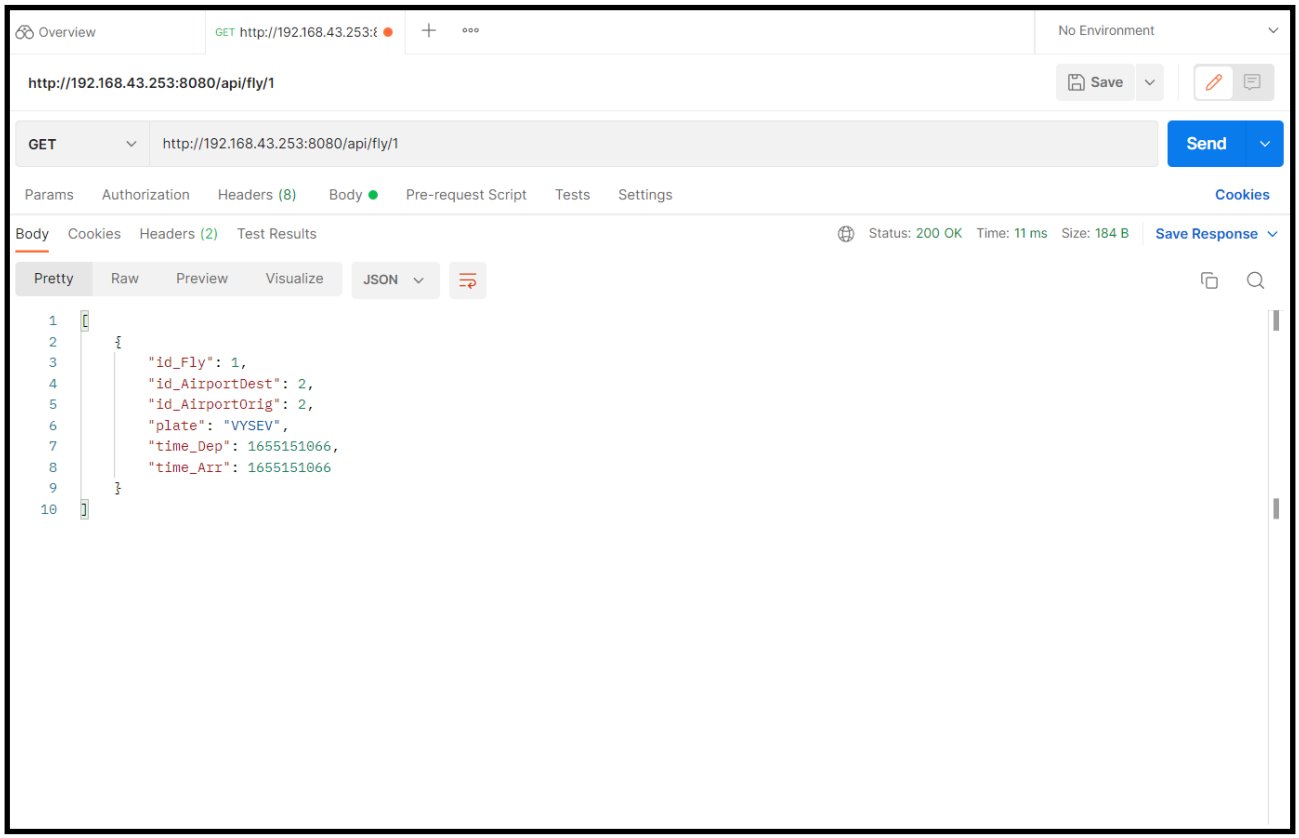


Post *postGps*

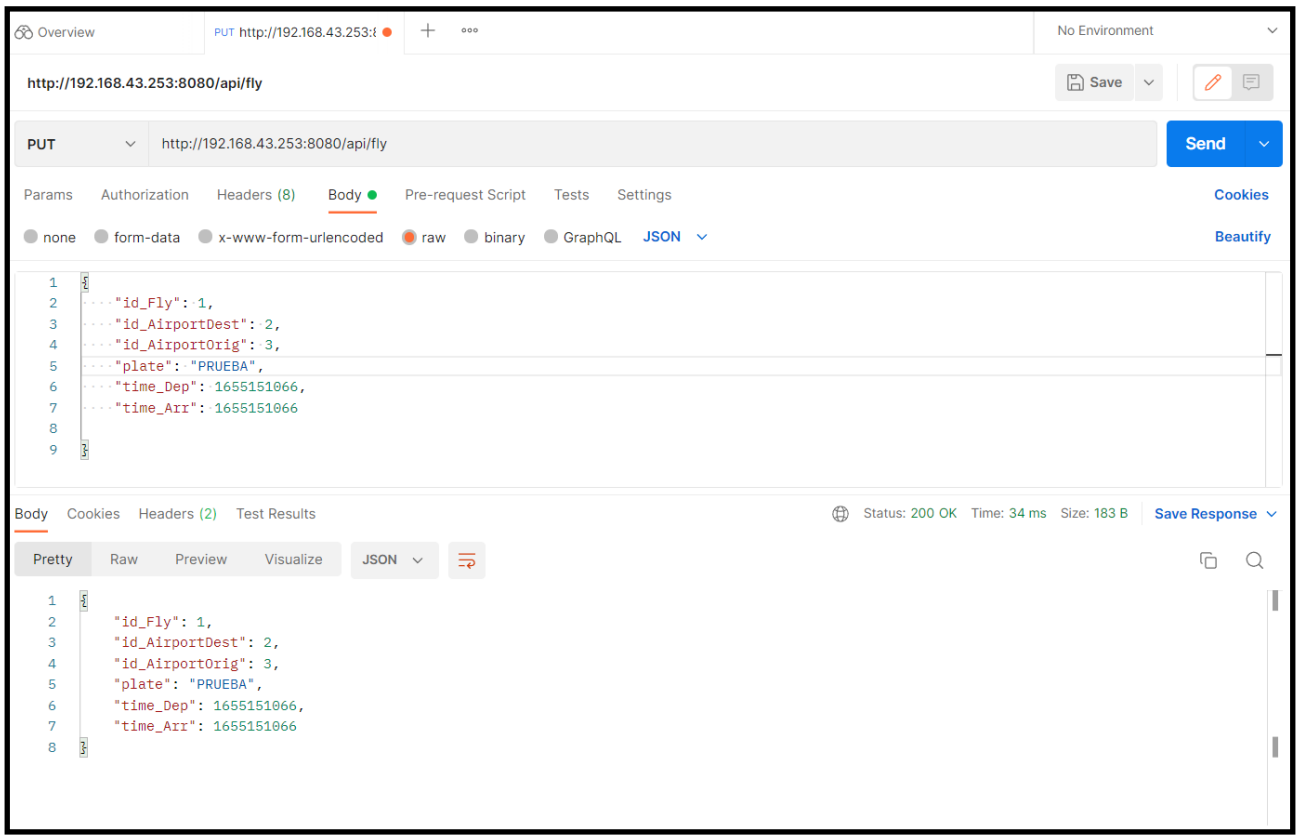
- **Peticiones Fly**



Get *getAllWithConnectionFly*

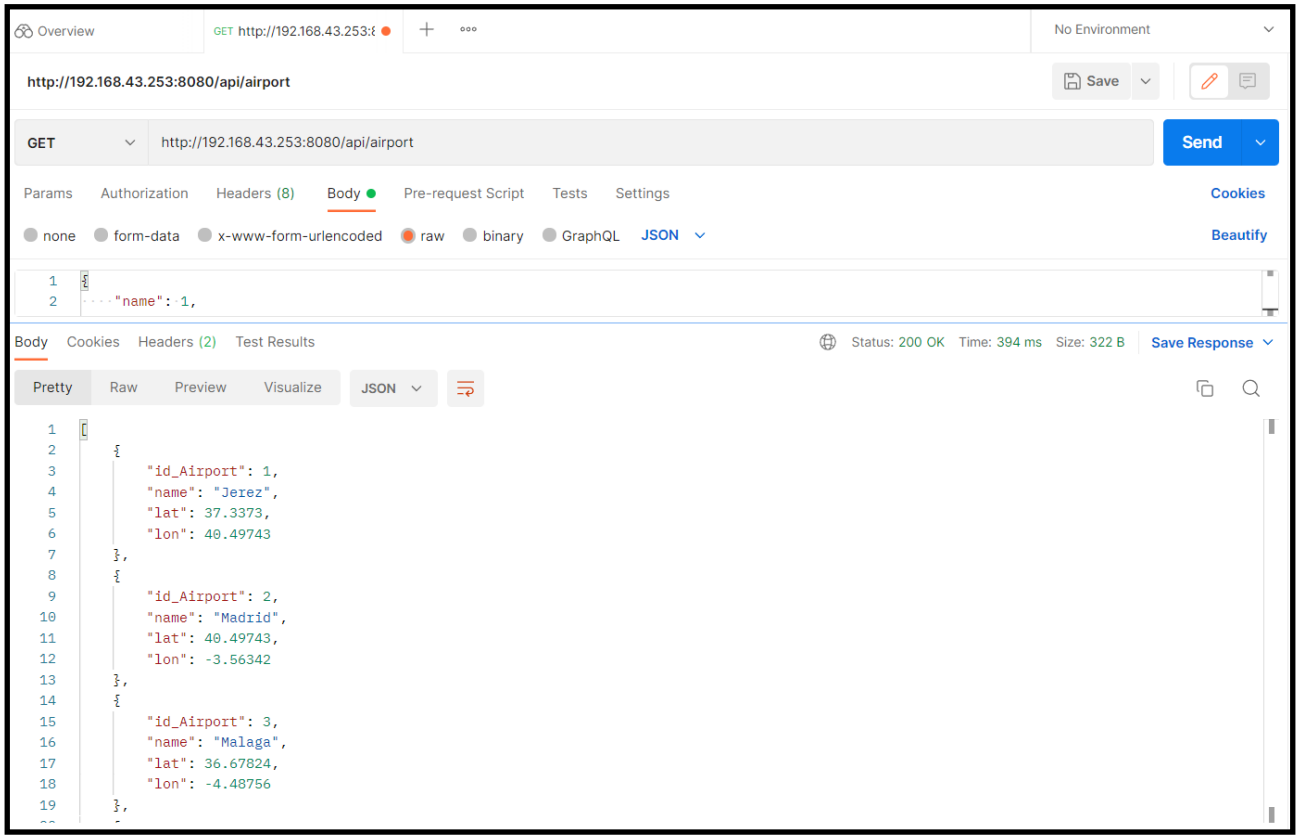


Get *getById_Fly*

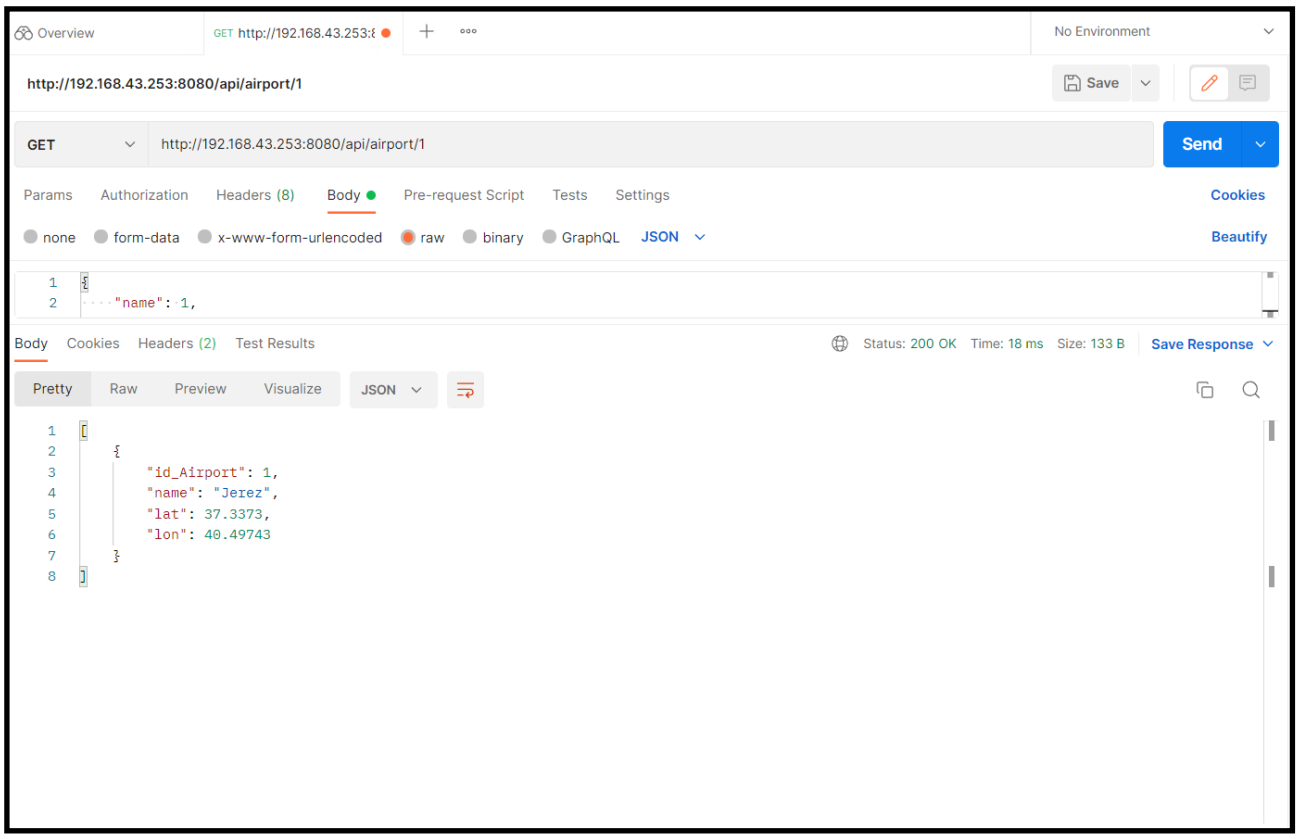


Put *putFly*

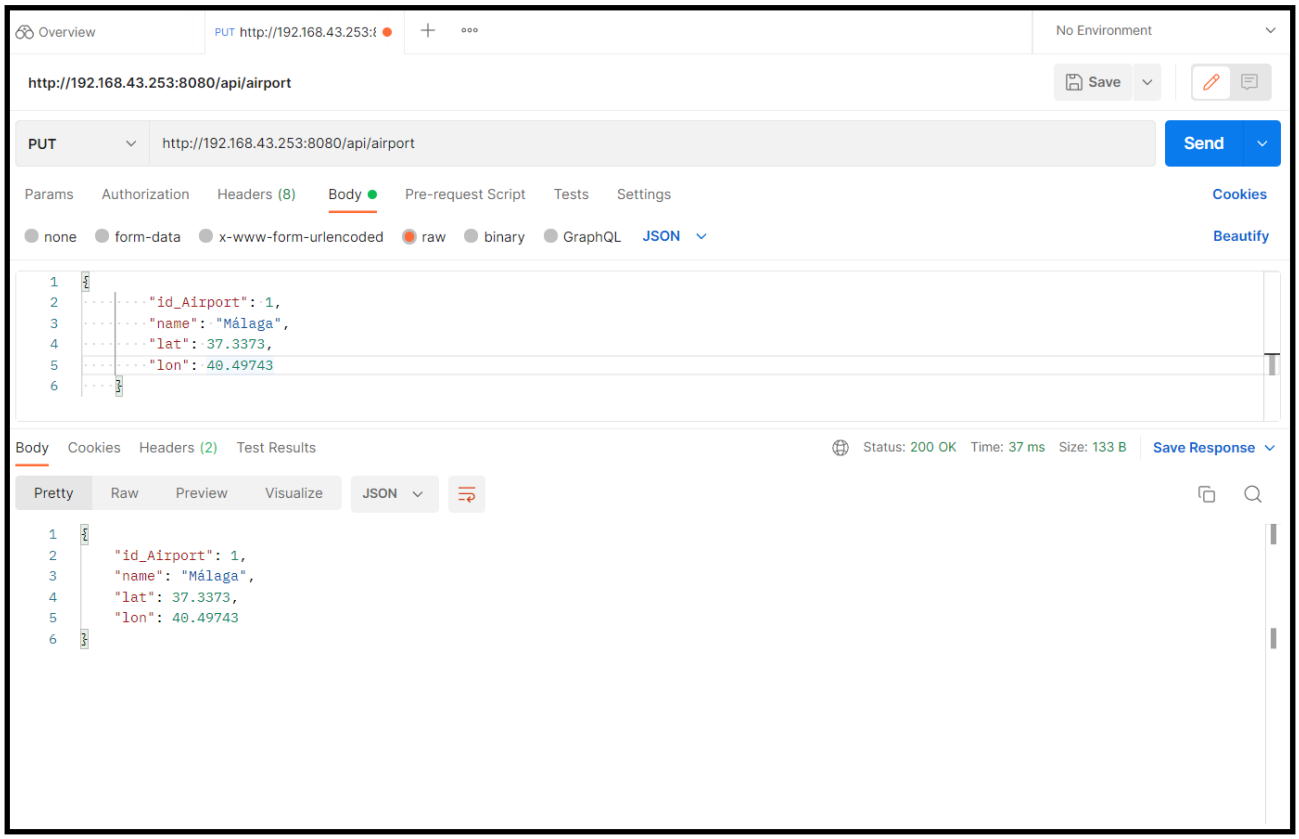
- **Peticiones Airport**



Get *getAllWithConnectionAirport*

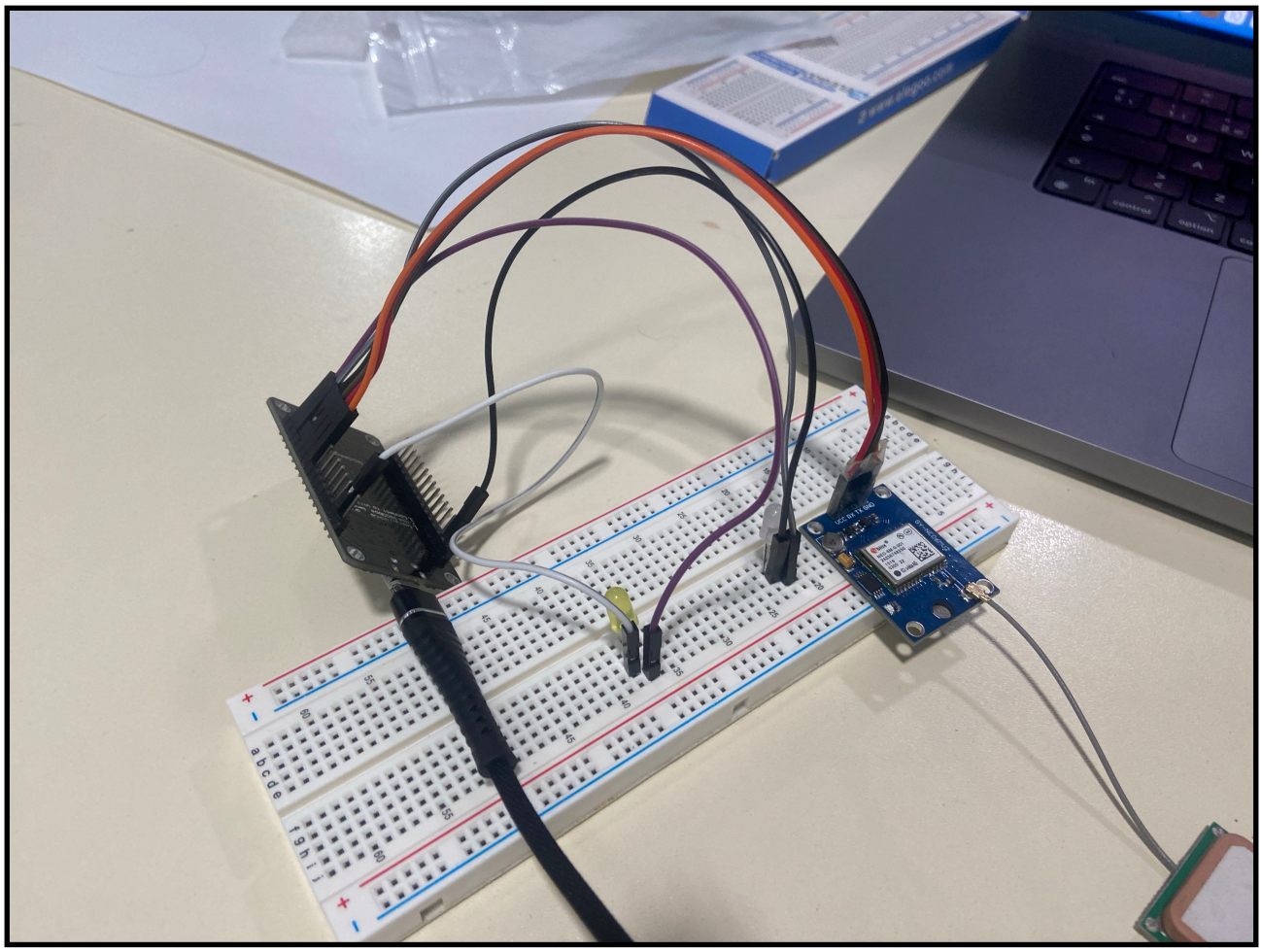


Get *getById_Airport*



Put *putAirport*

- **Montaje:**



Montaje con ambos leds y Gps

- **Link a repositorio de GitHub:**

https://github.com/F15javi/DAD.git