Ejercicio

FUNDAMENTOS DE INTERNET DE LA COSAS

2024-2025

Universidad Carlos III de Madrid

Cuaderno 2- Ejercicio - 10

2024/2025

MICROSERVICIOS PARA IOT

Cuaderno 2 - Ejercicio 10

Universidad Carlos III de Madrid. Escuela Politécnica Superior

Javier García Guzmán

Sección 1

Objetivos

El propósito de este ejercicio consiste en:

- Poner en práctica los conceptos necesarios para desarrollar microservicios que implementen los componentes de nube en IoT.
- Generar un contenedor con un gestor de base de datos para almacenar los datos operativos de una plataforma IoT.
- Desarrollar un microservicio para registrar los tacógrafos que se conecten a la red IoT.
- Desarrollar un microservicio para gestionar las conexiones y las sesiones de los tacógrafos cuando se conectan a la red IoT.
- Desarrollar un microservicio para registrar y consultar las telemetrías proporcionadas por los tacógrafos.
- Desarrollar un microservicio para registrar y consultar los eventos proporcionados por los tacógrafos.
- Modificar el componente de Message Router para que se integre con los microservicios desarrollados.

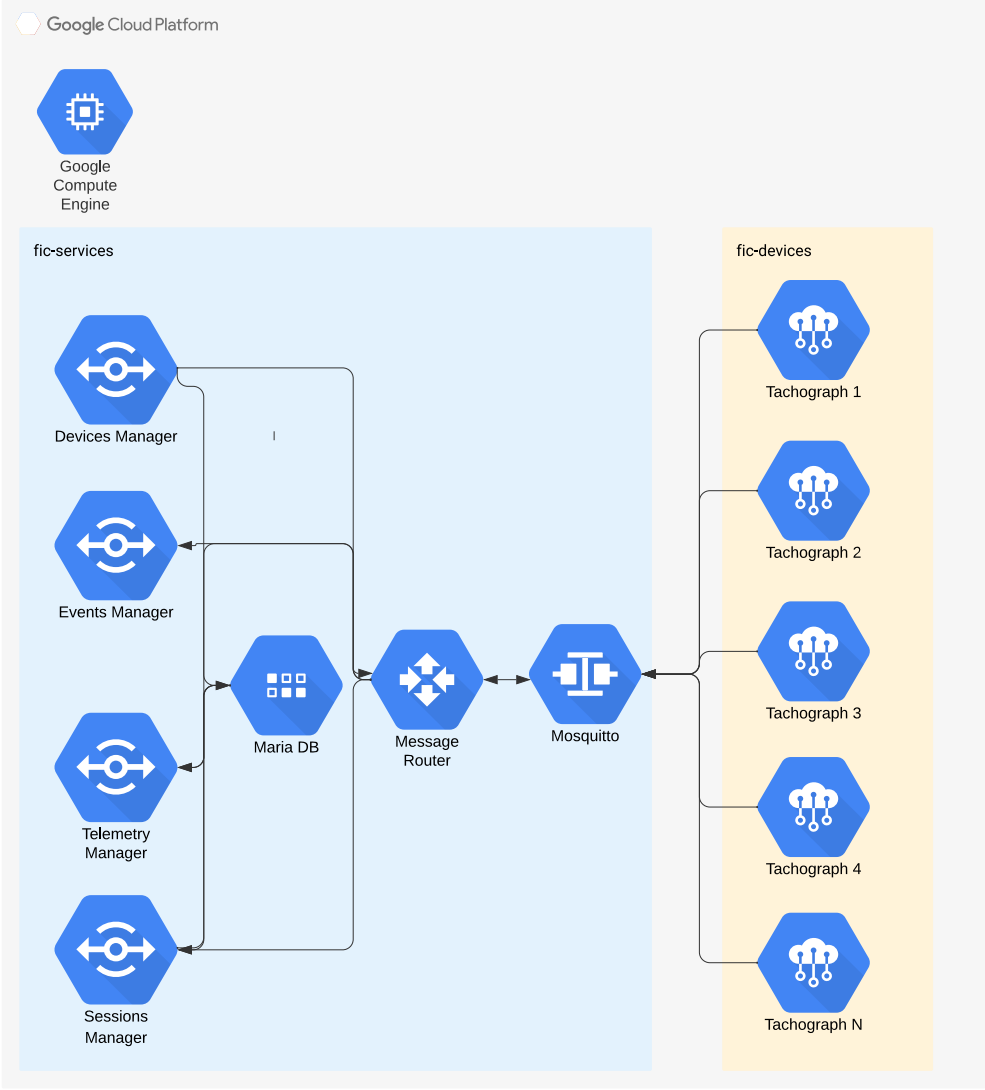
Sección 2

Introducción y pasos iniciales

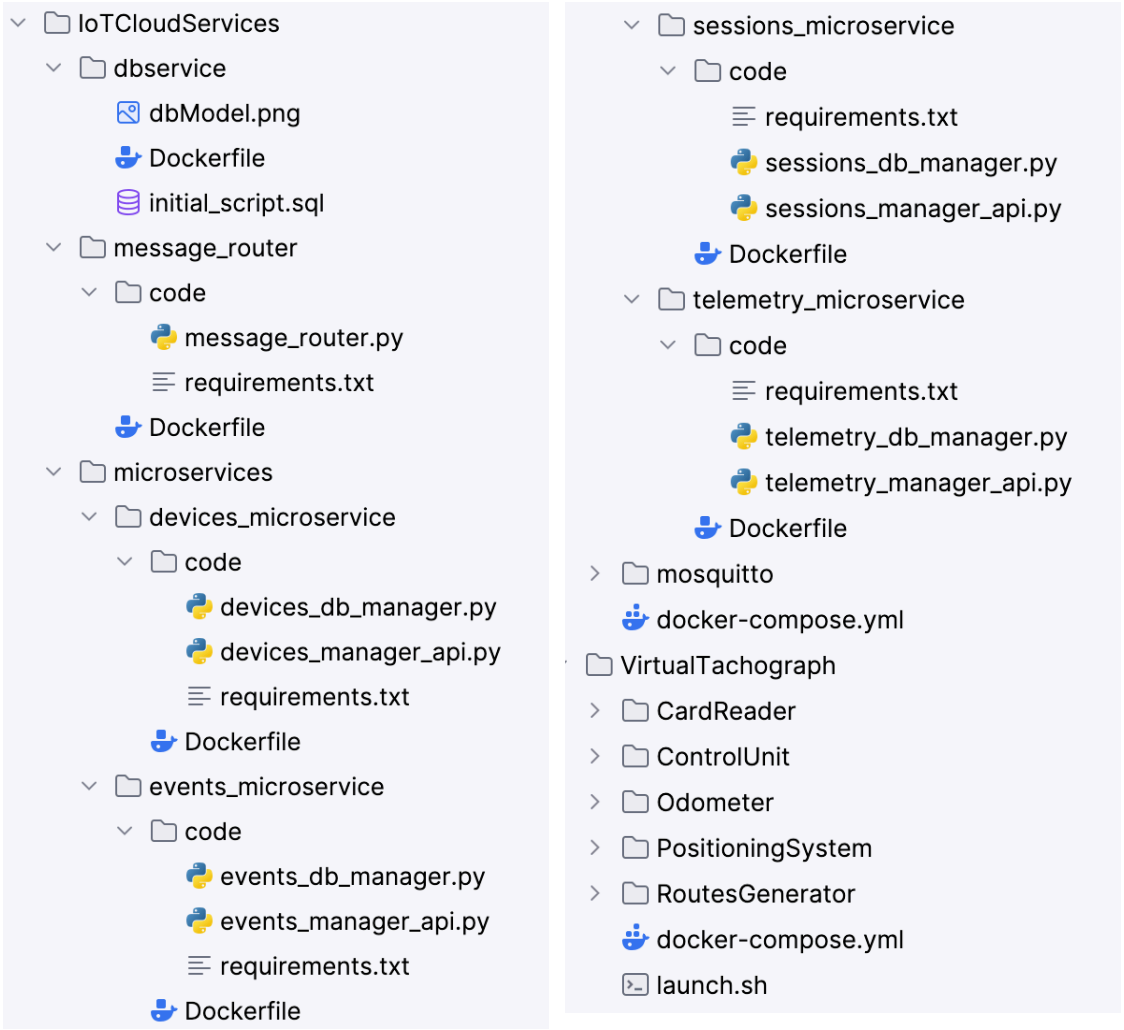
En el ámbito de este proyecto se partirá de la solución obtenida en la sesión 9.

Arquitectura y organización del proyecto

La arquitectura del proyecto es la siguiente:



La organización del código y los ficheros del proyecto será la siguiente:



El código se estructura en dos carpetas: *IoTCloudServices* su código se ejecutará en la máquina *fic-services*) y *VirtualTachograph* (su código se ejecutará en la máquina *fic-devices*)

- En *IoTCloudServices* habrá cuatro carpetas:
  - *mosquitto* contendrá el Dockerfile y los ficheros de configuración necesarios para el despliegue de esta solución MQTT de código libre en un contenedor.
  - *message_router* contendrá el código fuente en Python para la implementación de este servicio (este código se colocará en la subcarpeta *code* donde también existirá un fichero requirements.txt con las dependencias necesarias para el funcionamiento del código desarrollado). Además, esta carpeta tiene el fichero Dockerfile para el despliegue mediante un contenedor del componente Message Router.
  - *dbService* contendrá el Dockerfile para la creación del servicio de base de datos que se utilizará para almacenar los datos operativos de la red de tacógrafos IoT. También

contendrá un fichero con el script para la creación de la BD con el modelo definido para este caso práctico.

- *microservices* contendrá el código de los microservicios que se desarrollarán en el ejercicio. En el ámbito de este ejercicio, los microservicios a desarrollar son:
  - *devices_microservice*. Esta carpeta contiene los elementos de este microservicio está estructurada de la siguiente manera: la carpeta *code* tiene los ficheros de código fuente que se utilizarán para la implementación del microservicio (*devices_manager_api.py* y *devices_db_manager.py*). Asimismo, en la carpeta raíz del microservicio se incluye el *Dockerfile* que define la imagen a partir de la cual se creará el contenedor para el despliegue del microservicio.
  - *sessions_microservice*. Esta carpeta contiene los elementos de este microservicio está estructurada de la siguiente manera: la carpeta *code* tiene los ficheros de código fuente que se utilizarán para la implementación del microservicio (*sessions_manager_api.py* y *sessions_db_manager.py*). Asimismo, en la carpeta raíz del microservicio se incluye el *Dockerfile* que define la imagen a partir de la cual se creará el contenedor para el despliegue del microservicio.
  - *telemetry_microservice*. Esta carpeta contiene los elementos de este microservicio está estructurada de la siguiente manera: la carpeta *code* tiene los ficheros de código fuente que se utilizarán para la implementación del microservicio (*telemetry_manager_api.py* y *telemetry_db_manager.py*). Asimismo, en la carpeta raíz del microservicio se incluye el *Dockerfile* que define la imagen a partir de la cual se creará el contenedor para el despliegue del microservicio.
  - *events_microservice*. Esta carpeta contiene los elementos de este microservicio está estructurada de la siguiente manera: la carpeta *code* tiene los ficheros de código fuente que se utilizarán para la implementación del microservicio (*events_manager_api.py* y *events_db_manager.py*). Asimismo, en la carpeta raíz del microservicio se incluye el *Dockerfile* que define la imagen a partir de la cual se creará el contenedor para el despliegue del microservicio.

Por último, en la carpeta *IoTCloudServices* se incluirá el fichero *docker-compose.yml* con la orquestación de los contenedores que se despliegan en la máquina *fic-services*.

- En *VirtualTachographs* habrá una carpeta *VehicleDigitalTwin* y contendrá el código fuente en Python para la implementación de este servicio (este código se colocará en la subcarpeta *code* donde también existirá un fichero requirements.txt con las dependencias

necesarias para el funcionamiento del código desarrollado). Además, esta carpeta tiene el fichero Dockerfile para el despliegue mediante un contenedor de cada gemelo digital.

Por último, en la carpeta *VirtualTachographs* se incluirá el fichero *docker-compose.yml* con la orquestación de todos los gemelos digitales a contemplar en la máquina *fic-devices*.

Preparación del proyecto

En primer lugar, es necesario clonar el proyecto en gitlab (https://teaching.sel.inf.uc3m.es) correspondiente a la sesión 10.

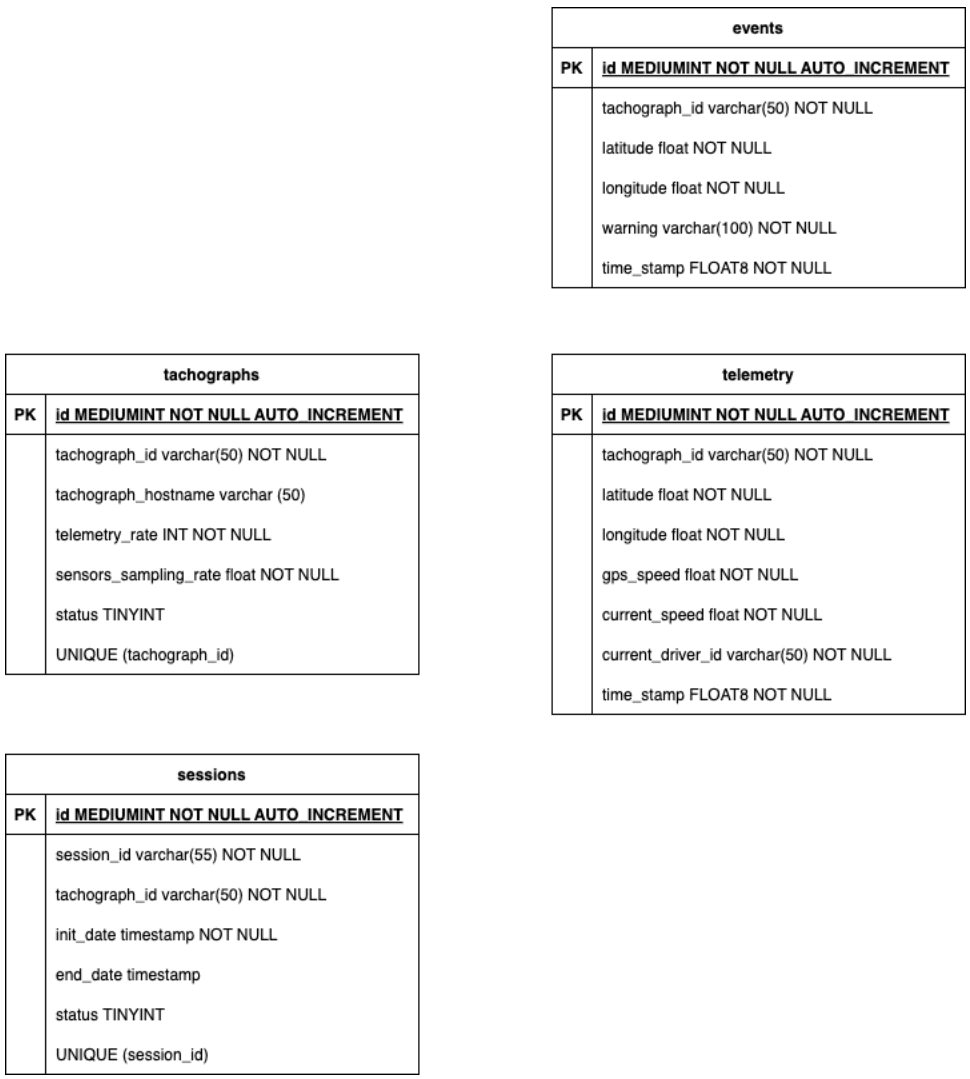
Se recomienda que, en este momento, se copien todos los ficheros de código y configuración desarrollados en la sesión 9 a la carpeta en la que se está desarrollando el código de la sesión 10, contemplando la estructura de carpetas presentada en el apartado anterior.

Sección 3

Despliegue del Servicio de Base de Datos

Modelo de la base de datos a desplegar

El modelo de la base de datos que se utilizará para desarrollar los microservicios incluidos en el ámbito del cuaderno de ejercicios 2, se muestra en la siguiente figura.



Se ha proporcionado un modelo simplificado sin restricciones de Foreing Key para facilitar la realización del ejercicio, sin embargo, se considerará positivamente su inclusión en el script que los estudiantes adapten para la generación de la base de datos.

En el fichero *initial_script.sql* que acompaña a este enunciado se proporciona un script inicial de la base de datos que puede ser adaptado por los estudiantes para la resolución del ejercicio 10 del cuaderno 2.

Creación de la imagen con Dockerfile

El fichero Dockerfile que se utilizará para generar la imagen necesaria para el contenedor de la base de datos se ubicará en la carpeta IoTCloudServices/dbService y su contenido tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a `mariadb:latest`
- Ejecutar el script inicial de creación de la base de datos. Un ejemplo de comando de este tipo es el siguiente:

```
ADD initial_script.sql /docker-entrypoint-initdb.d/ddl.sql
```

Si se necesita soporte acerca de los comandos para implementar los pasos anteriores en un Dockerfile, utilizar la hoja de resumen de comandos que se incluye en la sección 4 del enunciado del ejercicio 7.

Configuración de la orquestación del servicio de base de datos

Para desplegar la imagen de la base de datos en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-cloud-services*. Para ello, en la carpeta *IoTCloudServices* se actualizará el fichero denominado *docker-compose.yml*. Las instrucciones que se utilizarán para lanzar el servicio de base de datos son las siguientes:

```
dbService:
  build: ./dbService
  environment:
    - MYSQL_ROOT_PASSWORD=3u&Um%k{Jr
  ports:
    - '3306:3306'
```

El servicio de la base de datos se denomina dbService y va a construir un contendor a partir del Dockerfile que se encuentra en la carpeta *IoTCloudServices/dbService*.

Este servicio publicará el puerto 3306 del contenedor a través del puerto 3306 de la máquina que lo alberga (máquina host).

También se configurará la contraseña de root para poder acceder al MariaDB en el caso de que se deseen realizar comprobaciones y ejecuciones de script desde la línea de comandos del gestor de base de datos.

Despliegue del servicio de base de datos

En este momento, es necesario conectarse por *ssh* a la máquina *fic-cloud-services*. A continuación, se debe actualizar el código proveniente del repositorio correspondiente al código de esta sesión en esta máquina virtual (no olvidar que, con anterioridad, se ha tenido que publicar en el repositorio el código actualizado en la máquina de desarrollo).

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose build

docker compose up -d
```

Para verificar que la base de datos se encuentra en ejecución, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor de la base de datos>
```

Se puede ejecutar el siguiente comando para comprobar que el contenedor funciona correctamente y la base de datos está correctamente generada:

```
docker exec -it <nombre de contenedor de mariadb> mariadb -uroot -
p<password de root incluida en docker-compose.yml>
Nota: No hay espacio entre -p y la contraseña.
```

Una vez que se haya ingresado a la interfaz de línea de comandos de MariaDB, se pueden ejecutar los siguientes comandos:

```
use fic_data;
SELECT logic_id, is_assigned FROM available_ids ORDER BY logic_id;
```

Sección 4

Desarrollo del microservicio para la gestión de los vehículos

¿Qué es Flask?

Flask es un "micro" framework que pretende proporcionar lo mínimo necesario para que puedas poner a funcionar una API REST en cuestión de minutos.

La estructura básica de una aplicación flask para publicar una API REST es la siguiente:

```
from flask import Flask

from flask_cors import CORS

app = Flask(__name__)

CORS(app)

@app.route('/')

def hello_world():

    return 'Hello, World!'

if __name__ == '__main__':

                    app.run()
```

El objeto app de la clase Flask es nuestra aplicación WSGI, que nos permitirá posteriormente desplegar nuestra aplicación en un servidor Web. Se le pasa como parámetro el módulo actual (__name__).

A continuación, se inicializa la extensión Flask-Cors con argumentos por defecto para permitir CORS para todos los dominios en todas las rutas[1].

El prefijo app.route('/') permite filtrar la petición HTTP recibida, de tal forma que si la petición se realiza a la URL / se ejecutará la función vista hello_word.

La función vista que se ejecuta devuelve una respuesta HTTP. En este caso devuelve una cadena de caracteres que se será los datos de la respuesta.

Finalmente, si se ejecuta este módulo se ejecuta el método run que ejecuta un servidor web para que podamos probar la aplicación.

Las dependencias que se tienen que instalar en Python para el correcto funcionamiento de una API REST con Flask son las siguientes: *Flask* y *Flask-Cors*.

Desarrollo de la interfaz API REST con Flask

Para desarrollar el microservicio de tacógrafos es necesario crear dos ficheros *devices_manager_api.py* y *devices_db_manager.py*. El primero se utilizará para definir la interfaz HTTP expuesta por el microservicio y el segundo incluirá la implementación del microservicio.

Los requisitos que se tienen que satisfacer para la implementación de esta interfaz son los siguientes:

1. Se tienen que importar todas las librerías correspondientes al framework de Flask y sus extensiones:

[1] *El uso compartido de recursos entre orígenes (CORS) es un mecanismo para integrar aplicaciones. CORS define una forma con la que las aplicaciones web clientes cargadas en un dominio pueden interactuar con los recursos de un dominio distinto. Esto resulta útil porque las aplicaciones complejas suelen hacer referencia a API y recursos de terceros en el código del cliente. Por ejemplo, la aplicación puede utilizar su navegador para extraer videos de la API de una plataforma de video, utilizar fuentes de una biblioteca pública de fuentes o mostrar datos meteorológicos de una base de datos meteorológica nacional. CORS permite que el navegador del cliente compruebe con los servidores de terceros si la solicitud está autorizada antes de realizar cualquier transferencia de datos.*

```
from flask import Flask, request
from flask_cors import CORS
```

2. Se tienen que crear el esqueleto de la aplicación Flask de acuerdo con el esquema presentado en la sección 4. Sin embargo, en este caso, es necesario configurar la URL y el puerto por el cual se publicará la API REST. Esto se hace introduce sustituyendo la instrucción app.run() por la siguiente:

```
HOST = os.getenv('HOST')
PORT = os.getenv('PORT')
app.run(host= HOST, port=PORT)
```

   Las variables de entorno HOST y PORT nos indicarán cuál es la dirección y puerto en el que se publicará la API REST. HOST debe tomar el valor 0.0.0.0 y PORT debe tomar el valor 5001.

3. La API REST tendrá los siguientes métodos:

   - @app.route('/tachographs/', methods=['POST'])

     Este método recibe como parámetros de entrada los datos del tacógrafo en formato JSON.

     ```
     {"tachograph_id":"<id>"}
     ```

     La instrucción para obtener los parámetros de entrada es la siguiente:

     ```
     params = request.get_json()
     ```

     El procesamiento de los datos recibidos se hará por un método denominado *register_new_tachograph*, que se implementará en el fichero *devices_db_manager.py*. Su lógica se especifica en el siguiente apartado.

     En caso de que los datos proporcionados sean correctos se obtendrá el siguiente resultado:

     ```
     return "tachograph_id": tachograph_id}, 201
     ```

     En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

     ```
     return {"result": "error inserting a new tachograph"}, 500
     ```

   - @app.route('/tachographs/', methods=['GET'])

Este método no recibe parámetros de entrada y devuelve como resultado una lista en formato JSON con todos los IDs de los tacógrafos que se encuentran activos en el momento de realizar la consulta.

El procesamiento de los datos recibidos se hará por un método denominado *get_active_tachographs*, que se implementará en el fichero *devices_db_manager.py*. Su lógica se especifica en el siguiente apartado.

- @app.route('/tachographs/params/', methods=['GET'])

Este método recibe como parámetros de entrada los datos del tacógrafo en formato JSON.

```
{"tachograph_id":"<id>"}
```

El resultado proporcionado será la configuración de parámetros del tacógrafo seleccionado, siempre y cuando, esté en funcionamiento cuando se realiza la consulta.

En caso de que el tacógrafo cumpla con las condiciones anteriormente establecidas devolverá un JSON con una estructura similar a la siguiente:

```
{"tachograph_id": tachograph_id, "telemetry_rate": telemetry_rate,
  "sensors_sampling_rate": sensors_sampling_rate, "status": status}
```

En caso de que se produzca alguna situación de error se proporcionará un resultado similar al siguiente:

```
return {"result": "Error: Tachograph not found"}, 500
```

El procesamiento de los datos recibidos se hará por un método denominado *retrieve_tachograph(params)*, que se implementará en el fichero *devices_db_manager.py*. Su lógica se especifica en el siguiente apartado.

Desarrollo de la lógica para el almacenamiento y consulta de datos

La lógica para el almacenamiento y consulta de los datos gestionados por el microservicio de los vehículos son los siguientes:

- connect_database()

  El código para conectar a la base de datos se muestra a continuación:

```
mydb = mysql.connector.connect(
    host=os.getenv('DBHOST'),
    user=os.getenv('DBUSER'),
    password=os.getenv('DBPASSWORD'),
    database=os.getenv('DBDATABASE')
)
return mydb
```

- get_active_vehicles()

  Este método ejecuta una consulta a la tabla de tacógrafos que tengan como status el valor 1 (valor que indica que un vehículo está activo).

  La estructura básica de la ejecución de queries SQL es la que se muestra a continuación.

```
with mydb.cursor() as mycursor:
    mycursor.execute(sql)
    myresult = mycursor.fetchall()
    for tachograph in myresult:
        data = {"tachograph_id": tachograph}
        plates.append(data)
    mydb.commit()
```

  sql es una variable que contiene la query a ejecutar.

  Este código, para cada uno de los IDs obtenidos como resultados de la consulta, las agrega a una estructura de datos en JSON.

- register_new_tachograph(params)

  En primer lugar, es necesario consultar si el id que se incluye en el parámetro de entrada ya tiene una matrícula asignada:

```
SELECT tachograph_id FROM tachographs WHERE tachograph_id = %s
ORDER BY tachograph_id ASC LIMIT 1;
```

  - En caso de que haya un tacógrafo que tenga el ID asignado, se devolverá un mensaje que indique que ese ID no se puede volver a utilizar para que, posteriormente, se procese la salida de error correspondiente.

- En caso de que el id no haya sido previamente asignado, se ejecutará la siguiente consulta para saber si ese forma parte de los tacógrafos que se pueden instalar en vehículos:

```
SELECT logic_id FROM available_ids WHERE is_assigned = 0 AND
logic_id = %s ORDER BY logic_id ASC LIMIT 1;
```

En caso de que se obtenga un ID registrado entre los disponibles, se procederá a insertar en la BD los datos correspondientes al tacógrafo:

```
INSERT INTO tachographs (tachograph_id, telemetry_rate,
sensors_sampling_rate, status) VALUES (%s, 1, 1.0, 0);
```

y se actualiza la tabla de IDs disponibles para que ya aparezca como asignada.

```
UPDATE available_ids SET is_assigned = 1 WHERE logic_id = %s;
```

En caso de que no haya ID disponible, se devolverá como resultado "" para que la interfaz API REST considere de que hay una error asignando un ID al tacógrafo.

- retrieve_tachograph(params):

Este método ejecuta una consulta a la tabla de tacógrafos para obtener los valores de telemetry_rate, tachograph_hostname, sensors_sampling_rate que tengan como status el valor 1 (valor que indica que un vehículo está activo) y su ID coincida con el que se ha proporcionado como parámetro.

La estructura básica de la ejecución de queries SQL es la que se muestra a continuación.

```
with mydb.cursor() as mycursor:
    mycursor.execute(sql)
    myresult = mycursor.fetchall()

   for tachograph_id, telemetry_rate, tachograph_hostname,
   sensors_sampling_rate, status in myresult:
       data = {"tachograph_id": tachograph_id,
   "telemetry_hostname": tachograph_hostname,
               "sensors_sampling_rate": sensors_sampling_rate,
   "telemetry_rate": telemetry_rate, "status": status}
       tachographs.append(data)
mydb.commit()
```

sql es una variable que contiene la query a ejecutar.

Este código agrega a una estructura de datos JSON cada uno de los ítems incluidos en el result set.

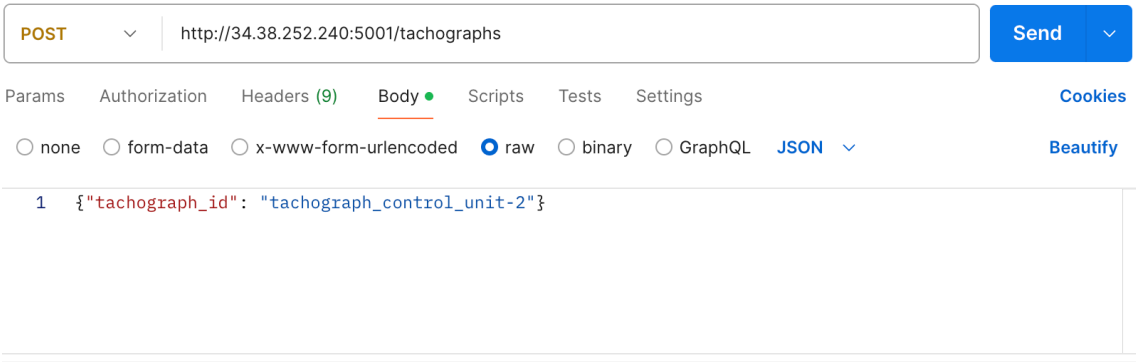
Prueba de Devices Microservice

Una vez que se haya finalizado el desarrollo, se debe probar el correcto funcionamiento del código desarrollado.

Para ello, se recomienda utilizar Postman. Esta es una herramienta útil para el diseño y prueba de API REST. Se puede descargar en el siguiente enlace: https://www.postman.com/downloads/?utm_source=postman-home

Los dos tipos de prueba a realizar con Postman son los siguientes:

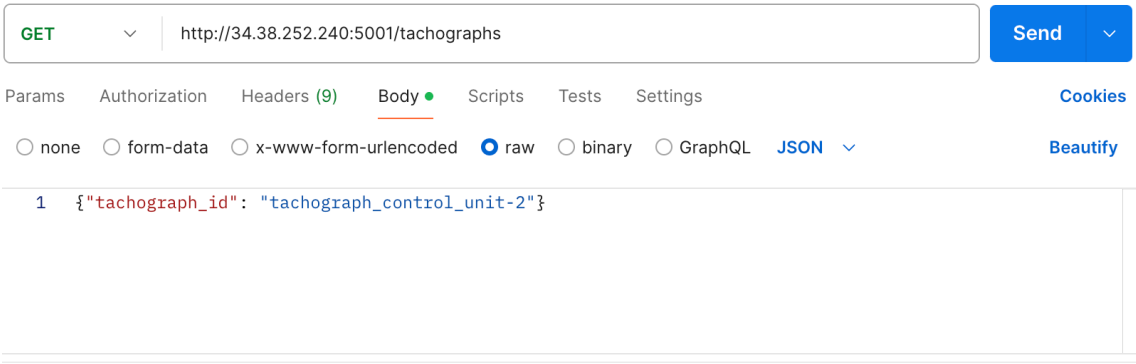
- @app.route('/tachographs/', methods=['POST'])



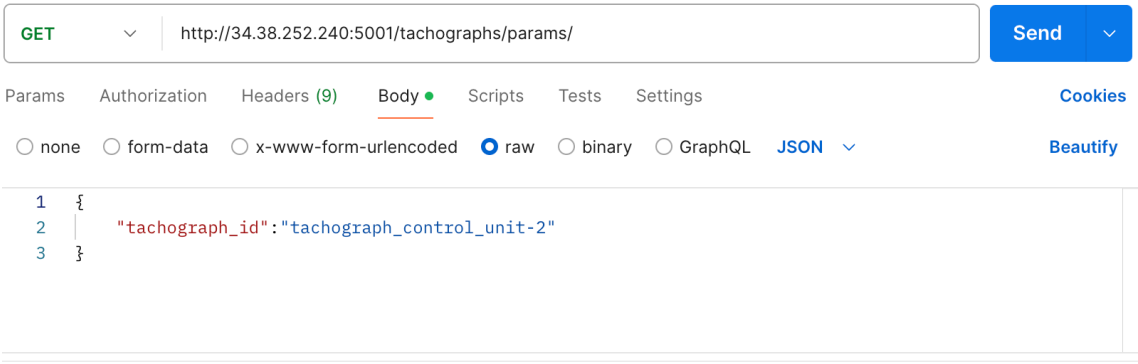
  Se utiliza localhost porque se están realizando las pruebas en local. Cuando se realicen las pruebas sobre la máquina virtual *fic-services*, habrá que sustituir localhost por la dirección IP que tenga la máquina virtual. El puerto es el 5001 porque es el configurado para la publicación del microservicio.

  El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- @app.route('/tachographs/active/', methods=['GET'])



- @app.route('/tachographs/active/', methods=['GET'])



Cuando finalice la verificación de que el código funciona como se espera, el código desarrollado se debe publicar en el repositorio de control de versiones.

Creación de la imagen con Dockerfile

Una vez que se ha probado que el código del microservicio funciona correctamente en el equipo de desarrollo, se procederá a la preparación para su despliegue con contenedores.

Para ello, en la carpeta *IoTCloudServices/microservices/devices_microservice* se creará el fichero Dockerfile.

El contenido de este fichero tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a python:3.11.1
- Copiar el código que está en la carpeta *./code* a la carpeta */etc/usr/src/app*
- Establecer esta carpeta como directorio de trabajo.
- Instalar los paquetes necesarios especificados en requirements.txt. Los paquetes necesarios son: *Flask*, *Flask-Cors* y *mysql-connector-python,* para ello ejecutar pip install <paquetes necesarios>
- Ejecutar el código incluido en el fichero *devices_manager_api.py*

Configuración de la orquestación del microservicio

Para ello, en la carpeta *IoTCloudServices* se modificará el fichero denominado *docker-compose.yml.* El nuevo contenido de este fichero debe incluir la definición de la orquestación del servicio devices_microservice de acuerdo con el siguiente código:

```
devices_microservice:
  build: ./microservices/devices_microservice
  ports:
    - '5001:5001'
  links:
    - "dbservice:dbservice"
  environment:
    - HOST=0.0.0.0
    - PORT=5001
    - DBHOST=dbservice
    - DBUSER=fic_db_user
    - DBPASSWORD=secret1234
    - DBDATABASE=fic_data
    - PYTHONUNBUFFERED=1
  volumes:
    - "./microservices/devices_microservice/code:/etc/usr/src/app"
  depends_on:
    - dbservice
```

Despliegue del microservicio

Para desplegar las versiones actualizadas del gemelo digital del simulador de vehículos en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-services*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-services*. A continuación, se debe clonar el repositorio correspondiente al código de esta sesión en esta máquina virtual.

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose build

docker compose up -d
```

Para verificar el despliegue, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor>
```

Sección 5

Desarrollo del microservicio para la gestión de las sesiones

Para desarrollar el microservicio de tacógrafos es necesario crear dos ficheros *sessions_manager_api.py* y *sessions_db_manager.py*. El primero se utilizará para definir la interfaz HTTP expuesta por el microservicio y el segundo incluirá la implementación del microservicio.

Los requisitos que se tienen que satisfacer para la implementación de esta interfaz son los siguientes:

1. Se tienen que importar todas las librerías correspondientes al framework de Flask y sus extensiones como con el microservicio descrito en la sección 4.
2. Se tienen que crear el esqueleto de la aplicación Flask de acuerdo con el esquema presentado en la sección 4. Sin embargo, en este caso, es necesario configurar la URL y el puerto por el cual se publicará la API REST. Las variables de entorno HOST y PORT nos indicarán cuál es la dirección y puerto en el que se publicará la API REST. HOST debe tomar el valor 0.0.0.0 y PORT debe tomar el valor 5002.
3. La API REST tendrá los siguientes métodos:
   - @app.route('/sessions/', methods=['PUT'])

     Este método recibe como parámetros de entrada los datos del tacógrafo en formato JSON.

     ```
     {"tachograph_id":"<id>",

     "tachograph_hostname":<hostname>}
     ```

     La instrucción para obtener los parámetros de entrada es la siguiente:

     ```
     params = request.get_json()
     ```

     El procesamiento de los datos recibidos se hará por un método denominado *register_new_session*, que se implementará en el fichero *sessions_db_manager.py*.

El propósito de este método consiste en registrar una nueva sesión en la base de datos, incluyendo su session_id (que será una cadena de caracteres única que impida que dos sesiones tengan el mismo id), tachograph_id (que es el mismo que se proporciona como parámetro de entrada, init_date (una marca de tiempo del momento en el que se registra la sesión en la base de datos, y el status (que será 1 porque la sesión está activa).

Asimismo, en caso de que haya sido posible registrar la sesión, se actualizará la fila correspondiente de la tabla de tacógrafos, indicando que el *status* es igual a 1 y que el campo *tachograph_hostname* es igual al recibido como parámetro de entrada.

En caso de que los datos proporcionados sean correctos proporcionará el siguiente resultado:

```
return {"session_id": session_id}, 201
```

En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

```
return {"session_id": "", "Error":"Access not granted"}, 500
```

- @app.route('/sessions/', methods=['POST'])

  Este método recibe como parámetros de entrada los datos del tacógrafo en formato JSON.

  ```
  {"tachograph_id":"<id>",

  "tachograph_hostname":<hostname>}
  ```

  La instrucción para obtener los parámetros de entrada es la siguiente:

  ```
  params = request.get_json()
  ```

  El procesamiento de los datos recibidos se hará por un método denominado *register_session_disconnection*, que se implementará en el fichero *sessions_db_manager.py*.

  En primer lugar, de la tabla tachogrpahs se obtiene el hostname actual del tacógrafo que está consultando y que está activo.

Posteriormente, se actualizan las sesiones activas correspondientes al tacógrafo que se proporciona como parámetro, estableciendo su status igual a 0 y su end_date con la marca de tiempo del momento en el que se está realizando la actualización.

Por último, se tiene que actualizar la información del tacógrafo estableciendo su *status* igual a 0 (desconectado) y si *tachograph_hostname* con una cadena de texto vacía.

En caso de que el registro de la desconexión sea correcta, se devolverá el siguiente resultado:

```
return {"tachograph_hostname": tachograph_hostname, "sessions":
sessions}, 201
```

En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

```
return {"tachograph_hostname": "", "sessions": []}, 500
```

- @app.route('/sessions/', methods=['GET'])

Este método recibe como parámetros de entrada los datos del tacógrafo en formato JSON.

```
{"tachograph_id":"<id>"}
```

La instrucción para obtener los parámetros de entrada es la siguiente:

```
params = request.get_json()
```

El procesamiento de los datos recibidos se hará por un método denominado *is_connected*, que se implementará en el fichero *sessions_db_manager.py*.

Este método permitirá comprobar si un tacógrafo tiene sesiones abiertas.

En caso de que el registro de la desconexión sea correcta, se devolverá el siguiente resultado:

```
return {"tachograph_hostname": tachograph_hostname, "sessions":
sessions}, 201
```

En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

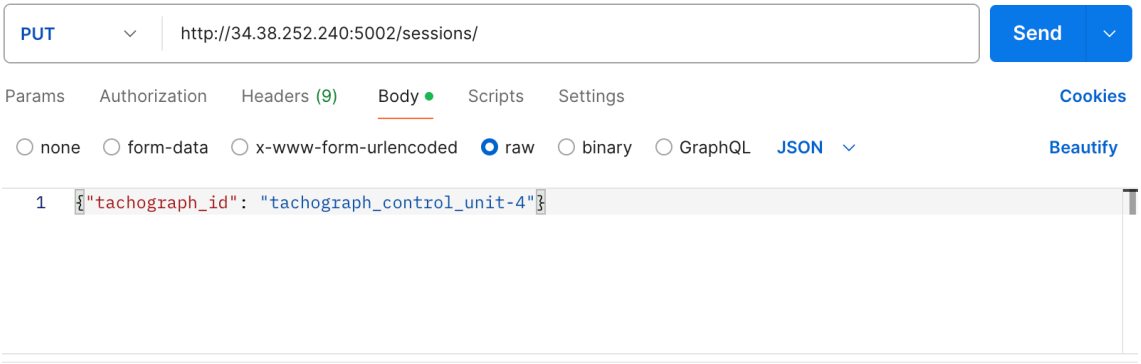
```
return {"tachograph_hostname": "", "sessions": []}, 500
```

Prueba de Sessions Microservice

Una vez que se haya finalizado el desarrollo, se debe probar el correcto funcionamiento del código desarrollado.

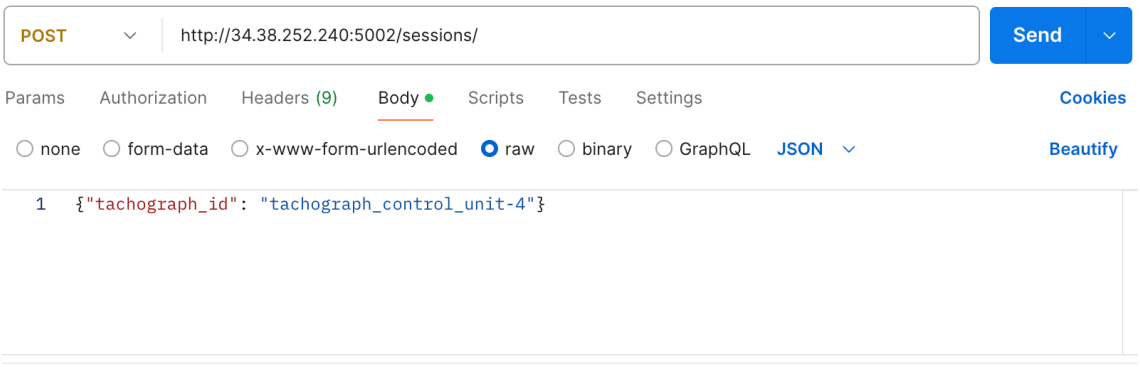
Los dos tipos de prueba a realizar con Postman son los siguientes:

- @app.route('/sessions/', methods=['PUT'])



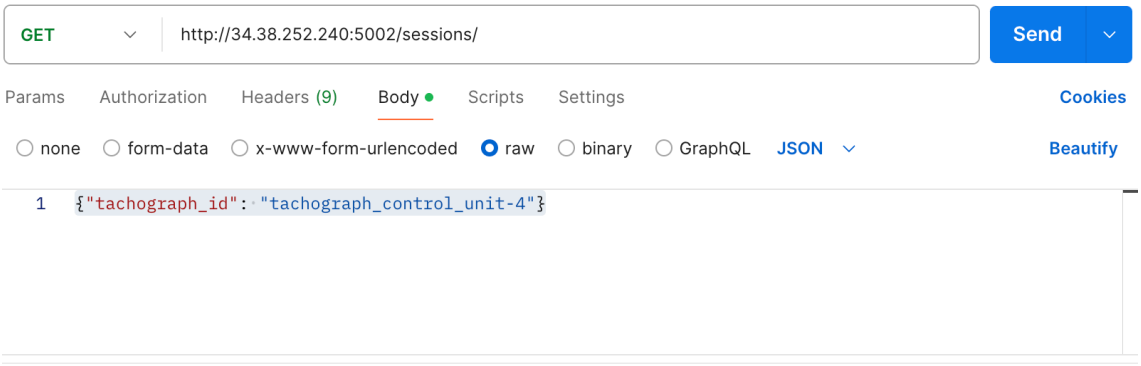
El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- @app.route('/sessions/', methods=['POST'])



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- @app.route('/sessions/', methods=['GET'])



Cuando finalice la verificación de que el código funciona como se espera, el código desarrollado se debe publicar en el repositorio de control de versiones.

Creación de la imagen con Dockerfile

Una vez que se ha probado que el código del microservicio funciona correctamente en el equipo de desarrollo, se procederá a la preparación para su despliegue con contenedores.

Para ello, en la carpeta *IoTCloudServices/microservices/sessions_microservice* se creará el fichero Dockerfile.

El contenido de este fichero tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a python:3.11.1
- Copiar el código que está en la carpeta *./code* a la carpeta */etc/usr/src/app*
- Establecer esta carpeta como directorio de trabajo.
- Instalar los paquetes necesarios especificados en requirements.txt. Los paquetes necesarios son: *Flask*, *Flask-Cors* y *mysql-connector-python,* para ello ejecutar pip install <paquetes necesarios>
- Ejecutar el código incluido en el fichero *sessions_manager_api.py*

Configuración de la orquestación del microservicio

Para ello, en la carpeta *IoTCloudServices* se modificará el fichero denominado *docker-compose.yml*. El nuevo contenido de este fichero debe incluir la definición de la orquestación del servicio sessions_microservice de acuerdo con el siguiente código:

```
sessions_microservice:
  build: ./microservices/sessions_microservice
  ports:
    - '5002:5002'
  links:
```

```
    - "dbservice:dbservice"
  environment:
    - HOST=0.0.0.0
    - PORT=5002
    - DBHOST=dbservice
    - DBUSER=fic_db_user
    - DBPASSWORD=secret1234
    - DBDATABASE=fic_data
    - PYTHONUNBUFFERED=1
  volumes:
    - "./microservices/sessions_microservice/code:/etc/usr/src/app"
  depends_on:
    - dbservice
```

Despliegue del microservicio

Para desplegar las versiones actualizadas del gemelo digital del simulador de vehículos en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-services*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-services*. A continuación, se debe clonar el repositorio correspondiente al código de esta sesión en esta máquina virtual.

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose build

docker compose up -d
```

Para verificar el despliegue, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor>
```

Sección 6

Desarrollo del microservicio para la gestión de telemetrías

En la carpeta *IoTCloudServices/microservices/telemetry_microservice/code* se tienen que crear dos ficheros con el código fuente que será necesario desarrollar: *telemetry_manager_api.py* y *telemetry_db_manager.py*.

Desarrollo de la interfaz API REST con Flask

La interfaz API REST que se implementará con Flask para el microservicio telemetry_microservice, se incluirá en el fichero *telemetry_manager_api.py*.

Los requisitos que se tienen que satisfacer para la implementación de esta interfaz son los siguientes:

1. Se tienen que importar todas las librerías correspondientes al framework de Flask y sus extensiones como con el microservicio anterior.
2. Se tienen que crear el esqueleto de la aplicación Flask de acuerdo con el esquema presentado en la sección 4. Las variables de entorno HOST y PORT nos indicarán cuál es la dirección y puerto en el que se publicará la API REST. HOST debe tomar el valor 0.0.0.0 y PORT debe tomar el valor 5003.
3. La API REST tendrá los siguientes métodos:
   - @app.route('/telemetry /', methods=['POST'])

     Este método recibe como parámetros de entrada los datos de telemetría enviados por el message router.

```
{
    " Tachograph_id":"1234BBC",
    "position": {"Latitude":40.28908, "Longitude":-4.01197},
    "GPSSpeed":0.0,
    "Speed":0.0,
    "Driver":”Driver 1”,
    "time_stamp":"2023-11-27 17:48:52"
}
```

En caso de que todo haya ido bien, este método devolverá un mensaje informativo, junto con el código de éxito 201:

```
return {"result": "Telemetry registered"}, 201
```

En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

```
return {"result": "Error registering telemetries "}, 500
```

El procesamiento de los datos recibidos se hará por un método denominado *register_new_telemetry*, que se implementará en el fichero *telemetry_db_manager.py*.

- @app.route('/telemetry/', methods=['GET'])

Este método recibe como parámetros de entrada los datos del tacógrafo y del intervalo temporal que se quiere consultar en formato JSON.

```
{
    " Tachograph_id":"<Tachograph_id>",
    "init_interval":<Date Time>,
    "end_interval":<Date Time>
}
```

Este método devuelve como resultado una lista en formato JSON con las telemetrías registradas para el tacógrafo indicado en el periodo de tiempo que se ha indicado.

El procesamiento de los datos recibidos se hará por un método denominado *query_telemetry*, que se implementará en el fichero *telemetry_db_manager.py*.

```
return result, 201
```

- @app.route('/telemetry/positions/', methods=['GET'])

Este método no recibe parámetros de entrada.

Devuelve como resultado una lista en formato JSON que incluya, para cada uno de los vehículos activos, la matrícula, latitud y longitud de su última posición.

```
error_message, result = get_tachogrpah_last_position()
if error_message == "":
    return result, 201
else:
    return {"Error Message": error_message}, 500
```

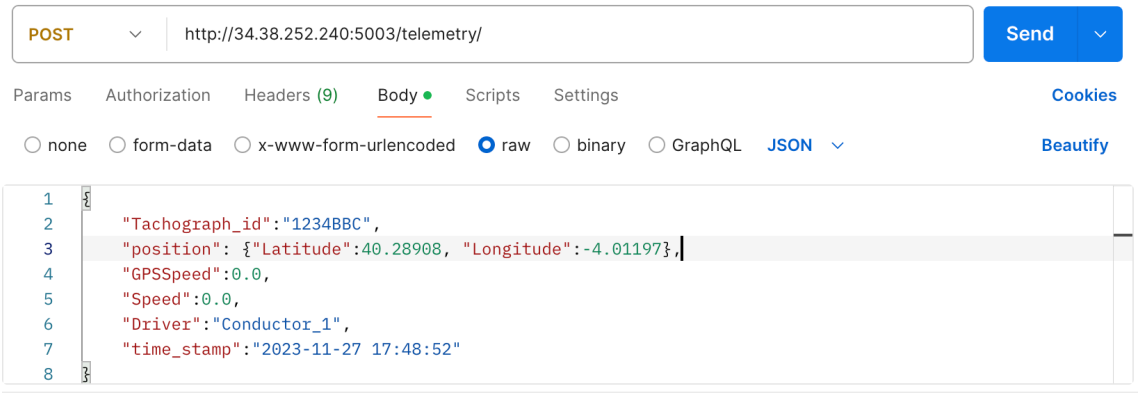
El procesamiento de los datos recibidos se hará por un método denominado *retrieve_vehicles_last_position*, que se implementará en el fichero *telemetry_db_manager.py*.

Prueba de Telemetry Microservice

Una vez que se haya finalizado el desarrollo, se debe probar el correcto funcionamiento del código desarrollado.

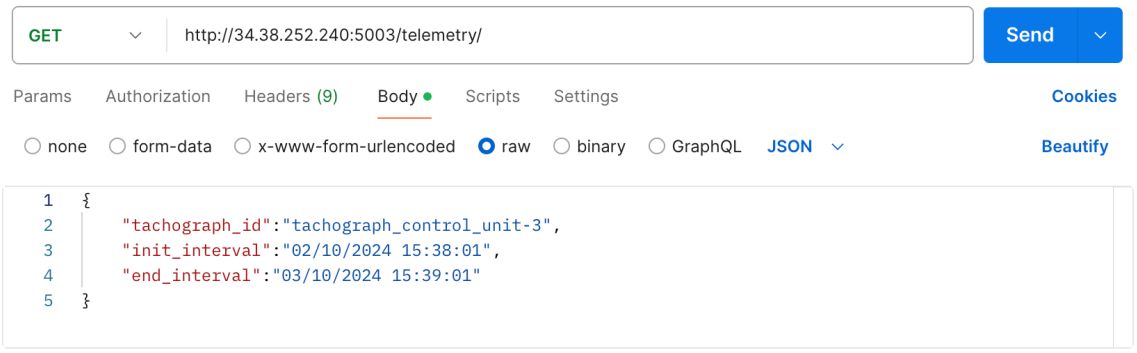
Los dos tipos de prueba a realizar con Postman son los siguientes:

- @app.route('/telemetry /', methods=['POST'])



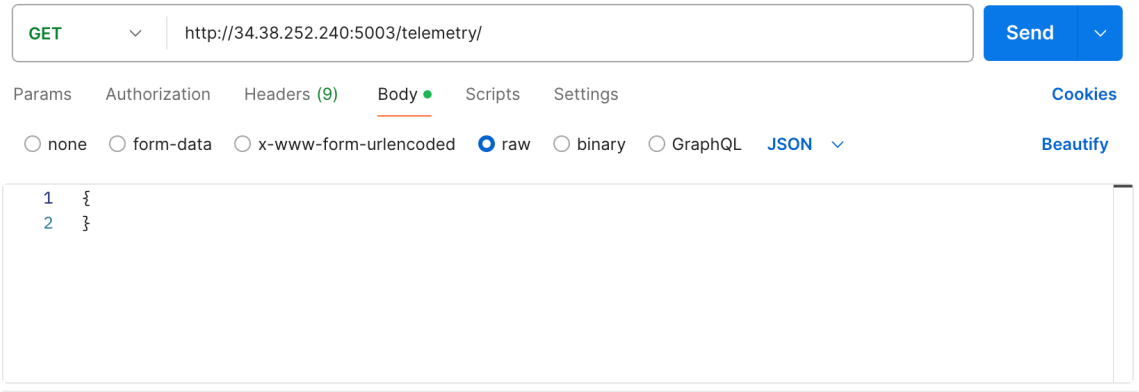
El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- @app.route('/telemetry/', methods=['GET'])



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- @app.route('/telemetry/vehicle/positions/', methods=['GET'])



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

Cuando finalice la verificación de que el código funciona como se espera, el código desarrollado se debe publicar en el repositorio de control de versiones.

Creación de la imagen con Dockerfile

Una vez que se ha probado que el código del microservicio funciona correctamente en el equipo de desarrollo, se procederá a la preparación para su despliegue con contenedores.

Para ello, en la carpeta *IoTCloudServices/microservices/telemetry_microservice* se creará el fichero Dockerfile.

El contenido de este fichero tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a python:3.12
- Copiar el código que está en la carpeta *./code* a la carpeta */etc/usr/src/app*
- Establecer esta carpeta como directorio de trabajo.
- Instalar los paquetes necesarios especificados en requirements.txt. Los paquetes necesarios son: *Flask*, *Flask-Cors* y *mysql-connector-python,* para ello ejecutar pip install <paquetes necesarios>
- Ejecutar el código incluido en el fichero *telemetry_manager_api.py*

Configuración de la orquestación del microservicio

Para ello, en la carpeta *IoTCloudServices* se modificará el fichero denominado *docker-compose.yml.* El nuevo contenido de este fichero debe incluir la definición de la orquestación del servicio telemetry_microservice de acuerdo con el siguiente código:

```
telemetry_microservice:
  build: ./microservices/telemetry_microservice
  ports:
    - '5003:5003'
  links:
    - "dbservice:dbservice"
  environment:
    - HOST=0.0.0.0
    - PORT=5003
    - DBHOST=dbservice
    - DBUSER=fic_db_user
    - DBPASSWORD=secret1234
    - DBDATABASE=fic_data
    - PYTHONUNBUFFERED=1
  volumes:
    - "./microservices/telemetry_microservice/code:/etc/usr/src/app"
  depends_on:
    - dbservice
```

Despliegue del microservicio

Para desplegar las versiones actualizadas del gemelo digital del simulador de vehículos en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-cservices*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-services*. A continuación, se debe clonar el repositorio correspondiente al código de esta sesión en esta máquina virtual.

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose build

docker compose up -d
```

Para verificar el despliegue, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor>
```

Sección 7

Desarrollo del microservicio para la gestión de eventos

En la carpeta *IoTCloudServices/microservices/events_microservice/code* se tienen que crear dos ficheros con el código fuente que será necesario desarrollar: *events_manager_api.py* y *events_db_manager.py*.

Desarrollo de la interfaz API REST con Flask

La interfaz API REST que se implementará con Flask para el microservicio events_microservice, se incluirá en el fichero *events_manager_api.py*.

Los requisitos que se tienen que satisfacer para la implementación de esta interfaz son los siguientes:

1. Se tienen que importar todas las librerías correspondientes al framework de Flask y sus extensiones como con el microservicio anterior.
2. Se tienen que crear el esqueleto de la aplicación Flask de acuerdo con el esquema presentado en la sección 4. Las variables de entorno HOST y PORT nos indicarán cuál es la dirección y puerto en el que se publicará la API REST. HOST debe tomar el valor 0.0.0.0 y PORT debe tomar el valor 5004.
3. La API REST tendrá los siguientes métodos:
   - @app.route('/event/', methods=['POST'])

     Este método recibe como parámetros de entrada los datos de eventos enviados por el message router.

```
{
   "tachograph_id":"tachograph_control_unit-4",
   "position": {"Latitude":40.28908, "Longitude":-4.01197},
   "warning": "Warning Message",
   "Timestamp": "2023-11-27 17:48:52"
}
```

En caso de que todo haya ido bien, este método devolverá un mensaje informativo, junto con el código de éxito 201:

```
return {"result": "Event registered"}, 201
```

En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

```
return {"result": "Error registering an event "}, 500
```

El procesamiento de los datos recibidos se hará por un método denominado *register_new_telemetry*, que se implementará en el fichero *events_db_manager.py*.

- @app.route('/telemetry/', methods=['GET'])

Este método recibe como parámetros de entrada los datos del tacógrafo y del intervalo temporal que se quiere consultar en formato JSON.

```
{
    " Tachograph_id":"<Tachograph_id>",
    "init_interval":<Date Time>,
    "end_interval":<Date Time>
}
```

Este método devuelve como resultado una lista en formato JSON con los eventos registradas para el tacógrafo indicado en el periodo de tiempo que se ha indicado.

El procesamiento de los datos recibidos se hará por un método denominado *query_events*, que se implementará en el fichero *events_db_manager.py*.

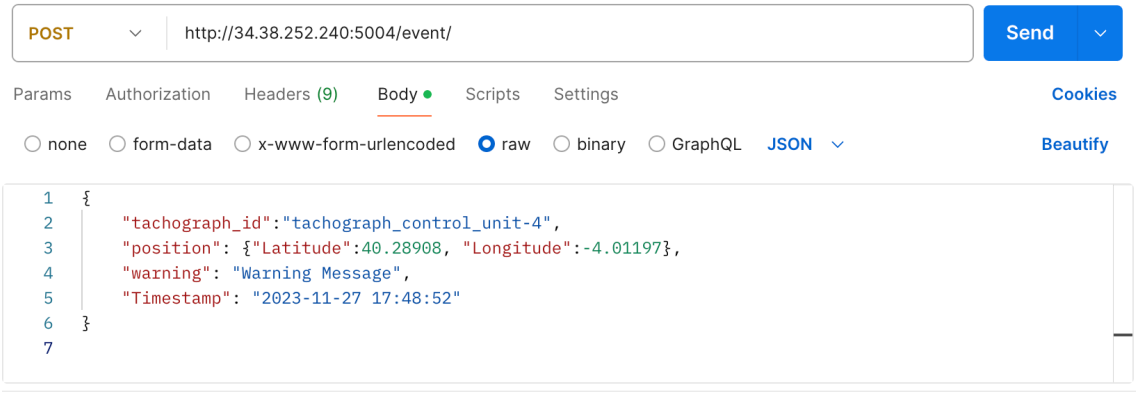
```
return result, 201
```

Prueba de Events Microservice

Una vez que se haya finalizado el desarrollo, se debe probar el correcto funcionamiento del código desarrollado.

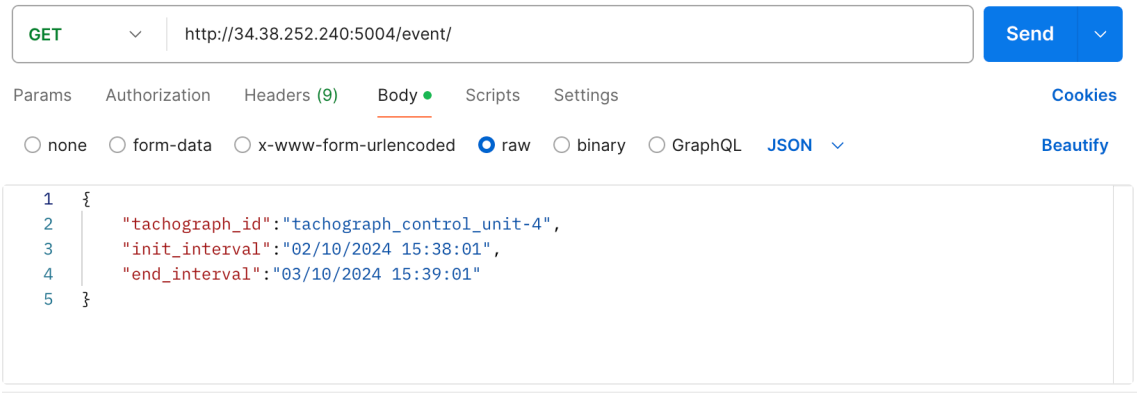
Los dos tipos de prueba a realizar con Postman son los siguientes:

- @app.route('/event/', methods=['POST'])



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- @app.route('/event/', methods=['GET'])



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

Cuando finalice la verificación de que el código funciona como se espera, el código desarrollado se debe publicar en el repositorio de control de versiones.

Creación de la imagen con Dockerfile

Una vez que se ha probado que el código del microservicio funciona correctamente en el equipo de desarrollo, se procederá a la preparación para su despliegue con contenedores.

Para ello, en la carpeta *IoTCloudServices/microservices/events_microservice* se creará el fichero Dockerfile.

El contenido de este fichero tendrá que permitir lo siguiente:

- Crear la imagen del contenedor a partir de la correspondiente a python:3.12
- Copiar el código que está en la carpeta *./code* a la carpeta */etc/usr/src/app*
- Establecer esta carpeta como directorio de trabajo.
- Instalar los paquetes necesarios especificados en requirements.txt. Los paquetes necesarios son: *Flask*, *Flask-Cors* y *mysql-connector-python,* para ello ejecutar pip install <paquetes necesarios>
- Ejecutar el código incluido en el fichero *events_manager_api.py*

Configuración de la orquestación del microservicio

Para ello, en la carpeta *IoTCloudServices* se modificará el fichero denominado *docker-compose.yml*. El nuevo contenido de este fichero debe incluir la definición de la orquestación del servicio events_microservice de acuerdo con el siguiente código:

```
events_microservice:
  build: ./microservices/events_microservice
  ports:
    - '5004:5004'
  links:
    - "dbservice:dbservice"
  environment:
    - HOST=0.0.0.0
    - PORT=5004
    - DBHOST=dbservice
    - DBUSER=fic_db_user
    - DBPASSWORD=secret1234
    - DBDATABASE=fic_data
    - PYTHONUNBUFFERED=1
  volumes:
    - "./microservices/events_microservice/code:/etc/usr/src/app"
  depends_on:
    - dbservice
```

Despliegue del microservicio

Para desplegar las versiones actualizadas del gemelo digital del simulador de vehículos en un contenedor Docker, se recomienda utilizar Docker Compose porque permitirá orquestar este servicio con los otros que se van a ir desplegando en la máquina virtual *fic-cservices*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-services*. A continuación, se debe clonar el repositorio correspondiente al código de esta sesión en esta máquina virtual.

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose build

docker compose up -d
```

Para verificar el despliegue, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor>
```

Sección 8

Actualización del servicio Message Router

Para actualizar el servicio denominado Message Router será necesario, en primer lugar, desarrollar y probar el código de su funcionamiento en el entorno personal de desarrollo del estudiante y, posteriormente, utilizar el fichero *Dockerfile* y *docker-compose.yml* desarrollado en el ejercicio anterior para su despliegue y orquestación con el resto de servicios considerados para la arquitectura de nube.

Desarrollo del conector para el registro de una nueva sesión

En el método *register_tachograph_connection(tachograph_id, tachograph_hostname)*, se tendrá que modificar el funcionamiento que se había implementado en la sesión anterior.

El código a implementar debería ser similar al que se ofrece a continuación.

```
    session_id = ""
    data = {"tachograph_id": tachograph_id, "tachograph_hostname": tachograph_hostname}
    print("{} - Solicitando creación de sesión para: {}".format( *args: datetime.datetime.now(), json.dumps(data)))
    host = os.getenv('SESSIONS_MICROSERVICE_ADDRESS')
    port = os.getenv('SESSIONS_MICROSERVICE_PORT')
    r = requests.put('http://' + host + ':' + port + '/sessions/', json=data)
    if r.status_code == 201:
        print("{} - Tacógrafo conectado".format(datetime.datetime.now()))
        complete_result = r.json()
        session_id = complete_result["session_id"]
    return session_id
```

La variable de entorno SESSIONS_MICROSERVICE_ADDRESS indica la URL donde se encuentra publicado el microservicio.

La variable de entorno SESSIONS_MICROSERVICE_PORT indica el puerto donde se encuentra publicado el microservicio.

Estas variables de entorno se configurarán en el fichero *docker-compose.yml* cuando se orquesten los distintos servicios de la arquitectura de nube.

Desarrollo del conector con el microservicio de gestión de telemetrías

En el método *register_tachograph_disconnection(tachograph_id)*, se tendrá que modificar el funcionamiento que se había implementado en la sesión anterior.

El código a implementar debería ser similar al que se ofrece a continuación.

```
data = {"tachograph_id": tachograph_id}
host = os.getenv('SESSIONS_MICROSERVICE_ADDRESS')
port = os.getenv('SESSIONS_MICROSERVICE_PORT')
r = requests.post('http://' + host + ':' + port + '/sessions/',
json=data)
```

La variable de entorno SESSIONS_MICROSERVICE_ADDRESS indica la URL donde se encuentra publicado el microservicio.

La variable de entorno SESSIONS_MICROSERVICE_PORT indica el puerto donde se encuentra publicado el microservicio.

En caso de obtener un resultado 201 del microservicio, se procederá a eliminar la suscripción a los canales de telemetría y eventos que el Message Router se había suscrito anteriormente.

Estas variables de entorno se configurarán en el fichero *docker-compose.yml* cuando se orquesten los distintos servicios de la arquitectura de nube.

Desarrollo del conector para el registro de telemetría

En el método *store_telemetry(data)*, se tendrá que modificar el funcionamiento que se había implementado en la sesión anterior.

El código a implementar debería ser similar al que se ofrece a continuación.

```
host = os.getenv('TELEMETRY_MICROSERVICE_ADDRESS')
port = os.getenv('TELEMETRY_MICROSERVICE_PORT')
r = requests.post('http://' + host + ':' + port + '/telemetry/',
json=data)
```

La variable de entorno TELEMETRY_MICROSERVICE_ADDRESS indica la URL donde se encuentra publicado el microservicio.

La variable de entorno TELEMETRY_MICROSERVICE_PORT indica el puerto donde se encuentra publicado el microservicio.

Estas variables de entorno se configurarán en el fichero *docker-compose.yml* cuando se orquesten los distintos servicios de la arquitectura de nube.

Desarrollo del conector para el registro de telemetría

En el método *store_event(data)*, se tendrá que modificar el funcionamiento que se había implementado en la sesión anterior.

El código a implementar debería ser similar al que se ofrece a continuación.

```
host = os.getenv('EVENTS_MICROSERVICE_ADDRESS')
port = os.getenv('EVENTS_MICROSERVICE_PORT')
r = requests.post('http://' + host + ':' + port + '/event/',
json=data)
```

La variable de entorno EVENTS_MICROSERVICE_ADDRESS indica la URL donde se encuentra publicado el microservicio.

La variable de entorno EVENTS_MICROSERVICE_PORT indica el puerto donde se encuentra publicado el microservicio.

Estas variables de entorno se configurarán en el fichero *docker-compose.yml* cuando se orquesten los distintos servicios de la arquitectura de nube.

Creación de la imagen con Dockerfile

El fichero Dockerfile que se utilizará para el Message Router en este ejercicio es igual al utilizado en el ejercicio anterior. Sin embargo, es necesario añadir la dependencia requests para poder utilizar la librería para la realización de peticiones http.

Orquestación de los servicios con Docker Compose

Para ello, en la carpeta *IoTCloudServices* se modificará el fichero denominado *docker-compose.yml*. El nuevo contenido de este fichero debe incluir la definición de la orquestación del servicio message_router de acuerdo con el siguiente código:

```
message_router:
  build: ./message_router
  environment:
    - MQTT_SERVER_ADDRESS=mosquitto
    - MQTT_SERVER_PORT=1883
    - HOST=0.0.0.0
    - PORT=5000
    - SESSIONS_MICROSERVICE_ADDRESS=sessions_microservice
    - SESSIONS_MICROSERVICE_PORT=5002
```

```
    - TELEMETRY_MICROSERVICE_ADDRESS=telemetry_microservice
    - TELEMETRY_MICROSERVICE_PORT=5003
    - EVENTS_MICROSERVICE_ADDRESS=events_microservice
    - EVENTS_MICROSERVICE_PORT=5004
    - PYTHONUNBUFFERED=1
  volumes:
    - "./message_router/code:/etc/usr/src/app"
  depends_on:
    - mosquitto
```

Despliegue de la imagen con Docker Compose

Para desplegar el componente message_router es necesario conectarse por *ssh* a la máquina *fic-services* creada anteriormente.

A continuación, se debe obtener la última versión del código de esta sesión que se encuentra en el repositorio en esta máquina virtual mediante un comando *git pull*.

Una vez que se haya obtenido en la máquina virtual el código correspondiente a esta sesión, hay que desplazarse a la carpeta *IoTCloudServices* y lanzar los servicios de esta máquina ejecutando los comandos:

```
docker compose stop

docker compose build

docker compose up -d
```

Para verificar que el Message Router se encuentra en ejecución, se pueden ejecutar los comandos siguientes:

```
docker ps -a

docker logs <nombre del contenedor del message router>
```

Sección 9

Actualización del servicio Message Router y del microservicio de gestión de tacógrafos

Para que Message Router funcione como un microservicio, se debe actualizar su funcionamiento, eliminando el código que aleatoriamente cambiaba los parámetros de frecuencia de muestreo y de envío de telemetrías e introducir el código que permita recibir las modificaciones de estos parámetros de configuración a través de una petición HTTP REST. Posteriormente, será necesario actualizar los elementos incluidos en el fichero *Dockerfile* y *docker-compose.yml* desarrollado en el ejercicio anterior para la correcta ejecución de los elementos de código introducidos.

Retirada del código anterior

El código que se ha desarrollado en la sesión anterior correspondiente a la determinación aleatoria y envío de los valores de frecuencia de muestreo y de envío de telemetrías que es similar al que se coloca de ejemplo, debe eliminarse.

```
while True:
    for tachograph in connected_tachographs:
        telemetry_frequency = random.randint( a: 5,  b: 60)
        send_configuration (client, tachograph["tachograph_id"],tachograph["tachograph_hostname"],
                            config_item: "telemetry_frequency",
                            telemetry_frequency)
        sensors_frequency = random.randint( a: 30,  b: 200) / 100.0
        send_configuration (client, tachograph["tachograph_id"],tachograph["tachograph_hostname"],
                            config_item: "sensors_frequency",
                            sensors_frequency)
    time.sleep(random.randint( a: 5, b: 15))
```

Configuración de Message Router como microservicio con API REST

En la carpeta *IoTCloudServices/message_router/code* se tiene que actualizar el código incluido en el fichero denominado *message_router.py*.

Los elementos de código que se deben incluir en la versión actualizada de Message Router son los siguientes:

1. Se tienen que importar todas las librerías correspondientes al framework de Flask y sus extensiones como con los microservicios desarrollados en las secciones anteriores.
2. Se tienen que crear el esqueleto de la aplicación Flask de acuerdo con el esquema presentado en la sección 4. Sin embargo, en este caso, es necesario configurar la URL y el puerto por el cual se publicará la API REST. Las variables de entorno HOST y PORT nos indicarán cuál es la dirección y puerto en el que se publicará la API REST. HOST debe tomar el valor 0.0.0.0 y PORT debe tomar el valor 5000.
3. La API REST tendrá el siguiente método:
   - @app.route('/tachographs/params/', methods=['POST'])

     Este método recibe como parámetros de entrada los datos de configuración para el tacógrafo seleccionado.

```
{
    "tachograph_id":"tachograph_control_unit-2",
    "telemetry_rate": 2,
    "sensors_sampling_rate":2.0
}
```

     El procesamiento de los datos recibidos se hará publicando los datos la ruta recibida por el correspondiente topic:

```
params = request.get_json()
route = {"Origin": params["Origin"], "Destination":
params["Destination"]}
client.publish("/fic/vehicles/" + params["Plate"] + "/routes",
               payload=json.dumps(route), qos=1, retain=False)
```

     En caso de que el envío se haya podido realizar correctamente, se devolverá un mensaje informativo, junto con el código de éxito 201:

```
return {"Result": "Route successfully sent"}, 201
```

     En caso contrario, se debe proporcionar un mensaje de error y el código de error 500 de http:

```
return {"Result": "Internal Server Error"}, 500
```

Aspectos a tener en cuenta con respecto a Flask-CORS

Para que el servidor Flask pueda ejecutarse al mismo tiempo que se realizan las comunicaciones a través del protocolo MQTT es necesario que se ejecute la aplicación Flask de la siguiente manera:

```
app.run(host=API_HOST, port=API_PORT, debug=True, use_reloader=False)
```

Creación de la imagen con Dockerfile

El fichero Dockerfile que se utilizará para el Message Router en este ejercicio es igual al utilizado en el ejercicio anterior.

Ampliación del microservicio de Gestión de Tacógrafos

Para realizar esta ampliación, es necesario modificar los ficheros *devices_manager_api.py* y *devices_db_manager.py*. El primero se utilizará para actualizar la interfaz HTTP expuesta por el microservicio y el segundo incluirá la implementación del nuevo método considerado.

1. La API REST tendrá el siguiente método:
   - @app.route('/tachographs/params/', methods=['POST'])

     Este método recibe como parámetros de entrada los datos de configuración para el tacógrafo seleccionado.

     ```
     {
         "tachograph_id":"tachograph_control_unit-2",
         "telemetry_rate": 2,
         "sensors_sampling_rate":2.0
     }
     ```

     El procesamiento de los datos recibidos se hará por un método denominado *save_tachograph_config*, que se implementará en el fichero *devices_db_manager.py*.

     Este método comprobará si el tacógrafo está activo. En caso de no estarlo se producirá un mensaje de error indicando que no es posible modificar la configuración del tacógrafo. En caso afirmativo, se obtendrá el hostname que está utilizando el tacógrafo en esta sesión y se actualizará en la base de datos los parámetros correspondientes a la frecuencia de muestro y la frecuencia de envío de telemetrías.

     En caso de que exista algún error durante este proceso, el método *save_tachograph_config* devolverá un hostname y una información de configuración del tacógrafo vacía.

     El método de API REST, a partir de los resultados proporcionados por *save_tachograph_config* enviará la nueva configuración registrada a través de la API REST de Message Router, utilizando un código similar al siguiente:

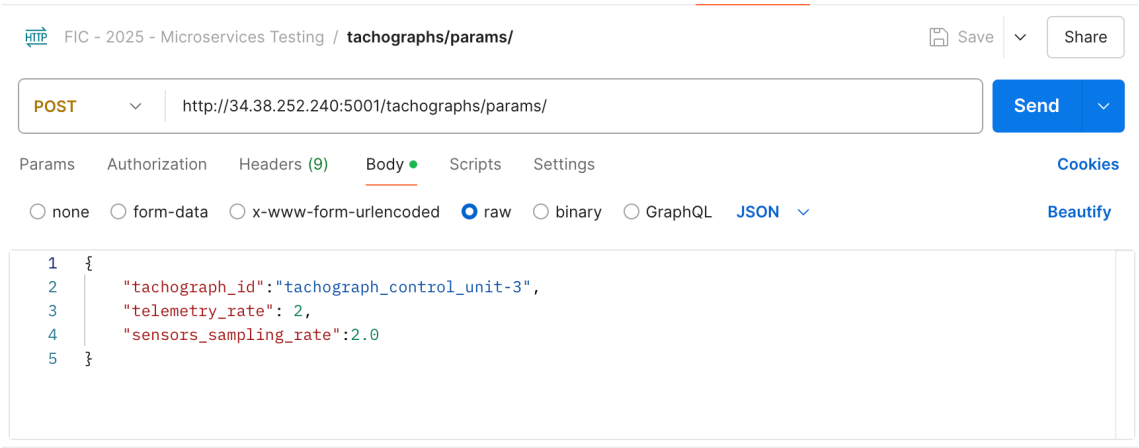
```
if configuration != {}:
    host = os.getenv('MESSAGE_ROUTER_ADDRESS')
    port = os.getenv('MESSAGE_ROUTER_PORT')
    r = requests.post('http://' + host + ':' + port +
'/tachographs/params/', json=configuration)
    if r.status_code == 201:
        return {"result": "Success: Tachograph params updated"}, 201
    else:
        return {"result": "Error: Tachograph params not updated"}, 500
else:
    return {"result": "Error: Tachograph params not updated"}, 500
```

Prueba de Devices Microservice

Una vez que se haya finalizado el desarrollo, se debe probar el correcto funcionamiento del código desarrollado.

Los dos tipos de prueba a realizar con Postman son los siguientes:

- @app.route('/tachographs/params/', methods=['POST'])



El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

Sección 10

Criterios de evaluación y procedimiento de entrega

Criterios de Evaluación

- Database Service - Correcta configuración y lanzamiento – 0,5 puntos
- Devices Microservice – 2 puntos
- Sessions Microservice – 2 puntos
- Telemetry Microservice – 2 puntos
- Events Microservice – 2 puntos
- Message Router – Conexión con los microservicios – 1,5 puntos

Entrega de la solución del ejercicio guiado

La entrega se realizará de dos maneras:

1) **Código Fuente**. En el repositorio de código de la asignatura tendrá en el proyecto Sesion10 correspondiente al grupo de prácticas los ficheros de código con la organización en directorios y nomenclatura de los ficheros que se han indicado a lo largo de este guion.
2) **Video demostrativo.** En una tarea Kaltura Capture Video habilitada para este ejercicio se entregará un vídeo que demuestre el correcto funcionamiento de la solución elaborada.

   Para este programa, se debe proporcionar una breve explicación del código generado (código en Python, dockerfile y docker compose) para Microservicio de Vehículos, Microservicio de Telemetría, Message Router y Gemelo Digital, mostrar el lanzamiento de la ejecución de la solución en la GCP y la visualización de las trazas con los resultados.

La fecha límite para la entrega del ejercicio es 08/05/2025 a las 23:59 h.

De acuerdo con las normas de evaluación continua en esta asignatura, si un estudiante no envía la solución de un ejercicio antes de la fecha límite, el ejercicio será evaluado con 0 puntos.

Cada estudiante debe guardar una copia de la solución entregada hasta la publicación de las calificaciones finales de la asignatura.