



FUNDAMENTOS DE INTERNET DE LA COSAS

2023-2024

Universidad Carlos III de Madrid

Cuaderno 2- Ejercicio - 10

2023/2024

MICROSERVICIOS PARA IOT - REGISTRAR
VEHICULOS Y TELEMETRIAS

Cuaderno 2 - Ejercicio 10

Universidad Carlos III de Madrid. Escuela Politécnica Superior

Objetivos

El propósito de este ejercicio consiste en:

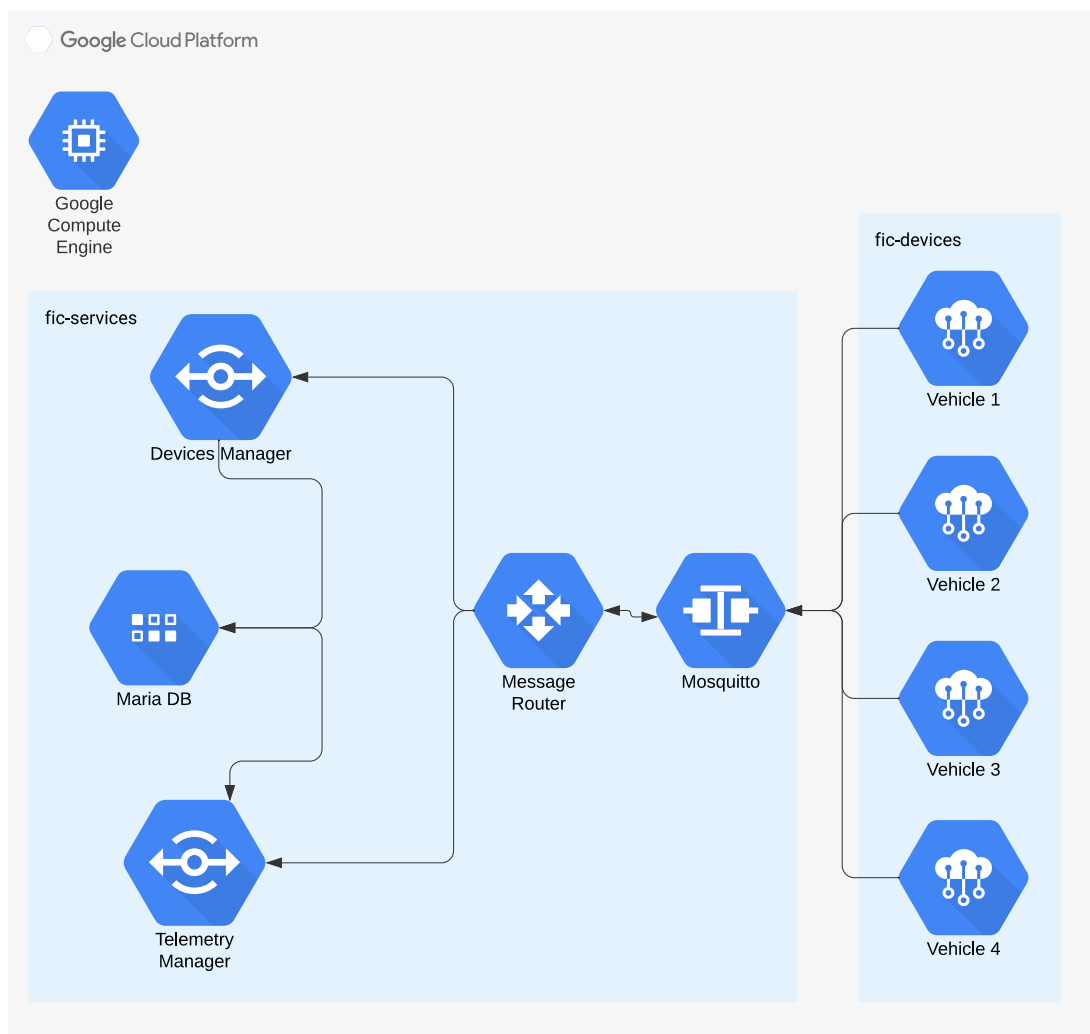
- Poner en práctica los conceptos necesarios para desarrollar microservicios que implementen los componentes de nube en IoT.
- Generar un contenedor con un gestor de base de datos para almacenar los datos operativos de una plataforma IoT.
- Desarrollar un microservicio para registrar los vehículos que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

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:

- ▼  Sesion 10
 - ▼  IoTCloudServices
 - ▼  dbService
 -  Dockerfile
 -  initial_script.sql
 - ▼  message_router
 - ▼  code
 -  message_router.py
 -  requirements.txt
 -  telemetry_register_interface.py
 -  vehicle_register_interface.py
 -  Dockerfile
 - ▼  microservices
 - ▼  telemetry_microservice
 - ▼  code
 -  requirements.txt
 -  telemetry_db_manager.py
 -  telemetry_manager_api.py
 -  Dockerfile
 - ▼  vehicles_microservice
 - ▼  code
 -  requirements.txt
 -  vehicles_db_manager.py
 -  vehicles_manager_api.py
 -  Dockerfile
 - ▼  mosquitto
 - ▼  code
 -  mosquitto.conf
 -  Dockerfile
 -  docker-compose.yml
 - ▼  VirtualVehicles
 - ▼  VehicleDigitalTwin
 - ▼  code
 -  requirements.txt
 -  VehicleDigitalTwin.py
 -  Dockerfile
 -  docker-compose.yml

El código se estructura en dos carpetas: *IoTCloudServices* su código se ejecutará en la máquina *fic-cloud-services*) y *VirtualVehicles* (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 vehículos 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:
 - *vehicles_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 (*vehicles_manager_api.py* y *vehicles_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.

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-cloud-services*.

- En *VirtualVehicles* 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 *VirtualVehicles* 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

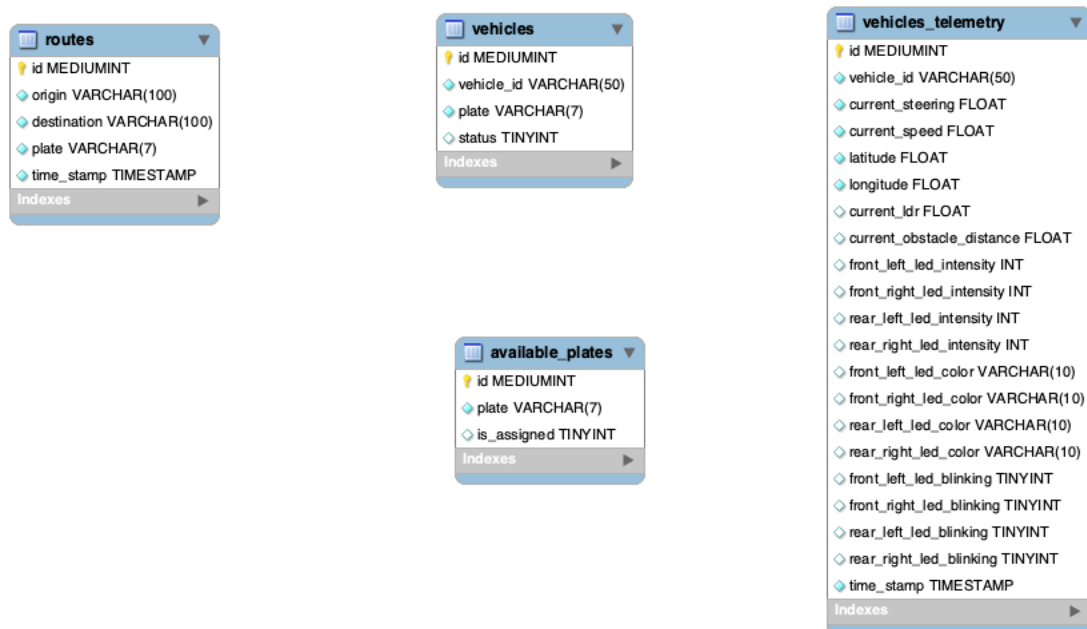
En primer lugar, es necesario clonar el proyecto en gitlab (<https://teaching.sel.inf.uc3m.es>) correspondiente a la sesión 10.

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.

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 Foreign 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 de los ejercicios 10 y 11 del cuaderno 2.

En el ámbito del ejercicio 10 solamente se utilizarán las tablas *available_plates*, *vehicles* y *vehicles_telemetry*.

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 contenedor 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> mysql -u root -  
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 plate, is_assigned FROM available_plates ORDER BY plate DESC;
```

Desarrollo del microservicio para la gestión de los vehículos

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

¿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 rutas¹.

El decorador router nos 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

La interfaz API REST que se implementará con Flask para el microservicio `vehicles_microservice`, se incluirá en el fichero `vehicles_manager_api.py`.

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

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.

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

```
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('/vehicles/register/', methods=['POST'])`

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

```
{"vehicle_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_vehicle*, que se implementará en el fichero *vehicles_db_manager.py*. Su lógica se especifica en el siguiente apartado.

En caso de que los datos proporcionados sean correctos y se hayan podido realizar la inserción en la base de datos, se devolverá la matrícula asignada, junto con el código de éxito `201`:

```
return {"Plate": plate}, 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 vehicle"}, 500
```

- `@app.route('/vehicles/retrieve/', methods=['GET'])`

Este método no recibe parámetros de entrada y devuelve como resultado una lista en formato JSON con todos los IDs y las matrículas 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_vehicles*, que se implementará en el fichero *vehicles_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 table vehículos 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 plate in myresult:  
        data = {"Plate": plate}  
        plates.append(data)  
    mydb.commit()
```

`sql` es una variable que contiene la query a ejecutar.

Este código, para cada una de las matrículas devuelta en la consulta, las agrega a una estructura de datos en JSON.

- `register_new_vehicle(params)`

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

```
SELECT plate FROM vehicles WHERE vehicle_id = %s ORDER BY plate ASC LIMIT 1;
```

- En caso de que el vehículo ya tenga matrícula asignada, se devolverá esta.
- En caso de que el id no tenga matrícula asignada, se ejecutará la siguiente consulta para conseguir una:

```
SELECT plate, is_assigned FROM available_plates WHERE is_assigned = 0 ORDER BY plate ASC LIMIT 1;
```

En caso de que se obtenga una matrícula, se procederá a insertar en la BD la matrícula para el ID recibido

```
INSERT INTO vehicles (vehicle_id, plate) VALUES (%s, %s);
```

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

```
UPDATE available_plates SET is_assigned = 1 WHERE plate = %s;
```

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

Prueba de Vehicle 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:

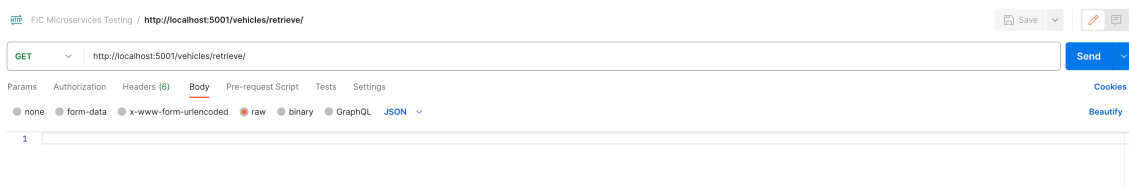
- `@app.route('/vehicles/register/', methods=['POST'])`



Se utiliza localhost porque se están realizando las pruebas en local. Cuando se realicen las pruebas sobre la máquina virtual *fic-cloud-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('/vehicles/retrieve/', methods=['GET'])`



Se utiliza localhost porque se están realizando las pruebas en local. Cuando se realicen las pruebas sobre la máquina virtual *fic-cloud-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.

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/vehicles_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 *vehicles_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 *message_router* de acuerdo con el siguiente código:

```
vehicles_microservice:
  build: ./microservices/vehicles_microservice
  ports:
    - '5001:5001'
  links:
    - "dbService:dbService"
  environment:
    - HOST=0.0.0.0
    - PORT=5001
    - DBHOST=dbService
    - DBUSER=fic_db_user
    - DBPASSWORD=RP#64nY7*E*H
    - DBDATABASE=fic_data
  volumes:
    - ".:/microservices/vehicles_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-cloud-services*.

Posteriormente, es necesario conectarse por *ssh* a la máquina *fic-cloud-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
```

MICROSERVICIOS PARA IOT - REGISTRAR VEHICULOS Y TELEMETRIAS

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

```
docker ps -a
```

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

Desarrollo del microservicio para la gestión de telemetrías

En la carpeta *IoTCloudServices/microservices/vehicles_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. 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.

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

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

```
{
  "vehicle_id": "1234BBC",
  "current_steering": 0.0,
  "current_speed": 0.0,
  "current_position": {"Latitude": 40.28908, "Longitude": -4.01197},
  "current_ldr": 0.0,
  "current_obstacle_distance": 1000.0,
```

```

        "current_leds": [{"Intensity": 0.0,
        "Color": "white", "Blinking": 0}, {"Intensity": 0.0,
        "Color": "white", "Blinking": 0}, {"Intensity": 0.0,
        "Color": "red", "Blinking": 0}, {"Intensity": 0.0,
        "Color": "red", "Blinking": 0}],
        "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 *telemetry_retriever*, que se implementará en el fichero *telemetry_db_manager.py*. Su lógica se especifica en el siguiente apartado.

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

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

```

{"vehicle_id": "<id>"}

```

Este método devuelve como resultado una lista en formato JSON con las últimas telemetrías correspondientes al vehículo que se indica como parámetro de entrada o un mensaje de error en caso de que haya ocurrido alguna incidencia en el procesamiento de la petición recibida.

```

if result["Error Message"] is None:
    return result, 201
else:
    return result, 500

```

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

- `@app.route('/telemetry/vehicle/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.

```
if result["Error Message"] is None:
    return result, 201
else:
    return result, 500
```

El procesamiento de los datos recibidos se hará por un método denominado *get_vehicles_last_position*, que se implementará en el fichero *telemetry_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 telemetrías son los siguientes:

- `connect_database()`. Se emplea el mismo código que el especificado para el microservicio anterior.
- `register_new_telemetry(params)`

Este método, después de la llamada para la conexión a la base de datos, inserta en la tabla de telemetrías los valores recibidos en el parámetro de entrada. En caso de que el resultado sea correcto, se devuelve True y, en caso de que haya algún error en la inserción de la telemetría, este método devolverá el resultado False.

- `get_vehicle_detailed_info()`

Este método, después de la llamada para la conexión a la base de datos, debe ejecutar una consulta para la obtención de las últimas telemetrías.

```
SELECT vehicle_id, current_steering, current_speed, current_ldr,
current_obstacle_distance, front_left_led_intensity,
front_right_led_intensity, rear_left_led_intensity,
rear_right_led_intensity, front_left_led_color,
front_right_led_color, rear_left_led_color, rear_right_led_color,
front_left_led_blinking, front_right_led_blinking,
rear_left_led_blinking, rear_right_led_blinking, time_stamp FROM
vehicles_telemetry WHERE vehicle_id = %s ORDER BY time_stamp LIMIT
20"
```

Una vez ejecutada la consulta, si no se han producido errores (si los hubiera, se debería proporcionar como resultado un mensaje indicativo), se debe agregar en formato JSON los datos de las telemetrías obtenidas de la consulta realizada.

```
vehicle_id = params["vehicle_id"]
query_params = (vehicle_id, )
```

```
my_cursor.execute(sql, query_params)
my_result = my_cursor.fetchall()
for vehicle_id, current_steering, current_speed, current_ldr,
current_obstacle_distance, front_left_led_intensity,
front_right_led_intensity, rear_left_led_intensity,
rear_right_led_intensity, front_left_led_color,
front_right_led_color, rear_left_led_color, rear_right_led_color,
front_left_led_blinking, front_right_led_blinking,
rear_left_led_blinking, rear_right_led_blinking, time_stamp in
my_result:
    item = {"Vehicle_id": vehicle_id, "Current Steering":
current_steering, "Current Speed": current_speed, "Current LDR":
current_ldr, "Obstacle Distance": current_obstacle_distance, "Front
Left Led Intensity": front_left_led_intensity, "Front Right Led
Intensity": front_right_led_intensity, "Rear Left Led Intensity":
rear_left_led_intensity, "Rear Right Led Intensity":
rear_right_led_intensity, "Front Left Led Color":
front_left_led_color, "Front Right Led Color":
front_right_led_color, "Rear Left Led Color": rear_left_led_color,
"Rear Right Led Color": rear_right_led_color, "Front Left Led
Blinking": front_left_led_blinking, "Front Right Led Blinking":
front_right_led_blinking, "Rear Left Led Blinking":
rear_left_led_blinking, "Rear Right Led Blinking":
rear_right_led_blinking, "Time Stamp": time_stamp}
    result.append(item)
```

- get_vehicle_detailed_info()

Este método, después de la llamada para la conexión a la base de datos, realiza la consulta de la última posición de los vehículos activos.

```
SELECT vehicle_id, plate, latitude, longitude, time_stamp FROM
vehicles, vehicles telemetry WHERE
vehicles.vehicle_id=vehicles telemetry.vehicle_id AND status = 1
AND vehicle.MAX(time_stamp
```

Una vez ejecutada la consulta, si no se han producido errores (si los hubiera, se debería proporcionar como resultado un mensaje indicativo), se debe agregar en formato JSON los datos de la última posición de los vehículos activos.

```
my_cursor.execute(sql)
my_result = my_cursor.fetchall()
for vehicle_id, plate, latitude, longitude, time_stamp in
my_result:
    item = {"Vehicle_id": vehicle_id, "Plate": plate, "Latitude":
latitude, "Longitude": longitude, "Time Stamp": time_stamp}
    result.append(item)
```

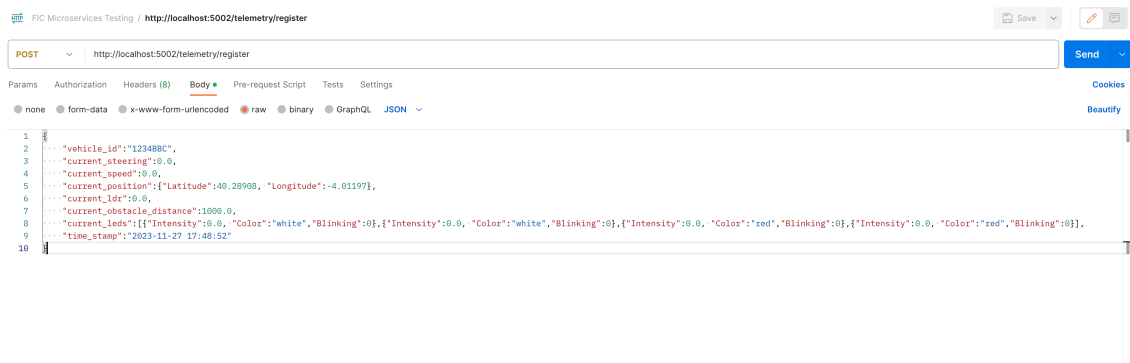
Prueba de Telemetry Microservice

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:

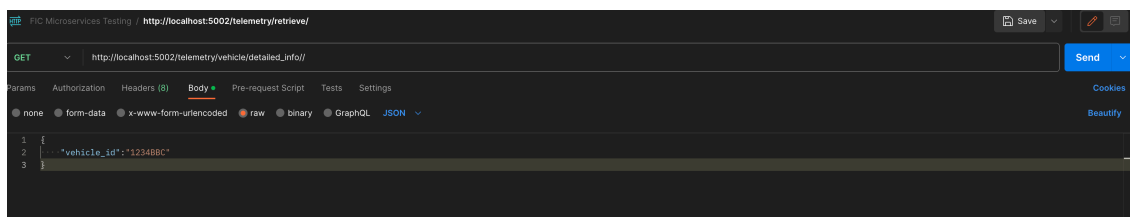
MICROSERVICIOS PARA IOT - REGISTRAR VEHICULOS Y TELEMETRIAS

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



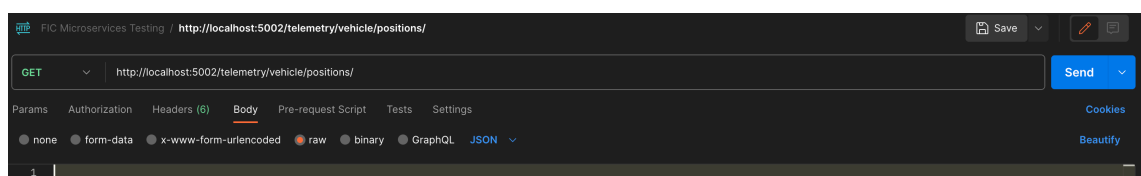
El valor resultante debe ser uno de los especificados para los casos en el apartado anterior de esta sección.

- `@app.route('/telemetry/vehicle/detailed_info/', 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:3.11.1
- 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 *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:
    - '5002:5002'
  links:
    - "dbService:dbService"
  environment:
    - HOST=0.0.0.0
    - PORT=5002
    - DBHOST=dbService
    - DBUSER=fic_db_user
    - DBPASSWORD=RP#64nY7*E*H
    - DBDATABASE=fic_data
  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-cloud-services*.

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

MICROSERVICIOS PARA IOT - REGISTRAR VEHICULOS Y TELEMETRIAS

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>
```

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 con el microservicio de gestión de vehículos

En la carpeta *IoTCloudServices/message_router/code* se tiene que crear un fichero que se denomine *vehicle_register_interface.py* en el que se incluya el siguiente método:

```
import requests
import os
def register_vehicle (data):
    host = os.getenv('VEHICLES_MICROSERVICE_ADDRESS')
    port = os.getenv('VEHICLES_MICROSERVICE_PORT')
    r = requests.post('http://' + host + ':' + port +
        '/vehicles/register', json=data)
```

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

La variable de entorno `VEHICLES_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 la carpeta *IoTCloudServices/message_router/code* se tiene que crear un fichero que se denomine *telemetry_register_interface.py* en el que se incluya el siguiente método:

```
import requests
import os

def register_telemetry (data):
    host = os.getenv('TELEMETRY_MICROSERVICE_ADDRESS')
    port = os.getenv('TELEMETRY_MICROSERVICE_PORT')
    r = requests.post('http://' + host + ':' + port +
        '/telemetry/register', 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 y prueba local del código de Message Router

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. En primer lugar, se deben importar los módulos que se han desarrollado en los apartados anteriores de esta sección.

```
from telemetry_register_interface import register_telemetry
from vehicle_register_interface import register_vehicle
```

2. El método *on_message* se tendrá que modificar para que se cambie el funcionamiento cuando se reciben mensajes de *request_plate* y *telemetry*:
 - En caso de recibir un mensaje en el topic *request_plate*, se tiene que invocar al método *register_vehicle* del *vehicle_register_interface* y si la matrícula recibida se envía al gemelo digital del simulador del vehículo.

```
input_data = json.loads(msg.payload.decode())
request_data = {"device_id":input_data["device_id"]}
vehicle_plate = register_vehicle(request_data)
plate_json = {"Plate":vehicle_plate}
client.publish("/fic/vehicles/" + msg.payload.decode() +
    "/config", payload=plate_json, qos=1, retain=False)
print("Publicado", vehicle_plate, "en TOPIC", msg.topic)
```

- En caso de recibir un mensaje en el topic *telemetry*, se tiene que invocar al método *register_telemetry* del *telemetry_register_interface*.

```
str_received_telemetry = msg.payload.decode()  
received_telemetry = json.loads(str_received_telemetry)  
register_telemetry (received_telemetry)
```

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.

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  
    - VEHICLES_MICROSERVICE_ADDRESS= vehicles_microservice  
    - VEHICLES_MICROSERVICE_PORT=5001  
    - TELEMETRY_MICROSERVICE_ADDRESS= telemetry_microservice  
    - TELEMETRY_MICROSERVICE_PORT=5002  
    - 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-cloud-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:

MICROSERVICIOS PARA IOT - REGISTRAR VEHICULOS Y TELEMETRIAS

```
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>
```

Criterios de evaluación y procedimiento de entrega



Criterios de Evaluación

- Database Service - Correcta configuración y lanzamiento – 2 puntos
- Vehicles Microservice – Registro de Vehículo – 1,5 puntos
- Vehicles Microservice – Consulta de Vehículo – 1,5 puntos
- Telemetry Microservice – Registro de Telemetría – 1,5 puntos
- Telemetry Microservice – Consulta de Telemetría – 1,5 puntos
- Message Router – Conexión con los microservicios – 2 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 Sesión10 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 09/05/2024 a las 23:59 h.

MICROSERVICIOS PARA IOT - REGISTRAR VEHICULOS Y TELEMETRIAS

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.



Sugerencias

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