Endpoint Attributes

***Resumen*—En este laboratorio, se implementó y configuró la plataforma Traccar utilizando Docker para la gestión de sus principales componentes: base de datos, servidor, y frontend. A partir de la clonación de un repositorio existente, se procedió a crear un entorno de contenedores aislados, lo que permitió una organización clara y segura del sistema. Mediante Docker Compose, se gestionó la construcción y despliegue de estos contenedores de manera ordenada, asegurando la correcta interacción entre los servicios.**

Además, se identificaron los componentes necesarios para la funcionalidad de la API, centrándose en el endpoint Attributes, ubicado en el contenedor del servidor. Se demostró la viabilidad de dockerizar este endpoint, lo cual facilita la escalabilidad y modularidad del sistema.

Finalmente, se obtuvieron conclusiones sobre la efectividad de Docker para gestionar aplicaciones complejas y recomendaciones para futuros proyectos, subrayando la importancia de una correcta segmentación de servicios y de pruebas exhaustivas en diferentes entornos. Este enfoque garantiza la replicabilidad y el mantenimiento a largo plazo de la infraestructura desarrollada.

I. Objetivos

A. Objetivo General

Configurar y desplegar un sistema de monitoreo GPS utilizando Traccar mediante contenedores Docker, asegurando la correcta comunicación entre los servicios de base de datos, servidor y web, y optimizando la gestión de redes.

B. Objetivos Específicos

- Clonar y configurar el repositorio de Traccar para su despliegue en un entorno de contenedores Docker, asegurando la correcta estructura de los servicios db, server y web.
- Configurar y gestionar redes personalizadas en Docker para permitir la comunicación eficiente entre los contenedores, evitando conflictos de direcciones IP y garantizando la conectividad.
- Analizar y probar la funcionalidad del endpoint de la API ”Attributes” dentro del contenedor del servidor, asegurando que los componentes de la API puedan ser dockerizados y funcionales en el entorno desplegado.g

II. INTRODUCCIÓN

En el presente laboratorio, se ha llevado a cabo la dockerización del endpoint Attributes de Traccar, un sistema de rastreo GPS ampliamente utilizado en soluciones de monitoreo de flotas y gestión de dispositivos. Traccar, desarrollado como una plataforma de código abierto, permite a los usuarios recibir, procesar y visualizar datos de ubicación provenientes de dispositivos GPS, lo cual es esencial para empresas que requieren un seguimiento en tiempo real de sus activos [1]. La dockerización de este componente específico tiene como objetivo principal comprender cómo los microservicios pueden ser desplegados y escalados en un entorno de contenedores, garantizando así un despliegue eficiente y consistente en diferentes entornos.

El proceso de dockerización involucra la creación de un contenedor que encapsula el código fuente, las dependencias y las configuraciones necesarias para ejecutar el endpoint Attributes de Traccar. Para lograr esto, se han utilizado herramientas como Docker y Docker Compose, las cuales permiten definir y gestionar de manera eficiente los servicios que componen la aplicación. La utilización de Docker no solo facilita la portabilidad del software entre diferentes entornos de desarrollo y producción, sino que también mejora la seguridad y la administración de los recursos del sistema [2]. A través de este laboratorio, se ha demostrado cómo los contenedores Docker pueden simplificar el proceso de despliegue, reduciendo las inconsistencias que suelen surgir al mover aplicaciones entre distintos entornos.

Además, se ha explorado la estructura interna del código fuente del endpoint Attributes para identificar los componentes clave que forman parte de la API. Este análisis ha permitido entender cómo interactúan los diferentes módulos del sistema y cómo se pueden optimizar estos servicios para un rendimiento óptimo en un entorno de contenedores [3]. La capacidad de dockerizar aplicaciones como Traccar ilustra la potencia y la flexibilidad de Docker como herramienta para el desarrollo moderno, permitiendo a los desarrolladores y administradores de sistemas desplegar aplicaciones de manera más rápida y con mayor confiabilidad.

III. Materiales

1) **Computador Personal:**
   - **Computador:** HP Pavilion Gaming Laptop 15-dk1xxx
   - **Procesador:** Intel(R) Core(TM) i5-10300H CPU @ 2.50GHz 2.50 GHz

- **Tipo de Sistema:** Sistema operativo de 64 bits, procesador x64
- **RAM:** 8,00 GB (7,78 GB utilizable)
- **SO:** Windows 11 Home Single Language
- **Versión:** 23H2

2) **Complementos generales necesarios:**
   - **IDE:** Code::Visual Studio Code
   - **IDE Versión:** 1.92.2 (user setup)
   - **Commit:** fee1edb8d6d72a0ddff41e5f71a671c23ed924b9
   - **Date:** 2024-08-14T17:29:30.058Z
   - **Electron:** 30.1.2
   - **ElectronBuildId:** 9870757
   - **Chromium:** 124.0.6367.243
   - **Node.js:** 20.14.0
   - **V8:** 12.4.254.20-electron.0
   - **OS:** Windows-NT x64 10.0.22631
   - **Máquina Virtual:** Virtual Box
   - **Versión Virtual Box:** 7.0.18 r162988 (Qt5.15.2)
   - **Framework FrontEnd:** Next.js (React)
   - DOCKER desktop
   - Cuenta en TRACCAR

IV. METODOLOGÍA

TRACCAR CON DOCKER:

El Readme describe el proceso para configurar y ejecutar Traccar utilizando Docker, aprovechando la capacidad de contenedores para gestionar tanto el backend como el frontend de la aplicación.

Requisitos:

Es necesario tener Docker y Docker Compose instalados en tu sistema.

Instrucciones de uso:

1) **Realizar un Fork del Repositorio:**
   El primer paso consiste en realizar un fork del repositorio de Traccar que contiene la configuración de Docker y los archivos necesarios. Esto permite obtener una copia personal del proyecto para realizar modificaciones y actualizaciones según las necesidades específicas.
2) **Crear las Redes para los Contenedores**
   Traccar se estructura en dos partes principales: el backend y el frontend. Para separar estas dos partes y mejorar tanto la organización como la seguridad, se crearán dos redes Docker específicas:
   - **Red para Backend**
     Ejecuta el siguiente comando para crear la red del backend: **docker network create –driver bridge –subnet=172.18.0.0/24 backend-network**
   - **Red para Frontend**
     Ejecuta el siguiente comando para crear la red del frontend **docker network create –driver bridge –subnet=172.19.0.0/24 frontend-network**
3) **Crear los Contenedores con Docker Compose**
   Para desplegar los contenedores de la base de datos, el servidor y la interfaz web de Traccar, se utiliza Docker Compose. Cada componente tiene su propio archivo `docker-compose.yaml` que define su configuración. Se recomienda construir en el siguiente orden: base de datos (db), servidor (server), y finalmente, la interfaz web (web).
   - Ejecuta el siguiente comando para levantar los contenedores:
     docker-compose -f ¡nombre de carpeta de contenedor¿/docker-compose.yaml up

   Espera a que la construcción finalice y repite el proceso para cada contenedor en el orden mencionado.

IDENTIFICAR EL ENDPOINT ATRRIBUTES:

Para identificar los componentes necesarios para que el endpoint Attributes funcione correctamente, se siguieron los siguientes pasos:

1) **Acceso al Contenedor Server**: Se comenzó accediendo al contenedor `server`, donde se encuentra alojado el código que maneja los endpoints de la API.
2) **Exploración de la Carpeta API**: Dentro del contenedor `server`, se navegó a la carpeta que contiene el código fuente de la API (`src/api`). Esta carpeta es donde se organizan los diferentes módulos y modelos que interactúan para definir los endpoints.
3) **Identificación del Modelo Attributes**: En la carpeta `src/api/resources`, se localizó el archivo `attributeResource.java`, que define la estructura y comportamiento de `Attribute`. Este archivo es crucial ya que contiene las propiedades y métodos que la API utiliza para manejar los atributos.
4) **Análisis del Código**: Se revisó el contenido del archivo `attributes.java` para entender cómo se implementan los métodos y propiedades del modelo `Attribute`. Este análisis permitió comprender qué partes del código son esenciales para que el endpoint `Attributes` funcione correctamente.
5) **Revisión de Dependencias**: Finalmente, se verificaron las posibles dependencias y configuraciones adicionales que podrían estar involucradas en el correcto funcionamiento del endpoint `Attributes`. Esto incluyó la inspección de otros archivos que podrían estar relacionados o que interactúan con el modelo `Attribute`.

Estos pasos permitieron identificar los componentes críticos que crean y soportan el funcionamiento del endpoint `Attributes` en la API.

V. RESULTADOS

Comprobamos el correcto funcionamiento de los contenedores con sus respectivas redes creadas. Despues levantamos los distintos contenedores, esto lo hacemos con los códigos de terminal antes mencionados y su ejecución se vería de la siguiente forma:

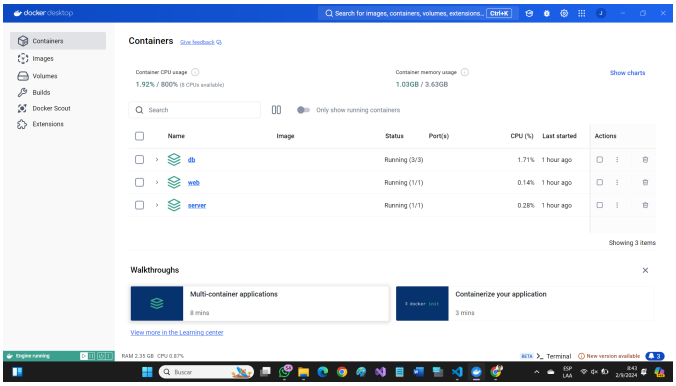

Fig. 1. Contenedores.

Las redes creadas (frontend y backend) las podemos observar usando el comando: **docker network ls.**

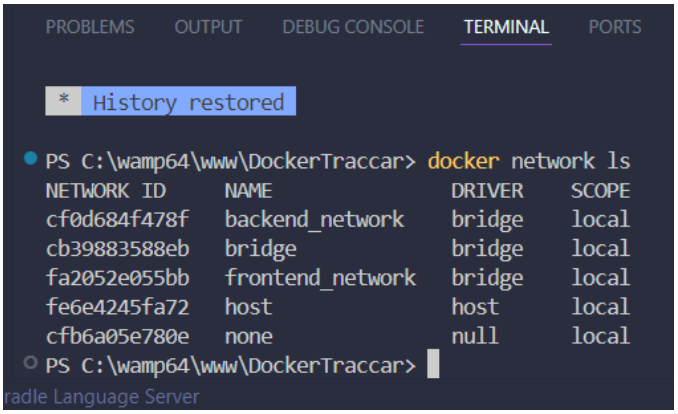

Fig. 2. Listado de Redes Creadas.

Ahora, si tenemos todo listo y corriendo de forma correcta, en el puerto 3000, estrá ejecutandose el contenedor web de traccar, esto lo podemos observar desde el navegador, ingresamos a la web local para ver los servicios de **TRACCAR:**

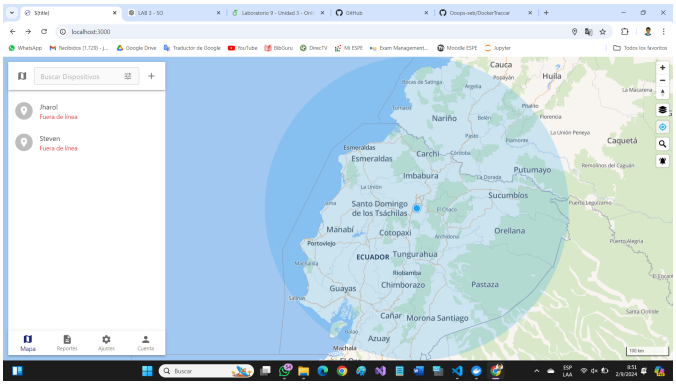

Fig. 3. Página web local de TRACCAR.

- **Docker-Compose.yaml de db:**

```
version: '3.8'

services:
  postgres_db:
    build:
      context: ./postgres
      dockerfile: Dockerfile
    container_name: postgres_db
    env_file:
      - .env
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
    volumes:
      - pg_data:/var/lib/postgresql/data
    networks:
      backend_network:
          ipv4_address: 172.18.0.10
    ports:
      - "5434:5432"
  mysql_db:
    build:
      context: ./mysql
      dockerfile: Dockerfile
    container_name: mysql_db
    env_file:
      - .env
    environment:
      MYSQL_ROOT_PASSWORD: ${MYSQL_ROOT_PASSWORD
    }
      MYSQL_USER: ${MYSQL_USER}
      MYSQL_PASSWORD: ${MYSQL_PASSWORD}
    networks:
      backend_network:
          ipv4_address: 172.18.0.11
    ports:
      - "3306:3306"
    volumes:
      - db_data:/var/lib/mysql
  mongo_db:
    build:
      context: ./mongo
      dockerfile: Dockerfile
    container_name: mongo_db
    networks:
      backend_network:
          ipv4_address: 172.18.0.12
    ports:
      - "27017:27017"
    env_file:
      - .env
    environment:
      MONGO_INITDB_ROOT_USERNAME: ${
    MONGO_ROOT_USERNAME}
      MONGO_INITDB_ROOT_PASSWORD: ${
    MONGO_ROOT_PASSWORD}
    volumes:
      - mongo_data:/data/db

volumes:
  pg_data:
  db_data:
  mongo_data:

networks:
  backend_network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/24
    external: true
```

Listing 1. docker de las Bases de Datos

- **Docker-Compose.yaml del server:**

```
version: "3.8"
```

```
services:
  server:
    build:
      context: ./
      dockerfile: Dockerfile
    container_name: server
    networks:
      backend_network:
        ipv4_address: 172.18.0.13
    ports:
      - "8082:8082"
    command: sh -c "java -jar target/tracker-server.jar"

networks:
  backend_network:
    driver: bridge
    ipam:
      config:
        - subnet: 172.18.0.0/24
    external: true
```

Listing 2. docker del Servidor Traccar

- **Docker-Compose.yaml de web:**

```
version: "3.8"

services:
  traccar:
    build:
      context: .
      dockerfile: Dockerfile
    env_file:
    - .env
    container_name: traccar
    working_dir: /app
    volumes:
      - .:/app
      - /app/node_modules
    networks:
      backend_network:
      frontend_network:
        ipv4_address: 172.19.0.10
    ports:
      - "3000:3000"

networks:
  frontend_network:
    driver: bridge
    external: true
    ipam:
      config:
        - subnet: 172.19.0.0/24

  backend_network:
    driver: bridge
    external: true
    ipam:
      config:
        - subnet: 172.18.0.0/24
```

Listing 3. docker de la Web de Traccar

ANÁLISIS DEL CÓDIGO

1) **Variables de Entorno:** Configuración Centralizada: Se ha utilizado archivos .env para almacenar configuraciones sensibles y parámetros específicos de cada aplicación, lo que ha simplificado la gestión de configuraciones. Estos archivos han permitido centralizar y gestionar de manera segura variables como credenciales de bases de datos y otros parámetros de entorno, evitando la necesidad de modificar el código fuente para ajustar configuraciones.
2) **Facilidad de Ejecución:** Con Docker Compose, se ha ejecutado todos los servicios del sistema con un solo comando (docker-compose up). Esta característica ha simplificado el proceso de despliegue y configuración, proporcionando una solución eficiente y rápida para iniciar y gestionar todos los contenedores simultáneamente, lo cual ha sido especialmente beneficioso en entornos de desarrollo y equipos de trabajo.
3) **Aislamiento de Entornos:** Cada componente del sistema ha operado en su propio contenedor, asegurando que las dependencias y configuraciones no entren en conflicto entre aplicaciones. Este aislamiento ha mejorado la estabilidad y seguridad, además de facilitar la solución de problemas y el mantenimiento de la aplicación a lo largo del tiempo.
4) En conjunto, esta arquitectura ha permitido una implementación robusta y escalable de aplicaciones, facilitando la integración de servicios y la gestión eficiente de recursos mediante el uso de contenedores y Docker Compose.

ANÁLISIS DEL ENDPOINT "ATTRIBUTES"

El código del archivo situado en la ruta específicada en la meotodología describe la implementación de un recurso REST en Java para manejar las operaciones CRUD relacionadas con los "Attributes" (atributos) computados en un sistema. Este código se encarga de gestionar los endpoints relacionados con la API attributes/computed. A continuación, desgloso los componentes clave que crean y permiten el funcionamiento de este endpoint.

Componentes Clave:

1) **Clase AttributeResource:**
   - Extiende **ExtendedObjectResource¡Attribute¿**, lo que implica que hereda métodos genéricos para manejar recursos basados en el tipo **Attribute**.
   - La clase está anotada con **@Path("attributes/computed")**, lo que define el endpoint base de la API para los atributos computados.
2) **Dependencias Inyectadas: ComputedAttributesHandler computedAttributesHandler**: Se inyecta para manejar la lógica de los atributos computados. Esta dependencia es crucial para la manipulación y procesamiento de atributos en el sistema.
3) **Métodos Principales:**
   - **test(@QueryParam("deviceId") long deviceId, Attribute entity)**:

  - Este método maneja una solicitud POST en la ruta **/attributes/computed/test**.
  - Verifica los permisos de administración y luego obtiene la última posición del dispositivo especificado por **deviceId**.
  - Usa **computedAttributesHandler** para calcular el valor del atributo basado en la posición del dispositivo.
  - Devuelve una respuesta HTTP con el resultado del cálculo o un contenido vacío si no se puede computar el atributo.
- **add(Attribute entity)**:
  - Maneja la solicitud POST en la ruta base **/attributes/computed**.
  - Verifica que el usuario tenga permisos de administrador antes de agregar un nuevo **Attribute** a la base de datos.
- **update(Attribute entity)**:
  - Maneja la solicitud PUT en la ruta **/attributes/-computed/id**.
  - Verifica los permisos de administrador antes de actualizar un **Attribute** existente.
- **remove(@PathParam("id") long id)**:
  - Maneja la solicitud DELETE en la ruta **/attributes/computed/id**.
  - Verifica los permisos de administrador antes de eliminar un **Attribute** existente.

Otros Componentes Importantes:

- **permissionsService**: Este servicio es utilizado para verificar los permisos del usuario que realiza las operaciones, asegurando que solo los administradores puedan modificar los atributos.
- **storage**: Este componente es utilizado para interactuar con la base de datos, en este caso para obtener la última posición del dispositivo.

Resumen de los Módulos que Necesita la API para Funcionar:

1) **Controlador (AttributeResource)**: Gestiona las solicitudes HTTP y las rutas del endpoint.
2) **Modelo (Attribute)**: Representa los datos de los atributos computados.
3) **Manejador (ComputedAttributesHandler)**: Contiene la lógica para calcular los atributos.
4) **Servicio de Permisos (permissionsService)**: Verifica que las acciones realizadas estén autorizadas.
5) **Almacenamiento (storage)**: Accede a la base de datos para realizar operaciones CRUD y obtener datos como las posiciones.

Modelo Attribute:

1) **Paquete y Licencia:**
   - El archivo está en el paquete `org.traccar.model`.
   - Incluye una licencia Apache 2.0, que establece los términos bajo los cuales se puede usar este código.
2) **Clase Attribute:** Extiende **BaseModel**, lo que sugiere que hereda funcionalidades básicas de un modelo base, posiblemente relacionado con operaciones CRUD (crear, leer, actualizar, eliminar).
3) **Anotación @StorageName("tc-attributes"):** La anotación **@StorageName** parece definir el nombre de la tabla en la base de datos a la que este modelo está asociado. En este caso, la tabla se llama **tc-attributes**.
4) **Propiedades y Métodos:**
   - **description:** Una cadena de texto que probablemente describe el atributo.
   - **attribute:** Una cadena de texto que parece ser el nombre del atributo.
   - **expression:** Una cadena de texto que podría representar una expresión o fórmula asociada con el atributo.
   - **type:** Una cadena de texto que especifica el tipo del atributo.
   - **priority:** Un entero que indica la prioridad del atributo.

   Cada propiedad tiene métodos **get** y **set**, lo que permite acceder y modificar sus valores.

Relación con el Endpoint Attributes

Este archivo define la estructura de los datos para el modelo **Attribute**. Los detalles que contiene son esenciales para las operaciones relacionadas con el endpoint **Attributes** de la API. Aquí está cómo se relaciona:

- **Interacción con la Base de Datos:** El modelo **Attribute** define cómo los datos del endpoint **Attributes** se almacenan y se recuperan de la base de datos. La anotación **@StorageName** sugiere que este modelo está vinculado a una tabla específica en la base de datos.
- **Lógica de Negocio:** Los métodos **get** y `set` permiten la manipulación de los datos del atributo. Esto es crucial para las operaciones de la API como la creación, actualización y consulta de atributos.
- **Integración con Otros Componentes:** Este modelo es probablemente utilizado por los controladores y servicios que gestionan las solicitudes para el endpoint **Attributes**. Los controladores usarán este modelo para acceder y modificar los datos que llegan a través de la API.

ALGORITMO DEL ENDPOINT

```
1 package org.traccar.api.resource;
2
3 import jakarta.inject.Inject;
4 import jakarta.ws.rs.Consumes;
```

```java
import jakarta.ws.rs.DELETE;
import jakarta.ws.rs.POST;
import jakarta.ws.rs.PUT;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.QueryParam;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

import org.traccar.api.ExtendedObjectResource;
import org.traccar.model.Attribute;
import org.traccar.model.Device;
import org.traccar.model.Position;
import org.traccar.handler.ComputedAttributesHandler;
import org.traccar.storage.StorageException;
import org.traccar.storage.query.Columns;
import org.traccar.storage.query.Condition;
import org.traccar.storage.query.Request;

@Path("attributes/computed")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class AttributeResource extends ExtendedObjectResource<Attribute> {

    @Inject
    private ComputedAttributesHandler computedAttributesHandler;

    public AttributeResource() {
        super(Attribute.class);
    }

    @POST
    @Path("test")
    public Response test(@QueryParam("deviceId") long deviceId, Attribute entity) throws StorageException {
        permissionsService.checkAdmin(getUserId());
        permissionsService.checkPermission(Device.class, getUserId(), deviceId);

        Position position = storage.getObject(Position.class, new Request(
                new Columns.All(),
                new Condition.LatestPositions(deviceId)));

        Object result = computedAttributesHandler.computeAttribute(entity, position);
        if (result != null) {
            return switch (entity.getType()) {
                case "number", "boolean" -> Response.ok(result).build();
                default -> Response.ok(result.toString()).build();
            };
        } else {
            return Response.noContent().build();
        }
    }

    @POST
    public Response add(Attribute entity) throws Exception {
        permissionsService.checkAdmin(getUserId());
        return super.add(entity);
    }

    @Path("{id}")
    @PUT
    public Response update(Attribute entity) throws Exception {
        permissionsService.checkAdmin(getUserId());
        return super.update(entity);
    }

    @Path("{id}")
    @DELETE
    public Response remove(@PathParam("id") long id) throws Exception {
        permissionsService.checkAdmin(getUserId());
        return super.remove(id);
    }

}
```

Listing 4. Algoritmo del Endpoint Atributte

¿Se puede Dockerizar el Endpoint Attributes?

Es posible dockerizar el endpoint **Attributes** porque Docker permite encapsular aplicaciones y sus dependencias en contenedores, asegurando que todo el entorno necesario para la ejecución del código esté presente y sea consistente en cualquier sistema donde se despliegue.

En este caso, el archivo **attributes.java**, ubicado en el contenedor server bajo src/api/model, define un modelo que es parte de la API. Al contenerizar la aplicación, se incluye todo el código fuente, configuraciones, y dependencias necesarias para que este endpoint funcione correctamente.

Esto facilita la replicación del entorno de desarrollo, pruebas, y producción de manera confiable y eficiente. Dockerizar el endpoint también permite escalar y administrar de forma sencilla los servicios asociados, mejorando la portabilidad y consistencia de la aplicación.

VI. Discusiones/Conclusiones

- El laboratorio permitió comprender a profundidad la utilidad y versatilidad de Docker para gestionar aplicaciones complejas como Traccar. La creación y configuración de contenedores específicos para el backend, frontend, y la base de datos no solo facilitó la segregación de responsabilidades, sino que también garantizó un entorno controlado y replicable para la ejecución del software. Además, el uso de Docker Compose simplificó significativamente la orquestación de estos contenedores, permitiendo un despliegue ordenado y eficiente de la aplicación en su totalidad.
- La identificación de componentes clave como el endpoint Attributes y su integración en la arquitectura de contenedores resaltó la capacidad de Docker para manejar microservicios de manera modular y escalable. Este enfoque modular también mejora la mantenibilidad y escalabilidad del sistema, ya que cada componente puede ser actualizado o reemplazado de manera independiente, sin afectar el resto de la infraestructura.

VII. Recomendaciones

- Para futuros proyectos que involucren la contenedorización de aplicaciones, es recomendable seguir una metodología similar, donde se segmentan las partes de la aplicación en diferentes contenedores según su función. Esto no solo mejora la organización del proyecto, sino que también facilita el diagnóstico de problemas y la implementación de mejoras.
- Además, se sugiere realizar pruebas exhaustivas en diferentes entornos para asegurarse de que los contenedores se comporten de manera consistente. Docker es una herramienta poderosa, pero su eficacia depende de una configuración adecuada y un entendimiento claro de cómo interactúan los diferentes componentes en contenedores aislados.
- Finalmente, es importante mantenerse actualizado con las mejores prácticas y novedades en el ecosistema Docker, ya que esto permitirá optimizar el uso de la herramienta y aprovechar al máximo sus capacidades en proyectos futuros.

Referencias

[1] A. Tananaev, "Traccar: Open source gps tracking system," 2017, gitHub Repository. [Online]. Available: https://github.com/traccar/traccar

[2] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, no. 239, p. 2, 2014.

[3] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.