

API REST CARRERA DE PODS

Por:
Pablo Felice

Versión 1.0

Tabla de contenido

Objetivo	3
Enlaces	3
Código Fuente en GitHub	3
Detalles	4
Nivel 1	4
Nivel 2	5
Nivel 3	5
Diagrama UML	6
Principal	6
Otras Clases e interfaces	7
Documentación adicional	8
Swagger	8
Nivel 4	9

Objetivo

El objetivo de esta API REST es proporcionar a los usuarios información precisa sobre la ubicación y las métricas de los pods que participan en las carreras. A través de esta API, los clientes podrán obtener las coordenadas de los Pods en la pista y los detalles de las métricas de las naves.

Además, la API está pensada para ser escalable y adaptable, lo que permite la incorporación de nuevas antenas para mejorar la precisión.

El objetivo final es proporcionar una solución tecnológica para el seguimiento y control de las “Carreras de Pods”, mejorando la seguridad de los competidores y la calidad de la experiencia de los espectadores.

Stack de tecnologías

- Java versión 17 - Spring Boot: 3.0.4 - Framework
- Apache Kafka (Confluent Cloud) - Publicar y consumir eventos
- Google Cloud (App Engine y Cloud Run) Se probó la implementación en ambas plataformas
- Swagger - Para documentar la API REST
- Herramientas (Git, Docker, Postman, Apache Jmeter, Offset Explorer, otras)

Enlaces

Código Fuente en GitHub: https://github.com/pablo228751/Carrera_de_Pods_ApiRest

Documentación Swagger: <https://pod-rxtzbm6vfq-rj.a.run.app/doc.html>

Enlaces:

- Endpoint /podhealth/: <https://pod-rxtzbm6vfq-rj.a.run.app/podhealth/>
- Endpoint /podhealth_split/{antena_name}: https://pod-rxtzbm6vfq-rj.a.run.app/podhealth_split/antena1
- Endpoint /podhealth_split/{pod_name}: https://pod-rxtzbm6vfq-rj.a.run.app/podhealth_split/Anakin_Skaywoker

Nivel 1

Para el Nivel 1 se creó la clase “LocationAndMessage” que contiene métodos para calcular la ubicación de un vehículo en un plano bidimensional, utilizando la información recibida de las antenas.

El constructor recibe una lista de objetos antena, y crea un vector de tipo lista con las distancias, esto con la finalidad de establecer la cantidad de antenas presentes y su ubicación, datos necesarios para los cálculos de trilateración y triangulación (para ello se creó la clase “CrearAntenas” con la idea de aplicar paradigma Orientado a Objetos permitiendo agregar más antenas).

El método **getLocation** recibe una lista de distancias desde cada antena hasta el “Pod”, y utiliza estas distancias junto con las coordenadas de las antenas para calcular la ubicación del Pod en un plano bidimensional. Si la cantidad de antenas es mayor a 3, llama al método getLocation2.

El método getLocation2 es utilizado cuando hay más de 3 antenas, y utiliza un método de cálculo diferente para obtener la ubicación del vehículo en un plano bidimensional. Este método también recibe una lista de distancias desde el vehículo hasta cada antena, y utiliza la información de todas las antenas para calcular la ubicación.

El método **getMessage()** toma como entrada una lista de datos de antenas y devuelve una lista de “metrics”.

El método recorre todas las antenas en la lista de entrada y completa las métricas faltantes en caso de que alguna esté vacía. Si una métrica está vacía, busca en todas las demás antenas en la lista para encontrar una métrica no vacía y la agrega a la lista de métricas finales. De esta manera, getMessage() intenta completar todas las métricas con los datos obtenidos.

Nivel 2

Para el Nivel 2 se creó una API REST para obtener la ubicación de la nave y las métricas que emite, la cual fue hosteada en Google App Engine (implementada en esta URL: <https://glass-marker-382018.rj.r.appspot.com/podhealth/>) y luego fue optimizada para funcionar en **Cloud Run** de Google Cloud.

El servicio /podhealth/ recibe información de la nave a través de un HTTP POST con un payload en formato JSON. El controlador de la API, la clase "PostDatosController", envía los datos recibidos a la clase "DatosAntenaService" para que se procesen. El método "datosAntena" de la clase "DatosAntenaService" procesa las coordenadas y métricas recibidas, y retorna una respuesta en formato JSON con la ubicación y métricas de la nave.

Para optimizar el funcionamiento de la API, se trabajó principalmente con Cloud Run de Google Cloud (utilizando Docker), lo que permitió mejorar la velocidad de respuesta y el rendimiento en general. La API se encuentra disponible en la siguiente URL: <https://pod-rxtzbm6vfq-rj.a.run.app/podhealth/>

Nivel 3

Para cumplir con el objetivo del Nivel 3, se crearon 2 clases, "GetPodSplitController" y "PostPodSplitController", que manejan los endpoints GET y POST respectivamente. Estas clases trabajan con las clases "GetEventsService" y "SubirEventsService", que manejan el acceso a los datos y las operaciones necesarias.

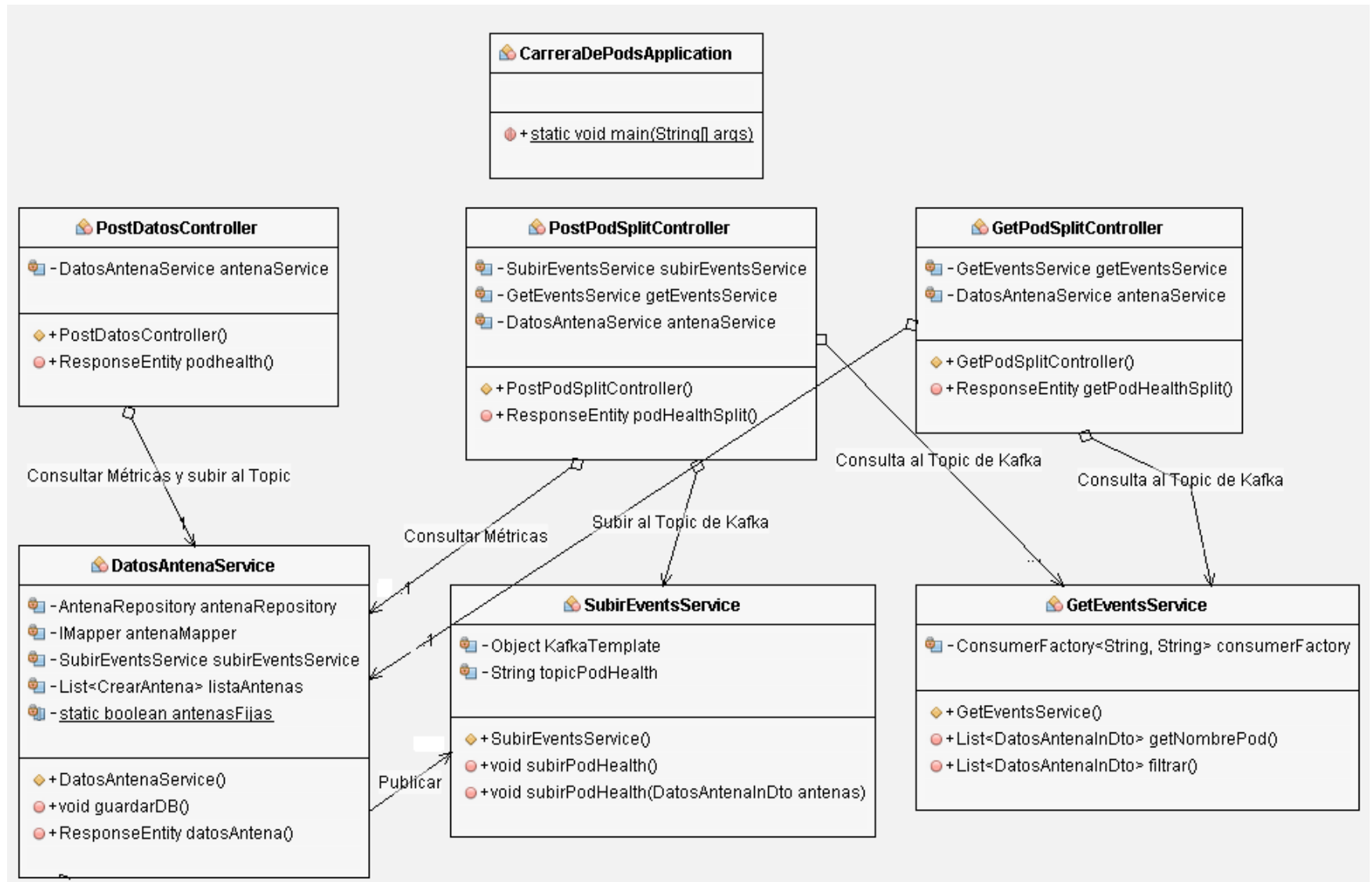
Además, se utilizó Apache Kafka para lograr la asincronía necesaria y permitir que los datos estén disponibles y puedan ser consultados en cualquier momento. Se creó un topic llamado "podhealth" donde se almacenan todos los body que van llegando a los endpoints POST.

Para respetar la misma respuesta que antes, se creó un nuevo endpoint POST en el servicio /podhealth_split/{**antena_name**} que acepta los mensajes y los almacena en el topic de Kafka. También se creó un endpoint GET /podhealth_split/{**pod_name**} que devuelve la posición y las métricas si es posible determinarlas a partir de los datos almacenados en el topic.

En caso de que no haya suficiente información para determinar la posición y las métricas, se devuelve un mensaje de error. Con esta implementación, se logró cumplir con el objetivo del nivel 3 de recibir el mensaje en diferentes POST al nuevo servicio /podhealth_split/ y tener la misma estructura que antes.

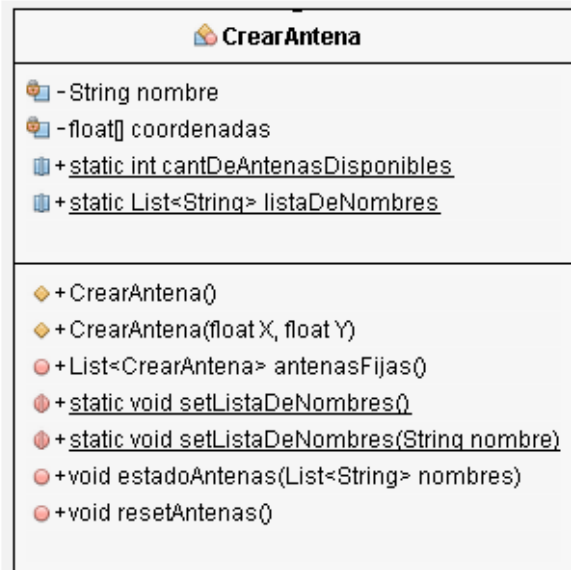
Diagrama UML

Principal

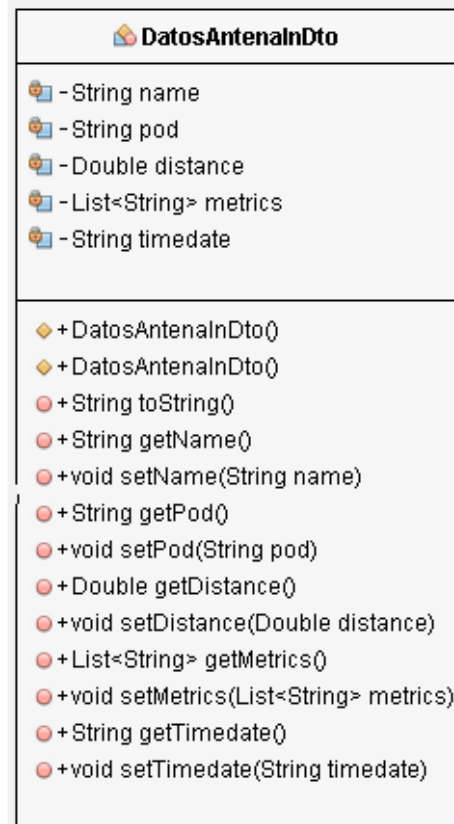


Otras Clases e Interfaces

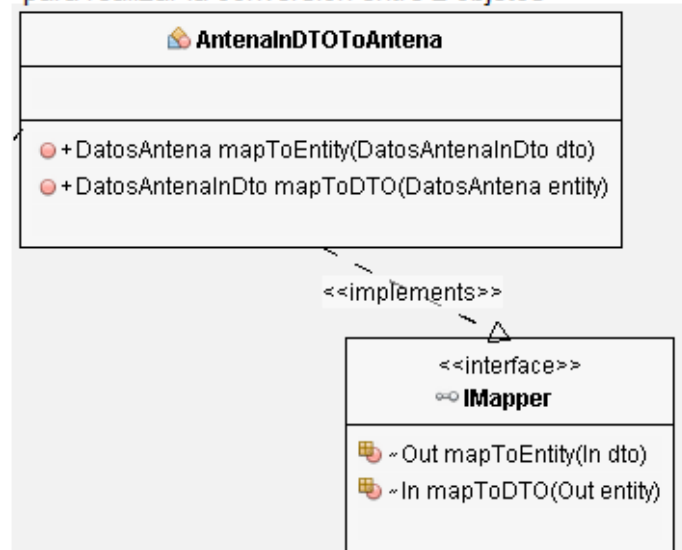
Representa las antenas fijas, se pueden crear tantos objetos como sean necesarios para recibir los datos



Posee similares campos que la entidad "DatosAntena", pero se utiliza para transferir datos entre capas de la aplicación



Realiza la conversión bidireccional entre la clase **DatosAntenaInDto** y la clase **DatosAntena**. Implementa la interfaz "**IMapper**", que define la estructura de una clase que proporciona los métodos para realizar la conversión entre 2 objetos



Documentación adicional

En este proyecto se ha utilizado Swagger para documentar la API REST. Con esta herramienta se definieron los endpoints disponibles en la API, los parámetros y cuerpo que se deben enviar en cada endpoint, los códigos de respuesta que se pueden recibir, entre otros detalles.

Con el fin de facilitar el uso y entendimiento de lo desarrollado, se puede acceder a la documentación desde la siguiente URL: <https://pod-rxtzbm6vfq-rj.a.run.app/doc.html>

Nivel 4

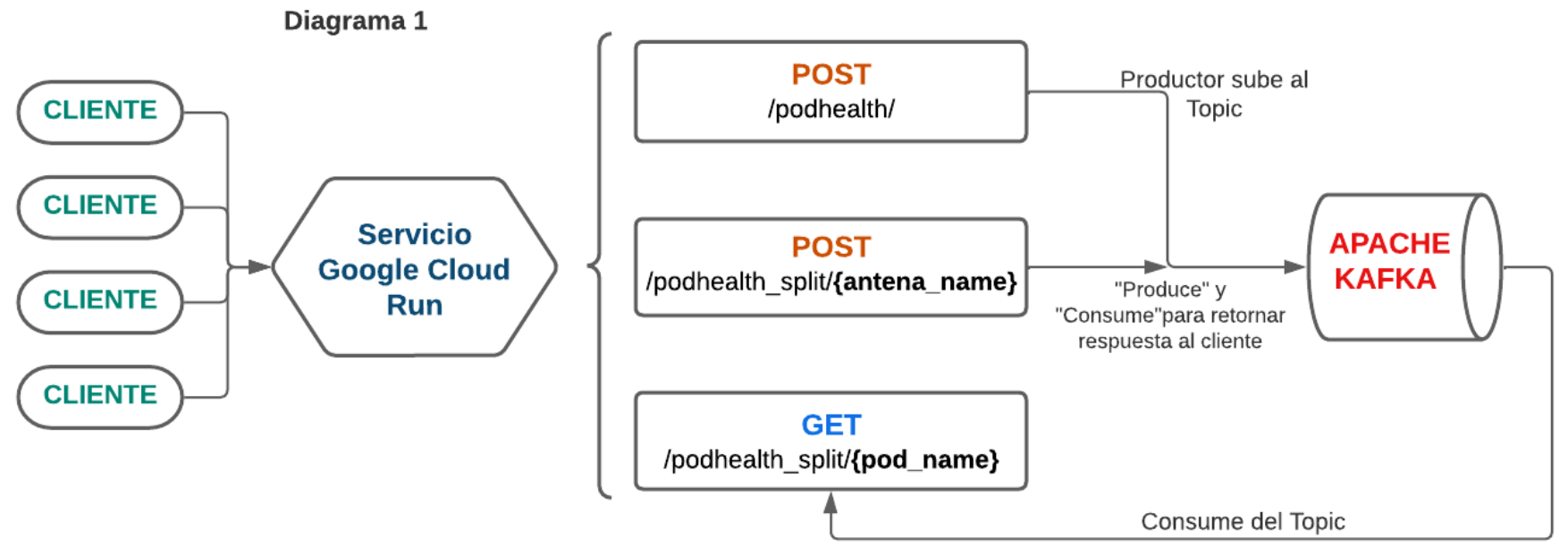
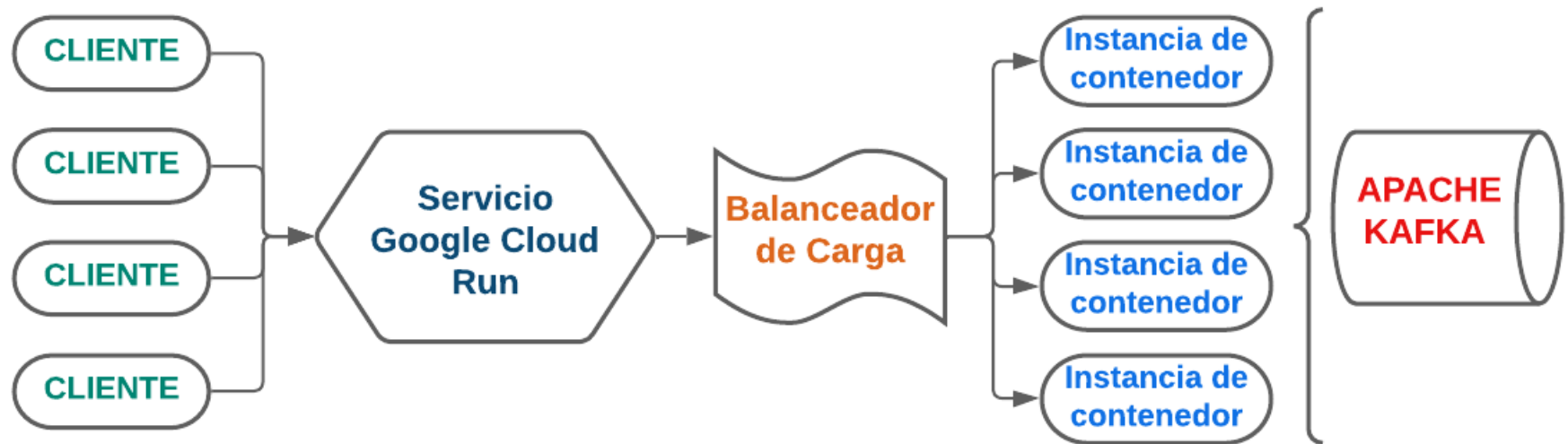


Diagrama 2



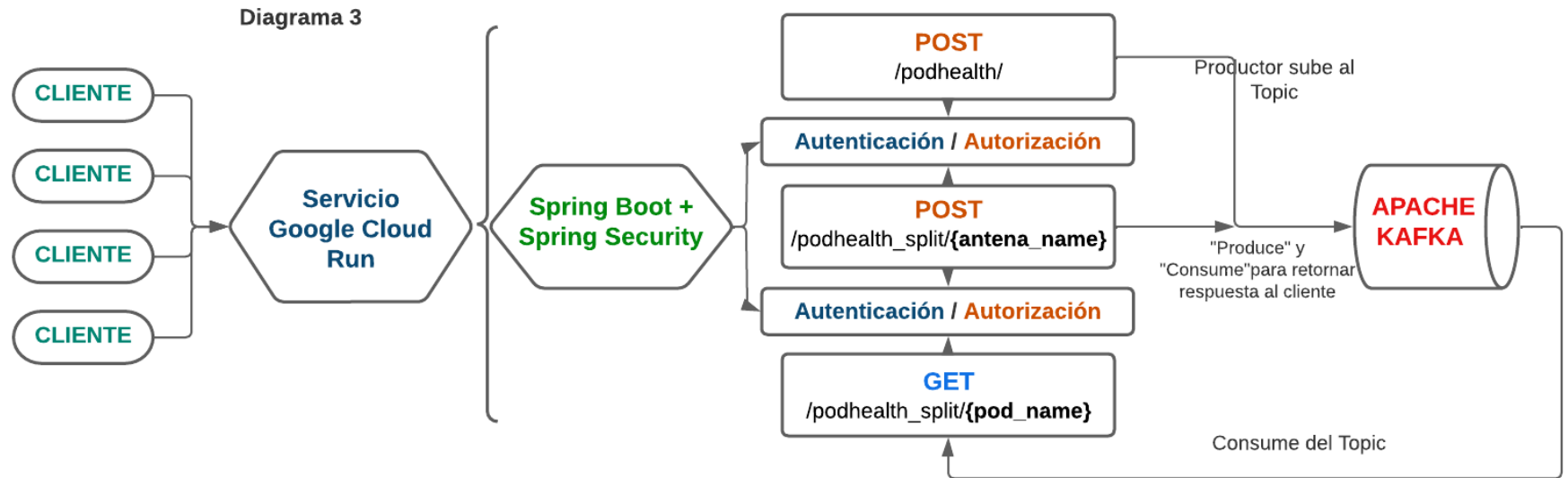
En el “Diagrama 1” se observa el flujo de lo desarrollado, se hizo uso de Apache Kafka para transmitir datos entre diferentes componentes del sistema. Cuando un cliente envía una solicitud a la API REST a través de un endpoint POST o GET, la instancia de contenedor que procesa la solicitud consume datos de Apache Kafka para generar una respuesta que se envía de vuelta al cliente.

En este caso se usó Cloud Run como muestra el “Diagrama 2”, para tener un mínimo de 10 instancias, esto significa que Cloud Run siempre mantendrá al menos 10 instancias de contenedores disponibles para ejecutar la aplicación. Cada instancia de contenedor es una copia de la aplicación en ejecución dentro de un contenedor y es capaz de procesar solicitudes de forma independiente.

Esto significa que siempre habrá al menos 10 contenedores a la escucha y listos para procesar las solicitudes entrantes a la API REST. Si el tráfico aumenta y se necesita más capacidad para manejar las solicitudes entrantes, Cloud Run puede escalar automáticamente el número de instancias de contenedores hasta el máximo de 100 que son las que se configuraron en el Cloud para este proyecto.

Esta es la arquitectura propuesta para cumplir con los requerimientos no funcionales, además, como opciones de software de código abierto (OSS) para ejecutar contenedores, podemos usar Kubernetes o OpenShift ya que permiten la creación de clústeres de contenedores y además ofrece también características como el balanceo de carga, entre otras.

- Para agregar una capa de seguridad al programa se puede utilizar la combinación de Spring Boot con Spring Security, esto nos proporciona funciones para la autenticación, autorización y protección contra ataques. Agregando las dependencias necesarias al proyecto es posible incluir en las clases las anotaciones (@EnableWebSecurity) para configurar la autenticación y la autorización, que puede incluir la definición de usuarios y roles, la configuración de proveedores de autenticación como OAuth, y la definición de reglas de acceso.



- Para complementar la escalabilidad horizontal de los recursos que venimos desarrollando se buscó optimizar el código utilizando Paradigma Orientado a Objetos, para tal fin se creó la clase “CreaAntena” que tiene como función definir y manejar los atributos de las antenas que se usarán en el programa. En su constructor, se define el nombre y coordenadas.

La clase tiene 2 métodos estáticos que se usan para actualizar una lista de nombres de antenas y para actualizar la cantidad de antenas disponibles. Esto con la finalidad que este recurso pueda ser accedido rápidamente desde las diferentes clases y tomar acciones según corresponda. *La estructura de esta clase se muestra en los diagramas UML

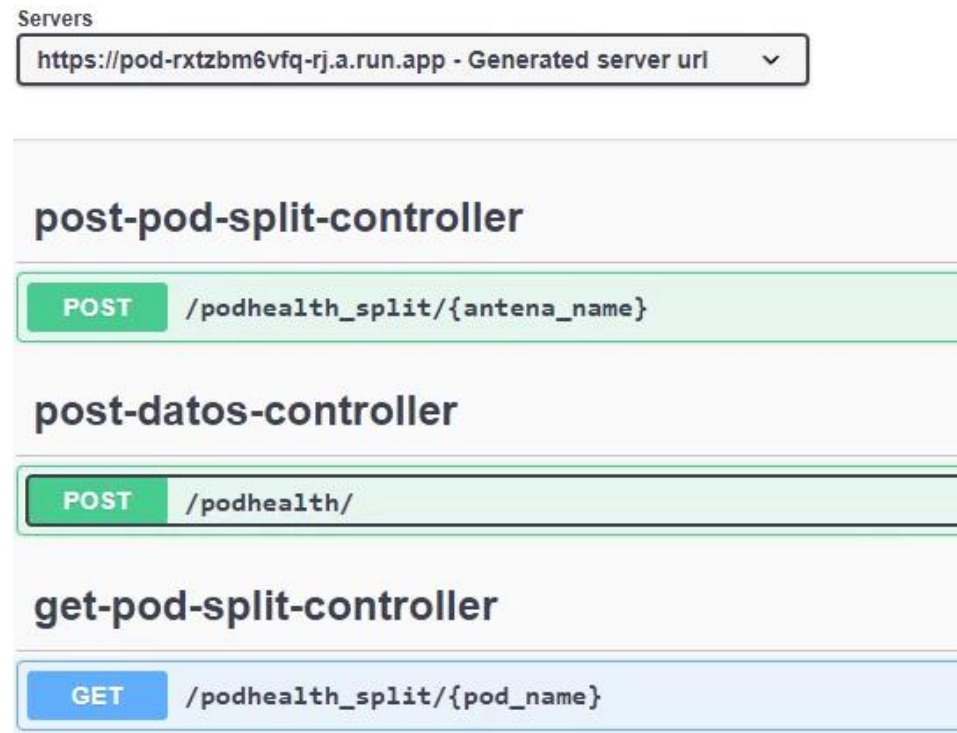
- Para cumplir con el requisito de almacenar registros de logs se puede usar la biblioteca de registro Logback que viene por defecto en Spring Boot y puede personalizarse.

Además, los registros generados podrían enviarse a un sistema de registro centralizado como “Splunk”. Lo que permitirá almacenar y consultar los registros de logs para realizar análisis históricos.

Finalmente, a continuación, se comparten algunas pruebas realizadas en el entorno:

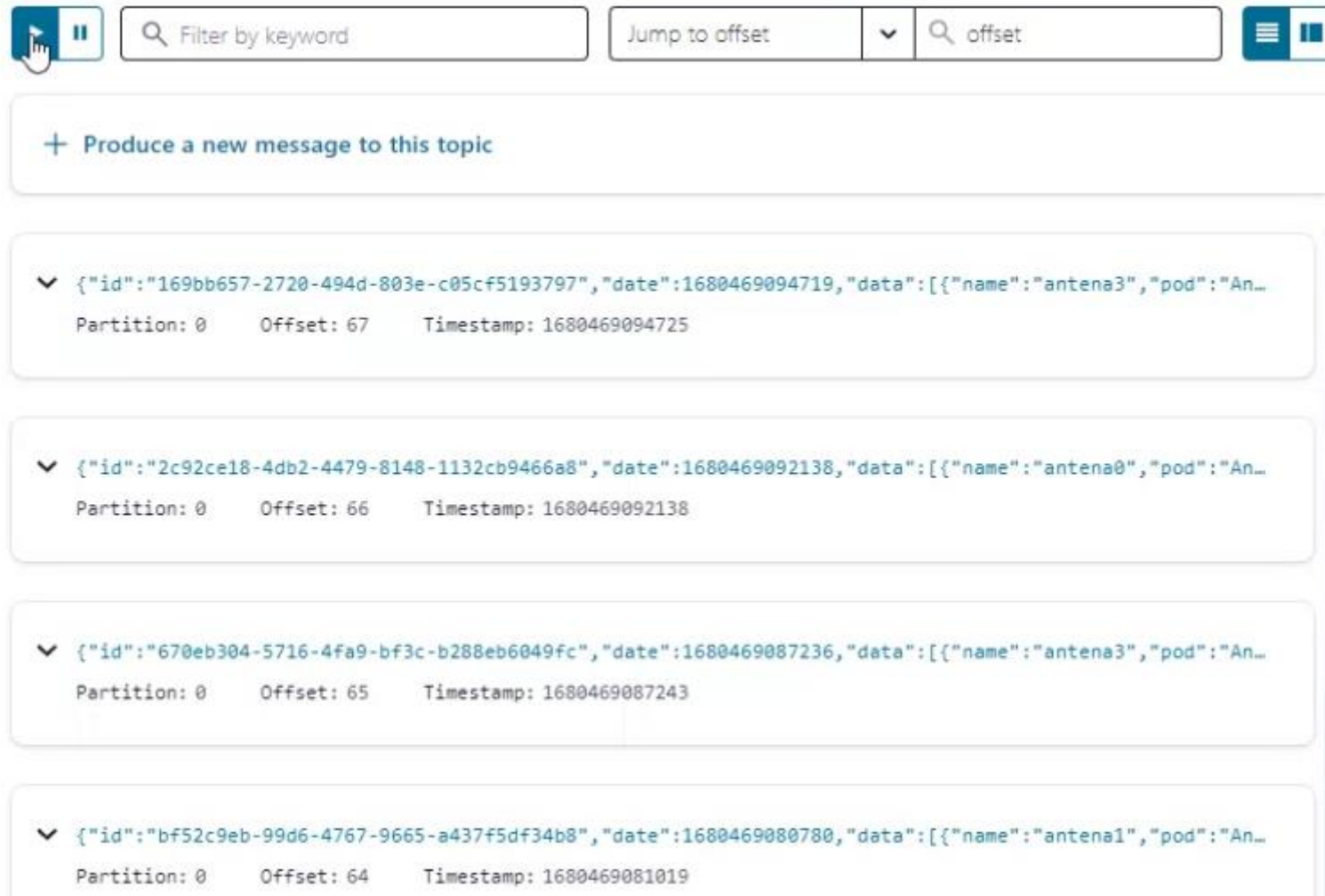
Probando Módulo desde Swagger

Link del **Video**: <https://youtu.be/TaH5F-3e59g>



Produciendo un mensaje en un Topic de Kafka

Link del **Video**: <https://youtu.be/vTg6MuNFHds>



The screenshot displays the Kafka UI interface for producing and viewing messages. At the top, there are controls for filtering by keyword, jumping to a specific offset, and a search bar. Below these controls is a button labeled '+ Produce a new message to this topic'. The main area shows a list of four messages, each with a dropdown arrow, a JSON payload, and metadata including Partition, Offset, and Timestamp.

Filter by keyword

Jump to offset

offset

+ Produce a new message to this topic

▼ {"id":"169bb657-2720-494d-803e-c05cf5193797","date":1680469094719,"data":[{"name":"antena3","pod":"An..."}]
Partition: 0 Offset: 67 Timestamp: 1680469094725

▼ {"id":"2c92ce18-4db2-4479-8148-1132cb9466a8","date":1680469092138,"data":[{"name":"antena0","pod":"An..."}]
Partition: 0 Offset: 66 Timestamp: 1680469092138

▼ {"id":"670eb304-5716-4fa9-bf3c-b288eb6049fc","date":1680469087236,"data":[{"name":"antena3","pod":"An..."}]
Partition: 0 Offset: 65 Timestamp: 1680469087243

▼ {"id":"bf52c9eb-99d6-4767-9665-a437f5df34b8","date":1680469080780,"data":[{"name":"antena1","pod":"An..."}]
Partition: 0 Offset: 64 Timestamp: 1680469081019

Prueba POST del endpoint /podhealth_split/{antena_name}

Link del **Video**: <https://youtu.be/k0x5ppkykT0>

* En esta prueba se envía el body con el parámetro requerido, el programa lo recibe en la clase **PostPodHealthController** y realiza la acción de Publicarlo en el topic enviando los datos a la clase **SubirEventsService**. Inmediatamente envía el parámetro con la clave **Pod** a la clase **GetEventsService** que se encarga de consumir el topic y recuperar los últimos 10 eventos (esta cantidad fue establecida arbitrariamente para pruebas). El sentido de esto es que la clase busque en esta lista los datos necesarios para completar la información del resto de las antenas. Para ello compara una variable de tipo **Timedate** para filtrar aquellos mensajes que coincidan con una marca de tiempo menor o mayor a 2 segundos (esta marca también se establece arbitrariamente suponiendo que se intenta agrupar datos lo más cercano al tiempo real, por ende, descarta aquellos que superan este tiempo). Finalmente, si no es posible agrupar el total de antenas (establecidas en un mínimo de 3 para los cálculos) devuelve un mensaje con Status 404 y la respuesta: "No hay suficiente información para procesar la solicitud".

