Escuela Técnica Superior de Ingeniería de Telecomunicación

Diseño y desarrollo de un microservicio para la gestión de información de monitorización y predicciones de tráfico en red

7 de diciembre de 2022

TRABAJO FIN DE MÁSTER

Máster Universitario en Ingeniería de Telecomunicación

Autor: Enrique Fernández Sánchez

Tutor: Pablo Pavón Mariño

Universidad Politécnica de Cartagena

Índice general

Índice de figuras

Listado de ejemplos

Capítulo 1

Introducción

1.1 Contexto del trabajo

1.2 Motivación

1.3 Descripción Global

1.4 Objetivos

1.5 Resumen capítulos de la memoria

Capítulo 2

Tecnologías empleadas

En este capítulo, se van a presentar las diferentes tecnologías utilizadas para la implementación de la aplicación.

2.1 Arquitectura y microservicios

En primer lugar, se va a comentar acerca de la arquitectura escogida. En este caso, se decide realizar una implementación basada en microservicios utilizando una REST API.

Arquitectura basada en microservicios

Lo primero, es entender en que consiste un microservicio. Para ello, podemos definirlo como los sistemas que cumplen las siguientes premisas: [2]

- Los microservicios son sistemas pequeños, independientes y poco "acoplados" (ver figura 2.1).
- Cada servicio tiene su propio código fuente, que esta separado del resto de códigos de los servicios.
- Cada servicio se puede desplegar de manera independiente.
- Cada servicio es responsable de la persistencia de sus datos.
- Los servicios se comunican entre sí utilizando APIs
- Además, como ventaja, los servicios no tienen por qué estar implementados todos en el mismo lenguaje de programación.

Por lo tanto, dado los objetivos presentados en este trabajo, se llegó a la conclusión de que tratar el sistema propuesto como un microservicio podría aportar numerosas ventajas, ya que permitiría ser utilizado por otros servicios, extendiendo la funcionalidad de estos y añadiendo un valor extra. Para ello, será necesario definir la API que utilizaremos para comunicarnos con el sistema.

Comunicación basada en API

Una API permite a dos componentes comunicarse entre sí mediante una serie de reglas. Además, supone un "contrato" en el que se establecen las solicitudes y respuestas esperadas en la comunicación. [1]

Dependiendo de la implementación de la API que se realice, distinguimos cuatro tipos de API:

- API de SOAP. Utilizan un protocolo de acceso a objetos. Los interlocutores intercambian mensajes XML. En general, es una solución poco flexible.
- API de RPC. Basado en llamadas de procedimientos remotos. El cliente ejecuta una función en el servidor, y este responde con la salida de la función.
- API de WebSocket. Solución moderna de desarrollo de API, que utiliza objetos JSON y un canal bidireccional para realizar la comunicación entre el cliente y el servidor.
- API de REST. Solución más popular. El cliente envía solicitudes al servidor como datos, utilizando métodos HTTP. Es una opción muy flexible.

En en caso de nuestra aplicación, se decidió utilizar el tipo REST API, ya que permite una sencilla implementación de cara al cliente que quiera utilizar dicha interfaz.

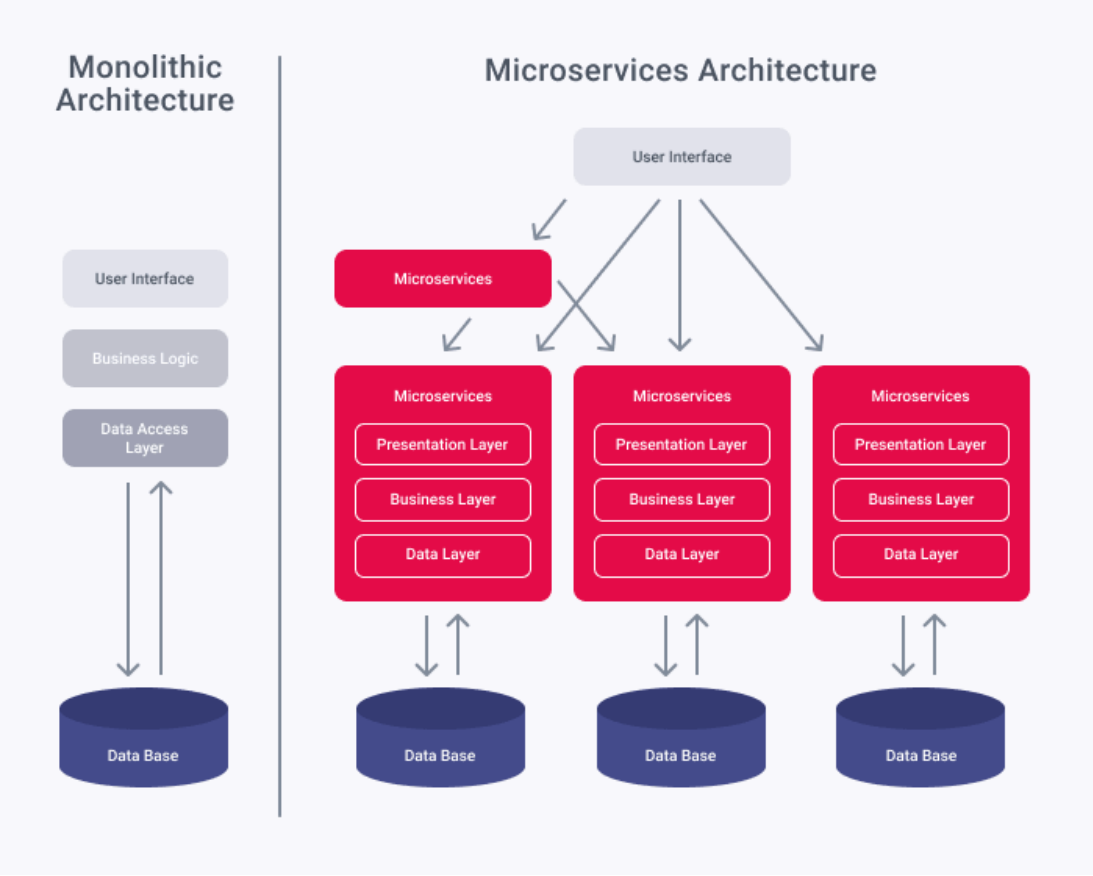

Figura 2.1: Comparativa entre arquitectura de microservicios y arquitectura "monolítica". [1]

2.2 Bases de datos

Para asegurar la persistencia de los datos en nuestra aplicación, es necesario utilizar una base de datos. En dicha base de datos, guardaremos información relevante para el correcto funcionamiento del sistema, en nuestro caso, redes y/o interfaces a monitorizar, o los datos monitorizados.

En la aplicación de monitorización, distinguimos entre datos de dos tipos:

- Datos clásicos. Como por ejemplo, la información asociada a una red a monitorizar.
- Datos de tipo "time series". Como por ejemplo, las muestras de monitorización de una red.

En primer lugar, se diseña una base de datos tipo SQL para almacenar los "datos clásicos". Y por otro lado, se diseña una base de datos diferente, especializada para el almacenamiento de datos tipo "time series", en este caso, se elige una base de datos llamada InfluxDB.

2.2.1 Base de datos tipo relacional/SQL

SQL es una base de datos de tipo relacional. Dichas bases de datos, suponen una colección de información que organizan los datos en una serie de "relaciones" cuando la información es almacenada en una o varias "tablas". Por lo tanto, las relaciones suponen conexiones entre diferentes tablas, permitiendo así una asociación entre información diferente. [3]

Por ejemplo, si vemos la figura 2.2, podemos comprobar como se realizan las relaciones entre las diferentes tablas (Ratings, Users, Movies o Tags), se realiza mediante uno de los campos definidos en la propia tabla. Por ejemplo, el campo "user_id" de la tabla Ratings, permite una relación con la tabla Users, con el campo "id".

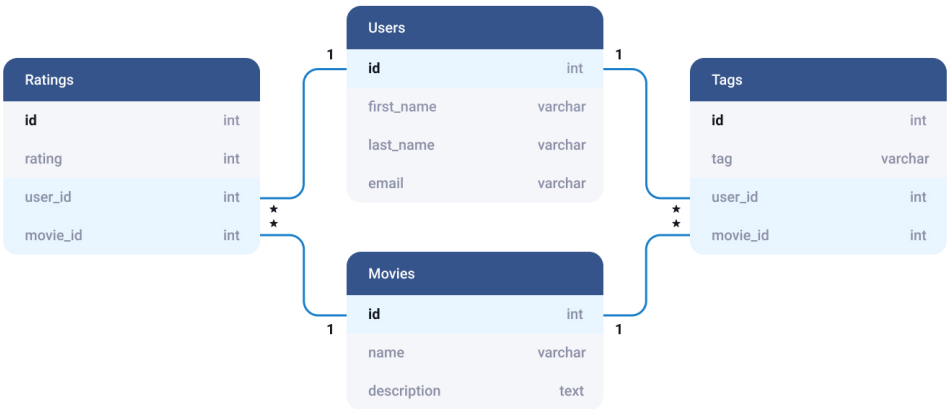

Figura 2.2: Ejemplo de relaciones dentro de una base de datos SQL. [2]

Para el caso de nuestra aplicación de monitorización de tráfico en red, modelamos la base de datos según la información que vamos a almacenar. Por lo que tenemos que definir la estructura de tablas, los campos que van a tener cada una de las tablas, y los campos por los que se van a relacionar entre sí. A esta información la llamamos modelo de datos.

2.2.2 Base de datos tipo "time series"/InfluxDB

InfluxDB es una base de datos diseñada para trabajar con datos tipo time-series. Si bien es cierto que SQL puede gestionar este tipo de datos, no fue creado estrictamente para este objetivo. En este caso, InfluxDB esta diseñado para almacenar grandes volúmenes de datos, y además realizar análisis en tiempo real sobre esos datos.

En comparación con SQL, en InfluxDB un "timestamp" identifica un punto en cualquier serie de datos. Esto sería equivalente a SQL, si la clave primaria de una tabla es establecida por el sistema y siempre es equivalente al tiempo. Además, InfluxDB permite reconocer el "schema" de manera dinámica, además de que no estas obligado a definir el "schema" y seguirlo, es decir, se permiten cambios dentro de la misma serie de datos. [6]

Algunas de las razones destacadas para elegir InfluxDB son: [5]

- Perfecto para almacenar datos de telemetría, como métricas de aplicaciones o sensores IoT.
- Los datos son comprimidos automáticamente para ser eficientes con el espacio disponible.
- Se realizan tareas automáticas de "downsampling" para reducir el uso de disco.
- Lenguaje para hacer consultas que permite analizar en profundidad los datos almacenados.
- Disponible una aplicación web para realizar consultas y comprobar los datos disponibles en la base de datos.

Terminología

Comparando con los conceptos ya existentes en bases de datos de tipo SQL, se definen los siguientes conceptos en InfluxDB:

- "measurement": equivalente a una tabla.
- "tags": equivalente a columnas indexadas dentro de una tabla.
- "fields": equivalente a columnas no indexadas dentro de una tabla.
- "points": similar a las filas en una tabla.

Figura 2.3: Logotipo base de datos InfluxDB [3].

2.3 Lenguajes de programación y frameworks

En resumen, para el desarrollo de esta aplicación se ha utilizado el lenguaje de programación Python, con el framework de desarrollo para APIs llamado FastAPI.

2.3.1 Python

Python [7] es un lenguaje de programación orientado a objetos, interpretado y de alto nivel con tipado dinámico. Es muy atractivo ya que permite un desarrollo rápido de aplicaciones, además de ser muy adecuado para realizar tareas de "scripting". Python es un lenguaje simple y sencillo de aprender. Por otro lado, dispone de multitud de "librerías" o "módulos" publicados por usuarios, dando lugar a una gran comunidad y una gran variedad de alternativas para implementar soluciones.

Actualmente, Python destaca como lenguaje de programación en los siguientes ámbitos:

- Desarrollo de aplicaciones web, utilizando los frameworks Django, Flask o FastAPI.
- Tareas asociadas a "data science", utilizando librerías como Pandas o NumPy.
- Inteligencia artificial, utilizando frameworks como TensorFlow o scikit-learn.

Figura 2.4: Logotipo Python [4].

2.3.2 FastAPI

FastAPI [8] es un framework moderno y rápido para construir APIs utilizando la versión de Python 3.7+. Algunas de las características más destacadas son:

- Rápido: rendimiento muy alto, prácticamente a la par con otros lenguajes de programación destinados al desarrollo de backend (como NodeJS o Go).
- Intuitivo: soporta auto completado en el código.
- Robusto: código pensado para entornos de producción, además de incluir documentación automática (usando Swagger o ReDoc).
- Basado en estándares: al utilizar estándares de tipo de datos, permite ser totalmente compatible con los estándares de OpenAPI [9] y JSON Schema [10].

Figura 2.5: Logotipo FastAPI [5].

2.3.3 Prophet

Prophet [11] es un framework del lenguaje de programación Python, desarrollado por Meta, que recoge una serie de procedimientos que permiten realizar predicciones de un dataset de series temporales, en el que se pueden encontrar diferentes efectos no lineales, llamados tendencias (como puede ser una tendencia anual, semanal, o mensual), además de otros efectos causados por fechas concretas.

Es un framework de predicción basado en inferencia estadística, lo que permite tener un rendimiento mayor que si utilizamos técnicas de Machine Learning para solucionar el mismo problema de predicción de datos. Además, es robusto a datos no disponibles y a modificaciones aleatorias sobre las tendencias.

Figura 2.6: Logotipo Prophet [6].

2.4 Tecnologías utilizadas en un despliegue en producción

Otro de los aspectos importantes para la realización de este proyecto, es el hecho de que la aplicación desarrollada debe ser apta para desplegarse en un entorno de producción y funcionar correctamente para que sea implementada como microservicio por otras aplicaciones. Es por este motivo por el que nos decantamos por implementar este microservicio dentro de un contenedor.

Docker

Utilizamos Docker como la tecnología para realizar un contenedor de nuestra aplicación, implementando dentro del contenedor todas las librerías y código necesario para hacer funcionar la aplicación.

Por otro lado, también será necesario desplegar diferentes contenedores que tendrán alojadas las bases de datos que utilizaremos en el proyecto. En este caso, al utilizar dos bases de datos, tendremos que hacer uso de un contenedor para desplegar una base de datos SQL, y otro contenedor para desplegar una base de datos InfluxDB.

Los contenedores Docker necesarios se desplegarán sobre una máquina host que tenga en funcionamiento el "daemon" de Docker.

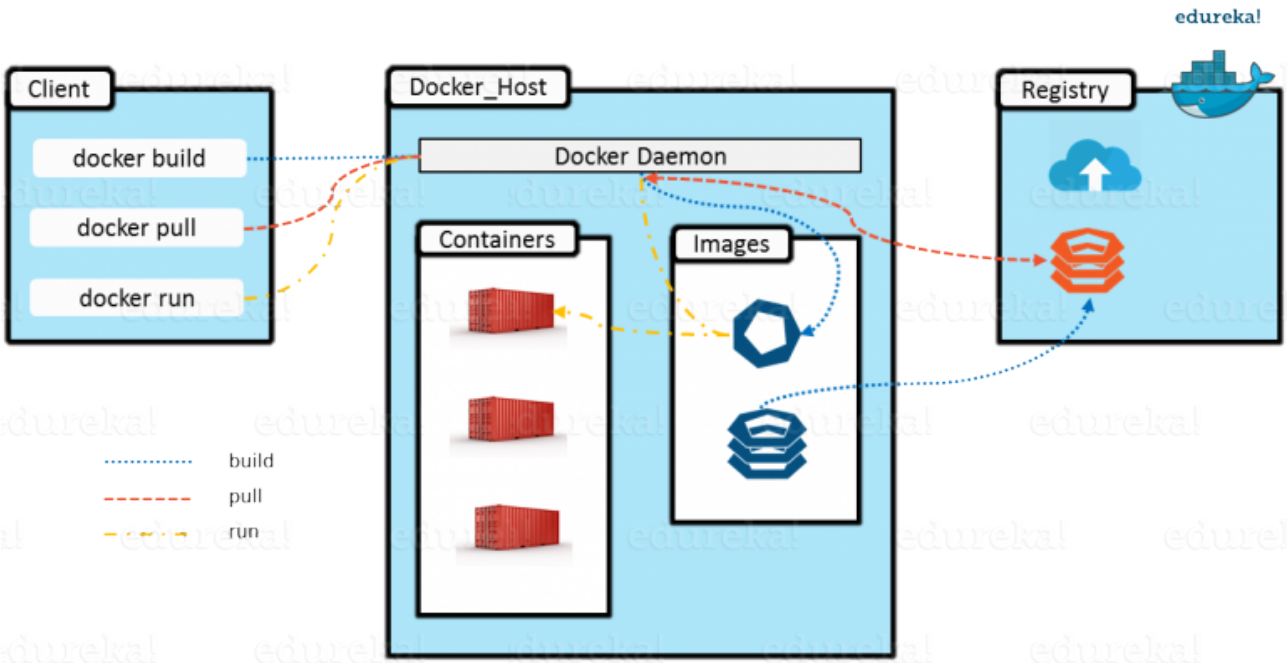

Figura 2.7: Docker

docker-compose

Traefik

Capítulo 3

Diseño e implementación del sistema

3.1 Descripción REST API

3.2 Estructura de la aplicación

3.3 Modelos de datos

3.3.1 Base de datos SQL

Modelos de datos

El primer modelo de datos de la aplicación, es el referido a la información de una red a monitorizar. La descripción del modelo la podemos ver en la figura 3.1.

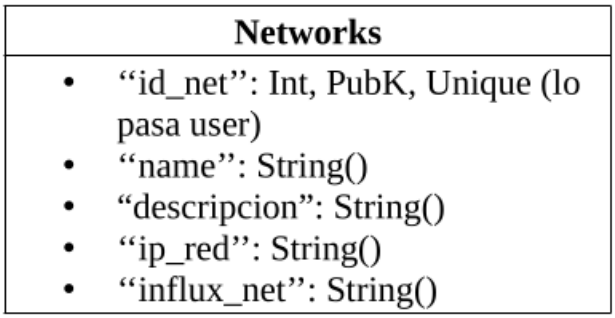

Figura 3.1: Modelo de datos para las redes a monitorizar. Equivale con la tabla ”networks”.

El segundo modelo de datos de la aplicación, es el referido a la información de una interfaz a monitorizar, que esta dentro de una red monitorizada. La descripción del modelo la podemos ver en la figura 3.2.
Se establece una relación del modo que una interfaz solo puede pertenecer a una red monitorizada, y que además, una red monitorizada puede tener muchas interfaces. Dicha relación se realiza mediante el campo ”id_net” de la tabla Interfaces.

Por último, la base de datos SQL utilizada para esta aplicación es PostgreSQL [4], ya que además de ser Open Source, permite una gran escalabilidad, amoldándose a los recursos de la máquina en la que esté funcionando.

Interfaces
• "id_if": Int, PubK, Unique (lo pasa user) • "id_net": Int, ForK • "name": String() • "description": String() • "influx_if_rx": String() • "influx_if_tx": String()

Figura 3.2: Modelos de datos para las interfaces a monitorizar. Equivale con la tabla "interfaces"

3.3.2 Base de datos InfluxDB

Modelo de datos

Para la aplicación a diseñar, se plantea la premisa de que en la base de datos InfluxDB solo se van a guardar los datos de cada muestra de monitorización de una interfaz. Además, se pueden tener tantas redes como sean necesarias, y dentro de cada red, tantas interfaces como creamos necesarias.

En resumen, definimos el modelo de datos que tiene que seguir nuestra aplicación:

- "measurement": equivalente a una red a monitorizar, valor almacenado en Networks::influx_net.
- "fields": nombre del valor a monitorizado, en este caso el default es *link_count*.
- "tags": solo tenemos un tag llamado *interface*, su objetivo es identificar a que interfaz pertenece el punto de monitorización. El valor puede estar ser Interfaces::influx_if_rx o Interfaces::influx_if_tx.
- "points": corresponde con el valor numérico del "field". En este caso, corresponde con el valor númerico de link_count en ese periodo de 5 minutos.

3.4 Endpoints

3.5 Implementación del sistema

Capítulo 4

Pruebas y validación del sistema

Capítulo 5

Conclusiones

5.1 Propuestas futuras

Capítulo 6

Bibliografía

Enlaces y referencias

1. ¿Qué es una API?
2. Microservice architecture style
3. What is a relational database?
4. PostgreSQL
5. InfluxDB: Introduction
6. InfluxDB: Comparison to SQL
7. Python
8. FastAPI framework
9. GitHub: OpenAPI-Specification
10. JSON Schema
11. Prophet

Figuras

1. Monolithic vs Microservices Architecture
2. Relational Databases
3. InfluxDB
4. Python
5. FastAPI
6. Prophet

Anexos

Anexo I. Generación dataset sintético