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

13 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

Índice de tablas

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.

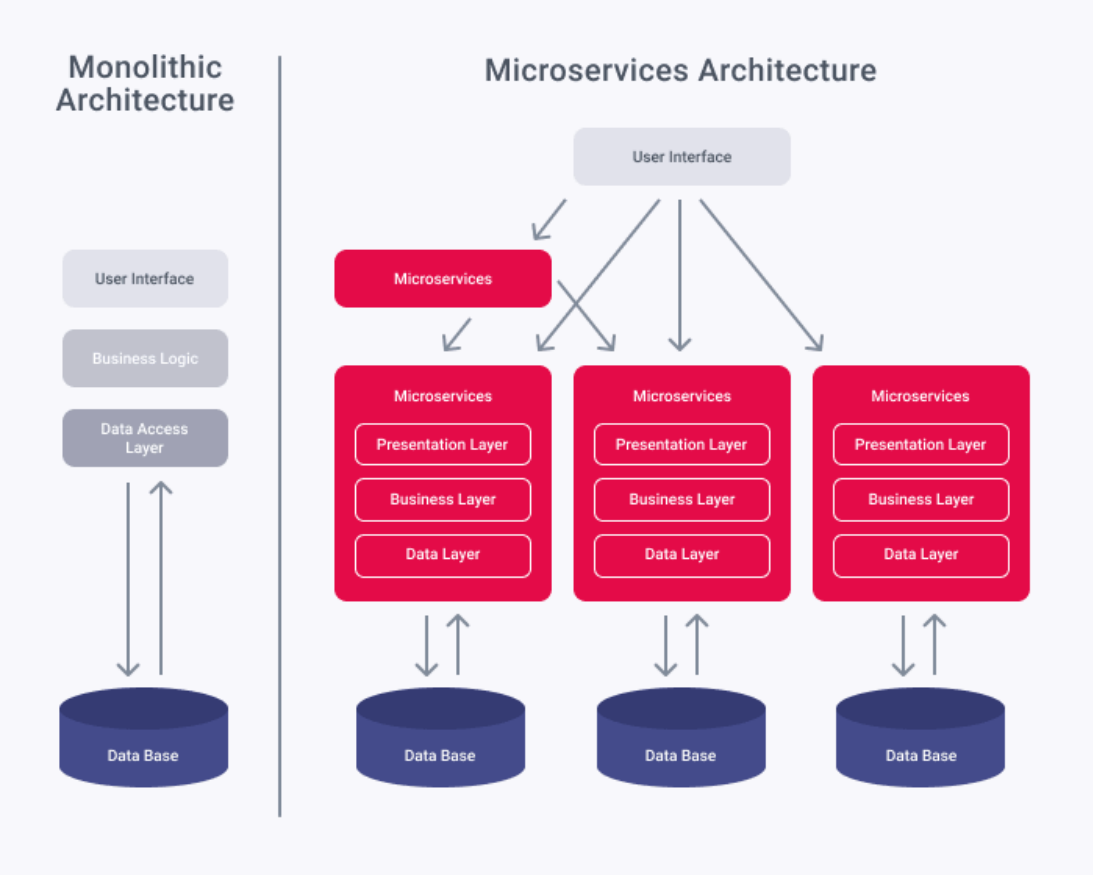

Figura 2.1: Comparativa entre arquitectura de microservicios y arquitectura "monolítica". [1]

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".

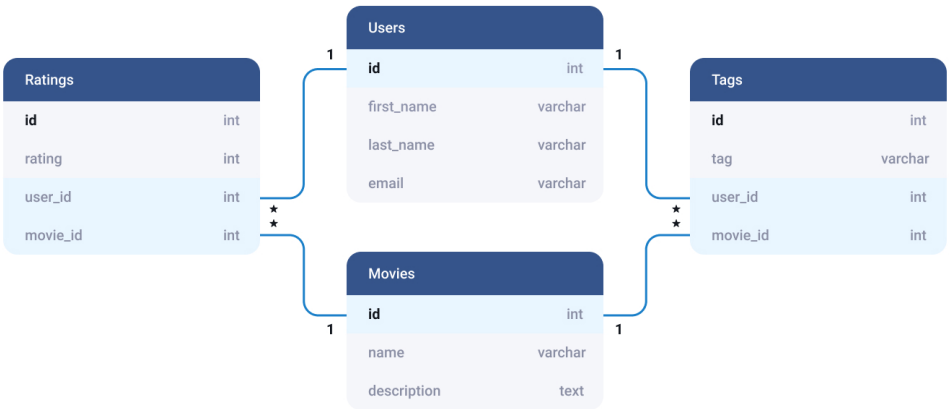

Figura 2.2: Ejemplo de relaciones dentro de una base de datos SQL. [2]

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.

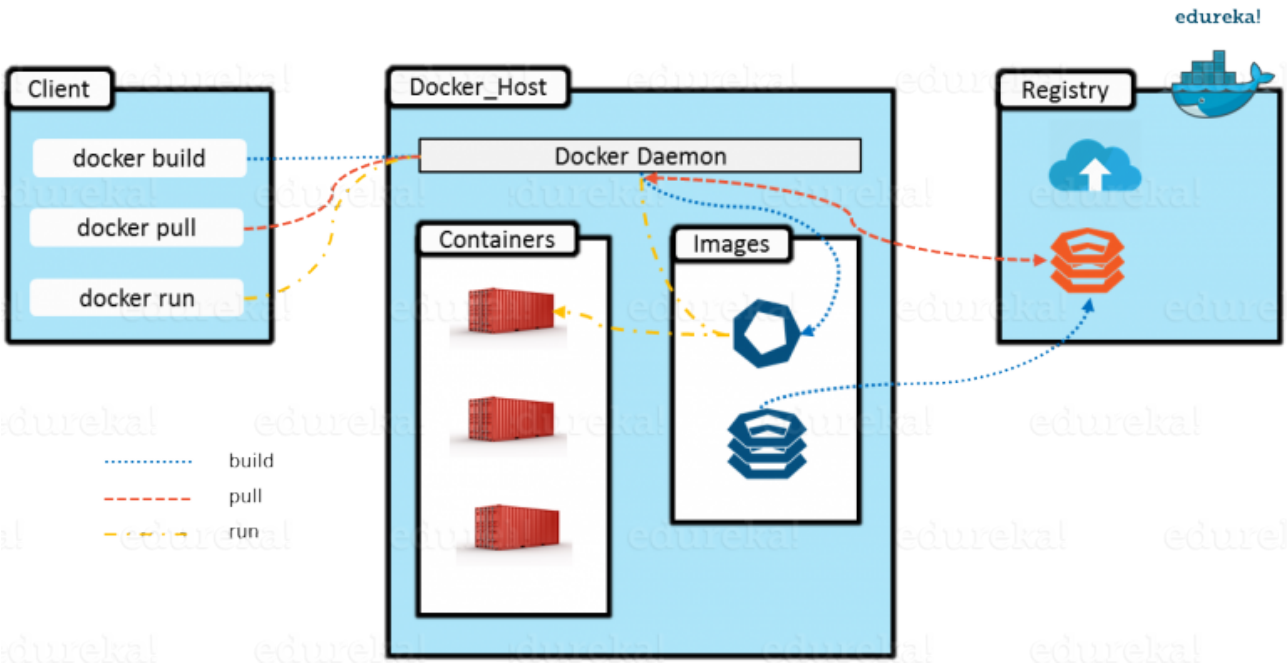

Figura 2.7: Docker

Docker Compose

Compose es una herramienta de Docker que permite definir y ejecutar instancias multi contenedores de Docker [12]. Con la herramienta Compose, podemos utilizar un archivo YAML para describir y configurar los contenedores que vamos a utilizar, dando lugar a que con un único comando puedas crear y ejecutar todos los servicios de la aplicación, y con la misma configuración.

Compose funciona para los diferentes entornos: producción, desarrollo, y o pruebas. Además, tiene disponibles comandos para hacer más sencilla la tarea de iniciar o parar servicios, o ver la salida de consola producida por los contenedores.

En el caso de nuestra aplicación, el archivo "*docker-compose.yml*" utilizado para el entorno de desarrollo sería el siguiente:

Ejemplo 2.1: Archivo configuración de contenedores para entorno de desarrollo.

```
version: '3.8'

services:
  traffic_forecast:
    build: ./backend
    ports:
     - 5000:5000
    environment:
     - POSTGRES_USER=monitor
     - POSTGRES_PASSWORD=forecast2022
     - POSTGRES_DB=traffic-forecast
     - INFLUX_TOKEN=*******
     - INFLUX_ORG=e-lighthouse
     - INFLUX_BUCKET=traffic-forecast
     - SECRET_KEY=upct2022_sk
     - FASTAPI_CONFIG=development
    volumes:
     - ./backend:/app
    depends_on:
     - db
     - influxdb

  db:
    image: postgres:13
    expose:
     - 5432
    environment:
     - POSTGRES_USER=monitor
     - POSTGRES_PASSWORD=forecast2022
     - POSTGRES_DB=traffic-forecast
    volumes:
     - ./data/postgres_db:/var/lib/postgressql/data

   influxdb:
     image: influxdb:latest
     volumes:
       - ./data/influxdb/data:/var/lib/influxdb2:rw
     ports:
       - 8086:8086

```

Para desplegar los contenedores, según el archivo YAML definido, tenemos que ejecutar el siguiente comando:

```
docker compose up -f <filename> traffic_forecast
```

Traefik

Traefik es una herramienta que permite hacer de "reverse proxy" y "load balancer" en contenedores Docker, permitiendo el despliegue sencillo de microservicios en un entorno de producción.

Traefik esta diseñado para ser simple de operar, pero con una gran capacidad de gestión en entornos complejos. Además, se integra perfectamente con la infraestructura ya existente y configurar el sistema de manera dinámica. Algunas de las tareas que realiza Traefik son: [13]

- Gestionar middlewares necesarios para la aplicación (forzar protocolos específicos, configurar contraseña para el servicio...).
- Funcionar con API Gateway, gestionando los dominios y certificados para cada uno de los microservicios desplegados.
- Orquestador de nodo de entrada, gestionando la comunicación de tráfico de un dominio hacia el servicio asociado.

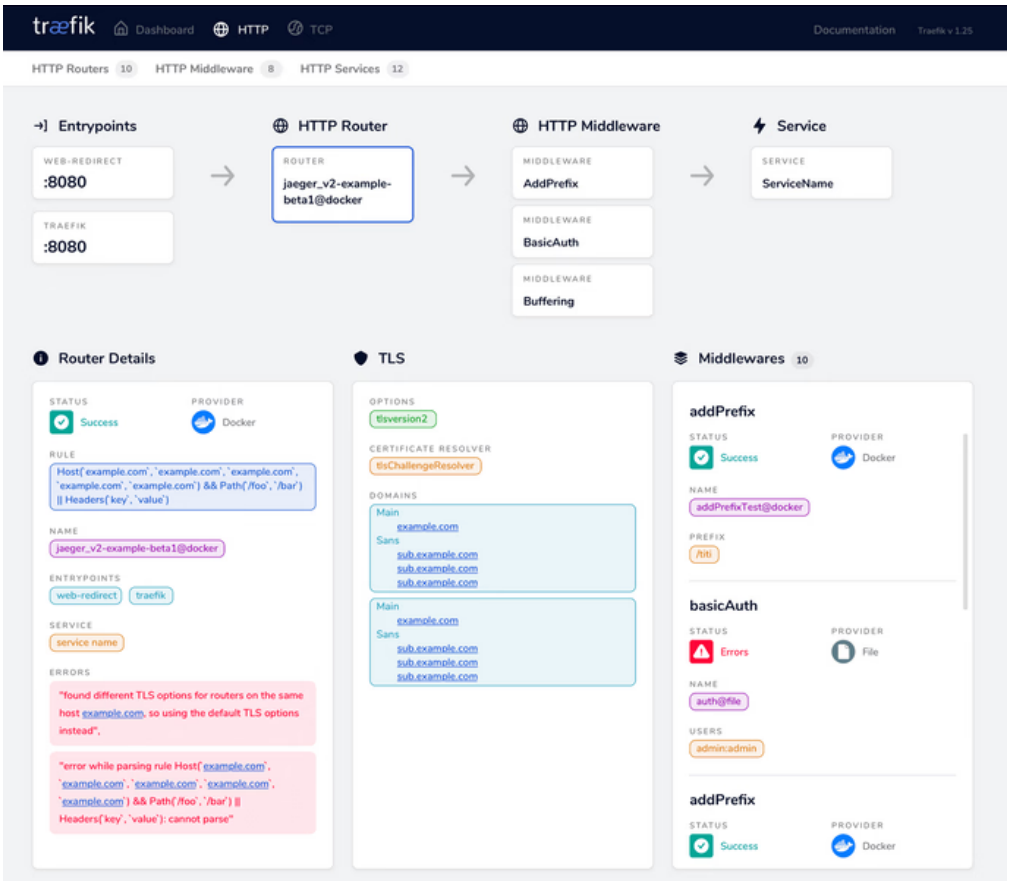

Figura 2.8: Captura portal de gestión de Traefik. Detalle de enrutado hacia un servicio.

En el caso de nuestra aplicación, para el entorno de producción usamos el siguiente archivo que permite a la herramienta docker compose ejecutar los servicios con sus configuraciones asociadas. Como podemos ver, en la sección de "labels" se encuentra la configuración asociada a Traefik. Dicha configuración permite a Traefik asignar un servicio a un dominio, crear un certificado SSL y configurar los puertos de acceso para dicho servicio. [14]

Ejemplo 2.2: Archivo configuración de contenedores para entorno de producción.

```
version: '3.8'

services:
  traffic_forecast:
    build: ./backend
    container_name: traffic_forecast-api
    restart: unless-stopped
    environment:
      - POSTGRES_USER=monitor
      - POSTGRES_PASSWORD=forecast2022
      - POSTGRES_DB=traffic-forecast
      - INFLUX_URL=https://tfm-influx.ranii.pro:8443/
      - INFLUX_TOKEN=****
      - INFLUX_ORG=e-lighthouse
      - INFLUX_BUCKET=traffic-forecast
      - SECRET_KEY=upct2022_sk
    volumes:
      - ./backend:/app
    labels:
      - traefik.enable=true
      - traefik.http.routers.tf-api.entryPoints=web-secure
      - traefik.http.routers.tf-api.rule=Host(`tfm-api.ranii.pro`)
      - traefik.http.routers.tf-api.tls.certresolver=default
      - traefik.http.services.tf-api.loadbalancer.server.port=5000
    depends_on:
      - db
      - influxdb

  db:
    image: postgres:13
    container_name: traffic_forecast-postgres
    restart: unless-stopped
    environment:
      - POSTGRES_USER=monitor
      - POSTGRES_PASSWORD=forecast2022
      - POSTGRES_DB=traffic-forecast
    volumes:
      - ./data/postgres_db:/var/lib/postgressql/data

  influxdb:
    image: influxdb:latest
    container_name: traffic_forecast-influx
    restart: unless-stopped
    volumes:
      - ./data/influxdb/data:/var/lib/influxdb2:rw
    labels:
      - traefik.enable=true
      - traefik.http.routers.tf-influx.entryPoints=web-secure
      - traefik.http.routers.tf-influx.rule=Host(`tfm-influx.ranii.pro`)
      - traefik.http.routers.tf-influx.tls.certresolver=default
```

```
51      - traefik.http.services.tf-influx.loadbalancer.server.port=8086
52
```

Para desplegar los contenedores, según el archivo YAML definido, tenemos que ejecutar el siguiente comando:

```
docker compose up -f <filename> traffic_forecast
```

En el caso de querer profundizar más en el funcionamiento de Traefik, se puede consultar el siguiente enlace Traefik: Get Started

Capítulo 3

Diseño e implementación del sistema

En este capítulo se va a comentar más en detalle la implementación del sistema propuesto para la monitorización y predicción de tráfico en una red. Tal y como vimos en el capitulo anterior, se va a utilizar el lenguaje de programación Python, con el framework de desarollo FastAPI.

3.1 Descripción REST API

Tal y como pudimos ver en la sección 2.1, para la aplicación propuesta hemos elegido que siga la estructura de REST API, ya que es la más sencilla de implementar y la más flexible.

Para nuestro caso particular de monitorización del tráfico de una red, se han detectado tres "agentes" involucrados dentro del sistema. Dichos agentes son los siguientes:

- Redes (desde ahora `networks`). Corresponde con una red en la que queremos monitorizar el tráfico de ciertas interfaces.
- Interfaces de red (desde ahora `interfaces`). Corresponde con las interfaces de red de las que queremos monitorizar su tráfico.
- Muestras de monitorización (desde ahora `samples`). Corresponden con los diferentes datos de monitorización de tráfico asociados a una interfaz.

Definidos los agentes, se procede a diseñar el almacenamiento de la información de cada uno de ellos. Para ello, en primer lugar, nos decantamos por aplicar el concepto de CRUD en los agentes `networks` y `interfaces`.

CRUD: `networks`

Tal y como se refiere la definición de CRUD (**C**reate **R**ead **U**pdate **D**elete), en el caso del agente de `networks`, queremos definir el funcionamiento que tiene que seguir la aplicación cuando queramos referirnos a una red a monitorizar. Además, asumimos que dentro de una misma red podemos tener diferentes interfaces, que también serán dadas de altas en la aplicación. Por lo tanto, una única red puede tener muchas interfaces de red monitorizadas.

Dado que estamos planteando un servicio HTTP, debemos asignar una ruta (desde ahora `endpoint`) asociada a la colección de métodos de la CRUD de `networks`. En este caso, el endpoint de la colección sera: "`/networks`"

A modo de resumen, recogemos en la siguiente tabla el funcionamiento esperado de la aplicación, en función del método HTTP recibido y el recurso sobre el que se ejecute la petición HTTP.

Recurso	GET	POST	PATCH	DELETE
Colección de redes: `/networks`	Lista de redes dadas de alta	Añade una nueva red	*	*
Red en particular: `/networks/<id_net>`	Información de una red en particular	*	Modificamos la informacion de una red	Eliminamos una red

Tabla 3.1: CRUD `networks` (* equivale a "acceso denegado")

CRUD: `interfaces`

Para el caso del agente `interfaces`, definimos el funcionamiento de la aplicación. Entendemos una interfaz como aquellas interfaces de red que queremos monitorizar su tráfico dentro de una red, que previamente ya ha sido dada de alta en el sistema.

Cómo compromiso de diseño, se entiende que cada tendremos que dar de alta dos interfaces en el sistema, una interfaz para monitorizar los datos de transmisión (desde ahora, TX) y otra interfaz para monitorizar los datos de recepción (desde ahora, RX). De esta manera, tendríamos completamente monitorizada la interfaz de red.

Una interfaz monitorizada puede tener muchas muestras guardadas en el sistema, dichas muestras serán almacenadas en la base de datos para datos temporales (en nuestro caso InfluxDB).

Al igual que en el caso de `networks`, debemos asignar un `endpoint` asociado a la colección de los métodos CRUD de `interfaces`. En este caso, el `endpoint` asociado de la colección será: "`/networks/<id_net>/interfaces`".

Recurso	GET	POST	PATCH	DELETE
Colección de interfaces: `/networks/<id_net>/interfaces`	Listado de interfaces dentro de una red.	Añade una nueva interfaz a una red.	*	*
Interfaz en particular: `/networks/<id_net>/interfaces/<id_if>`	Información de una interfaz en particular.	*	Modificamos la información de una interfaz.	Eliminamos una interfaz

Tabla 3.2: CRUD `interfaces` (* equivale a "acceso denegado")

Métodos non CRUD

Por otro lado, la aplicación también estará compuesta por otros métodos que no encajan en el esquema CRUD. Estos métodos son:

- Muestras monitorizadas. Permite importar muestras al sistema. Corresponde con el `endpoint: /samples`
- Consultas de muestras monitorizadas. Permite acceder a la información de monitorización que esta almacenada en el sistema. Corresponde con el `endpoint: /query`
- Ejecución de una predicción. Permite generar una predicción de tráfico. Corresponde con el `endpoint: /forecast`

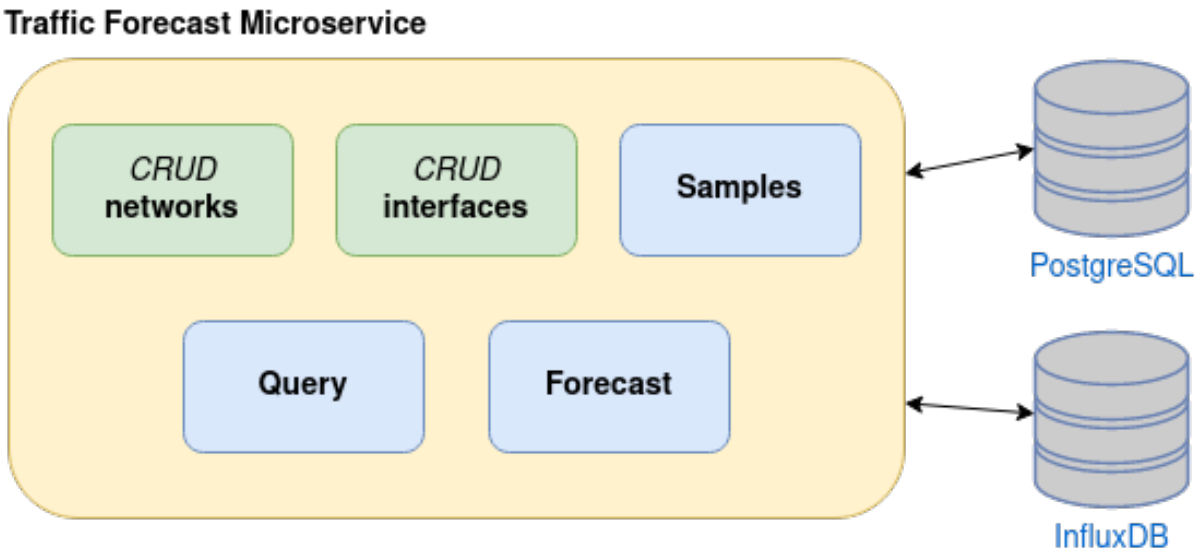

Figura 3.1: Diagrama resumen de la estructura de la aplicación.

3.2 Implementación de la aplicación

Tal y como se adelantó en la sección 2.3.2, el lenguaje utilizado para implementar la aplicación es Python, utilizando el framework de desarrollo FastAPI.

Para la implementación de la aplicación, se ha seguido la metodología de diseño "*Factory Pattern*". Dicha metodología permite estructurar un proyecto software de modo que la implementación de nuevas funcionalidades sea sencilla y no implique modificar partes ya validadas del sistema. Esto se basa en que permitimos a una clase crear objetos derivados de esta, en función de las necesidades del sistema. A grandes rasgos, intentamos emular el funcionamiento de una producción en cadena de una fabrica. Podemos encontrar más información en: [15] [16].

A consecuencia de esta metodología, el proyecto se estructura según la imagen 3.2. Dicha estructura tiene como raíz la carpeta `backend`, y esta se divide tal que:

- `migrations`. Carpeta que contiene las utilidades asociadas a la migración de la base de datos SQL. Permite realizar modificaciones en los modelos de datos y que dichas modificaciones se apliquen en la base de datos final.
- `src`. Carpeta en la que encontramos todo el código asociado a la aplicación.
  - `crud`. Contiene los archivos de funcionalidad para cada una de las CRUD (`networks` y `interfaces`).
  - `database`. Contiene los archivos asociados a los modelos de datos.
  - `noncrud`. Contiene la funcionalidad de los agentes que no siguen la estructura CRUD (`samples`, `query` y `forecast`).
  - `routes`. Contiene los `endpoints` de la aplicación, redirige a la funcionalidad en las carpetas `crud` y/o `noncrud`.
  - `schemas`. Contiene los diferentes esquemas de datos que se utilizan en la aplicación, tanto de entrada de datos del usuario, como de salida de datos del sistema.
  - `utils`. Contiene funciones de ayuda para simplificar el código, por ejemplo para manejar el InfluxDB.
  - `config.py`. Contiene las opciones configurables de la aplicación. Se definen dos entornos de ejecución: `development` o `production`.
  - `main.py`. Archivo principal del sistema, aquí comienza la ejecución de la aplicación.
- `Dockerfile`. Archivo que describe los pasos para crear un contenedor para nuestra aplicación.
- `requirements.txt`. Archivo que contiene los diferentes paquetes de Python utilizados.

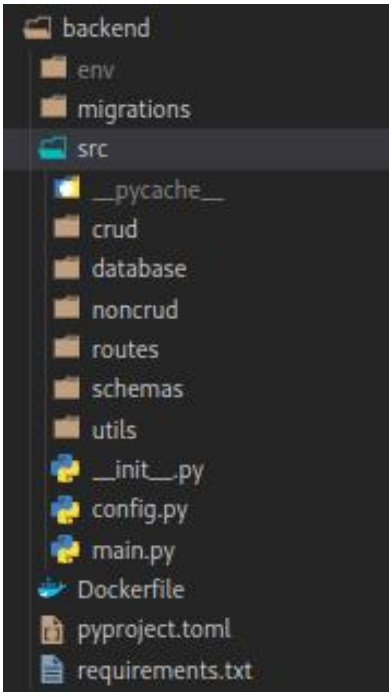

Figura 3.2: Captura estructura de carpetas de la aplicación.

De esta manera, en el caso de querer añadir una funcionalidad nueva, solo tendríamos que modificar los archivos pertinentes. Las modificaciones en archivos ya existentes serían mínimas. Permitiendo de esta manera un fácil mantenimiento y desarrollo sobre la aplicación.

3.2.1 Migración de base de datos SQL

Para permitir que sea sencillo extender los modelos de datos de la aplicación es necesario tener en cuenta que va a suceder si en algún momento decidimos cambiar algo de dichos modelos y que tenemos que hacer para que esos cambios se apliquen correctamente en la base de datos que estemos utilizando. Es por esto por lo que aparecen dos herramientas muy importantes: `ORM` y `migrations`.

En primer lugar, `ORM` (Object-relational Mappers) permite una abstracción de alto nivel que da lugar a describir un modelo de datos de SQL en Python, en vez de utilizar lenguaje SQL directamente. Además, permite acceder a la información de la base de datos de manera más sencilla, ya que para el programa es como si el modelo de datos fuera un objeto, pero internamente se están realizando las consultas SQL que sean necesarias. [17] [18]

En segundo lugar, `migrations` permite tener un control de versiones dentro de los modelos de datos que estemos usando en la aplicación. Además, permite aplicar automaticamente modificaciones en los modelos de datos, de modo que al ejecutar la aplicación, la base de datos actualizará su estructura (por ejemplo, añadir una nueva columna en una tabla) en función del modelo de datos que hayamos actualizado del `ORM`. [19]

3.2.2 OpenAPI Specification. `Swagger`

Una de las ventajas que supone el uso de FastAPI, es que de manera automática genera la documentación asociada a OpenAPI de la aplicación, dando lugar a que el endpoint `/docs` nos redirige directamente a la aplicación web `Swagger`, que nos muestra de una manera sencilla todos los endpoints asociados de nuestra aplicación, permitiendo realizar peticiones y comprobando la funcionalidad de esta.

De esta manera, si navegamos a dicha ruta (`/docs`), podemos ver como nos aparece una web similar a la de la figura 3.3.

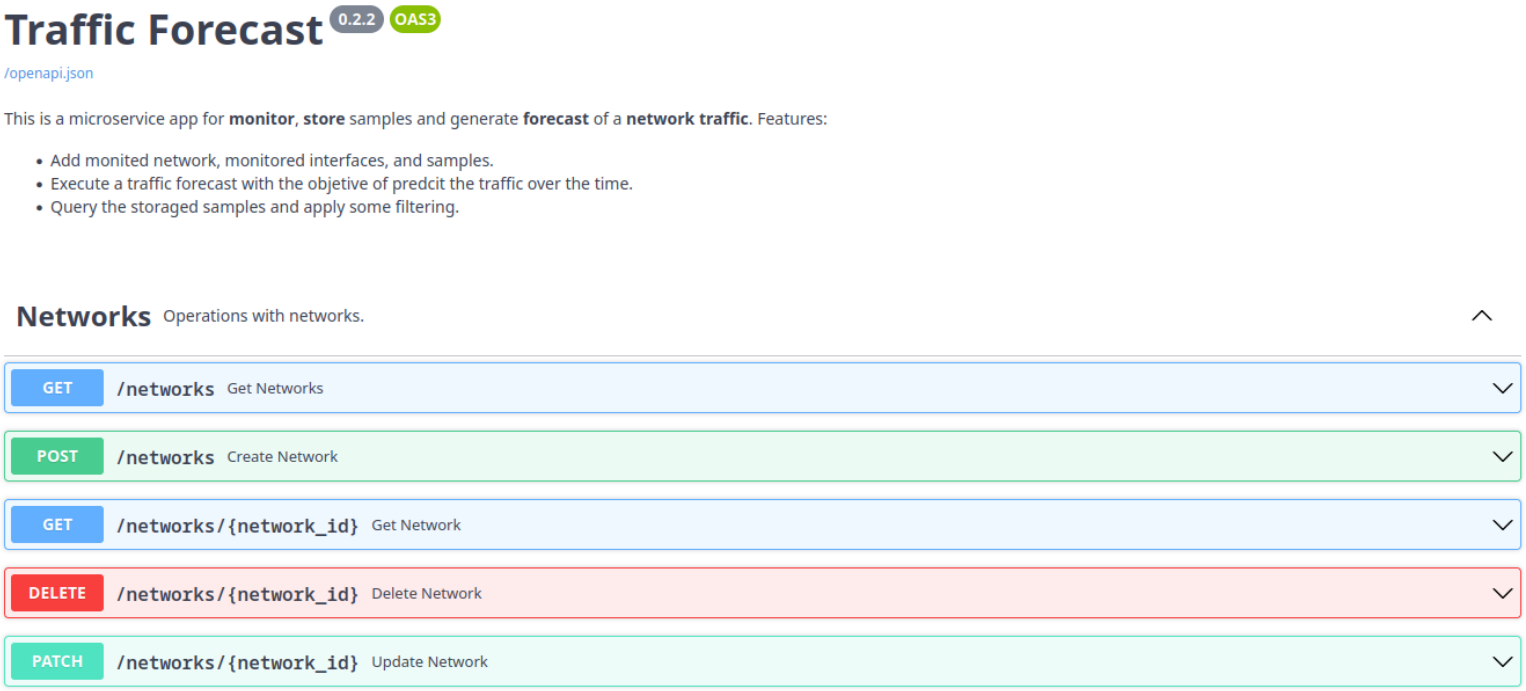

Figura 3.3: Captura Swagger. Vista por defecto.

A modo de ejemplo, si hiciéramos *click* en uno de los `endpoints`, podremos comprobar como se nos despliega información asociada. Por ejemplo, del esquema de datos de entrada, o el esquema de datos de salida, al igual que los códigos HTTP esperados. En la figura 3.4, podemos ver el detalle de la información necesaria para crear una red.

Además, una de las funcionalidades más interesantes que nos aporta Swagger, es la documentación necesaria sobre los esquemas de los JSON de salida de la aplicación. Esto es muy importante ya que nos permitiría conectar con otras aplicaciones, y presentar la información recibida. En la figura 3.5, podemos ver un detalle de algunos de los esquemas que se utilizan en la aplicación.

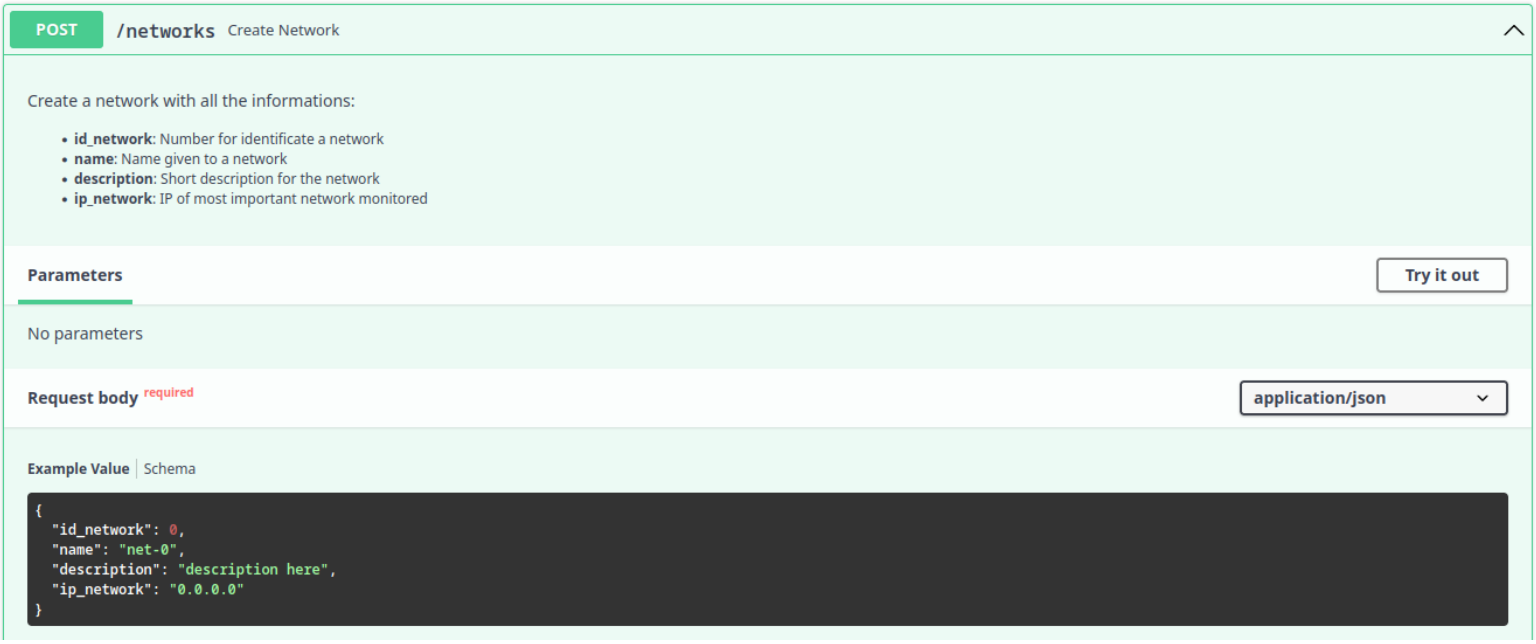

Figura 3.4: Captura Swagger. Detalle del método de crear red. (`POST` a `/networks`)

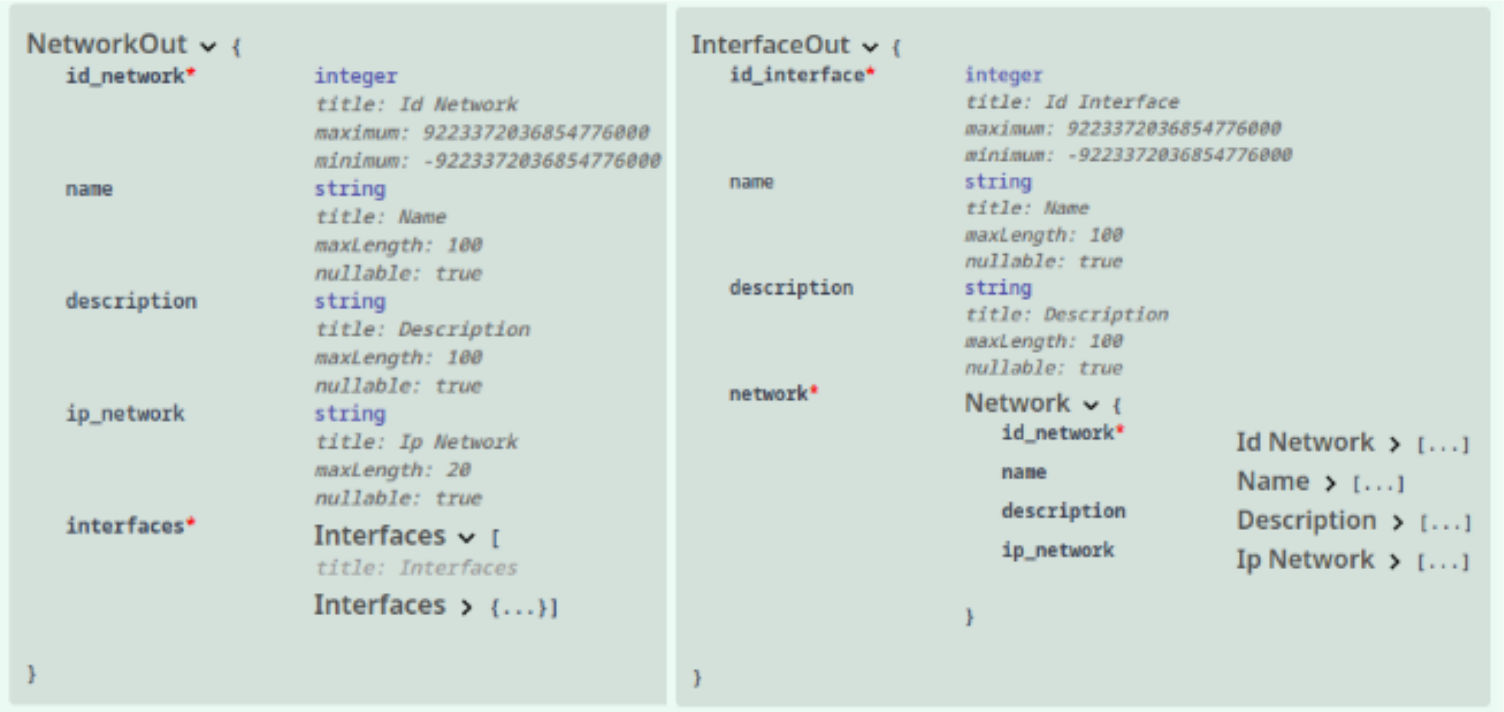

Figura 3.5: Captura Swagger. Detalle de esquemas JSON de salida de `networks` e `interfaces`.

3.3 Modelos de datos

Por otro lado, debemos definir los datos y su representación dentro de nuestra aplicación. Esto nos permite adaptar la información en función del tipo de base de datos que vayamos a consultar, y en función del modelo a utilizar. Para ello, diferenciamos entre los modelos que estarán almacenados en la base de datos tipo SQL, y los datos almacenados en la base de datos tipo serie temporal.

3.3.1 Base de datos SQL

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 3.6.

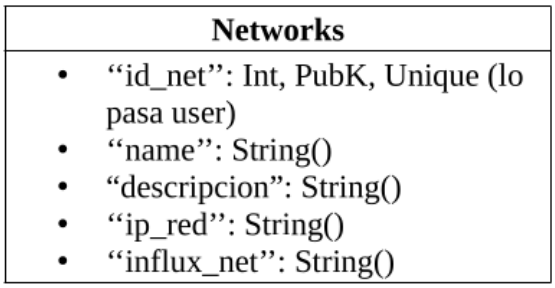

Figura 3.6: Modelo de datos para las redes a monitorizar. Equivale con la tabla ”networks”.

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 3.7.

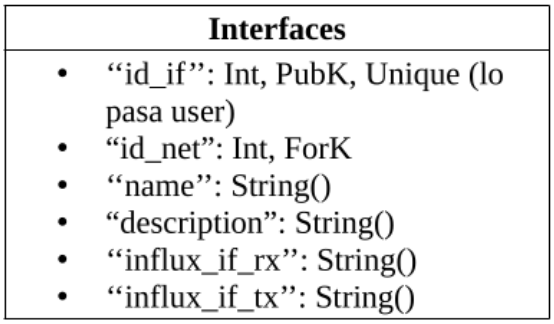

Figura 3.7: Modelos de datos para las interfaces a monitorizar. Equivale con la tabla ”interfaces”

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.

3.3.2 Base de datos InfluxDB

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 Predicción de tráfico

Uno de los objetivos más importantes que tiene satisfacer la aplicación es la posibilidad de predecir tráfico de red a partir de muestras de monitorización almacenadas en el sistema. Para ello, tal y como introducimos en 2.3.3, vamos a utilizar la herramienta de predicción Prophet.

En primer lugar, se decide que la predicción de tráfico va a consumir información que esta almacenada en el sistema. Por lo tanto, es necesario crear una red a la que monitorizar, y añadir muestras asociadas a la monitorización. En resumen, si quisiéramos ejecutar una predicción de tráfico desde cero, tendríamos que seguir los siguientes pasos dentro de la aplicación:

1. Creamos una red a monitorizar. Utilizaremos el endpoint POST: `/networks`.
2. Creamos una interfaz a monitorizar, dentro de la red previa. Utilizamos el endpoint POST: `/networks/<network_id>/interfaces`.
3. Importamos datos de monitorización. Utilizamos alguno de los endpoints asociados para importar datos, por ejemplo:
   - `/samples/<network_id>/import_topology`
   - `/samples/<network_id>/import_interface/<interface_id>`
4. Ejecutamos una predicción de tráfico. Usamos el endpoint POST: `/forecast`.

Llegados a este punto, una vez ejecutada la predicción y en el caso de que la predicción pueda completarse, el sistema nos devolverá los puntos asociados con toda la serie temporal, incluidos los valores predichos por el sistema. Como característica adicional, se ha implementado la posibilidad de elegir si queremos la salida de los datos como CSV o como JSON.

En el último paso, en el que ejecutamos la predicción de tráfico, podemos configurar en cierta manera cómo se va a realizar la predicción. Los parámetros a configurar son los siguientes: (ver figura 3.8)

- `id_network`
- `id_interface`
- `field`
- `days`
- `options`
  - `holidays_region`
  - `flexibility_trend`
  - `flexibility_season`
  - `flexibility_holidays`

Figura 3.8: Datos de entrada para la ejecución de una predicción de tráfico.

3.5 Endpoints

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
12. Docker Compose Overview
13. Traefik Webpage
14. Traefik Wiki
15. Factory Method - Python Design Patterns
16. FastAPI Doc: Bigger Applications
17. Object-relational Mappers (ORMs)
18. FastAPI Doc: SQL (Relational) Databases. ORM
19. FastAPI Doc: SQL (Relational) Databases. Migrations

Figuras

Anexos

Anexo I. Generación dataset sintético

Con el objetivo de tener un `dataset` que se adapte lo suficiente a la situación a validar por nuestra aplicación, se decide generar un `dataset` sintético, en el que podamos modificar ciertos parámetros que provoquen cambios en el mismo (por ejemplo, cambio de ruido, cambio en las tendencias...).

Para dar lugar a un `dataset` sintético, se utiliza el lenguaje de programación Python, y se siguen los siguientes pasos:

1. Generamos un archivo de dos columnas que simulan el tráfico de red en una interfaz durante un día.

   En este caso, tenemos dos opciones, la primera sería generar un dato de tráfico cada hora (ver figura 6.1), y la segunda generar un dato de tráfico cada 5 minutos (ver figura 6.2). Este último sería la opción más cercana a la realidad, ya que los datos de tráfico que aporta SNMP se obtienen cada 5 minutos por convenio.

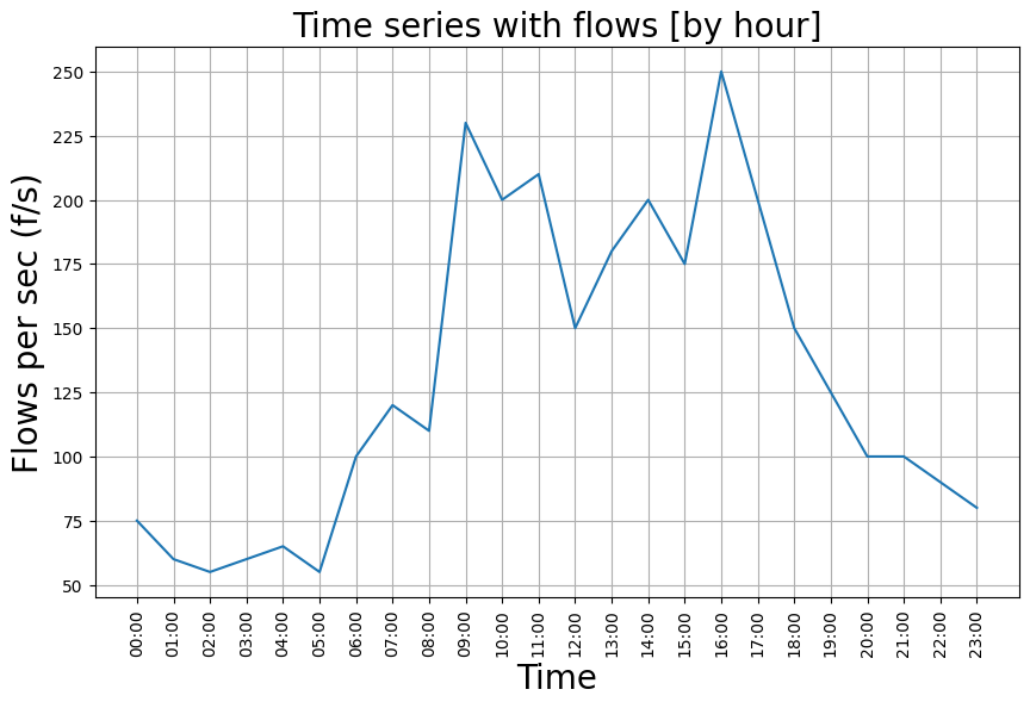

Figura 6.1: Simulación de tráfico de red. Un día, dividido por horas.

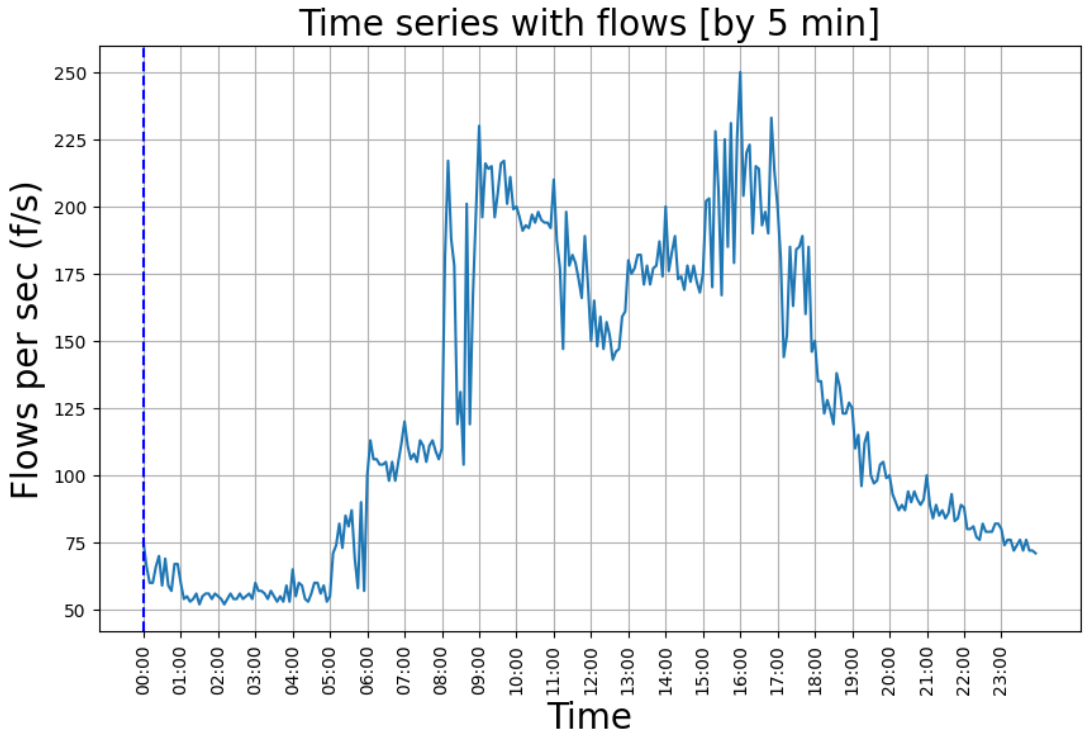

Figura 6.2: Simulación de tráfico de red. Un día, dividido por `slots` de 5 minutos.

2. A partir de estos datos, generamos una semana. En esta etapa, añadimos la primera regla: los fines de semana el tráfico se ve reducido, en comparación con el resto de la semana. Ver figura 6.3.

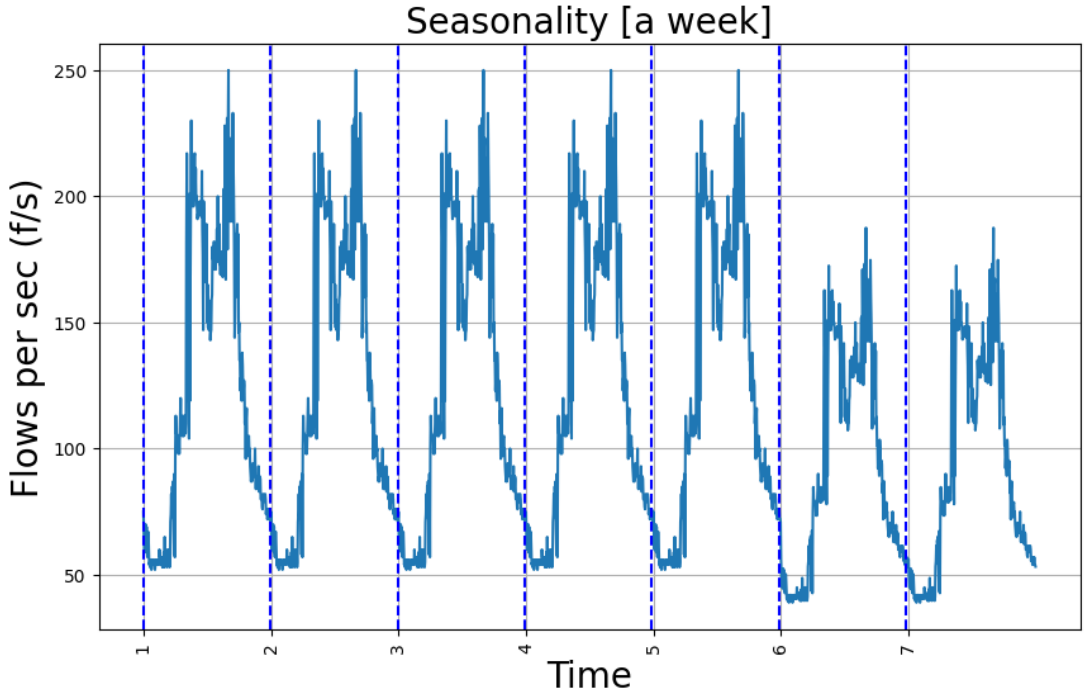

Figura 6.3: Simulación de tráfico de red. Una semana con `slots` de 5 minutos. Reducción de tráfico en el fin de semana

3. A los datos anteriores, podemos añadir las reglas de:

   - Ruido. Sumamos ruido blanco a la muestra sintética, de este modo podemos aumentar la varianza de los datos.
   - Tendencia. Aplicamos un crecimiento exponencial a la serie temporal. En el caso de tráfico en red, buscamos seguir el dato de CAGR (Tasa de crecimiento anual compuesto), propuesto por Cisco (enlace here), por lo que lo implementamos con una exponencial de crecimiento 27 %.

   El resultado al aplicar estas reglas se puede ver en la figura 6.4.

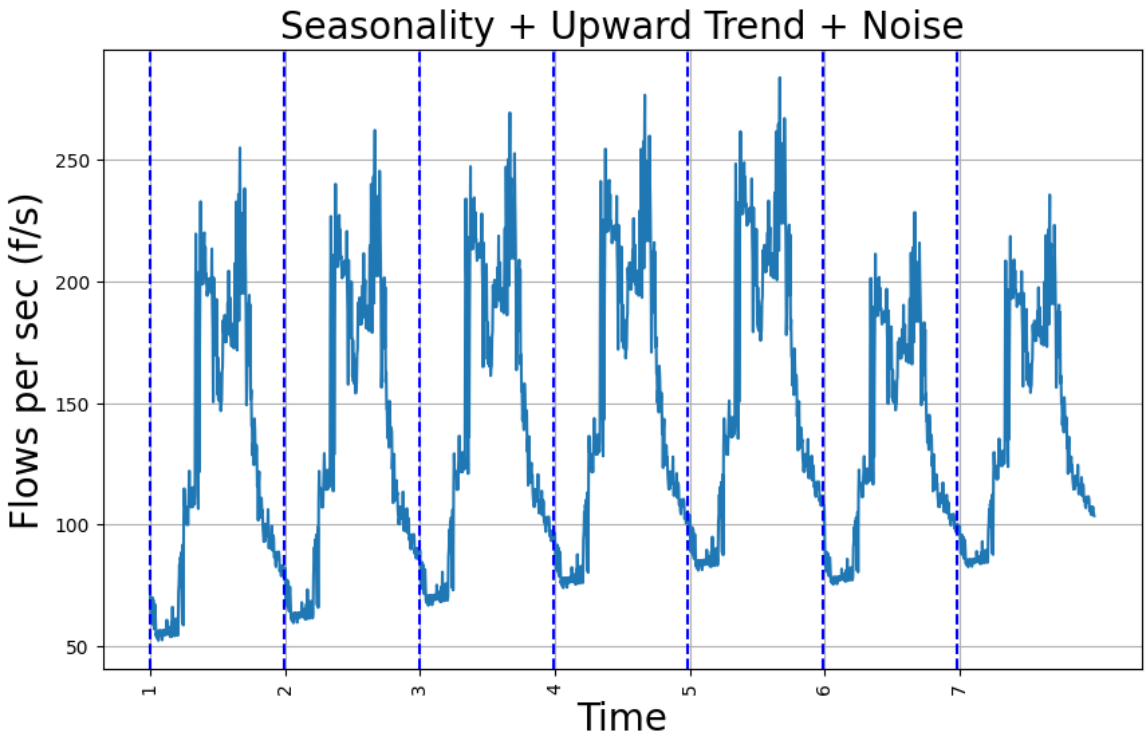

Figura 6.4: Simulación de tráfico de red. Una semana con `slots` de 5 minutos. Reducción de tráfico en fin de semana, tendencia exponencial y ruido blanco.

4. Llegados a este punto, podemos extender el paso anterior para tener muestras durante un mes completo. Esto daría lugar a la figura 6.5.

5. Por último, podemos extender el punto anterior si generamos un año completo. Además, en este punto podemos añadir otra regla más: el tráfico en verano aumenta. De esta manera, podemos observar un efecto de incremento en los meses de Junio, Julio y Agosto. El resultado de esta regla lo podemos ver en la figura 6.6.

   Además, si aplicamos el percentil 95 a la serie temporal de un año, podemos ver más claramente las diferentes reglas aplicadas (ver figura 6.7).

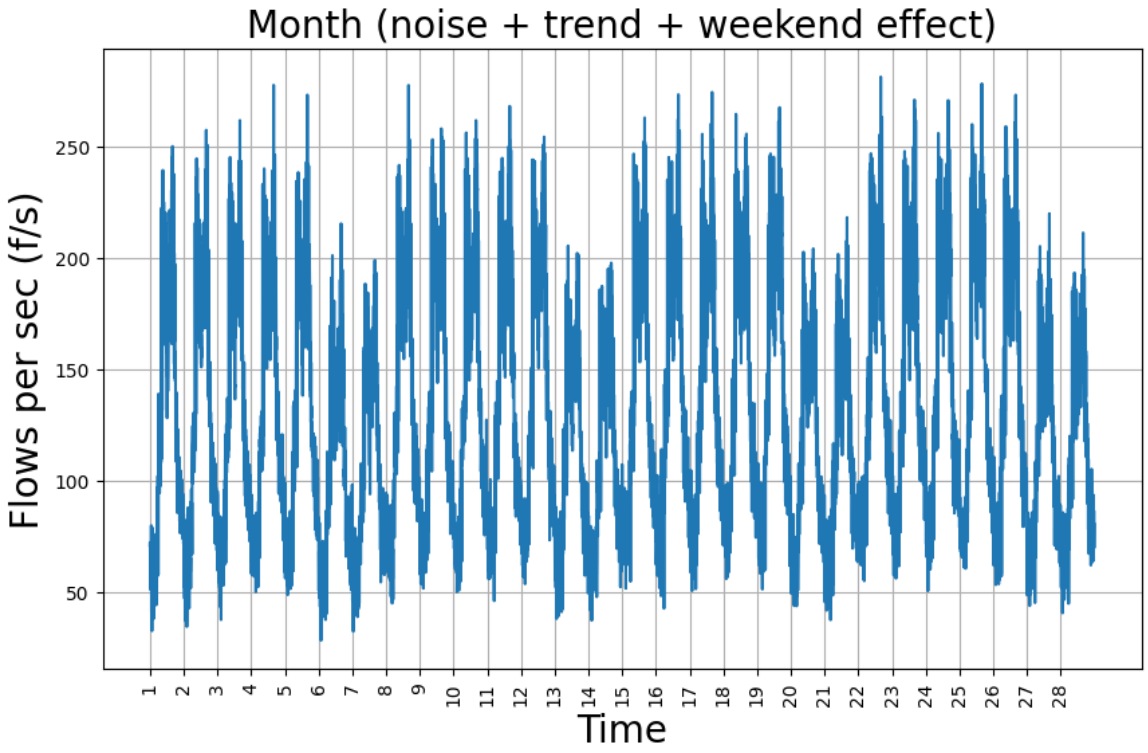

Figura 6.5: Simulación de tráfico de red. Un mes de datos, manteniendo reglas aplicadas.

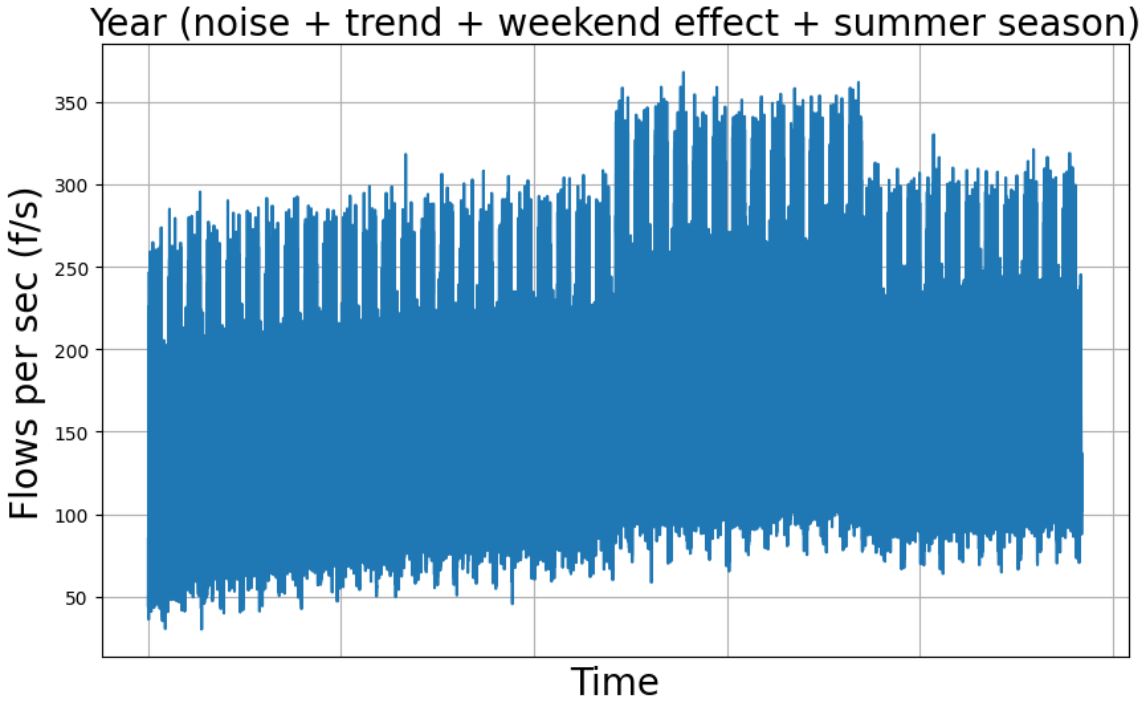

Figura 6.6: Simulación de tráfico de red. Un año de datos, se añade la regla del incremento de tráfico en verano.

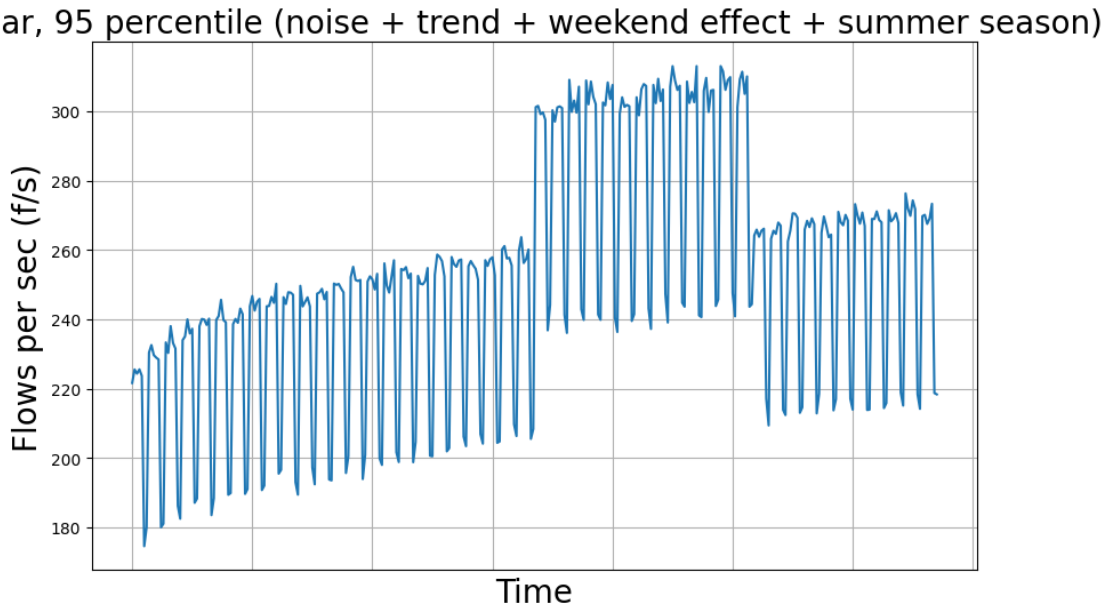

Figura 6.7: Simulación de tráfico de red. Un año de datos, se aplica el percentil 95 para cada día.

Una vez llegados a este punto, tenemos un dataset sintético de un año, en el que podemos modificar diferentes parámetros. De esta manera, podemos validar la predicción de tráfico de nuestra aplicación.