

**Escuela Técnica
Superior
de Ingeniería de
Telecomunicación**



**Universidad
Politécnica
de Cartagena**

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

14 de diciembre de 2022

TRABAJO FIN DE MÁSTER

Máster Universitario en Ingeniería de
Telecomunicación

Autor: Enrique Fernández Sánchez

Tutor: Pablo Pavón Mariño

Índice general

Índice de figuras	4
Índice de tablas	6
Listado de ejemplos	7
1 Introducción	8
1.1 Contexto del trabajo	8
1.2 Motivación	8
1.3 Descripción Global	8
1.4 Objetivos	8
1.5 Resumen capítulos de la memoria	8
2 Tecnologías empleadas	9
2.1 Arquitectura y microservicios	9
2.2 Bases de datos	11
2.2.1 Base de datos tipo relacional/SQL	11
2.2.2 Base de datos tipo “time series”/InfluxDB	12
2.3 Lenguajes y frameworks	13
2.3.1 Python	13
2.3.2 FastAPI	13
2.3.3 Prophet	14
2.4 Despliegue en producción	15
3 Diseño e implementación del sistema	20
3.1 Descripción REST API	20
3.2 Implementación de la aplicación	23
3.2.1 Migración de base de datos SQL	24
3.2.2 OpenAPI Specification. Swagger	25
3.3 Modelos de datos	27
3.3.1 Base de datos SQL	27
3.3.2 Base de datos InfluxDB	28
3.4 Predicción de tráfico de red	28
3.5 Rutas HTTP (endpoints)	30
3.5.1 Colección Networks	30
3.5.2 Colección Interfaces	33
3.5.3 Colección Samples	36
3.5.4 Colección Query samples	36
3.5.5 Colección Forecast	37

4	Validación del sistema	39
4.1	Crear una red a monitorizar	39
4.2	Crear una interfaz de red monitorizada	40
4.3	Cargar muestras en interfaz de red	41
4.4	Cargar una topología de red	44
4.5	Consultar datos de monitorización	45
4.6	Predicción de tráfico de un año	47
5	Conclusiones	50
5.1	Propuestas futuras	50
6	Bibliografía	51
	Enlaces y referencias	51
	Imagenes	52
Anexos		53
	Anexo I. Generación dataset sintético	53

Índice de figuras

2.1	Comparativa entre arquitectura de microservicios y arquitectura “monolítica”. [1]	10
2.2	Ejemplo de relaciones dentro de una base de datos SQL. [2]	11
2.3	Logotipo base de datos InfluxDB [3].	12
2.4	Logotipo Python [4].	13
2.5	Logotipo FastAPI [5].	14
2.6	Logotipo Prophet [6].	14
2.7	Docker .	15
2.8	Captura portal de gestión de Traefik. Detalle de enrutado hacia un servicio.	17
3.1	Diagrama resumen de la estructura de la aplicación.	22
3.2	Captura estructura de carpetas de la aplicación.	24
3.3	Captura Swagger. Vista por defecto.	25
3.4	Captura Swagger. Detalle del método de crear red. (POST a /networks)	26
3.5	Captura Swagger. Detalle de esquemas JSON de salida de networks e interfaces.	26
3.6	Modelo de datos para las redes a monitorizar. Equivale con la tabla ”networks”.	27
3.7	Modelos de datos para las interfaces a monitorizar. Equivale con la tabla ”interfaces”	27
3.8	Datos de entrada para la ejecución de una predicción de tráfico.	29
3.9	Parametros GET Networks	30
3.10	Respuesta GET Networks	30
3.11	Parametros POST Networks	31
3.12	Respuesta POST Networks	31
3.13	Respuesta GET Network	32
3.14	Respuesta DELETE Network	32
3.15	Parámetros PATCH Network	33
3.16	Parámetros GET Interfaces	33
3.17	Parámetros POST Interfaces	34
3.18	Parámetros POST Interfaces	34
3.19	Parámetros PATCH Interfaces	35
3.20	Parámetros POST Forecast	37
3.21	Respuesta del servidor de una predicción de tráfico, en formato CSV.	37
3.22	Respuesta del servidor de una predicción de tráfico, en formato JSON.	38
4.1	Petición para crear red a monitorizar.	39
4.2	Respuesta servidor después de crear red.	40
4.3	Petición para crear una interfaz de red monitorizada.	40
4.4	Respuesta del servidor después de crear una interfaz de red.	41
4.5	Dataset sintético de un año	42
4.6	Petición para añadir muestras de monitorización a una interfaz.	42

4.7	Respuesta del servidor una vez ha completado la petición de importar datos de monitorización.	43
4.8	Captura InfluxDB. Datos de monitorización correctamente subidos al sistema.	43
4.9	Petición para cargar una topología de red en la red 100.	44
4.10	Respuesta del servidor al cargar una topología de red.	44
4.11	Respuesta del servidor al cargar una topología de red.	45
4.12	Petición para consultar datos de monitorización del sistema.	46
4.13	Respuesta del servidor al consultar datos de monitorización, salida en formato JSON.	46
4.14	Petición para ejecutar una predicción de tráfico.	47
4.15	Respuesta del servidor de ejecución predicción (salida en CSV).	47
4.16	Respuesta del servidor de ejecución predicción (salida en JSON).	48
4.17	Gráfica datos de salida de la predicción de tráfico.	49
4.18	Gráfica datos de salida de la predicción de tráfico. A la izquierda de la linea roja, datos de muestras; a la derecha, datos predichos por el sistema.	49
6.1	Simulación de tráfico de red. Un día, dividido por horas.	53
6.2	Simulación de tráfico de red. Un día, dividido por slots de 5 minutos.	54
6.3	Simulación de tráfico de red. Una semana con slots de 5 minutos. Reducción de tráfico en el fin de semana	54
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.	55
6.5	Simulación de tráfico de red. Un mes de datos, manteniendo reglas aplicadas.	56
6.6	Simulación de tráfico de red. Un año de datos, se añade la regla del incremento de tráfico en verano.	56
6.7	Simulación de tráfico de red. Un año de datos, se aplica el percentil 95 para cada día.	57

Índice de tablas

3.1 CRUD networks (* equivale a “acceso denegado”)	21
3.2 CRUD interfaces (* equivale a “acceso denegado”)	21

Listado de ejemplos

2.1	Archivo configuración de contenedores para entorno de desarrollo.	16
2.2	Archivo configuración de contenedores para entorno de producción.	18

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 está 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 el 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.

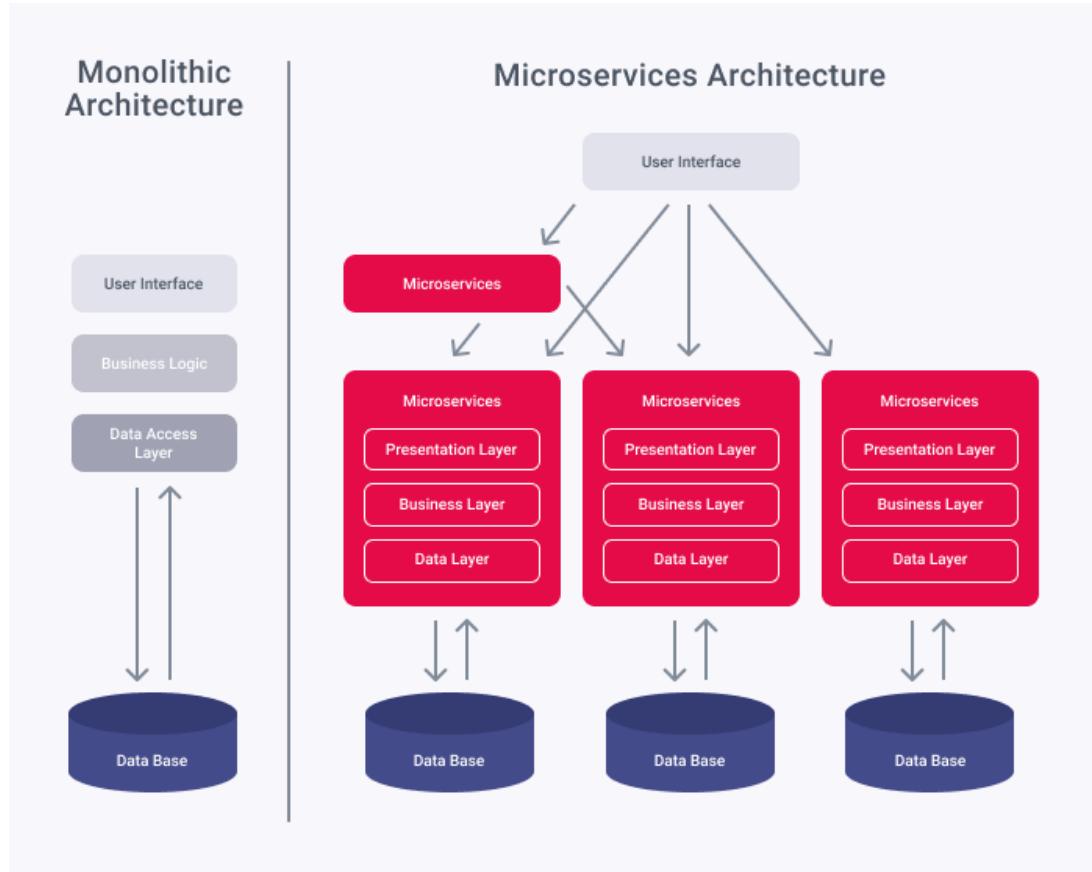


Figura 2.1: Comparativa entre arquitectura de microservicios y arquitectura “monolítica”. [1]

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 monitoreados.

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".

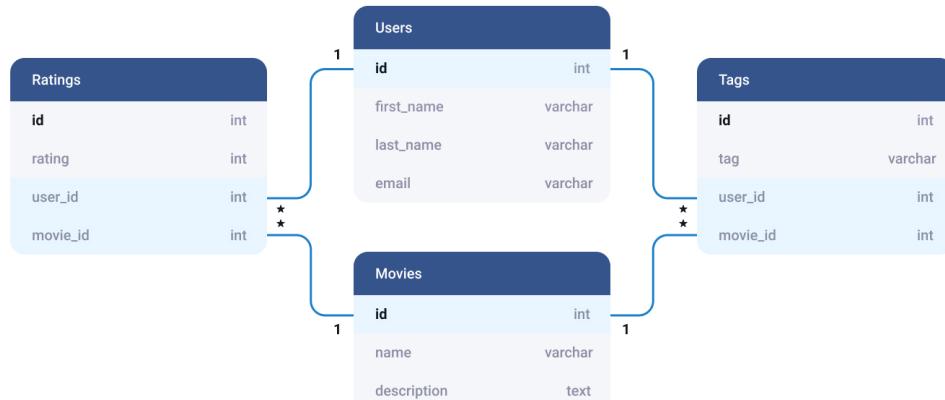


Figura 2.2: Ejemplo de relaciones dentro de una base de datos SQL. [2]

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 está 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 estás 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.

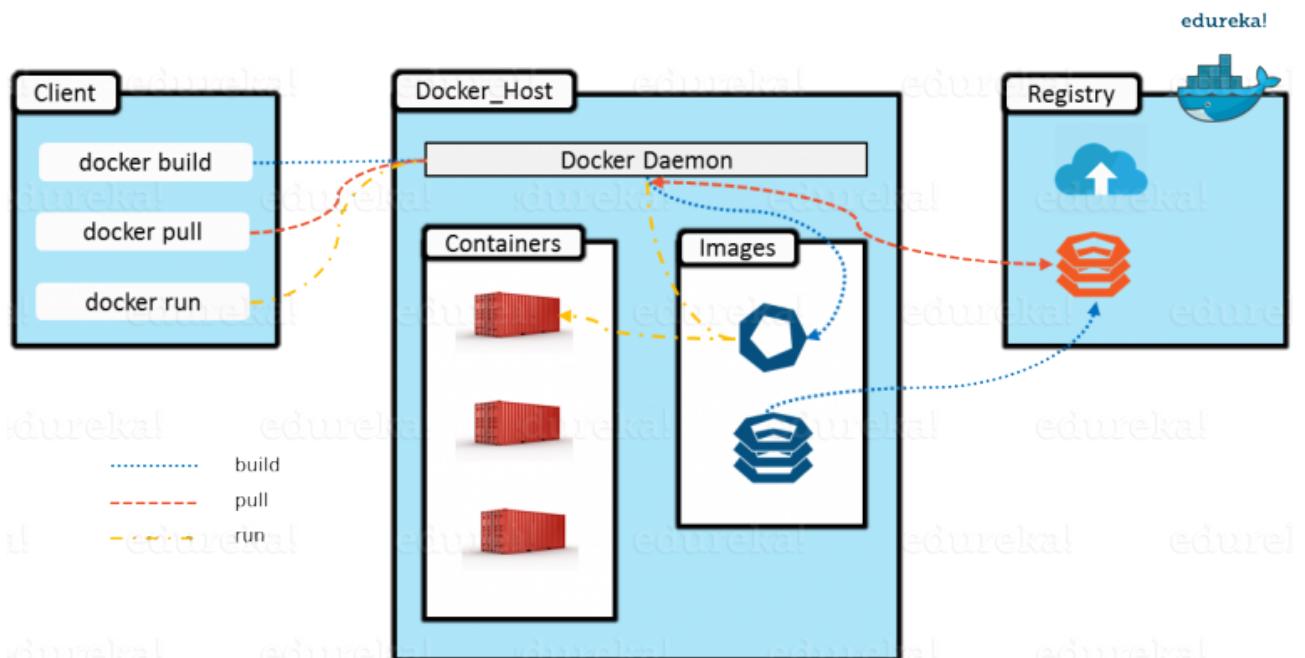


Figura 2.7: Docker

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.

```

1 version: '3.8'
2
3 services:
4   traffic_forecast:
5     build: ./backend
6     ports:
7       - 5000:5000
8     environment:
9       - POSTGRES_USER=monitor
10      - POSTGRES_PASSWORD=forecast2022
11      - POSTGRES_DB=traffic-forecast
12      - INFLUX_TOKEN=*****
13      - INFLUX_ORG=e-lighthouse
14      - INFLUX_BUCKET=traffic-forecast
15      - SECRET_KEY=upct2022_sk
16      - FASTAPI_CONFIG=development
17     volumes:
18       - ./backend:/app
19     depends_on:
20       - db
21       - influxdb
22
23 db:
24   image: postgres:13
25   expose:
26     - 5432
27   environment:
28     - POSTGRES_USER=monitor
29     - POSTGRES_PASSWORD=forecast2022
30     - POSTGRES_DB=traffic-forecast
31   volumes:
32     - ./data/postgres_db:/var/lib/postgresql/data
33
34 influxdb:
35   image: influxdb:latest
36   volumes:
37     - ./data/influxdb/data:/var/lib/influxdb2:rw
38   ports:
39     - 8086:8086
40

```

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 está 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.

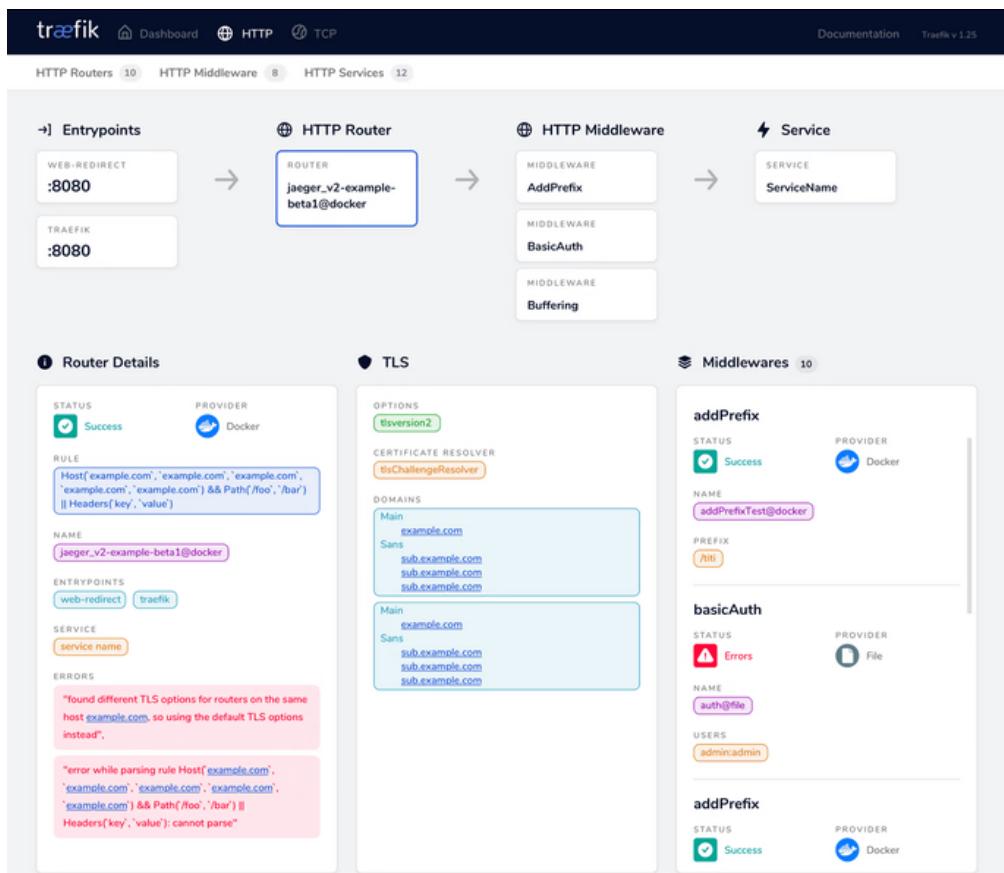


Figura 2.8: Captura portal de gestión de Traefik. Detalle de enruteado hacia un servicio.

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.

```

1 version: '3.8'
2
3 services:
4   traffic_forecast:
5     build: ./backend
6     container_name: traffic_forecast-api
7     restart: unless-stopped
8     environment:
9       - POSTGRES_USER=monitor
10      - POSTGRES_PASSWORD=forecast2022
11      - POSTGRES_DB=traffic-forecast
12      - INFLUX_URL=https://tfm-influx.ranii.pro:8443/
13      - INFLUX_TOKEN=*****
14      - INFLUX_ORG=e-lighthouse
15      - INFLUX_BUCKET=traffic-forecast
16      - SECRET_KEY=upct2022_sk
17     volumes:
18       - ./backend:/app
19     labels:
20       - traefik.enable=true
21       - traefik.http.routers.tf-api.entryPoints=web-secure
22       - traefik.http.routers.tf-api.rule=Host('tfm-api.ranii.pro')
23       - traefik.http.routers.tf-api.tls.certresolver=default
24       - traefik.http.services.tf-api.loadbalancer.server.port=5000
25   depends_on:
26     - db
27     - influxdb
28
29 db:
30   image: postgres:13
31   container_name: traffic_forecast-postgres
32   restart: unless-stopped
33   environment:
34     - POSTGRES_USER=monitor
35     - POSTGRES_PASSWORD=forecast2022
36     - POSTGRES_DB=traffic-forecast
37   volumes:
38     - ./data/postgres_db:/var/lib/postgresql/data
39
40 influxdb:
41   image: influxdb:latest
42   container_name: traffic_forecast-influx
43   restart: unless-stopped
44   volumes:
45     - ./data/influxdb/data:/var/lib/influxdb2:rw
46   labels:
47     - traefik.enable=true
48     - traefik.http.routers.tf-influx.entryPoints=web-secure
49     - traefik.http.routers.tf-influx.rule=Host('tfm-influx.ranii.pro')
50     - 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 capítulo anterior, se va a utilizar el lenguaje de programación Python, con el framework de desarrollo 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: <code>/networks</code>	Lista de redes dadas de alta	Añade una nueva red	*	*
Red en particular: <code>/networks/<id_net></code>	Información de una red en particular	*	Modificamos la información 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.

Como 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: <code>/networks/<id_net>/interfaces</code>	Listado de interfaces dentro de una red.	Añade una nueva interfaz a una red.	*	*
Interfaz en particular: <code>/networks/<id_net>/interfaces/<id_if></code>	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 está almacenada en el sistema. Corresponde con el endpoint: /query
- Ejecución de una predicción. Permite generar una predicción de tráfico. Corresponde con el endpoint: /forecast

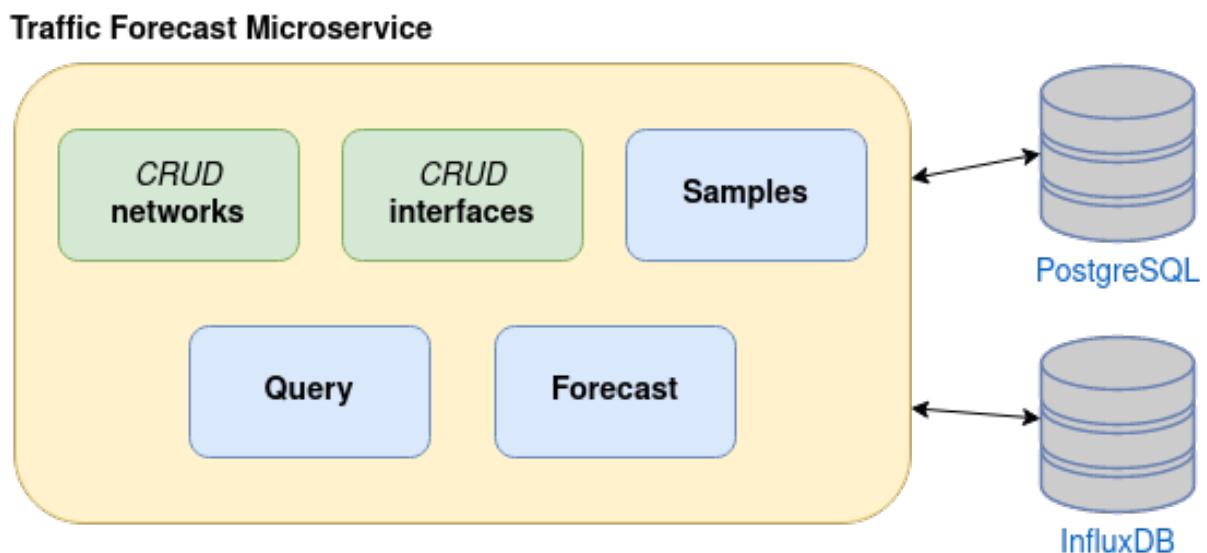


Figura 3.1: Diagrama resumen de la estructura de la aplicación.

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 fábrica. 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.

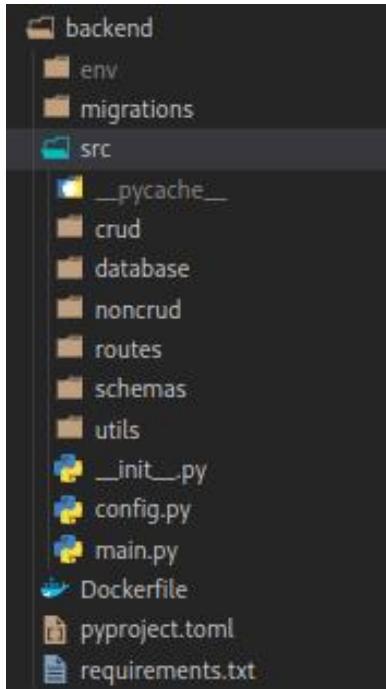


Figura 3.2: Captura estructura de carpetas de la aplicación.

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 automáticamente 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.

The screenshot shows the Swagger UI interface for a microservice app named "Traffic Forecast". At the top, there's a header with the app name and version (0.2.2 OAS3). Below it, a link to `/openapi.json`. The main content area has a heading "Networks" with the subtext "Operations with networks". It lists several API endpoints:

- GET /networks** Get Networks
- POST /networks** Create Network
- GET /networks/{network_id}** Get Network
- DELETE /networks/{network_id}** Delete Network
- PATCH /networks/{network_id}** Update Network

Each endpoint entry includes a dropdown arrow icon on the right side.

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.

POST /networks Create Network ^

Create a network with all the informations:

- **id_network**: Number for identifyate a network
- **name**: Name given to a network
- **description**: Short description for the network
- **ip_network**: IP of most important network monitored

Parameters

No parameters Try it out

Request body required application/json

Example Value | **Schema**

```
{
  "id_network": 0,
  "name": "net-0",
  "description": "description here",
  "ip_network": "0.0.0.0"
}
```

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

NetworkOut ↴ {

- id_network*** integer
- name** string
- description** string
- ip_network** string
- interfaces*** Interfaces ↴ [
- title: Interfaces
- Interfaces > {...}]

}

InterfaceOut ↴ {

- id_interface*** integer
- name** string
- description** string
- network*** Network ↴ {
- id_network*** Id Network > [...]
- name** Name > [...]
- description** Description > [...]
- ip_network** Ip Network > [...]

}

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.

Networks
<ul style="list-style-type: none"> • “id_net”: Int, PubK, Unique (lo pasa user) • “name”: String() • “descripcion”: String() • “ip_red”: String() • “influx_net”: String()

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.

Interfaces
<ul style="list-style-type: none"> • “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 3.7: 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 qué 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 numérico de link_count en ese periodo de 5 minutos.

3.4 Predicción de tráfico de red

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 está 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`. Identificador de red, se obtiene una vez creas la red a monitorizar o haciendo un GET a `/networks`.
- `id_interface`. Identificador de interfaz de red, es necesario que este asociado con una red a monitorizar. Se obtiene una vez se crea la interfaz, o bien haciendo un GET a `/networks/<id_network>/interfaces`.
- `field`. Permite seleccionar el tráfico de una interfaz, elegimos entre recepción (“RX”) o transmisión (“TX”).
- `days`. Número de días que queremos predecir. Si quisiéramos predecir el tráfico dentro de un año, pondremos 365.
- `options`. Opciones configurables del modelo de predicción de tráfico. [20]
 - `holidays_region`. Permite elegir que días festivos se aplican en la predicción. Esto permite detectar sucesos en días de festividad de las diferentes regiones. En el caso de España, tendremos que utilizar “ES”.
 - `flexibility_trend`. Parámetro que más impacto tiene en la predicción. Determina la flexibilidad respecto a una tendencia, es decir, cuanto puede cambiar una tendencia en los diferentes puntos. El valor por defecto es 0.05, aunque se puede modificar en un rango de [0.001, 0.5]. Es un parámetro que afecta a la penalización por regulación. [21]
 - `flexibility_season`. Controla la flexibilidad de los efectos estacionales. El valor por defecto es 10, aunque se puede modificar en un rango de [0.01, 10].
 - `flexibility_holidays`. Controla la flexibilidad de los efectos por días festivos. El valor por defecto es 10, aunque se puede modificar en un rango de [0.01, 10].



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

3.5 Rutas HTTP (`endpoints`)

En esta sección se pretende recapitular todos los endpoints disponibles en la aplicación, además de sus esquemas de entrada y de salida. Es importante recordar que la aplicación tiene la aplicación documental Swagger implementada, por lo que accediendo a la ruta /doc, podemos ver toda la documentación asociada a la API.

3.5.1 Colección Networks

1. Ruta /networks:

- **Método HTTP:** GET
- **Descripción:** permite listar todas las redes monitorizadas del sistema.

Name	Description
page	Default value : 1 integer (query) minimum: 1
size	Default value : 50 integer (query) maximum: 100 minimum: 1

Figura 3.9: Parámetros GET Networks

Code	Description	Code	Description
200	Successful Response	200	Successful Response

Media type: application/json

Example Value | **Schema**

```
{
  "items": [
    {
      "id_network": 9223372036854776000,
      "name": "string",
      "description": "string",
      "ip_network": "string",
      "interfaces": [
        {
          "id_interface": 9223372036854776000,
          "name": "string",
          "description": "string"
        }
      ]
    },
    {
      "total": 0,
      "page": 1,
      "size": 1
    }
  ]
}
```

Page[NetworkOut] v {

items* Items v [

title: Items

NetworkOut v {

id_network* Id Network > [...]

name Name > [...]

description Description > [...]

ip_network Ip Network > [...]

interfaces* Interfaces > [...]

}]

total* integer title: Total minimum: 0

page* integer title: Page minimum: 1

size* integer title: Size minimum: 1

}

Figura 3.10: Respuesta GET Networks

2. Ruta /networks:

- **Método HTTP:** POST
- **Descripción:** permite crear una nueva red en el sistema.

Parameters

No parameters

Request body required

[Example Value](#) | [Schema](#)

```
{
  "id_network": 0,
  "name": "net-0",
  "description": "description here",
  "ip_network": "0.0.0.0"
}
```

Figura 3.11: Parámetros POST Networks

Code	Description	Code	Description
200	<p>Successful Response</p> <p>Media type</p> <p>application/json</p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ "id_network": 9223372036854776000, "name": "string", "description": "string", "ip_network": "string", "interfaces": [{ "id_interface": 9223372036854776000, "name": "string", "description": "string" }] }</pre>	200	<p>Successful Response</p> <p>Media type</p> <p>application/json</p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>NetworkOut ▾ { id_network* Id Network > [...] name Name > [...] description Description > [...] ip_network Ip Network > [...] interfaces* Interfaces > [...] }</pre>

Figura 3.12: Respuesta POST Networks

3. Ruta /networks/{network_id}:

- **Método HTTP:** GET
- **Descripción:** devuelve la información de una red, dado un identificador de red.
- **Parámetros:** network_id, como identificador de red.

Code	Description	Description
200	Successful Response	Successful Response
	Media type application/json <small>Controls Accept header.</small>	Media type application/json <small>Controls Accept header.</small>
	Example Value Schema	Example Value Schema
	<pre>{ "id_network": 9223372036854776000, "name": "string", "description": "string", "ip_network": "string", "interfaces": [{ "id_interface": 9223372036854776000, "name": "string", "description": "string" }] }</pre>	<pre>NetworkOut ▾ { id_network* Id Network > [...] name Name > [...] description Description > [...] ip_network Ip Network > [...] interfaces* Interfaces ▾ [title: Interfaces Interfaces ▾ { id_interface* Id Interface > [...] name Name > [...] description Description > [...] }] }</pre>

Figura 3.13: Respuesta GET Network

4. Ruta /networks/{network_id}:

- **Método HTTP:** DELETE
- **Descripción:** elimina una red del sistema, dado un identificador de red.
- **Parámetros:** network_id, como identificador de red.

200	Successful Response
	Media type application/json <small>Controls Accept header.</small>
	Example Value Schema
	<pre>Status ▾ { message* string title: Message }</pre>

Figura 3.14: Respuesta DELETE Network

5. Ruta /networks/{network_id}:

- **Método HTTP:** PATCH
- **Descripción:** actualiza una red del sistema, dado un identificador de red.
- **Parámetros:** network_id, como identificador de red.
- **Respuesta:** equivalente a 3.13, con la información ya modificada.

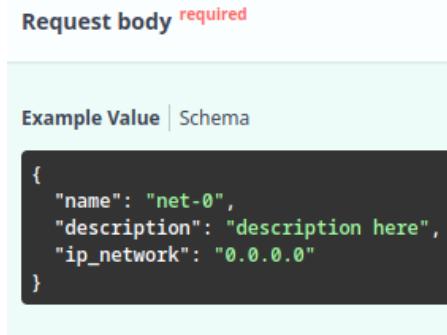


Figura 3.15: Parámetros PATCH Network

3.5.2 Colección Interfaces

1. Ruta /networks/{network_id}/interfaces:

- **Método HTTP:** GET
- **Descripción:** devuelve un listado de interfaces asociadas a una red.
- **Parámetros:** network_id, como identificador de red. Similar a figura 3.9.

Code	Description	Description
200	Successful Response	Successful Response
	Media type <input type="button" value="application/json"/> Controls Accept header.	Media type <input type="button" value="application/json"/> Controls Accept header.
	Example Value Schema	Example Value Schema
	<pre>{ "items": [{ "id_interface": 9223372036854776000, "name": "string", "description": "string", "network": { "id_network": 9223372036854776000, "name": "string", "description": "string", "ip_network": "string" } }], "total": 0, "page": 1, "size": 1 }</pre>	<pre>Page[InterfaceOut] ▾ { items* ▾ { Items ▾ [title: Items InterfaceOut ▾ { id_interface* ▾ { Id Interface > [...] Name > [...] Description > [...] Network > [...] } }] } total* page* size* }</pre>

Figura 3.16: Parámetros GET Interfaces

2. Ruta /networks/{network_id}/interfaces:

- **Método HTTP:** POST
- **Descripción:** permite crear una interfaz monitorizada, asociada a una red.
- **Parámetros:** network_id, como identificador de red. Similar a figura 3.11.

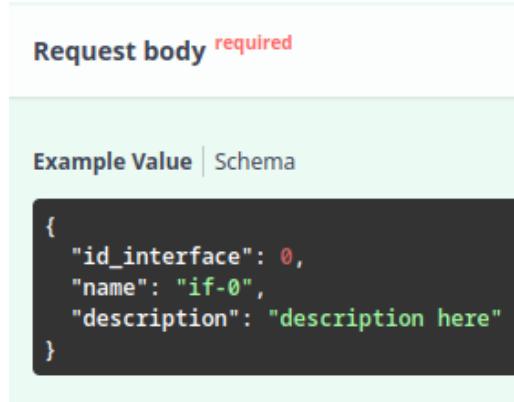


Figura 3.17: Parámetros POST Interfaces

Code	Description	Description
200	Successful Response	<p>Successful Response</p> <p>Media type</p> <p>application/json</p> <p>Controls Accept header.</p> <p>Example Value Schema</p> <pre>{ "id_interface": 9223372036854776000, "name": "string", "description": "string", "network": { "id_network": 9223372036854776000, "name": "string", "description": "string", "ip_network": "string" } }</pre> <p>InterfaceOut ↴ {</p> <p>id_interface* integer title: Id Interface maximum: 9223372036854776000 minimum: -9223372036854776000</p> <p>name string title: Name maxLength: 100 nullable: true</p> <p>description string title: Description maxLength: 100 nullable: true</p> <p>network* Network ↴ {</p> <p>id_network* integer name string description string ip_network string</p> <p>Id Network > [...] Name > [...] Description > [...] Ip Network > [...]</p>

Figura 3.18: Parámetros POST Interfaces

3. Ruta /networks/{network_id}/interfaces/{interface_id}:

- **Método HTTP:** GET
- **Descripción:** devuelve la información de una interfaz, asociada a una red.
- **Parámetros:** network_id e interface_id, como identificadores.
- **Respuesta:** similar a la obtenida en la respuesta de POST a interfaces, ver figura 3.18.

4. Ruta /networks/{network_id}/interfaces/{interface_id}:

- **Método HTTP:** DELETE
- **Descripción:** elimina una interfaz de red.
- **Parámetros:** network_id e interface_id, como identificadores.
- **Respuesta:** similar a la obtenida en el caso de networks, ver figura 3.14.

5. Ruta /networks/{network_id}/interfaces/{interface_id}:

- **Método HTTP:** PATCH
- **Descripción:** actualiza una interfaz del sistema, dado un identificador.
- **Parámetros:** network_id e interface_id, como identificadores.
- **Respuesta:** equivalente a 3.18, con la información ya modificada.

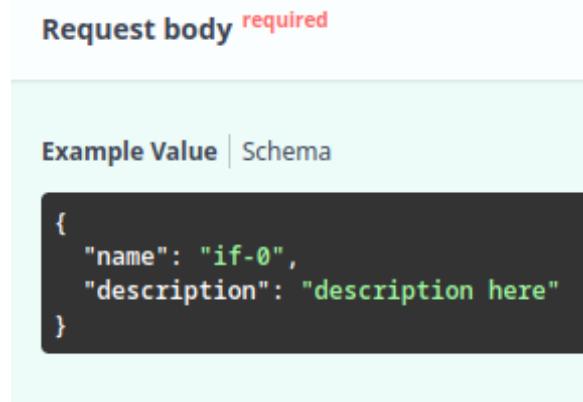


Figura 3.19: Parámetros PATCH Interfaces

3.5.3 Colección Samples

1. Ruta /samples/{network_id}/import_topology:

- **Método HTTP:** POST
- **Descripción:** permite subir un archivo tipo .csv que contenga información de más de una interfaz. Las columnas permitidas del .csv son: timestamp, interface, TX y RX. El sistema dará de alta las interfaces asociadas y cargará en la base de datos las muestras para cada una de las interfaces involucradas.
- **Parámetros:** network_id como identificador de la red a la que se va a subir los datos de monitorización.
- **Respuesta:** una vez se carga la topología, el servidor nos devuelve un mensaje confirmando que se han creado las interfaces de red y que se han subido correctamente los datos a la base de datos.

2. Ruta /samples/{network_id}/import_interface/{interface_id}:

- **Método HTTP:** POST
- **Descripción:** permite subir un archivo tipo .csv que contenga muestras de tráfico de una única interfaz, especificando si es de tipo RX o TX. El sistema subirá la información de las muestras a la base de datos, asociadas a la interfaz de red que especificamos como parámetro.
- **Parámetros:** network_id e interface_id, como identificadores tanto de red como de interfaz monitorizada.
- **Respuesta:** una vez cargados los datos de monitorización, el servidor devuelve un mensaje confirmando que se han subido correctamente los datos.

3.5.4 Colección Query samples

1. Ruta /query:

- **Método HTTP:** GET
- **Descripción:** permite consultar la información de monitorización almacenada en el sistema. Además, permite modificar si queremos la salida como CSV o como JSON.
- **Parámetros:** network_id e interface_id, como identificadores de los datos a consultar, field para elegir entre interfaz de transmisión (TX) o interfaz de recepción (RX), y to_csv para elegir entre si queremos la salida en formato CSV (campo en true) o la salida en formato JSON (campo en false).
- **Respuesta:** el servidor devuelve los datos almacenados en el sistema. En función del parámetro to_csv, la salida será un archivo con formato CSV o un archivo con formato JSON.

3.5.5 Colección Forecast

1. Ruta /forecast:

- **Método HTTP:** POST
- **Descripción:** permite ejecutar una predicción de tráfico de red sobre una interfaz en específico. Más información en la sección 3.4
- **Parámetros:** además del campo body que vemos en la figura 3.20, tenemos otro parámetro que permite elegir el formato de respuesta del servidor, este parámetro es `to_csv` y funciona de manera equivalente que en la ruta de /query.
- **Respuesta:** el servidor devuelve la serie temporal completa, tanto la almacenada como los datos predichos. En función del parámetro `to_csv`, la salida tendrá formato CSV (ver figura 3.21) o bien JSON (ver figura 3.22).

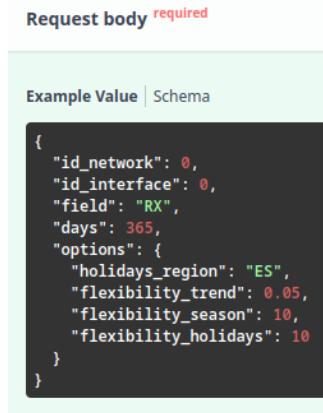


Figura 3.20: Parámetros POST Forecast

Server response	
Code	Details
200	<p>Response body</p> <pre> ds,yhat,yhat_lower,yhat_upper,trend,trend_lower,trend_upper 2016-01-02,130.07371694170945,128.6345343124219,131.5475132590133,130.4092962534722,130.4092962534722,130.4092962534722 2016-01-03,124.91834922199348,123.42425323379909,126.37687284988988,130.51623367105677,130.51623367105677,130.51623367105677 2016-01-04,125.44874317158398,124.0347319256543,126.84861705748202,130.62317108864133,130.62317108864133,130.62317108864133 2016-01-05,130.81058009380456,129.3736623842338,132.1875130730626,130.7301085062259,130.7301085062259,130.7301085062259 2016-01-06,131.1668118719372,129.67818917587798,132.5523260780218,130.83704592381045,130.83704592381045,130.83704592381045 2016-01-07,131.07807351306556,129.7279016224091,132.5648262470814,130.943983341395,130.943983341395,130.943983341395 2016-01-08,131.19622306906209,129.6213827429884,132.70656392158517,131.05092075897954,131.05092075897954,131.05092075897954 2016-01-09,130.88152673696558,129.43508905790986,132.34178954711632,131.15785817656413,131.15785817656413,131.15785817656413 2016-01-10,125.74215233286952,124.27660415979358,127.2233674402817,131.2647955941487,131.2647955941487,131.2647955941487 2016-01-11,126.320806680479494,124.87439622848211,127.70657136342398,131.37173301173323,131.37173301173323,131.37173301173323 2016-01-12,131.75713736838665,130.26516714282236,133.18366084126865,131.4786704293178,131.4786704293178,131.4786704293178 2016-01-13,131.94830962811926,130.5210898278658,133.5042376017436,131.58560784690235,131.58560784690235,131.58560784690235 2016-01-14,132.10530858132478,130.60865352961613,133.60735006993957,131.6925452644869,131.6925452644869,131.6925452644869 2016-01-15,132.25761463287142,130.90634420206518,133.64380460423143,131.79948268207147,131.79948268207147,131.79948268207147 2016-01-16,131.9694284096115,130.58332056032748,133.47311930714457,131.90642009965603,131.90642009965603,131.90642009965603 2016-01-17,126.82319078007218,125.4220989620355,128.251766210797274,132.01335751724056,132.01335751724056,132.01335751724056 2016-01-18,127.42120166383975,126.04268863251424,128.9235642476408,132.12029493482515,132.12029493482515,132.12029493482515 2016-01-19,132.89773855744755,131.39034947653815,134.30039436122829,132.2272323524097,132.2272323524097,132.2272323524097 2016-01-20,133.0922006977247,131.56779709828456,134.6615535281576,132.33416976999425,132.33416976999425,132.33416976999425 2016-01-21,133.24521144076604,131.86375593826475,134.64573402397713,132.4411071875788,132.4411071875788,132.4411071875788 2016-01-22,133.38653747937147,132.05671870682798,134.83580143074215,132.54804460516337,132.54804460516337,132.54804460516337 2016-01-23,133.078194352028,131.62292254475383,134.55951661084882,132.65498202274793,132.65498202274793,132.65498202274793 </pre>

Figura 3.21: Respuesta del servidor de una predicción de tráfico, en formato CSV.

Server response	
Code	Details
200	<p>Response body</p> <pre>{ "0": { "ds": "2016-01-02T00:00:00.000", "yhat": 130.0737169417, "yhat_lower": 128.5746625173, "yhat_upper": 131.6747859504, "trend": 130.4092962535, "trend_lower": 130.4092962535, "trend_upper": 130.4092962535 }, "1": { "ds": "2016-01-03T00:00:00.000", "yhat": 124.918349222, "yhat_lower": 123.5160311448, "yhat_upper": 126.36964403, "trend": 130.5162336711, "trend_lower": 130.5162336711, "trend_upper": 130.5162336711 }, "2": { "ds": "2016-01-04T00:00:00.000", "yhat": 125.4487431716, "yhat_lower": 123.9264583571, "yhat_upper": 126.9700318558 } }</pre>

Figura 3.22: Respuesta del servidor de una predicción de tráfico, en formato JSON.

Capítulo 4

Validación del sistema

En este capítulo vamos a demostrar y validar el funcionamiento del sistemas, además de verificar que este cumple con los objetivos propuestos para el trabajo. A modo de resumen, las pruebas realizadas son las siguientes:

1. Crear una red a monitorizar.
2. Crear una interfaz dentro de una red a monitorizar.
3. Cargar muestras en interfaz de red.
4. Cargar una topología de red sobre una red a monitorizar.
5. Consultar datos de monitorización almacenados en el sistema.
6. Ejecutar una predicción de tráfico de un año.

4.1 Crear una red a monitorizar

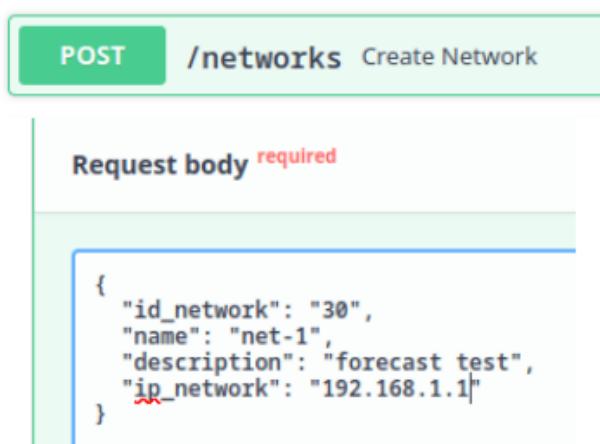
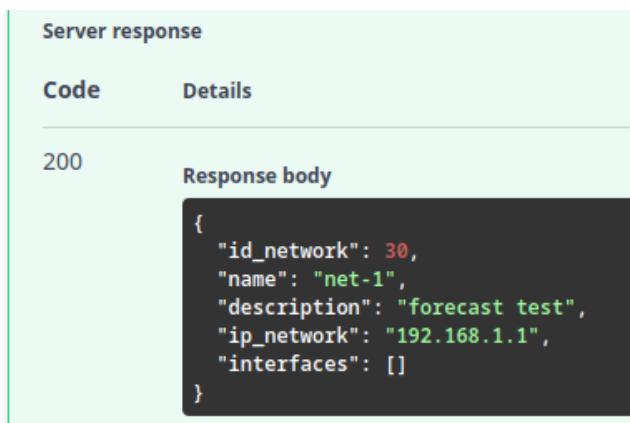


Figura 4.1: Petición para crear red a monitorizar.

Tal y como vemos en la figura 4.1, realizamos una petición POST a la ruta /networks, con el request body que vemos también en la figura. De este modo, en el sistema se creará una red a monitorizar llamada net-1 con el identificador 30.

La respuesta recibida por el servidor ante esta petición la podemos ver en la figura 4.2.



```

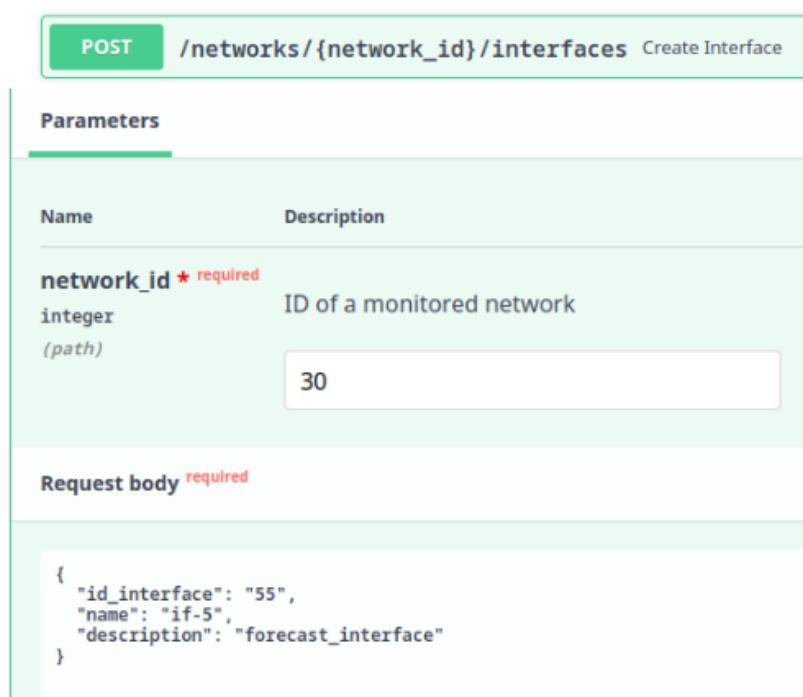
Server response

Code Details
200 Response body
{
  "id_network": 30,
  "name": "net-1",
  "description": "forecast test",
  "ip_network": "192.168.1.1",
  "interfaces": []
}

```

Figura 4.2: Respuesta servidor después de crear red.

4.2 Crear una interfaz de red monitorizada



POST /networks/{network_id}/interfaces Create Interface

Parameters	
Name	Description
network_id * required integer (path)	ID of a monitored network <input type="text" value="30"/>

Request body required

```
{
  "id_interface": "55",
  "name": "if-5",
  "description": "forecast_interface"
}
```

Figura 4.3: Petición para crear una interfaz de red monitorizada.

A la red que acabamos de crear (`net-1`, con id 30), le añadimos una interfaz de red monitorizada con el identificador 55. Dicha interfaz de red monitorizada tiene por nombre `if-5`. Ver figura 4.3.

La respuesta recibida por el servidor ante esta petición, la podemos ver en la figura 4.4.

Server response

Code	Details
200	Response body <pre>{ "id_interface": 55, "name": "if-5", "description": "forecast_interface", "network": { "id_network": 30, "name": "net-1", "description": "forecast test", "ip_network": "192.168.1.1" } }</pre>

Figura 4.4: Respuesta del servidor después de crear una interfaz de red.

4.3 Cargar muestras en interfaz de red

El siguiente paso para validar el sistema es cargar datos de monitorización al sistema. Para ello se planteó la opción de generar un dataset de datos sintéticos en el que se puedan aplicar ciertos efectos de tendencia o estacionales, de modo que se pueda validar la correcta predicción del tráfico. Los pasos que se han seguido para generar dicho dataset se pueden ver en anexo I (6).

En este caso, utilizamos un dataset de dos años con muestras de tráfico cada 5 minutos. En la figura 4.5 podemos ver que forma tiene los datos de monitorización. Este dataset lo subimos al sistema para que se añadan a la interfaz monitorizada con identificación 55.

En la figura 4.6 podemos ver como se selecciona el archivo de muestras sintéticas, y además se realiza la petición para que las muestras se cargue con muestras de monitorización de recepción (ver detalle en el campo `field`).

Una vez enviamos la petición, el servidor tarda aproximadamente 5-6 segundos en dar una respuesta. Para este caso, el dataset de dos años tiene un peso de 5.8 MB. En la figura X podemos ver la respuesta recibida por el servidor.

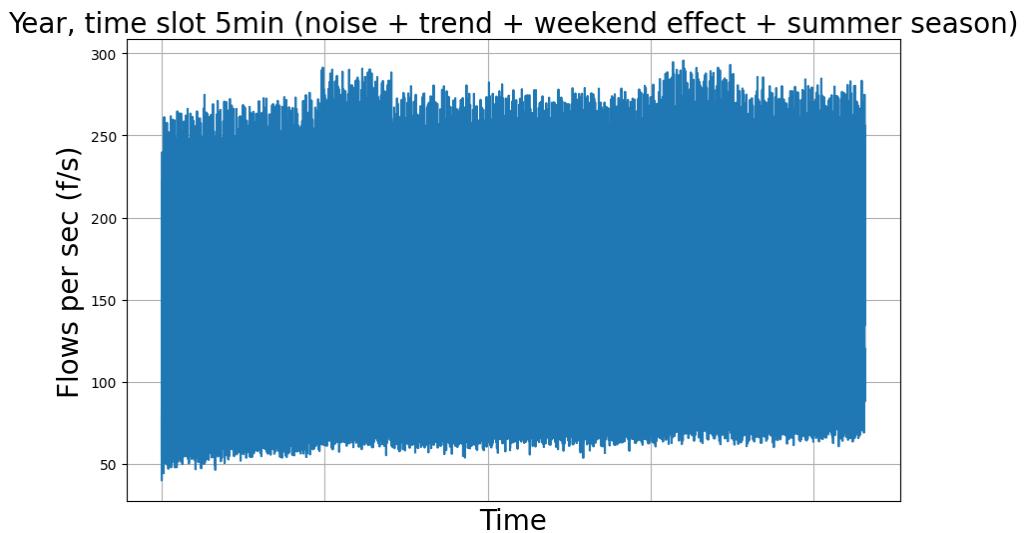


Figura 4.5: Dataset sintético de un año

POST /samples/{network_id}/import_interface/{interface_id} Import Interface

Parameters	
Name	Description
network_id * required string (path)	30
interface_id * required string (path)	55
field string (query)	RX

Request body required

file * required string(\$binary) <input type="button" value="Browse..."/> output_synthetic_dataset.csv

Figura 4.6: Petición para añadir muestras de monitorización a una interfaz.

```
Server response

Code Details

200 Response body
{
  "message": "Interface imported"
}
```

Figura 4.7: Respuesta del servidor una vez ha completado la petición de importar datos de monitorización.

En este punto, es interesante comprobar si de verdad se han subido los datos de monitorización al sistema. Recapitulando en los modelos de datos, los datos de monitorización están almacenados en la base de datos InfluxDB, por lo que podemos hacer uso del portal web que viene por defecto en la aplicación para comprobar si se han subido los datos correctamente. Una vez accedemos al portal web, y seleccionamos el bucket de nuestra aplicación, procedemos a buscar los datos de monitorización que acabamos de subir. En la figura 4.8 podemos ver una captura de pantalla de la aplicación web de InfluxDB y los datos de monitorización.

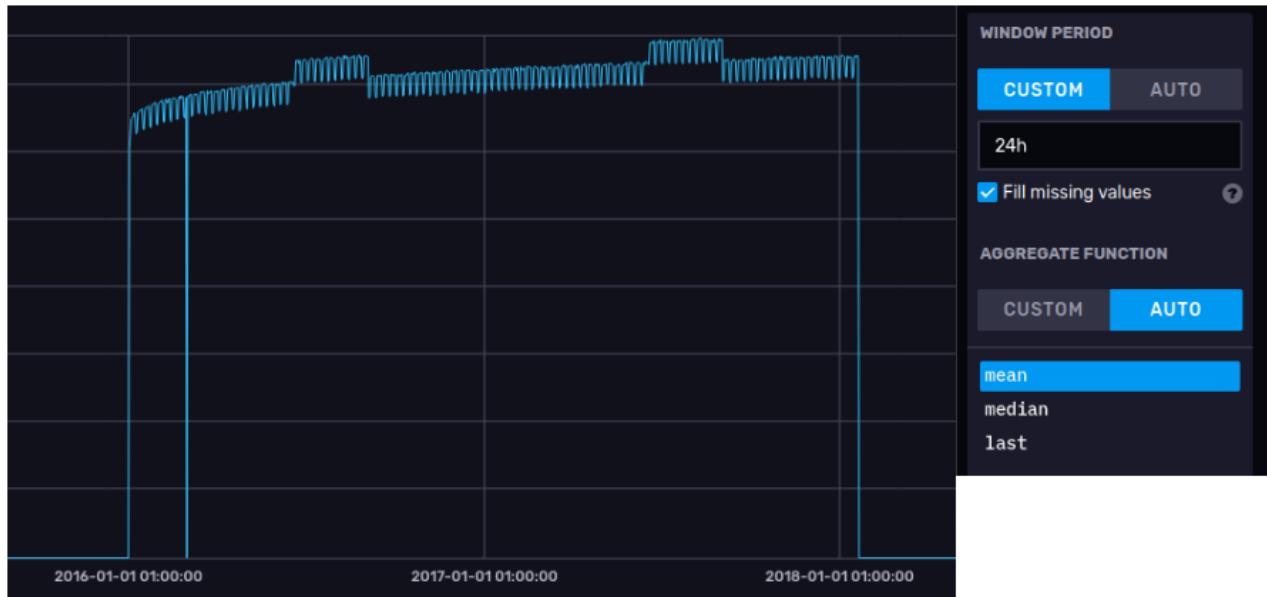


Figura 4.8: Captura InfluxDB. Datos de monitorización correctamente subidos al sistema.

4.4 Cargar una topología de red

Otra forma de cargar muestras de monitorización a una red es utilizando la ruta: `/samples/{network_id}/import_topology`, ya que permite subir información de más de una interfaz de forma simultánea. Para ello, tenemos que subir un archivo CSV que tenga las siguientes columnas:

- `timestamp`. Valor temporal de la muestra, cada 5 minutos.
- `interface`. Nombre de la interfaz monitorizada.
- `TX`. Valor de tráfico de red en la interfaz de transmisión.
- `RX`. Valor de tráfico de red en la interfaz de recepción.

Para este ejemplo, hemos generado un pequeño archivo de monitorización que vamos a subir a una red de monitorización que previamente tenemos creada (en este caso, red con identificación 100). Una vez el sistema lea el archivo, creará las interfaces que sean necesarias para dar lugar a la topología. Una vez termine, subirá los datos de monitorización para cada una de las muestras.

POST /samples/{network_id}/import_topology Import Topology

Parameters

Name	Description
network_id * required	100
string (path)	

Request body required

file * required	Browse... network0-test_data.csv
string(\$binary)	

Figura 4.9: Petición para cargar una topología de red en la red 100.

Server response

Code	Details
200	Response body
	{ "message": "Successfully imported monitored data of topology on network 100" }

Figura 4.10: Respuesta del servidor al cargar una topología de red.

Para comprobar que se ha subido correctamente los datos, vamos al portal web de InfluxDB. En la figura 4.11 podemos ver como los datos se han subido correctamente. En esta captura también podemos comprobar como se agrupan los datos dentro de la base de datos de tipo serie temporal.



Figura 4.11: Respuesta del servidor al cargar una topología de red.

4.5 Consultar datos de monitorización

Otra funcionalidad importante del sistema es que además de poder cargar información de tráfico, podemos recuperar dicha información de modo que el servidor nos devuelva la información de monitorización asociada a la petición que nosotros le hemos realizado. Para ello, utilizamos la ruta /query con el método GET.

En este caso, vamos a validar que el sistema correctamente nos puede dar la información de monitorización de una red con identificador 100, que tiene una interfaz con identificador 171, y elegimos que nos devuelva los datos de recepción (field equivalente a RX). Ver figura 4.12.

La respuesta del servidor puede ser en formato JSON o CSV en función del parámetro `to_csv` de la petición. En la figura 4.13 podemos ver como el servidor nos devuelve los datos de la interfaz solicitada en formato JSON.

GET /query/ Query Interface

Parameters

Name	Description
network_id * required string (query)	100
interface_id * required string (query)	171
field string (query)	RX
to_csv boolean (query)	false

Figura 4.12: Petición para consultar datos de monitorización del sistema.

Server response

Code	Details
200	Response body <pre>{ "0": { "time": "2022-11-02T07:10:00.000Z", "value": 5, "field": "link_count", "interface": "171-if-1-RX" }, "1": { "time": "2022-11-02T07:15:00.000Z", "value": 20, "field": "link_count", "interface": "171-if-1-RX" }, "2": { "time": "2022-11-02T07:20:00.000Z", "value": 25, "field": "link_count", "interface": "171-if-1-RX" }, "3": { "time": "2022-11-02T07:25:00.000Z", "value": 100, "field": "link_count", "interface": "171-if-1-RX" } }</pre>

Figura 4.13: Respuesta del servidor al consultar datos de monitorización, salida en formato JSON.

4.6 Predicción de tráfico de un año

En este punto, queremos realizar una predicción de tráfico de red a partir de las muestras subidas al sistema en el apartado 4.3. Las muestras subidas van desde 2016 hasta inicios de 2018, por lo que queremos predecir desde inicios de 2018 hasta inicios de 2019.

En la figura 4.14 podemos ver como se realiza la petición para la predicción de tráfico.

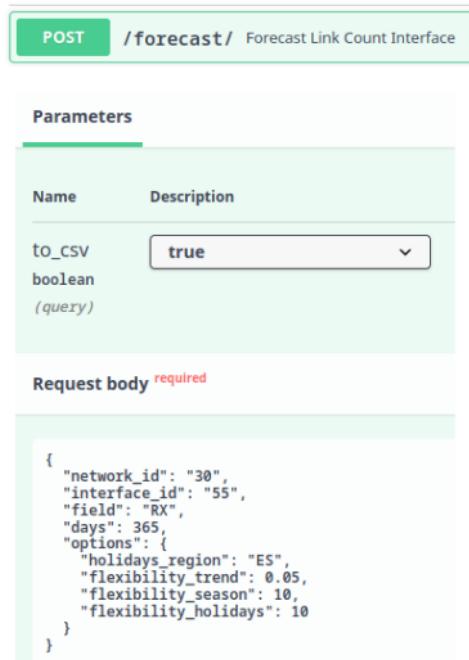


Figura 4.14: Petición para ejecutar una predicción de tráfico.

```
Server response
Code Details
200 Response body
ds,yhat,yhat_lower,yhat_upper,trend,trend_lower,trend_upper
2016-01-02,130.07371694170945,128.6345343124219,131.5475132950133,130.4092962534722,130.4092962534722,130.4092962534722
2016-01-03,124.91834922199348,123.42425323779909,126.37687284988988,130.51623367105677,130.51623367105677,130.51623367105677
2016-01-04,125.44874317158398,124.0347319256543,126.84861705748202,130.62317108864133,130.62317108864133,130.62317108864133
2016-01-05,130.81058009380456,129.3736623842338,132.1875130730626,130.7301085062259,130.7301085062259,130.7301085062259
2016-01-06,131.1668118719372,129.67818917587798,132.5523260788218,130.83704592381045,130.83704592381045,130.83704592381045
2016-01-07,131.07807351306556,129.7279016224091,132.5648262470814,130.943983341395,130.943983341395,130.943983341395
2016-01-08,131.19622306906209,129.6213827429884,132.70656392158517,131.05092075897954,131.05092075897954,131.05092075897954
2016-01-09,130.88152673696558,129.43508905790986,132.34178954711632,131.15785817656413,131.15785817656413,131.15785817656413
2016-01-10,125.74215233286952,124.27968415979358,127.2233674402817,131.2647955941487,131.2647955941487,131.2647955941487
2016-01-11,126.32088680479494,124.87439622848211,127.70657136342398,131.37173301173323,131.37173301173323,131.37173301173323
2016-01-12,131.75713736838665,130.26516714202236,133.18366084126865,131.4786704293178,131.4786704293178,131.4786704293178
2016-01-13,131.94830962811926,130.5210898278658,133.5042376017436,131.58560784690235,131.58560784690235,131.58560784690235
2016-01-14,132.16530858132478,130.60865352961613,133.60735006693957,131.6925452644869,131.6925452644869,131.6925452644869
2016-01-15,132.25761463287142,130.90634428206518,133.64380468423143,131.79948268207147,131.79948268207147,131.79948268207147
2016-01-16,131.9694284096115,130.58332056032748,133.47311930714457,131.90642009965603,131.90642009965603,131.90642009965603
2016-01-17,126.82319078007218,125.42208986220355,128.25176621079274,132.01335751724056,132.01335751724056,132.01335751724056
2016-01-18,127.42120166383975,126.04268863251424,128.9235642476408,132.12029493482515,132.12029493482515,132.12029493482515
2016-01-19,132.8977385744755,131.39034947653815,134.30039436122829,132.2272323524097,132.2272323524097,132.2272323524097
2016-01-20,133.0922006977247,131.56779709828456,134.6615535281576,132.33416976999425,132.33416976999425,132.33416976999425
2016-01-21,133.24521144076604,131.86375593826475,134.64573402397713,132.4411071875788,132.4411071875788,132.4411071875788
2016-01-22,133.38653747937147,132.05671870682798,134.83580143074215,132.54804460516337,132.54804460516337,132.54804460516337
2016-01-23,133.078194352028,131.6229254475383,134.55951661084882,132.65498202274793,132.65498202274793,132.65498202274793
```

Figura 4.15: Respuesta del servidor de ejecución predicción (salida en CSV).

Tal y como podemos ver en la figura 4.15, si activamos la opción de `to_csv`, la respuesta del servidor estará preparada para ser leída como un CSV, si no lo activamos la salida será en formato JSON (ver figura 4.16). Los campos de salida son:

- `ds`. Valor temporal de la muestra.
- `yhat`. Valor medio en ese instante temporal.
- `yhat_lower`. Valor mínimo en ese instante temporal.
- `yhat_upper`. Valor máximo en ese instante temporal.
- `trend`. Valor medio de la tendencia.
- `trend_lower`. Valor mínimo de la tendencia.
- `trend_upper`. Valor máximo de la tendencia.

Server response	
Code	Details
200	Response body <pre>{ "0": { "ds": "2016-01-02T00:00:00.000", "yhat": 130.0737169417, "yhat_lower": 128.5746625173, "yhat_upper": 131.6747859504, "trend": 130.4092962535, "trend_lower": 130.4092962535, "trend_upper": 130.4092962535 }, "1": { "ds": "2016-01-03T00:00:00.000", "yhat": 124.918349222, "yhat_lower": 123.5160311448, "yhat_upper": 126.36964403, "trend": 130.5162336711, "trend_lower": 130.5162336711, "trend_upper": 130.5162336711 }, "2": { "ds": "2016-01-04T00:00:00.000", "yhat": 125.4487431716, "yhat_lower": 123.9264583571, "yhat_upper": 126.9709985238 } }</pre>

Figura 4.16: Respuesta del servidor de ejecución predicción (salida en JSON).

Para comprobar la predicción realizada, procedemos a representar los datos que hemos recibido en la respuesta del servidor. Para ello, vemos las figuras 4.17 y 4.18, en las que podemos comprobar que el dataset ahora dura hasta principios de 2019, y que además ha podido predecir correctamente la tendencia de crecimiento exponencial anual y los efectos de tráfico añadidos sintéticamente (como por ejemplo, efecto fin de semana o el efecto verano).

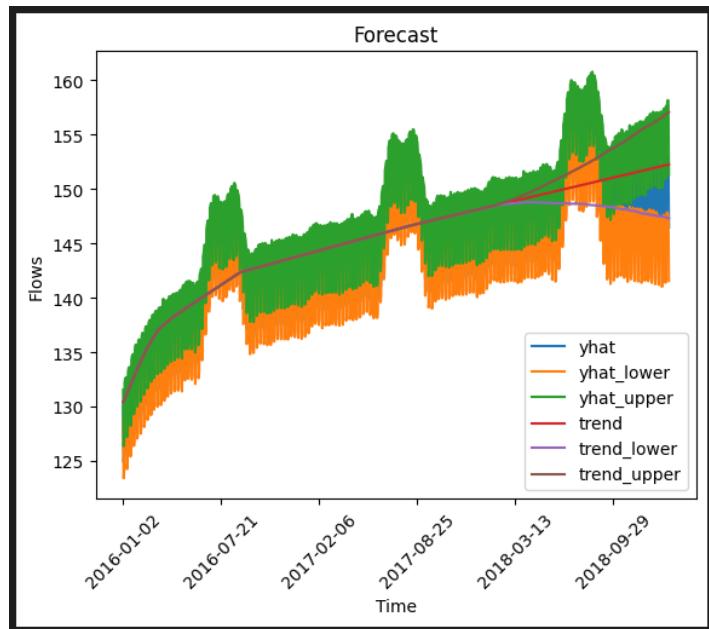


Figura 4.17: Gráfica datos de salida de la predicción de tráfico.

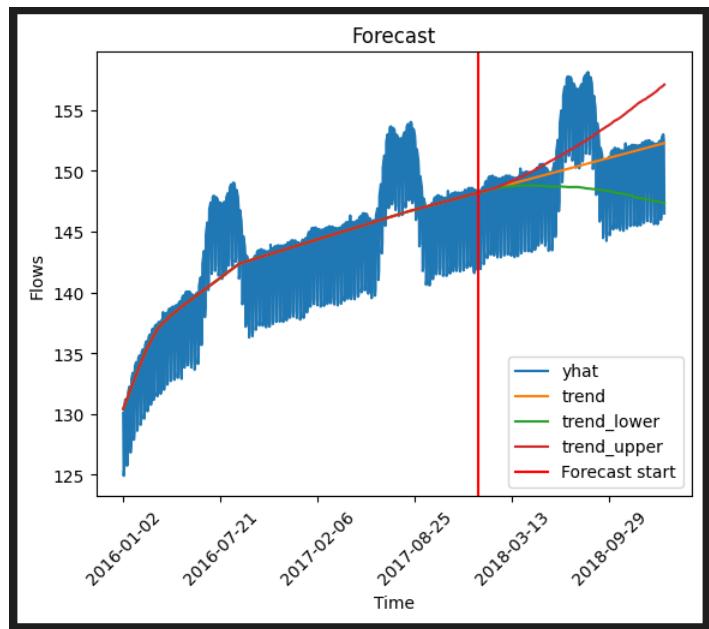


Figura 4.18: Gráfica datos de salida de la predicción de tráfico. A la izquierda de la linea roja, datos de muestras; a la derecha, datos predichos por el sistema.

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](#)
20. [Prophet Doc: Hyperparameter-tuning](#)
21. [Prophet Doc: Adjusting trend flexibility](#)

Figuras

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

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.

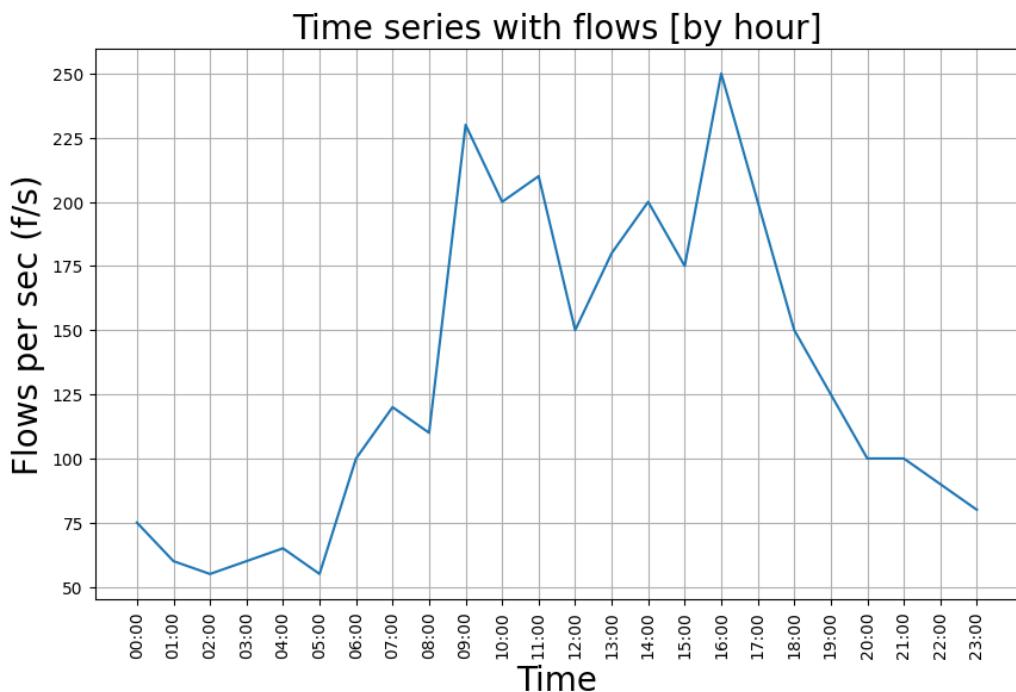


Figura 6.1: Simulación de tráfico de red. Un día, dividido por horas.

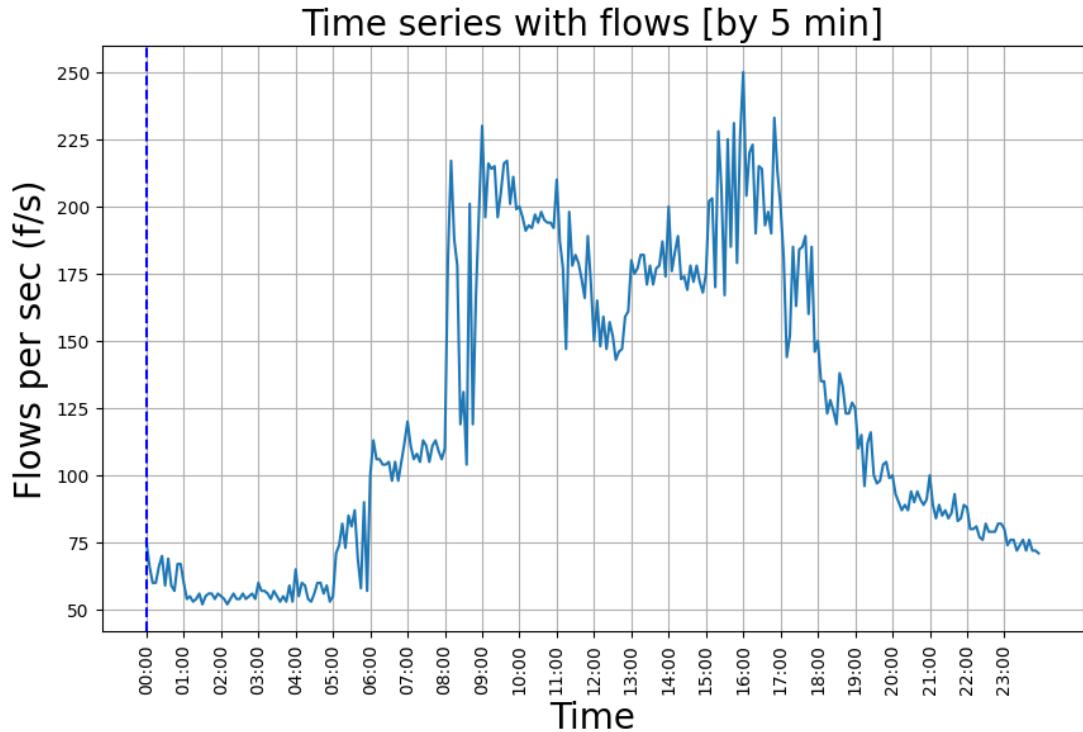


Figura 6.2: Simulación de tráfico de red. Un día, dividido por slots de 5 minutos.

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.

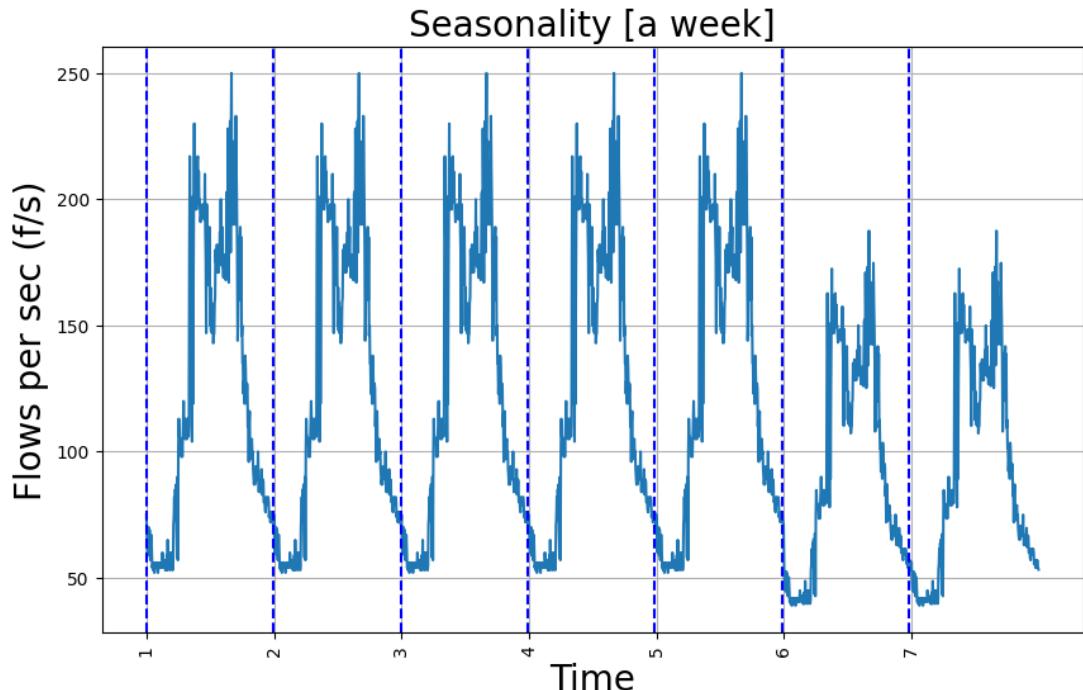


Figura 6.3: Simulación de tráfico de red. Una semana con slots de 5 minutos. Reducción de tráfico en el fin de semana

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.

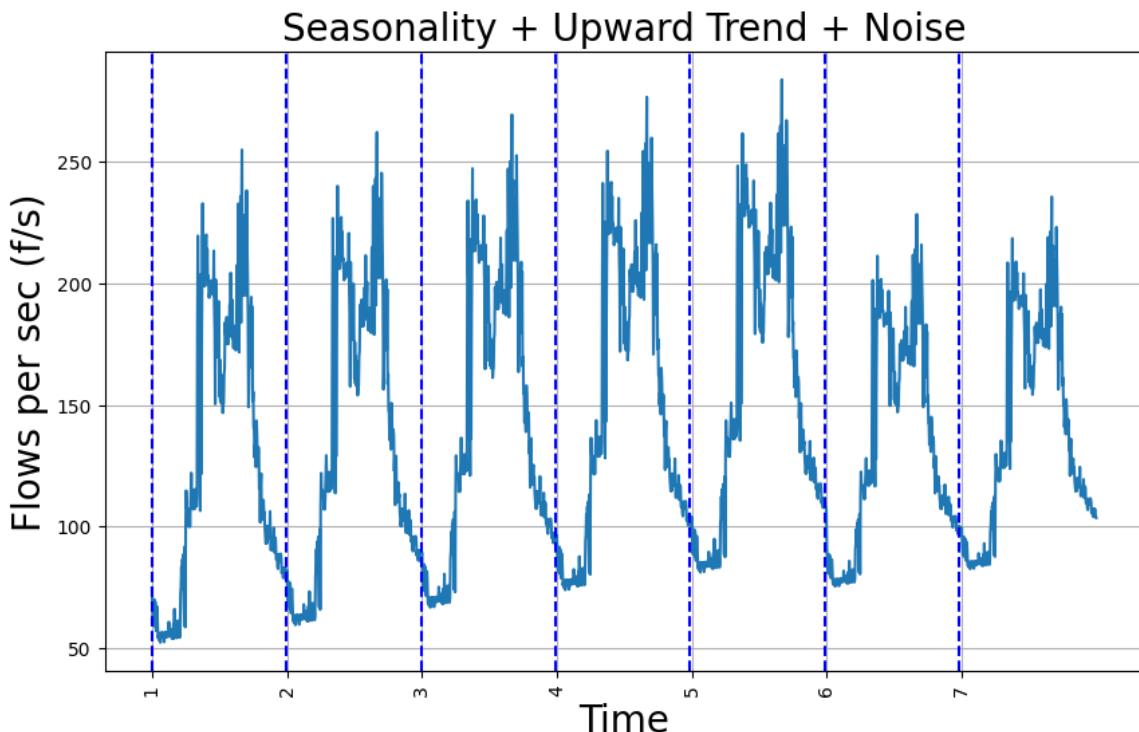


Figura 6.4: Simulación de tráfico de red. Una semana con slots de 5 minutos. Reducción de tráfico en fin de semana, tendencia exponencial y ruido blanco.

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).

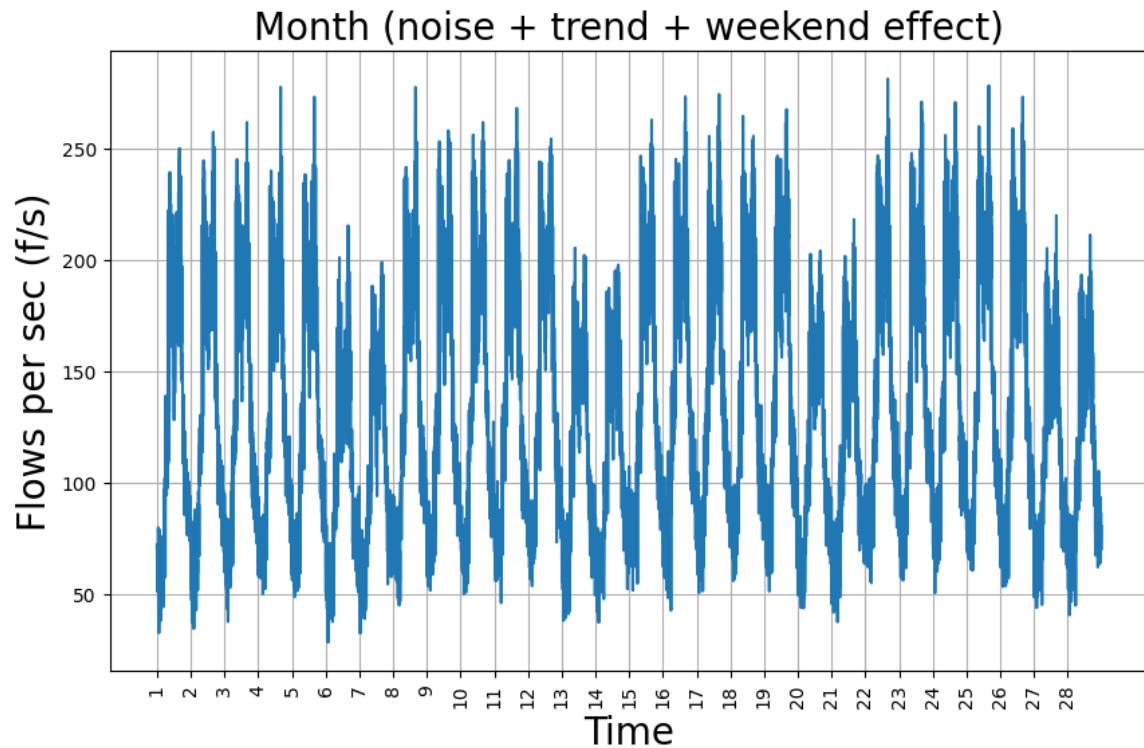


Figura 6.5: Simulación de tráfico de red. Un mes de datos, manteniendo reglas aplicadas.

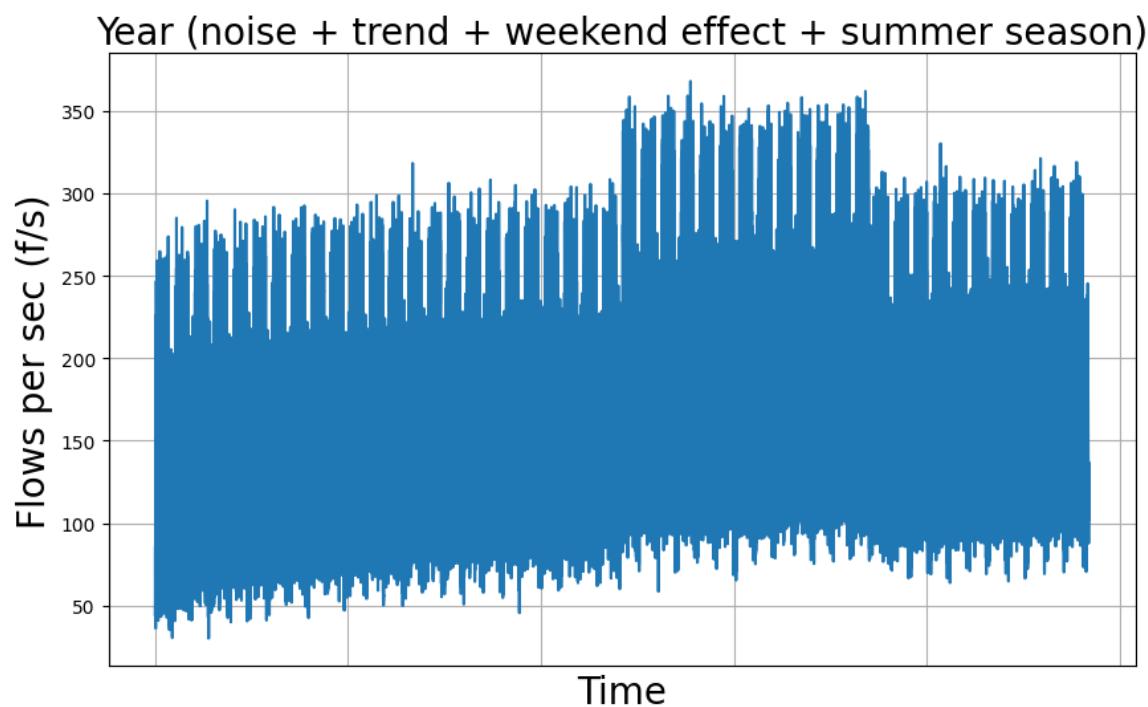


Figura 6.6: Simulación de tráfico de red. Un año de datos, se añade la regla del incremento de tráfico en verano.

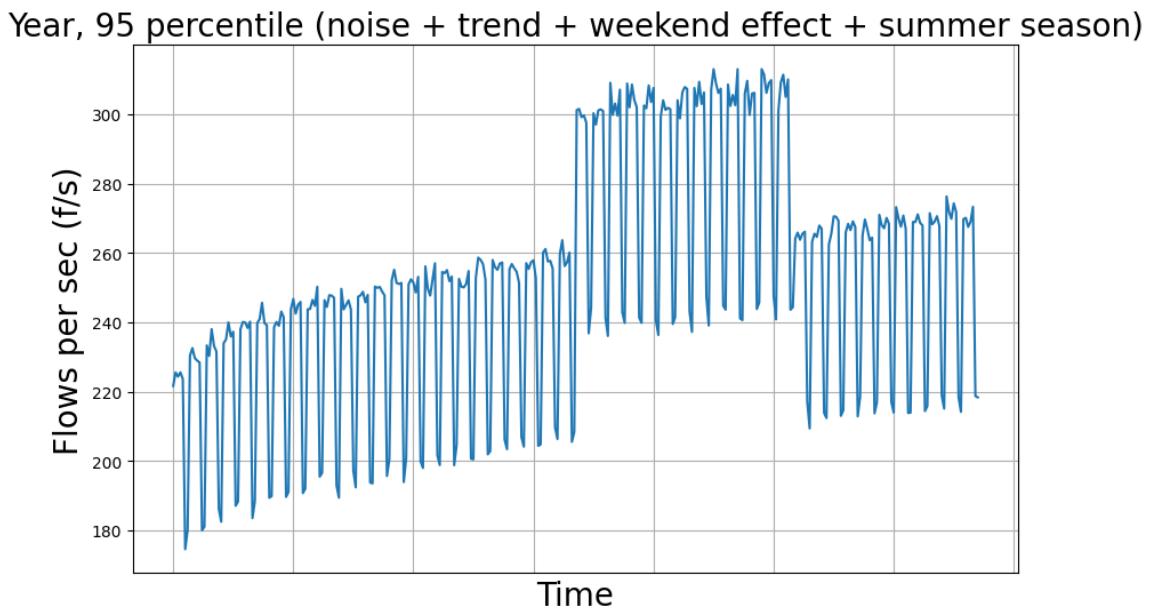


Figura 6.7: Simulación de tráfico de red. Un año de datos, se aplica el percentil 95 para cada día.

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.

Nota: las columnas del CSV generado son: `time` y `flow`