

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

*Autor:* Enrique Fernández Sánchez

*Tutor:* Pablo Pavón Mariño

Universidad Politécnica de Cartagena (UPCT)



9 de enero de 2023

1. Introducción
2. Tecnologías empleadas
3. Implementación del sistema
4. Validación del sistema
5. Conclusiones
6. Bibliografía

- *Abstract*: Aplicación que permite almacenar muestras de monitorización de tráfico en red, y a su vez, generar predicciones futuras del tráfico de red, en función de la información almacenada.

## Objetivos del proyecto

- Diseñar una **aplicación** siguiendo la metodología de **microservicios**.
- Investigar herramientas de **predicción de series temporales**.
- Investigar **opciones de almacenamiento** para **muestras temporales**.
- Utilizar **herramientas de documentación** que permitan conocer la estructura de la aplicación.

## Definición de microservicio

Sistemas que cumplen las siguientes premisas:

- Sistemas **pequeños, independientes** y poco “*acoplados*”.
- **Código fuente separado** entre los diferentes servicios, no necesariamente mismo lenguaje.
- Los servicios **se comunican** entre sí utilizando **APIs**.

## Importancia de utilizar microservicios

- Permite que otras **aplicaciones más complejas** puedan **implementar el servicio** propuesto.

- Una API permite a dos **componentes comunicarse** entre sí **mediante** una serie de **reglas**.
- Supone un “**contrato**” en el que **se establecen las solicitudes** y respuestas **esperadas en la comunicación**.

## Tipos de API

Dependiendo de la implementación, distinguimos entre cuatro tipos:

- **SOAP**. Protocolo tradicional, usa mensajes XML (HTTP solo transporte). Ambos interlocutores deben conocer la estructura de los objetos. Poco flexible.
- **RPC**. Basado en llamadas a procedimientos remotos.
- **WebSocket**. Solución moderna, usa objetos JSON y un canal bidireccional de comunicación.
- **REST**. Solución más popular y flexible. Usa los métodos HTTP. **Elegimos este tipo.**

- En **función** del **tipo de dato** a almacenar, se distinguen dos bases de datos dentro de la aplicación:

## Tipo relacional (PostgreSQL)

- Se almacenan **datos** que puedan tener una **relación** entre ellos. Por ejemplo: información de una red a monitorizar.
- Se organizan los datos en una serie de “*relaciones*”, y estas se almacenan en una o más “*tablas*”.
- Cada tabla dispone de una serie de columnas. La **información** se agrega en **forma de filas** a la tabla.

## Tipo serie temporal (InfluxDB)

- Necesario para almacenar **datos** en forma de **serie temporal** de manera eficiente.
- En comparación con el tipo relacional, la **fila de datos** está **identificada** por un **valor temporal**.
- Algunas ventajas:
  - Útil para almacenar datos de telemetría.
  - Datos comprimidos automáticamente.

## Python

**Lenguaje de programación orientado a objetos**, interpretado y de alto nivel. **Muy popular** en los siguientes ámbitos:

- Aplicaciones web.
- *Data Science*.
- Inteligencia Artificial.



## FastAPI

**Framework moderno** y rápido para construir **APIs**. Características:

- **Rápido**: rendimiento equivalente a otros lenguajes (NodeJS o Go).
- **Intuitivo**: soporta autocompletado.
- **Robusto**: herramienta Swagger/ReDoc automática.



## Prophet

Librería del lenguaje de programación Python, desarrollada por Meta (Facebook). Agrupa una serie de **procedimientos** que permiten **realizar predicciones** en un **dataset temporal**. Características:

- Permite encontrar y tener en cuenta **efectos no lineales** (tendencias diarias, semanales, mensuales...).
- Permite predecir datos en **días vacacionales**.
- Basado en inferencia estadística, **más eficiente** que si utilizáramos técnicas de *Machine Learning*.



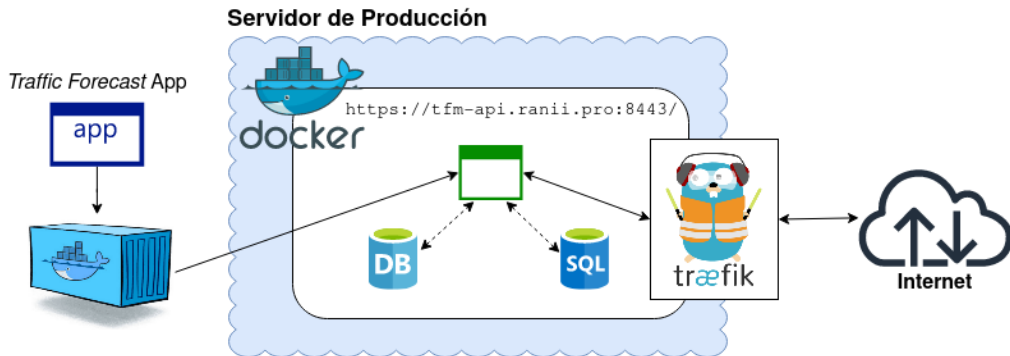
PROPHET



# Despliegue en producción

Herramientas utilizadas para el despliegue del sistema en un entorno de producción:

- Docker. Contenedores.
- docker-compose. Despliegue automatizado de contenedores Docker.
- Traefik. Enrutador que permite conectar dominios con servicios Docker.



Para la aplicación se implementan los diferentes agentes utilizando **dos metodologías**:

## 1. CRUD

- Aquellas que siguen la definición Create, Read, Update & Delete.
- *Ejemplo*: redes o interfaces.

## 2. No CRUD

- Aquellas que no tienen por qué seguir las reglas CRUD.
- *Ejemplo*: ejecutar predicción de red.

## Agentes

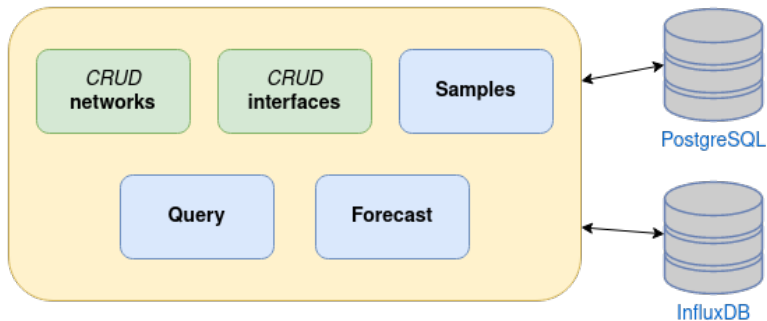
Se identifican los diferentes “*agentes*” presentes en el sistema:

- **Redes** (*networks*), corresponde con una red que contiene interfaces a monitorizar.
- **Interfaces** (*interfaces*), corresponde con las interfaces de red que queremos monitorizar.
- **Muestras de monitorización** (*samples*). Valor de tráfico asociado a una interfaz en un intervalo determinado.

# Descripción API REST (II)

## Esquema de la aplicación implementada

Traffic Forecast Microservice



## Factory Pattern

Patrón de diseño que **permite estructurar** un **proyecto** software de modo que permita:

- Basado en la **creación de objetos** sin especificar su clase.
- Permite **añadir funcionalidades** de manera sencilla. Ordena los archivos del proyecto dependiendo de la funcionalidad.
- *Ventaja*: crecimiento ordenado y fácil de la aplicación.

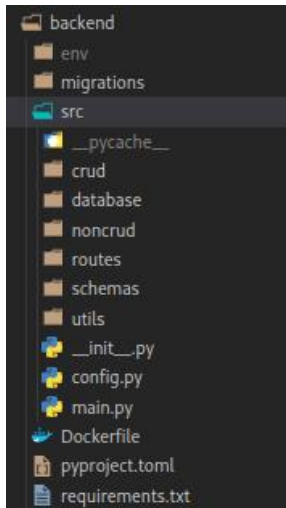
## ORM & migrations

- **ORM** (*Object-relational Mappers*). Abstracción de alto nivel que permite **definir modelos** de datos SQL con el lenguaje **Python**.
- **migrations**. Permite tener un **control** de **versiones** dentro de los **modelos** de datos de la aplicación. **Facilita** el **despliegue** de la base de datos, en lo que a estructura de tablas se refiere.

# Implementación (II)

## Estructura de ficheros del proyecto

- `migrations`. Contiene las versiones de la DB.
- `src`
  - `crud`. Funcionalidad de agentes CRUD.
  - `database`. Contiene modelos de datos SQL.
  - `noncrud` Funcionalidad agentes no CRUD.
  - `routes`. Rutas de la aplicación.
  - `schemas`. Esquema de datos.
  - `utils`. Utilidades, wrapper InfluxDB.
  - `config.py`. Configuración app.
  - `main.py`. Archivo principal.
- `Dockerfile`. Instrucciones para contenedor Docker.
- `requirements.txt`. Módulos Python utilizados.



- FastAPI **genera de manera automática** una serie de rutas que sirven como **documentación del proyecto**.
- En estas herramientas se muestran al usuario la información de:
  - Rutas de la aplicación.
  - Parámetros de entrada y de salida para una petición.
  - Modelos de datos para cada una de las rutas.
  - Posibles códigos HTTP de respuesta para las peticiones.
- **Permite** al usuario **realizar peticiones al servidor** y ver su resultado, desde la misma herramienta.

## Acceso a las herramientas de documentación

- **Swagger:** <https://tfm-api.ranii.pro:8443/docs>
- **ReDoc:** <https://tfm-api.ranii.pro:8443/redoc>

- Definimos los datos y su **representación dentro** de nuestra **aplicación**.
- **En función del tipo de dato**, almacenaremos en **base de datos tipo relacional** (SQL) o **base de datos tipo serie temporal** (InfluxDB).

## Tablas utilizadas en la base de datos SQL

Networks
id_network: Int, Public Key, Unique
name: String
description: String
ip_red: String
influx_net: String

Interfaces
id_interface: Int, Public Key, Unique
name: String
description: String
influx_if_rx: String
influx_if_tx: String
network: Int, Foreign Key

- En InfluxDB **almacenamos las muestras de tráfico** en red.

## Configuración InfluxDB para el sistema

InfluxDB	Descripción
measurement	Red a monitorizar, valor almacenado en <code>Networks::influx_net</code>
fields	Nombre del valor a monitorizar ( <i>link_count</i> )
tags	Solo disponemos un tag, llamado <code>interface</code> , contiene la información del identificador de una interfaz
points	Corresponde con el valor numérico del campo <code>field</code> .



- El sistema es **capaz** de **ejecutar predicciones** de tráfico de red, en función de las **muestras de monitorización** previamente **almacenadas en el sistema**.

## Pasos para realizar una predicción de tráfico

1. **Creamos una red** a monitorizar. POST a /networks.
2. **Creamos una interfaz** a monitorizar. POST a /networks/<net>/interfaces.
3. **Importamos datos** de monitorización. Dos maneras:
  - POST a ruta: /samples/<net>/import\_topology.
  - POST a ruta: /samples/<net>/import\_interface/<if>.
4. **Ejecutamos predicción de tráfico** en red. Configuramos la predicción y lanzamos la petición POST a /forecast.

Request body **required**

Example Value | Schema

```
{
  "id_network": 0,
  "id_interface": 0,
  "field": "RX",
  "days": 365,
  "options": {
    "holidays_region": "ES",
    "flexibility_trend": 0.05,
    "flexibility_season": 10,
    "flexibility_holidays": 10
  }
}
```

## Parámetros para la ejecución de la predicción de tráfico

- `field`. Campo para elegir si RX o TX.
- `days`. Número de días a predecir.
- `options`. Opciones que modifican la flexibilidad de la predicción:
  - `holidays_region`
  - `flexibility_trend`
  - `flexibility_season`
  - `flexibility_holidays`

## CRUD: networks (/networks)

- **Información de todas** las redes:  
*GET* - /networks
- **Crear** una red:  
*POST* - /networks
- **Información de una** red:  
*GET* - /networks/<net\_id>
- **Eliminar** una red:  
*DELETE* - /networks/<net\_id>
- **Actualizar** una red:  
*PATCH* - /networks/<net\_id>

## CRUD: interfaces (/networks/<id1>/interfaces)

- **Información de todas** las interfaces:  
*GET* - /networks/id/interfaces
- **Crear** una interfaz:  
*POST* - /networks/id/interfaces
- **Información de una** interfaz:  
*GET* - ../interfaces/<id2>
- **Eliminar** una interfaz:  
*DELETE* - ../interfaces/<id2>
- **Actualizar** una interfaz:  
*PATCH* - ../interfaces/<id2>

## Samples

- **Importar** datos de topología con **más de una interfaz**:  
*POST* -  
`/samples/id/import_topology`
- **Importar** datos de **una interfaz**:  
*POST* -  
`/samples/id/import_interface/id`

## Query Samples

- **Consultar datos** de monitorización almacenados:  
*GET* - `/query/`

## Forecast

- **Ejecutar predicción** de tráfico en red:  
*POST* - `/forecast/`

Para demostrar el funcionamiento del sistema, se realiza una **batería de pruebas** de las funcionalidades implementadas. Dichas pruebas son:

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.
6. **Ejecutar una predicción** de un año.

Estas pruebas corresponden con las **pruebas unitarias** del sistema.

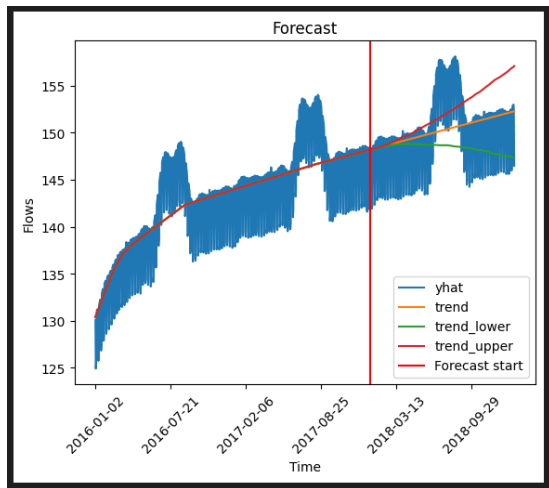
# Validación del sistema (II)

## Demo! (2 minutos)

Realizamos una validación del sistema.

<https://tfm-api.ranii.pro:8443/docs>

- *Objetivo*: Completar una predicción de un año.
- *Requisito*: Asumir sistema sin datos almacenados.
- *Resultado*: Una predicción similar a la de la figura.



- Alcanzados todos los objetivos propuestos.
- Se ha desarrollado un **microservicio completo**, permitiendo ser implementado por otras aplicaciones.
- El sistema es **capaz** de **almacenar** muestras de monitorización, además de poder **generar predicciones** de tráfico en red en la escala temporal que el usuario solicite.

## Propuestas futuras

- Permitir la **importación** de **datos de monitorización** de **herramientas específicas** de planificación de red.
- Añadir **funcionalidad de SSE** (*Server Side Event*) para **tareas** que requieran un **largo tiempo de ejecución**.
- Extender la funcionalidad de las **predicciones**, permitiendo tener una selección **más selectiva**, o añadir **filtrados extra**.

- La contenida en la memoria del proyecto: páginas 53 - 54



**Muchas gracias por su atención**

¿Preguntas?

Enlace a la aplicación:

`https://tfm-api.ranii.pro:8443/`