TECNICATURA SUPERIOR EN

Telecomunicaciones

Proyecto Integrador I

Optimización I: Infraestructura

ISPC
INSTITUTO SUPERIOR POLITÉCNICO CÓRDOBA

Dirección General de EDUCACIÓN TÉCNICA Y FORMACIÓN PROFESIONAL
Ministerio de EDUCACIÓN
CÓRDOBA entre todos

Optimización I: Infraestructura

Optimización I: Infraestructura

1. Introducción

- **1.1 Procesamiento en Cloud**
  - Infraestructura del servidor (hardware, virtualización, software, etc.).
  - Componente Software del Cloud (microservicios, APIs, middleware).
  - Aplicaciones del procesamiento en Cloud (visualización, generación de informes, toma de decisiones automatizadas).

- **1.2 Presentacion del proyecto de la Unidad**
  - Este proyecto busca implementar prácticas de sensorización en una arquitectura Edge/Fog utilizando ESP32 para la adquisición de datos y herramientas como Grafana para su visualización.

Optimización I: Infraestructura

- **1.3 Objetivos de las prácticas**
  - Implementar un sistema de sensorización utilizando la arquitectura Edge/Fog.Configurar el stack tecnológico para visualizar los datos en Grafana.
  - Explorar la eficiencia de protocolos de comunicación en la infraestructura IoT.
  - Optimizar el uso de servidores y herramientas para la gestión y análisis de datos en tiempo real.

2. Stack Tecnológico

- **2.1 Protocolos de Comunicación en Edge**
  - Wi-Fi: Configuración y aplicaciones para IoT.
  - Bluetooth Low Energy (BLE): Casos de uso y ventajas en IoT.
  - ESP-NOW: Comunicación directa entre dispositivos sin enrutador.
  - ESP-Mesh: Creación de redes Mesh con ESP32.
  - Matter (opcional): Interoperabilidad en IoT, potencial uso.

Optimización I: Infraestructura

- **2.2 Protocolos de Comunicación en Fog**
  - HTTP: Comunicación estándar entre Edge y Fog.
  - MQTT: Protocolo ligero para IoT, eficiencia en la transmisión de datos.
  - WebSocket: Comunicación bidireccional en tiempo real, casos de uso.

- **2.3 Infraestructura de Visualización**
  - Node-Red: Orquestación de dispositivos y flujos de trabajo IoT.
  - Grafana: Herramienta de visualización de datos IoT.
  - InfluxDB: Base de datos para series temporales, integración con Grafana.
  - Flask: Framework para construir una API RESTful y backend del sistema.
  - Nginx: Servidor web y balanceador de carga para mejorar la eficiencia.

Optimización I: Infraestructura

3. Prácticas de Implementación

- **3.1. Implementación 1: Sensorización y Visualización en Grafana**
  - Configuración del servidor con Flask.
  - Almacenamiento de datos en InfluxDB.
  - Visualización de los datos de los sensores en Grafana.

- **3.2. Implementación 2: Orquestación con Node-Red**
  - Implementación de flujos de trabajo con Node-Red.
  - Sensorización y visualización en tiempo real.
  - Comparación con la visualización en Grafana.

Optimización I: Infraestructura

- **3.3. Implementación 3: Integración con MySQL**
  - Configuración de MySQL como base de datos principal.
  - Uso de Nginx y Flask para manejo del servidor.
  - Sensorización con almacenamiento y representación de datos.

- **3.4. Implementación 4: Sensorización Completa con Nginx, Flask y Grafana**
  - Integración total del stack con Nginx, Flask, InfluxDB y Grafana.
  - Visualización y monitoreo en tiempo real.
  - Optimización del sistema de sensorización.

Optimización I: Infraestructura

4. Conclusiones

- Principales aprendizajes sobre la infraestructura Edge/Fog.
- Ventajas y desventajas de los diferentes protocolos y tecnologías.
- Recomendaciones para mejorar la infraestructura en proyectos futuros.

Optimización I: Infraestructura

Componentes y Aplicaciones

Optimización I: Infraestructura

Infraestructura del Servidor
La infraestructura en la nube proporciona la capacidad de escalabilidad y flexibilidad necesarias para manejar grandes volúmenes de datos. Algunos componentes son:

- **Hardware**: Puede variar entre servidores dedicados, sistemas distribuidos o soluciones en la nube como AWS, Azure o Google Cloud.
- **Virtualización**: El uso de máquinas virtuales permite la creación de entornos separados que optimizan el uso de los recursos físicos.
- **Contenedores**: Tecnologías como **Docker** o **Kubernetes** permiten un mejor control de la infraestructura y una implementación más ágil de servicios.
- **Software**: Desde sistemas operativos específicos (Linux) hasta entornos de desarrollo como **Flask** y **Nginx**, que gestionan la lógica de negocio y la interfaz de comunicación entre dispositivos Edge/Fog.

Optimización I: Infraestructura

Componentes Software del Cloud

- **Microservicios**: Arquitectura basada en servicios independientes que se comunican a través de APIs. Facilitan la escalabilidad y el mantenimiento de aplicaciones IoT.

- **APIs**: Las **APIs RESTful** permiten la interacción entre los diferentes componentes de la infraestructura (dispositivos, bases de datos, visualización).

- **Middleware**: Actúa como puente entre los servicios, asegurando la correcta transmisión de datos y la integración de sistemas. Incluye herramientas como **MQTT**, **RabbitMQ**, o **Apache Kafka** para mensajería.

Optimización I: Infraestructura

Aplicaciones del Cloud

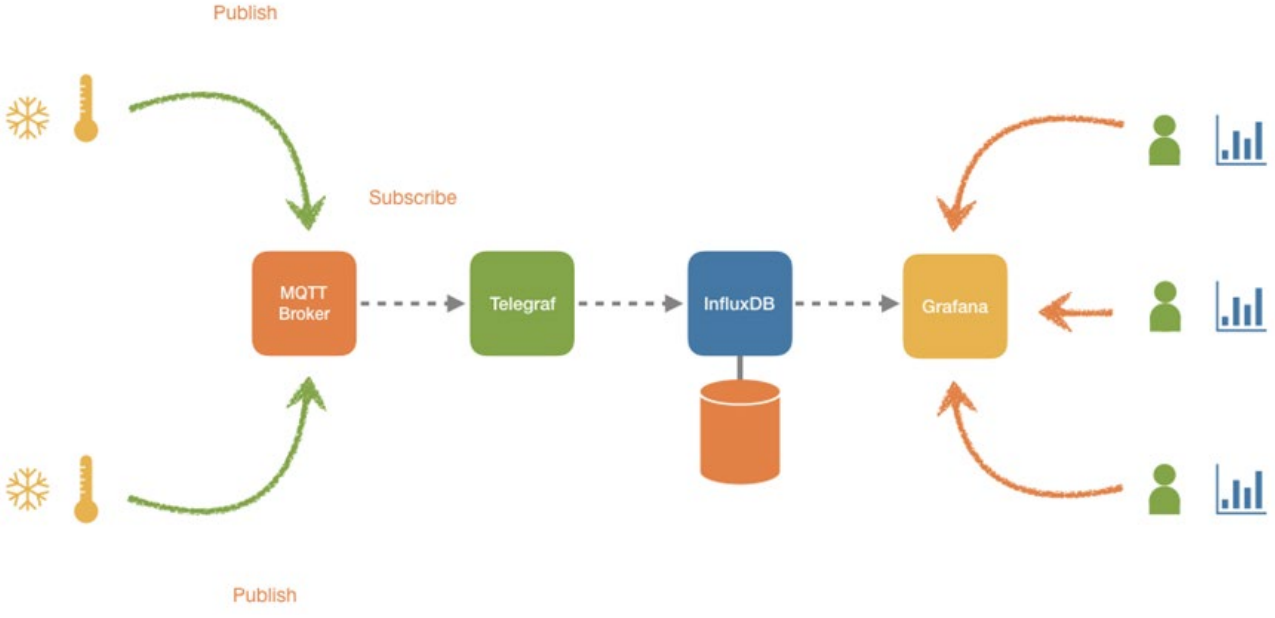
- **Visualización**: Herramientas como **Grafana** permiten crear dashboards interactivos para monitorear los datos recolectados desde la infraestructura Edge/Fog en tiempo real.

- **Generación de informes**: A partir de los datos recolectados y procesados en la nube, se pueden generar informes automáticos que faciliten la toma de decisiones.

- **Toma de decisiones automatizadas**: Mediante el uso de **Machine Learning** o reglas predefinidas, el sistema puede realizar ajustes automáticos en la infraestructura (ej. activar alertas, optimizar recursos).
- El procesamiento en la nube facilita una integración más eficiente con sistemas de inteligencia artificial y análisis predictivo.

Optimización I: Infraestructura

Proyectos de ejemplos



Optimización I: Infraestructura

Descripción general del proyecto IoT

Arquitectura Edge/Fog y Visualización de Datos

- El proyecto se enfoca en implementar una arquitectura **Edge/Fog** utilizando dispositivos **ESP32** para sensorización.
- Se capturan datos de humedad del suelo, temperatura y luz utilizando sensores como el **DHT11** y el **LDR**.
- Los datos serán enviados a la infraestructura Fog para su procesamiento inicial y luego almacenados en bases de datos locales como **InfluxDB**.
- **Grafana** se utiliza para la visualización de los datos en tiempo real.
- El servidor local (Fog) usará **Flask** y **Nginx** para gestionar la recepción y exposición de los datos.
- Este enfoque permite reducir la latencia y mejorar la toma de decisiones en el procesamiento de los datos IoT.

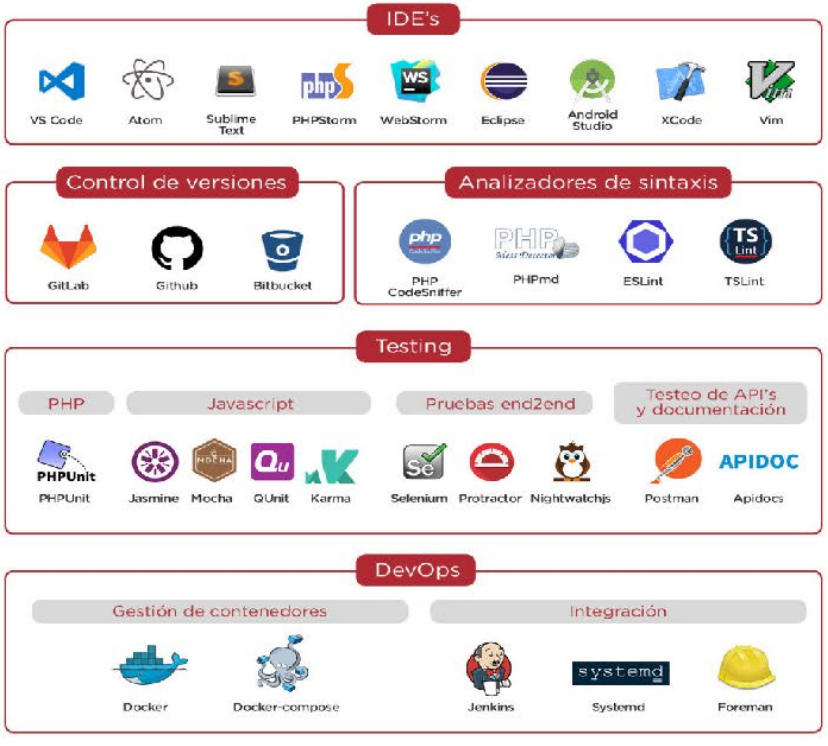
Optimización I: Infraestructura

Objetivos

- **Implementar un sistema de sensorización** que recoja datos del entorno utilizando dispositivos **ESP32** en la arquitectura Edge/Fog.
- **Configurar el stack tecnológico** que incluye **Grafana** para la visualización e **InfluxDB** como soporte para el almacenamiento.
- **Evaluar los protocolos de comunicación** en Edge y Fog, como **MQTT**, **HTTP**, y **Protocolos Mesh**, utilizados en diferentes escenarios.
- **Optimizar el uso de servidores locales** mediante la integración de tecnologías como **Flask**, **Nginx**, y **Node-Red** para la gestión y visualización de los datos en tiempo real.

Optimización I: Infraestructura

Optimización I: Infraestructura



Optimización I: Infraestructura

El stack Tecnológico del Proyecto: “Los protocolos”

- En una arquitectura IoT, los **protocolos de comunicación** desempeñan un papel fundamental al facilitar la interacción entre los dispositivos Edge (sensores y actuadores) y los sistemas Fog/Cloud, donde se procesan y analizan los datos.
- La selección adecuada de estos protocolos depende de factores como la **latencia**, el **consumo energético**, el **alcance** y la **capacidad de transmisión de datos**.

Optimización I: Infraestructura

El ESP32 y los Protocolos del Edge

Optimización I: Infraestructura

- **Protocolos en Edge**

- En el Edge, donde se encuentran los dispositivos que capturan los datos del entorno, se utilizan diferentes tecnologías de comunicación para transmitir esa información hacia el Fog. Para el ESP32-Wroom, el controlador que utilizaremos, estas tecnologías incluyen **Wi-Fi**, **Bluetooth Low Energy (BLE)**, **ESP-NOW**, y **ESP-Mesh**, cada una con características específicas que las hacen más o menos adecuadas según el caso de uso. La decisión sobre qué protocolo utilizar depende de las necesidades de **alcance**, **autonomía energética** y **velocidad de transmisión de datos**.

Optimización I: Infraestructura

Wi-Fi, Configuración y Aplicaciones para IoT

- **Wi-Fi** es uno de los protocolos más utilizados en IoT gracias a su alta velocidad de transmisión (hasta 150 Mbps) y amplio alcance (100-200 metros).
- En el ESP32-WROOM, se configura fácilmente mediante el uso de librerías disponibles en el framework Arduino y PlatformIO.

- **Aplicaciones comunes**:
  - **Monitorización remota**: Transmisión de datos en tiempo real desde sensores.
  - **Automatización del hogar**: Control de dispositivos inteligentes.
  - **Enrutamiento de datos**: Envío de datos a plataformas en la nube como AWS o Azure para procesamiento adicional.

- **Ventajas**: Amplia compatibilidad y acceso directo a la nube.
- **Desventajas**: Consumo elevado de energía, no ideal para dispositivos que dependen de baterías.

Optimización I: Infraestructura

Bluetooth Low Energy (BLE), Casos de Uso y Ventajas en IoT

- **Bluetooth Low Energy (BLE)** es un protocolo diseñado para minimizar el consumo energético, ideal para dispositivos IoT que requieren larga duración de la batería.
- En el ESP32-WROOM, BLE permite la conexión de corto alcance (10-100 metros), optimizando el uso de energía.

- **Casos de uso**:
  - **Sensores portátiles** (wearables): Monitorización de actividad física o salud.
  - **Beacons**: Localización y seguimiento en interiores.
  - **Automatización industrial**: Comunicación entre dispositivos cercanos.

- **Ventajas**: Muy bajo consumo energético y suficiente para transmisión de datos periódica.
- **Desventajas**: Limitado en alcance y velocidad comparado con Wi-Fi.

Optimización I: Infraestructura

ESP-NOW: Comunicación Directa en Edge

- **ESP-NOW** es un protocolo de comunicación exclusivo del ESP32 que permite la transmisión de datos directamente entre dispositivos sin requerir un router o punto de acceso.

- **Características**:
  - Baja latencia y menor consumo de energía que Wi-Fi.
  - Ideal para aplicaciones IoT donde los dispositivos necesitan comunicarse de manera rápida y sin infraestructura de red.
- **Aplicaciones**:
  - **Redes de sensores** en áreas sin acceso a internet o donde no se desea desplegar infraestructura Wi-Fi.
  - **Sistemas de alerta**: Comunicación rápida entre dispositivos para sistemas de seguridad.

- **Ventajas**: No depende de un router y es eficiente en consumo.
- **Desventajas**: Limitado a dispositivos ESP y un alcance menor que Wi-Fi.

Optimización I: Infraestructura

ESP-Mesh: Redes Mesh con ESP32

- **ESP-Mesh** permite crear redes de malla con dispositivos ESP32, donde cada nodo puede conectarse a varios otros nodos, formando una red auto-reparable y escalable.

- **Aplicaciones**:
  - **Automatización del hogar**: Interconexión de múltiples dispositivos en una casa inteligente.
  - **Sistemas de monitoreo distribuidos**: Ideal para grandes áreas como fábricas o granjas.
  - **Redes resilientes**: Si un nodo falla, la red se reconfigura automáticamente.

- **Ventajas**: Escalabilidad y resiliencia; permite que los datos fluyan entre nodos.
- **Desventajas**: Complejidad en la configuración inicial y mayor demanda de recursos en comparación con ESP-NOW.

Optimización I: Infraestructura

Comparativa de Protocolos en el ESP32-WROOM

- **Wi-Fi**: Alta velocidad, pero alto consumo energético; ideal para transmisión de grandes volúmenes de datos.

- **BLE**: Muy bajo consumo de energía, pero limitado en alcance y velocidad.
- **ESP-NOW**: Comunicación rápida y directa sin necesidad de infraestructura de red; ideal para áreas sin acceso a internet.

- **ESP-Mesh**: Red de malla auto-organizada y escalable, adecuada para sistemas distribuidos en grandes áreas.
- **Matter:** Protocolo emergente para interoperabilidad en IoT.

Optimización I: Infraestructura

Protocolo	Consumo Energético	Alcance	Velocidad	Ventajas	Desventajas
Wi-Fi	Alto	100-200 m	Alta	Conexión directa a la nube	Alto consumo
BLE	Muy bajo	10-100 m	Baja	Muy bajo consumo energético	Alcance limitado
ESP-NOW	Bajo	100 m	Moderada	Baja latencia, sin infraestructura	Limitado a dispositivos ESP
ESP-Mesh	Moderado	Depende de la red	Moderada	Auto-reparable y escalable	Configuración compleja

Optimización I: Infraestructura

EL ESP32 y los Protocolos del Fog

Optimización I: Infraestructura

- **Protocolos en Fog**

- Por otro lado, en el Fog Computing, la comunicación con los dispositivos Edge se realiza mediante protocolos como **HTTP**, **MQTT**, y **WebSocket**. Cada uno de estos protocolos ofrece ventajas según la **latencia**, **eficiencia en el uso de ancho de banda**, y **modelo de comunicación** (unidireccional o bidireccional).

- En las siguientes diapositivas veremos los **protocolos de comunicación en el Fog,** utilizados en el proyecto IoT, comparando sus características y casos de uso. También presentaremos una **tabla comparativa** para evaluar qué protocolo es el más adecuado según los requisitos de cada práctica.

Optimización I: Infraestructura

HTTP: Comunicación Estándar en Fog Computing

- **HTTP (Hypertext Transfer Protocol)** es un protocolo de comunicación ampliamente utilizado para la transmisión de datos entre nodos Edge y servidores Fog.

- **Características**:
  - Facilidad de uso mediante solicitudes **GET** y **POST**.
  - Compatibilidad con casi todos los dispositivos IoT y servidores web.
- **Aplicaciones**:
  - **Monitorización remota**: Los sensores IoT pueden enviar datos a un servidor Fog usando HTTP.
  - **Automatización**: Los servidores Fog pueden enviar comandos a los nodos Edge usando peticiones HTTP.

- **Ventajas**: Sencillo y universal.
- **Desventajas**: Mayor latencia y consumo de ancho de banda comparado con otros protocolos como MQTT.

Optimización I: Infraestructura

MQTT: Protocolo Ligero para IoT

- **MQTT (Message Queuing Telemetry Transport)** es un protocolo ligero diseñado para IoT, basado en un modelo de **publicación / suscripción.**

- **Características**:
  - Muy eficiente en redes con poco ancho de banda.
  - Garantiza la entrega de mensajes con diferentes niveles de calidad de servicio (QoS).
- **Aplicaciones**:
  - **Monitoreo de sensores**: Los sensores publican datos a temas específicos y los servidores Fog se suscriben para recibir esa información.
  - **Sistemas distribuidos**: Ideal para sistemas IoT con múltiples dispositivos en diferentes ubicaciones.

- **Ventajas**: Muy bajo consumo de ancho de banda y confiable.
- **Desventajas**: Requiere un **broker** para gestionar las conexiones, como Mosquitto.

Optimización I: Infraestructura

WebSocket: Comunicación Bidireccional en Tiempo Real

- **WebSocket** es un protocolo que permite la comunicación bidireccional y en tiempo real entre dispositivos IoT y servidores Fog.
- A diferencia de HTTP, WebSocket mantiene una conexión persistente, eliminando la necesidad de reabrir la conexión cada vez que se intercambian datos.
- **Características**:
  - Comunicación en **tiempo real**: Ideal para aplicaciones donde se necesita monitoreo continuo y de baja latencia.
  - **Baja latencia**: Ya que no se cierra la conexión, el intercambio de mensajes es mucho más rápido que en HTTP.
  - **Compatibilidad**: Funciona en muchos entornos web y en dispositivos IoT que requieren flujos de datos continuos.
- **Aplicaciones**:
  - **Monitoreo en tiempo real**: Dispositivos IoT que envían datos de forma continua a servidores Fog, por ejemplo, en fábricas inteligentes o domótica.
  - **Sistemas interactivos**: Permite la interacción continua entre nodos Edge y servidores Fog para un control fluido y en tiempo real.
- **Ventajas**: Alta eficiencia en la transmisión continua de datos y baja latencia.
- **Desventajas**: Mayor complejidad de implementación en comparación con HTTP o MQTT.

Optimización I: Infraestructura

Comparativa de Protocolos de Comunicación en Fog

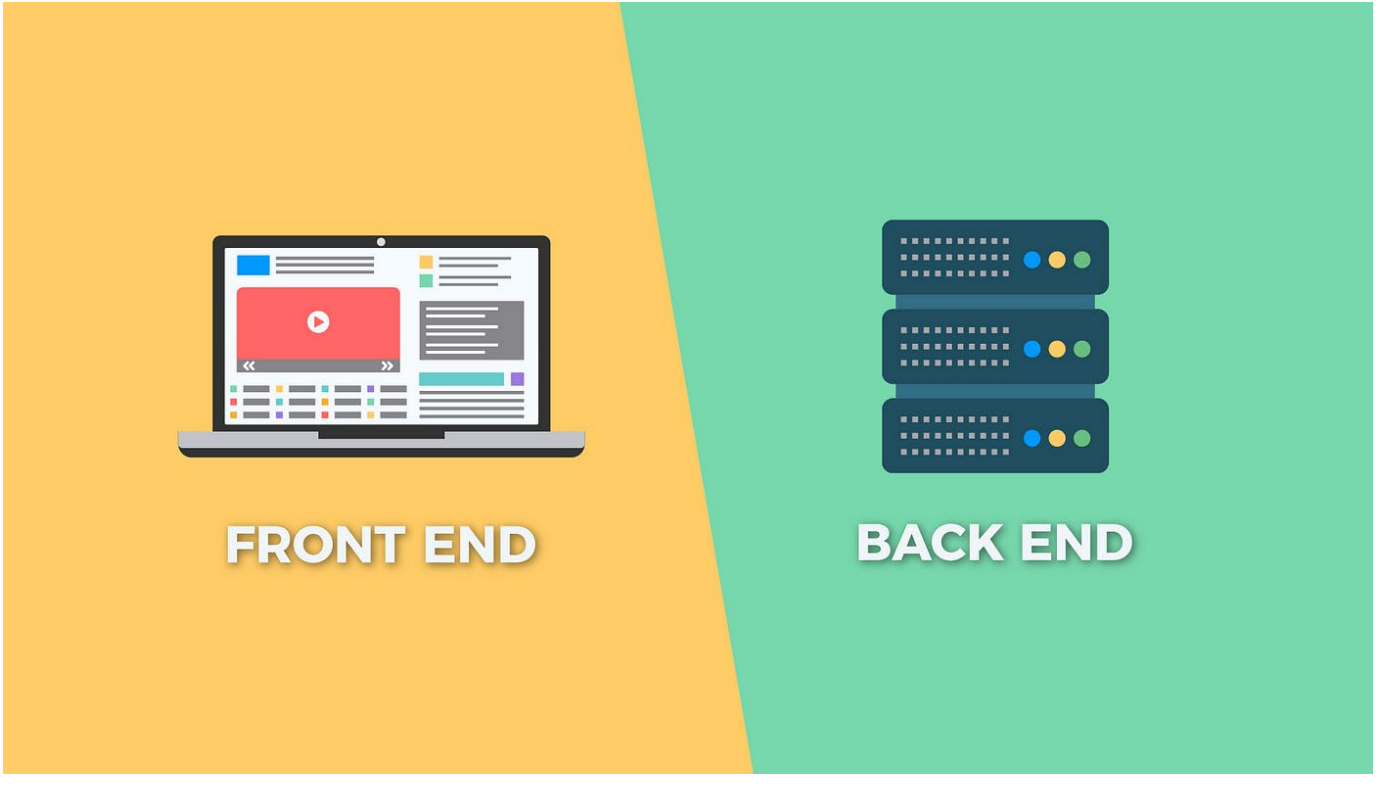
- **HTTP**: Protocolo de solicitud-respuesta estándar, fácil de implementar, pero con mayor consumo de ancho de banda y mayor latencia.
- **MQTT**: Protocolo ligero de publicación / suscripción, eficiente en redes con baja capacidad y con alta fiabilidad.
- **WebSocket**: Protocolo bidireccional y en tiempo real, ideal para aplicaciones que requieren comunicaciones de baja latencia.

Optimización I: Infraestructura

Protocolo	Eficiencia de Consumo de Ancho de Banda	Latencia	Modelo de Comunicación	Ventajas	Desventajas	Aplicaciones IoT
HTTP	Moderada	Alta	Request/Response	Amplia compatibilidad, fácil de usar	Mayor consumo de ancho de banda, latencia alta	Monitorización remota, automatización
MQTT	Alta (muy eficiente)	Baja	Publish/Subscribe	Ligero, eficiente, escalable	Requiere un broker para gestionar conexiones	Monitorización de sensores, sistemas distribuidos
WebSocket	Moderada	Muy baja	Bidireccional	Comunicación en tiempo real, baja latencia	Complejidad de implementación	Control en tiempo real, sistemas interactivos

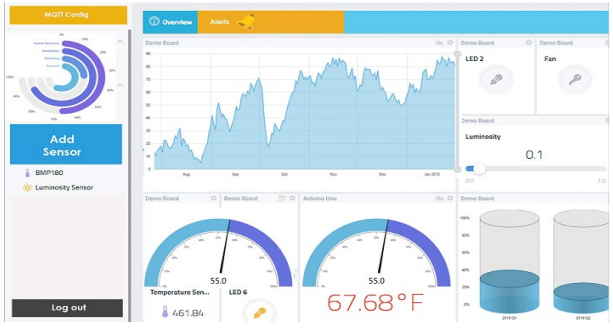
Optimización I: Infraestructura

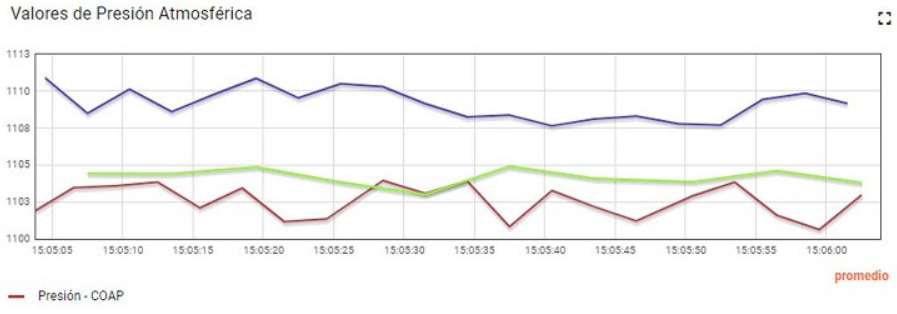
Optimización I: Infraestructura



Optimización I: Infraestructura

Introducción a la Infraestructura de Visualización en IoT





- La **visualización** de los datos recolectados en un sistema IoT es crucial para el análisis, monitoreo y toma de decisiones. Para lograrlo, es fundamental contar con una infraestructura robusta que integre tanto herramientas de visualización (front-end), como el manejo del back-end (API y bases de datos), y el servidor que soporta esta arquitectura.

Optimización I: Infraestructura

Front-end: Herramientas de Visualización

- La presentación clara y eficiente de los datos recolectados en tiempo real es un aspecto clave para cualquier aplicación IoT. Herramientas como **Grafana** y **Node-Red** permiten a los usuarios visualizar y gestionar la información de los sensores de forma intuitiva, mediante gráficos interactivos y flujos de trabajo configurables. Estas herramientas no solo facilitan el monitoreo, sino que también permiten tomar decisiones basadas en datos de manera ágil y precisa.
- **Node-Red**: Es una plataforma que permite la **orquestación de flujos de trabajo IoT**, facilitando la interconexión entre dispositivos y la toma de decisiones en tiempo real mediante flujos visuales de datos.
- **Grafana**: Es una herramienta de **visualización de datos** especialmente potente para IoT, que permite crear dashboards interactivos para monitorear datos históricos y en tiempo real desde bases de datos como **InfluxDB**.

Optimización I: Infraestructura

Back-end: Almacenamiento y Modelo de Negocio

- El **back-end** en un sistema IoT gestiona la comunicación entre el front-end y los dispositivos Edge. Aquí, entra en juego **Flask**, que permite crear una **API RESTful** que interconecta los dispositivos, gestiona la comunicación con la base de datos, y sirve como puente entre el front-end y los datos en el sistema. **InfluxDB** se utiliza como una base de datos de series temporales, diseñada específicamente para almacenar datos IoT que dependen del tiempo, lo que permite consultas rápidas para la visualización en Grafana.
- **InfluxDB**: Esta base de datos está optimizada para el almacenamiento de datos temporales, lo que la hace perfecta para el monitoreo constante de dispositivos IoT.
- **Flask**: Es un framework ligero para construir **APIs RESTful**, que permiten la comunicación eficiente entre los dispositivos IoT, el almacenamiento de datos y la visualización en tiempo real.

Optimización I: Infraestructura

- **Servidor Software: Nginx**

- El servidor que maneja la infraestructura IoT debe ser capaz de gestionar múltiples solicitudes de manera eficiente, servir contenido web y balancear la carga del sistema. **Nginx** juega un papel importante al actuar como **servidor web** y **balanceador de carga**, asegurando que el flujo de datos entre los dispositivos, el almacenamiento y la visualización sea continuo y rápido, sin sobrecargar el sistema.

- **Nginx**: Se encarga de gestionar el tráfico de solicitudes HTTP, sirviendo la API creada con Flask y asegurando que las peticiones a la base de datos y la visualización sean rápidas y estables.

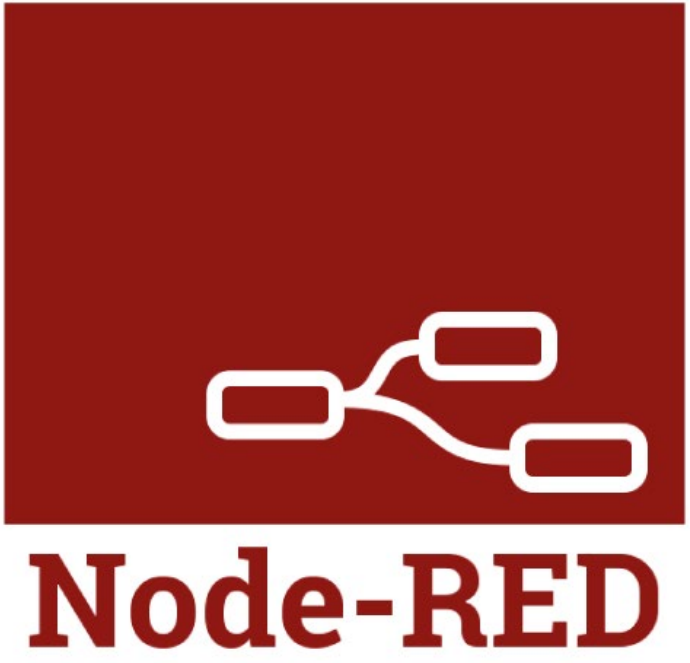
Optimización I: Infraestructura

Resumen de los componentes:

- **Node-Red** y **Grafana** permiten la **visualización** de datos IoT y la **gestión de flujos de trabajo**.
- **InfluxDB** se encarga del **almacenamiento de series temporales** para gestionar los datos recolectados por los sensores.
- **Flask** actúa como el **back-end**, sirviendo como interfaz entre el front-end y las bases de datos.
- **Nginx** optimiza el sistema, actuando como **servidor web** y balanceador de carga para garantizar la eficiencia del flujo de datos.

A continuación, veremos cada una de estas tecnologías, sus características, ventajas y desventajas. Así como otras herramientas similares.

Optimización I: Infraestructura

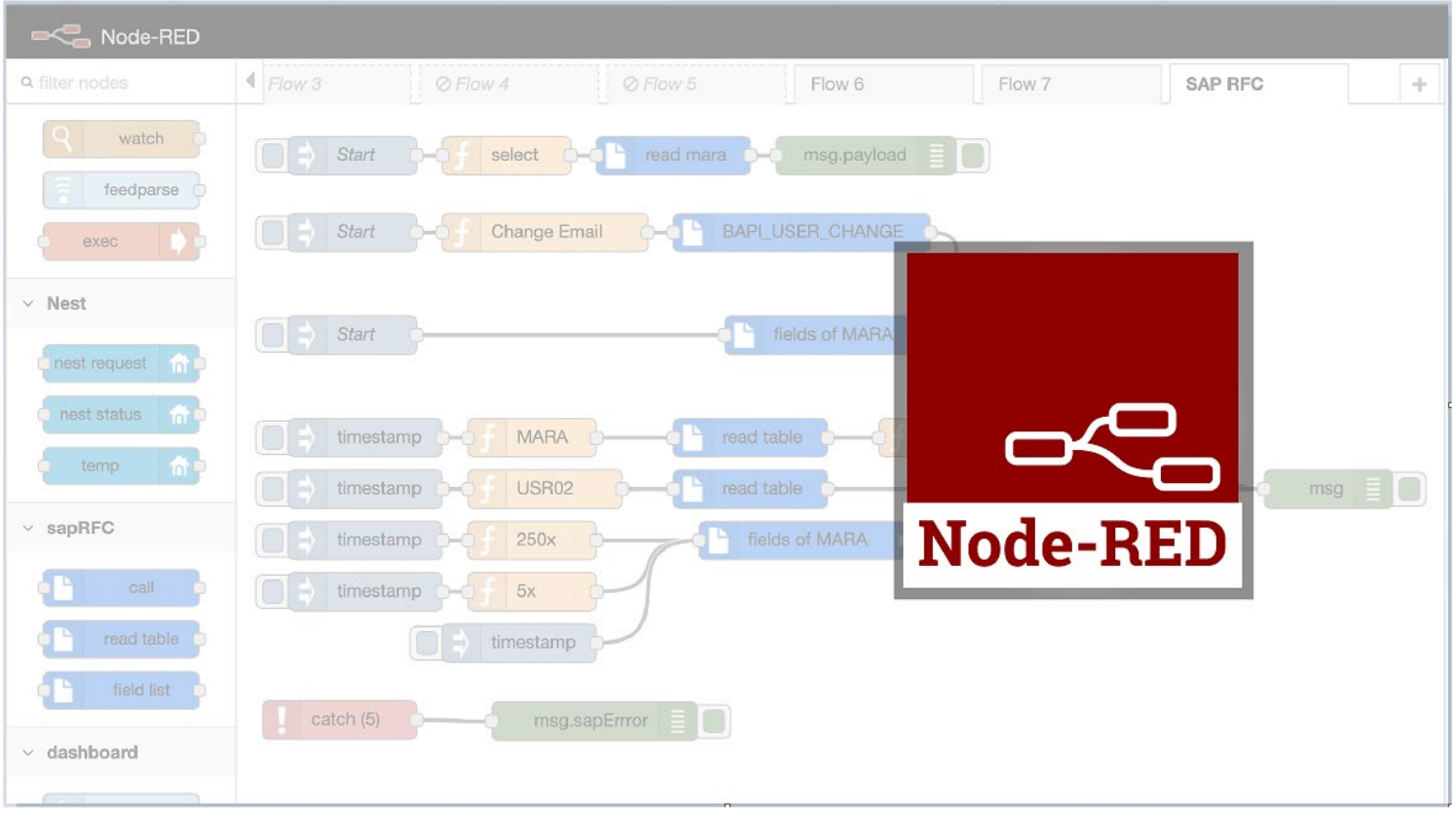


Optimización I: Infraestructura

- **Introducción a Node-RED: Plataforma de Desarrollo Basada en Flujos**

- **Node-RED** es una plataforma de desarrollo visual que permite crear aplicaciones interconectadas mediante flujos, utilizando nodos predefinidos que representan funciones específicas.
- Desarrollada inicialmente por IBM, se ha convertido en una herramienta popular en **IoT**, **automatización**, e **integración de sistemas**.
- Node-RED está construido sobre **Node.js**, lo que le otorga flexibilidad y escalabilidad, siendo ideal para dispositivos embebidos y servidores en la nube.
- **Características principales**:
  - **Interfaz gráfica** que facilita la creación de flujos sin escribir mucho código.
  - **Gran cantidad de nodos disponibles** para conectar dispositivos, servicios y APIs.
  - **Facilidad de integración** con múltiples tecnologías como MQTT, HTTP, WebSocket, y bases de datos.

Optimización I: Infraestructura



Optimización I: Infraestructura

- **¿Cómo Funciona Node-RED?**

- **Node-RED** permite crear aplicaciones conectadas arrastrando y soltando nodos en su interfaz visual.
- **Flujos de trabajo**: Los nodos están conectados por flujos, que representan el paso de datos entre funciones. Los flujos pueden incluir:
  - **Nodos de entrada**: Reciben datos de sensores, APIs o servicios.
  - **Nodos de procesamiento**: Ejecutan transformaciones o análisis de los datos (usando JavaScript, por ejemplo).
  - **Nodos de salida**: Envían resultados a otros dispositivos o servicios externos.
- **Componentes de la interfaz**:
  - **Paleta de nodos**: Selección de nodos disponibles agrupados por categoría.
  - **Área de trabajo**: Donde se arrastran y conectan los nodos para formar un flujo.
  - **Panel de propiedades**: Configuración específica de cada nodo.

Optimización I: Infraestructura

Características Principales de Node-RED

- **Interfaz gráfica basada en flujo**: Ideal para usuarios que prefieren un enfoque visual y rápido en el desarrollo de aplicaciones IoT.
- **Nodos predefinidos**: Gran variedad de nodos disponibles para conectar dispositivos y servicios, desde bases de datos hasta APIs como MQTT, HTTP, y WebSocket.
- **Escalabilidad y flexibilidad**: Al estar construido sobre **Node.js**, se adapta a dispositivos embebidos o servidores más robustos.
- **Ecosistema activo**: Al ser de código abierto, cuenta con una comunidad activa que contribuye constantemente con nuevos nodos y funcionalidades.
- **Integración fácil**: Compatible con una amplia gama de tecnologías, ideal para proyectos que involucren IoT, bases de datos, y servicios en la nube.

Optimización I: Infraestructura

Instalación y Configuración de Node-RED

- Node-RED se instala sobre **Node.js**, lo que facilita su configuración en sistemas embebidos y servidores.
- **Instalación**:
  - Instalar Node.js.
  - Ejecutar el comando: npm install -g node-red.
  - Iniciar Node-RED con node-red, lo que abrirá su interfaz gráfica en un navegador en http://localhost:1880.
- **Extensiones útiles para Visual Studio Code (VSCode)**:
- **Node-RED Extension**: Ejecuta y monitoriza Node-RED desde VSCode.
- **REST Client**: Para probar APIs que interactúan con Node-RED.
- **Node.js Debugging**: Para depurar flujos y scripts personalizados.

Optimización I: Infraestructura

- **Creación de Flujos en Node-RED**

- En Node-RED, los **flujos** se crean conectando nodos que representan distintas funciones.
- **Ejemplo de flujo IoT**:
  - **Nodo de entrada**: Recibe datos de sensores vía MQTT o HTTP.
  - **Nodo de procesamiento**: Analiza los datos y verifica si cumplen con ciertas condiciones (por ejemplo, un umbral de temperatura).
  - **Nodo de salida**: Envia alertas, guarda datos en una base de datos o los muestra en un dashboard.
- **Ventajas**: Permite la creación de aplicaciones de manera rápida y eficiente, sin la necesidad de escribir largas líneas de código.

Optimización I: Infraestructura

Dashboards y Visualización en Node-RED

- **Node-RED** ofrece la posibilidad de crear **dashboards interactivos** para la visualización de datos en tiempo real.
- Usando nodos como **node-red-dashboard**, puedes crear interfaces web personalizadas para monitorear sensores y dispositivos IoT.
- **Ventajas**:
  - **Visualización en tiempo real**: Ideal para monitorear dispositivos y recibir alertas inmediatas.
  - **Control remoto**: Los usuarios pueden interactuar con los dispositivos IoT desde un navegador.
- **Caso de uso**:
  - Monitorear temperatura, humedad o luz en tiempo real desde sensores IoT conectados a Node-RED.

Optimización I: Infraestructura

Aplicaciones de Node-RED en IoT

- **Node-RED** es ampliamente utilizado en diversos proyectos de **IoT** y automatización:
  - **Automatización del hogar**: Control de luces, cerraduras y sistemas de seguridad.
  - **Proyectos IoT**: Gestión de dispositivos conectados, recopilación de datos de sensores y control remoto.
  - **Integración de servicios**: Conectar APIs, bases de datos y servicios web con dispositivos IoT.
  - **Automatización empresarial**: Integrar y automatizar flujos de trabajo en entornos empresariales.
- **Versatilidad**: Es utilizado tanto en pequeños proyectos personales como en grandes infraestructuras empresariales.

Optimización I: Infraestructura

Conclusión sobre Node-RED

- **Node-RED** es una plataforma potente y flexible que permite a los usuarios desarrollar aplicaciones IoT de manera rápida y sencilla.
- Su **interfaz gráfica**, junto con la gran cantidad de nodos disponibles, facilita la **creación de flujos complejos** sin necesidad de escribir grandes cantidades de código.
- **Ideal para IoT y automatización**, permite la integración de dispositivos, bases de datos y servicios en la nube con facilidad.
- Su integración con herramientas como **VSCode** añade aún más potencia, permitiendo desarrollar tanto la lógica del dispositivo como los flujos de integración.

Optimización I: Infraestructura

Optimización I: Infraestructura

Introducción a Grafana

- **Grafana** es una plataforma de visualización y análisis de datos que permite monitorear y visualizar datos en tiempo real.
- Es ampliamente utilizada en proyectos de **IoT**, monitoreo de infraestructura, y análisis de aplicaciones.
- **Características principales**:
  - **Dashboards interactivos**: Personalizables y fáciles de usar.
  - **Visualización de datos**: Desde múltiples fuentes como bases de datos relacionales y de series temporales.
  - **Alertas configurables**: Notificaciones automáticas basadas en umbrales definidos.
  - **Compatibilidad**: Se integra con diversas bases de datos como **InfluxDB**, **MySQL**, **Prometheus**, entre otras.

Optimización I: Infraestructura

Características Principales de Grafana

- **Visualización interactiva**: Gráficos de líneas, barras, tablas y mapas de calor, totalmente personalizables.
- **Fuentes de datos compatibles**: Grafana se integra con una variedad de bases de datos, tales como:
  - **InfluxDB**: Ideal para series temporales en proyectos IoT.
  - **MySQL** y **PostgreSQL**: Para datos estructurados.
  - **Prometheus**: Enfocado en monitoreo.
  - **Elasticsearch** y **Loki**: Para datos de logs.
- **Alertas configurables**: Define umbrales y recibe notificaciones a través de canales como Slack, email o webhook.
- **Paneles colaborativos**: Los dashboards pueden compartirse o publicarse para diferentes equipos o propósitos.

Optimización I: Infraestructura

Optimización I: Infraestructura

Uso Básico de Grafana

- **Instalación**: Grafana puede instalarse en **Linux**, **Windows**, **macOS**, o como un contenedor **Docker**.
- **Conexión a fuentes de datos**: Después de la instalación, el primer paso es conectar Grafana a una base de datos:
  - InfluxDB, MySQL, PostgreSQL, entre otras.
  - Configuración desde el panel de administración: **Configuration > Data Sources**.
- **Creación de dashboards**:
  - Se crean paneles que contienen gráficos basados en consultas de las fuentes de datos.
  - Se pueden agregar múltiples gráficos a un mismo dashboard, permitiendo ver diversas métricas en un solo lugar.

Optimización I: Infraestructura

Ejemplo Práctico: Monitoreo de un Sistema IoT

- **Caso IoT**: Monitoreo de sensores de temperatura y humedad en un sistema IoT.
- **Conexión con InfluxDB**:
  - Los datos de los sensores se almacenan en una base de datos de series temporales como InfluxDB.
- **Dashboard en Grafana**:
  - Crear un gráfico de línea que muestre la evolución de la **temperatura**.
  - Otro gráfico para la **humedad** en tiempo real.
- **Alertas**: Configuración de alertas para que se envíe una notificación si la temperatura o humedad superan umbrales críticos.

Optimización I: Infraestructura

Configuración de Alertas en Grafana

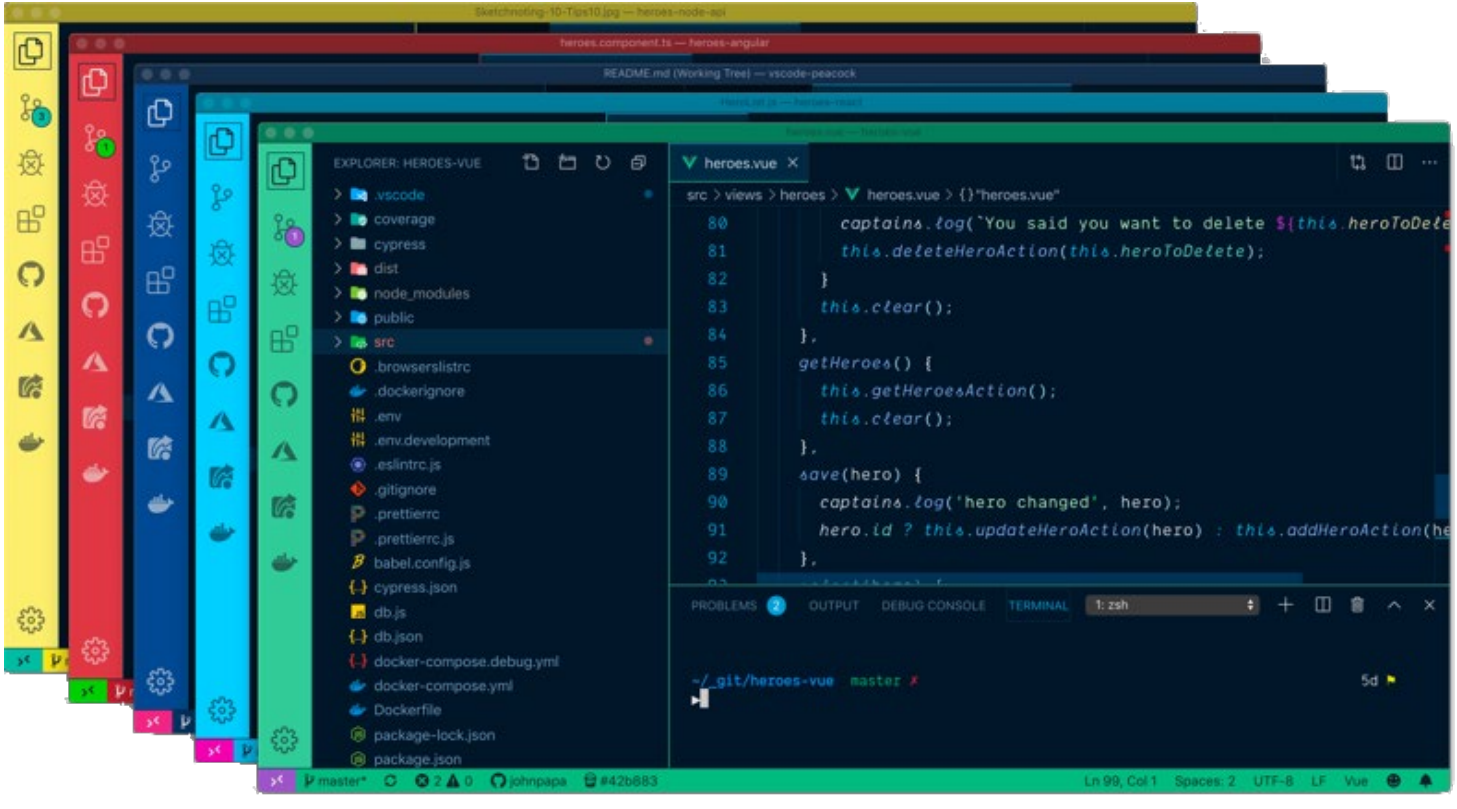
- **Alertas en tiempo real**: Grafana permite configurar alertas cuando ciertos valores de las métricas exceden umbrales predefinidos.
- **Proceso de configuración**:
  - Desde el panel de Grafana, selecciona el gráfico sobre el que quieres aplicar la alerta.
  - Navega a la pestaña de **Alertas** y define una **regla de umbral**.
  - Define las acciones que se ejecutarán cuando se dispare la alerta (enviar notificación por correo electrónico, Slack, etc.).
- **Ejemplo**: Si la **temperatura** en un sensor IoT excede los 30°C, se envía un correo de alerta.

Optimización I: Infraestructura

Cómo Integrar Grafana en un Proyecto IoT con VSCode

- **Arquitectura típica en un proyecto IoT**:
  - **Dispositivos IoT**: Los sensores (como ESP32) recolectan datos del entorno.
  - **Servidor HTTP o MQTT**: Los datos son enviados a un servidor central.
  - **Almacenamiento en InfluxDB**: Los datos se almacenan en una base de datos de series temporales.
  - **Visualización en Grafana**: Los datos son visualizados en tiempo real mediante dashboards dinámicos.
- **VSCode**: Se usa para desarrollar el software embebido y la lógica del servidor (HTTP o MQTT).
- **Integración con Python**:
  - Usa **Flask** o **Mosquitto** en VSCode para recibir los datos de los dispositivos.
  - Almacena los datos en **InfluxDB** para visualizarlos en **Grafana**.

Optimización I: Infraestructura



Optimización I: Infraestructura

Ejemplo Práctico: Proyecto IoT con Grafana y VSCode

- **Sistema IoT de monitoreo de cultivo**:

1. Los sensores de **temperatura** y **humedad** en un cultivo vertical envían datos cada 10 minutos.
2. Un servidor HTTP con **Flask** recibe los datos y los almacena en **InfluxDB**.
3. **Grafana** visualiza estos datos en tiempo real mediante gráficos interactivos.
4. Se configuran **alertas** si la temperatura o humedad exceden los umbrales críticos.

Optimización I: Infraestructura

Conclusión: El Papel de Grafana en Proyectos IoT

- **Grafana** es una herramienta que puede ser muy util en proyectos IoT que requieren monitoreo y visualización de datos en tiempo real.
- Su capacidad para integrarse con múltiples fuentes de datos y su flexibilidad en la personalización de dashboards hacen que sea ideal para proyectos de monitoreo de sensores.
- **Integración en VSCode**: Permite el desarrollo y gestión completa del proyecto IoT desde la recopilación de datos hasta su visualización, mejorando la eficiencia y la capacidad de análisis.
- **Alertas configurables**: Agrega valor al permitir notificaciones inmediatas cuando las métricas superan ciertos umbrales.

Optimización I: Infraestructura

Optimización I: Infraestructura

Introducción a InfluxDB

- **InfluxDB** es una base de datos de **series temporales** diseñada para manejar grandes volúmenes de datos que varían con el tiempo, como métricas de servidores, sensores IoT y aplicaciones financieras.
- Parte del ecosistema **TICK Stack** (Telegraf, InfluxDB, Chronograf y Kapacitor).
- **Optimización para series temporales**: InfluxDB almacena y consulta datos de manera eficiente en función de sus marcas temporales.
- **Flexibilidad**: No requiere esquemas rígidos, lo que facilita la inserción de datos sin cambios estructurales.
- **Escalabilidad**: Diseñada para manejar grandes volúmenes de datos de manera escalable.

Optimización I: Infraestructura

Características Principales de InfluxDB

- **Optimización para datos temporales**: Diseñada para manejar datos con marcas de tiempo.
- **Consultas SQL-like**: Utiliza **InfluxQL** (similar a SQL) para consultas avanzadas.
- **Políticas de retención de datos**: Gestión automática de eliminación de datos antiguos para optimizar el almacenamiento.
- **No requiere esquemas**: Flexibilidad para agregar nuevos datos sin necesidad de definir esquemas complejos.
- **Alta escalabilidad**: Maneja grandes volúmenes de datos de forma horizontal y vertical.

Optimización I: Infraestructura

Componentes Clave de InfluxDB

1. **Measurement (Medición)**: Similar a una tabla en bases de datos relacionales, agrupa series temporales con una métrica común.
2. **Tags (Etiquetas)**: Claves/valores que organizan y clasifican los datos, por ejemplo, por sensor o ubicación.
3. **Fields (Campos)**: Los valores reales de los datos, como temperatura o presión.
4. **Timestamp (Marca de tiempo)**: Fecha y hora en que se registró el dato.

- Ejemplo:
  - Medición: **temperatura**
  - Tags: **ciudad = "Córdoba"**, **sensor = "sensor_01"**
  - Field: **valor_temperatura = 25.3**
  - Timestamp: **2024-09-07T14:00:00Z**

Optimización I: Infraestructura

Medicion #1

tag set

tag key

Sensor	Location	Temperature	time
Sensor 1	USA	40	12/14 @ 15:16 pm
Sensor 1	France	38	13/14 @ 11:16 am
Sensor 1	USA	41	12/11 @ 11:16 am
Sensor 1	Luxembourg	38	11/11 @ 12:14 am

tag value

Optimización I: Infraestructura

Casos de Uso Comunes de InfluxDB

- **Monitoreo de Infraestructura**:
  - Recolección de métricas de servidores, bases de datos y aplicaciones.
- **Internet de las Cosas (IoT)**:
  - Almacena datos de sensores distribuidos que generan información constante.
- **Aplicaciones Financieras**:
  - Seguimiento de valores financieros y su variación en el tiempo.
- **Monitoreo de rendimiento de aplicaciones**:
  - Permite realizar análisis y visualizar tendencias en métricas clave de rendimiento.

Optimización I: Infraestructura

- **Flujo de Trabajo Básico de InfluxDB**

- **Instalación y configuración**: InfluxDB puede instalarse localmente, en la nube o utilizar la versión gestionada (InfluxDB Cloud).
- **Recolección de datos**:
  - Utiliza **Telegraf** para capturar métricas de servidores, sensores o aplicaciones.
  - Datos insertados a través de la API o clientes como Python, Node.js.
- **Almacenamiento y consulta de datos**:
  - Los datos se almacenan con **marcas de tiempo** y se consultan usando **InfluxQL** o **Flux** para análisis avanzados.
- **Visualización**: Integración con herramientas como **Grafana** para dashboards interactivos.

Optimización I: Infraestructura

Integración de InfluxDB en Proyectos IoT

- **InfluxDB** es ideal para almacenar datos generados por sensores IoT.

- **Arquitectura básica**:
  - **Microcontroladores** (ESP32, Raspberry Pi) recopilan datos.
  - Los datos se envían a un servidor central vía **HTTP** o **MQTT**.
  - **InfluxDB** almacena estos datos para análisis y consultas posteriores.

- **Integración con Grafana**: Grafana se conecta a InfluxDB para visualizar estos datos en tiempo real.

Optimización I: Infraestructura

Optimización I: Infraestructura

Optimización I: Infraestructura

- **¿Qué es un Framework? Presentación y Usos**

- Un **framework** es una estructura o plataforma de trabajo predefinida que proporciona un conjunto de herramientas, bibliotecas y directrices para desarrollar software. Los frameworks son fundamentales para ayudar a los desarrolladores a construir aplicaciones de manera más eficiente y con menos errores, proporcionando un marco estandarizado para resolver problemas comunes.

Optimización I: Infraestructura

Definición de Framework

- Un **framework** es un conjunto de bibliotecas y herramientas reutilizables que simplifican el desarrollo de aplicaciones al ofrecer componentes predefinidos, organización y flujos de trabajo comunes. En lugar de escribir código desde cero, los desarrolladores pueden aprovechar el framework para acceder a funcionalidades listas para usar y enfocarse en la lógica específica de su aplicación.

Optimización I: Infraestructura

Características principales de un Framework:

1. **Estructura**: Proporciona una estructura organizada que define cómo se debe organizar y construir el código.
2. **Reutilización**: Permite reutilizar componentes de software comunes, reduciendo el tiempo de desarrollo.
3. **Abstracción**: Simplifica la complejidad técnica al ocultar detalles internos, ofreciendo herramientas listas para resolver problemas comunes.
4. **Estándares y Buenas Prácticas**: Unifica el código de una manera que sigue buenas prácticas de la industria.

Optimización I: Infraestructura

Tipos de Frameworks según su Aplicación

- **a) Frameworks de Desarrollo Web**

- Estos frameworks facilitan la creación de aplicaciones web. Algunos ejemplos conocidos incluyen:
- **Django** (Python): Altamente estructurado, incluye ORM, autenticación, y muchas más características listas para usar.
- **Rails** (Ruby): Otro framework completo que sigue el principio de "Convención sobre Configuración".
- **Express** (Node.js): Minimalista y rápido, utilizado principalmente para construir APIs.

Optimización I: Infraestructura

- **b) Frameworks de Desarrollo de APIs**

- **Flask** y **FastAPI** (Python): Diseñados para crear **APIs REST** y aplicaciones web ligeras, proporcionando una gran flexibilidad en el manejo de rutas y métodos HTTP.

- **Spring Boot** (Java): Un framework popular para construir microservicios y APIs REST con soporte completo para integración con bases de datos, seguridad, etc.

Optimización I: Infraestructura

- **c) Frameworks de Desarrollo de Aplicaciones Móviles**
- **Flutter** (Dart): Facilita el desarrollo de aplicaciones móviles nativas para Android e iOS desde una base de código única.
- **React Native** (JavaScript): Permite el desarrollo de aplicaciones móviles multiplataforma con una base de código compartida.

- **d) Frameworks de Desarrollo de Escritorio**
- **Qt** (C++): Para crear aplicaciones de escritorio en múltiples plataformas.
- **Electron** (JavaScript/HTML/CSS): Facilita la creación de aplicaciones de escritorio con tecnologías web.

Optimización I: Infraestructura

Cómo se Usa un Framework

- El uso de un framework implica seguir ciertas convenciones predefinidas. A continuación, se presenta un flujo general de cómo se utiliza un framework:

- **a) Instalación y Configuración Inicial**
- El primer paso es instalar el framework, lo que generalmente incluye la configuración de un entorno de desarrollo y la instalación de dependencias.
- Por ejemplo, en un framework de desarrollo web como **Django**, la instalación podría ser algo así:

```
pip install django
django-admin startproject mi_proyecto
```


Optimización I: Infraestructura

- **b) Definir la Estructura del Proyecto**
- Un framework proporciona una estructura de proyecto estándar. En un proyecto web, por ejemplo, define directorios para **vistas**, **modelos** (si usa un ORM), **controladores** y **archivos de configuración**.

- **c) Agregar Funcionalidades**
- A medida que el proyecto crece, el desarrollador agrega nuevas funcionalidades utilizando los componentes y las herramientas que el framework proporciona. Esto incluye definir rutas, interactuar con bases de datos, y añadir controladores para manejar la lógica de negocio.

- **d) Extender el Framework**
- Muchos frameworks permiten ser extendidos con **plugins** o **extensiones**. Si el framework no tiene alguna funcionalidad nativa, generalmente puedes integrarla mediante módulos adicionales.

Optimización I: Infraestructura

- **Ventajas y Desventajas de Usar un Framework**

- **Ventajas:**
- **Ahorro de Tiempo**: Ya no es necesario reinventar la rueda para funcionalidades comunes como autenticación, validación de datos, etc.
- **Estandarización**: El código sigue un patrón que facilita la colaboración y el mantenimiento a largo plazo.
- **Seguridad**: Los frameworks suelen incluir medidas de seguridad integradas, como protección contra ataques **CSRF**, **SQL Injection**, etc.
- **Mantenibilidad**: Al tener una estructura clara y organizada, es más fácil mantener y escalar la aplicación.

- **Desventajas:**
- **Curva de Aprendizaje**: Puede ser necesario aprender las convenciones y patrones específicos de cada framework.
- **Limitaciones**: Algunos frameworks son más rígidos que otros, lo que puede limitar la flexibilidad del desarrollador.

Optimización I: Infraestructura

- Un **framework** proporciona una base estructurada y herramientas listas para que los desarrolladores puedan crear aplicaciones más rápidamente y con menos errores. Dependiendo del tipo de aplicación (web, móvil, escritorio, APIs), existen diferentes frameworks que ayudan a abordar tareas comunes, como el manejo de protocolos de comunicación.

- Aunque frameworks como **Flask** y **FastAPI** son principalmente para **HTTP**, existen otros frameworks como **Node.js** y **Phoenix** que permiten manejar múltiples protocolos en forma nativa, como **HTTP**, **WebSockets**, **MQTT**, y más. Esto facilita el desarrollo de aplicaciones distribuidas y sistemas que requieren alta concurrencia y soporte para diferentes tipos de comunicación.

Optimización I: Infraestructura

Flask y FastAPI

- **Flask** es un framework de Python minimalista y liviano que se utiliza para crear aplicaciones web y APIs. Fue diseñado para ofrecer flexibilidad, dejando muchas decisiones de arquitectura al desarrollador. Flask no impone una estructura rígida ni muchas dependencias, lo que lo convierte en una excelente opción para proyectos pequeños o para desarrolladores que desean tener control total sobre su aplicación.
- **FastAPI** es un framework moderno para construir APIs web utilizando Python. Se destaca por su velocidad, soporte para programación asíncrona, y su integración directa con la validación de tipos gracias a **Pydantic** y la documentación automática con **Swagger**.

Optimización I: Infraestructura

- **Características Principales de Flask:**

1. **Simplicidad y Flexibilidad**: Flask permite a los desarrolladores crear aplicaciones web rápidamente y sin mucho código innecesario.
2. **Escalabilidad**: Aunque es simple, Flask se puede extender con módulos adicionales para agregar funcionalidades como autenticación, manejo de bases de datos, etc.
3. **Soporte para APIs**: Flask es ideal para construir APIs RESTful. Utiliza rutas (URLs) para definir diferentes puntos finales y puede interactuar con bases de datos para devolver datos.

Optimización I: Infraestructura

- **Características Principales de FastAPI:**

1. **Alto Rendimiento**: FastAPI es extremadamente rápido, comparable a frameworks en otros lenguajes de alto rendimiento como Node.js o Go. Esto lo convierte en una excelente opción para APIs que requieren muchas solicitudes simultáneas.
2. **Programación Asíncrona**: FastAPI permite usar async y await para manejar de manera eficiente tareas de entrada/salida, como consultas a bases de datos.
3. **Validación Automática de Datos**: Utiliza Pydantic para la validación automática de datos de entrada basados en los tipos definidos.
4. **Documentación Automática**: Genera documentación interactiva de la API automáticamente con Swagger y Redoc.

Optimización I: Infraestructura

Características	Flask	FastAPI
Velocidad	Buena	Excelente (optimizada para alto rendimiento)
Programación Asíncrona	No está integrado nativamente	Soporte nativo para async y await
Simplicidad	Muy simple, ideal para aplicaciones pequeñas	Simplicidad con validación automática
Escalabilidad	Escalable, pero necesita más configuración	Altamente escalable
Documentación	Se requiere generar manualmente	Documentación automática con Swagger
Validación de Datos	No nativa, se necesita una biblioteca externa	Validación automática con Pydantic

Optimización I: Infraestructura

Uso en un Proyecto con Base de Datos:

- Ambos frameworks son compatibles con varias bases de datos, pero FastAPI tiene una ventaja si trabajas en proyectos donde la **alta concurrencia** o la **escalabilidad** es una prioridad, ya que permite manejar múltiples conexiones a bases de datos de forma más eficiente.

- **Arquitectura Típica:**

1. **Servidor API**: El API se ejecuta en Flask o FastAPI y recibe solicitudes HTTP (GET, POST, PUT, DELETE).
2. **Base de Datos**: Se puede conectar a una base de datos (MySQL, PostgreSQL, MongoDB, etc.) utilizando un ORM (Object Relational Mapping) como SQLAlchemy.
3. **Clientes**: Las aplicaciones móviles o web consumen el API enviando solicitudes y recibiendo datos en formato JSON.

Optimización I: Infraestructura

Uso de Flask y FastAPI para desarrollar servidores de software

- Ambos frameworks son ampliamente utilizados para crear **servidores HTTP**, lo que los convierte en una excelente opción para manejar solicitudes **RESTful APIs**, donde HTTP es el protocolo subyacente. Sin embargo, para manejar otros protocolos, como MQTT o WebSocket, necesitarás bibliotecas adicionales, ya que Flask y FastAPI están centrados en HTTP.

- **Protocolos Soportados: HTTP y WebSocket**
- Tanto Flask como FastAPI son ideales para manejar el protocolo HTTP. Puedes desarrollar APIs REST que funcionen bajo métodos HTTP como **GET**, **POST**, **PUT**, **DELETE**, etc.
- Las aplicaciones que desarrolles con Flask o FastAPI se ejecutarán típicamente en servidores HTTP (como **Gunicorn** o **Uvicorn**).
- En **FastAPI**, el soporte para WebSocket es nativo.
- **Flask** no tiene soporte nativo para WebSocket, pero puedes agregarlo utilizando bibliotecas adicionales como **Flask-SocketIO**.

Optimización I: Infraestructura

Conclusión

- Tanto **Flask** como **FastAPI** son muy versátiles y, si bien están optimizados para el protocolo **HTTP**, pueden manejar otros protocolos como **MQTT**, **WebSocket**, **gRPC**, y otros a través de bibliotecas especializadas. FastAPI es particularmente adecuado para aplicaciones que requieren un rendimiento superior y soporte asíncrono.
- Esto te permite trabajar con servidores web completos con capacidades extendidas para IoT, mensajería en tiempo real y servicios distribuidos.
- En **Python** existen varios frameworks además de **Flask** y **FastAPI** que permiten crear servidores y manejar múltiples protocolos. Algunos de los más destacados que ofrecen más versatilidad para trabajar con diferentes protocolos o que pueden manejar múltiples servidores son:

Optimización I: Infraestructura

- **1. Tornado**
- **Tornado** es un framework de Python diseñado para manejar aplicaciones web en tiempo real y de alta concurrencia. A diferencia de Flask o FastAPI, **Tornado** está optimizado para trabajar de forma asíncrona desde su base, lo que lo hace muy eficiente para manejar miles de conexiones simultáneas, como en el caso de **WebSockets** o servicios que requieren tiempos de respuesta muy bajos.
- **Características:**
- **Manejo Asíncrono**: Permite manejar múltiples solicitudes de forma concurrente sin bloquear el servidor.
- **Soporte para WebSockets**: Tornado es ideal para aplicaciones que necesitan comunicación bidireccional en tiempo real.
- **Manejo de Servidores HTTP y TCP**: Tornado puede manejar no solo HTTP, sino también otros protocolos como TCP.

Optimización I: Infraestructura

- **2. Twisted**
- **Twisted** es uno de los frameworks más completos y versátiles en Python para manejar redes, diseñado específicamente para crear aplicaciones que necesitan manejar múltiples protocolos de red. **Twisted** soporta **HTTP**, **FTP**, **SMTP**, **TCP**, **UDP**, **SSH**, **IRC**, entre otros, lo que lo convierte en una excelente opción para servidores que necesitan manejar múltiples tipos de comunicación.
- **Características:**
- **Soporte para múltiples protocolos**: HTTP, TCP, UDP, WebSockets, y muchos más.
- **Altamente escalable**: Ideal para aplicaciones de red complejas y distribuidas.
- **Manejo asíncrono**: Al igual que Tornado, está diseñado para manejar múltiples conexiones de forma concurrente.
- Con **Twisted**, es posible ejecutar múltiples servidores (por ejemplo, uno HTTP y uno TCP) dentro de la misma aplicación, utilizando diferentes puertos o protocolos.

Optimización I: Infraestructura

- **3. Sanic**
- **Sanic** es otro framework web asíncrono que está diseñado para manejar solicitudes HTTP de manera extremadamente rápida, similar a **FastAPI**. **Sanic** es ideal si estás buscando una alternativa ligera que ofrezca un alto rendimiento, pero también puede extenderse para manejar otros tipos de protocolos mediante módulos adicionales.
- **Características:**
- **Asíncrono**: Sanic es nativamente asíncrono, lo que lo hace adecuado para aplicaciones que requieren manejar múltiples conexiones.
- **Alto rendimiento**: Diseñado para ser uno de los frameworks web más rápidos de Python.
- **Soporte para WebSockets**: Sanic tiene soporte nativo para WebSockets, lo que permite el manejo de aplicaciones en tiempo real.

Optimización I: Infraestructura

- **4. AIOHTTP**
- **AIOHTTP** es un framework web asíncrono en Python que se basa en la biblioteca **asyncio**. Este framework es una excelente opción para aplicaciones que necesitan manejar grandes cantidades de tráfico de red de forma eficiente. **AIOHTTP** no solo maneja HTTP, sino que también permite gestionar conexiones **WebSocket** y puede ser adaptado para manejar otros protocolos.
- **Características:**
  - **Asíncrono**: Utiliza asyncio para manejar solicitudes de forma eficiente.
  - **Soporte para WebSockets:** AIOHTTP tiene soporte nativo para WebSockets.
  - **Altamente configurable:** Se puede adaptar para manejar varios tipos de protocolos de red, no solo HTTP.
- **Múltiples Protocolos:**
- Aunque **AIOHTTP** es conocido por manejar **HTTP** y **WebSockets**, puede ampliarse con **asyncio** para manejar otras formas de entrada/salida, como **TCP** y **UDP**, en paralelo.

Optimización I: Infraestructura

Framework	Lenguaje	Protocolos Soportados	Asíncrono	Usos Principales	Escalabilidad	Documentación Automática
Django	Python	HTTP	No	Desarrollo web completo, APIs	Alta	No
Flask	Python	HTTP	No	APIs REST y aplicaciones ligeras	Media	No
FastAPI	Python	HTTP, WebSockets	Sí	APIs REST de alto rendimiento	Alta	Sí (Swagger/OpenAPI)
Tornado	Python	HTTP, WebSockets, TCP, UDP	Sí	Aplicaciones en tiempo real, alta concurrencia	Alta	No
Twisted	Python	HTTP, FTP, SMTP, TCP, UDP, SSH, IRC, WebSockets	Sí	Redes, múltiples protocolos, IoT	Alta	No

Optimización I: Infraestructura

Framework	Lenguaje	Protocolos Soportados	Asíncrono	Usos Principales	Escalabilidad	Documentación Automática
Sanic	Python	HTTP, WebSockets	Sí	APIs REST rápidas, aplicaciones ligeras	Alta	No
AIOHTTP	Python	HTTP, WebSockets, TCP, UDP	Sí	APIs REST y aplicaciones en tiempo real	Alta	No
Express.js	JavaScript	HTTP	No	APIs REST, aplicaciones web	Alta	No
Koa.js	JavaScript	HTTP	Sí	APIs minimalistas	Alta	No
Spring Boot	Java	HTTP, WebSockets	No	Aplicaciones empresariales, microservicios	Alta	Sí

Optimización I: Infraestructura

Framework	Lenguaje	Protocolos Soportados	Asíncrono	Usos Principales	Escalabilidad	Documentación Automática
Phoenix	Elixir	HTTP, WebSockets, TCP	Sí	Aplicaciones en tiempo real	Muy Alta	No
Ruby on Rails	Ruby	HTTP	No	Desarrollo web completo, APIs	Alta	No
Node.js	JavaScript	HTTP, WebSockets, TCP, UDP, MQTT	Sí	APIs, microservicios, IoT	Muy Alta	No
Gin	Go	HTTP	Sí	APIs REST rápidas, microservicios	Alta	No
ASP.NET Core	C#	HTTP, WebSockets	No	Aplicaciones empresariales	Alta	Sí

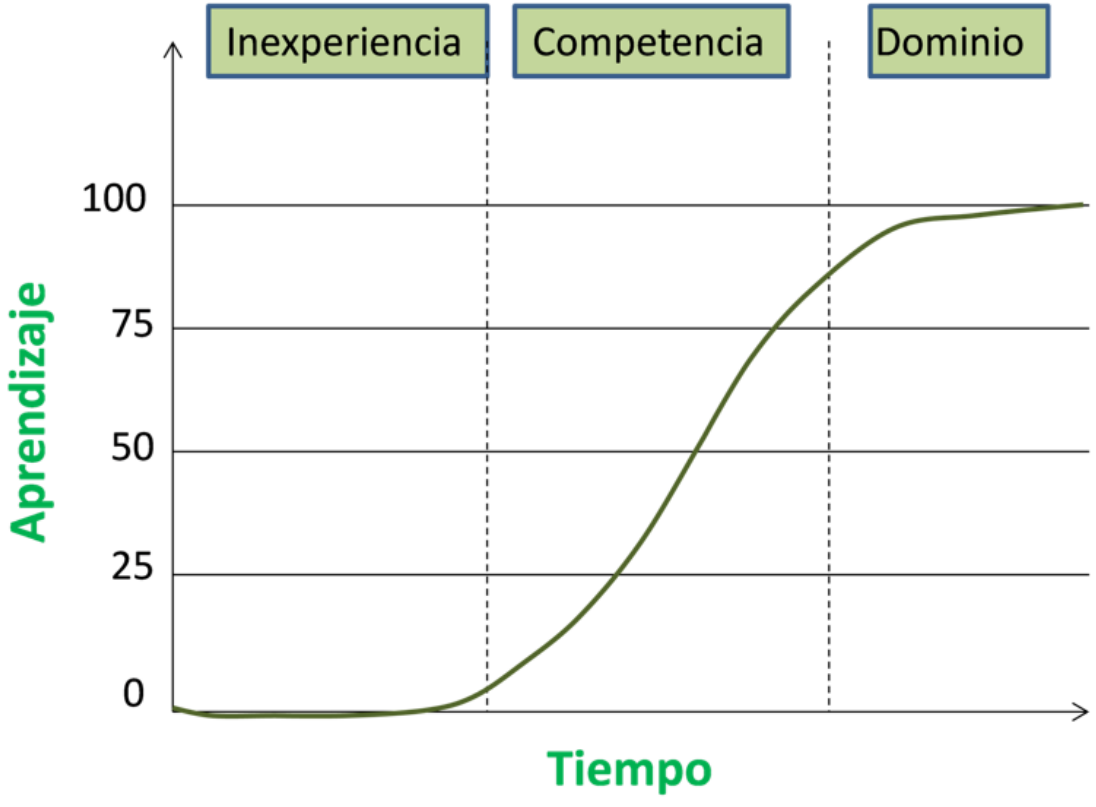
Optimización I: Infraestructura

Resumen

1. **Django**: Ofrece un enfoque completo para desarrollar aplicaciones web, pero no es asíncrono, por lo que no es la mejor opción para alta concurrencia o tiempo real.
2. **Flask**: Excelente para aplicaciones ligeras y APIs REST simples, pero no maneja asincronía de forma nativa.
3. **FastAPI**: Destacado por su rendimiento y soporte nativo para asincronía, ideal para APIs de alto rendimiento con documentación automática.
4. **Tornado**: Perfecto para aplicaciones que requieren manejo en tiempo real y alto volumen de conexiones, ya que soporta varios protocolos como TCP y UDP además de HTTP.
5. **Twisted**: Uno de los más versátiles, con soporte para múltiples protocolos, lo que lo convierte en la opción ideal para aplicaciones de red complejas o IoT.
6. **Sanic** y **AIOHTTP**: Alternativas rápidas y asíncronas, especialmente adecuadas para manejar grandes cantidades de tráfico y aplicaciones en tiempo real.

Optimización I: Infraestructura

La curva de aprendizaje



Optimización I: Infraestructura

Curva de Aprendizaje de Frameworks en Diferentes Lenguajes

- La **curva de aprendizaje** de un framework se refiere a qué tan fácil o difícil es aprender a usarlo de manera efectiva.
- Para **principiantes**, frameworks como **Flask**, **Express.js**, y **Node.js** son ideales por su sencillez y flexibilidad, lo que permite aprender rápidamente los fundamentos del desarrollo de servidores y APIs.
- **Frameworks intermedios** como **Django** y **FastAPI** proporcionan más herramientas y control, pero requieren un mayor nivel de comprensión del lenguaje y las buenas prácticas.
- Frameworks más **complejos** como **Tornado**, **Twisted**, y **Phoenix** están diseñados para aplicaciones en tiempo real y de alta concurrencia, lo que los convierte en opciones poderosas, pero con una curva de aprendizaje pronunciada.
- La elección del framework depende del nivel de experiencia del desarrollador, el tipo de proyecto, y la necesidad de manejar múltiples protocolos o escalabilidad.

Optimización I: Infraestructura

Factores a Considerar en la Curva de Aprendizaje

1. **Similitud con el lenguaje base**: Frameworks que siguen una estructura similar al lenguaje base tienden a ser más fáciles de aprender. Por ejemplo, **Flask** es simple porque sigue las convenciones básicas de Python, mientras que **Twisted** requiere aprender conceptos más avanzados de programación de redes.
2. **Funcionalidades integradas**: Frameworks como **Django** y **Ruby on Rails** vienen con muchas herramientas preconstruidas, lo que puede acelerar el desarrollo. Sin embargo, esta cantidad de herramientas también puede requerir más tiempo para aprender a usarlas eficientemente.
3. **Programación asíncrona**: Los frameworks que soportan la **programación asíncrona** (como **Tornado**, **Twisted**, **FastAPI**, y **AIOHTTP**) generalmente tienen una curva de aprendizaje más pronunciada debido a la necesidad de comprender conceptos avanzados de concurrencia y async/await.
4. **Escalabilidad y complejidad**: Frameworks como **Spring Boot**, **Twisted** y **Phoenix** están diseñados para manejar aplicaciones de gran escala, lo que los hace más complejos de aprender y dominar, pero ofrecen mayor capacidad para proyectos grandes.

Optimización I: Infraestructura

Curva de Aprendizaje de Frameworks en Python

Framework	Curva de Aprendizaje	Comentarios
Django	Media	Ofrece muchas funcionalidades listas para usar, pero tiene una estructura y configuración más rígidas.
Flask	Baja	Muy sencillo y flexible, ideal para principiantes que quieren aprender a crear APIs rápidamente.
FastAPI	Media	Fácil de aprender, pero requiere cierto conocimiento de asíncronía para aprovechar todo su potencial.
Tornado	Alta	Su enfoque asíncrono y en tiempo real lo hace más difícil de dominar.
Twisted	Muy Alta	Complejo, especialmente si se desea aprovechar sus capacidades para manejar múltiples protocolos.
Sanic	Media	Similar a Flask, pero su enfoque asíncrono añade un nivel extra de dificultad.
AIOHTTP	Media	Flexible y potente, pero exige un conocimiento adecuado de la programación asíncrona en Python.

Optimización I: Infraestructura

Conclusión

- Para **principiantes**, frameworks como **Flask**, **Express.js**, y **Node.js** son ideales por su sencillez y flexibilidad, lo que permite aprender rápidamente los fundamentos del desarrollo de servidores y APIs.
- **Frameworks intermedios** como **Django** y **FastAPI** proporcionan más herramientas y control, pero requieren un mayor nivel de comprensión del lenguaje y las buenas prácticas.
- Frameworks más **complejos** como **Tornado**, **Twisted**, y **Phoenix** están diseñados para aplicaciones en tiempo real y de alta concurrencia, lo que los convierte en opciones poderosas, pero con una curva de aprendizaje pronunciada.
- La elección del framework depende del nivel de experiencia del desarrollador, el tipo de proyecto, y la necesidad de manejar múltiples protocolos o escalabilidad.

Optimización I: Infraestructura

Optimización I: Infraestructura

Client	Servers
Hardware	Hardware
	
Software	Software
	

Optimización I: Infraestructura

Servidores

- El término "servidor" puede referirse tanto al **hardware** que almacena, procesa y distribuye información, como al **software** que gestiona cómo esta información se entrega a los usuarios o a otros sistemas. Existen distintos tipos de servidores que se clasifican según el tipo de servicios que proveen, como servidores de **almacenamiento**, **procesamiento**, **bases de datos**, o **comunicación**. Estos servidores pueden estar desplegados en infraestructuras físicas, virtualizadas o en la nube.
- Los servidores software son componentes fundamentales para que las aplicaciones modernas funcionen de manera eficiente. Sin embargo, cada servidor tiene un propósito específico y no puede actuar solo en un entorno complejo sin interactuar con otros sistemas.

Optimización I: Infraestructura

Tipos de Servidores Software

- Cuando hablamos de servidores software, nos referimos a las aplicaciones encargadas de gestionar peticiones de los usuarios o dispositivos, procesarlas y devolver respuestas. Ejemplos de servidores software incluyen:

1. **Servidores Web** (Nginx, Apache): Gestionan solicitudes HTTP y entregan contenido web a los navegadores.
2. **Brokers MQTT** (como Mosquitto): Facilitan la comunicación entre dispositivos IoT utilizando el protocolo MQTT.
3. **Servidores de Aplicaciones** (Flask, FastAPI): Procesan la lógica de negocio y manejan las peticiones API.

Optimización I: Infraestructura

Nginx: El Proxy Inverso

- Uno de los componentes clave en la arquitectura de servidores modernos es el **proxy inverso**, una capa intermedia que gestiona las solicitudes entrantes y las redirige a los servidores backend correctos. **Nginx**, una de las soluciones más populares, no solo funciona como servidor web, sino que también actúa como proxy inverso, equilibrador de carga y acelerador de contenido.
- La razón por la cual soluciones como **Nginx** son esenciales es porque los servidores de aplicaciones (como Flask o FastAPI) no están diseñados para manejar directamente grandes volúmenes de tráfico o gestionar directamente las conexiones seguras a través de HTTPS en producción. Ahí es donde entra Nginx, ofreciendo un manejo eficiente de solicitudes, seguridad, y escalabilidad.

Optimización I: Infraestructura

- **La Relación entre Nginx y Otros Servidores**

- Nginx se coloca frente a los servidores de aplicaciones para gestionar las solicitudes HTTP en el puerto 80 o HTTPS en el puerto 443. Luego, redirige esas solicitudes a los servidores backend que manejan la lógica de negocio, como Flask o FastAPI, que suelen ejecutarse en puertos internos como el 5000 o 8000. Este patrón de diseño garantiza que las aplicaciones sean escalables, seguras y capaces de manejar múltiples usuarios simultáneos sin sobrecargar el servidor de backend.

Optimización I: Infraestructura

Resumen de Roles

1. **Servidor físico o virtual**: La computadora (real o virtual) que corre un sistema operativo.
2. **Sistema Operativo**: Linux, Windows, etc., que administra los recursos del hardware y permite instalar software.
3. **Servicios (SGDB, APIs, Contenedores)**: Son los programas que realmente procesan los datos, manejan las bases de datos o ejecutan la lógica de negocio (por ejemplo, una API).
4. **Nginx**: Un **servidor especializado** que maneja eficientemente las solicitudes HTTP y HTTPS, balancea la carga entre varios servicios, actúa como proxy inverso, gestiona la caché y optimiza el rendimiento del sistema en general.

Optimización I: Infraestructura

Optimización I: Infraestructura

Introducción a Nginx

- Nginx es un servidor web de alto rendimiento y código abierto que también puede funcionar como servidor proxy inverso, balanceador de carga, y servidor de caché HTTP. Fue creado por Igor Sysoev en 2004 con el objetivo de gestionar problemas de **concurrencia masiva**, lo que lo convierte en una opción popular para manejar un gran número de conexiones simultáneas de manera eficiente.

Optimización I: Infraestructura

Características Principales de Nginx

1. **Servidor Web**: Es capaz de servir contenido estático (como archivos HTML, CSS, JavaScript e imágenes) de manera rápida y eficiente.
2. **Proxy Inverso**: Nginx puede actuar como intermediario entre los clientes y los servidores backend, lo que mejora el rendimiento y la seguridad al distribuir las solicitudes.
3. **Balanceador de Carga**: Nginx distribuye el tráfico entre varios servidores, mejorando la disponibilidad y escalabilidad de una aplicación.
4. **Servidor de Caché HTTP**: Nginx puede almacenar en caché respuestas del servidor backend para reducir la carga y acelerar el tiempo de respuesta.
5. **Alta Concurrencia**: Su arquitectura basada en eventos lo hace extremadamente eficiente, capaz de manejar decenas de miles de conexiones concurrentes sin consumir demasiados recursos.

Optimización I: Infraestructura

¿Cómo Funciona Nginx?

- Nginx utiliza un **modelo basado en eventos** para gestionar las conexiones, a diferencia de los servidores tradicionales que utilizan un modelo basado en procesos o hilos. En lugar de asignar un hilo por conexión, Nginx maneja las solicitudes de manera asíncrona, lo que reduce significativamente el uso de memoria y mejora el rendimiento bajo cargas altas.

Casos de Uso

- **Aplicaciones web de alto tráfico**: Gracias a su capacidad de manejar miles de conexiones simultáneas, es utilizado por grandes plataformas como Netflix, GitHub, y WordPress.
- **Mejora de seguridad**: Al utilizar Nginx como proxy inverso, se oculta la arquitectura interna de la red y se pueden aplicar medidas de seguridad adicionales.
- **Optimización de tiempos de respuesta**: La capacidad de cacheo y balanceo de carga contribuye a mejorar la velocidad de entrega de contenido.

Optimización I: Infraestructura

Integración de Nginx

- En un proyecto de **Internet de las Cosas (IoT)**, Nginx puede desempeñar un papel fundamental como servidor para manejar la comunicación entre dispositivos IoT y servidores backend, gestionar las solicitudes HTTP y actuar como proxy inverso para servicios en la nube. Su capacidad para manejar grandes volúmenes de tráfico de manera eficiente es ideal para proyectos IoT donde múltiples dispositivos están enviando datos simultáneamente. En este contexto, utilizar **Visual Studio Code (VSCode)** para el desarrollo del proyecto ofrece una experiencia de desarrollo ágil, con herramientas integradas para trabajar con Nginx.

Optimización I: Infraestructura

- **¿Por qué usar Nginx?**

1. **Balanceo de Carga y Proxy Inverso**: Los proyectos IoT pueden incluir varios servidores backend, como bases de datos y APIs, que procesan los datos que envían los dispositivos. Nginx actúa como proxy inverso, gestionando y distribuyendo estas solicitudes entre varios servidores backend.
2. **Gestión de Concurrencia**: En un sistema IoT, múltiples dispositivos pueden enviar datos a intervalos regulares. Nginx maneja eficientemente conexiones simultáneas, lo que mejora la escalabilidad del sistema.
3. **Seguridad**: Nginx proporciona una capa de seguridad al funcionar como proxy inverso, protegiendo los servidores backend y gestionando certificados SSL/TLS para las conexiones seguras con dispositivos IoT.
4. **Caché de Respuesta**: Si el sistema IoT requiere respuestas repetidas de los servidores backend (por ejemplo, actualizaciones de firmware o configuraciones), Nginx puede implementar una capa de caché para reducir la carga en el backend.

Optimización I: Infraestructura

- **¿Entonces, para qué necesitas Nginx?**

- **Para gestionar eficientemente las solicitudes HTTP/HTTPS** y no sobrecargar los servicios backend.
- **Para mejorar la seguridad y disponibilidad**, protegiendo los servidores backend y gestionando conexiones seguras.
- **Para actuar como intermediario** entre los usuarios (o dispositivos IoT) y los servicios internos, haciendo que tu infraestructura sea más modular y escalable.
- **Para balancear la carga** cuando tienes múltiples instancias de tus servicios backend, distribuyendo el tráfico de manera uniforme.
- En resumen, aunque tu servidor físico o virtual puede ejecutar los programas, **Nginx agrega una capa esencial de optimización, escalabilidad y seguridad** que te permite manejar de manera efectiva grandes cantidades de tráfico en entornos complejos como los proyectos IoT o aplicaciones web.

Optimización I: Infraestructura

¡Muchas gracias!