# 

# Uso de Docker

# 

#### Contenedores con Docker

#### 

#### **Juan Miguel Sarria Orozco**

1. **Parte Inicial**
2. **Construcción de nuestra API REST**

**Código:**

Antes de crear el contenedor Docker, subirlo a Docker Hub, conectarlo a un Redis subido a Docker Hub, etc., vamos a implementar nuestra aplicación en Python utilizando Flask para que sea una página web funcional. Aparte de mostrar la web, crearemos tres funciones básicas:

* Insertar datos
* Listar datos
* Borrar lista

El código de la aplicación será el siguiente:

# -\*- coding: utf-8 -\*-

from flask import Flask, request,  render\_template\_string

from redis import Redis, RedisError

import os

import socket

# Connect to Redis

REDIS\_HOST = os.getenv('REDIS\_HOST', "localhost")

print("REDIS\_HOST: "+REDIS\_HOST)

redis = Redis(host=REDIS\_HOST, db=0, socket\_connect\_timeout=2, socket\_timeout=2)

app = Flask(\_\_name\_\_)

HTML\_TEMPLATE = """

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Redis App</title>

</head>

<body>

    <h1><b>Mediciones de Juan Miguel Sarria Orozco</b></h1>

    <h2>Servidor: {{ hostname }}</h2>

    <form action="/nuevo" method="get" style="margin-bottom: 10px;">

        <input type="text" name="dato" placeholder="Introduce un número" required>

        <button type="submit">Insertar</button>

    </form>

    <form action="/limpiar" method="post">

        <button type="submit">Limpiar</button>

    </form>

    <h2>Lista de mediciones guardadas:</h2>

    <ul>

        {% for numero, index in numeros %}

            <li>Medición {{ index }}: {{ numero }}</li>

        {% endfor %}

    </ul>

</body>

</html>

"""

# Función para renderizar la lista desde Redis

def obtener\_lista():

    try:

        mediciones = redis.lrange("mediciones", 0, -1)

        numeros = [(m.decode("utf-8"), idx + 1) for idx, m in enumerate(mediciones)]  # Incluye el índice (1-based)

        return numeros

    except RedisError:

        return [("Error al conectar con Redis", 0)]

@app.route("/", methods=["GET", "POST"])

@app.route("/listar", methods=["GET", "POST"])

def listar():

    # Renderizar la lista con el formulario

    numeros = obtener\_lista()

    hostname = socket.gethostname()  # Obtener el nombre del servidor

    return render\_template\_string(HTML\_TEMPLATE, numeros=numeros, hostname=hostname)

#/nuevo?dato=valor para introducir datos manualmente

@app.route("/nuevo", methods=["GET"])

def nuevo():

    dato = request.args.get("dato")

    if not dato:

        return "Por favor, introduce un dato válido", 400

    # Validar que el dato sea un número (incluyendo decimales)

    try:

        float(dato)  # Intentar convertir el dato a número

        redis.rpush("mediciones", dato)  # Almacenar el dato en Redis

        return "<script>window.location='/';</script>"

    except ValueError:

        return "Solo se permiten números (enteros o decimales). Intenta nuevamente.", 400

    except RedisError as e:

        return f"Error al conectar con Redis: {str(e)}", 500

@app.route("/limpiar", methods=["POST"])

def limpiar():

    try:

        redis.delete("mediciones")  # Vaciar la lista en Redis

    except RedisError:

        return "Error al conectar con Redis", 500

    return "<script>window.location='/';</script>"

if \_\_name\_\_ == "\_\_main\_\_":

    PORT = os.getenv('PORT', 80)

    print("PORT: " + str(PORT))

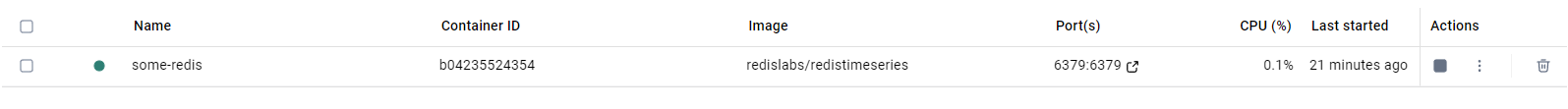
    app.run(host='0.0.0.0', port=PORT)

**Ejecución en local:**

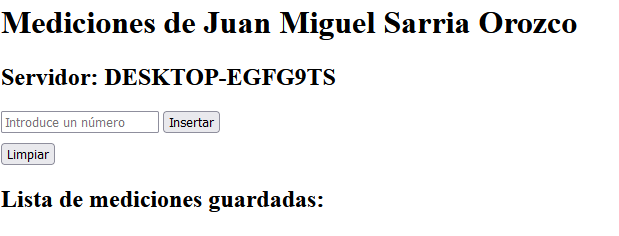
Para ejecutar esta aplicación en local, necesitamos tener Redis en funcionamiento. Esto se puede hacer ejecutando un RedisTimeSeries en un contenedor Docker.

Para ello, ejecutamos el siguiente comando en la terminal de Windows (CMD) para iniciar Redis:  
***docker run --name some-redis -p 6379:6379 -d redislabs/redistimeseries***

Este comando descargará y ejecutará el contenedor de Redis con el plugin RedisTimeSeries. Puedes comprobar que Redis está en ejecución abriendo Docker Desktop y verificando que el servicio Redis está activo.



**Acceso a la aplicación:**

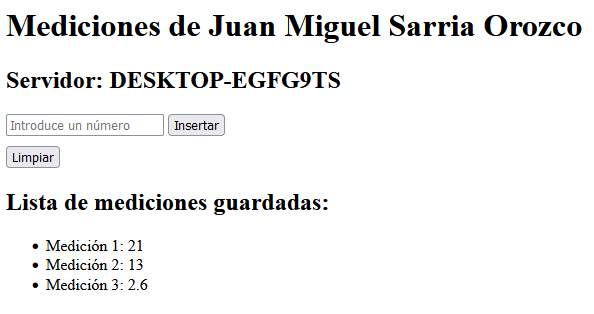
Una vez que el contenedor de Redis esté en funcionamiento, podemos iniciar nuestra aplicación Flask ejecutando el código Python. Después, abrimos nuestro navegador web y accedemos a la dirección: http://127.0.0.1

Al ingresar esta URL en el navegador, se mostrará la aplicación funcionando correctamente.

**Añadir Mediciones:**

Podemos añadir nuevas mediciones a través de dos métodos:

1. Introduciendo directamente un valor en la URL de la siguiente manera: **http://127.0.0.1/nuevo?dato=21**
2. Usando el formulario de la página web, donde podemos escribir un número en el campo y hacer clic en el botón "Insertar".



1. **Crear nuestro contenedor**

**DockerFile:**

Dado que nuestra aplicación se llama app.py, está escrita en Python y va a estar expuesta por el puerto 80 (HTTP), nuestro archivo Dockerfile será el siguiente:

# Use an official Python runtime as a parent image

FROM python:3.11-slim

# Set the working directory to /app

WORKDIR /app

# Copy the current directory contents into the container at /app

ADD . /app

# Install any needed packages specified in requirements.txt

RUN pip install --trusted-host pypi.python.org -r requirements.txt

# Make port 80 available to the world outside this container

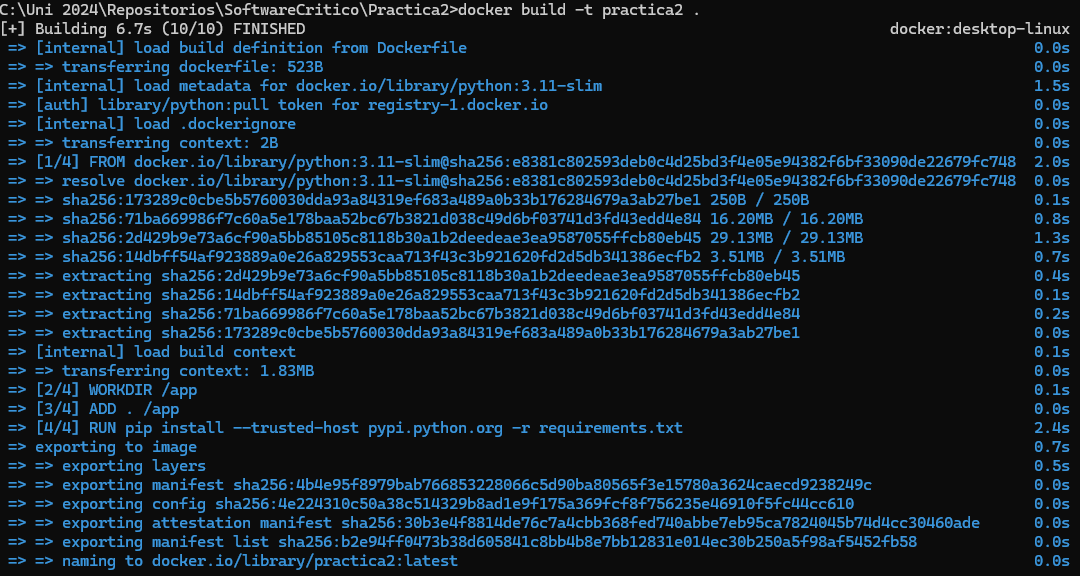
EXPOSE 80

# Run app.py when the container launches

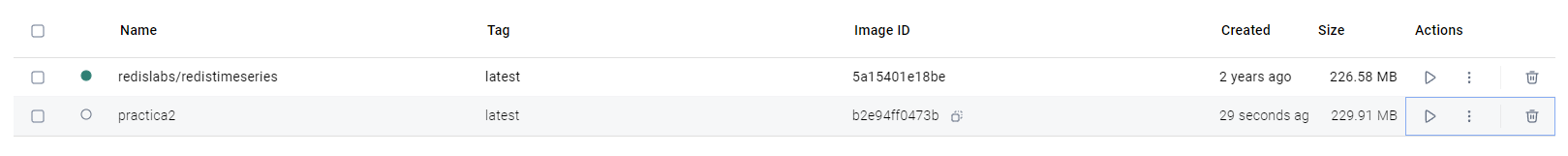
CMD ["python", "app.py"]

Este archivo Dockerfile es bastante similar al ejemplo de la práctica. A continuación, para crear nuestro contenedor, ejecutaremos el siguiente comando en el CMD de Windows:

**docker build -t practica2 .**



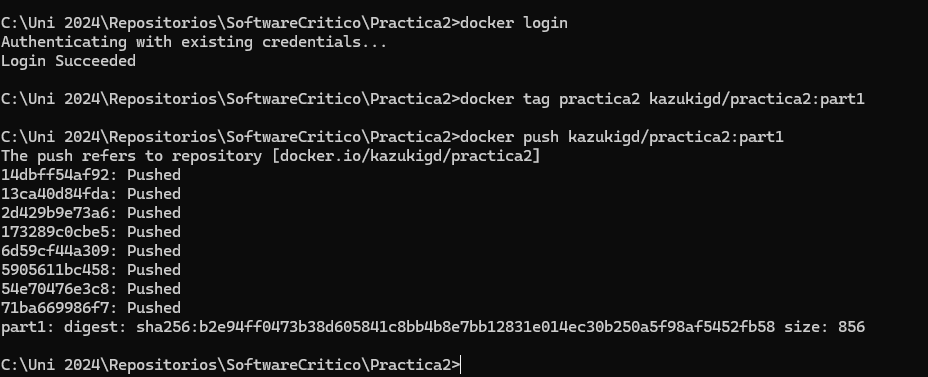
Una vez que el contenedor haya sido creado, podremos verlo en la sección de Imágenes de Docker Desktop, junto al contenedor de Redis que creamos anteriormente.

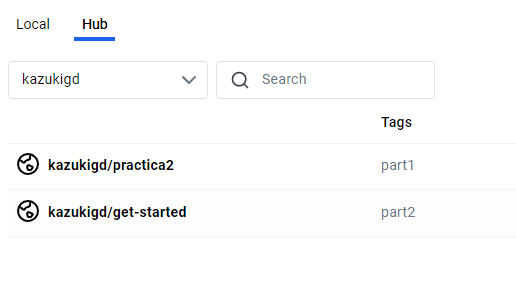


**Subir el contenedor a DockerHub:**

Procedemos a realizar el push del contenedor a DockerHub para tenerlo disponible en nuestro repositorio. El proceso será el siguiente:

* Inicia sesión en DockerHub con el comando docker login.
* Etiqueta la imagen para subirla a DockerHub:
* Realizamos el push a DockerHub:



Una vez subido, podremos ver la imagen en nuestro repositorio de DockerHub.  


**Crear el archivo docker-compose.yml**El siguiente paso es crear un archivo **docker-compose.yml** para gestionar todos nuestros servicios. El contenido de este archivo será el siguiente:

version: "3"

services:

  web:

    # replace username/repo:tag with your name and image details

    image: kazukigd/practica2:part1

    deploy:

      replicas: 5

      restart\_policy:

        condition: on-failure

    ports:

      - "4000:80"

    environment:

      - REDIS\_HOST=redis

    networks:

      - webnet

  visualizer:

    image: dockersamples/visualizer:stable

    ports:

      - "8080:8080"

    volumes:

      - "/var/run/docker.sock:/var/run/docker.sock"

    deploy:

      placement:

        constraints: [node.role == manager]

    networks:

      - webnet

  redis:

    image: redis

    ports:

      - "6379:6379"

    deploy:

      placement:

        constraints: [node.role == manager]

    networks:

      - webnet

  grafana:

    image: grafana/grafana

    ports:

      - 3000:3000

    volumes:

      - grafana\_data:/var/lib/grafana

    depends\_on:

      - redis

    environment:

      GF\_INSTALL\_PLUGINS: redis-datasource

    networks:

      - webnet

volumes:

  grafana\_data:

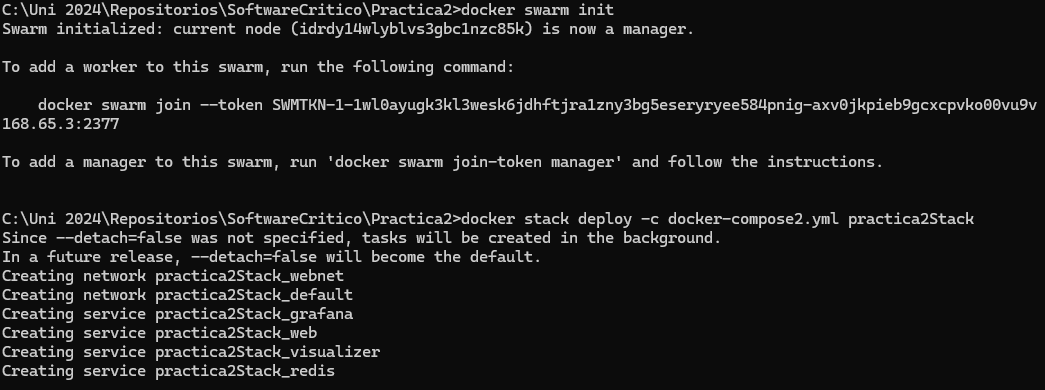
networks:

  webnet:

En este archivo docker-compose.yml, hemos incluido nuestra imagen de DockerHub **kazukigd/practica2:part1**, y hemos creado **5 réplicas** del servicio web, tal como lo requiere la práctica. También hemos añadido los servicios **Visualizer, Redis y Grafana**, exponiendo sus puertos correspondientes.

**Desplegar el stack**

Para desplegar nuestro stack de Docker, primero iniciaremos con el comando docker swarm init y después utilizaremos el comando **docker stack deploy -c docker-compose.yml practica2Stack**



Esto iniciará todos los servicios definidos en el archivo docker-compose.yml. Podemos comprobar que el stack se ha desplegado correctamente ejecutando el siguiente comando: **docker stack ls**



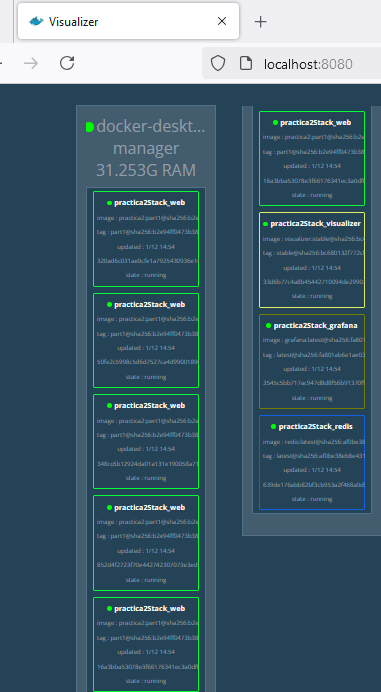
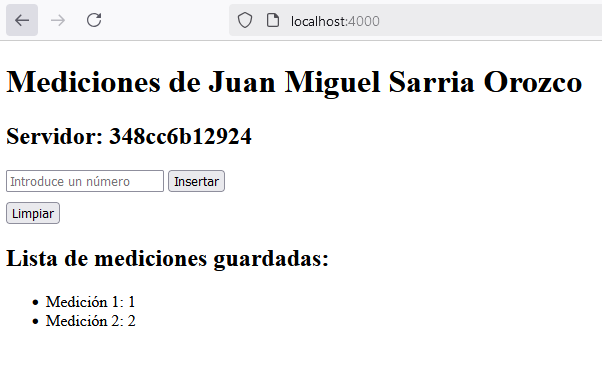
Esto mostrará el estado de nuestra stack, y podremos verificar si todos los servicios están funcionando correctamente.

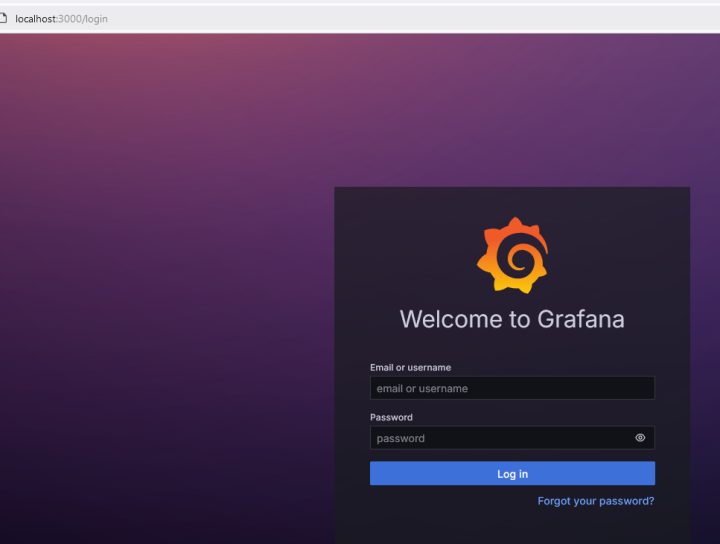
**Acceder a los servicios**

Una vez que el stack esté corriendo, podemos acceder a los diferentes servicios desde el navegador:

* Aplicación web: accediendo a **http://localhost:4000/**
* Visualizer: accediendo a **http://localhost:8080/**
* Grafana: accediendo a **http://localhost:3000/**

Para Grafana, la contraseña predeterminada es admin, y una vez dentro, podremos configurar Grafana para visualizar los datos almacenados en Redis.

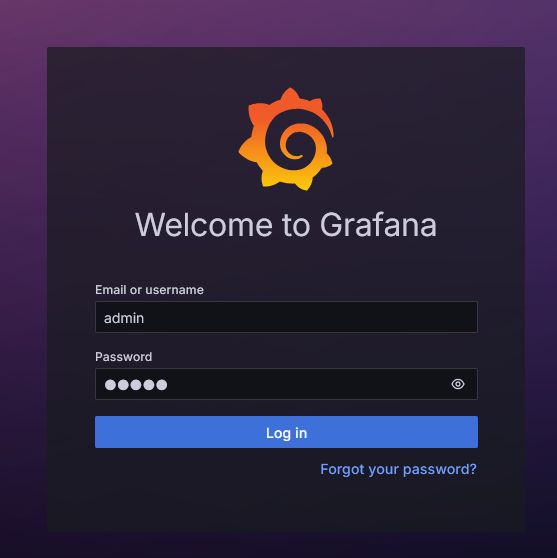


1. **Grafana**

Una vez que todo el entorno de Docker esté funcionando y nuestra aplicación esté conectada a Grafana y Redis, podemos utilizar Grafana para visualizar y comprobar los datos almacenados.

**Acceso a Grafana:**

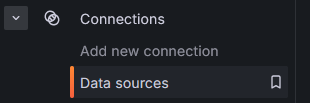
Primero, accedemos a Grafana con las credenciales predeterminadas: admin/admin.

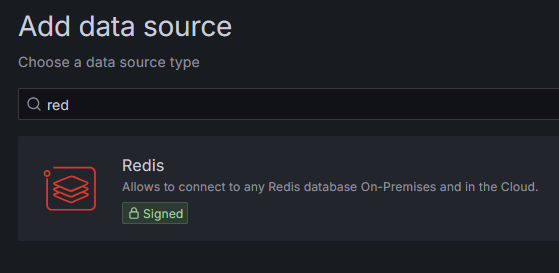
****

**Conectar Grafana a Redis:**

Después de iniciar sesión en Grafana, vamos a la sección de "Connections" y seleccionamos "Data Sources" para configurar nuestra fuente de datos de Redis.

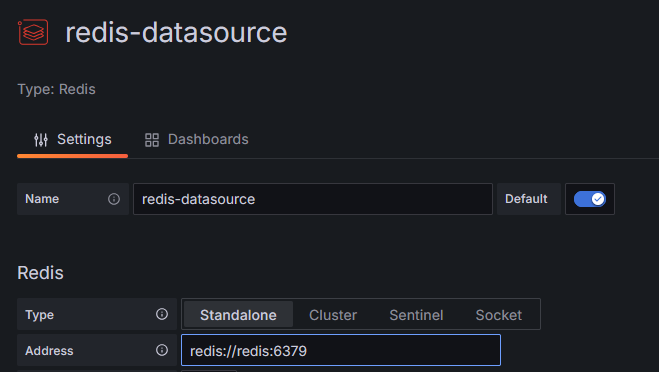
Elegimos Redis como tipo de fuente de datos.





**Configuración de Redis en Grafana:**

En la configuración de Redis, asignamos el puerto 6379, que es el puerto que hemos configurado previamente en Docker para el servicio Redis.





**Problema con las series temporales:**

Al intentar configurar el Dashboard, nos dimos cuenta de que nuestra instancia de Redis no soporta series temporales, lo cual es necesario para trabajar con datos que incluyen marcas de tiempo (como las mediciones).

Además, nuestro código actual no está diseñado para almacenar datos como series temporales.

Para corregir este problema, hemos realizado un cambio en nuestro código, específicamente en la parte donde almacenamos los datos. Aquí es donde agregamos la funcionalidad de RedisTimeSeries para que los datos se almacenen como series temporales.

Hemos tenido que sustituir la línea para insertar datos:

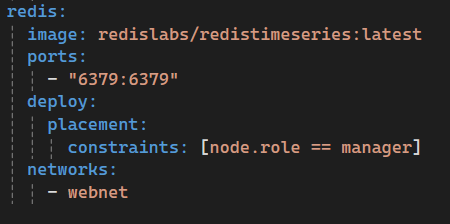
**redis.rpush("mediciones", dato)**

Por la siguiente línea;

**redis.execute\_command('TS.ADD', 'mediciones', '\*', dato)**

**Cambio en el docker-compose:**

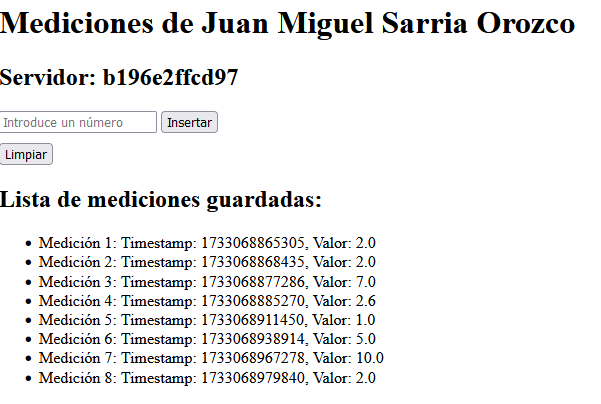
Además, hemos actualizado el archivo docker-compose.yml para utilizar una imagen de Redis que soporte el módulo RedisTimeSeries. La imagen que hemos utilizado es redislabs/redistimeseries, que incluye el módulo de series temporales.



**Resultados después de los cambios:**

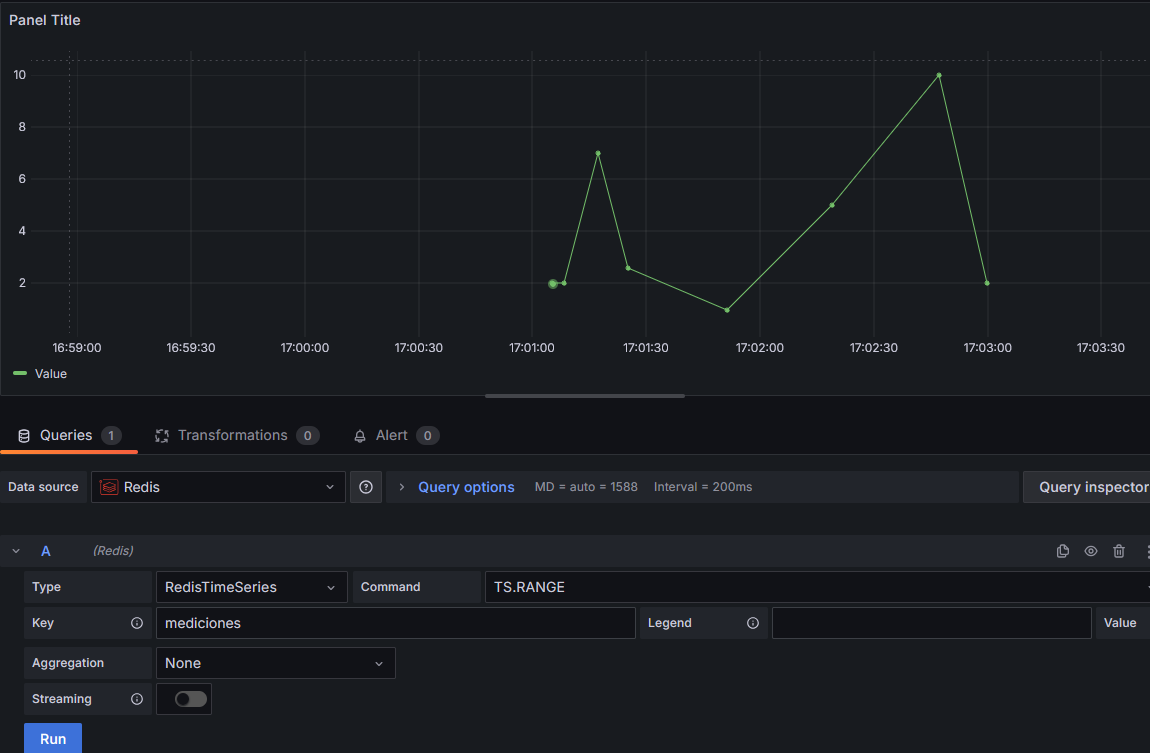
Con estas modificaciones, ahora nuestro programa almacena las mediciones en un formato de serie temporal, lo que nos permite realizar consultas y visualizaciones más precisas en Grafana.

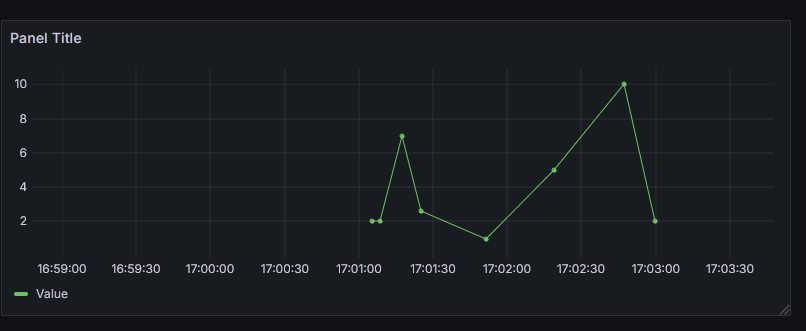
El formato de los datos ahora incluye la marca de tiempo junto con la medición, como se muestra a continuación:



**Configuración del Dashboard en Grafana:**

Finalmente, una vez que los datos están siendo almacenados correctamente como series temporales, podemos configurar el dashboard en Grafana para visualizarlos. Grafana nos permite crear gráficos en los datos de Redis, que ahora están en formato de series temporales y listos para ser visualizados.





1. **Detección de anomalías**

Para detectar anomalías en nuestro modelo, primero guardamos el modelo entrenado en la **Práctica 1** como “modelo.keras”, y el valor del umbral en un archivo llamado “threshold.txt”. A continuación, creamos la función detectar() de la siguiente manera:

@app.route("/detectar", methods=["GET"])

def detectar():

    # Cargar el dato

    dato = request.args.get("dato")

    if not dato:

        return "Por favor, introduce un dato válido", 400

    try:

        dato\_float = float(dato)

        # Cargar el modelo

        modelo = tf.keras.models.load\_model('modelo.keras')

        # Cargar el umbral

        with open("threshold.txt", "r") as file:

            threshold = float(file.read().strip())

        # Agregar el nuevo dato a Redis

        redis.execute\_command('TS.ADD', 'mediciones', '\*', dato)

        # Obtener las mediciones

        mediciones = redis.execute\_command('TS.RANGE', 'mediciones', '-', '+')

        if len(mediciones) <= 3:

            # Si hay menos de 3 mediciones, registrar siempre como no anomalía

            pos = len(mediciones);

            resultado = {

             f"medicion {pos}": {

             "time": int(mediciones[pos-1][0]),  # timestamp

                "valor": float(mediciones[pos-1][1]),  # valor de la medición

               "anomalia": "no"

            }

         }

        else:

            # Tomar las últimas 3 mediciones

           ventana = mediciones[-4:-1]

            valores\_ventana = [float(m[1]) for m in ventana]

            valores\_ventana = np.array(valores\_ventana).reshape((1, 3, 1))

            #Realizar predicción con el modelo

            prediccion = modelo.predict(valores\_ventana)[0][0]

            # Calcular la diferencia y verificar anomalía

            error = abs(prediccion - dato\_float)

            anomalia = "si" if error > threshold else "no"

            # Preparar el resultado

            pos = len(mediciones);

resultado = {

              f"medicion {pos}": {  # El número de la medición será el tamaño de la lista

                "time": int(mediciones[pos-1][0]),  # timestamp de la última medición

                  "valor": float(mediciones[pos-1][1]),  # valor de la última medición

                  "prediccion": float(prediccion),

                  "anomalia": anomalia  # Anomalía calculada

                }

              }

        # Escribir el resultado en el archivo, añadiendo una línea

        with open("resultados.txt", "a") as file:

            file.write(json.dumps(resultado, ensure\_ascii=False) + "\n")

        # Devolver el resultado como respuesta

        return "<script>window.location='/';</script>"

    except ValueError:

        return "Solo se permiten números válidos. Intenta nuevamente.", 400

    except FileNotFoundError as e:

        return f"No se encontró el archivo necesario: {str(e)}", 500

    except RedisError as e:

        return f"Error al conectar con Redis: {str(e)}", 500

**Explicación de la detección de anomalías:**

Con el fin de detectar anomalías en tiempo real, hemos ampliado la funcionalidad de la aplicación con la función detectar() para realizar los siguientes pasos:

1. **Predicción con el modelo entrenado**:

Usamos un modelo **LSTM** previamente entrenado (guardado como modelo.keras) que se alimenta de las últimas tres mediciones almacenadas en Redis. Si aún no existen tres mediciones, se asumirá que la primera medición no es una anomalía.

            ventana = mediciones[-3:]

            valores\_ventana = [float(m[1]) for m in ventana]

            valores\_ventana = np.array(valores\_ventana).reshape((1, 3, 1))

            #Realizar predicción con el modelo

            prediccion = modelo.predict(valores\_ventana)[0][0]

1. **Cálculo del error**:

Para cada nueva medición, el modelo genera una predicción. Calculamos la diferencia entre la predicción del modelo y el valor del dato recibido. Si el valor absoluto de esta diferencia es mayor que un umbral previamente definido, consideramos que el dato es una **anomalía**.

            error = abs(prediccion - dato\_float)

            anomalia = "si" if error > threshold else "no"

1. **Registro de resultados:**

El resultado de la predicción, junto con el valor real, la predicción y si es o no una anomalía, se guarda en un archivo de texto (resultados.txt). Este archivo almacena los resultados, permitiendo su posterior análisis.

resultado = {

              f"medicion {pos}": {

                "time": int(mediciones[pos-1][0]),

"valor": float(mediciones[pos-1][1]),

                  "prediccion": float(prediccion),

                  "anomalia": anomalia

                }

              }

        with open("resultados.txt", "a") as file:

            file.write(json.dumps(resultado, ensure\_ascii=False) + "\n")

**Detalles del proceso:**

**Ventana de 3 mediciones**: La función utiliza una ventana de 3 mediciones consecutivas para realizar la predicción. Si aún no hay 3 mediciones, el dato que llega se marca como no anómalo por defecto.

**Formato de entrada para el modelo**: Las tres mediciones más recientes se convierten a formato **NumPy** para ser compatibles con la entrada del modelo. Luego, el modelo realiza la predicción sobre estos valores.

**Anomalía**: Si la diferencia entre el dato y la predicción es mayor que el umbral predefinido, el dato se marca como una anomalía. Esta verificación permite identificar posibles datos atípicos de manera eficiente.

**Pruebas utilizando docker compose:**

En este proceso, vamos a actualizar nuestra práctica en Docker Hub (part2) y vamos a volver a subir nuestra práctica a docker hub para hacer la prueba y probar todo el docker-compose de nuevo para ver cono funciona con la detección de anomalía:

**Actualizar el archivo requirements.txt:** Para crear el nuevo contenedor, es necesario actualizar el archivo requirements.txt con las siguientes dependencias, ya que ahora estamos utilizando las bibliotecas de TensorFlow y NumPy::

Flask

Redis

tensorflow

numpy

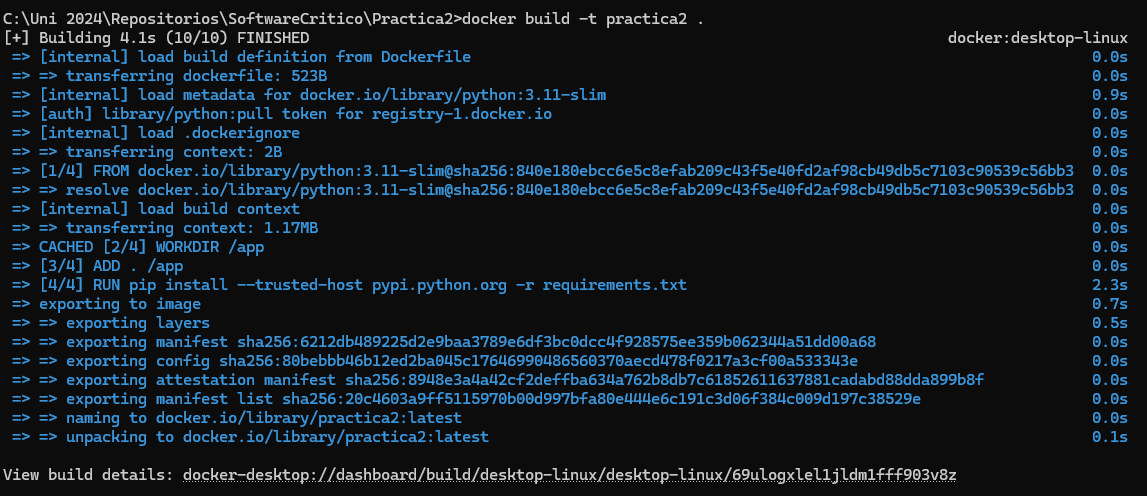
**Crear un volumen compartido entre réplicas**: Durante las pruebas, descubrimos que los archivos no se sincronizan entre las réplicas, lo que significa que necesitamos configurar un volumen compartido en Docker. Para ello, vamos a modificar el archivo docker-compose.yml e incluir las siguientes líneas dentro del servicio web::

    volumes:

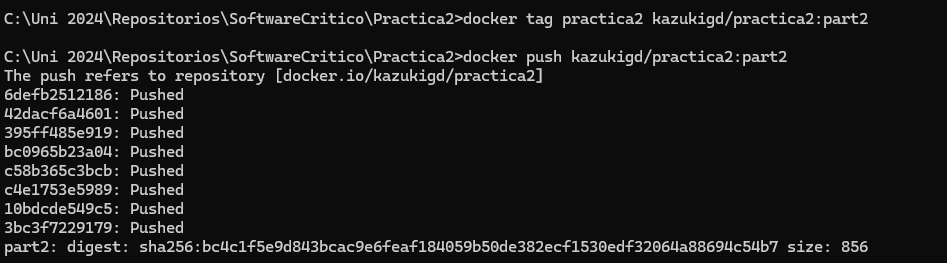
      - shared-data:/app  # Volumen compartido entre las replicas

Esto permitirá que las réplicas compartan los datos generados (resultados.txt), como los archivos de anomalías.

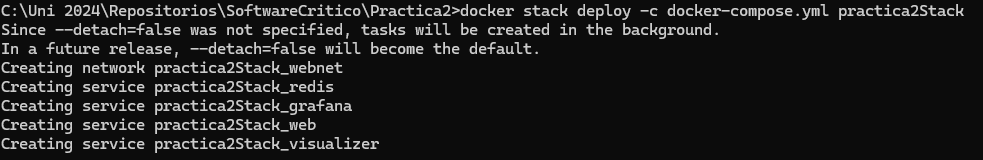
**Recrear el contenedor con los cambios**: Con los cambios realizados, vamos a reconstruir nuestro contenedor para aplicar las actualizaciones:



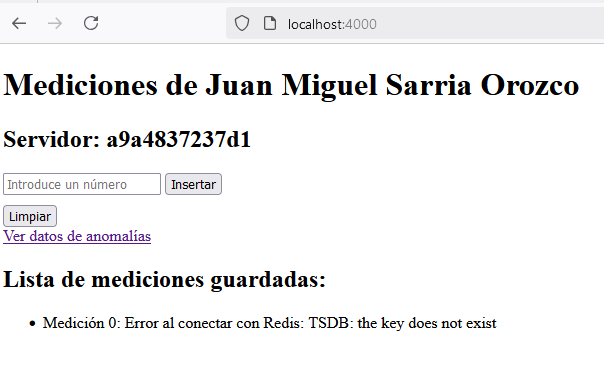
**Subir el contenedor actualizado a Docker Hub:** Después de reconstruir el contenedor, lo subimos nuevamente a Docker Hub para asegurarnos de que todos los cambios sean accesibles desde cualquier entorno



**Actualizar y desplegar el docker-compose.yml**: Ahora que nuestro contenedor está actualizado y disponible en Docker Hub, volvemos a crear el archivo docker-compose.yml para configurar correctamente los servicios y contenedores. Esto incluye el volumen compartido entre las réplicas y la configuración de las dependencias:



**Verificación en el navegador:** Al acceder a localhost:4000, podremos ver el nuevo formato de la aplicación, que ahora incluye un botón llamado "Ver datos de anomalías".



**Prueba de la detección de anomalías:** Para probar el sistema, incluimos datos que fuerzan anomalías. Al hacer clic en el botón "Ver anomalías", podremos observar cómo se descarga un archivo de texto con las anomalías pertinentes:

