Desarrollo de Software Crítico
Práctica 2

Despliegue de un Sistema de Monitorización Tolerante a Fallos con Docker Swarm, Redis y Grafana

Daniil Nemchenko
3ºA Ingeniería del Software

Parte 1: Contenedor Docker y Stack con Grafana

1 Desarrollo de API

En primer lugar, se ha desarrollado una API REST en Python utilizando el framework Flask. Esta aplicación funciona como intermediaria para gestionar los datos en Redis (usando el módulo RedisTimeSeries).

Las funcionalidades principales que se han implementado son:

- **Registro de datos (`/nuevo?dato=VALOR`):** Recibe un valor numérico y lo almacena en la base de datos con su marca de tiempo.
- **Consulta (`/listar`):** Recupera las mediciones almacenadas y devuelve también el `hostname` del contenedor (socket.gethostname()) que nos ayudará a verificar que el tráfico se está repartiendo correctamente entre las réplicas del sistema.

Detalle relevante de la implementación:

Se ha utilizado os.getenv('REDIS_HOST', 'localhost') para definir el host de la base de datos. Esto permite que el mismo código funcione en local y dentro del Swarm. También se ha implementado una función auxiliar (convert_timestamp(timestamp)) para convertir los timestamps a fechas legibles.

2. Contenerización de API

Una vez terminada y probada la API en local, se ha procedido a su contenerización. A continuación, se muestra el Dockerfile diseñado para construir la imagen y subirla a HubDocker.

Dockerfile > ...

```
# 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

# Define environment variable
ENV NAME=API-Measurements

# Run app.py when the container launches
CMD ["python", "src/app.py"]
```

Subimos la imagen a HubDocker:

```
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2> docker tag api-measurements danielnem/api-measurements:v2.0
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2> docker push danielnem/api-measurements:v2.0
The push refers to repository [docker.io/danielnem/api-measurements]
75e545ab592d: Layer already exists
c93582c38de6: Already exists
b811cf73f1b5: Layer already exists
22b63e76fde1: Layer already exists
b3dd773c3296: Layer already exists
1771569cc129: Layer already exists
591214a5418f: Layer already exists
```

3. Despliegue en Docker Swarm (El Stack)

Después de haber probado la API en un contenedor junto a una instancia de Redis separada, creamos un Docker Swarm usando la imagen subida a Docker Hub, e implementamos herramientas auxiliares como Grafana, Redis y Visualizer. Esto se configuró con el siguiente archivo docker-compose.yml

```yaml
version: "3"
  Run All Services
services:
  Run Service
  web-api:
    image: danielnem/api-measurements:v2.0
    deploy:
      replicas: 5
      restart_policy:
        condition: any # not just 'on-failure', so new replicas are started if one is stopped manually
    ports:
      - "4000:80"
    environment:
      - REDIS_HOST=redis
    networks:
      - webnet
  Run Service
  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
  Run Service
  redis:
    image: redis/redis-stack:latest
    ports:
      - "6379:6379"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
  Run Service
  grafana:
    image: grafana/grafana
    ports:
      - 3000:3000
    volumes:
      - grafana_data:/var/lib/grafana
    depends_on:
      - redis
    networks:
      - webnet
    environment:
      GF_INSTALL_PLUGINS: redis-datasource
volumes:
  grafana_data:
networks:
  webnet:
```

restart_policy: condition: any para que se inicie un nuevo contenedor no solo si uno cae, sino también en el caso de que se apague manualmente.

Creación y despliegue de docker swarm:

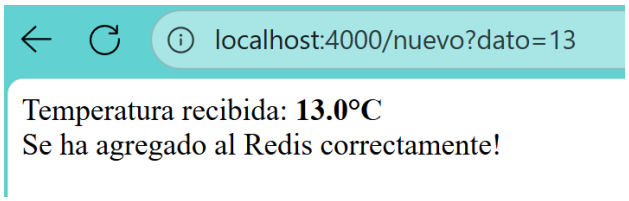
```
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\docker-swarm> docker stack deploy -c docker-compose2.yml api-swarm
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating network api-swarm_webnet
Creating service api-swarm_web-api
Creating service api-swarm_visualizer
Creating service api-swarm_redis
Creating service api-swarm_grafana
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\docker-swarm> docker service ls
ID             NAME                  MODE         REPLICAS   IMAGE                              PORTS
wqfm2ljsr4ph   api-swarm_grafana     replicated   1/1        grafana/grafana:latest             *:3000->3000/tcp
8dvqpmz36fku   api-swarm_redis       replicated   1/1        redis/redis-stack:latest           *:6379->6379/tcp
6pd4dhnxnkar   api-swarm_visualizer  replicated   1/1        dockersamples/visualizer:stable    *:8080->8080/tcp
jj2xcg4x5osj   api-swarm_web-api     replicated   5/5        danielnem/api-measurements:v2.0    *:4000->80/tcp
```

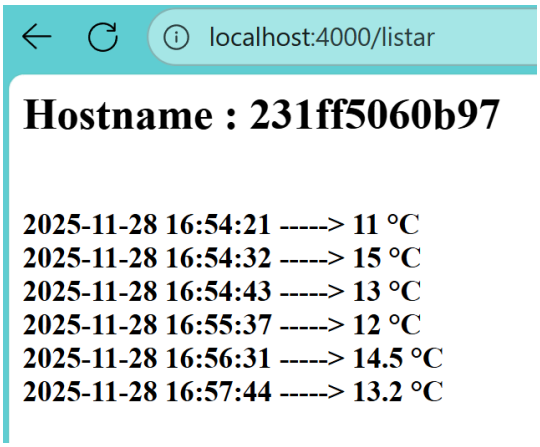
Comprobación del funcionamiento del sistema:

Accedemos al puerto 4000 para comprobar el funcionamiento de API

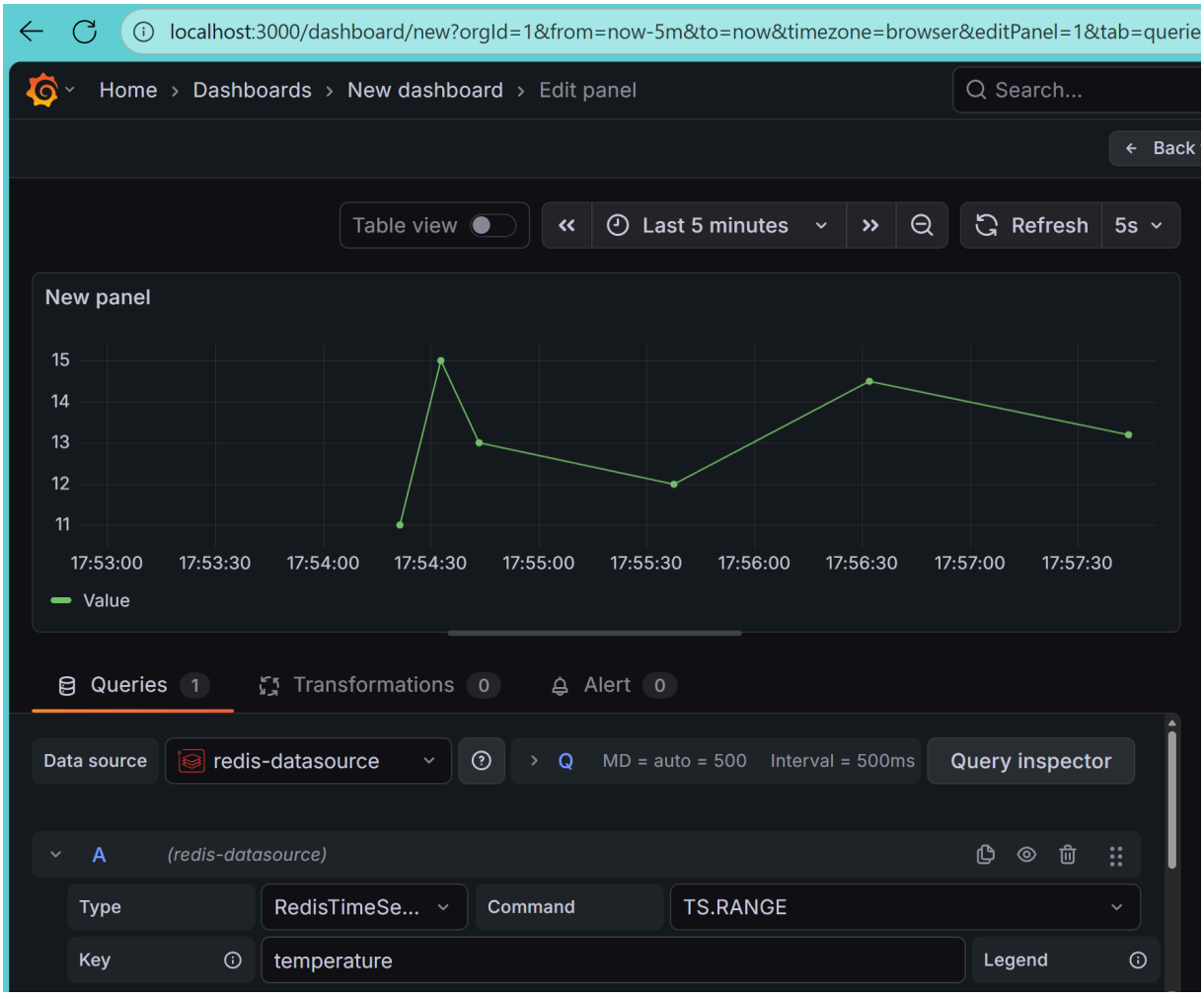
El hostname cambia de una visita a otra, lo que confirma que Docker Swarm está distribuyendo la carga de manera correcta entre las 5 réplicas de la API.

Comprobamos el funcionamiento de otras funciones:

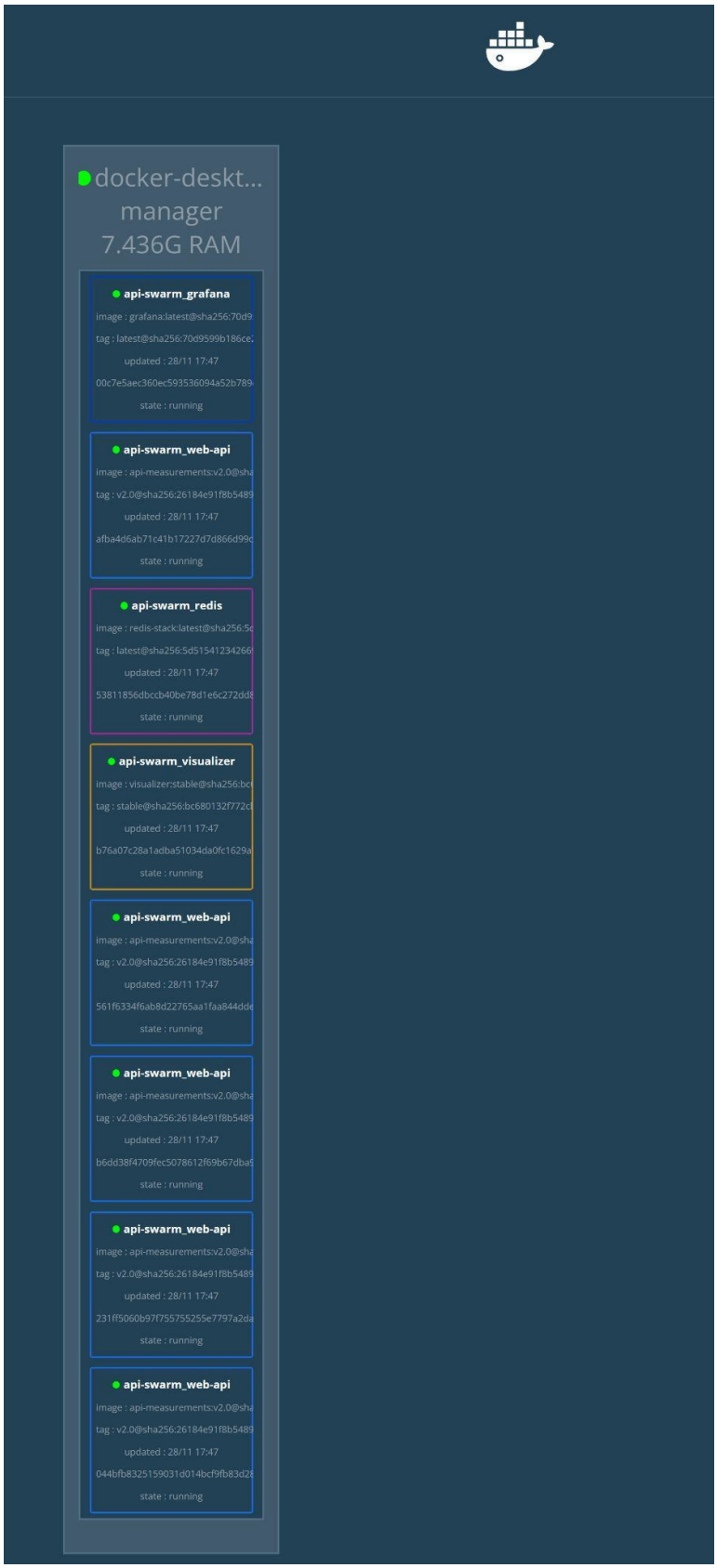




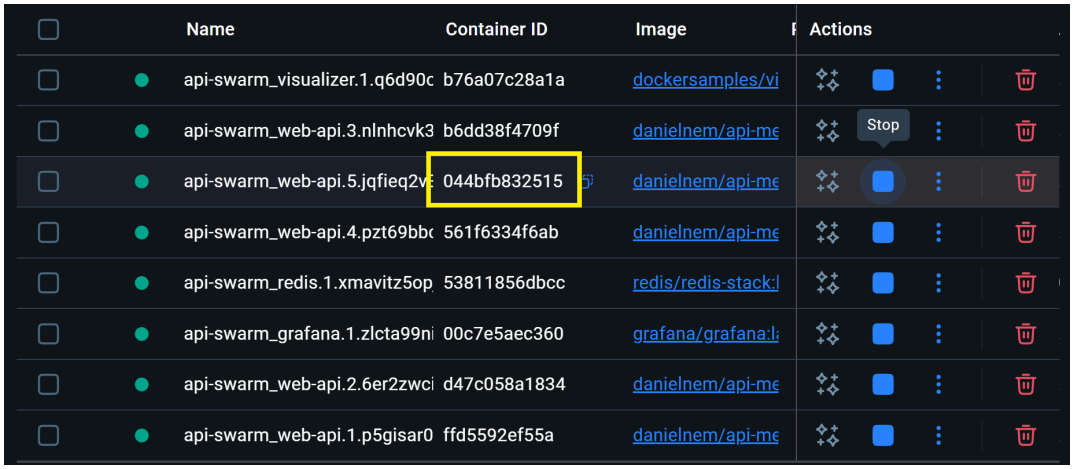
Comprobamos que Grafana se actualiza correctamente:

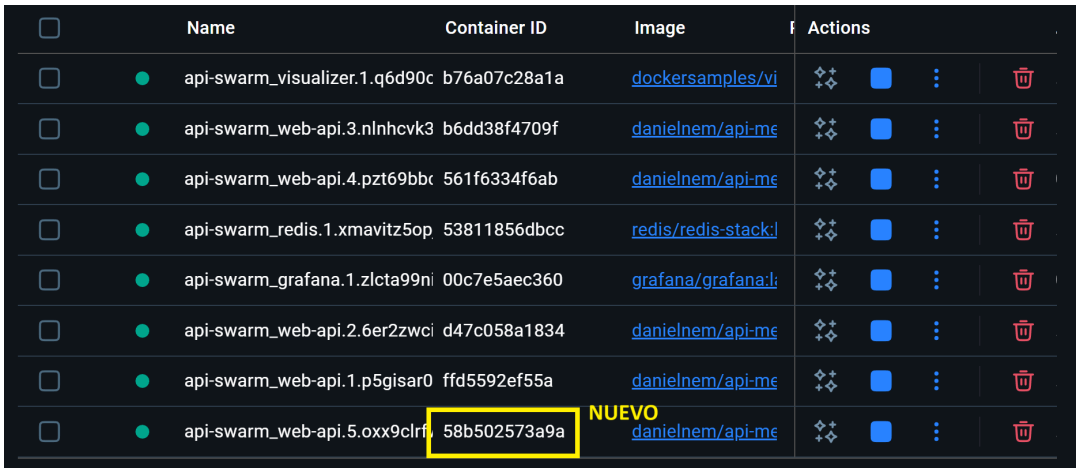


Comprobamos el estado de los contenedores en Visualiser:



Comprobamos que al parar una réplica del contenedor API se crea una nueva automáticamente por lo que siempre son 5:





Parte 2: Detección de Anomalía

1 Modificación de API

En primer lugar, modificamos el código de la práctica anterior para guardar el modelo y el umbral (threshold) para la detección de anomalías:

```
# fit model (Entrenar la RNN)
model_1.fit(X, y, epochs=10)
```
[9] ✓ 5.1s Python

Outputs are collapsed ···

```
model_1.save('lstm-model.keras')
```
[10] ✓ 0.0s Python

Para detectar las anomalías, primero calculamos el error absoluto entre los valores predichos y los valores reales. A continuación, consideramos como anomalías aquellos casos cuyo error supera el percentil 99 de la distribución de errores.

Se ha elegido este criterio porque permite identificar las anomalías de manera más precisa.

```
# Calculamos la diferencia absoluta entre los valores reales y las predicciones
absolute_errors = np.abs(y_test.to_numpy() - y_pred.flatten())

# Calculamos la media de los errores absolutos
mae = np.mean(absolute_errors)
print(f"Mean Absolute Error (MAE): {mae}")
```
[15] ✓ 0.0s Python

```
··· Mean Absolute Error (MAE): 0.7190428658274608
```

```
# Calulamos el valor umbral correspondiente al percentil 99
# Solo el 1% de los errores serán mayores que este valor y se considerarán anomalías
threshold = np.percentile(absolute_errors, 99)
print("Threshold: ", threshold)
```
[16] ✓ 0.0s Python

```
··· Threshold:  2.387054468521088
```

```
threshold_dict = {'threshold' : threshold}
# Guardamos threshold en un fichero json
with open('config-lstm-model.json', 'w') as file:
    json.dump(threshold_dict, file)
```
[19] ✓ 0.0s Python

Definimos también el tamaño de la ventana para el modelo en el config-lstm-model.json

```
{
"threshold": 2.387054468521088,
"window_size": 10
}
```

Así pues, el código del apartado anterior fue modificado para incluir la carga del modelo, del umbral (threshold) y del tamaño de ventana (window_size):

```
try:
    # Cargamos el modelo para deteccion de anomalías
    model = tf.keras.models.load_model('lstm-model.keras')
    # Cargamos el threshold desde un archivo para deteccion de anomalías
    with open('config-lstm-model.json', 'r') as config_file:
        config = json.load(config_file)
        threshold = config.get('threshold')
        window_size = config.get('window_size', 10)  # Tamaño de ventana por defecto 10
    print(f"Modelo y threshold cargados correctamente. Threshold: {threshold}")
except Exception as e:
    print(f"Error al cargar el modelo y threshold: {str(e)}")
```

Agregamos también un nuevo endpoin para detectar anomalías al código (@app.route('/detectar')).

2 Contenerización de nueva API

Después de haber probado la nueva API con detección de anomalías en local (junto a un contenedor de Redis aparte), conteneirizamos nuestra API con el siguiente Dockerfile:

```dockerfile
# 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

# Define environment variable
ENV NAME=API-Measurements-2

# Run app.py when the container launches
CMD ["python", "app2.py"]
```

Creamos una imagen:

```
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\anomalies-docker> docker build -t api-anomalies .
[+] Building 546.9s (10/10) FINISHED                                                                    docker:desktop-linux
 => [internal] load build definition from Dockerfile                                                                0.1s
 => => transferring dockerfile: 566B                                                                                0.0s
 => [internal] load metadata for docker.io/library/python:3.11-slim                                                 0.0s
 => [auth] library/python:pull token for registry-1.docker.io                                                       0.0s
 => [internal] load .dockerignore                                                                                   0.1s
 => => transferring context: 2B                                                                                     0.0s
 => [1/4] FROM docker.io/library/python:3.11-slim@sha256:193fdd0bbcb3d2ae612bd6cc3548d2f7c78d65b549fcaa8af75624c474744  0.0s
 => => resolve docker.io/library/python:3.11-slim@sha256:193fdd0bbcb3d2ae612bd6cc3548d2f7c78d65b549fcaa8af75624c474744  0.0s
 => [internal] load build context                                                                                   0.1s
 => => transferring context: 39.29kB                                                                                0.0s
 => CACHED [2/4] WORKDIR /app                                                                                       0.0s
 => [3/4] ADD . /app                                                                                                0.1s
 => [4/4] RUN pip install --trusted-host pypi.python.org -r requirements.txt                                      420.1s
 => exporting to image                                                                                            126.4s
 => => exporting layers                                                                                            97.1s
 => => exporting manifest sha256:7d637b859592a7ee766fa53ecd5269bddfd2f2e89b1a65eaab4a2a1478a325f3                    0.0s
 => => exporting config sha256:6d743ba017ffeb551cca43d85606b3a4fcb38ad89dde7db47c3897d0250c83f2                      0.0s
 => => exporting attestation manifest sha256:81be58e7a6752b4848072a3834442c85187885f40b6ab0f0c1a102a0227d4779        0.0s
 => => exporting manifest list sha256:7fb7858a23a446ff18014d73c5ec094fa273af5648a65f3cf14b98251cad2248              0.0s
 => => naming to docker.io/library/api-anomalies:latest                                                             0.0s
 => => unpacking to docker.io/library/api-anomalies:latest                                                         29.1s

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/jvjq06lkttm976zsscqlbknrx
```

Probamos que el contenedor creado con la nueva imagen funciona correctamente en local:

```
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\anomalies-docker> docker run -p 4001:80 -e REDIS_HOST=host.docker.i
nternal --name api-anomalies api-anomalies
2025-11-28 21:40:12.405338: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:31] Could not find cuda drivers on your machine,
 GPU will not be used.
2025-11-28 21:40:12.601001: I tensorflow/core/platform/cpu_feature_guard.cc:210] This TensorFlow binary is optimized to use a
vailable CPU instructions in performance-critical operations.
To enable the following instructions: AVX2 FMA, in other operations, rebuild TensorFlow with the appropriate compiler flags.
2025-11-28 21:40:15.981017: I external/local_xla/xla/tsl/cuda/cudart_stub.cc:31] Could not find cuda drivers on your machine,
 GPU will not be used.
2025-11-28 21:40:16.706712: E external/local_xla/xla/stream_executor/cuda/cuda_platform.cc:51] failed call to cuInit: INTERNA
L: CUDA error: Failed call to cuInit: UNKNOWN ERROR (303)
REDIS_HOST: host.docker.internal
Modelo y threshold cargados correctamente. Threshold: 2.387054468521088
PORT: 80
 * Serving Flask app 'app2'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:80
 * Running on http://172.17.0.3:80
Press CTRL+C to quit
```

Subimos la imagen creada a Docker Hub:

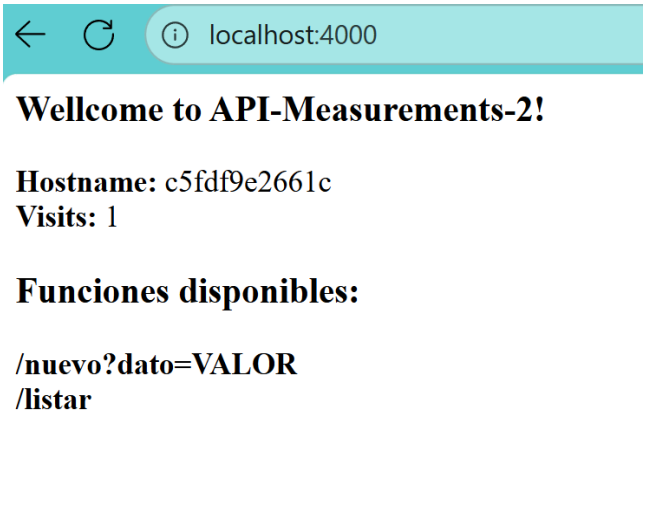
```
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\anomalies-docker> docker tag api-anomalies danielnem/api-anomalies:
v1.0
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\anomalies-docker> docker push danielnem/api-anomalies:v1.0
The push refers to repository [docker.io/danielnem/api-anomalies]
fd3998a53a38: Pushed
6a9d2b96921a: Pushed
1771569cc129: Mounted from danielnem/friendlyhello
22b63e76fde1: Mounted from danielnem/friendlyhello
591214a5418f: Mounted from danielnem/friendlyhello
b3dd773c3296: Mounted from danielnem/friendlyhello
0e4bc2bd6656: Mounted from danielnem/friendlyhello
cfbb943df131: Pushed
v1.0: digest: sha256:7fb7858a23a446ff18014d73c5ec094fa273af5648a65f3cf14b98251cad2248 size: 856
```

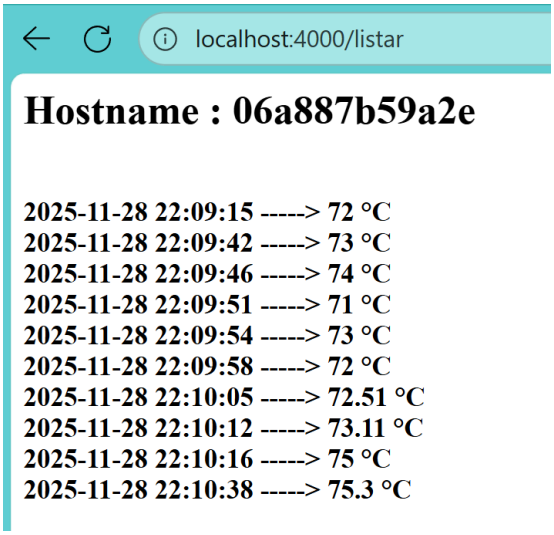
3. Despliegue en Docker Swarm (El Stack)

A continuación, se ha creado un Docker Swarm utilizando la nueva imagen con detección de anomalías (usando docker-compose-3.yml):
Finalmente, comprobamos que la API dentro del Swarm, así como los demás componentes, funciona correctamente:

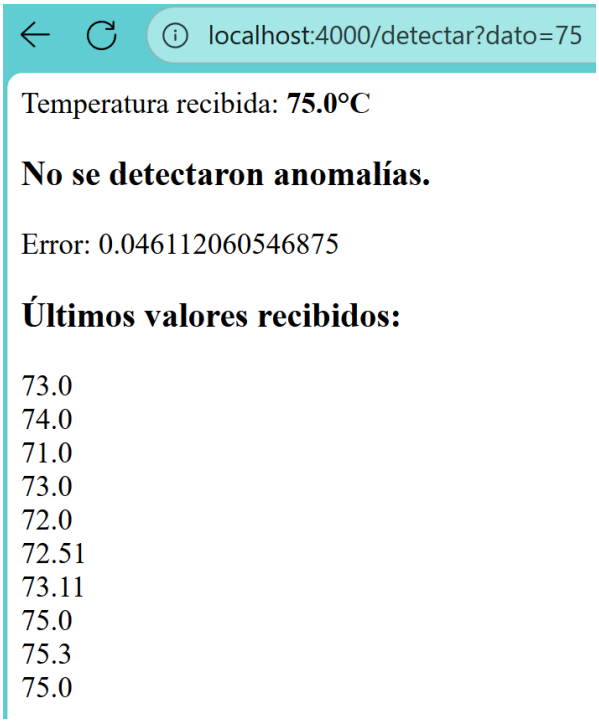
```
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\anomalies-docker> docker stack deploy -c docker-compose3.yml api-swarm-anomalies
Since --detach=false was not specified, tasks will be created in the background.
In a future release, --detach=false will become the default.
Creating network api-swarm-anomalies_webnet
Creating service api-swarm-anomalies_visualizer
Creating service api-swarm-anomalies_redis
Creating service api-swarm-anomalies_grafana
Creating service api-swarm-anomalies_web-api
(.venv) PS C:\Software\Python\Critical-Software\swc-lab-2\anomalies-docker> docker service ls
ID             NAME                              MODE         REPLICAS   IMAGE                                PORTS
q1ys7s7n6uo1   api-swarm-anomalies_grafana       replicated   1/1        grafana/grafana:latest               *:3000->3000/tcp
j0w30xxczl13   api-swarm-anomalies_redis         replicated   1/1        redis/redis-stack:latest             *:6379->6379/tcp
5jbd61ojkfxo   api-swarm-anomalies_visualizer    replicated   1/1        dockersamples/visualizer:stable      *:8080->8080/tcp
mqdcqja29vd2   api-swarm-anomalies_web-api       replicated   5/5        danielnem/api-anomalies:v1.0         *:4000->80/tcp
```

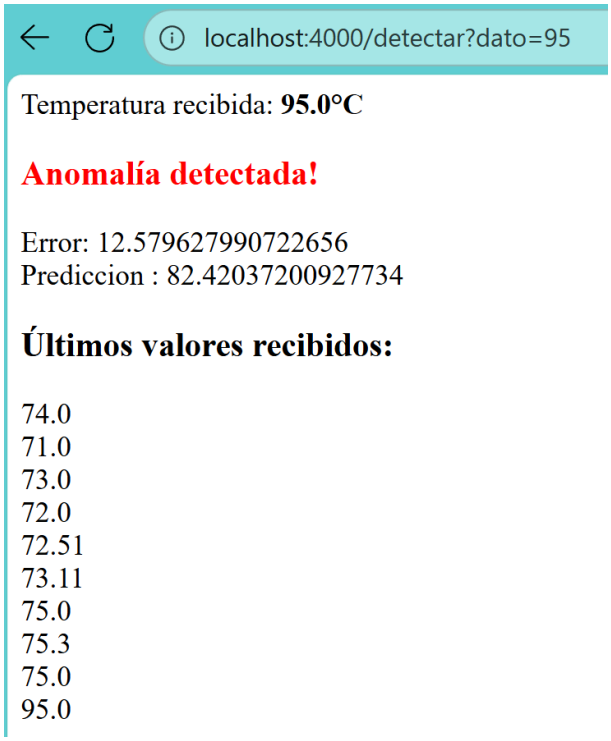
Finalmente, comprobamos que la API dentro del Swarm, así como los demás componentes, funciona correctamente.





Comprobamos nueva funcionalidad:





Las imágenes en Docker Hub:

Apartado 1: danielnem/api-measurements
Apartado 2: danielnem/api-anomalies