Monitorización de escenario con CMS

Gonzalo Marín Gómez

Proyecto Integrado

2º Administración de Sistemas Informáticos en Red

I.E.S. Gonzalo Nazareno

Índice de contenido

1. Objetivos que se quieren conseguir y se han conseguido.

Tras trabajar en múltiples entornos y con diferentes objetivos durante el ciclo formativo, he decicido enfocar el proyecto en la monitorización de un escenario con un CMS, es decir, recoger métricas de las máquinas, logs y más información relevante con el objetivo de tener controlado lo que pasa en el entorno, ya que durante mi etapa formativa he estado acostumbrado a crear escenarios, pero no he hecho mucho hincapié en el mantenimiento y el control del mismo.

Los objetivos propuestos y conseguidos en este PI han sido:

- Recolección de logs del CMS, es decir, tendremos centralizados los logs de aquellas máquinas que nos interesen, en el caso del escenario será de cada uno de los drupal.
- Monitorización de métricas, es decir, al trabajar con *docker-compose*, he considerado necesario monitorizar el host, pero si usásemos otro tipo de tecnología, podríamos monitorear tantas máquinas como sean necesarias.
- Monitorización de la BBDD, es decir, usaremos un software que nos permitirá visualizar datos referentes a la misma mediante el uso de gráficas y datos de interés.
- Alertas automáticas al suceder determinados eventos, es decir, podemos programar que al suceder cierto acontecimiento, nos alerte por el medio que elijamos.
- Facilitar un programa usando la API de Grafana para tareas rutinarias, es decir, podemos flexibilizar ciertas tareas comunes si usamos la API de Grafana, ya que ésta nos permitirá una mayor rapidez para tareas recurrentes.

2. Escenario necesario para la realización del proyecto (este apartado es optativo)

En este escenario necesitaremos cierto software:

- Un servidor web, siendo en mi caso Apache el seleccionado.
- Una base de datos, siendo MariaDB la usada.
- Un CMS + PHP-FPM, siendo Drupal el seleccionado.
- Una herramienta de visualización de datos, como Grafana.
- Un sistema de agregación de registros, como Loki.
- Un sistema de recolección de logs, como Fluent Bit.
- Un sistema de monitorización, como Prometheus.
- Un sistema de exportación de métricas, como Node Exporter.
- Un sistema de exportación de métricas de la BBDD, como MySQL Server Exporter

En este caso he usado el software mencionado anteriormente simulando un escenario real que se podría encontrar, aunque se podrían añadir y/o eliminar cierta parte del mismo.

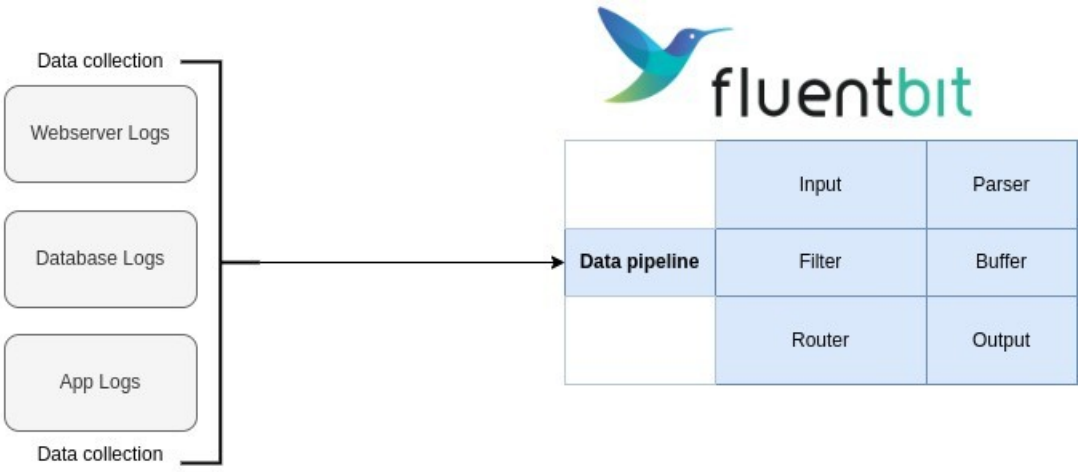
3. Si es necesario los fundamentos teóricos y conceptos.

Este apartado será dividido en bloques, ya que la monitorización del escenario está compuesta por elementos independientes, por lo que explicaré de manera conjunta aquellos elementos que tengan relación.

3.1 Fluent Bit y Loki

3.1.1 ¿Qué es Fluent Bit?

Fluent Bit es un recolector de registros (logs) y métricas de código abierto diseñado para recopilar y enviar datos de registro y métricas desde diferentes fuentes a múltiples destinos. Es parte de la familia de proyectos Fluentd, una plataforma de registro y agregación de datos en tiempo real.



3.1.2 ¿Por qué usar Fluent Bit?

Hay varias razones por las que se podría optar por utilizar Fluent Bit para la recolección y envío de registros y métricas en un sistema. Aquí te presento algunas de ellas:

1) Ligero y rápido: Fluent Bit es una herramienta de recolección de registros y métricas muy ligera y rápida, lo que lo hace adecuado para sistemas con recursos limitados, como dispositivos IoT y contenedores.
2) Integración con diferentes fuentes de datos: Fluent Bit puede recopilar datos de diferentes fuentes, como archivos de registro, syslog, Docker, Kubernetes, AWS, y muchos más, lo que lo hace muy versátil.

3) Múltiples opciones de salida: Una vez que los datos son recopilados, Fluent Bit ofrece una variedad de opciones de salida, incluyendo servicios de almacenamiento en la nube, bases de datos, sistemas de gestión de logs, y más.
4) Personalizable: Fluent Bit es altamente personalizable, lo que permite a los usuarios modificar su comportamiento mediante complementos y configuraciones personalizadas.
5) Parte de la familia de proyectos Fluentd: Fluent Bit es parte de la familia de proyectos Fluentd, una plataforma de registro y agregación de datos en tiempo real, lo que le da un respaldo de una comunidad activa y experimentada.

En resumen, Fluent Bit es una herramienta eficiente, versátil y personalizable para la gestión de registros y métricas en sistemas distribuidos, que se integra bien con otras herramientas de monitoreo y ofrece una alternativa más ligera y fácil de usar que las soluciones tradicionales.

3.1.3 ¿Qué es Loki?

Loki es un sistema de agregación de registros (logs) de código abierto. Fue creado por Grafana Labs y se basa en la misma tecnología utilizada en Prometheus.

Loki es una alternativa a los sistemas tradicionales de almacenamiento de registros, como Elasticsearch y Splunk, que suelen ser costosos y complejos de administrar. Loki utiliza una arquitectura de almacenamiento distribuido, escalable y altamente disponible, que permite almacenar y consultar grandes volúmenes de registros de forma eficiente.

Lo que distingue a Loki de otros sistemas de almacenamiento de registros es su enfoque en la etiquetación. Los registros son etiquetados con metadatos que los describen, lo que permite a los usuarios buscar y filtrar los registros de manera más eficiente y específica. Loki también es altamente integrable con otras herramientas de monitoreo, como Grafana, lo que permite una visualización y análisis fácil y personalizado de los datos recopilados.

En resumen, Loki es una solución de almacenamiento de registros escalable y altamente eficiente, que se integra bien con otras herramientas de monitoreo y ofrece una alternativa más económica y fácil de usar que las soluciones tradicionales.

3.1.4 ¿Por qué usar Loki?

Existen varias razones por las que podría considerarse el uso de Loki para la gestión de registros en sistemas distribuidos:

1) Escalabilidad: Loki está diseñado para ser altamente escalable, lo que significa que puede manejar grandes volúmenes de registros sin comprometer el rendimiento.
2) Eficiencia: Loki utiliza una tecnología de indexación basada en etiquetas que permite filtrar y buscar registros de manera eficiente y específica. Además, su arquitectura distribuida permite una alta disponibilidad de los datos.
3) Integración con otras herramientas: Loki se integra bien con otras herramientas de monitoreo y visualización de datos, como Grafana, lo que permite una visualización y análisis fácil y personalizado de los datos recopilados.
4) Bajo costo: En comparación con otras soluciones de almacenamiento de registros, Loki puede resultar más económico debido a su arquitectura distribuida y al uso de tecnología de indexación eficiente.
5) Flexibilidad: Loki es altamente personalizable, lo que permite a los usuarios modificar su comportamiento mediante complementos y configuraciones personalizadas.

En resumen, Loki ofrece una solución escalable, eficiente, integrable y flexible para la gestión de registros en sistemas distribuidos, que puede resultar más económico que otras soluciones de almacenamiento de registros. Su enfoque en la indexación basada en etiquetas y su integración con otras herramientas de monitoreo y visualización, como Grafana, lo hacen una opción atractiva para la gestión de logs en sistemas complejos.

3.1.5 Resumen y complementación de ambas herramientas

Loki y Fluent Bit son herramientas complementarias para la gestión de registros en sistemas distribuidos. Fluent Bit se encarga de la recolección y envío de registros, mientras que Loki se encarga del almacenamiento, indexación y consulta de los mismos.

En conjunto, Fluent Bit y Loki proporcionan una solución completa para la gestión de registros en sistemas distribuidos.

3.2. Prometheus y Node Exporter

3.2.1 ¿Qué es Prometheus?

Prometheus es un sistema de monitoreo y alerta de código abierto diseñado para monitorear aplicaciones y sistemas distribuidos. Fue creado por SoundCloud en 2012 y es ahora un proyecto de la Cloud Native Computing Foundation (CNCF).

Prometheus recopila métricas en tiempo real de diferentes fuentes, como servicios, aplicaciones y sistemas, utilizando una arquitectura de recolección de datos basada en modelos.

3.2.2 ¿Por qué usar Prometheus?

Algunas de las características que nos podrían servir de Prometheus incluyen:

1. Recolección flexible de métricas: Prometheus puede recopilar métricas de diferentes fuentes, incluyendo servicios HTTP, servicios DNS, bases de datos, sistemas operativos, entre otros.
2. Modelado de datos: Prometheus utiliza un modelo de datos basado en etiquetas (label-based) que permite una mejor organización y agrupación de las métricas recopiladas.
3. Lenguaje de consultas: Prometheus utiliza su propio lenguaje de consultas (PromQL) para consultar y visualizar los datos recopilados.
4. Almacenamiento de series de tiempo: Prometheus almacena las métricas recopiladas como series de tiempo, lo que permite un acceso eficiente y rápido a los datos históricos.
5. Alertas: Prometheus cuenta con un sistema de alertas integrado que permite definir alertas basadas en umbrales de métricas y enviar notificaciones a diferentes canales.

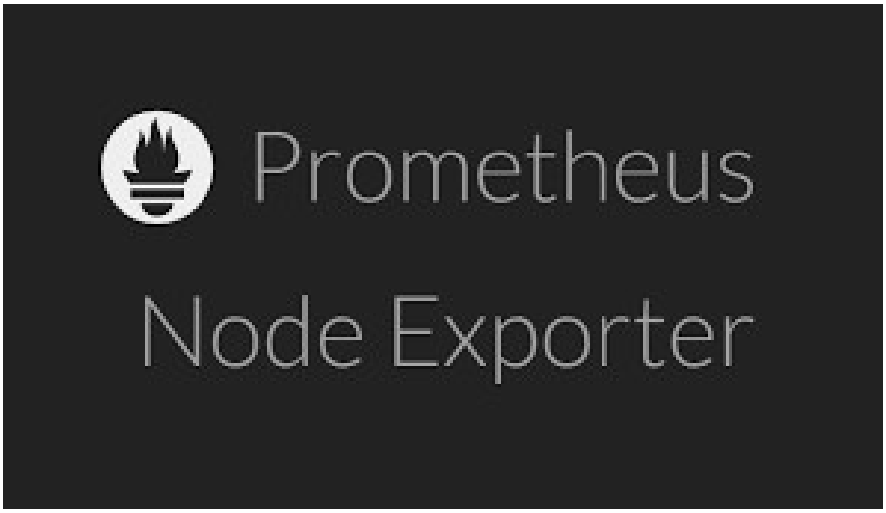
En resumen, Prometheus es un sistema de monitoreo y alerta de código abierto que recopila métricas en tiempo real de diferentes fuentes utilizando un modelo de datos basado en etiquetas. Su

arquitectura de recolección de datos basada en modelos, su lenguaje de consultas, su almacenamiento de series de tiempo y su sistema de alertas integrado lo hacen una herramienta útil y poderosa para monitorear aplicaciones y sistemas distribuidos.

3.2.3 ¿Qué es Node Exporter?

Node Exporter es un colector de métricas de sistema de código abierto utilizado para recopilar métricas del sistema operativo y del hardware de un servidor. Fue desarrollado por el equipo de Prometheus y es compatible con este sistema de monitoreo, aunque también puede ser utilizado con otros sistemas.

Node Exporter se ejecuta como un proceso en segundo plano en el servidor y recopila información del sistema operativo y del hardware, como la carga de CPU, la memoria utilizada, la utilización del disco y la red, entre otros. Luego, expone estas métricas a través de un servidor web HTTP en un formato que puede ser procesado por sistemas de monitoreo como Prometheus.



3.2.4 ¿Por qué usar Node Exporter?

Algunas de las características que nos podrían hacer decantarnos por Node Exporter son las siguientes:

1. Exportación de métricas a través de HTTP: Node Exporter expone las métricas recopiladas a través de un servidor web HTTP en un formato que puede ser procesado por sistemas de monitoreo.
2. Fácil integración con sistemas de monitoreo: Node Exporter es compatible con sistemas de monitoreo populares como Prometheus y puede ser utilizado en conjunto con ellos para monitorear el rendimiento del sistema y detectar posibles problemas.
3. Recopilación de métricas de sistema operativo y hardware: Node Exporter puede recopilar métricas de diferentes componentes del sistema operativo y del hardware, incluyendo CPU, memoria, disco, red, entre otros.

3.2.5 Resumen e implementación de ambas herramientas

Prometheus y Node Exporter son dos herramientas muy útiles para monitorear el rendimiento y la disponibilidad de sistemas y aplicaciones. Mientras que Prometheus es un sistema de monitoreo de métricas que recopila y almacena métricas en tiempo real, Node Exporter es un colector de métricas de sistema que recopila métricas del sistema operativo y del hardware de un servidor y las expone en un formato que puede ser procesado por sistemas de monitoreo como Prometheus.

5.3 MySQL Server Exporter

5.3.1 ¿Qué es MySQL Exporter?

MySQL Exporter es un colector de métricas de MySQL que recopila y expone métricas del motor de base de datos MySQL para su monitoreo y análisis. Fue desarrollado por el equipo de Prometheus y es compatible con este sistema de monitoreo, aunque también puede ser utilizado con otros sistemas.

MySQL Exporter se ejecuta como un proceso en segundo plano y recopila métricas de MySQL a través de consultas SQL específicas. Luego, expone estas métricas a través de un servidor web HTTP en un formato que puede ser procesado por sistemas de monitoreo como Prometheus.

Entre las métricas que MySQL Exporter puede recopilar y exponer se incluyen la utilización de CPU y memoria de MySQL, el número de conexiones activas y la cantidad de consultas por segundo, entre otros. Además, MySQL Exporter también puede recopilar métricas de la

configuración del servidor, lo que permite monitorear cambios en la configuración y detectar posibles problemas.

En resumen, MySQL Exporter es un colector de métricas de MySQL que recopila y expone métricas del motor de base de datos MySQL para su monitoreo y análisis. Su capacidad para recopilar métricas específicas de MySQL y exponerlas en un formato que puede ser procesado por sistemas de monitoreo lo hacen una herramienta útil para monitorear y analizar el rendimiento de las bases de datos MySQL.

5.3.2. ¿Por qué usar MySQL Exporter?

MySQL Exporter es una herramienta muy útil para monitorear y analizar el rendimiento de las bases de datos MySQL. Algunas de las razones por las cuales se puede utilizar MySQL Exporter son:

1. Identificar cuellos de botella: MySQL Exporter puede recopilar métricas de rendimiento clave de MySQL, como la utilización de CPU y memoria, la cantidad de conexiones activas

y la cantidad de consultas por segundo. Al analizar estas métricas, es posible identificar cuellos de botella y tomar medidas para optimizar el rendimiento de la base de datos.

2. Alertas de problemas: MySQL Exporter puede generar alertas en tiempo real cuando se detectan problemas en la base de datos, como una alta carga de consultas o una baja disponibilidad. Esto permite a los equipos de operaciones responder rápidamente a los problemas y minimizar el impacto en los usuarios.
3. Análisis de tendencias: Al monitorear las métricas de rendimiento de MySQL a lo largo del tiempo, es posible identificar tendencias y patrones en el rendimiento de la base de datos. Esto puede ayudar a los equipos de operaciones a planificar mejor las necesidades de capacidad y tomar medidas proactivas para optimizar el rendimiento.

En resumen, MySQL Exporter es una herramienta útil para monitorear y analizar el rendimiento de las bases de datos MySQL. Permite a los equipos de operaciones identificar cuellos de botella, generar alertas de problemas y realizar un análisis de tendencias a lo largo del tiempo, lo que puede mejorar el rendimiento y la disponibilidad de la base de datos.

5.4 Grafana

5.4.1 ¿Qué es Grafana?

Grafana es una plataforma de análisis y visualización de datos de código abierto que permite monitorear, analizar y visualizar datos de diversas fuentes en tiempo real. Con Grafana, los usuarios pueden crear dashboards personalizados y visualizar métricas y estadísticas en tiempo real, lo que facilita la identificación de patrones y tendencias en los datos.

Grafana se integra con una variedad de fuentes de datos, incluyendo bases de datos SQL y NoSQL, sistemas de métricas como Prometheus y Graphite, y herramientas de monitoreo como Zabbix y Nagios. Además, Grafana es altamente personalizable y admite una amplia variedad de complementos y extensiones.

5.4.2 ¿Por qué usar Grafana?

Hay varias razones por las cuales es recomendable utilizar Grafana en proyectos de análisis y visualización de datos, entre las cuales se incluyen:

1. Integración con una variedad de fuentes de datos: Grafana se integra con una amplia variedad de fuentes de datos, incluyendo bases de datos SQL y NoSQL, sistemas de métricas como Prometheus y Graphite, y herramientas de monitoreo como Zabbix y Nagios. Esto permite que los usuarios puedan conectar y visualizar datos de múltiples fuentes en un solo lugar.
2. Personalización y flexibilidad: Grafana es altamente personalizable y admite una amplia variedad de complementos y extensiones, lo que permite que los usuarios puedan personalizar sus dashboards y visualizaciones según sus necesidades específicas. Además, es posible integrar plugins personalizados desarrollados por la comunidad.
3. Interfaz de usuario intuitiva y fácil de usar: Grafana ofrece una interfaz de usuario intuitiva y fácil de usar que permite a los usuarios crear dashboards personalizados y widgets de visualización de datos de manera rápida y sencilla.
4. Admite alertas y notificaciones: Grafana permite la configuración de alertas y notificaciones, lo que permite a los usuarios monitorear en tiempo real sus métricas y recibir alertas cuando se detectan anomalías.
5. Comunidad activa: Grafana cuenta con una comunidad de usuarios y desarrolladores activa, lo que asegura que la plataforma esté en constante evolución y mejora, con nuevas características y funcionalidades agregadas regularmente.

En resumen, Grafana es una plataforma de análisis y visualización de datos que ofrece integración con una amplia variedad de fuentes de datos, personalización y flexibilidad, interfaz de usuario intuitiva y fácil de usar, capacidad para configurar alertas y notificaciones, y una comunidad activa de usuarios y desarrolladores.

5.5 Grafana API

5.5.1 ¿Qué debo saber de la API de Grafana?

La API de Grafana es una interfaz de programación de aplicaciones que permite a los desarrolladores interactuar con la plataforma de monitoreo y visualización Grafana. Algunas cosas que es importante saber sobre la API de Grafana son:

1. Recursos disponibles: La API de Grafana ofrece una variedad de recursos, como dashboards, paneles, alertas y usuarios, que pueden ser manipulados y personalizados mediante llamadas a la API.
2. Métodos disponibles: La API de Grafana ofrece una variedad de métodos HTTP para interactuar con los recursos, incluyendo GET, POST, PUT y DELETE.
3. Autenticación y autorización: La API de Grafana utiliza autenticación y autorización basada en tokens. Los usuarios deben generar un token de API válido para interactuar con la API.
4. Documentación de la API: La documentación de la API de Grafana está disponible en línea y proporciona detalles sobre los recursos disponibles, los métodos admitidos, los parámetros necesarios y los formatos de respuesta.
5. Integración con otras herramientas: La API de Grafana se puede integrar con otras herramientas y plataformas de monitoreo y visualización, como Prometheus y InfluxDB.

En resumen, la API de Grafana es una interfaz de programación de aplicaciones que permite a los desarrolladores interactuar con la plataforma de monitoreo y visualización Grafana. Ofrece una variedad de recursos y métodos HTTP para interactuar con ellos. Los usuarios deben autenticarse y autorizarse mediante tokens de API válidos. La documentación de la API está disponible en línea y la API se puede integrar con otras herramientas y plataformas.

5.6 MariaDB API

5.6.1 ¿Qué debo saber de la API de Mariadb?

La API de MariaDB es una interfaz de programación de aplicaciones que permite a los desarrolladores interactuar con una base de datos MariaDB. Algunas cosas que es importante saber sobre la API de MariaDB son:

1. Lenguajes de programación compatibles: La API de MariaDB es compatible con una variedad de lenguajes de programación, incluyendo PHP, Python, Ruby y Java, entre otros.
2. Funciones disponibles: La API de MariaDB ofrece una variedad de funciones para interactuar con la base de datos, como la ejecución de consultas SQL, la gestión de transacciones y la recuperación de datos.

3. Conexiones a la base de datos: Para utilizar la API de MariaDB, es necesario establecer una conexión a la base de datos. Esto se puede hacer utilizando una variedad de métodos, como proporcionar un nombre de host y un nombre de usuario y contraseña, o utilizando una conexión de socket.
4. Configuración de la API: La API de MariaDB se puede configurar para adaptarse a las necesidades específicas de una aplicación. Esto incluye opciones como la configuración del tamaño de búfer y la selección del conjunto de caracteres utilizado para comunicarse con la base de datos.

En resumen, la API de MariaDB es una interfaz de programación de aplicaciones que permite a los desarrolladores interactuar con una base de datos MariaDB. Ofrece una variedad de funciones para interactuar con la base de datos y es compatible con varios lenguajes de programación. Para utilizar la API, es necesario establecer una conexión a la base de datos y se pueden configurar opciones específicas de la API para adaptarse a las necesidades de la aplicación

4. Descripción detallada de los que se ha realizado

4.1 Instalación y configuración del escenario

Todo el escenario se monta usando docker y docker-compose, por lo que pasaré a detallar las líneas referentes a cada componente del escenario.

4.1.2 Drupal:

```
drupal:
  image: drupal:9.4-php8.0-apache-bullseye
  restart: always
  volumes:
    - drupal-data:/opt/drupal/web/
  depends_on:
    - mariadb
  ports:
    - "8081-8082:80"
  logging:
    driver: fluentd
    options:
      fluentd-async-connect: "true"
      fluentd-address: localhost:24224
      tag: drupal.logs
```

En este caso he usado una imagen que ya contiene apache y está basada en Debian, siendo además, una versión estable y con cierto recorrido por lo que ha sido la idónea para este escenario.

He creado un volúmen para guardar la información del drupal de forma persistente.

En cuanto al logging, hemos especificado que envíe los logs a fluent bit.

4.1.3 MariaDB:

En este caso he usado la última imagen disponible, y he especificado las variables de entorno correspondientes, creando también un volumen para guardar la información que necesitamos:

```
mariadb:
  container_name: mariadb
  image: mariadb
  restart: always
  ports:
    - "3306:3306"
  environment:
    MYSQL_ROOT_PASSWORD: root
    MYSQL_DATABASE: drupal
    MYSQL_USER: user
    MYSQL_PASSWORD: clave
  volumes:
    - mariadb-data:/var/lib/mysql
```

4.1.4 HAProxy

He elegido una versión de imagen que he probado bastante y no hay problema alguno de compatibilidad y además he añadido un volumen de tipo *bind* para aportar la configuración necesaria.

```
haproxy:
   image: haproxy:1.6
   container_name: haproxy
   restart: always
   ports:
    - 8080:80
   volumes:
    - "./haproxy/haproxy.cfg:/usr/local/etc/haproxy/haproxy.cfg:ro"
```

A continuación podemos ver el fichero de configuración, en el que podemos destacar el tipo de balanceo usado, que es *roundrobin* :

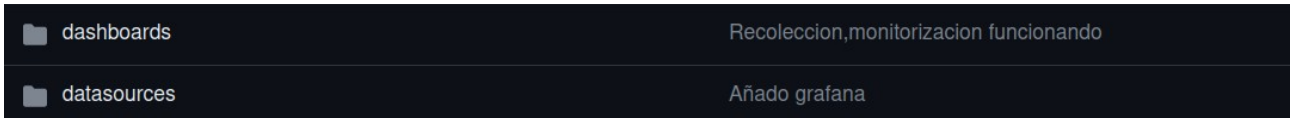
```
backend drupal

        balance roundrobin
        mode http
        server drupal drupal:80 weight 1 minconn 3 maxconn 500 check

        stats uri /haproxy_status

        stats auth admin:admin

        stats admin if TRUE
```

4.1.5 Grafana

```
grafana:
   image: grafana/grafana
   container_name: grafana
   ports:
    - 3000:3000
   depends_on:
     - loki
   environment:
     GF_SECURITY_ADMIN_PASSWORD: admin
     GF_SECURITY_API_KEY: eyJrIjoiTHhWbVlUR0NGUVJpTlNiSnB5SWZGdjNVNkUzOHNTMFUiLCJuIjoiYWRtaW4iLCJpZCI6MX0=
   volumes:
   - grafana-data:/var/lib/grafana
   - ./grafana/provisioning:/etc/grafana/provisioning
```

Como podemos ver en la configuración de grafana, he configurado la *key* para usar la API y la contraseña de *admin*. Tambien he creado dos volúmenes, uno tipo *bind* para proporcionar la configuración, y otro para almacenar la información del software.

En el volumen tipo *bind* hemos proporcionado a grafana la carpeta *provisioning*, que contendrá a su vez dos subcarpetas, una para almacenar los *dashboards* y otra para las fuentes de datos:



Si vemos el fichero relacionado con los *datasources* encontraremos que tenemos a *loki* y *prometheus*, de forma que tras levantar el escenario ya los tendremos añadidos:

```
datasources:
- name: loki
  type: loki
  access: proxy
  orgId: 1
  url: http://loki:3100
  basicAuth: false
  isDefault: true
  version: 1
  editable: true

- name: Prometheus
  type: prometheus
  access: proxy
  orgId: 1
  url: http://prometheus:9090
  basicAuth: false
  isDefault: false
  version: 1
```

4.1.6 Prometheus

```
prometheus:
  image: prom/prometheus:latest
  container_name: prometheus
  volumes:
    - ./config/prometheus.yml:/etc/prometheus/prometheus.yml
  command:
    - --config.file=/etc/prometheus/prometheus.yml
  ports:
   - 9090:9090
```

Como se puede ver, he usado la última imagen de *Prometheus*, además de especificar un volumen tipo *bind*, y además proporcionarle un comando que indique el fichero de configuración que podemos ver a continuación:

```yaml
global:
  scrape_interval: 5s

scrape_configs:
  - job_name: metricas
    scheme: http
    metrics_path: /metrics
    static_configs:
      - targets: ["node-exporter:9100"]

  - job_name: mariadb
    static_configs:
      - targets: ["mysql-exporter:9104"]
```

Como se puede observar, *Prometheus* está conectando con *Node Exporter* y *MySQL Exporter*.

4.1.7 Node Exporter

```yaml
    node-exporter:
      image: prom/node-exporter:latest
      restart: always
      container_name: node-exporter
      ports:
        - "9100:9100"
      volumes:
        - /etc/hostname:/etc/nodename:ro
        - /proc:/host/proc:ro
        - /sys:/host/sys:ro
        - /mnt/docker-cluster:/mnt/docker-cluster:ro
        - /etc/localtime:/etc/localtime:ro
        - /etc/timezone:/etc/TZ:ro
```

Para node exporter hemos usado la última imagen disponible, y además, siguiendo la documentación oficial, hemos especificado los volúmenes vistos en la captura anterior para poder obtener correctamente las métricas del host

4.1.8 Loki

```yaml
loki:
  image: grafana/loki:2.0.0
  container_name: loki
  volumes:
    - ./config/loki.yaml:/etc/config/loki.yaml
  ports:
    - "3100:3100"
```

Para *Loki* hemos usado la imagen 2.0.0, y hemos usado un volumen tipo *bind* para que obtenga el archivo de configuración:

```yaml
auth_enabled: false

server:
  http_listen_port: 3100

ingester:
  lifecycler:
    address: 127.0.0.1
    ring:
      kvstore:
        store: inmemory
      replication_factor: 1
    final_sleep: 0s
  chunk_idle_period: 5m
  chunk_retain_period: 30s
  max_transfer_retries: 0
```

El archivo de configuración es el proporcionado por la herramienta, por lo que no habrá mucho que detallar sobre él.

4.1.9 Fluent Bit

```
fluentd:
  container_name: fluentd
  image: grafana/fluent-bit-plugin-loki
  ports:
    - 24224:24224
    - 24224:24224/udp
    - 2020:2020
    - 2021:2021
  environment:
    LOKI_URL: http://loki:3100/loki/api/v1/push
  volumes:
    - ./fluent-bit/fluent-bit.conf:/fluent-bit/etc/fluent-bit.conf
```

He usado la imagen de *Fluent Bit* de *Grafana*, configurado la variable para que conecte a la API de *Loki* de forma correcta y he montado un volumen de tipo bind para pasar la configuración:

```
    HTTP_Server  On
    HTTP_Listen  0.0.0.0
    HTTP_Port    2020

[INPUT]
    Name              forward
    Listen            0.0.0.0
    Port              24224
    Buffer_Chunk_Size 1M
    Buffer_Max_Size   6M

[Output]
    Name grafana-loki
    Match *
    Url http://loki:3100/loki/api/v1/push
    RemoveKeys source,container_id
    LabelKeys container_name
    BatchWait 1s
    BatchSize 1001024
    LineFormat json
    LogLevel info
```

4.1.10 MySQL Exporter

Para *MySQL Exporter* la configuración más destacable sería la variable de entorno que conecta con la base de datos para obtener las métricas necesarias.

```
  mysql-exporter:
    container_name: mysql-exporter
    image: prom/mysqld-exporter
    environment:
      - DATA_SOURCE_NAME=root:root@(mariadb:3306)/
    ports:
      - "9104:9104"
```

4.1.11 Programa Python

Al ser un programa escrito en python, contiene bastantes líneas de código que serían muy extensas mostrarlas por aquí, por lo que es preferible consultar el mismo a través del repositorio. Aún así podemos repasar el programa en funcionamiento.

Este programa está claramente dividido en dos segmentos:

- **Test a la BBDD**: Parte del programa en la que ejecutaremos múltiples consultas al servidor a modo de prueba para consultar en las gráficas de grafana cómo suben las peticiones por minuto
- **Grafana API:** En esta segunda parte del programa, encontraremos un menú en el cual interactuaremos con la API de Grafana para realizar tareas, siendo las siguientes:

```
Menú de opciones:
1. Ver Data Sources vinculados
2. Información sobre el usuario conectado
3. Añadir usuario
4. Información sobre usuario
5. Ver dashboards
6. Eliminar usuario
7. Modificar contraseña usuario
8. Ver carpetas
9. Crear carpeta
10. Eliminar carpeta
11. Ver organizaciones
12. Salir
```

4.2 Levantando el escenario y comprobando los servicios

Comenzamos clonando el repositorio y levantándolo con el siguiente comando:

```
docker-compose up -d --scale drupal=2
```

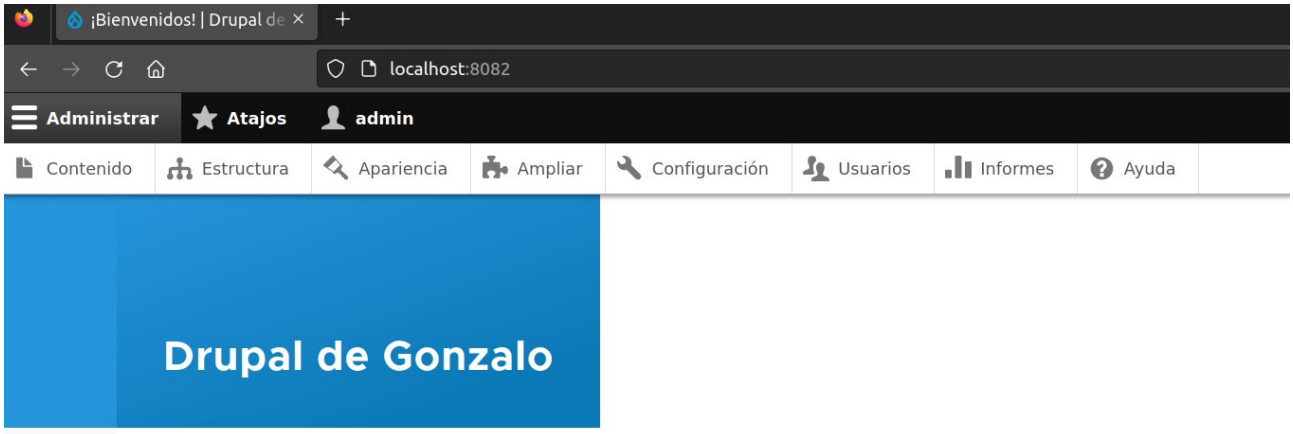
```
gmarin@ubuntu-gmarin:~/github/pi-drupal$ docker-compose up -d --scale drupal=2
Creating network "pi-drupal_default" with the default driver
Creating haproxy        ... done
Creating mariadb        ... done
Creating prometheus ...
Creating mysql-exporter ... done
Creating prometheus         ... done
Creating loki           ...
Creating node-exporter      ... done
Creating loki               ... done
Creating fluentd            ... done
Creating pi-drupal_drupal_1 ... done
Creating pi-drupal_drupal_2 ... done
Creating grafana            ... done
```

En este caso, hemos usado el parámetro *scale* para poder usar dos Drupal simultáneamente, como podemos comprobar visualizando la salida del comando:

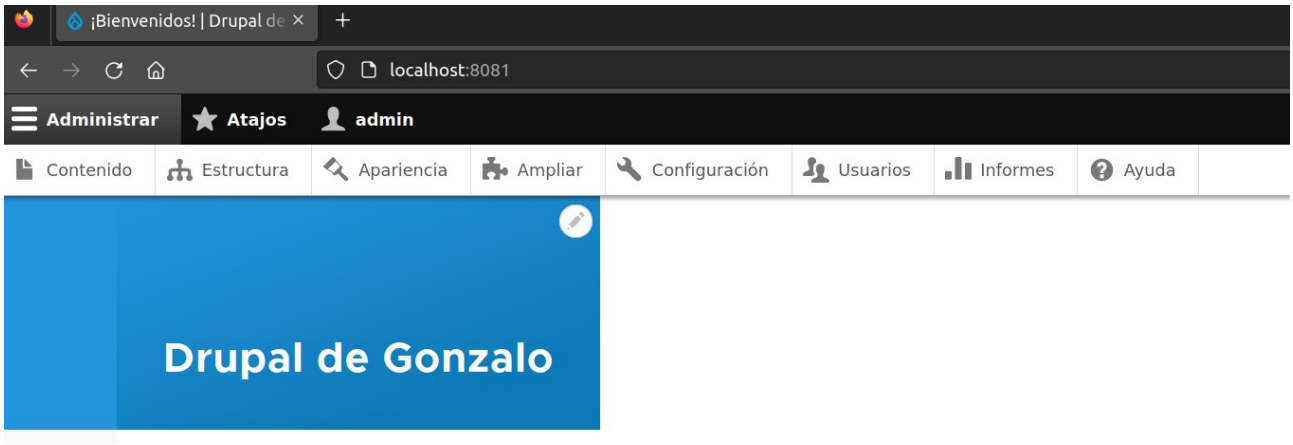
```
docker-compose ps
```

```
                                                  >9100/tcp
pi-drupal_drupal_1   docker-php-entrypoint      Up      0.0.0.0:8082-
                     apac ...                           >80/tcp,:::8082->80/tcp
pi-drupal_drupal_2   docker-php-entrypoint      Up      0.0.0.0:8081-
                     apac ...                           >80/tcp,:::8081->80/tcp
```

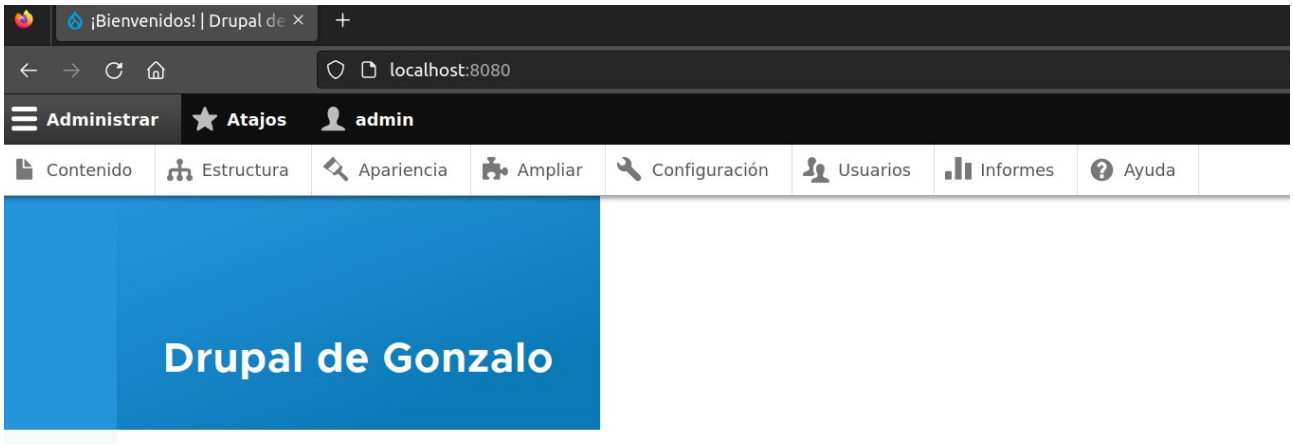
Asumiendo que hemos accedido previamente y hemos instalado el CMS, comenzaré mostrando el CMS, accediendo por el puerto correspondiente a un contenedor, en este caso *pi-drupal_drupal_1*:



También podríamos acceder mediante el puerto de *pi-drupal_drupal_2:*

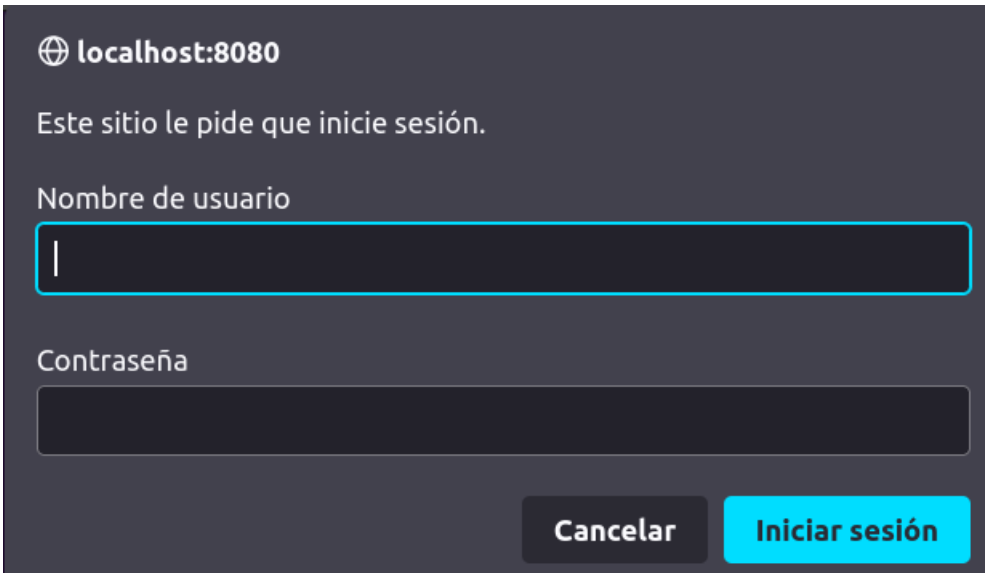


Continuaré accediendo a Drupal pero esta vez por la vía óptima, es decir, a través de HAProxy

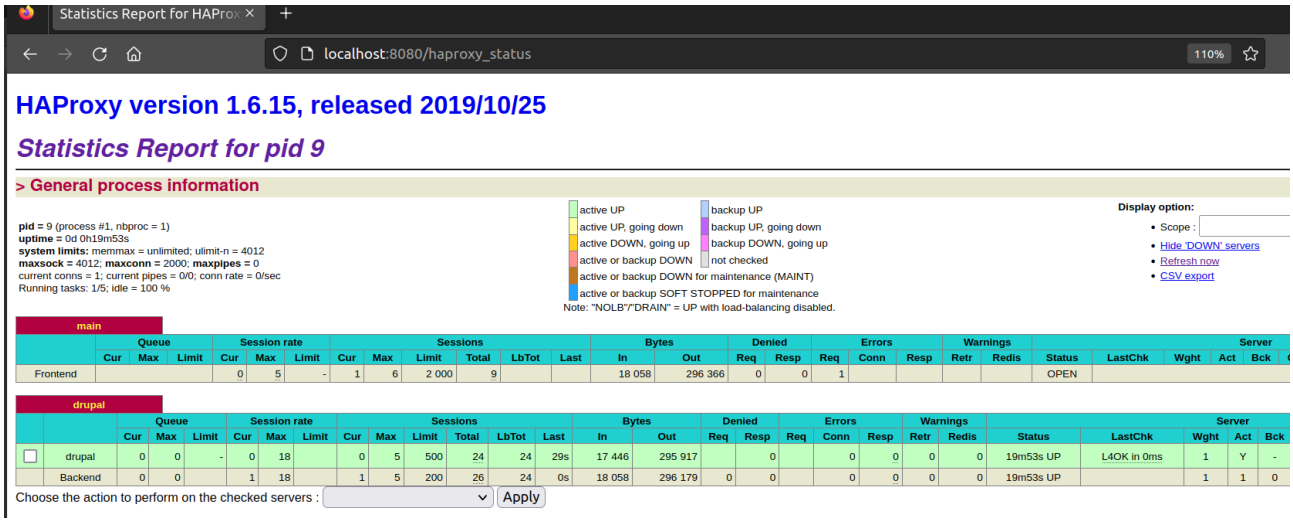


Una vez comprobado que podemos acceder a Drupal y que éste está funcionando correctamente, podemos comprobar que accedemos a las estadísticas de HAProxy, accediendo a la URL siguiente:

http://localhost:8080/haproxy_status



En un principio necesitaremos autenticarnos, y para ello, usaremos las credenciales por defecto (*admin:admin*) y ya podremos ver las estadísticas:

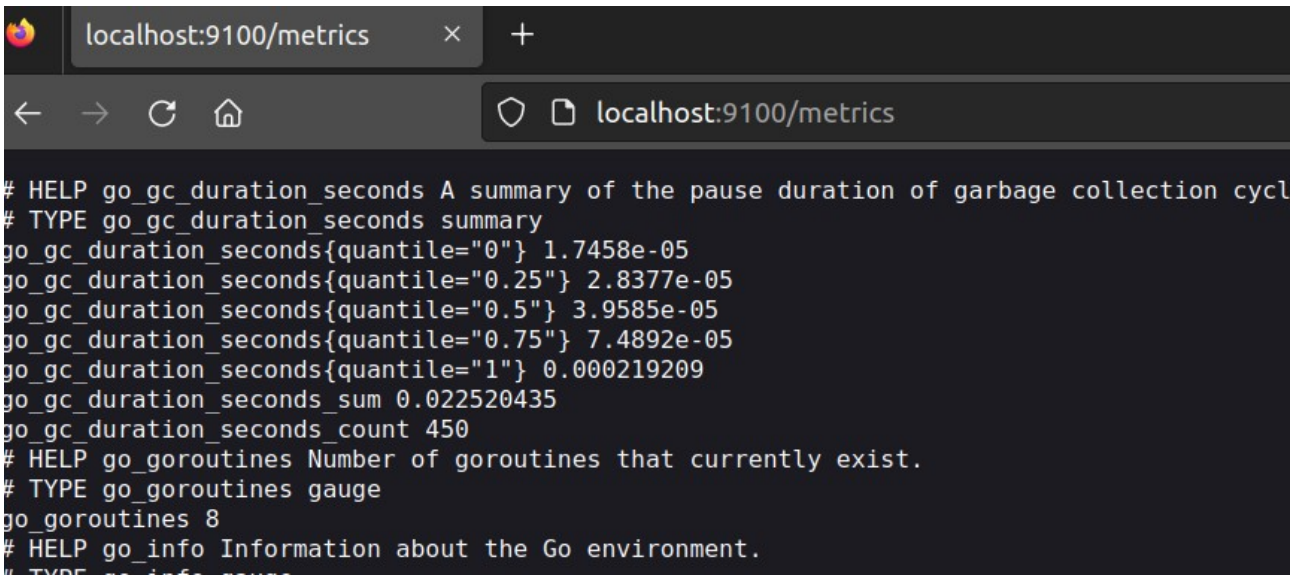


Sobre las estadísticas profundizaré algo más en las demostraciones, por lo que pasaremos a comprobar el siguiente contenedor, siendo *Node Exporter* y para ello, accedemos a la siguiente URL:

http://localhost:9100/

En la *URL* de *Node Exporter* podemos ver que tenemos un hiperenlace que nos redirige a la pestaña correspondiente a las métricas:

http://localhost:9100/metrics

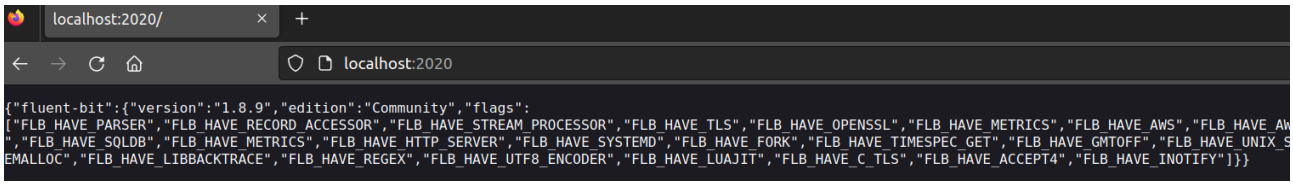


Estas métricas serán las que comprobaremos más a fondo posteriormente.

En cuanto a Loki no podremos comprobar nada del servicio hasta que no usemos Grafana, por lo que lo omitiré.

Continuando con Fluent Bit, tampoco podremos comprobar mucho más allá de Grafana, siendo la URL siguiente la única en la que podemos ver algo de información:

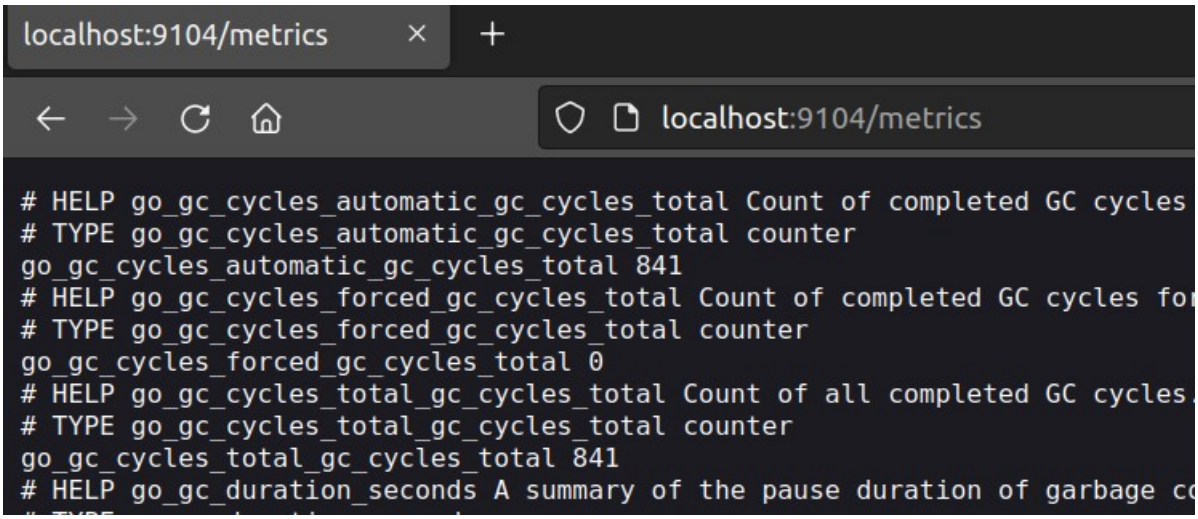
http://localhost:2020/



Antes de pasar al contenedor en el cual habrá más que explicar, accedemos a la URL de *MySQL Exporter*, muy similar a la de *Node Exporter* como podemos comprobar:

http://localhost:9104/

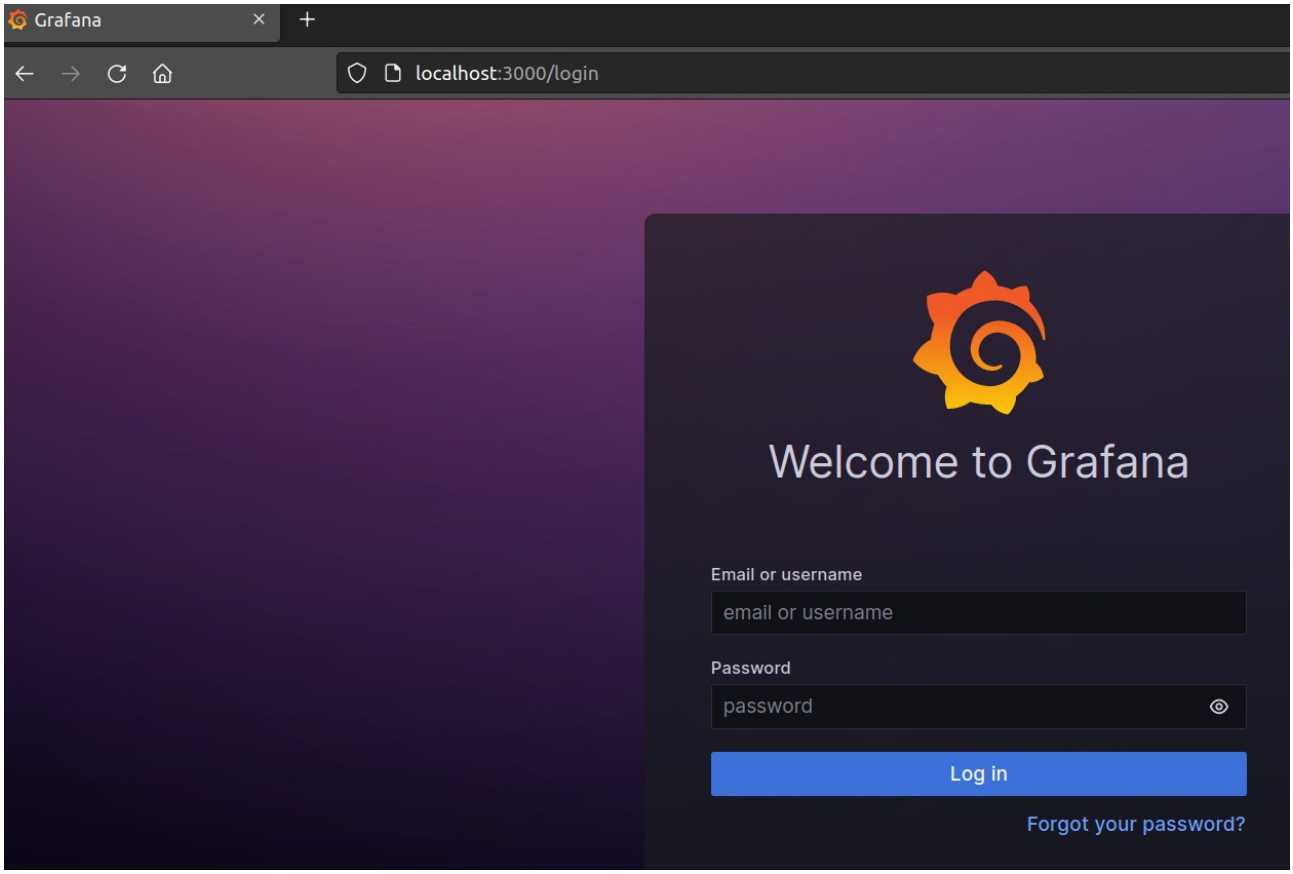
Accedemos a las métricas:



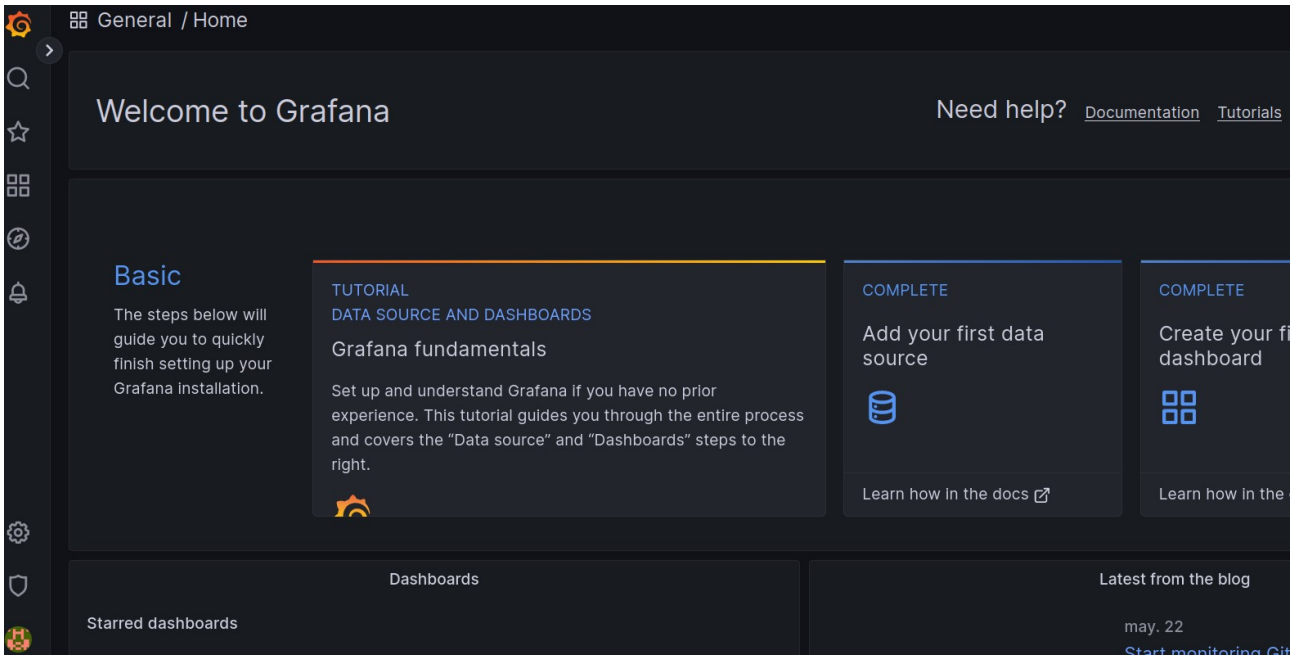
Como podemos observar, son similares a las vistas en *Node Exporter*, por lo que será más óptimo visualizarlas en Grafana de forma más gráfica.

Por último, accederemos al contenedor en el cual centralizamos la mayoría de la información del escenario, Grafana, por lo que accedemos a su URL:

http://localhost:3000/login

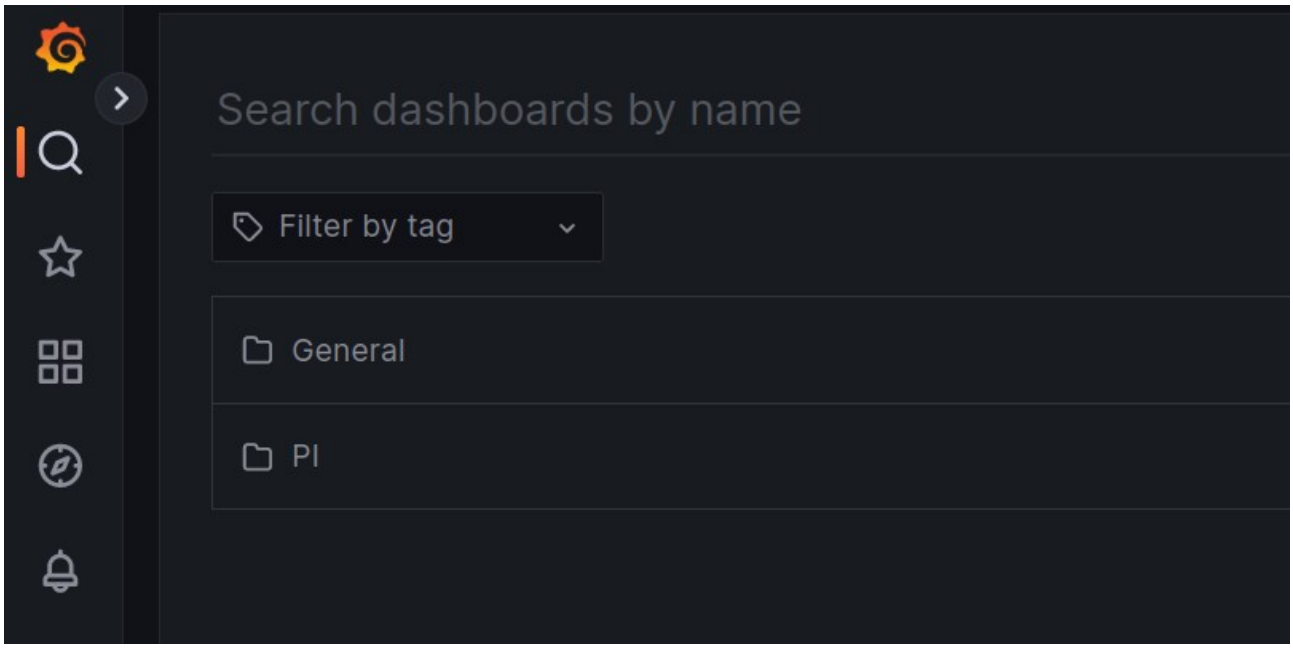


Accedemos con las credenciales por defecto (*admin:admin*) y estaremos en el *Home* :

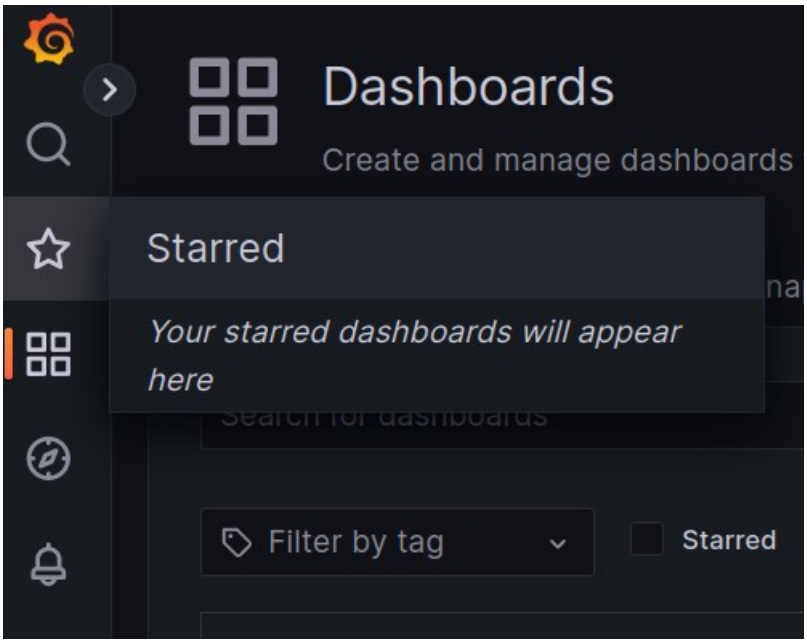


Comenzaremos detallando la interfaz de Grafana, especialmente en el menú de la izquierda donde veremos las distintas opciones.

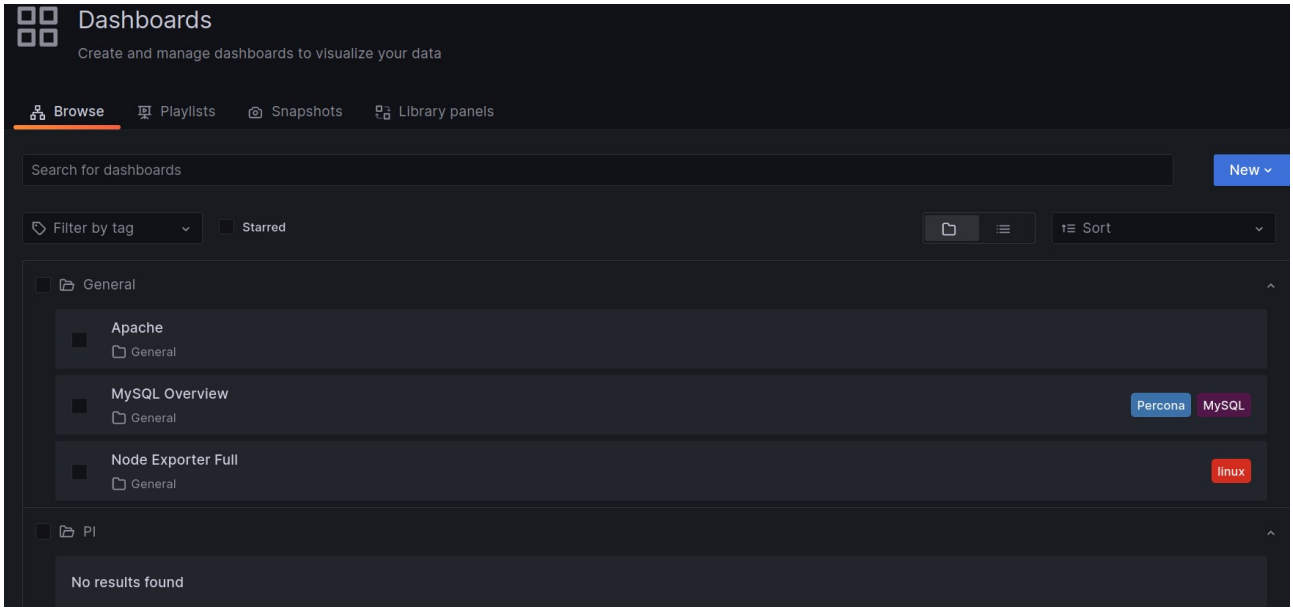
Comenzando por la primera (obviando el logo del servicio que nos redirige a la pantalla principal) donde encontraremos un buscador entre los proyectos y/o dashboards que tengamos agregados:



En la siguiente opción podemos comprobar los dashboards favoritos, apartado que no he usado por lo que será una opción que no detallaré mucho:



Avanzando en el menú, nos encontramos con una de las opciones más interesantes, siendo los dashboards:

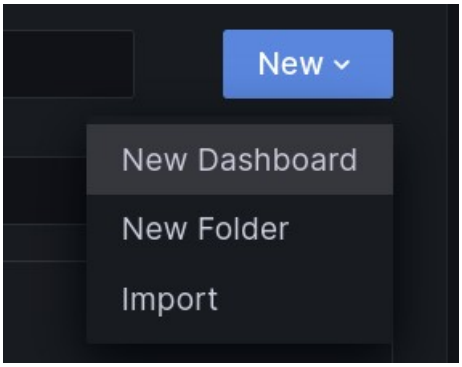


Como podemos ver, tenemos dos carpetas, una llamada *General* y otra llamada *PI*, en la cual podremos clasificar los dashboards.

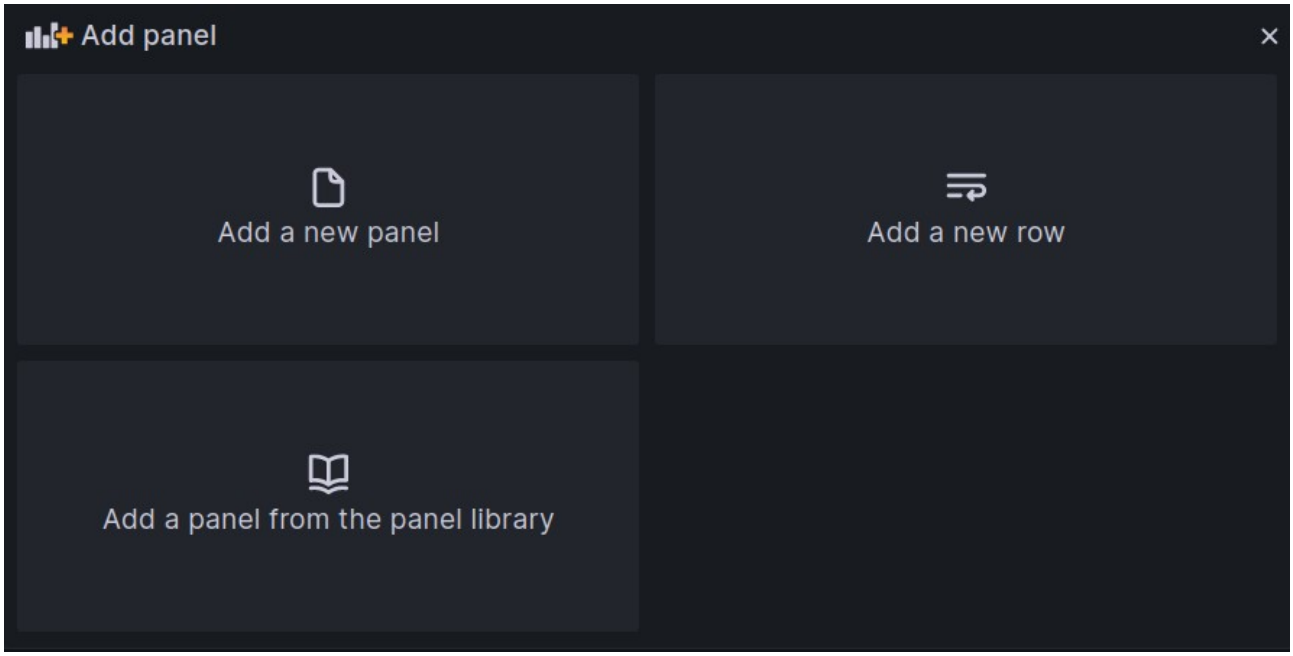
En mi caso, creé la carpeta *PI* para demostrar el uso de esta función, aunque todos los dashboards los tengo añadido en la carpeta general.

Observando la página, encontramos el botón *New*, mediante el cual podemos añadir dashboards y/o carpetas.

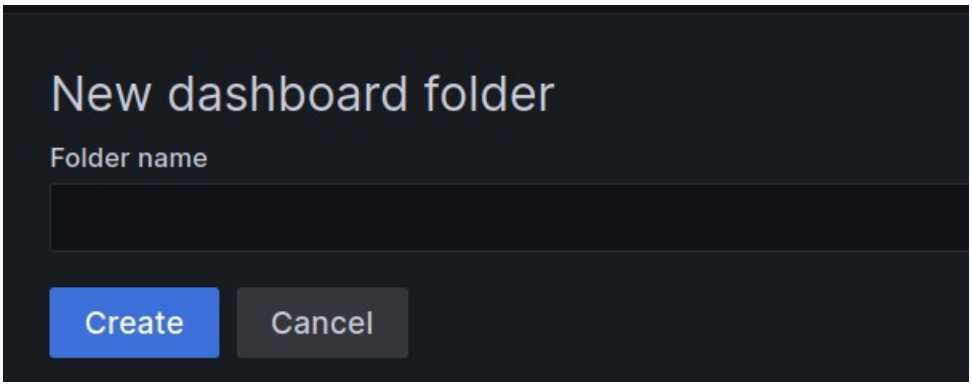
En mi caso, he importado los dashboards mediante volúmenes de tipo *bind* en docker, ya que he aportado un fichero JSON con la información necesaria en el directorio donde se guardan los mismos para que al levantar el escenario estén ya disponibles.



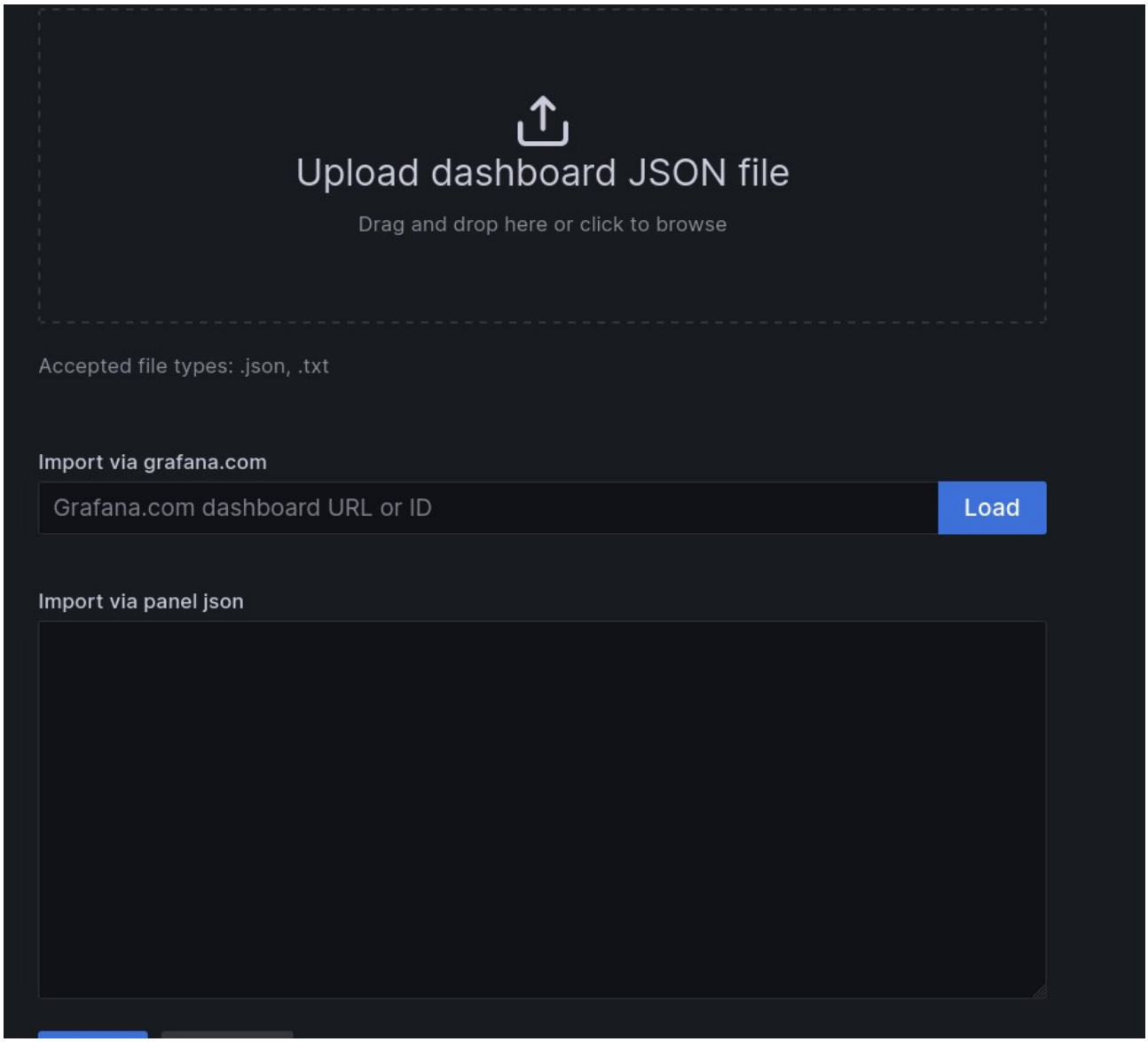
Si pulsamos "*New Dashboards*", veremos las opciones para crear un nuevo panel, opción que puede resultar muy compleja para elaborar un buen panel:



Añadir una carpeta será más sencillo, como podemos comprobar:

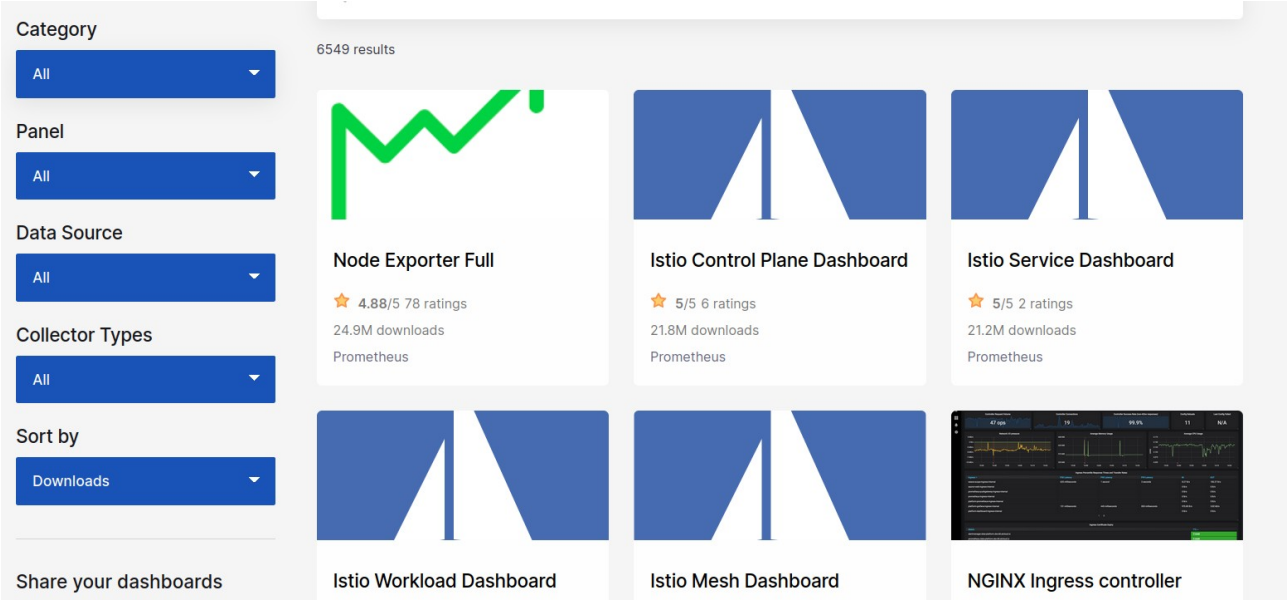


La última opción, referente a importar, es probablemente la más práctica, ya que podemos subir ficheros JSON con el contenido del panel o escribir directamente el código:

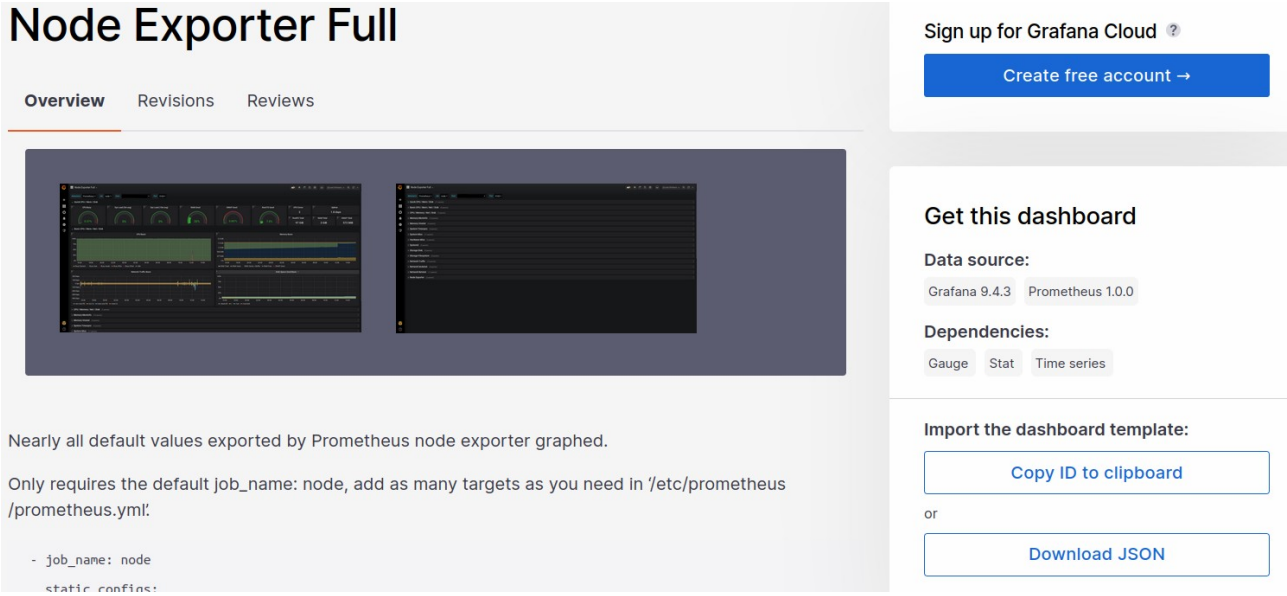


Como podemos comprobar, también existe la posibilidad de importar mediante el ID o URL.

Si accedemos a la siguiente página podemos ver una lista con distintos dashboards:

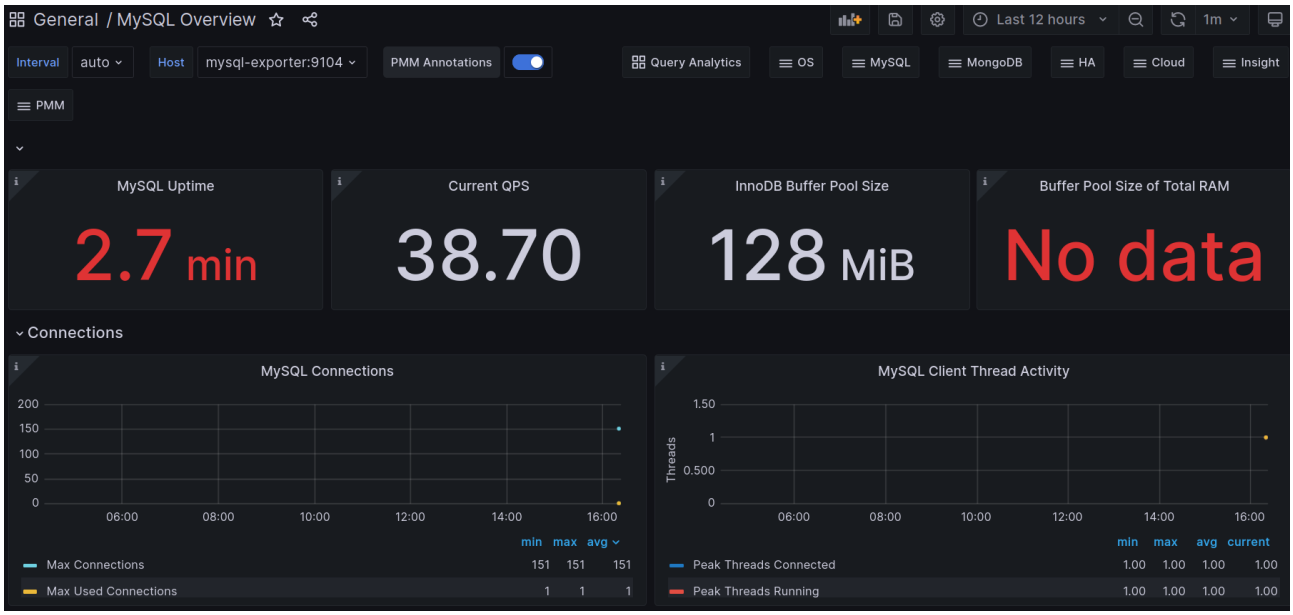


Como podemos comprobar, justamente la primera coincidencia es un panel con *Node Exporter*, que casualmente uso en el escenario por lo que si accedemos a su correspondiente página veremos las opciones para copiar su ID o para descargar el JSON con la información del dashboard:

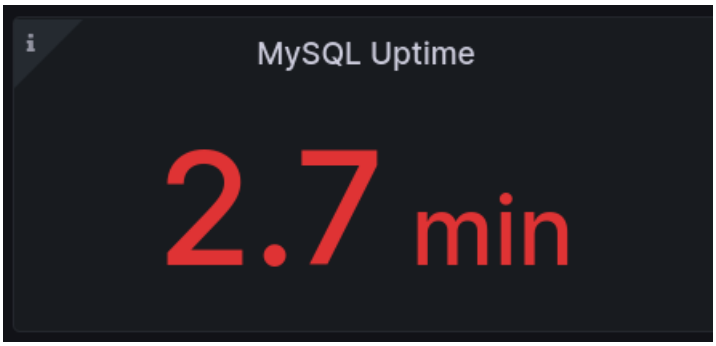


Ahora, repasaré los *Dashboards* que tengo añadidos a mi Grafana.

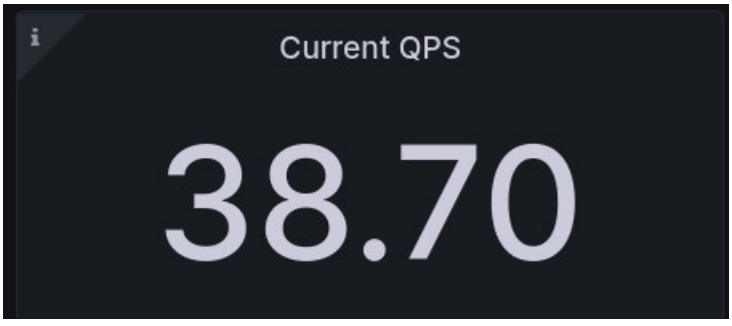
Comenzando por el *MySQL Overview*, estos son los principales paneles que observamos:



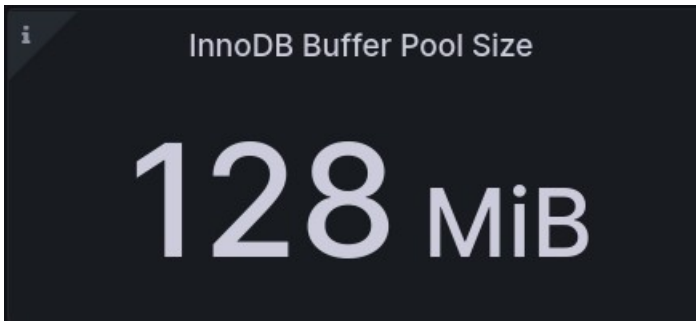
Lo primero a destacar podría ser el tiempo que lleva el contenedor corriendo:



Continuamos con las consultas por segundo:



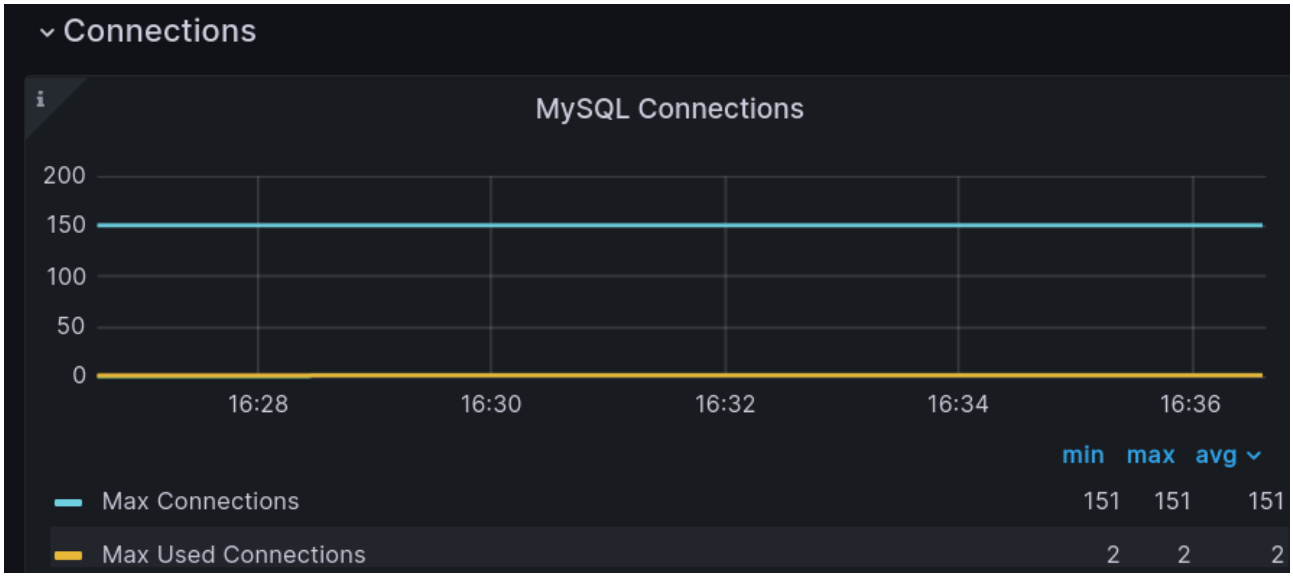
Continuamos con *InnoDB Buffer Pool Size*, que es el tamaño que tiene la memoria asignada para el búfer InnoDB



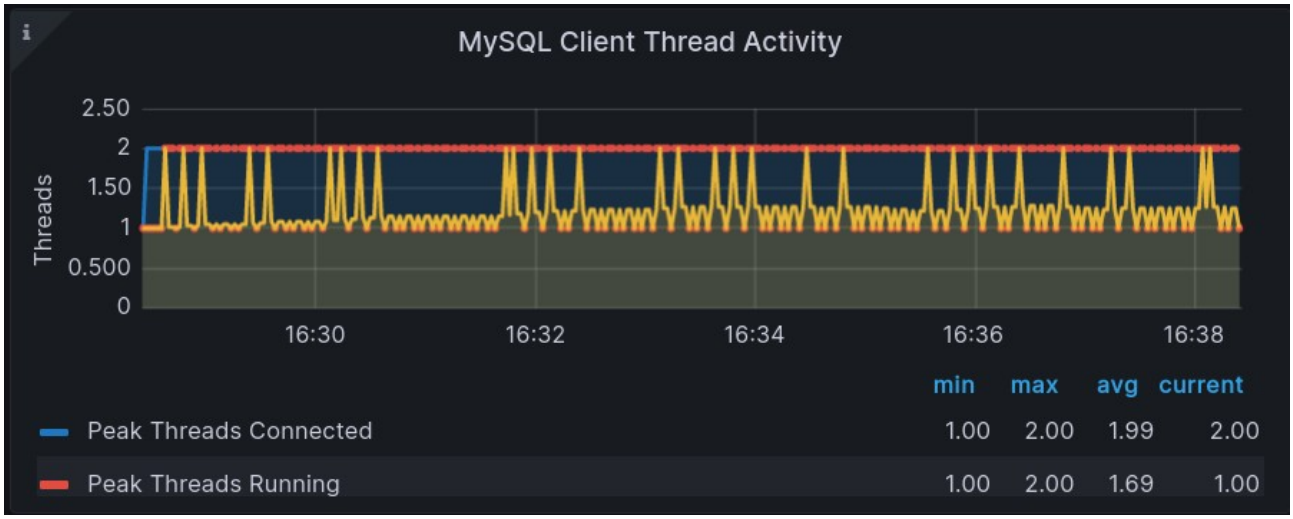
El siguiente panel, *Buffer Pool Size of Total RAM*, deberíamos poder visualizar el tamaño del búfer de memoria de *InnoDB* en relación a la memoria RAM. En este caso, la base de datos es un contenedor y hay conflictos, por lo que no mostrará información:

Ahora, repasaré las dos primeras gráficas y pasaremos al siguiente dashboard, ya que en éstos se muestra muchísima información y sería extenderse demasiado.

La primera es referente a las conexiones MySQL:

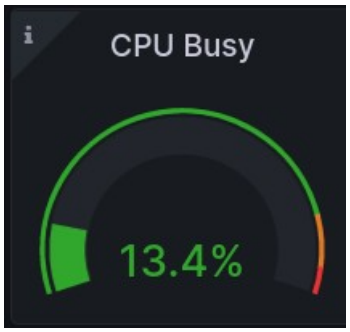


El último panel que mencionaré será el *MySQL Client Thread Activity*, que nos muestra la actividad de los hilos de *MySQL*, que son conexiones de los clientes con el servidor:

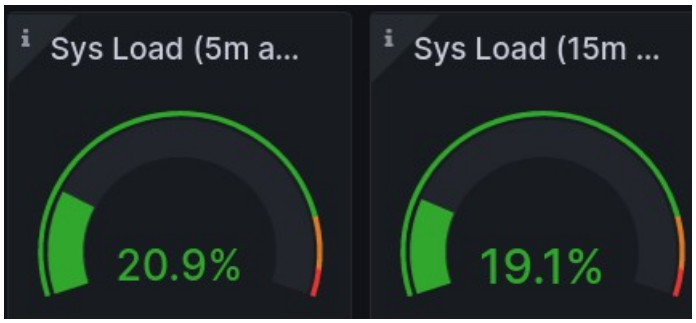


El segundo dashboard es el que nos muestra información de *Node Exporter*:

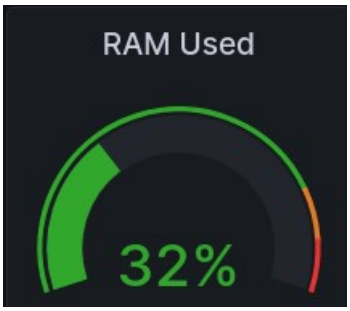
Comenzaremos con el panel, que nos mostrará información sobre la ocupación de la CPU:



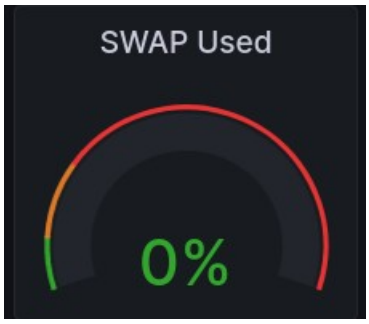
El segundo y tercer panel muestran la carga del sistema, siendo el primero en los últimos 5m y el tercero en los últimos 15m:



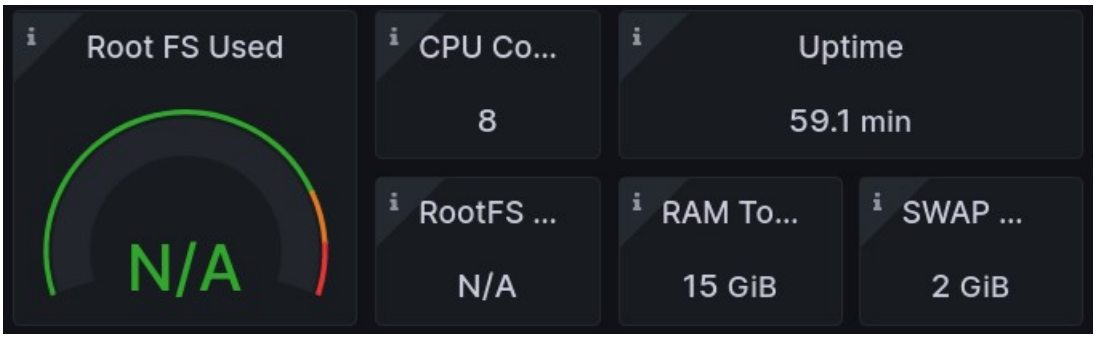
En el cuarto panel veremos el porcentaje de ocupación de la RAM:



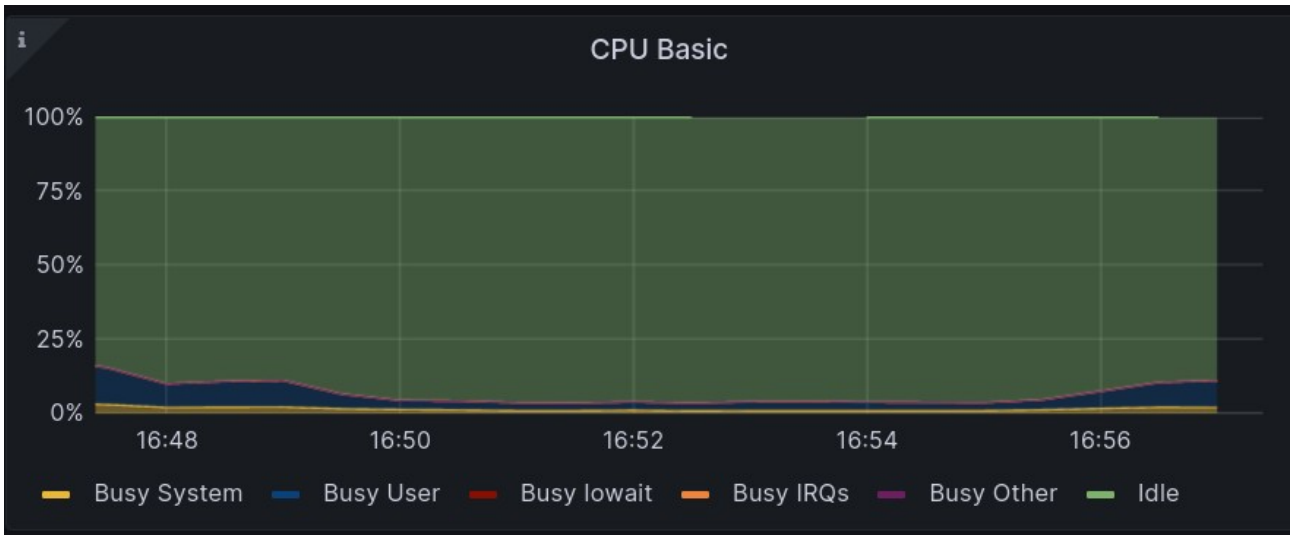
Continuamos con el uso de la SWAP:



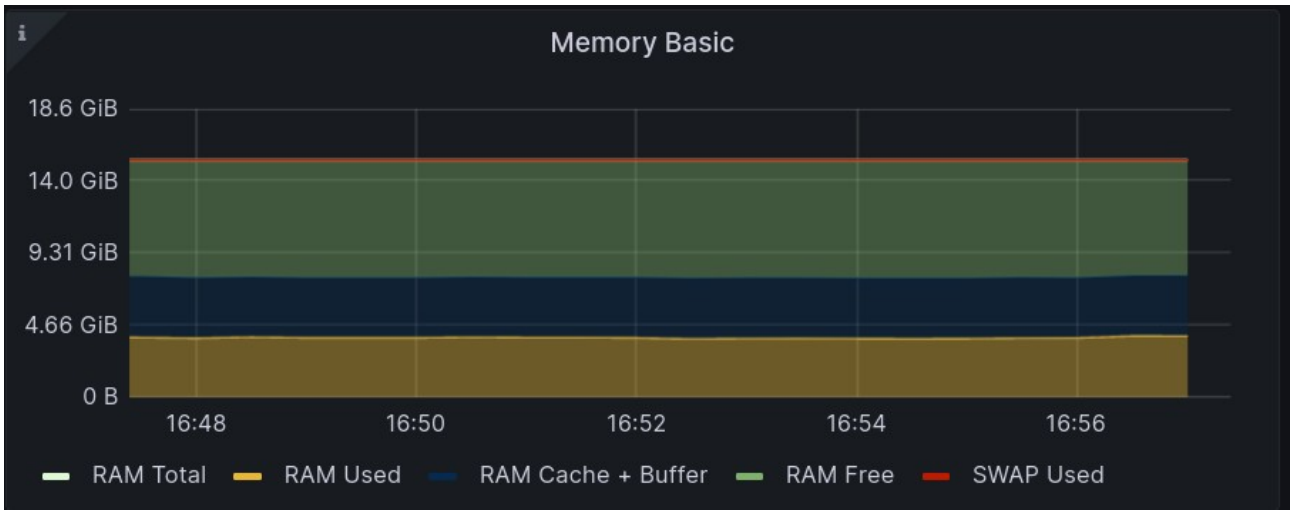
A continuación, hay varios paneles que muestran la información de su título:



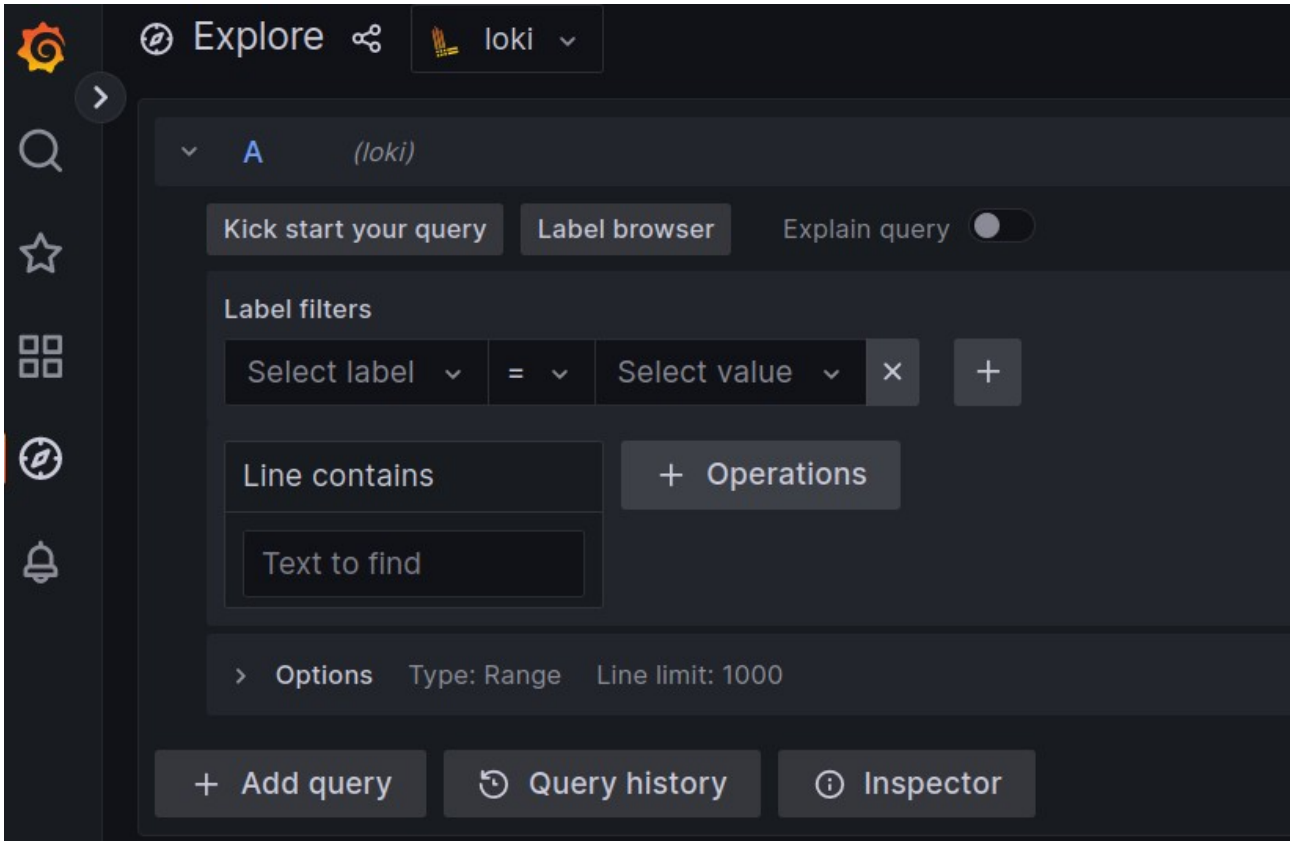
Al igual que con el panel de *MySQL*, veremos múltiples gráficas y solamente repasaré dos de ellas, siendo la primera referente a la ocupación de CPU:



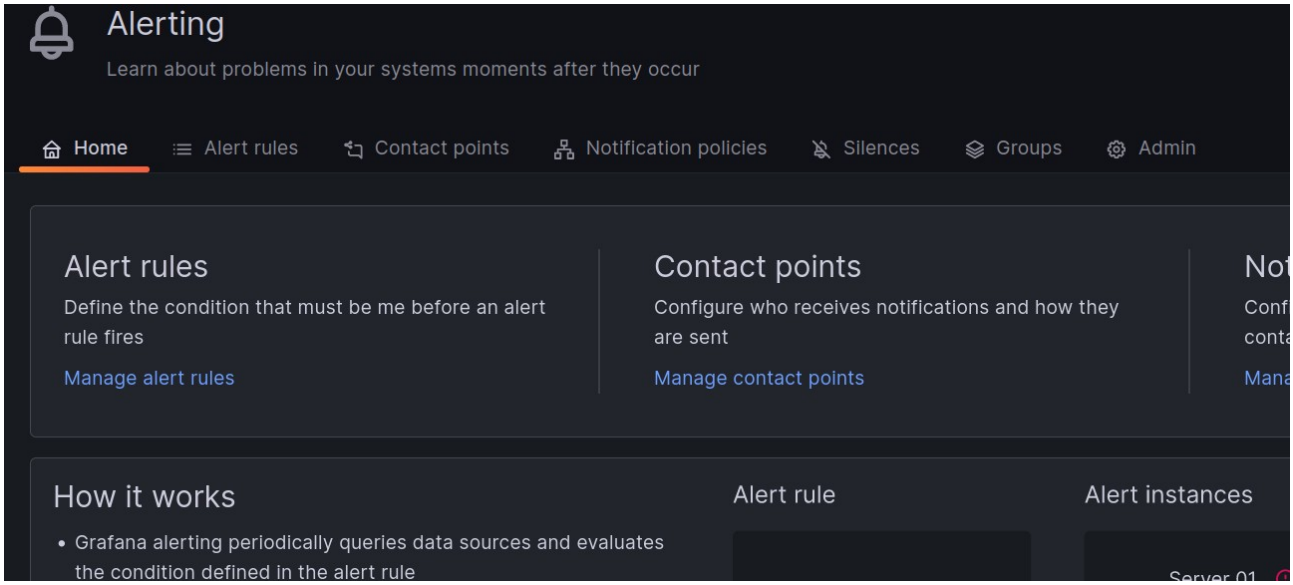
Y el segundo panel que mencionaré será similar al de la CPU pero con la memoria:



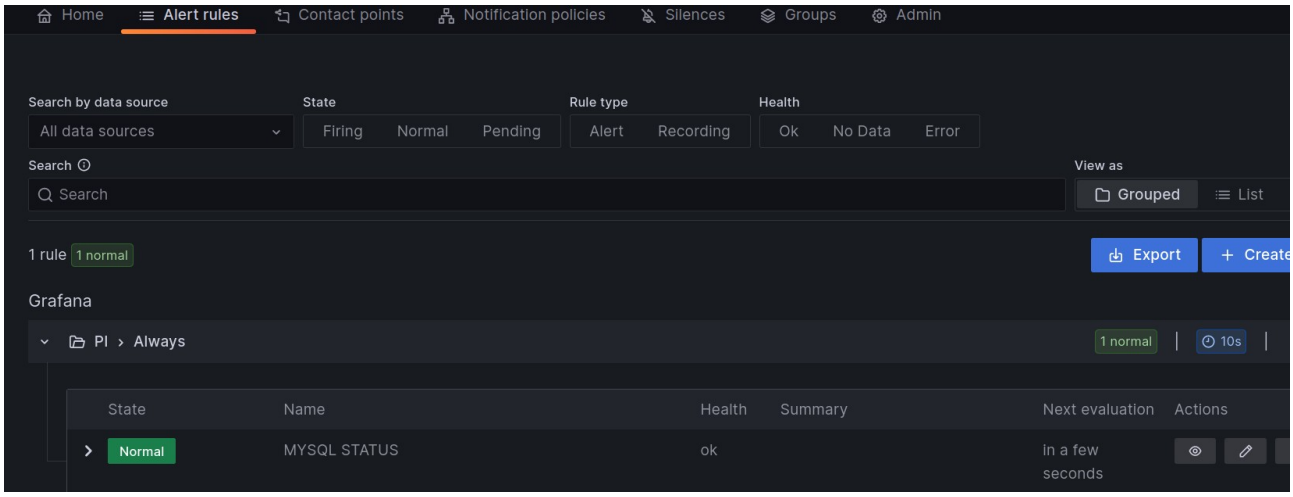
Al acceder, podemos explorar las fuentes de datos que tengamos añadidas. Será en las demostraciones donde profundicemos algo más en esta opción, y un poco más adelante donde explicaré las *Data Sources* o fuentes de datos.



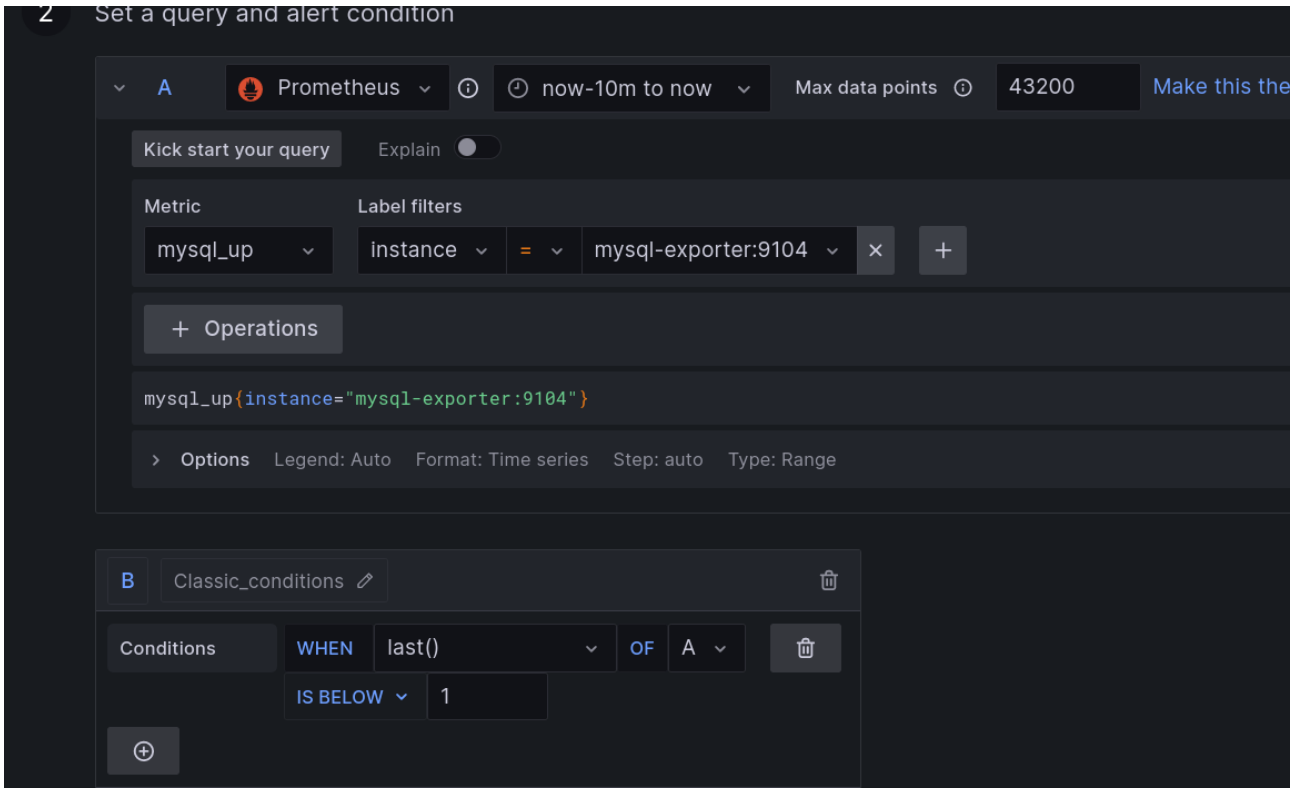
Continuamos con la parte del menú correspondiente a las alertas:



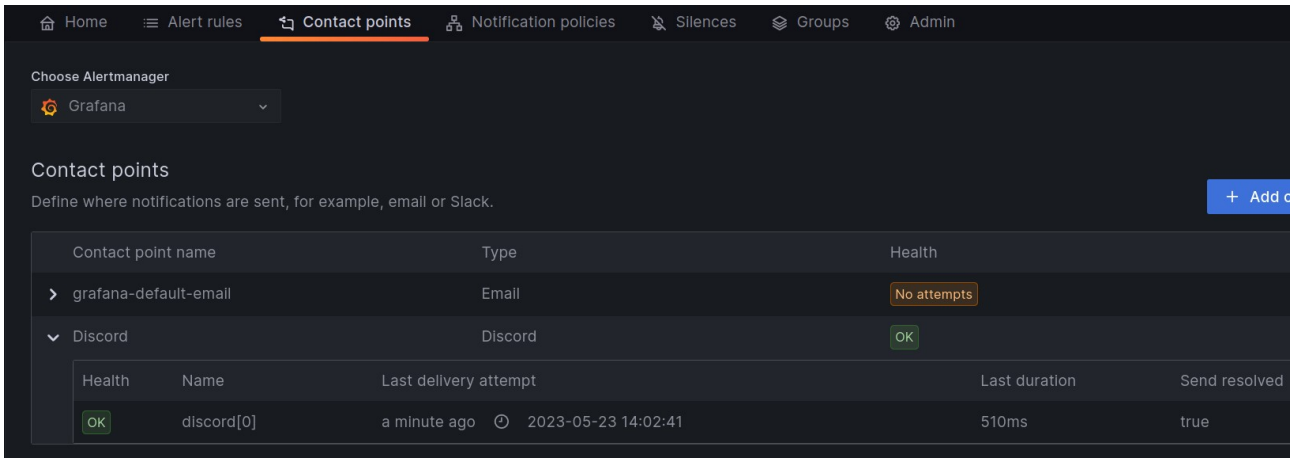
Seleccionaré la pestaña “*Alert Rules*” donde encontraremos las alertas:



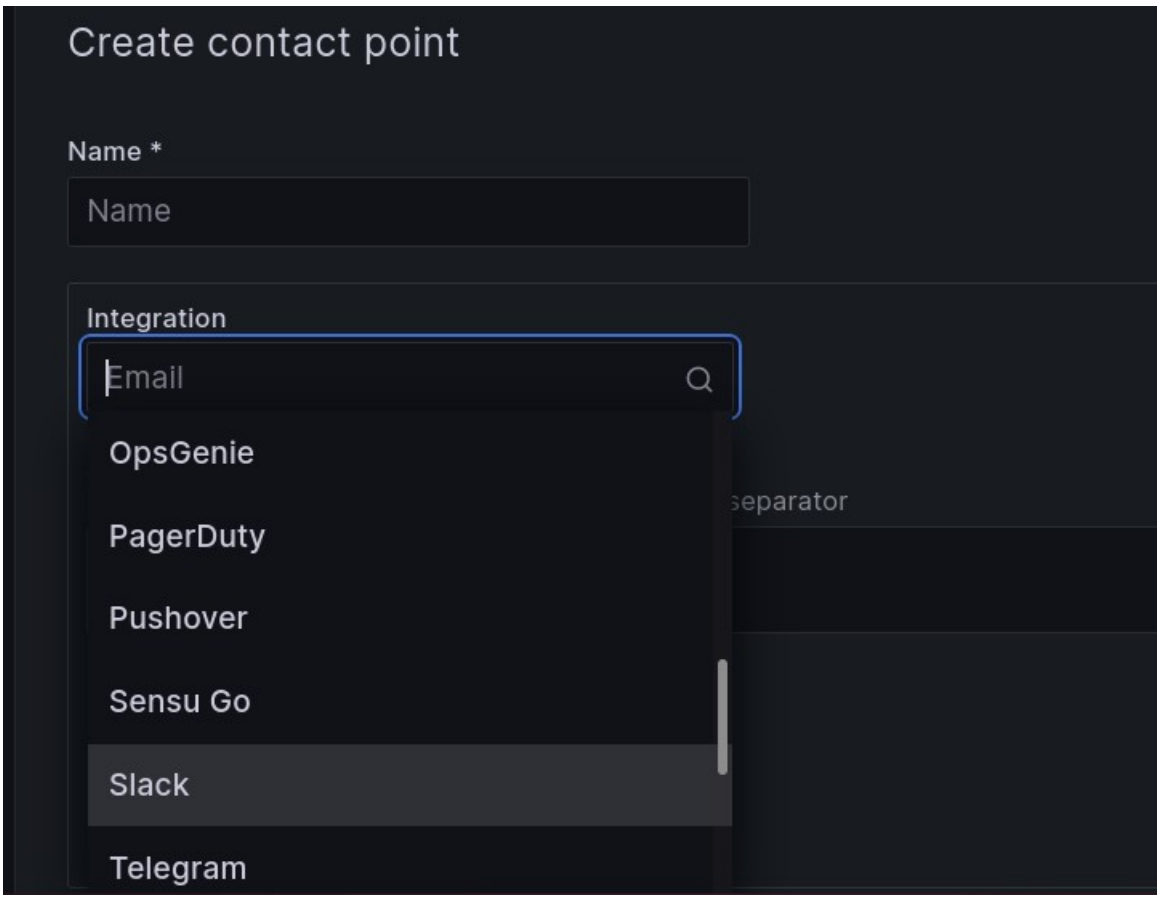
Como podemos comprobar, he añadido una alerta que nos avise cuando la base de datos no se encuentre operativa:



La alerta enviará una notificación por Discord, como podemos ver en “*Contact points*” :



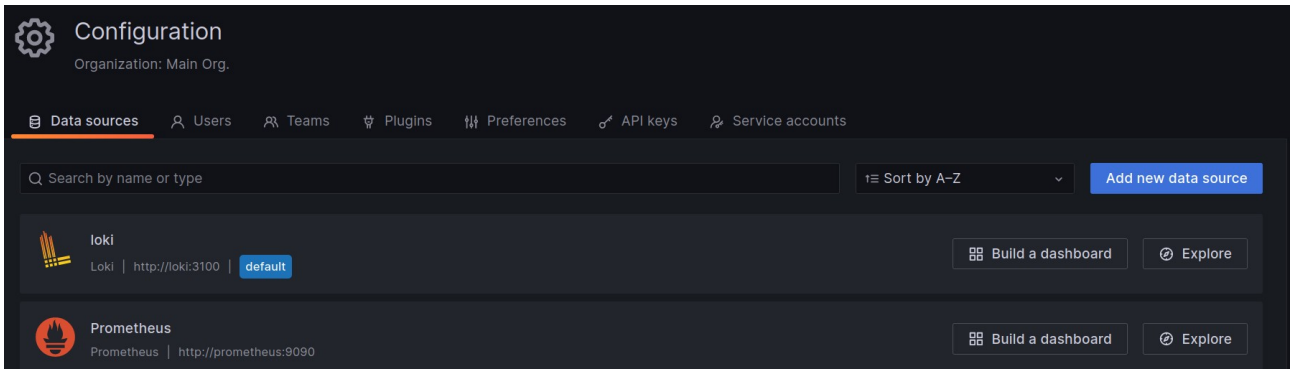
En este caso, he optado por usar *Discord* debido a que he estado en entornos reales donde se usa esta plataforma, pero podemos usar múltiples opciones:



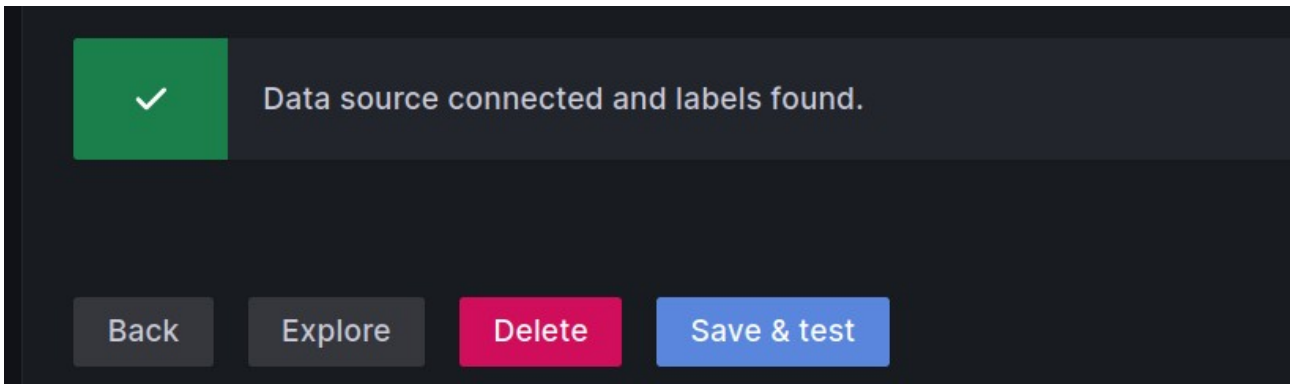
Entre ellas, podría destacar *Slack*, *Telegram*, e-*mail* ...

El funcionamiento de esta alerta se hará en el apartado correspondiente.

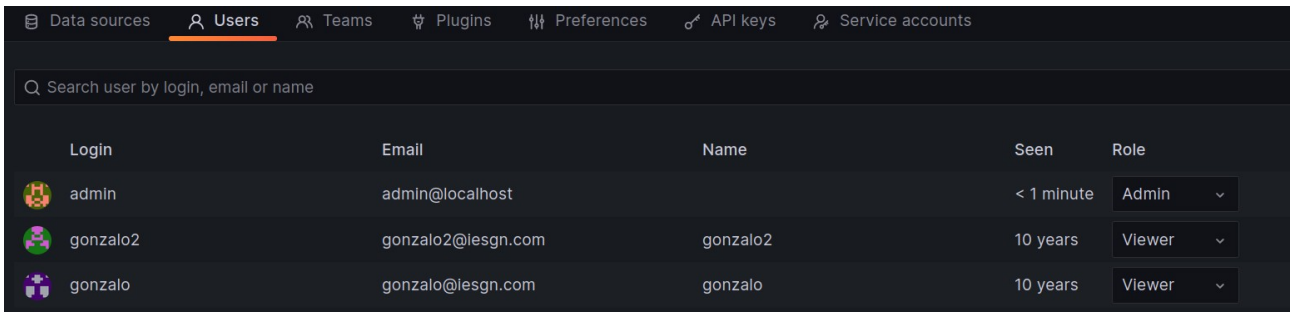
Continuando con Grafana, encontramos la parte referente a la configuración:



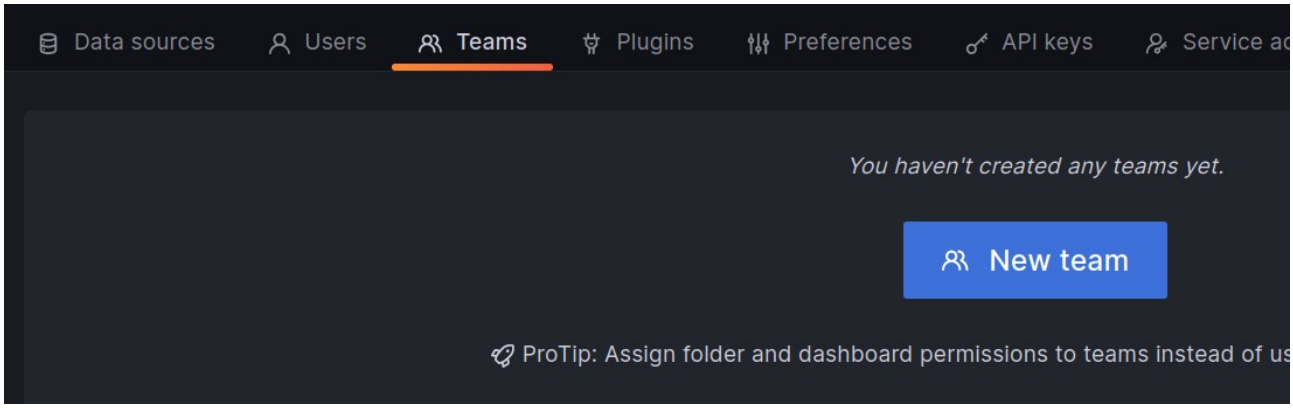
En la primera pestaña, encontramos los *Data Sources*, en los cuales podemos comprobar que Grafana está integrado con *Loki* y *Prometheus*. Si queremos comprobar la correcta integración, podemos seleccionar la fuente de datos y presionar "*Save & test*":



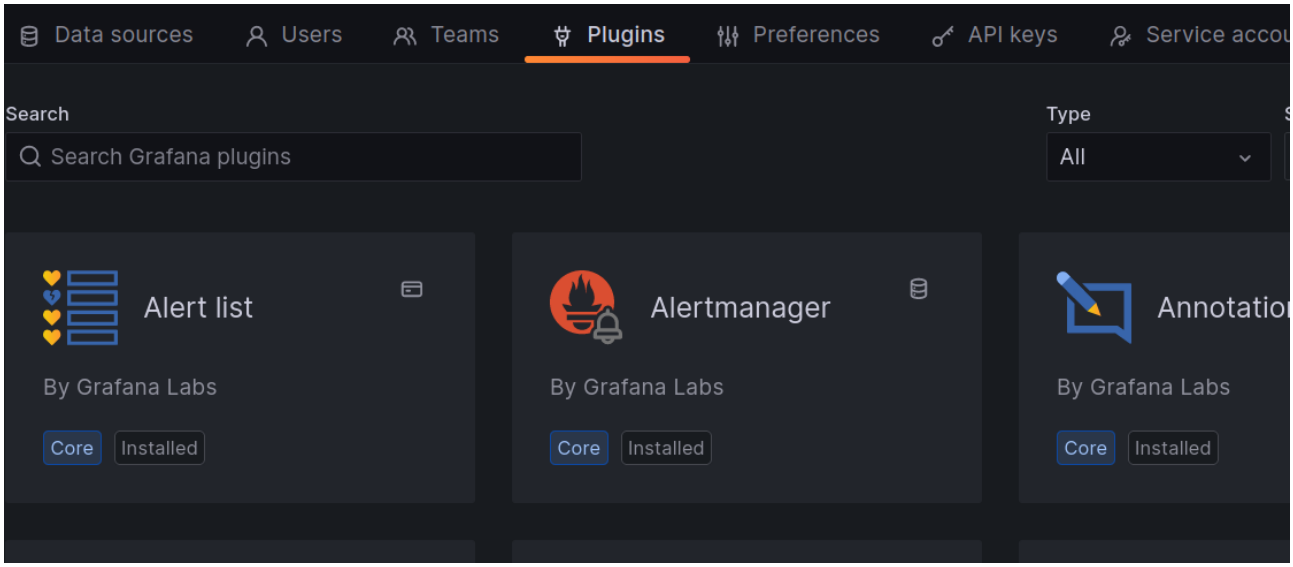
Pasando a la siguiente pestaña, vemos los usuarios:



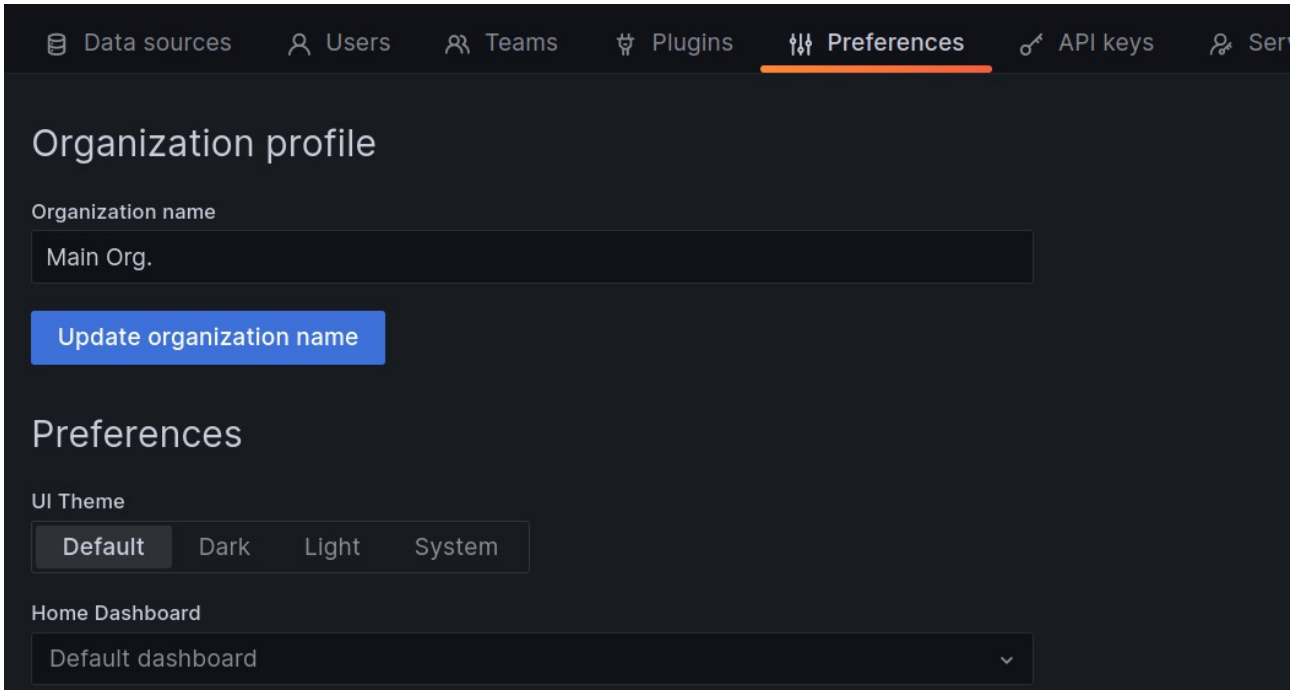
Continuamos con los Teams, que no he usado en este proyecto:



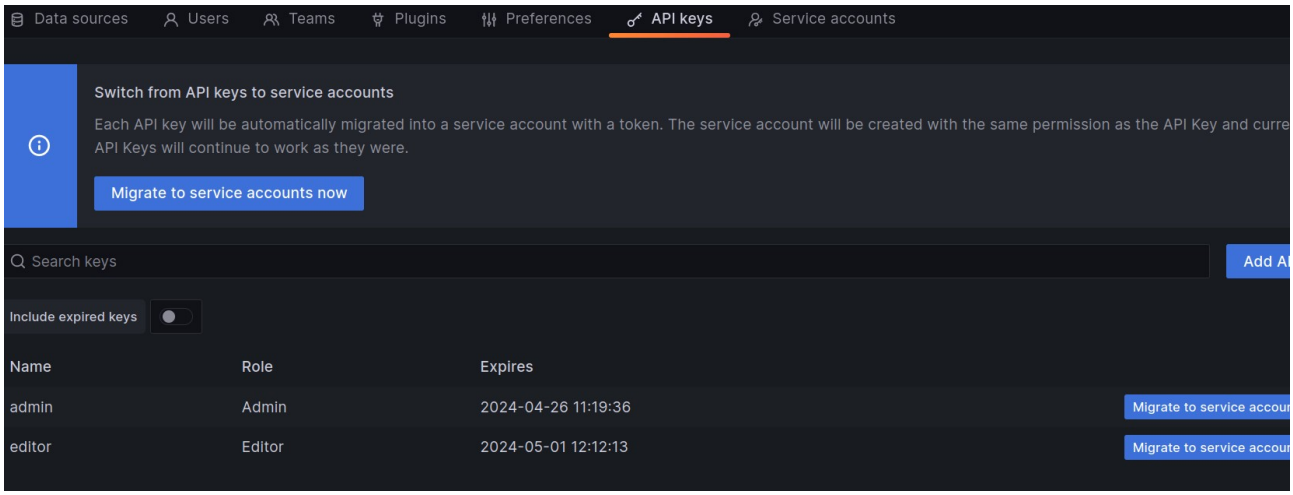
En la pestaña *Plugins*, veremos los que tenemos instalados:



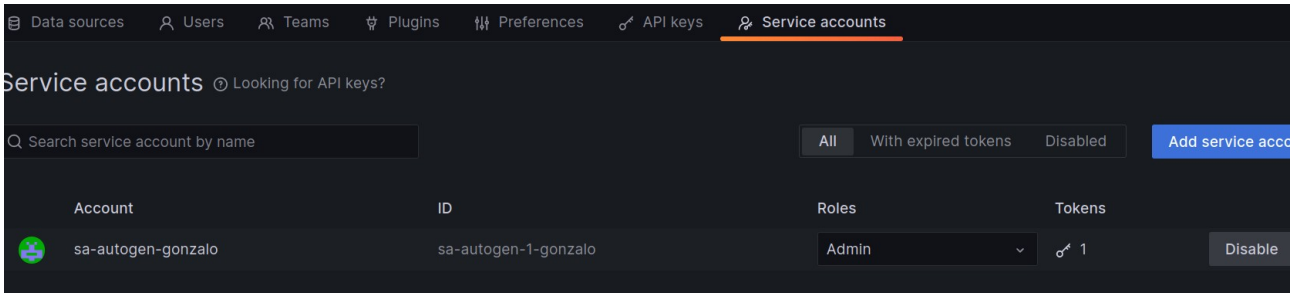
Continuamos con las preferencias de la organización, aspecto que no he profundizado tampoco:



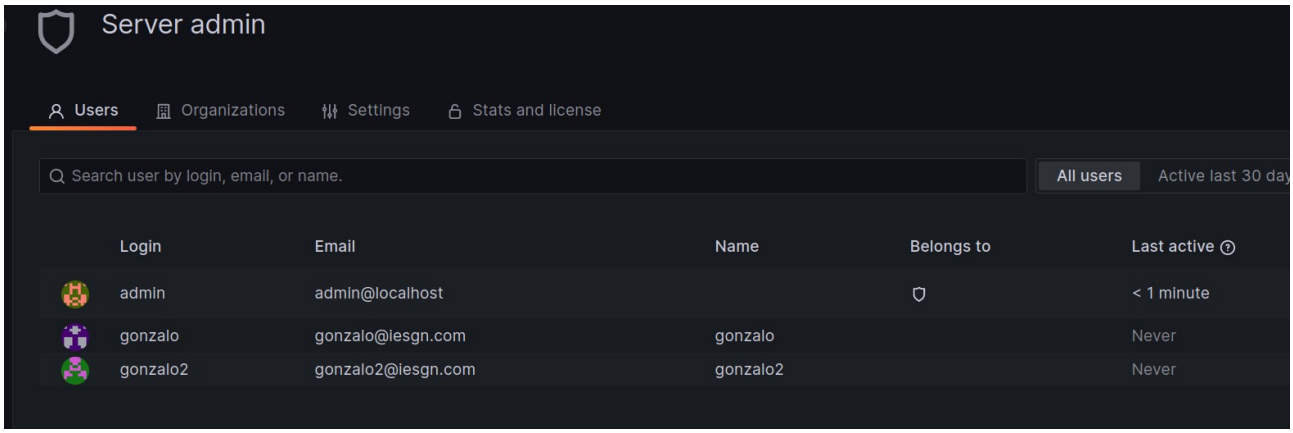
En la siguiente pestaña veremos las claves para conectar con la API:



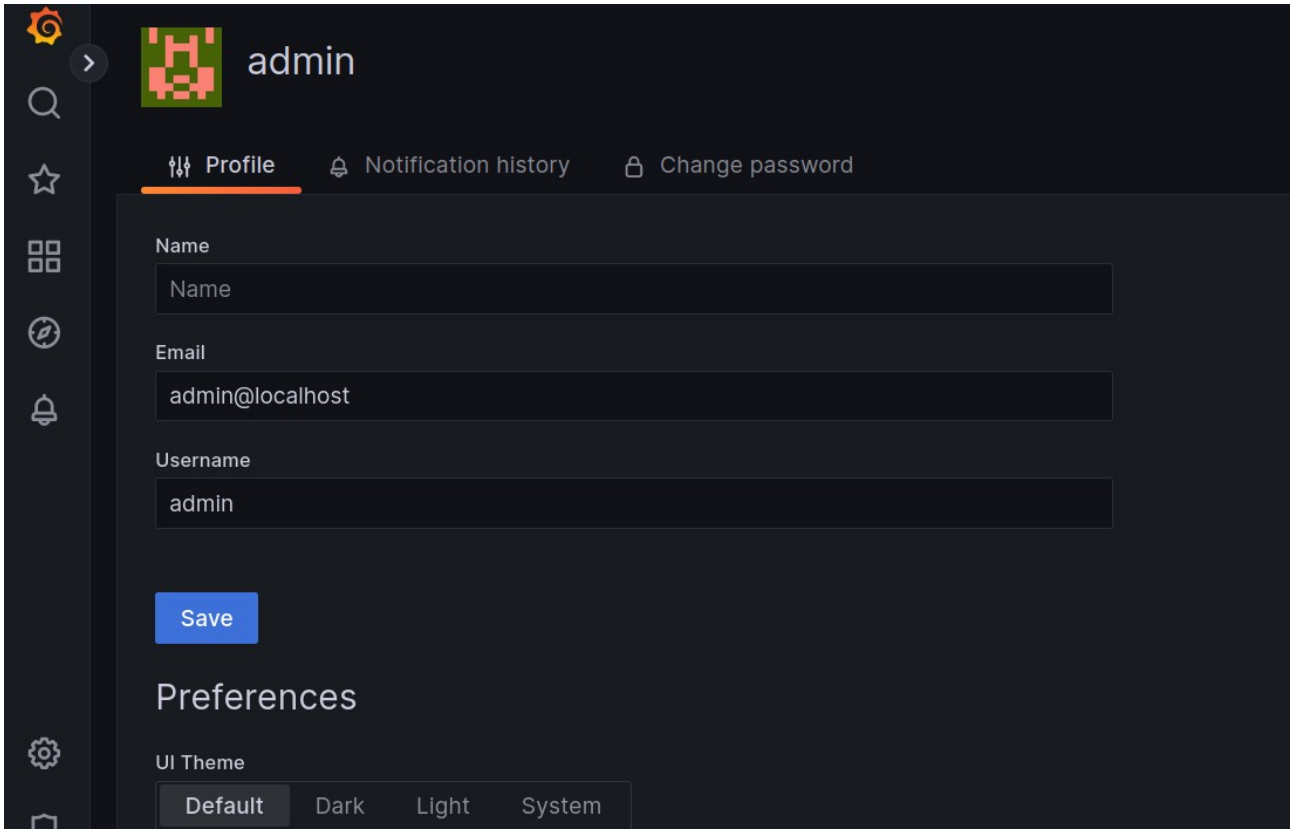
Por último en la configuracion, nos encontramos con “*Service accounts*” :



El siguiente apartado del menú es "*Server admin*", en el cual tenemos la organización de usuarios, organizaciones y ajustes relacionados, por lo que no profundizaré mucho en este aspecto:



El último apartado del menú que revisaré será en el cual podemos modificar información sobre nuestro usuario, por lo que no profundizaré más en él:



4.3 Prácticas y demostraciones a realizar

4.3.1.HAProxy:

La demostración consistirá en realizar múltiples peticiones usando el comando *ab*, y comprobaremos que se está balanceando la carga entre ambos drupal.

Para ello, ejecutaremos el siguiente comando en el *host*:

```
ab -n 1000  -k http://localhost:8080/
```

```
Server Software:        Apache/2.4.56
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        16439 bytes

Concurrency Level:      1
Time taken for tests:   4.462 seconds
Complete requests:      1000
Failed requests:        0
Keep-Alive requests:    0
Total transferred:      16947000 bytes
HTML transferred:       16439000 bytes
Requests per second:    224.13 [#/sec] (mean)
Time per request:       4.462 [ms] (mean)
Time per request:       4.462 [ms] (mean, across all concurrent requests)
Transfer rate:          3709.38 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:     4    4   2.7      4      77
Waiting:        3    4   2.6      4      77
Total:          4    4   2.7      4      78

Percentage of the requests served within a certain time (ms)
  50%      4
  66%      4
  75%      4
  80%      4
  90%      5
  95%      7
  98%      9
  99%     11
 100%     78 (longest request)
```

Una vez ejecutado, podemos ir a la página en la cual visualizamos el reporte de HAProxy:

localhost:8080/haproxy_status

Frontend			5	247	-	1	1	2 000	1 001			106 000

drupal

		Queue			Session rate			Sessions						B
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In
☐	drupal	0	0	-	2	123		0	1	500	500	500	1s	53 000
☐	drupal2	0	0	-	2	124		0	1	500	500	500	1s	53 000
	Backend	0	0		5	247		1	1	200	1 001	1 000	0s	106 000

Choose the action to perform on the checked servers : Apply

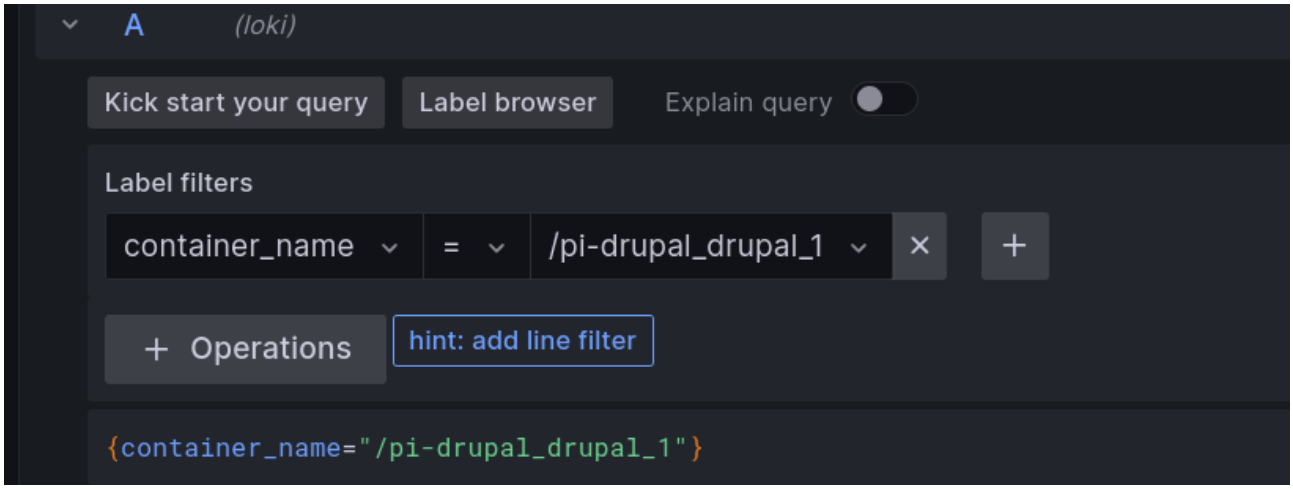
Como se puede comprobar, se ha repartido la carga entre ambos drupal, por lo que el balanceo tipo roundrobin está funcionando perfectamente.

4.3.2 Recolección de logs:

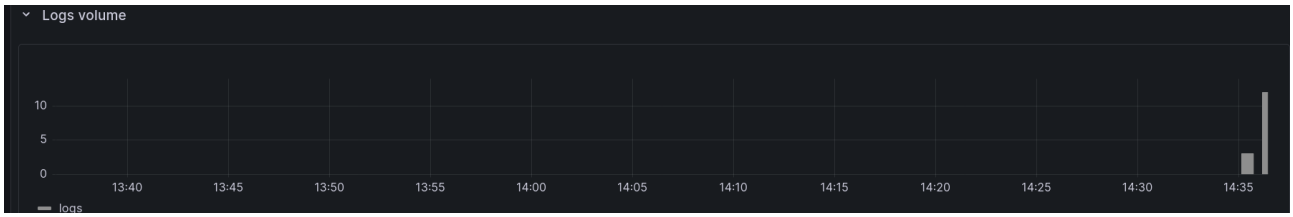
Accediendo a la parte "*Explore*" de Grafana, podemos explorar entre los distintos *Datasources* que tenemos vinculados en nuestro Grafana, por lo que si seleccionamos el correspondiente a *Loki*, podemos comprobar que se están recolectando los logs.

Para comprobar esto, podemos acceder a la página para que se guarde un registro de la acción.

Una vez realizado esto, accedemos a la parte anteriormente mencionada de Grafana y seleccionamos el contenedor al que queremos ver sus logs:



Ejecutaremos la consulta y recibiremos por pantalla tanto los logs, como un gráfico con los picos de éstos:



Comprobando los logs, podemos distinguir claramente los logs que se han generado mediante el comando *ab* y los que se han generado visitando la página:

4.3.3 Filtrado de logs:

En esta breve demostraremos el lenguaje LogQL, el cual es el lenguaje de consultas de Grafana Loki.

Podemos filtrar por ejemplo, por navegadores Mozilla:

{container_name="/pi-drupal_drupal_1"} |= `Mozilla`

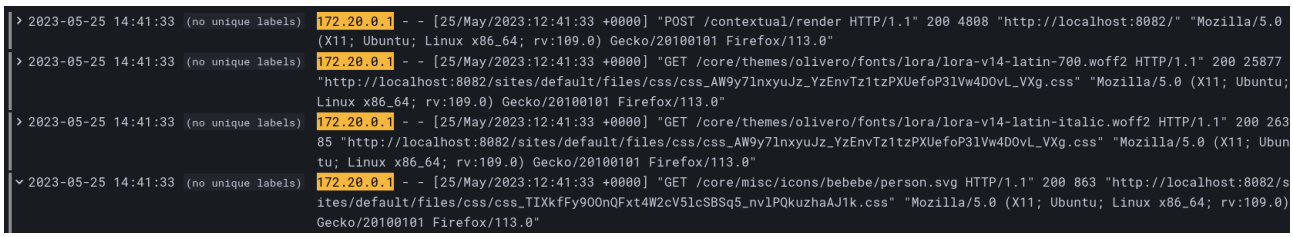
Como podemos comprobar, se ha filtrado por los logs que incluyen la palabra *Mozilla*.

Ahora, podríamos filtrar por ejemplo por los logs que se han creado con el *ApacheBench*:

{container_name="/pi-drupal_drupal_1"} |= `ApacheBench`

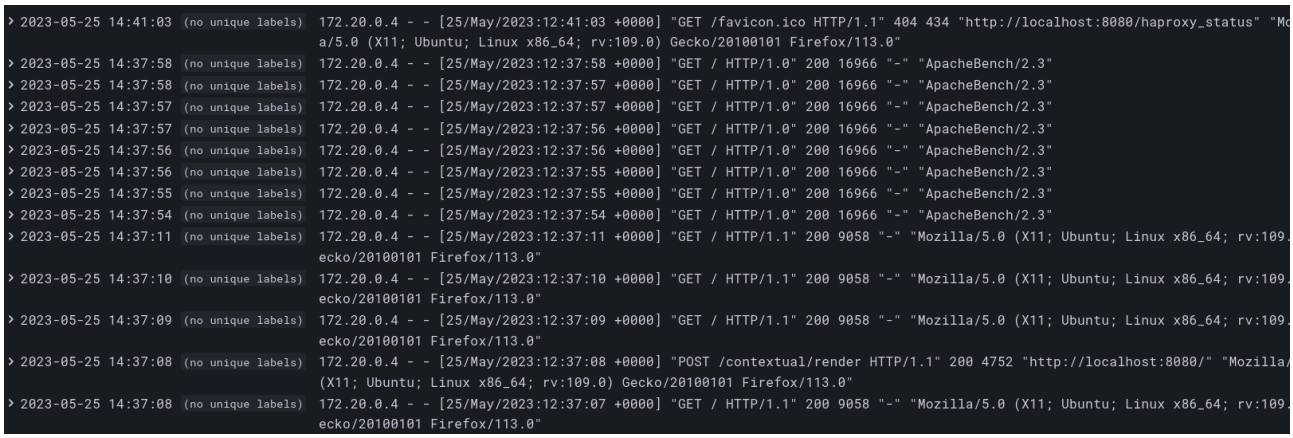
Las opciones de filtrado son variadas, podríamos hacerlo también por IP:

{container_name="/pi-drupal_drupal_1"} |= `172.20.0.1`



Por último, también podríamos ejecutar consultas para que nos devuelva los logs que no tienen coincidencia con la cadena a buscar, y ésto lo podríamos hacer por ejemplo, excluyendo a nuestra propia IP:

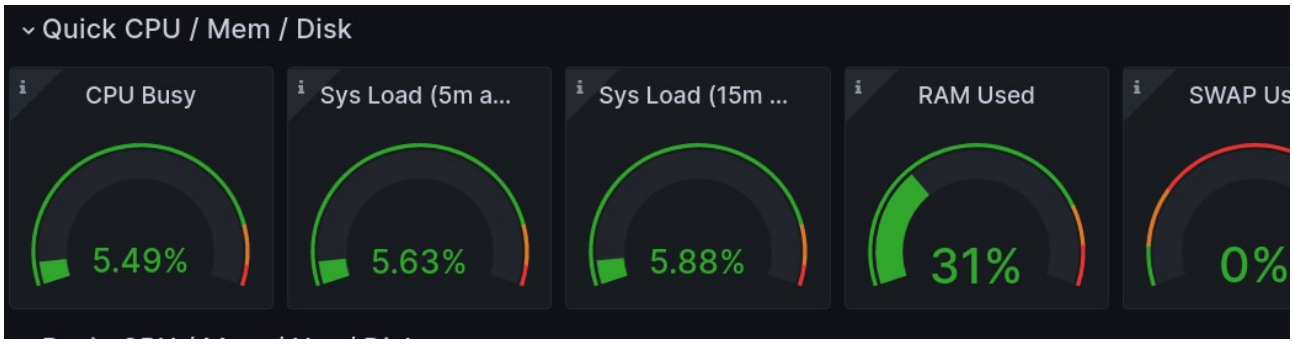
{container_name="/pi-drupal_drupal_1"} != `172.20.0.1`



4.3.4 Estrés de RAM:

Haremos un estŕes de RAM y comprobaremos a tiempo real que las métricas se están recogiendo bien, puesto que debería subir la carga de RAM en el dashboard de *Node Exporter.*

Comenzamos comprobando a qué porcentaje se encuentra la RAM durante los últimos 5 segundos:



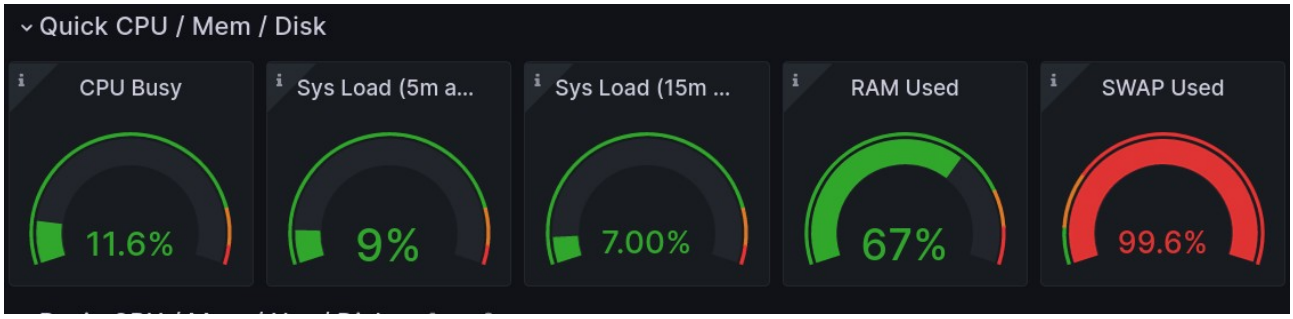
Comprobando la captura anterior, vemos que se encuentra a un 31%.

Para testear la RAM, usaré el comando *memtester*:

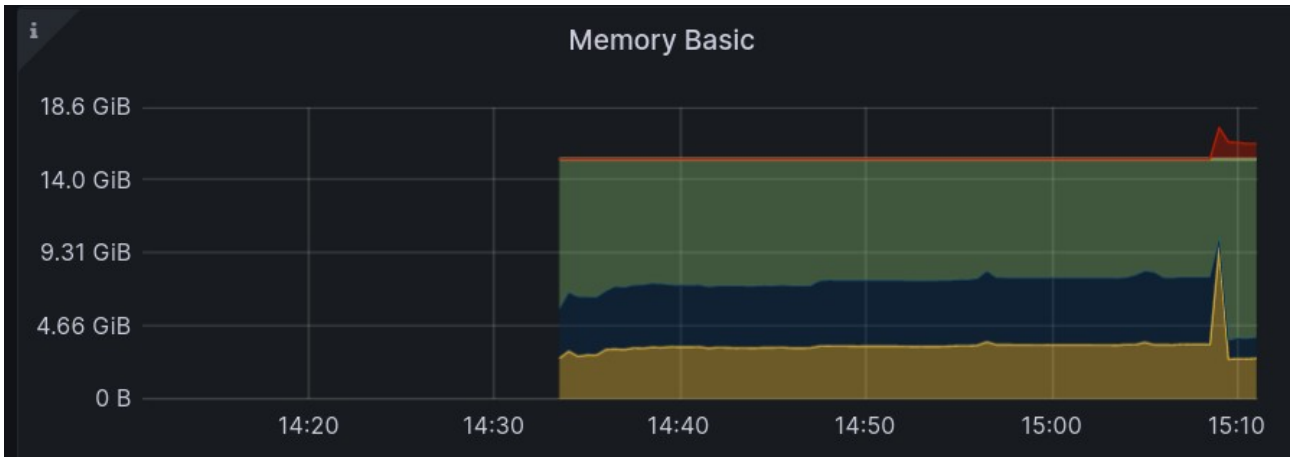
sudo memtester 16G 1

```
gmarin@ubuntu-gmarin:~/github/pi-drupal$ sudo memtester 16G 1
memtester version 4.5.1 (64-bit)
Copyright (C) 2001-2020 Charles Cazabon.
Licensed under the GNU General Public License version 2 (only).

pagesize is 4096
pagesizemask is 0xfffffffffffff000
want 16384MB (17179869184 bytes)
got  16384MB (17179869184 bytes), trying mlock ...
Terminado (killed)
```

Una vez ejecutado, volvemos a comprobar la RAM y la SWAP:



También podemos observar este pico de consumo mediante los gráficos:



4.3.5 Estrés de CPU:

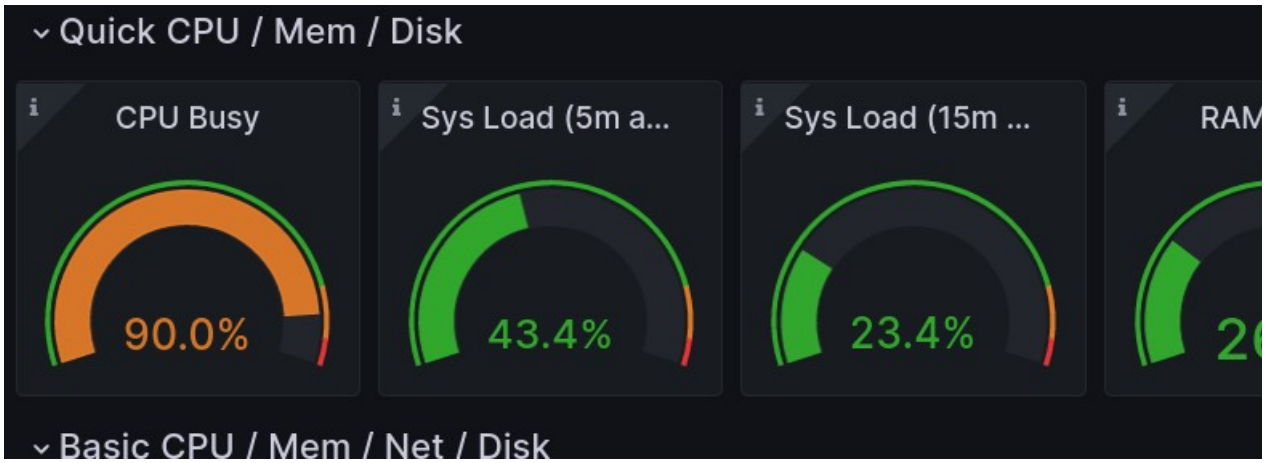
Al igual que con la RAM, haremos una prueba de estrés de la CPU y comprobaremos sus métricas a tiempo real, comprobando que efectivamente podemos ver como aumentan éstas.

Para esta demostración, usaré el comando *stress-ng*:

sudo stress-ng --cpu 7 -v

```
gmarin@ubuntu-gmarin:~/github/pi-drupal$ sudo stress-ng --cpu 7 -v
stress-ng: debug: [13583] stress-ng 0.13.12
stress-ng: debug: [13583] system: Linux ubuntu-gmarin 5.19.0-40-generic #41~22.
4.1-Ubuntu SMP PREEMPT_DYNAMIC Fri Mar 31 16:00:14 UTC 2 x86_64
stress-ng: debug: [13583] RAM total: 15.3G, RAM free: 10.6G, swap free: 1.1G
stress-ng: debug: [13583] 8 processors online, 8 processors configured
stress-ng: info:  [13583] defaulting to a 86400 second (1 day, 0.00 secs) run p
r stressor
stress-ng: info:  [13583] dispatching hogs: 7 cpu
stress-ng: debug: [13583] cache allocate: shared cache buffer size: 6144K
stress-ng: debug: [13583] starting stressors
stress-ng: debug: [13584] stress-ng-cpu: started [13584] (instance 0)
stress-ng: debug: [13584] stress-ng-cpu using method 'all'
stress-ng: debug: [13585] stress-ng-cpu: started [13585] (instance 1)
stress-ng: debug: [13585] stress-ng-cpu using method 'all'
stress-ng: debug: [13586] stress-ng-cpu: started [13586] (instance 2)
```

Tras poco tiempo ejecutando el comando, podemos ver en el gráfico cómo va subiendo la CPU:



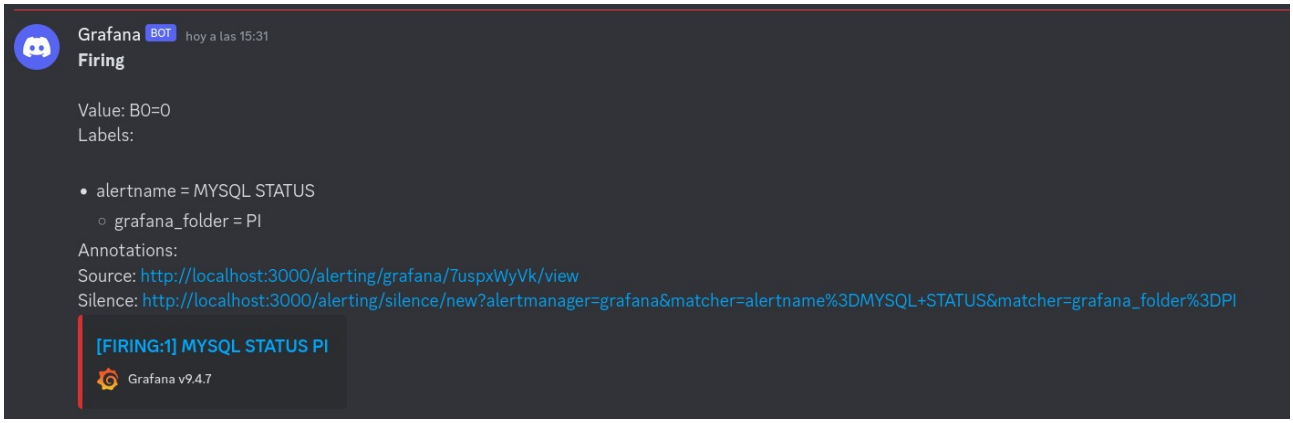
4.3.6 Alertas:

En esta demostración, trabajaremos con la alerta configurada, y para que se active, debemos bajar el contenedor de la base de datos y para ello ejecutaremos el siguiente comando:

docker-compose stop mariadb

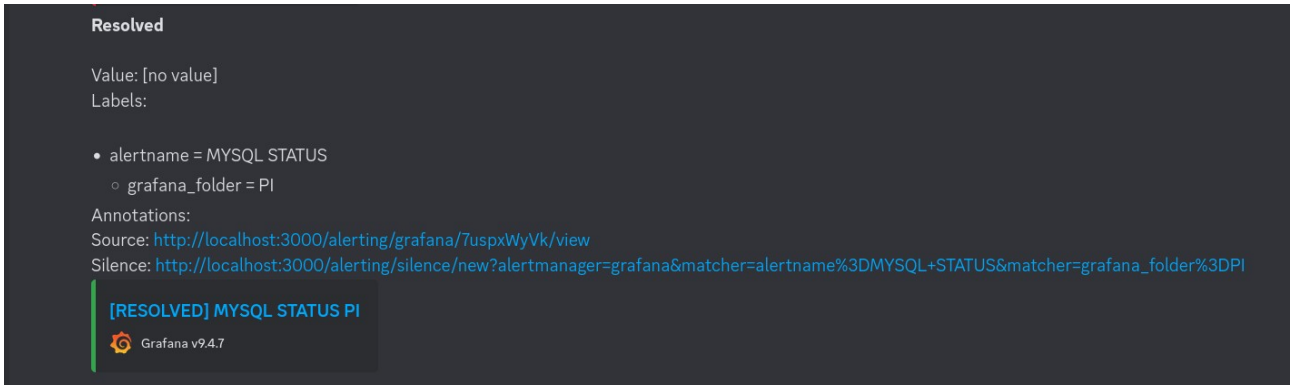
```
gmarin@ubuntu-gmarin:~/github/pi-drupal$ docker-compose stop mariadb
Stopping mariadb ... done
```

Accedemos a nuestro canal de Discord de alertas y comprobaremos que nos ha notificado correctamente:



Ahora, volveremos a levantar el contenedor y nos llegará otra notificación en la que nos informará de que el error ha sido subsanado:

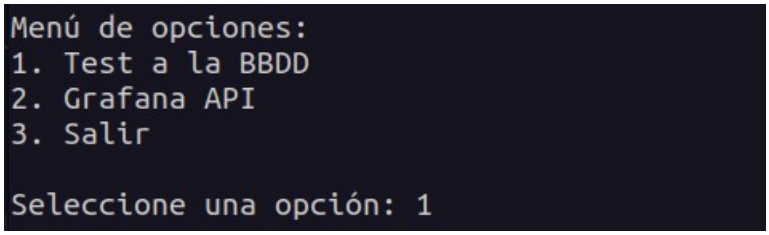
docker-compose start mariadb



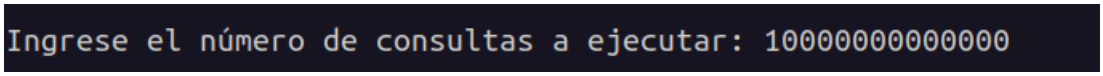
4.3.7 Monitorización de BBDD:

A través de mi programa Python someteremos al servidor MariaDB a un elevado número de consultas, que se verá reflejado en el dashboard del mismo.

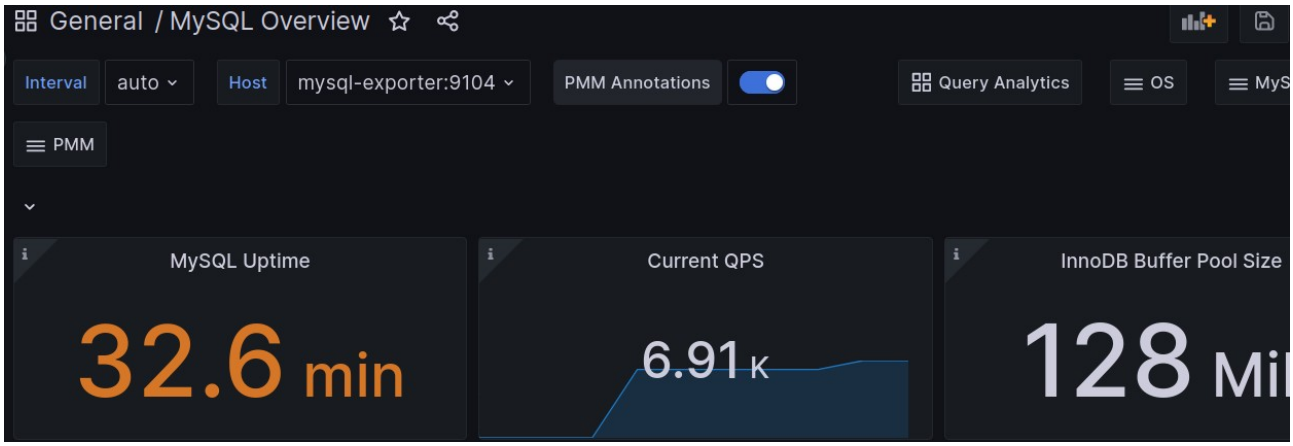
Para ello, ejecutaremos el programa python seleccionando la primera opción:



Ahora, ejecutaremos un número elevado de consultas:



Podemos ir al dashboard y comprobar que han subido las consultas por segundo:



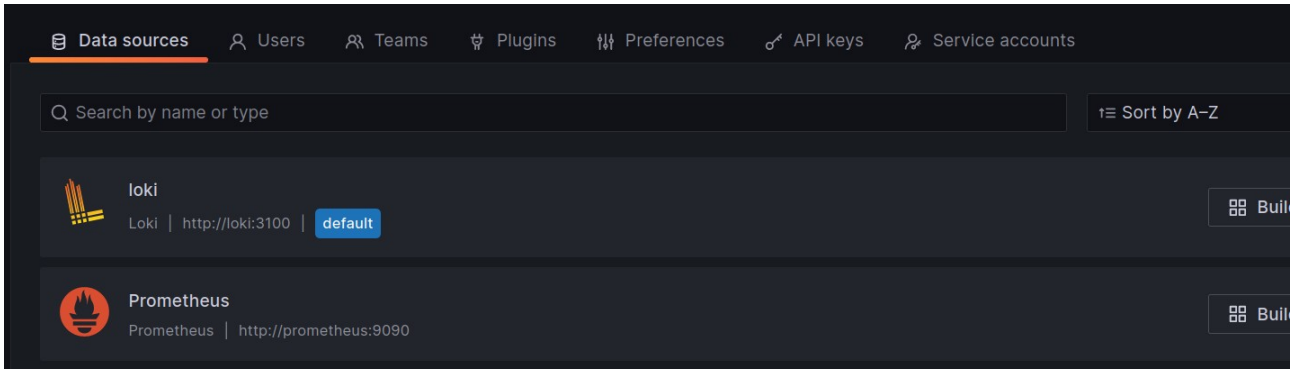
4.3.8 Programa Python:

Por último, veremos varios ejemplos de uso de la API de grafana usando el programa python, y para comprobar que funciona correctamente, comprobaremos la información con la interfaz gráfica de Grafana.

Comenzamos viendo la primera opción, con la que podremos ver todos los dashboard añadidos:

```
Seleccione una opción: 1

Nombres y URLs de los datasources:
loki: http://localhost:3100
Prometheus: http://localhost:9090
```

Comprobamos en Grafana:



Con la segunda opción, veremos información del usuario con el que nos conectamos:

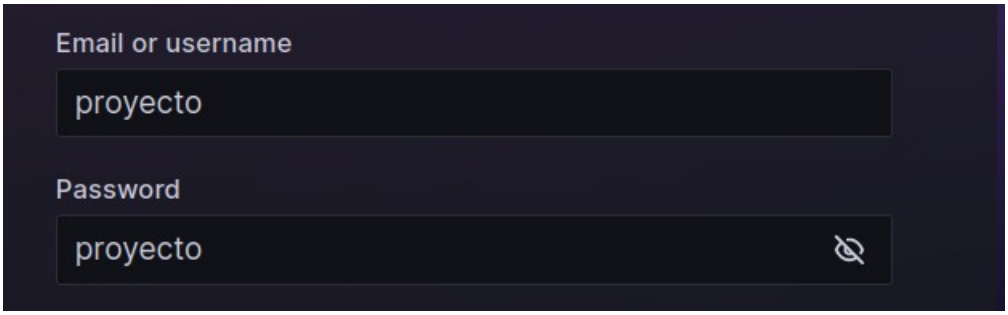
```
Seleccione una opción: 2

Nombre del usuario:  admin
Fecha de creación:  2023-04-25T12:13:03Z
¿Es admin?: Verdadero
```

Con la tercera opción, añadiremos un usuario:

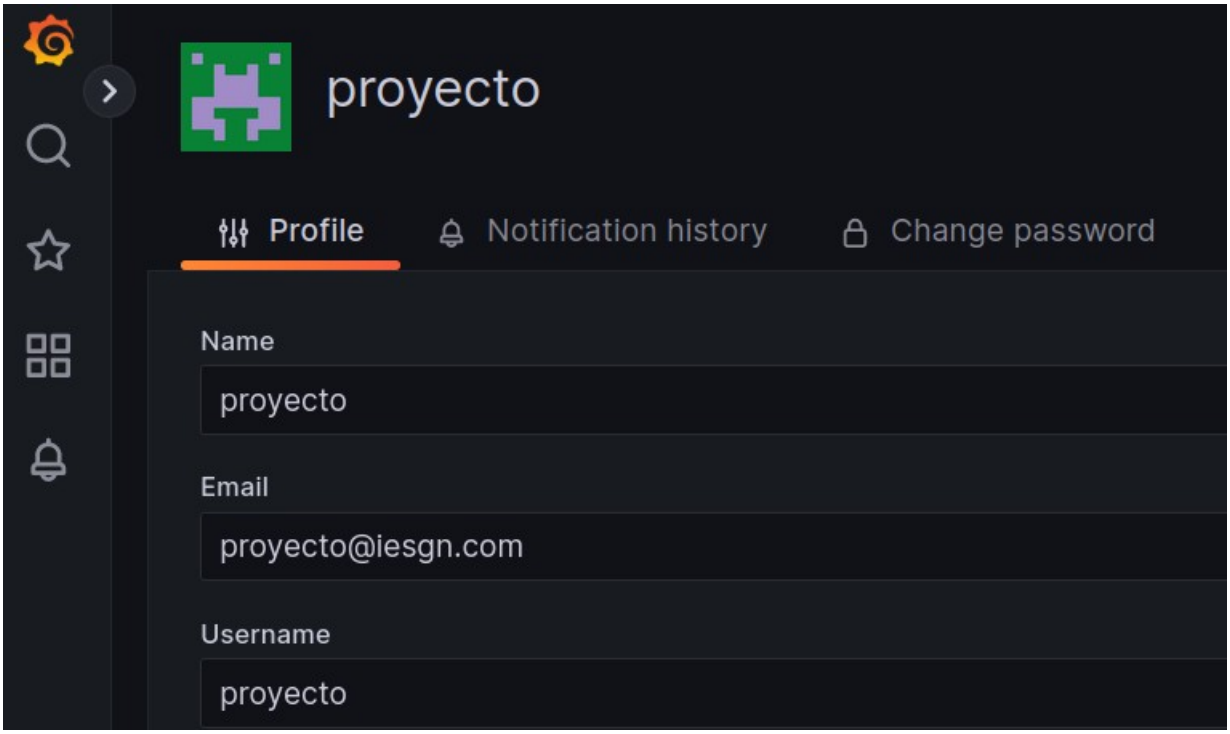
```
Seleccione una opción: 3

Introduzca el nombre del usuario: proyecto
Introduzca el email del usuario: proyecto@iesgn.com
Introduzca el nombre de login del usuario: proyecto
Introduzca la contraseña del usuario: proyecto
```

Podemos comprobar que se ha añadido correctamente:

También podríamos iniciar sesión con el usuario:



Como podemos comprobar, se ha iniciado sesión correctamente:



Continuando con la cuarta opción del programa, podemos comprobar información sobré algún usuario, y lo haremos del que acabamos de crear:

```
Seleccione una opción: 4

Introduzca el nombre del usuario: proyecto
Nombre del usuario:  proyecto
Fecha de creación:  2023-05-25T14:16:50Z
ID del usuario:  6
```

Continuando con la quinta, veremos la información referente a algún dashboard:

```
Seleccione una opción: 5

Dashboards disponibles:
1. MySQL
2. Node Exporter
3. Salir
```

Selecciono el referente a la BBDD:

```
Nombre del dashboard:  MySQL Overview
Descripción del dashboard:  Dashboard from Percona Monitoring and Management pro
ject.
UID del dashboard: MQWgroiiz
Fecha de creación:  2023-04-25T12:13:05Z
¿Se puede editar el dashboard?, True
¿Cada cuanto se refresca el panel? 1m
Fichero JSON del dashboard? mysql.json
```

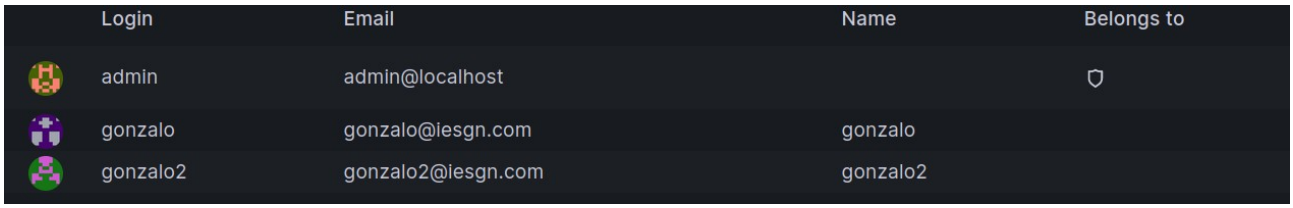
Continuamos con el *Node Exporter*:

```
Seleccione una opción: 2

Nombre del dashboard:  Node Exporter Full
UID del dashboard: rYdddlPWk
Fecha de creación:  2023-04-25T12:13:05Z
¿Se puede editar el dashboard?, True
Fichero JSON del dashboard: dashboard.json
```

Con la sexta opción del programa, podremos eliminar un usuario:

```
Seleccione una opción: 6

Introduzca el nombre del usuario: proyecto
¿Quieres borrar el usuario? (S/N): S
Usuario eliminado correctamente.
```

Como se puede ver, hemos eliminado el usuario creado anteriormente, así que vamos a comprobarlo en el panel de grafana:

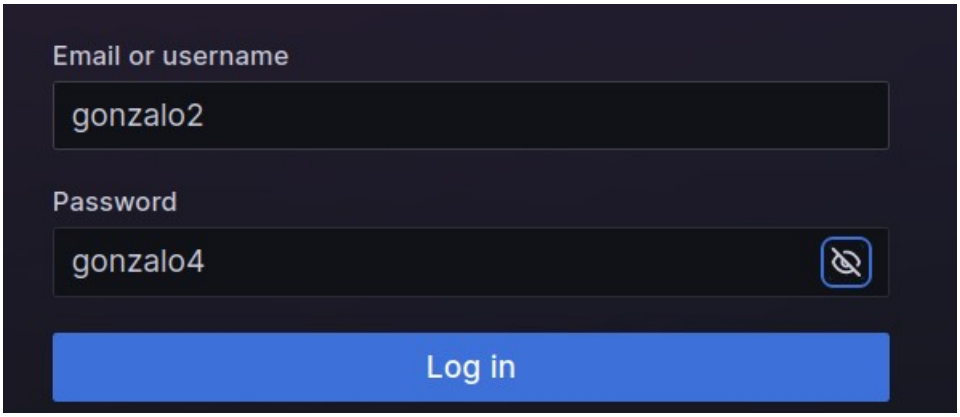


El usuario *proyecto* ya no se encuentra disponible.

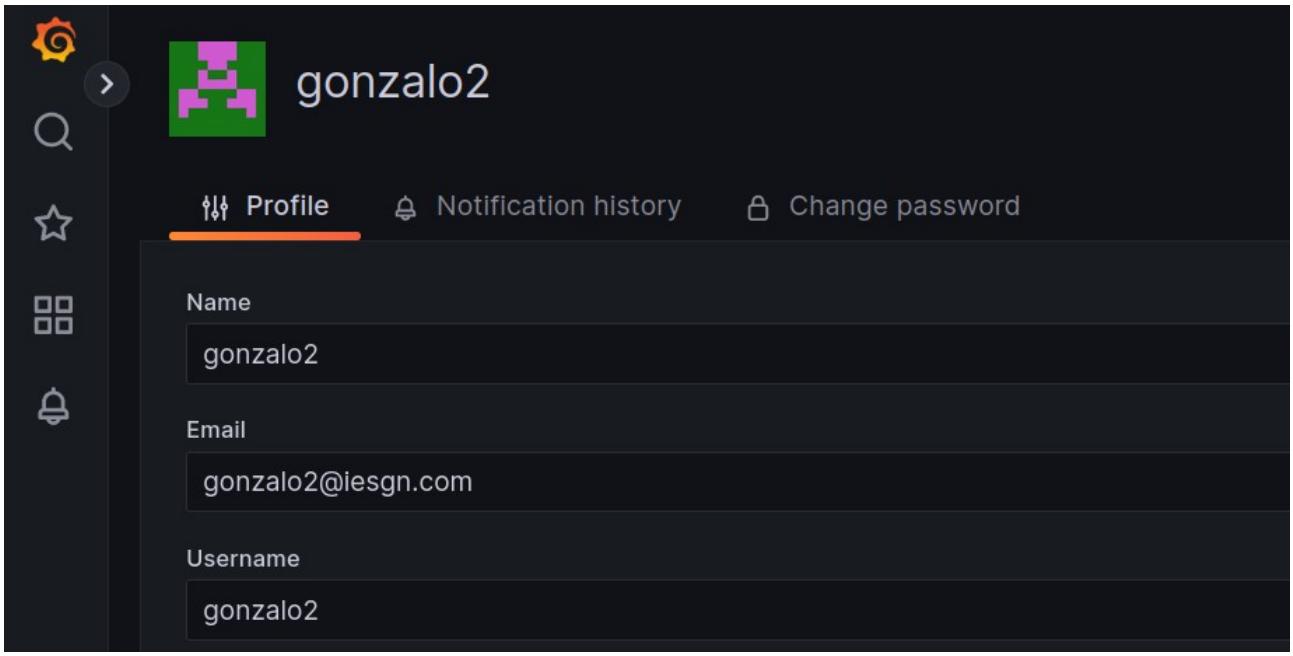
Con la séptima opción del programa podremos modificar la contraseña de un usuario:

```
Seleccione una opción: 7

Introduzca el nombre del usuario: gonzalo2
Introduzca la nueva contraseña: gonzalo4
¿Quieres modificar la contraseña? (S/N): S
Contraseña modificada correctamente.
```

En teoría la contraseña ha cambiado, por lo que iremos a comprobarlo:



La contraseña se ha modificado correctamente:



En la octava opción podemos ver las carpetas disponibles:

```
Seleccione una opción: 8

Carpetas disponibles:
Nombre de la carpeta:  PI
UID de la carpeta:  ln65xZs4z
---------------------------------
```

Ahora, con la novena opción crearemos una nueva carpeta:

```
Seleccione una opción: 9

Introduzca el nombre de la carpeta: pruebacarpeta
Carpeta creada correctamente.
```

Podemos volver a la opción 8 y comprobar que se añade:

```
Carpetas disponibles:
Nombre de la carpeta:  PI
UID de la carpeta:  ln65xZs4z
---------------------------------
Nombre de la carpeta:  pruebacarpeta
UID de la carpeta:  dpx3acw4z
---------------------------------
```

Como se puede ver en la captura, aparece la nueva carpeta.

En esa opción del programa solamente aparecerán carpetas creadas, es decir, la carpeta *General* no aparecerá.

Con la penúltima opción, eliminaremos la carpeta que acabamos de crear:

```
Seleccione una opción: 10

Introduzca el nombre de la carpeta: pruebacarpeta
¿Quieres borrar la carpeta? (S/N): S
Carpeta eliminada correctamente.
```

Comprobamos las carpetas a través del programa python:

```
Seleccione una opción: 8

Carpetas disponibles:
Nombre de la carpeta:  PI
UID de la carpeta:  ln65xZs4z
---------------------------------
```

Por último, podemos ver las organizaciones creadas:

```
Seleccione una opción: 11

Nombre de la organización:  Main Org.
ID de la organización:  1
```

En este caso, solamente hay una organización como podemos ver.

5. Conclusiones y propuestas para seguir trabajando sobre el tema

Finalizando el proyecto integrado, podría concluir que monitorear un escenario es una tarea imprescindible para cualquier entorno en producción, ya que debemos controlar en cada momento nuestro escenario, puesto que una caída o fallo del mismo podría desembocar en grandes pérdidas para la empresa. Destacaría el uso de alertas, puesto que esta herramienta puede hacernos evitar errores (por ejemplo, avisándonos cuando nuestros servidores están a punto de llegar al límite de cualquier recurso y ampliarlo) o que éstos sean subsanados lo más rápido posible (por ejemplo, una caída de la base de datos).

Las propuestas para seguir trabajando sobre el tema son amplias, puesto que siempre puede surgir alguna necesidad de tener controlado cierto servicio o parte de nuestro escenario. Podríamos ampliar el tema usando:

- **Software para análisis de datos**: Mediante estas herramientas, podríamos comprobar desde qué navegadores visitan nuestro CMS, el SO de los mismos, el país de la solicitud ... Podríamos hacer un estudio para reforzar los puntos más débiles del escenario (solucionar ralentizaciones debido a picos de visitas, hacer más atractivo el servicio...)
- **Usar kubernetes para el escenario:** Con kubernetes tendríamos mayor facilidad para distribuir el entorno en distintos nodos. Se puede usar como referencia mi repositorio en el cual mediante k8s tenemos el escenario funcionando con Drupal, MariaDB, Loki, Fluent Bit y Grafana, por lo que sería conveniente añadir la monitorización para completarlo
- **Apache Exporter:** Al igual que con MySQL Exporter, podríamos usar esta herramienta para controlar más a fondo nuestro Apache siguiendo el mismo procedimiento, a través de un dashboard en Grafana
- **Mejoras referentes al programa python:** Podríamos seguir añadiendo opciones al programa python, ya que con el que yo he hecho se podría decir que se ha rascado la superficie, ya que se podría hacer un proyecto entero usando la API de Grafana. También podríamos crear una interfaz gráfica de nuestro programa.
- **Clúster de base de datos:** Podríamos añadir un clúster de base de datos y usar más de un contenedor simultáneamente.

6. Bibliografía, enlaces, reseñas,...

Grafana Loki

Grafana

Repositorio oficial fluent bit

Manual fluent bit

Grafana API

Repositorio de Node Exporter

Repositorio de MySQL Exporter

How to run haproxy with docker

Grafana api library