

How To - Grupo 17 - Grafana

Promethoide

Promethoide es una red virtual desarrollada en Docker con la capacidad de monitorear hosts y servicios dentro de la misma, que cuenta con una interfaz visual en Grafana para observar las métricas recopiladas de cada uno de ellos. Algunos de los servicios nativos de la red son:

Prometheus

Este servicio es el orquestador del monitoreo de todos los otros hosts y servicios. Al estar configurado en docker que todos los contenedores forman parte de la misma red, este servicio gestiona y controla las métricas de cada uno de ellos y las expone para que desde Grafana se puedan consumir y visualizar en dashboards.

API Rest

La API Rest fue desarrollada en Fast API. Consta de la funcionalidad básica que utilizaría un operario de un puesto de inmigraciones en un aeropuerto. Esto sería validar que el DNI del viajero no aparezca en la base de datos de Interpol, verificar que el vuelo exista y asegurarse de que el hospedaje y el dinero que tiene declarado para su estadía son suficientes.

PostgreSQL

Este servicio fue levantado para almacenar la información de migraciones. Dentro de el mismo se encuentra la tabla de interpol que emula la base de datos de la organización de seguridad internacional, la tabla de vuelos y la de migraciones. Este servicio va a ser utilizado por la API para almacenar y consultar información.

Postgres Exporter

Este exporter es un servicio capaz de exponer métricas propias del servicio previamente explicado. Se le provee la manera para acceder al data source correspondiente para que pueda obtener métricas sobre la lectura y escritura en la base de datos.

Node Exporter

Este servicio nos permite montar en un contenedor de Docker características de hardware del host que levantó el contenedor. Se sirve de una copia en read-only del sistema operativo y del mismo obtiene métricas de hardware que luego expone en un endpoint para que prometheus consuma.

Apache

Apache es un servicio web del sistema que sirve una página muy básica hecha con javascript, html y css. Dicha página cuenta con el flujo básico que utilizaría un operario de inmigraciones, consumiendo la Rest API e impactando en el Postgres Server, ambos explicados anteriormente. A su vez, modificando la configuración y agregando un modulo para habilitar el monitoreo del servicio, se exponen en el endpoint `/server-status` las métricas del estado del mismo para luego ser consumidas.

Apache exporter

Este servicio permite hacer scrapping de las métricas del servicio Apache en el endpoint `/server-status` en intervalos de tiempo preestablecidos. Luego, estas métricas serán proporcionadas al servicio de Prometheus para que esta las provea como data source a grafana.

Blackbox exporter

El servicio de Blackbox Exporter nos permite hacer probing HTTP, DNS, etc. Nos da la posibilidad de setear los distintos endpoints de una web para verificar su tiempo de respuesta y el status code que devuelve entre otras cosas. Al igual que el servicio anterior, el probing se hace a intervalos de tiempo regulares.

k6

K6 es un servicio levantado para lograr realizar smoke tests lanzados por un cron job. El servicio se encarga de testear periodicamente el flujo principal de la aplicación y asegurarse de que la cantidad de requests hechas y sus correspondientes status codes sean los adecuados.

Pushgateway

Como bien indica su nombre, este servicio es la puerta de entrada para las métricas provistas por k6. Es necesario ya que a diferencia de los servicios exporters, pushgateway va a ser capaz de servir métricas a prometheus de jobs efímeros o "cron". Como este tipo de tareas puede que no existan el

tiempo suficiente para ser scrappeadas, pueden pushear sus métricas a este servicio y así persistirse en un endpoint hasta actualizarse en la próxima corrida de la tarea.

Grafana

El servicio de Grafana es el encargado de recopilar todas las métricas proporcionadas por prometheus y brindarle observabilidad al administrador de la red de aquello que esta ocurriendo en los distintos servicios. Dentro de este, se settean alertas que notifican a los distintos niveles de administradores de problemas tales como alta latencia en endpoints de la API, caída de un servicio y delays fuera de lo común en el servicio web. Las alertas se envían a través de un servidor SMTP configurado dentro del mismo servicio de grafana.

Instalación

Para instalar Promothoide, es necesario contar con una versión actualizada de Docker y Docker Compose. Los requerimientos de cada uno de los servicios van a ser manejados por las distintas imagenes que se descarguen en los contenedores de Docker.

```
├─ apache
│  ├─ Dockerfile
│  ├─ proxy.conf
│  ├─ script.js
│  ├─ styles.css
│  └─ index.html
├─ api
│  ├─ main.py
│  └─ requirements.txt
├─ blackbox
│  └─ blackbox.yml
├─ grafana
│  └─ provisioning
│     └─ alerting
│        └─ alerting.yml
│     └─ dashboards
│        └─ dashboards.yml
│     └─ datasources
│        └─ datasources.yml
```

```

├── k6-pycron
│   ├── lib/pycron
│   ├── Dockerfile
│   ├── my_config.yaml
│   ├── scheduler.py
│   └── web_workflow_test.js
├── postgres
│   └── init.sql
├── prometheus
│   └── prometheus.yml
├── .example.env
├── .gitignore
├── docker-compose.yaml
└── README.md

```

Aca podemos visualizar el arbol del proyecto y algunos de los archivos a los que haremos referencia. La configuración central de cada servicio estará por lo general en el archivo `docker-compose.yaml` el cual explicaremos en detalle a continuación.

Ejecución

Antes de comenzar, asegurarse de tener un archivo `.env` correspondiente en la raíz del directorio del repositorio con la siguiente configuración:

```

##### POSTGRES SECRETS #####
#####
POSTGRES_USER=<usuario-de-postgres>
POSTGRES_PASSWORD=<contraseña-de-postgres>
POSTGRES_DB=<base-de-datos-de-postgres>
POSTGRES_HOST=<host-de-postgres>
POSTGRES_PORT=<puerto-de-postgres>
##### GRAFANA SMTP SECRETS #####
#####
GF_SMTP_FROM_ADDRESS=<dirección-del-remite>
GF_SMTP_FROM_NAME=<nombre-del-remite>
GR_SMTP_HOST=<nombre:puerto>
GF_SMTP_PASSWORD=<contraseña-de-aplicación>

```

Uso

1. Navega al directorio del repositorio:

```
cd promethoide
```

2. Otorga permisos de ejecución al script:

```
chmod +x run.sh
```

3. Ejecuta el script:

```
./run.sh
```

Este script verificará si las imágenes necesarias para ejecutar el proyecto están construidas. Si no lo están, las construirá. Luego, ejecutará Docker Compose para iniciar los contenedores según la configuración definida en el archivo `docker-compose.yml`.

Estructura de Promethoide

La columna vertebral de este proyecto es, como en todo proyecto elaborado en docker, el archivo de configuración `docker-compose.yml`. En el mismo se explicita la configuración de cada uno de los servicios y el seteo de las distintas redes.

```
version: '3.8'

# https://docs.docker.com/network/drivers/: bridge mode by default
networks:
  operations-net:
  services-net:

volumes:
  # Metrics storage
  prometheus-data:
    driver: local
  # Config files and dashboards configurations
  grafana-data:
    driver: local
```

Para permitir que los servicios puedan interactuar entre ellos, es indispensable que estén dentro de la misma red. En este caso, se decidió utilizar dos redes: `operations-net` y `services-net`. Se hizo esta distinción ya que se consideró que en un caso real, el equipo que monitorea las métricas de algunos servicios podría no ser el mismo equipo que tiene acceso a los servicios en sí. Es por eso que la red de `operations-net` engloba al servicio de prometheus y grafana, mientras que `services-net` agrupa en una red todos los otros servicios y también prometheus.

Si bien esto podría haberse modularizado más, se considero que esta división era suficiente para hacer la división correspondiente entre grupos de usuarios.

Por otra parte, se crearon dos volúmenes para los servicios de monitoreo de grafana y prometheus, que permiten almacenar configuración como dashboards y alertas entre otras cosas.

Configuration de Prometheus

```
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/etc/prometheus/console_libraries'
      - '--web.console.templates=/etc/prometheus/consoles'
      - '--web.enable-lifecycle'
    expose:
      - 9090
    ports:
      - 9090:9090
    networks:
      - operations-net
      - services-net
```

Se puede observar que dicho servicio esta en ambas redes mencionadas previamente y cuenta con dos volúmenes. El primero se encarga de copiar la configuración establecida en el archivo local `prometheus.yml` y pasarlo a la configuración del servicio dentro de la carpeta `/etc/prometheus`, mientras que el segundo recopila la información de los datos recolectados.

El servicio de prometheus es el encargado de orquestar y gestionar las métricas expuestas por los distintos servicios. Es por ello que en su configuración debe identificar los puntos de acceso de cada uno de ellos para poder extraer la información en cuestión. Veamos en detalle el archivo `prometheus.yml` que es donde se explicita dicho comportamiento.

```
# prometheus.yml - 1/3

global:
  scrape_interval: 15s
  evaluation_interval: 15s
```

Tras setear tanto el intervalo de scrapping de las métricas como el intervalo de evaluación, se comienza con la configuración estática de los targets para hacer polling de las métricas.

```
# prometheus.yml - 2/3

scrape_configs:

  - job_name: 'node-exporter'
    static_configs:
      - targets: ['node-exporter:9100']

  - job_name: 'rest-api'
    static_configs:
      - targets: ['rest-api:5001']

  - job_name: 'apache-exporter'
    static_configs:
      - targets: ['apache-exporter:9117']

  - job_name: 'postgres-exporter'
```

```
static_configs:
  - targets: ['postgres-exporter:9187']
```

- `job_name`: Es el identificador del servicio sobre el cual se va a scrapear.
- `static_configs`: Es la lista de configuraciones estáticas donde se define un conjunto de targets a scrapear
- `targets`: Lista de <ip:puerto> que van a ser scrapeados

Si bien el servicio de Prometheus tiene la opción de configurar un Service Discovery para encontrar automáticamente los targets de donde obtener las métricas, al tener pocos servicios y tener control completo sobre el entorno que estamos trabajando nos pareció más sencillo configurar la lista de targets de manera estática.

```
# prometheus.yml - 3/3

- job_name: 'blackbox-exporter'
  metrics_path: /probe
  params:
    module: [http_2xx] # Look for a HTTP 200 response.
  static_configs:
    - targets:
      - http://apache:80
      - http://rest-api:5001
      - http://apache:80/delay
  relabel_configs:
    - source_labels: [__address__]
      target_label: __param_target
    - source_labels: [__param_target]
      target_label: instance
    - target_label: __address__
      replacement: blackbox-exporter:9115 # The blackbox e.
```

Esta configuración estática de scrapping es la única que se diferencia de las demás. Nos permite hacer probing de los targets listados dentro de `static_configs` y buscar códigos de respuesta 200. Nos va a permitir determinar si nuestros servicios están caídos o si comprobar el delay de un endpoint determinado dentro de nuestra web app.

Configuration de Rest API

```
services:
  rest-api:
    image: python:3.10
    container_name: rest-api
    restart: unless-stopped
    command: /bin/sh -c "pip install -r requirements.txt && python main.py"
    volumes:
      - ./api:/app
    working_dir: /app
    environment:
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_DB=${POSTGRES_DB}
      - POSTGRES_HOST=${POSTGRES_HOST}
      - POSTGRES_PORT=${POSTGRES_PORT}
    ports:
      - "5001:5001"
    depends_on:
      - postgres
    networks:
      - services-net
```

Este servicio esta configurado dentro de la red de servicios `services-net` y su inicialización depende del servicio de postgres. Se pasan como variables de entorno las credenciales necesarias para pegarle a la base de datos, y los requerimientos se proveen desde el archivo requirements.txt.

Para ejecutar la API se corre el comando explicitado bajo el tag command, que se encarga de instalar los requerimientos y ejecutar el archivo main.py. De este modo, queda expuesta la API en el puerto 5001 de nuestro contenedor.

Una vez levantada la API, más allá de generar los distintos endpoints que serán necesarios para la aplicación web, es vital hacer algunas configuraciones. En primer lugar, se requiere proporcionar un endpoint para que prometheus extraiga las métricas que queramos medir de nuestra Rest API. Es por esto que utilizando la librería prometheus_client recopilamos métricas y habilitamos el endpoint `/metrics` para exponerlas.

```
from prometheus_client import Counter, generate_latest, Histogram
```

```
@app.get("/metrics")
async def metrics():
    data = generate_latest()
    return Response(content=data, status_code=200, headers={"
```

Dentro de la misma API, se busca mediante el uso de un middleware medir la latencia de cada uno de los endpoints para luego desde Grafana poder visualizar la carga sobre ellos. En base a eso se configuran alarmas que notifican a los administradores de la red de servicios.

```
REQUEST_LATENCY = Histogram(
    "rest_api_request_latency",
    "Latency of HTTP requests in seconds",
    ["endpoint"],
    buckets=[0.050, 0.100, 0.150, 0.200, 0.250, 0.300, 0.350,
              0.800, 0.850, 0.900, 0.950, 1.0]
)
REQUEST_COUNT = Counter("rest_api_requests_total", "Total num

@app.middleware("http")
async def add_prometheus_metrics(request: Request, call_next)

    if request.url.path == "/metrics":
        return await call_next(request)

    start_time = time.time()

    response = await call_next(request)

    process_time = time.time() - start_time
    REQUEST_LATENCY.labels(request.scope['route'].name).observe

    REQUEST_COUNT.inc()
    return response
```

Por otra parte, cada uno de los endpoints tiene otra métrica donde se calcula la cantidad total de requests.

Configuration de Postgres

```
services:
  postgres:
    image: postgres
    container_name: postgres
    volumes:
      - ./postgres/init.sql:/docker-entrypoint-initdb.d
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
      POSTGRES_MAX_CONNECTIONS: 1000
    ports:
      - 5432:5432
    networks:
      - services-net
    restart: unless-stopped
```

La configuración de dicho servicio es bastante básica. En principio se utiliza la imagen de Postgres y se estipula un volumen para el entrypoint. Este último se utiliza para popular la base de datos al iniciar el servicio.

Por último, aparte de configurar la red a la que forma parte el servicio y el puerto que utilizara, se asignan las credenciales provenientes del archivo de

`.env`.

Configuración de Postgres Exporter

```
services:
  postgres-exporter:
    image: quay.io/prometheuscommunity/postgres-exporter
    container_name: postgres-exporter
    restart: unless-stopped
    environment:
      - DATA_SOURCE_NAME=postgres://${POSTGRES_USER}::
```

```
expose:
  - 9187
ports:
  - 9187:9187
networks:
  - services-net
```

El servicio postgres-exporter levantado en el puerto 9187 fue configurado con una variable de entorno DATA_SOURCE_NAME necesaria para indicar cual es el servicio postgres que se esta queriendo monitorear. Para construir su valor se utilizan secrets establecidos en el `.env`.

Configuración de Node Exporter

Este servicio es el encargado de exponer características de hardware y métricas del sistema operativo de kernels *NIX, es decir, centrados en Unix.

```
services:
  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    volumes:
      - /proc:/host/proc:ro
      - /sys:/host/sys:ro
      - /:/rootfs:ro
    command:
      - '--path.procfs=/host/proc'
      - '--path.rootfs=/rootfs'
      - '--path.sysfs=/host/sys'
    expose:
      - 9100
    networks:
      - services-net
```

En su configuración se puede ver cómo se montan en los volumen:

- `/proc`: Se monta dentro del contenedor el sistema de archivos virtual para acceder a datos del kernel y del sistema, relacionados principalmente con los procesos en ejecución.

- `/sys`: Se monta otro sistema de archivos virtual que exporta información sobre los dispositivos y controladores del sistema. Fue introducido para mejorar la organización de la información sobre los dispositivos hardware del sistema.
- `/`: Se monta el filesystem del root sobre un sistema de archivos virtual del contenedor

Los tres se encuentran en modo read-only de forma tal que no puedan realizarse cambios dentro del contenedor y que impacten en el host en el que se corrió el proyecto.

Estas rutas despues son agregadas a la configuración del contenedor usando los comandos correspondientes, y finalmente se expone el puerto en el que se levantará el servicio y se configura a qué red pertenecera.

Configuración de Apache

```
services:
  apache:
    build: ./apache
    container_name: apache
    ports:
      - "8080:80"
    networks:
      - services-net
    restart: unless-stopped
```

En el docker-compose.yml se define el servicio, y se declara que se va a buildear a partir del Dockerfile ubicado dentro de la carpeta local `/apache`.

```
# Dockerfile

FROM httpd:2.4

# Copy configuration files
COPY server.conf /usr/local/apache2/conf/extra/server.conf

# Enable modules
RUN echo "LoadModule proxy_module modules/mod_proxy.so" >> /u
```

```

&& echo "LoadModule proxy_http_module modules/mod_proxy_h
&& echo "LoadModule rewrite_module modules/mod_rewrite.so
&& echo "LoadModule cgi_module modules/mod_cgi.so" >> /us
&& echo "Include conf/extra/server.conf" >> /usr/local/ap

# Copy files
COPY ./html/ /usr/local/apache2/htdocs/

# Copy delay script
COPY delay.sh /usr/local/apache2/cgi-bin/
RUN chmod +x /usr/local/apache2/cgi-bin/delay.sh

```

Este Dockerfile se encarga de obtener la imagen de apache con su respectiva versión, copiar el archivo local server.conf y agregar, junto al mismo, varios módulos al archivo `httpd.conf`.

También se copian los archivos de la página web ubicados dentro del directorio `/html` y el script `delay.sh` que se utilizó como trigger para generar un delay y así lanzar una alarma desde Grafana.

```

<VirtualHost *:80>
    ServerName localhost
    DocumentRoot "/usr/local/apache2/htdocs"

    <Directory "/usr/local/apache2/htdocs">
        AllowOverride All
        Require all granted
    </Directory>

    ProxyPreserveHost On
    ProxyPass /api http://rest-api:5001
    ProxyPassReverse /api http://rest-api:5001

    ScriptAlias /delay /usr/local/apache2/cgi-bin/delay.sh
</VirtualHost>

<Location "/server-status">
    SetHandler server-status

```

```
#Require host apache-exporter
</Location>
```

Dentro del `server.conf` se puede observar cómo se configura para escuchar en todas las direcciones IP disponibles en el puerto 80. Se define un `ServerName`, se establece cuál es la ruta del sistema de archivos donde se encuentran los archivos del sitio web para dicho host y se establecen directivas de acceso para el directorio donde se encuentran las páginas del sitio web.

Por otra parte, se define un proxy inverso para poder manejar las requests a la api bajo la url `/api`. y también se define un alias para un script en el servidor, que se ejecutará cada vez que se llame a dicho endpoint.

Finalmente, se configura el manejo de solicitudes para el endpoint `/server-status`. Con la directiva `SetHandler server-status` se está configurando Apache para devolver información de estado del servidor cuando se accede a dicha ruta.

Configuración de Apache Exporter

```
services:
  apache-exporter:
    image: lusotycoon/apache-exporter:latest
    container_name: apache-exporter
    restart: unless-stopped
    expose:
      - 9117
    ports:
      - 9117:9117
    networks:
      - services-net
    command:
      - '--scrape_uri=http://apache:80/server-status?au
```

El `apache-exporter` es el encargado de recolectar las métricas expuestas en el endpoint mencionado en el servicio anterior haciendo scrapping definido bajo el label `--scrape_uri` en la sección de command.

Este servicio se incorporó al igual que el anterior a la red de services-net, y se levantó en el puerto 9117.

Configuración de Blackbox Exporter

En cuanto a la declaración de este servicio en el docker-compose, lo único que se destaca es la creación de un volumen propio para el archivo de configuración `blackbox.yml`. Al igual que varios servicios anteriormente, se expone el puerto sobre el cuál va a estar ubicado y se le declara que será parte de la red de servicios.

```
services:
  blackbox-exporter:
    image: prom/blackbox-exporter:latest
    container_name: blackbox-exporter
    restart: unless-stopped
    volumes:
      - ./blackbox/blackbox.yml:/config/blackbox.yml
    command:
      - '--config.file=/config/blackbox.yml'
    expose:
      - 9115
    ports:
      - 9115:9115
    networks:
      - services-net
```

Dentro del archivo `blackbox.yml` ubicado dentro del directorio `blackbox` del proyecto, nos encontramos con lo siguiente:

```
modules:
  http_2xx:
    prober: http
    timeout: 5s
    http:
      preferred_ip_protocol: 'ip4'
      method: GET
      valid_status_codes: [200]
```

Dentro del mismo archivo se permite la declaración de módulos de configuración. En este caso, se utilizó el módulo para los códigos de respuesta exitosos en el rango de los 2xx. Se establece cuál será el método y la lista

permitida de status codes que se aceptarán de hacer probing HTTP en IPs de tipo IPv4, con un timeout de 5 segundos.

Configuración de k6

```
services:
  k6:
    build: ./k6-pycron
    container_name: k6
    restart: unless-stopped
    environment:
      - PROMETHEUS_PUSHGATEWAY_URL=http://pushgatew
    depends_on:
      - apache
    networks:
      - services-net
```

En el archivo docker-compose.yml, lo único destacable aparte de su pertenencia a la red services-net es la variable de entorno que define la URL a la cual k6 va a enviar las métricas recolectadas. El resto de la configuración se explicita en el Dockerfile dentro de la carpeta /k6-pycron

```
FROM pycron-image

# Install Python and curl
RUN apt-get update && apt-get -y install curl

# Download and install k6
RUN curl -L -o /tmp/k6.tar.gz https://github.com/grafana/k6/releases/download/v0.14.0/k6-linux-amd64.tar.gz && tar -xzf /tmp/k6.tar.gz -C /usr/local/bin --strip-components=1 && rm /tmp/k6.tar.gz

# Copia el archivo de configuración
COPY my_config.yaml /app/config.yaml

# Copy the test script and Python scheduler
COPY web_workflow_test.js /scripts/web_workflow_test.js
```

El dockerfile se construye a partir de una imagen de pycron-image.

Pycron es una librería alternativa a `cron` desarrollada por nosotros que permite ejecutar comandos de manera periódica utilizando Python, lo cual facilita su integración y manejo en contenedores Docker. Pycron está ubicado en `/k6-pycron/lib/` y debe ser construido antes de ejecutar el docker compose up (esto se maneja en `run.sh`).

El archivo de configuración `my_config.yaml` se definen las tareas a ejecutar y sus intervalos. Es un archivo en formato YAML que describe los comandos que Pycron debe ejecutar y con qué frecuencia.

```
tasks:
  - command: '/usr/local/bin/k6 run /scripts/web_workflow_test.js'
    interval: "1m"
```

En `/k6-pycron/lib/pycron/README.md` se puede encontrar más información sobre el funcionamiento de pycron.

El Dockerfile realiza los siguientes pasos adicionales:

1. **Instalación de curl:** Se descarga e instala el paquete curl.
2. **Descarga de k6:** Utilizando curl, se descarga la versión 0.51 de k6 desde el repositorio oficial de Grafana, se desempaqueta y se elimina el archivo descargado.
3. **Copia de archivos:** Se copia el archivo de configuración `my_config.yaml` y el script de prueba `web_workflow_test.js` al contenedor.

El script `web_workflow_test.js` es llamado por pycron para realizar el testeo de flujo sobre nuestra aplicación:

```
// web_workflow_test.js

import http from 'k6/http';
import { check, sleep } from 'k6';
import { Counter } from 'k6/metrics';
import { parseHTML } from 'k6/html';

const successCounter = new Counter('successful_requests');
const failureCounter = new Counter('failed_requests');
```

```

...

export default function () {
  let res = http.get('http://apache:80/index.html');
  let success = check(res, { 'status was 200': (r) => r.status === 200 });
  success ? successCounter.add(1) : failureCounter.add(1);

  sleep(1);

  res = http.get('http://apache:80/dni.html');
  success = check(res, { 'status was 200': (r) => r.status === 200 });
  success ? successCounter.add(1) : failureCounter.add(1);

  res = res.submitForm({
    formSelector: 'form',
    fields: { dni: test.dni },
  });

  ...
}

```

Como se puede ver en el código, primero se hace uso de las librerías importadas de k6 para poder generar, mediante pedidos http, un test que simule una transacción sintética dentro de la página web.

```

export function handleSummary(data) {
  const successCount = data.metrics.successful_requests ? data.metrics.successful_requests : 0;
  const failureCount = data.metrics.failed_requests ? data.metrics.failed_requests : 0;

  const result = `
# TYPE successful_requests counter
k6_successful_requests ${successCount}
# TYPE failed_requests counter
k6_failed_requests ${failureCount}
`;

  const url = `${__ENV.PROMETHEUS_PUSHGATEWAY_URL}/metrics/job`
}

```

```

const headers = { 'Content-Type': 'text/plain' };

let res = http.put(url, result, { headers: headers });
if (res.status !== 200 && res.status !== 202) {
    console.error(`Failed to push metrics to Pushgateway: $`
} else {
    console.log(`Successfully pushed metrics to Pushgateway
}

return {
    'summary.json': JSON.stringify(data),
    'stdout': result
};
}

```

Al final de este archivo, se analiza el estado de la variable data que k6 maneja de manera interna, y se publica en el endpoint del pushgateway las métricas recolectadas. Dentro de las mismas veremos, de todos los pedidos http que se realizó cuántos de ellos fueron exitosos y cuántos tuvieron un comportamiento anómalo.

Configuración de Pushgateway.

```

services:
  pushgateway:
    image: prom/pushgateway:latest
    container_name: pushgateway
    restart: unless-stopped
    ports:
      - 9091:9091
    networks:
      - services-net

```

La configuración de Pushgateway es bastante básica. Alcanza con explicitar la red y el puerto sobre el cual está trabajando el servicio.

Configuration de Grafana

```

services:
  grafana:
    image: grafana/grafana-oss:latest
    container_name: grafana
    restart: unless-stopped
    volumes:
      - grafana-data:/var/lib/grafana
      - ./grafana/provisioning:/etc/grafana/provisioning
    environment:
      - GF_SMTP_ENABLED=true
      - GF_SMTP_HOST=${GF_SMTP_HOST}
      - GF_SMTP_USER=${GF_SMTP_FROM_ADDRESS}
      - GF_SMTP_PASSWORD=${GF_SMTP_PASSWORD}
      - GF_SMTP_FROM_ADDRESS=${GF_SMTP_FROM_ADDRESS}
      - GF_SMTP_FROM_NAME=${GF_SMTP_FROM_NAME}
      - GF_SMTP_SKIP_VERIFY=true
    expose:
      - 3000
    ports:
      - 3000:3000
    depends_on:
      - rest-api
      - apache
    networks:
      - operations-net

```

La configuración del servicio de Grafana dentro del `docker-compose.yml` comienza definiendo los volúmenes necesarios para el servicio. Primero se configura el volumen que almacenará la información recolectada en la sesión y luego se incluye un volumen de configuración. Sobre el directorio

`/etc/grafana/provisioning` se va a montar la carpeta local que contiene la configuración de los dashboards, alertas, puntos de contactos y políticas de notificación. Es desde esta carpeta que grafana podrá generar los dashboards necesarios para visualizar la información y notificar a alguien de ser necesario.

Por otra parte, se definen variables de entorno para poder inicializar un servidor SMTP dentro de Grafana, que va a ser el responsable de enviar las notificaciones via mail a los distintos niveles de administradores. Un detalle

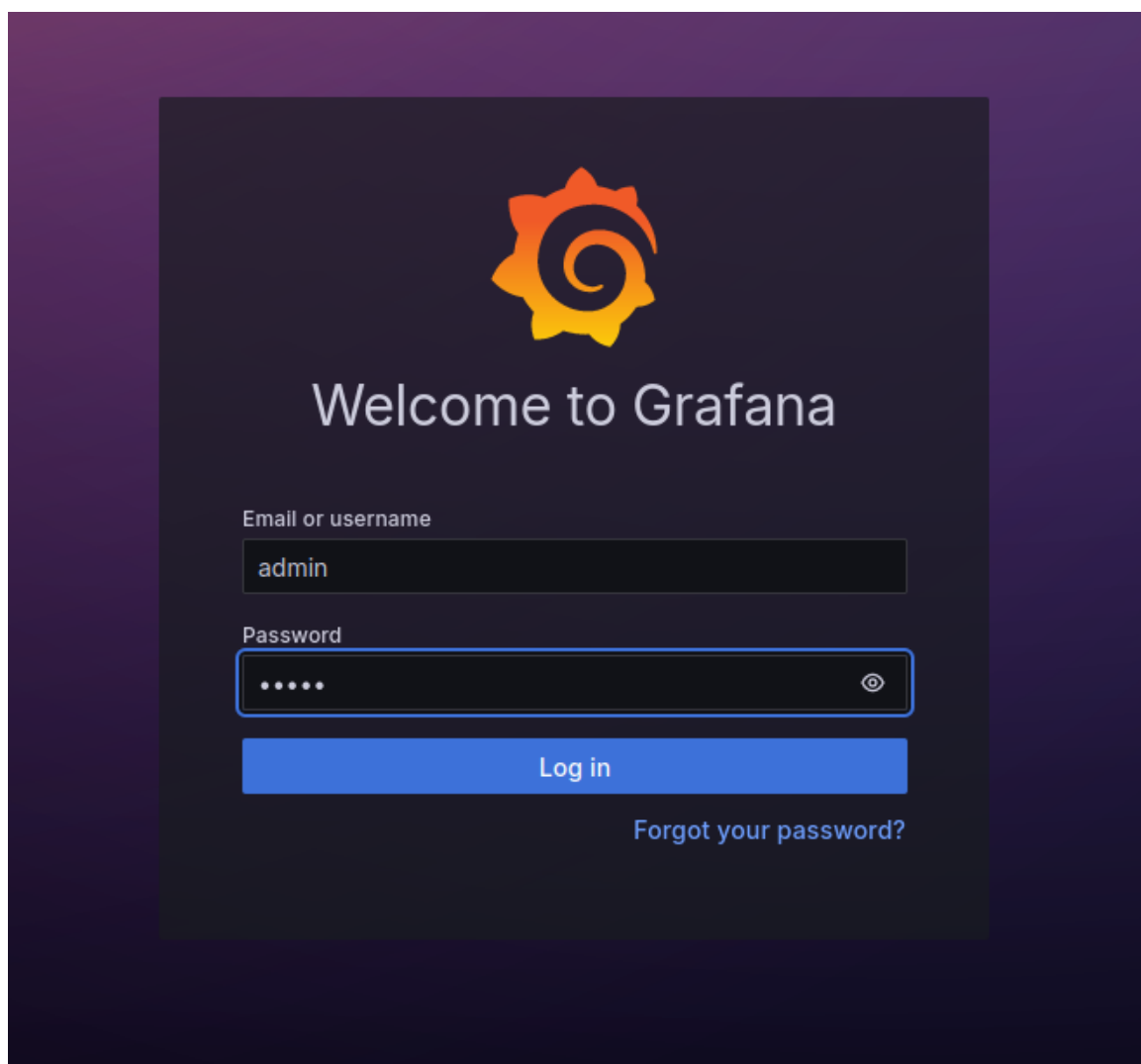
importante a la hora de utilizar este servidor de mail es que para poder utilizar un mail propio como remitente de los mails es necesario desde el cliente del proveedor (en nuestro caso Gmail) generar una clave de aplicación.

Cabe mencionar que el servicio de la inicialización de Grafana depende de los servicios de rest-api y apache, y pertenece a la red `operations-net`, que se encuentra aislada de la red de servicios.

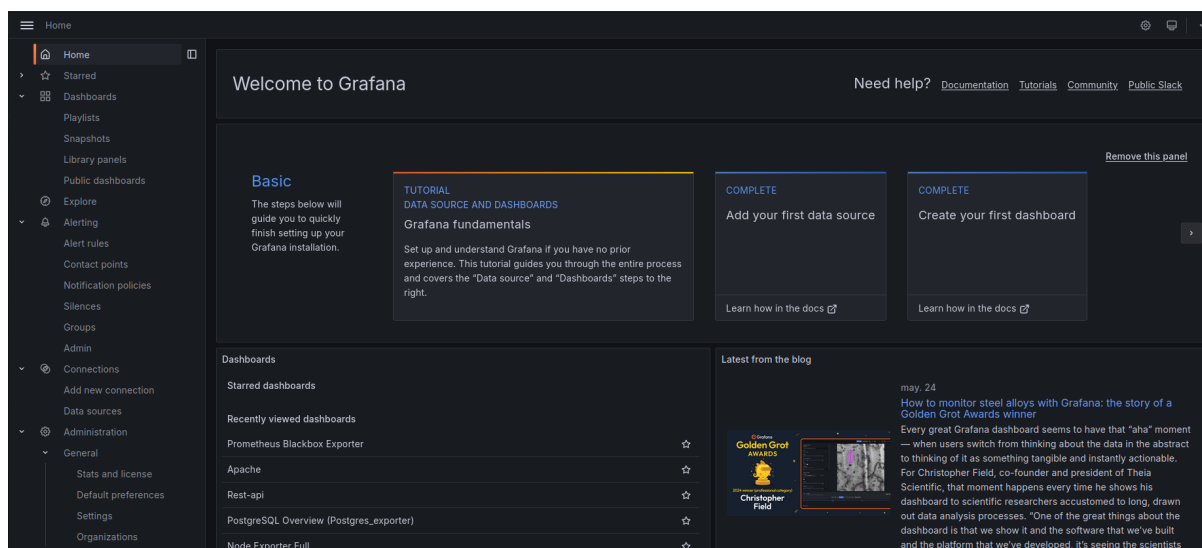
Monitoreo en Promethoide

Dejando de lado la configuración y adestrándonos un poco en el monitoreo en sí de la red, recorramos la herramienta utilizada y observemos cómo se definieron cada uno de los aspectos del monitoreo de este sistema.

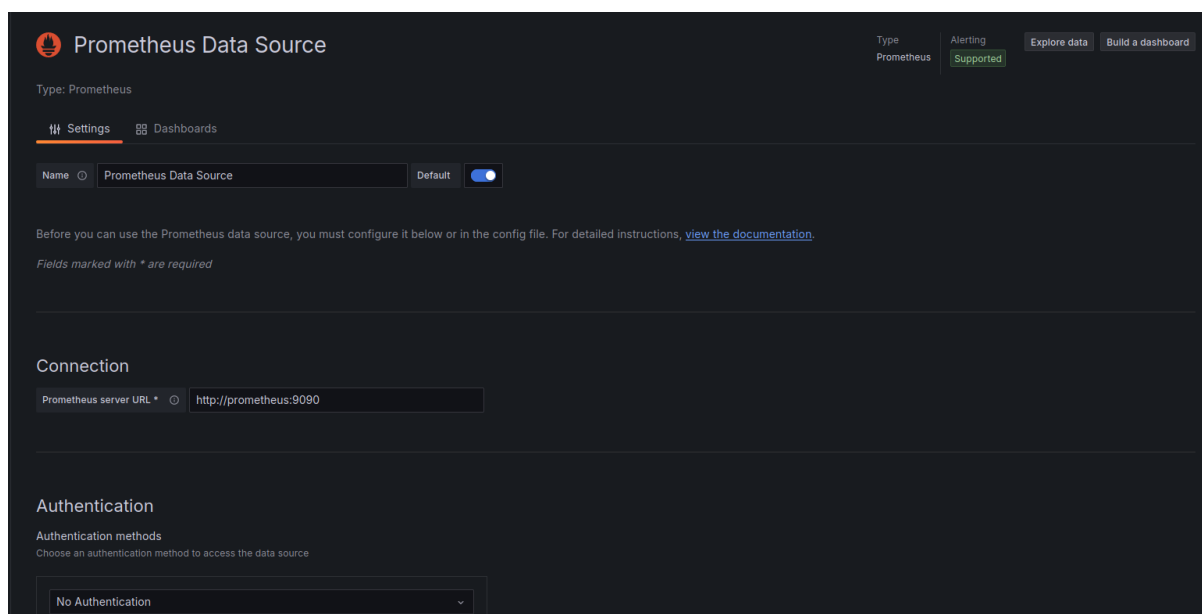
En el browser al ir a `http://localhost:3000`, lo primero que observaremos es el inicio de sesión del servicio de Grafana.



Con las credenciales admin:admin seteadas por default lograremos acceder al entorno de monitoreo previamente configurado.



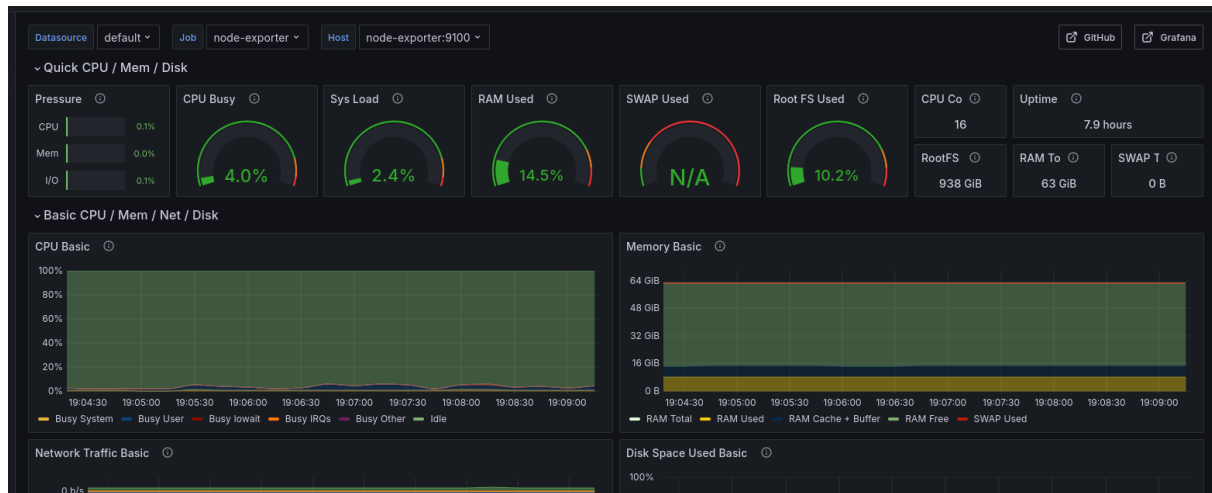
Yendo a la tab de **Connections > Data Sources**, observaremos que ya se encuentra seteado el servicio de prometheus como default. La URL fue seteada en `http://prometheus:9090`, que es donde esta levantado. Al estar Grafana y el servicio de prometheus en la misma red, la resolución del nombre se hace sin problema y conseguimos conectarnos con el servicio.



Al haber seteado el prometheus como Data Source, ahora estamos en condiciones de usar las métricas que se capturan por dicho servicio en los distintos dashboards. Observemos algunos de los provistos por el proyecto.

Dashboards


Para la generación de dashboards se optó tanto por el uso de templates para exporters conocidos como por la creación de nuevos dashboards. En el caso de Blackbox, Apache Exporter y Node Exporter se consultó el [Sitio oficial de Grafana](#) para obtener el ID de un template bastante completo y poder monitorear los distintos endpoints. Por otro lado, el dashboard de la API fue generado a mano por el equipo.



Para generar un dashboard en base a un ID, basta hacer click en "New Dashboard", seleccionar "Import", y cargar el "ID" elegido. Vamos a elegir el id 11159 solo a fines de demostrar cómo se importa un dashboard.

Import dashboard

Import dashboard from file or Grafana.com



Upload dashboard JSON file

Drag and drop here or click to browse

Accepted file types: .json, .txt

Find and import dashboards for common applications at grafana.com/dashboards

Import via dashboard JSON model

```
{
  "title": "Example - Repeating Dictionary variables",
  "uid": "_0HnEoN4z",
  "panels": [...]
  ...
}
```

Una vez hecho esto, se settea el Data Source correspondiente (que en este caso sería Prometheus).

Import dashboard

Import dashboard from file or Grafana.com

Importing dashboard from Grafana.com

| | |
|--------------|---------------------|
| Published by | Joey Yang |
| Updated on | 2019-11-10 10:39:46 |

Options

Name

NodeJS Application Dashboard

Folder


Dashboards

Unique Identifier (UID)
The unique identifier (UID) of a dashboard can be used for uniquely identify a dashboard between multiple Grafana installs. The UID allows having consistent URLs for accessing dashboards so changing the title of a dashboard will not break any bookmarked links to that dashboard.

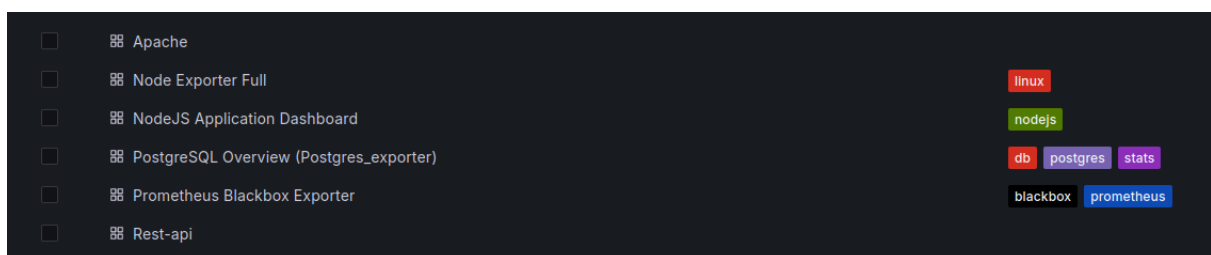
PTSqcpJWk [Change uid](#)

prometheus

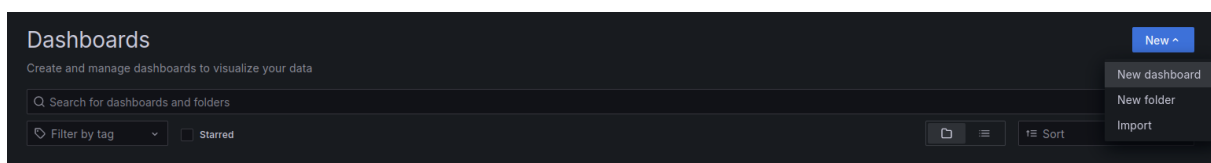
Select a Prometheus data source

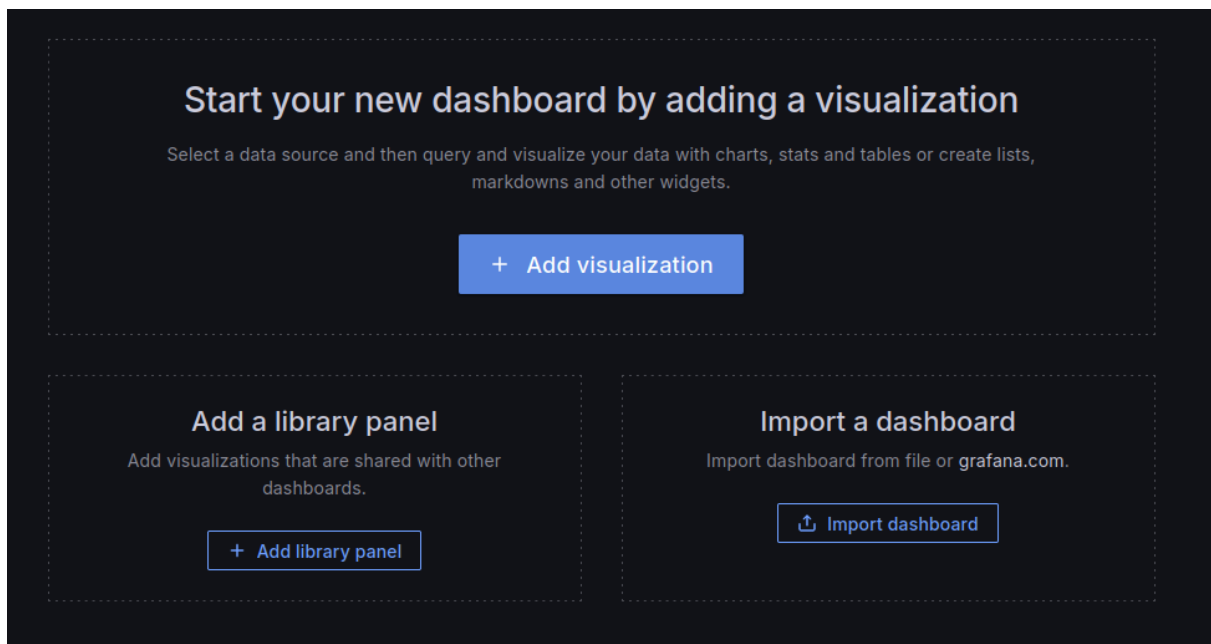
 Prometheus Data Source default Prometheus

Hecho esto, ya podremos Ubicar el nombre en el panel de Dashboards

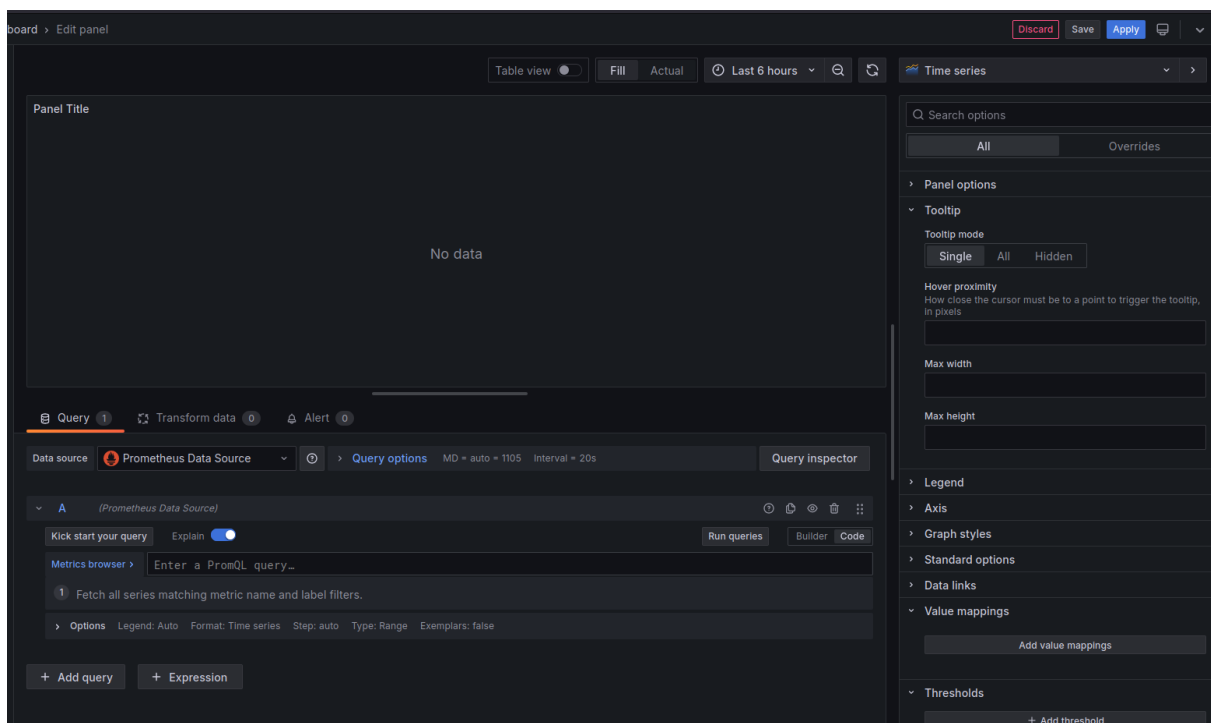


Si quisieramos crear un dashboard a mano, el procedimiento sería mucho más rudimentario y artesanal. Se debería generar un nuevo dashboard seleccionando New > New Dashboard y agregar una visualización:





Una vez agregada la visualización, podremos observar la siguiente pantalla.

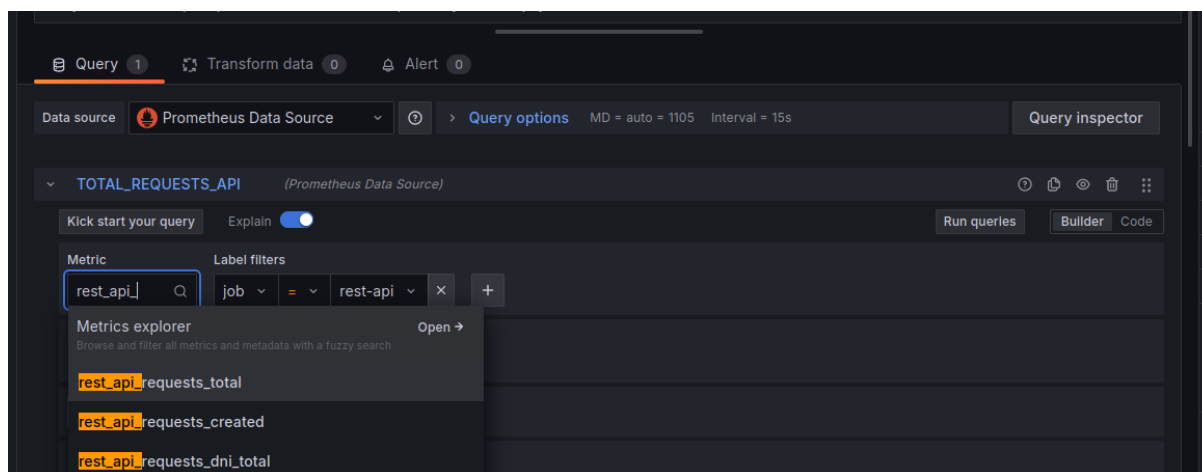


Un dashboard esta compuesto por muchas visualizaciones o “paneles”, que son componentes que dan observabilidad de una o varias métricas. Veamos cómo configurar el primer panel de este dashboard.

En la parte inferior izquierda podemos ver tres tabs: Query, Transform Data y Alert.

- **Query:** Permitirá definir consultas escritas en lenguaje PromQL sobre las métricas que nuestro Data Source nos provea. A su vez se pueden agregar operaciones tales como funciones de agregación, de rango, operadores binarios, trigonométricas, etc.
- **Transform Data:** Permitirá transformar la data antes de que se muestre la visualización. Se permite la unión, renombrado y fromateo de datos entre otras opciones.
- **Alert:** Se utiliza para definir alertas en base a posibles valores que tomen las consultas.

Supongamos que quisieramos hacer una query sobre la cantidad de request que tiene la API. Seleccionando la métrica, ya podemos hacer uso de la misma en nuestra consulta.

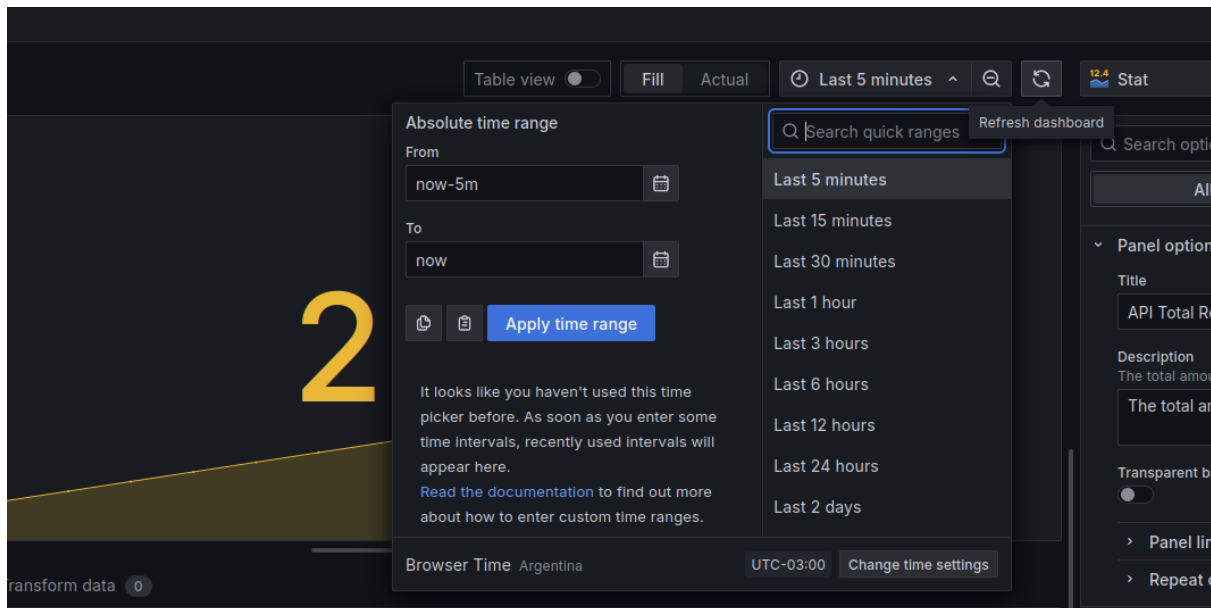


También se puede modificar el tipo de visualización que se quiere mostrar en el panel. Hay una gran variedad de opciones y la elección de la misma depende del la métrica que se desee monitorear.

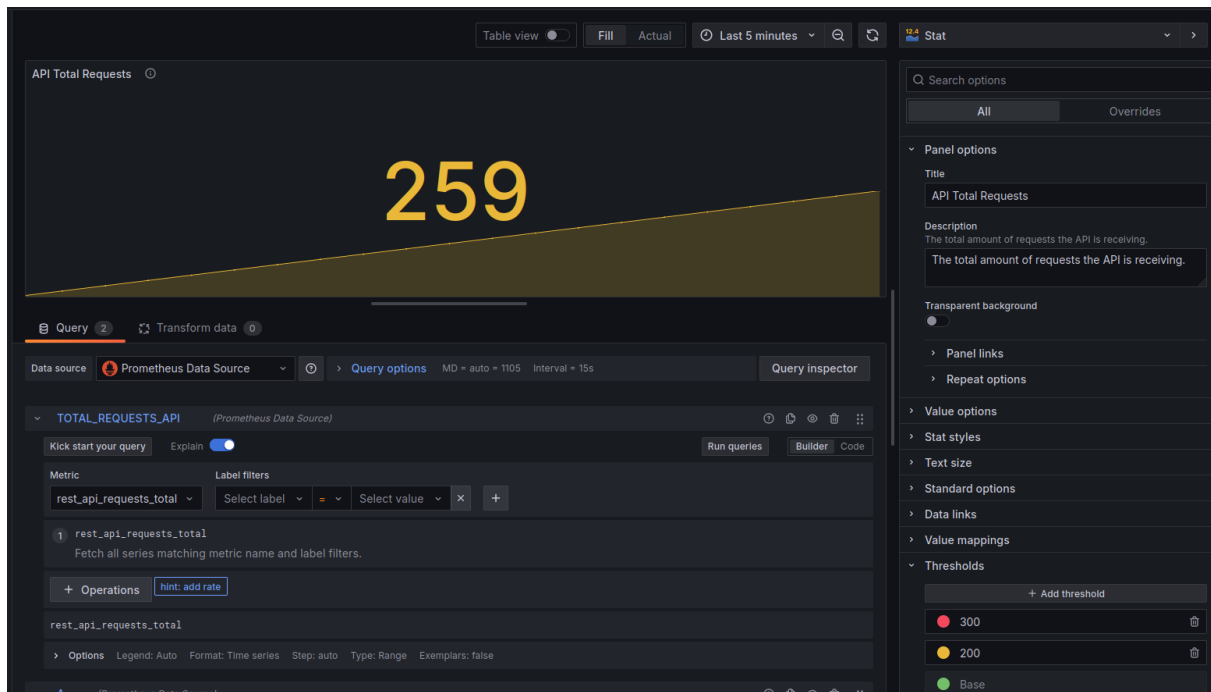




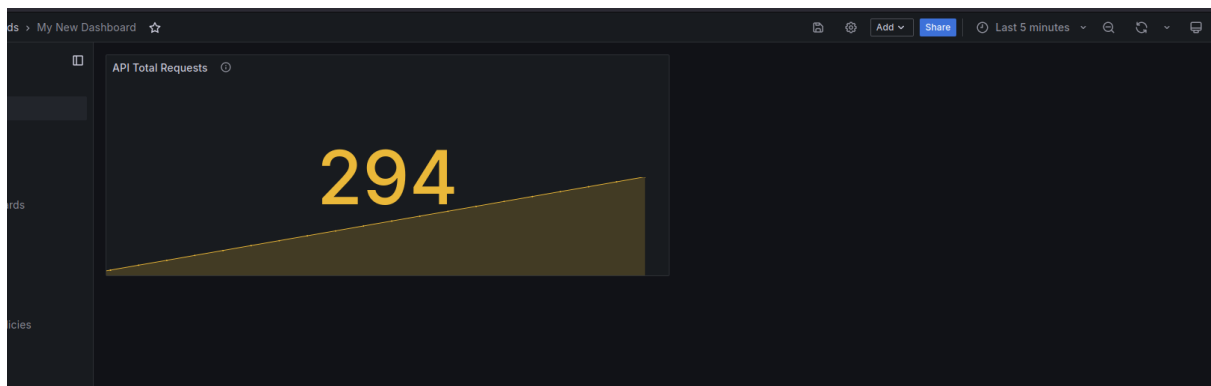
Observese que en la parte superior de la última imagen se nos permite modificar el rango de tiempo en el que se esta evaluando dicha query. Justo al lado del rango de evaluación, hay un boton para actualizar el panel a demanda, ignorando el intervalo de tiempo definido en las opciones de la query.



Finalmente, poniendo un título lo suficientemente descriptivo podemos guardar la visualización.



Una vez guardada, veremos el dashboard con el nuevo panel que acabamos de crear.



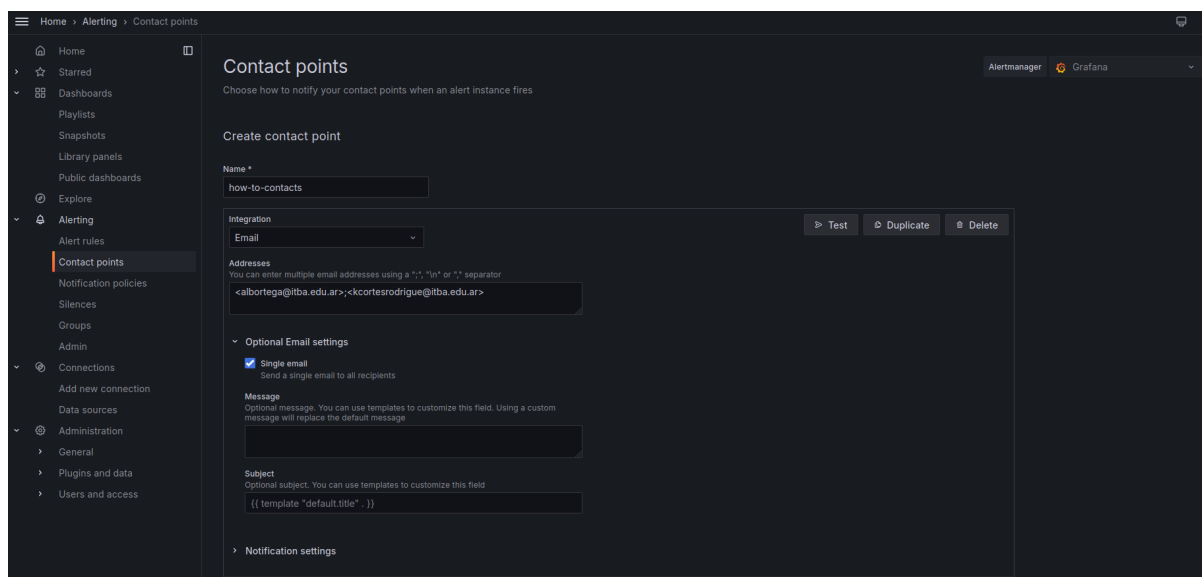
Alertas

Cuando se esta monitoreando un sistema es indispensable definir condiciones bajo las cuales se debe alarmar a los administradores para que revisen por qué una métrica esta dando un valor que no debería. Esto se logra configurando en grafana lo que se conoce como Alertas. Para poder configurar alertas en dicho servicio, es necesario definir lo siguiente:

- Contact Points
- Notification Policies
- Alert Rules

Contact Points

Son los grupos de contacto a los que puede dirigirse una alerta. En otras palabras, son los usuarios a los que grafana va a contactar, en este caso via mail, cuando una determinada alerta se dispare.

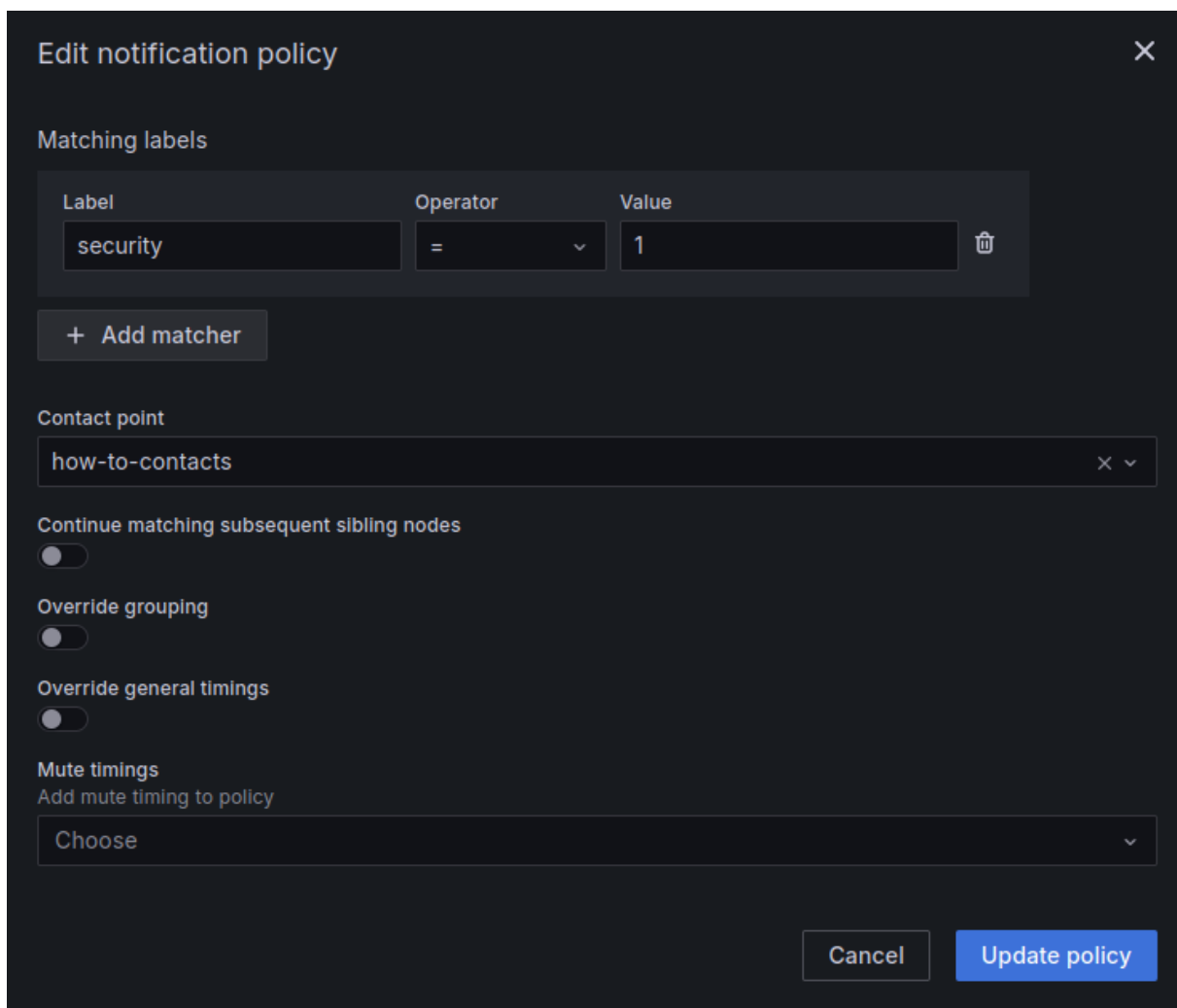
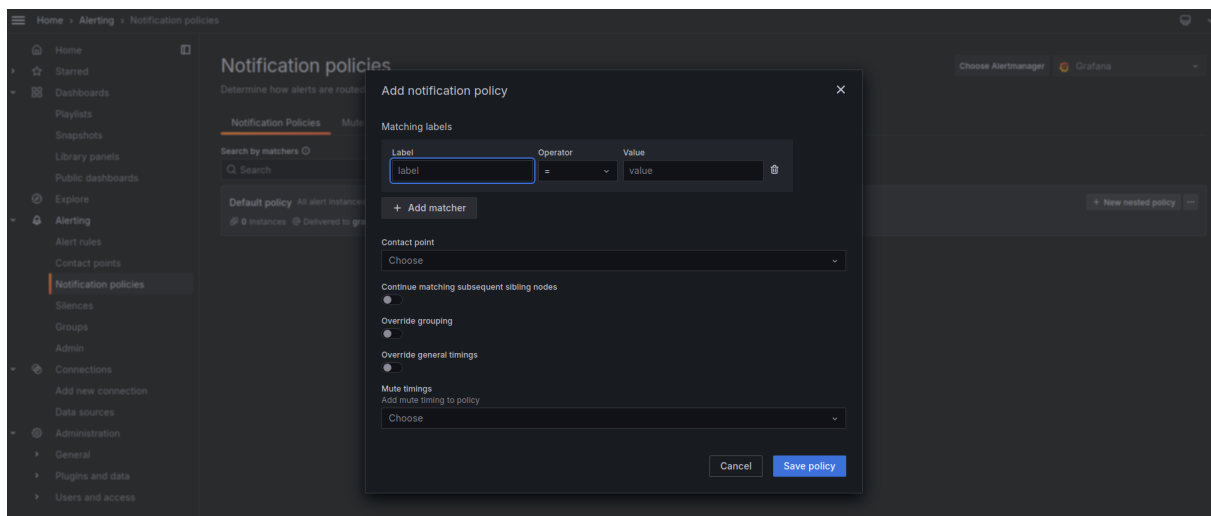


The screenshot shows the Grafana Alerting 'Contact points' configuration page. The left sidebar contains a navigation menu with options like Home, Starred, Dashboards, Playlists, Snapshots, Library panels, Public dashboards, Explore, Alerting, Alert rules, Contact points (selected), Notification policies, Silences, Groups, Admin, Connections, Add new connection, Data sources, Administration, General, Plugins and data, and Users and access. The main content area is titled 'Contact points' and includes a subtitle 'Choose how to notify your contact points when an alert instance fires'. Below this is a 'Create contact point' section with a form. The form has a 'Name' field with the value 'how-to-contacts'. The 'Integration' dropdown is set to 'Email'. There are buttons for 'Test', 'Duplicate', and 'Delete'. The 'Addresses' field contains two email addresses: 'calbortega@itba.edu.ar' and 'kcortesrodrigue@itba.edu.ar'. The 'Optional Email settings' section has a checked 'Single email' checkbox. The 'Message' field is empty, and the 'Subject' field contains the template '{{ template "default.title" }}'. A 'Notification settings' link is at the bottom.

En la sección de contact points definimos el nombre del punto de contacto, la integración correspondiente (que en este caso sería mail), definimos los mails separados por punto y coma y lo guardamos. Esto alcanza para poder e

Notification Policies

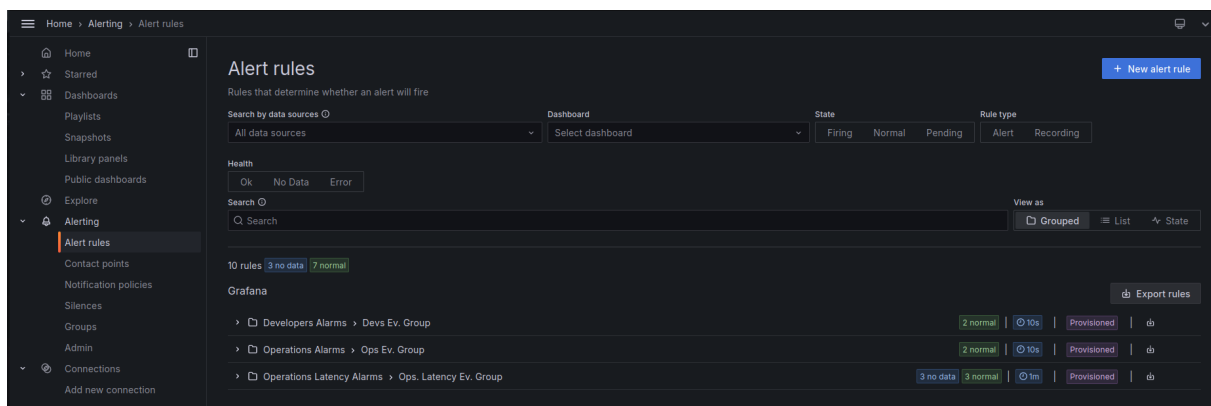
Las políticas de notificación nos permiten determinar cómo es que las alertas se van a routear a los puntos de contacto. Es decir, nos permiten vincular las reglas de las alertas con los puntos de contacto para que, cuando se defina una alerta con un label de nombre X con un valor específico, la alerta se envíe al punto de contacto establecido.



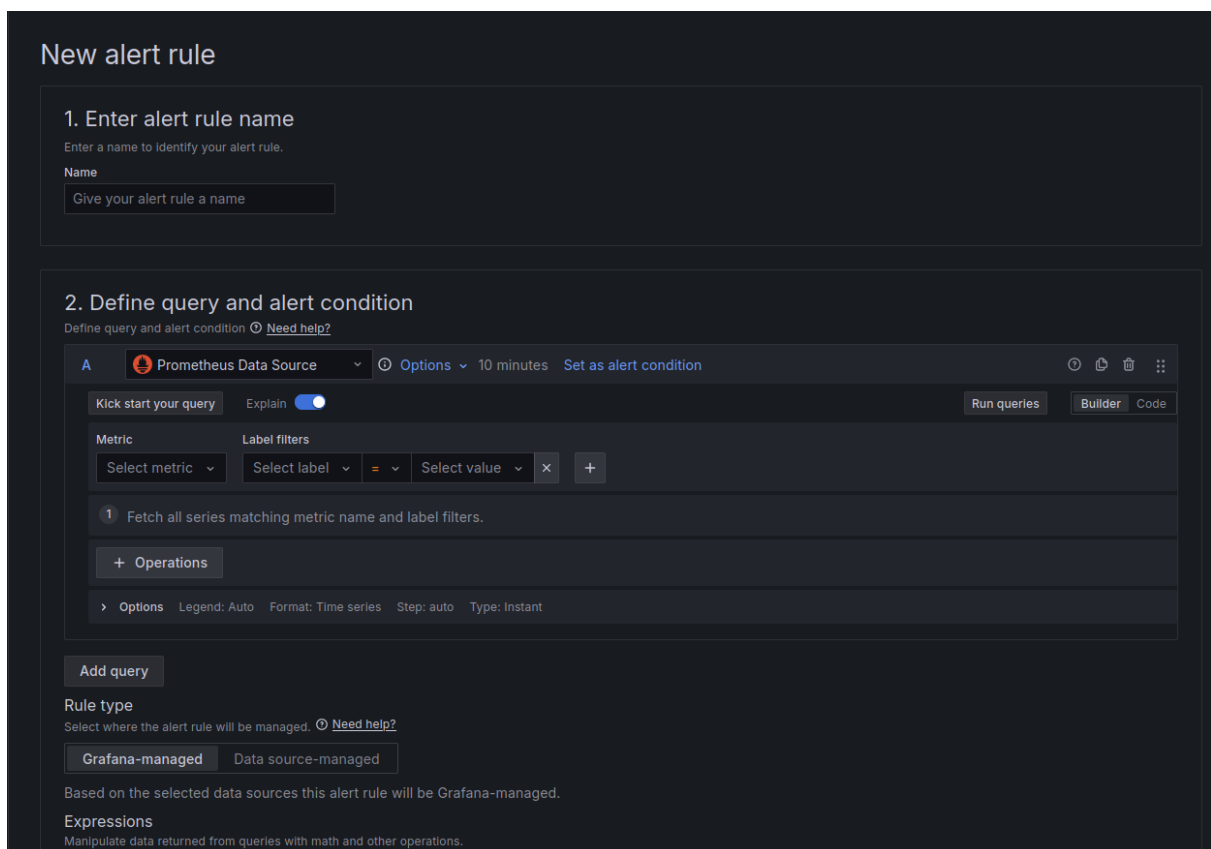
Una vez configurada de esta forma la política de notificación, cuando una alerta se configure con el label "security" con el valor 1, se van a mandar las notificaciones correspondientes al punto de contacto how-to-contacts.

Alert Rules

Son las reglas mismas que definen las condiciones que deben darse para que se informe a un punto de contacto acerca de un valor indebido en algún panel. Para generar una nueva regla de alerta podemos hacerlo desde la tab Alerts mencionada previamente en la sección explicativa de Dashboards, o bien podemos ir a la sección de Alerting y crear una nueva presionando **+New Alert Rule**. Creemos una alerta desde esta ultima vista.



Seleccionamos **+New Alert Rule** y podremos comenzar a crear nuestra alerta. Debemos definir la query que vamos a querer para luego evaluar su valor y establecer la condición para la alerta.



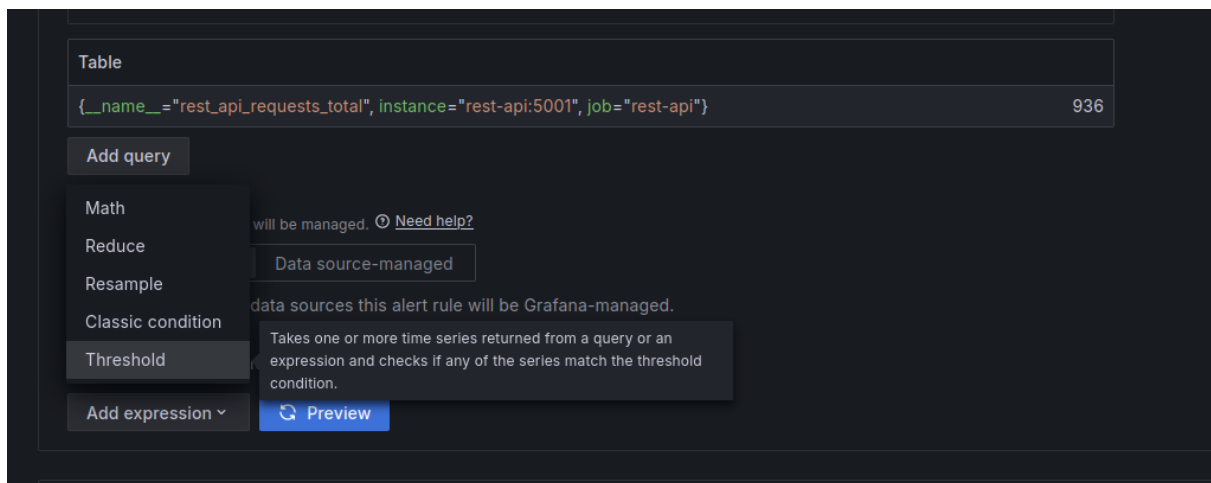
Una vez hecha la query, podemos ejecutarla presionando Run queries y así ver el valor correspondiente a la query evaluada en los últimos 10 minutos. El rango de tiempo puede configurarse en la parte de *Options* junto a la selección del Data Source. También se puede cambiar el nombre de dicha query a uno más descriptivo que "A".

The screenshot shows the Grafana Alerting configuration page. At the top, there's a 'Name' field with the placeholder 'Give your alert rule a name'. Below this is the section '2. Define query and alert condition' with a link to 'Need help?'. The main configuration area includes a 'Prometheus Data Source' dropdown, an 'Options' dropdown set to '10 minutes', and a 'Set as alert condition' button. There are also buttons for 'Run queries', 'Builder', and 'Code'. The 'Metric' field is set to 'rest_api_requests_total'. Below this, a list shows the query 'rest_api_requests_total' with the description 'Fetch all series matching metric name and label filters.' There's a '+ Operations' button with a hint 'add rate'. The resulting query is 'rest_api_requests_total'. Below the query, there are options for 'Legend: Auto', 'Format: Time series', 'Step: auto', and 'Type: Instant'. A table shows the result of the query:

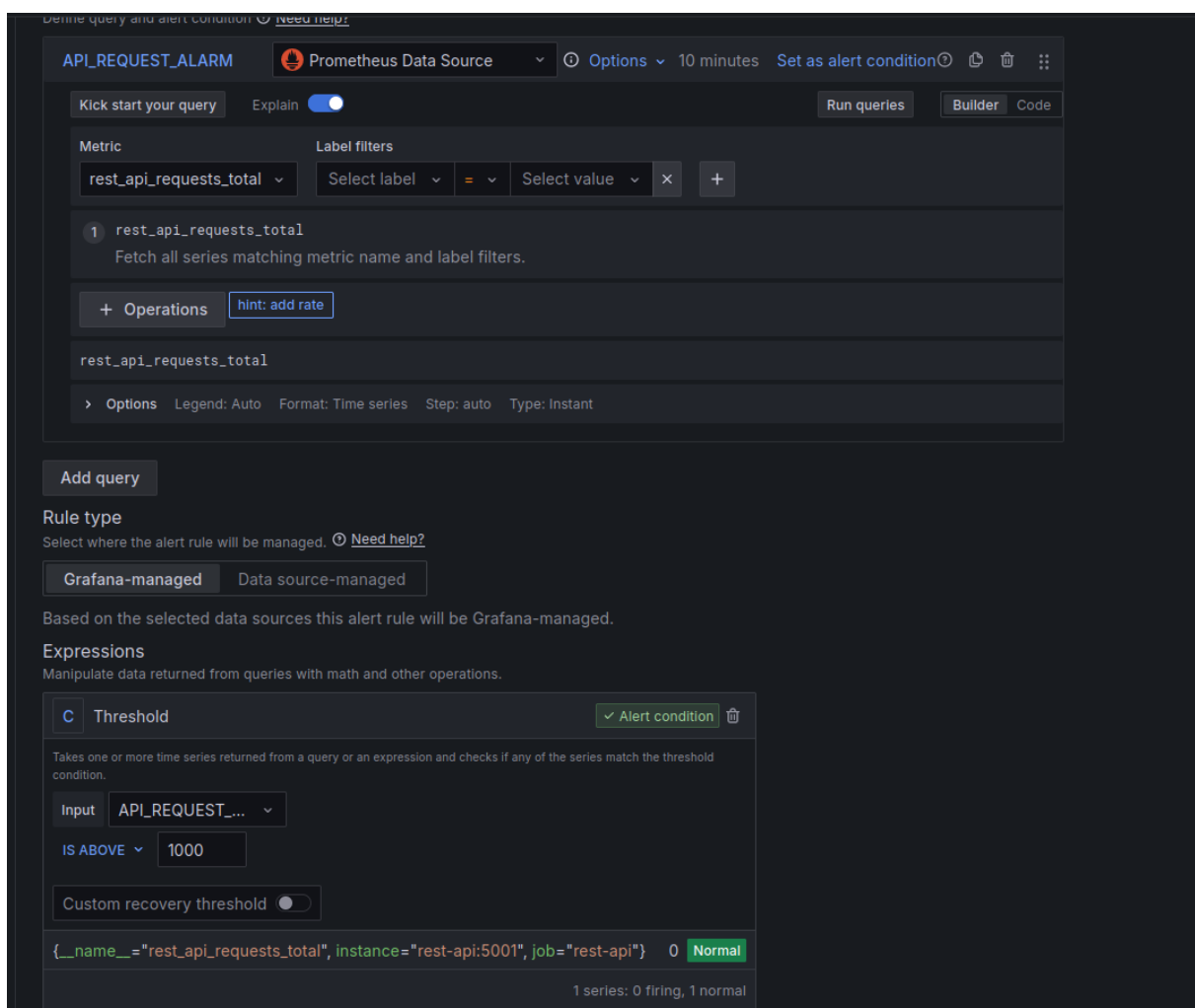
| Table | |
|--|-----|
| {__name__="rest_api_requests_total", instance="rest-api:5001", job="rest-api"} | 919 |

Below the table is an 'Add query' button. At the bottom, there's a 'Rule type' section with the text 'Select where the alert rule will be managed.' and a link to 'Need help?'.

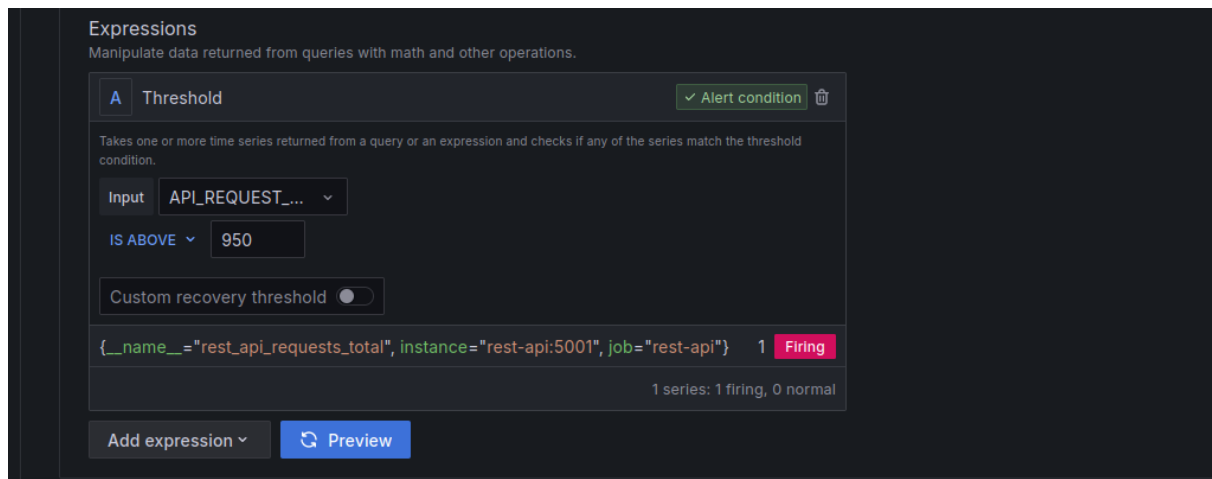
En queries simples como la de este ejemplo no es necesario aplicar ninguna función de reducción ni matemática para obtener un valor adecuado para una alarma. Basta con definir un threshold para compararlo con el resultado de la query y lanzar la correspondiente alarma.



Seleccionando Threshold y haciendo click en *Set as alert condition* en la parte superior de la expresión, vamos a lograr que si la query seleccionada supera cierto valor se lance la alerta correspondiente. Vemos que el estado de Normal representa que la query no arroja valores que cumplan la condición, por lo que no hay nada que alertar.

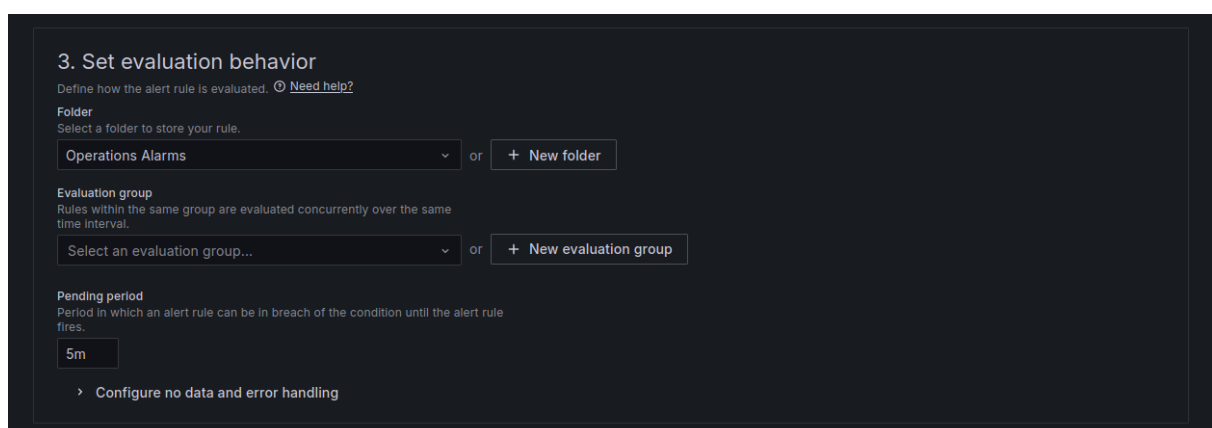


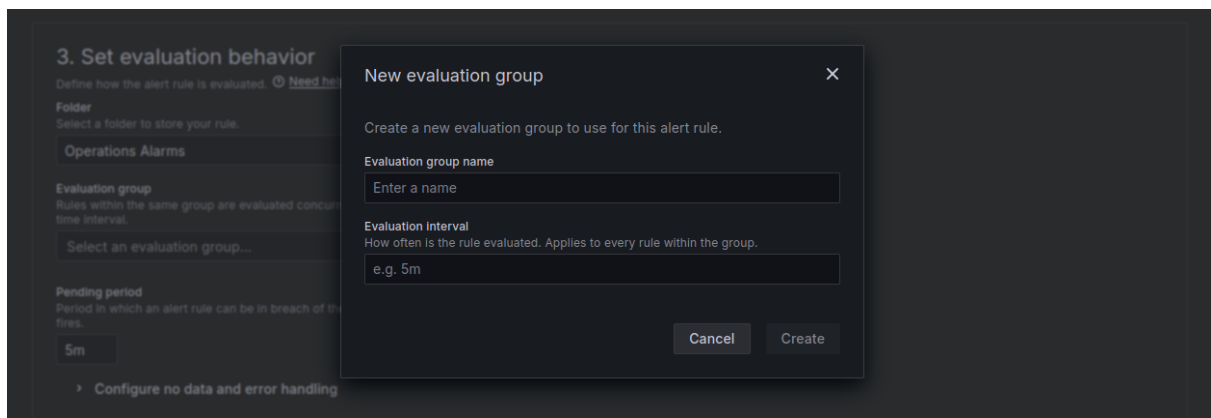
En cambio, si el estado es Firing, esto indicaría que la condición se cumple y se debe lanzar la alerta.



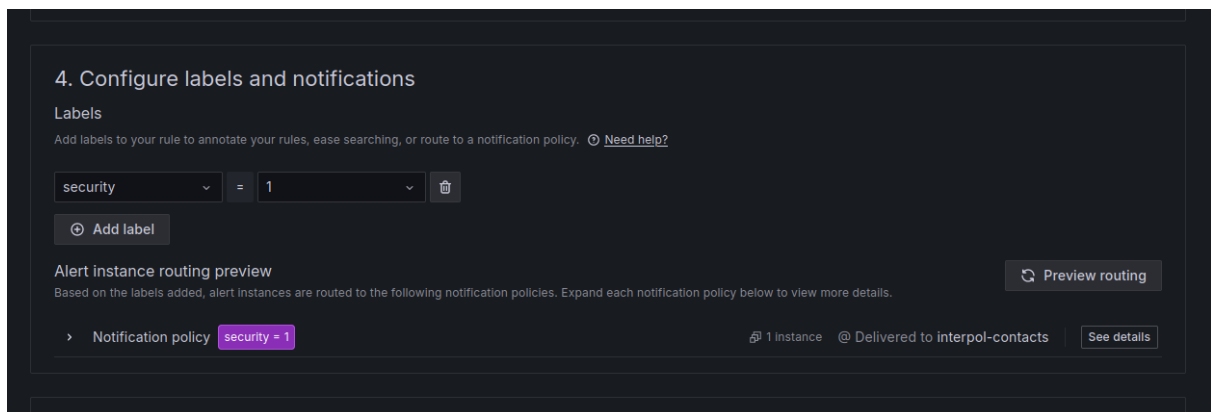
Por último, antes de guardar la alerta se debe configurar:

- Folder: La carpeta donde se guardara la regla que se acaba de configurar
- Evaluation Group: El grupo de evaluación del que formara parte. Las reglas dentro del mismo grupo se evalúan concurrentemente en el mismo intervalo de tiempo, que puede estar expresado tanto en minutos como en segundos.
- Pending Period: Es el tiempo en el que la regla puede trasgredirse sin lanzar la alarma. Pasado este período de tiempo la alerta es inminente. Este tiempo puede estar expresado tanto en minutos como en segundos.





Por último, debemos configurar las labels. Como dijimos anteriormente, son tags definidas en una Notification Policy. Es importante que pongamos tags que existan en alguna de esas políticas porque de lo contrario se enrutaran a la política default.



Por último podríamos agregar un mensaje custom a las alertas que se envían por mail, modificando el título y la descripción adecuada que expliqué qué generó la alarma.

