Expertise
and insight
for the future

Matti Holopainen

Monitoring Container Environment with Prometheus and Grafana

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

3.5.2021

Tekijä Otsikko Sivumäärä Aika	Matti Holopainen Monitoring Container Environment with Prometheus and Grafana 50 sivua 3.5.2021
Tutkinto	Insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Nina Simola, Projektipäällikkö Auvo Häkkinen, Yliopettaja

Insinöörityön tavoitteena oli oppia pystyttämään monitorointijärjestelmä konttiympäristön resurssien käytön seuraamista, monitorointia ja analysoimista varten. Tavoitteena oli helpottaa monitorointijärjestelmän käyttöönottoa. Työ tehtiin käytettävien ohjelmistojen dokumentaation ja käytännön tekemisellä opittujen asioiden pohjalta.

Insinöörityön alussa käytiin läpi työssä käytettyjä teknologioita. Tämän jälkeen käytiin läpi monitorointi järjestelmän konfiguraatio ja käyttöönotto. Seuraavaksi tutustuttiin PromQL-hakukieleen, jonka jälkeen näytettiin kuinka pystyttää valvontamonitori ja hälytykset sähköpostimuistutuksella. Työn lopussa käydään läpi kuinka monitorointijärjestelmässä saatua dataa analysoidaan ja mietitään miten monitorointijärjestelmää voisi parantaa.

Keywords	Monitorointi, Kontti, Prometheus, Grafana, Docker

Author Title Number of Pages Date	Matti Holopainen Monitoring Container Environment with Prometheus and Grafana 50 pages 3.5.2021
Degree	Bachelor of Engineering
Degree Programme	Information and Communication Technology
Professional Major	Software Development
Instructors	Nina Simola, Project Manager Auvo Häkkinen, Principal Lecturer

The goal of the study was to set up a monitoring stack for a container environment to monitor and analyze resource usage. The purpose was to make it easier to implement a monitoring stack. The study was carried out by applying knowledge gained by practice and getting familiar with the documentation of the software used.

First, the study introduces the current situation and then goes through the technologies used. Next, the study talks about setting up and configuring the monitoring stack is explained, followed by an introduction to the use of PromQL query language. Next, the study goes through how to set up a monitoring dashboard and how to get up alerts and email notifications. The last chapters shows how to analyze data from the monitoring stack and how to make imporve the monitoring stack

Keywords	Monitoring, Container, Prometheus, Grafana, Docker

Contents

List of Abbreviations

cAdvisor	container Advisor. Application collecting container metrics.
CNCF	Cloud Native Computing Foundation. A Linux Foundation project aiming to advance open-source usage of cloud and container technology
Container	A software unit where the application code and all its dependencies are packaged.
Docker	A Container platform that can build, store and run container images.
Grafana	Open-source Observability platform capable of visualizing metric data from multiple different sources.
Exporter	A software that collects metrics that Prometheus and then export.
Node-exporter	An exporter that collects OS and hardware metrics.
Prometheus	Open-source system monitoring and alerting toolkit.
PromQL	Prometheus Query Languages. The query language used for querying Prometheus.
Redis	An open-source in-memory structure store that can be used as a database or cache.
RHEL	Red Hat Enterprise Linux. Open-source operating system popular in enterprise use.
TSBD	Time Series Database. A database where the data is stored values as a stream of timestamped values.

1 Introduction

Containers are a very popular platform to run applications. Most of the applications run inside a container in a container environment. Therefore, it is a good idea to improve the ability to maintain a container environment.

When there is an issue in an application, one can use a stack trace and error logs to troubleshoot the issue. Stack trace shows the issue on the code level. However, issues in the application are not always on the code level, e.g. high memory use. Another issue with stack trace and error logs is that they require an error to trigger. In other words, something has to go wrong for them to appear. For example, if the database server runs out of disk space, the database crashes, and there is an error log about it. Of course, the database should always be up, and therefore there is a need to know when the disk space is low. To solve these issues, monitoring solutions are used to monitor the application and IT infrastructure.

Prometheus and Grafana are flexible tools for monitoring IT architecture, especially for containers and cloud environments. Prometheus supports metrics from multiple different exporters and offers powerful query language PromQL. With PromQL, one can efficiently analyze metric data from multiple sources. However, this flexibility results in a high learning curve and can be overwhelming for a beginner. Within the present study a simple monitoring stack for a Docker environment using Prometheus and Grafana was built. The goal was to make the reader familiar with Prometheus and Grafana and ready to make their monitoring stack.

First, the paper goes through the technologies used to get an idea of why they are used. Then the setting up of the monitoring stack is explained. Next, the paper introduces PromQL, the query language Prometheus uses. The metrics are used to creating a Grafana Dashboard. By creating a dashboard, one can quickly analyze the environment and see what is happening at one glance in real-time. The metric data is analyzed by using the dashboard. Lastly, the paper discusses how to improve the stack.

2 Container Environment and Monitoring

This chapter introduces monitoring metrics on a general level together with the container environments.

2.1 Monitoring Metrics

Metrics are units of standard measurement. For example, the memory usage of a host or an application, the time a web page takes to load or rate of successful API calls divided by failed API calls and many more. Monitoring metrics allows one to optimize the applications and IT infrastructure, and to identify and troubleshoot issues. For example, by monitoring the web page load times, one can identify a need to optimize it, improving the end-user experience. If a page takes an especially long time to load, it can consider an issue on itself and just something that improves an end-user experience. Another example is the memory use of an application. If an application uses a lot of memory on a server, it can slow down the server and therefore all the other applications on the server. In this case, it can be difficult to figure out which application causes the issue.

To collect monitor metrics, one has to put software on the server that collects the metrics. The name of the software depends on the monitoring solution used. Some call them exporter, others agent. There are different types of exporters or agents for different types of sources, for example, one for host metrics and another one for databases metrics, and yet another for Containers.

The metrics from the exporters or agents are then collected to a database for storage. An interface is used to analyze the collected metrics and configure alerts.

The exporter used depends on what is been monitored. If the host is monitored, one uses an exporter that collects metrics from the host. If the containers are monitored, one uses an exporter that collects metrics from the containers.

2.2 Container Environment

A container is a software unit where the application code and all its dependencies are packaged, making applications inside containers lightweight, quick, and easy to deploy. A container is also an isolated instance, making them secure. [1] When the environment is running containers, it is called a container environment. There are many container platforms. The container platform used in this study is called Docker.

The container environment has multiple applications running. Here is the problem. There is no way to track the CPU or memory usage or network traffic of the containers efficiently. One can only check current resources for one container at a time, making comparing resources usage between containers hard and knowing what a typical resource usage for a container is. Therefore, debug issues related to resource usage is difficult and time-consuming. For example, something in the environment uses a lot of swap, and there are too many containers to check manually. It is a needle in a haystack situation. Increasing the amount of memory or swap is not a solution either since it would only delay the appearance of the issue.

In order to solve the issue, within the study a metric monitoring stack for the environment was built. The monitoring stack collects the CPU, memory, and swap usage as well network traffic metrics from containers as well as from the server itself.

3 Using Grafana and Prometheus for Monitoring

In this study, a Grafana metrics stack was built. Grafana metrics stack uses Prometheus as a database that collects the data from exporter and Grafana for visualization. Exporter collects metric data that Prometheus can export from them. For exporters, Node-exporter was used for kernel and hardware metrics and cAdvisor for container metrics and Prometheus.

3.1 Prometheus and PromQL Query Language

Prometheus is the leading open-source monitoring system and a time-series database capable of scraping metrics from multiple sources. A time-series database is a database where the values are stored as a stream of timestamped values [2]. Prometheus was originally developed by SoundCloud [2], a music and audio platform [3]. Currently, Prometheus is a standalone project with active developers and a user community. Prometheus became part of CNCF or Cloud Native Computing Foundation in 2016 as the second-hosted project hosted by the foundation [2].

Figure 1 shows the architecture of Prometheus.

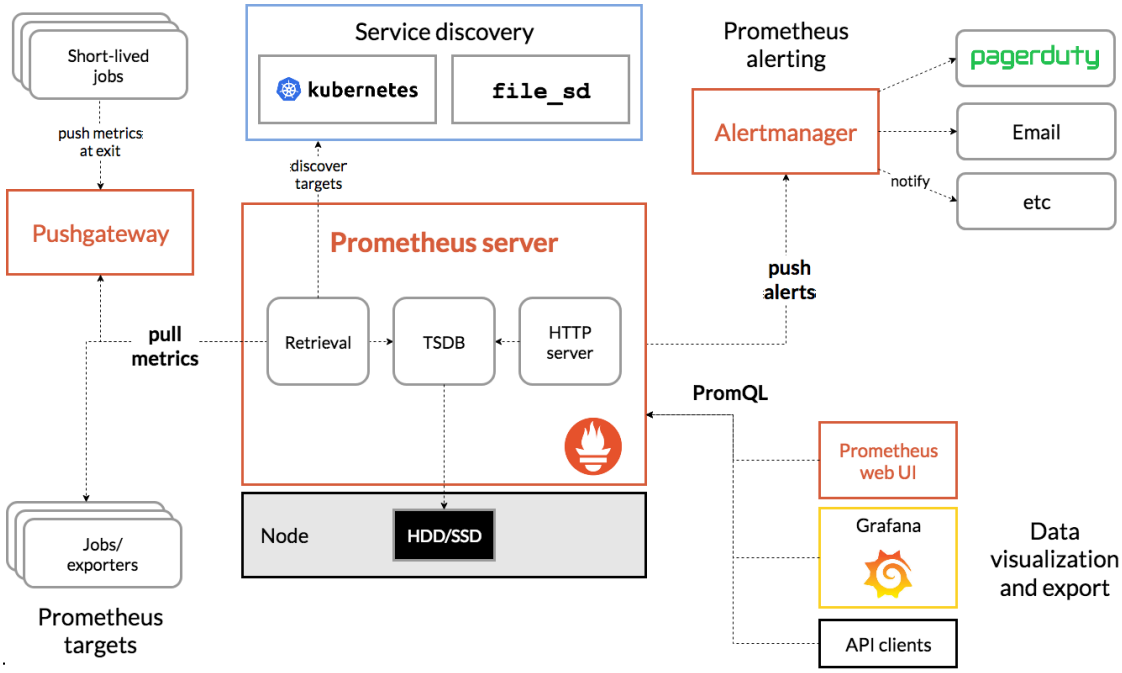


Figure 1. Prometheus architecture.[2]

Prometheus scrapes metrics directly from an exporter, e.g. Node-exporter, or from Pushgateway. Using exporters and Pushgateway, Prometheus can get metrics from multiple different sources. The Prometheus can be configured to pull metrics statically using the exporters or Pushgateway address or dynamically using the discovery target. Prometheus uses times series or streams of timestamped values, which belong to the same metric and key/value pair creating a multidimensional data model [2]. Storing values as streams of timestamps makes Prometheus database a time series database or TSDB.

Prometheus uses its query languages called Prometheus Query Language or PromQL. PromQL allows users to query metrics from Prometheus in real-time and view the query result in a graph or tabular format [4]. One can use Prometheus web UI, Grafana, or API clients to execute PromQL queries in Prometheus. PromQL uses metric names and labels in its query, making it very flexible but hard to learn.

The alerts can be set in Prometheus and use Alertmanager to send notifications to multiple different platforms, e.g. email and slack. In the monitoring stack, Grafana is used for alerts instead of Alertmanager and Prometheus.

3.2 Grafana: Open Observability Platform

Grafana is an open-source platform for observing and monitoring metrics. With Grafana, one can query and visualize metric data and set up alerts [5]. Grafana supports many different data sources, e.g. Prometheus, Google Cloud Monitor, Elsaticsearch, InfluxDB and MySQL

Grafana can create dynamic and reusable dashboards. Dashboards allow end-user to see the situations of the environments at a glance. End-users can share and import Dashboard using JSON or Grafanas' website [6], making Dashboards easy to share and fast to set up if one already knows a dashboard that fits the needs.

Grafana can visually configure alerts using dashboards [5]. Grafana can send notifications from these alerts to a different system, e.g. Microsoft Teams, Google Hangout, Discord and Email.

Grafana supports plugins developed by Grafana Labs and Grafana Community. Using these plugins, Grafana can visualize the metric data using panels and support more data sources. [7].

3.3 Container Advisor: Getting Resources of Containers

Container Advisor or cAdvisor is a running daemon that collects, processes, aggregates, and exports metrics from running containers. With cAdvisor, one can understand the performance and resource usage of the containers. [8]

cAdvisor supports Docker containers natively and should support other container types like containerd and cri-o. [8]

cAdvisor is compatible with Prometheus, and this study used it as an exporter for container metrics.

3.4 Redis In-memory Store

Redis is an open-source in-memory structure store that can be used as a database or cache. Redis offers many data structures such as streams, strings, hashes, and sets [9]. An in-memory database is extremely fast since it does not need to read from the hard drive making them a great choice when the application needs a cache.

Redis is used as a cache for cAdvisor data until Prometheus collects it.

4 Setting up Monitoring Stack

The monitoring stack was set up on a Docker container environment with RHEL 7.9 Maipo as its operating system. First, the paper discusses the monitoring stack architecture and file structure. Next, it introduces the set up for the containers and configuration files for Prometheus and Grafana

4.1 Architecture

Figure 2 shows the monitoring stack architecture.

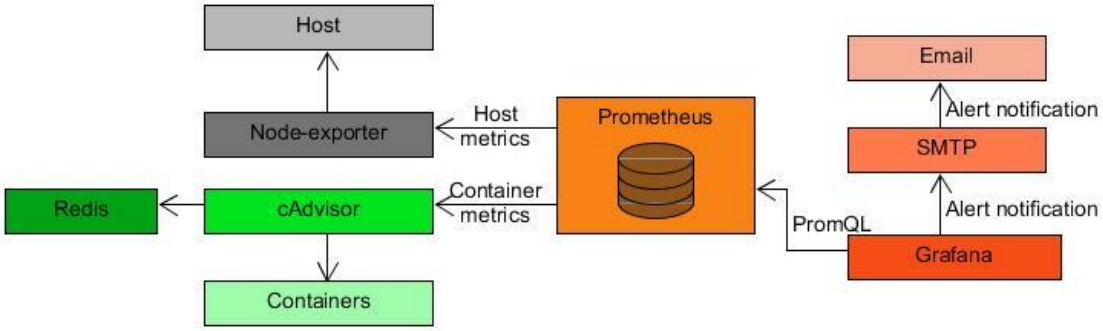
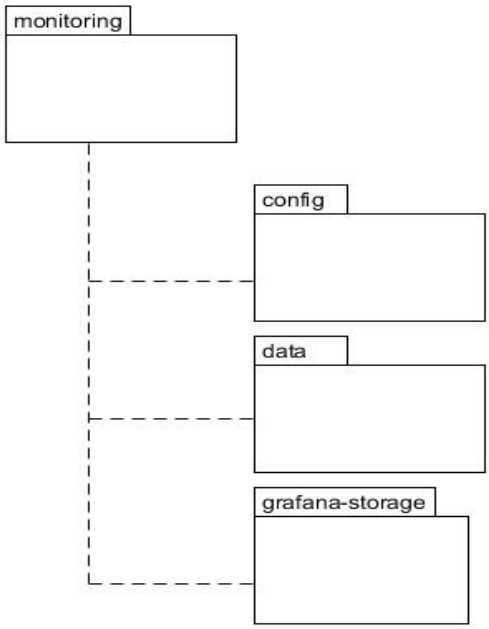


Figure 2. Monitoring stack architecture.

The monitoring stack uses cAdvisor to collect metrics from containers and Redis to cache them. For hardware and OS metrics, the stack uses Node-exporter. Prometheus pulls the data from cAdvisor and Node-exporter and stores it. Grafana queries the data from Prometheus using PromQL and visualizes it. The monitoring system uses Grafana for alerts, and Grafana sends an email notification when there is an alert.

4.2 Setting up

The study began by creating a file structure. Figure 3 shows the structure.



Project file structure.

Figure 3.

The monitoring folder serves as the root folder for the monitoring stack. Inside the monitoring folder, there are the following folders: config, data, and grafana-storage. The config folder contains configuration files for Prometheus and Grafana. The data folder holds the data stored by Prometheus and grafana-storage, the data stored by Grafana.

The environment, the monitoring stack monitors and where the monitoring stack is set up is a Docker environment that uses docker-compose to run the containers. Therefore, the monitoring stack was set up using a similar setup as the rest of the environment In other words, the monitoring stack is inside containers, and docker-compose is used to run them.

The configuration of the containers started by creating a `docker-compose.yml` file. The study used compose file version 2.2 because the rest of the docker-compose files in the environment uses that version.

The following lines were added to the docker-compose.yml file:

```
version: "2.2"
services:
```

Keyword `version` defines the docker-compose file version used, and `services` keyword defines the start of the services block.

4.3 Setting up cAdvisor

cAdvisor setup uses two containers. The first one is an in-memory database called Redis and the second one is the cAdvisor itself.

4.3.1 Redis Container

Redis was configured as temporary storage and linked to cAdvisor. Redis configuration looks as follows:

```
redis:
  image: redis
  ports:
  - "6397:6397"
```

The first line declares a service called `redis`. The `image` keyword tells Docker to use `redis` image and `ports` keyword to open container port 6397 and direct traffic from that port to Redis port 6397.

4.3.2 cAdvisor Container

Ath the cAdvisor github page [8] the default Docker configuration for the cAdvisor looks as follows:

```
cadvisor:
      image: gcr.io/cadvisor/cadvisor:latest
      volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
      ports:
      - "8080:8080"
```

However, the Docker configuration for cAdvisor is heavily depended on the operating system and the Docker version used [10]. This means that the is differently depending on the operating system and Docker version. For this study the configuration when running on RHEL 7.9 and Docker version 1.13.1 end up looking as follows:

```
cadvisor:
      image: gcr.io/cadvisor/cadvisor:latest
      privileged: true
      volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw

      - /sys/fs/cgroup/blkio:/sys/fs/cgroup/blkio:ro
      - /sys/fs/cgroup/cpuset:/sys/fs/cgroup/cpuset:ro
      - /sys/fs/cgroup/devices:/sys/fs/cgroup/devices:ro
      - /sys/fs/cgroup/freezer:/sys/fs/cgroup/freezer:ro
      - /sys/fs/cgroup/hugetlb:/sys/fs/cgroup/hugetlb:ro
      - /sys/fs/cgroup/memory:/sys/fs/cgroup/memory:ro
      - /sys/fs/cgroup/net_cls,net_prio:/sys/fs/cgroup/net_cls,net_prio:ro
      - /sys/fs/cgroup/pids:/sys/fs/cgroup/pids:ro
      - /sys/fs/cgroup/systemd:/sys/fs/cgroup/systemd:ro
      - /sys/fs/cgroup/cpu,cpuacct:/sys/fs/cgroup/cpuacct,cpu:Z

      - /sys/fs/selinux:/sys/fs/selinux:ro
      - /sys/fs/ext4:/sys/fs/ext4:ro
      - /sys/fs/fuse:/sys/fs/fuse:ro
```

```
    - /sys/fs/pstore:/sys/fs/pstore:ro
    - /sys/fs/bpf:/sys/fs/bpf:ro

    - /sys/block:/sys/block:ro
    - /sys/bus:/sts/bus:ro
    - /sys/class:/sys/class:ro
    - /sys/dev:/sys/dev:ro
    - /sys/devices:/sys/devices:ro

    - /sys/firmware/acpi:/sys/firmware/acpi:ro
    - /sys/firmware/dmi:/sys/firmware/dmi:ro
    - /sys/firmware/memmap:/sys/firmware/memmap:ro

    - /sys/hypervisor:/sys/hypervisor:ro
    - /sys/kernel:/sys/kernel:ro
    - /sys/module:/sys/module:ro
    - /sys/power:/sys/power:ro

    - /var/lib/docker/:/var/lib/docker:ro
    - /dev/disk/:/dev/disk:ro

    ports:
    - "9091:8080"

    depends_on:
    - redis
```

The first thing to note is how long the `volumes` configuration became. Instead of adding volume `/sys:/sys:ro` it was necessary to go inside the `sys` file structure and only mount the volumes cAdvisor has access to. It was necessary since cAdvisor is dependent on the operating system and Docker version. Another important note is the `privilege: true` line. RHEL and CentOS locks down their containers more tightly than others, so this line is necessary to access Docker daemon [10]. The `ports` configuration opened port 9091 on the container and directed the traffic to the cAdvisor default port 8080. The port 8080 on the container is not used since another application has taken port 8080 on the server. The last lines link Redis to cAdvisor and make cAdvisor depend on it [11]. In other words, if the Redis container does not go up, neither does the cAdvisor container.

4.4 Setting up Node-exporter

Setting up node-exporter was quite straightforward. It required to add the following lines to the compose file under services:

```
node-exporter:
  image: prom/node-exporter:latest
  cap_add:
  - SYS_TIME
  ports:
  - "9100:9100"
```

The `cap_add` flag was added with the value `SYS_TIME` to get access to time adjustment metrics since the container is running on Linux.

4.5 Setting up Prometheus

Setting up Prometheus required two things: configuring Prometheus itself and configuring Prometheus container.

4.5.1 Configuring Prometheus

First, a file called `prometheus.yml` was created under the config folder. This file is the configuration file for Prometheus.. The following lines were added to the file:

```
global:
 scrape_interval: 5s # By default, scrape targets every 15 seconds.

scrape_configs:
  - job_name: 'prometheus'
    scrape_interval: 5s
    static_configs:
    - targets:
      - localhost:9090

  - job_name: 'cadvisor'
    scrape_interval: 5s
    static_configs:
    - targets:
      - cadvisor:8080

  - job_name: 'node-exporter'
    scrape_interval: 5s
    static_configs:
    - targets:
      - node-exporter:9100
```

The first two lines are part of the global configuration block. The `scrape_interval` inside the global block was used to define the global scrape interval [12]. Scrape interval defines how often Prometheus scrapes metrics from the targets. Below the global configuration block, the scrape configs configuration block begins[12]. Inside scrape configs, `job_name` is used to name the jobs. Jobs is a collection of instances with a same purpose. The instance is the endpoint which Prometheus scrapes The `scrape_interval` was used to overwrite the global scrape interval for the job. Inside `static_configs` block, the `targets` keyword is used to list the instances for the job [12]. For this study, the Prometheus configuration had three jobs with one instance each. One job for Prometheus itself called

`prometheus` with instance `localhost:9090`, another for called `cadvisor` with instance `cadvisor:8080` and the last one called `node-exporter` with instance `node-exporter:9100`. Take note of the targets for cAdvisor and node-exporter. Instead of an actual address, the names used were ones used to link them to Prometheus in the `docker-compose.yml` file.

4.5.2 Prometheus Container

Setting up Prometheus inside a container was very simply. It was done by adding the following lines to the compose file under services:

```
prometheus:
   image: prom/prometheus:latest
   command:
   - --config.file=/etc/prometheus/prometheus.yml
   volumes:
   - /monitoring/config/prometheus.yml:/etc/prometheus/prometheus.yml:ro
   - /monitoring/data:/prometheus/data:Z
   ports:
   - "9090:9090"
   depends_on:
   - cadvisor
   - node-exporter
```

The `volumes` mount the configuration file to the Prometheus container and Prometheus database to the `data` folder. The Prometheus database was mounted to the `data` folder in order not to lose the data if there were a need to recreate the Prometheus container. The command `config.file=/etc/prometheus/prometheus.yml` tells Prometheus to use the configuration file mounted to it as its configuration file.

4.6 Setting up Grafana

Setting up Grafana requires three things: change user and group of the grafana-storage folder, configure container for Grafana, and configure Grafana itself.

When Grafana is running inside a container, it uses user id 472. For Grafana to access the grafana-storage folder, therefore it was necessary to change the folder's ownership by using the following command:

```
chown 742:742 grafana-storage.
```
[14]

Grafana container configuration goes as follows:

```
grafana:
  image: grafana/grafana:latest
  restart: always
  volumes:
  - ./grafana-strorage:/var/lib/grafana:Z
  - ./config/grafana.ini:/usr/share/grafana/conf/defaults.ini:Z
  ports:
  - "3000:3000"
  links:
  - prometheus:prometheus
```

By default, there is no need to configure Grafana. It works out of the boxes as it is. However, since email notifications are used for alerts, the default has to override.

A file called “grafana.ini” was created inside the config folder. The default Grafana configuration from Grafana’s GitHub pages [15] was copy-pasted to this file and use it as a template. The part edited looked as follows in the default file:

```
[smtp]
      enabled = false
      host = localhost:25
      user =
      # If the password contains # or ; wrap it with triple quotes. Ex
"""#password;"""
      password =
      cert_file =
      key_file =
      skip_verify = false
      from_address = admin@grafana.localhost
      from_name = Grafana
      ehlo_identity =
      startTLS_policy =
```

The enable was set to “true” and SMTP address as host value.

5 PromQL Query Language

PromQL or Prometheus Query Language is a query language that is used to query Prometheus for the time series Prometheus has collected. To use Prometheus, one must know how to write PromQL. First, the paper goes through the data types and how to get

metrics. Next, the paper introduces the instant vector selector followed by the range vector selector and offset modifier. Lastly, the paper discusses the operators and functions.

5.1 Data Types

Prometheus Query Language can evaluate three different type expressions: instant vectors, range vectors, and scalar vectors. An instant vector is a set of time series containing a single sample for each time series, all sharing the same timestamp. A range vector is a set of time series containing a range of data points over each time series. Scalar is a simple numeric floating-point value. [4]

5.2 Getting Metrics

The metric name is used to query Prometheus. The metric names depend on the scrape target Prometheus uses to get the metrics e.g. data from node-exporter starts with the prefix node_ and data from containers starts with the prefix container_. To find a metric name, Prometheus graphical interface can be used. Prometheus graphical interface is accessed by using a web browser. The default address for the Prometheus graphical interfaces is localhost:9090. To find a metric, the drop-down menu below the query field can be used to see all the metric names or by typing a keyword in the query field and seeing the suggestion Prometheus gives.

For example, by typing “http” in the query field, one can find a metric that returns the total HTTP request that Prometheus gets. (Figure 4)

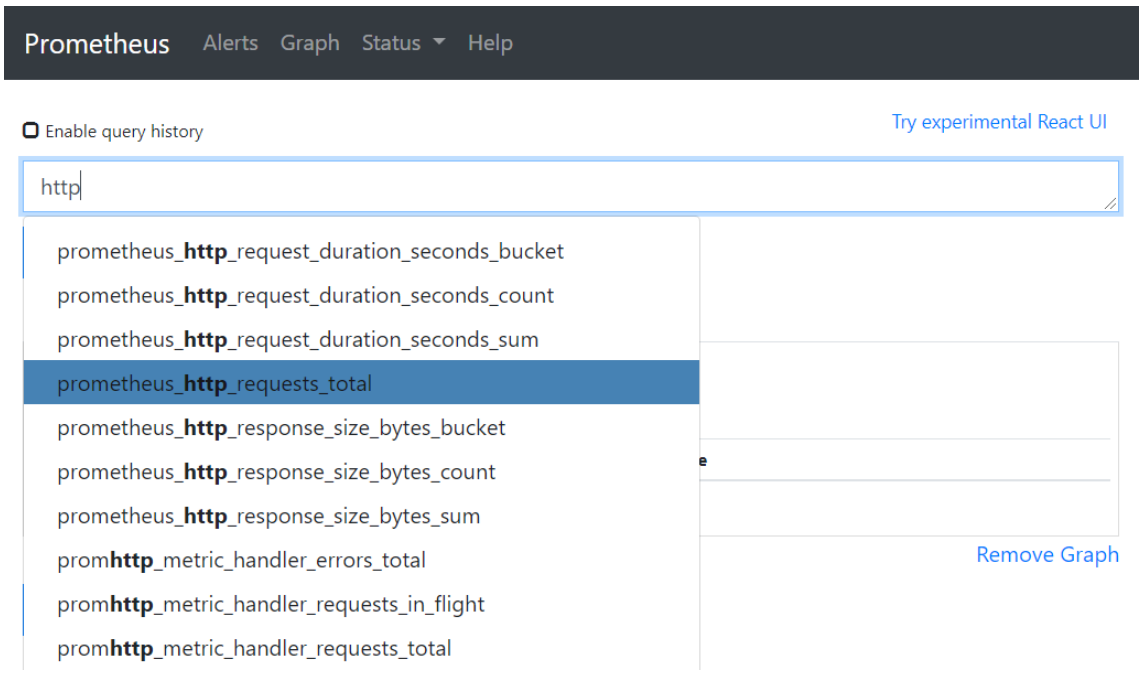


Figure 4. Metrics Prometheus founds using “http” keyword.

The keyword brings up many different metrics with very self-exploratory names. The query:

```
prometheus_http_requests_total
```

returns the total HTTP request that Prometheus gets. (Figure 5)

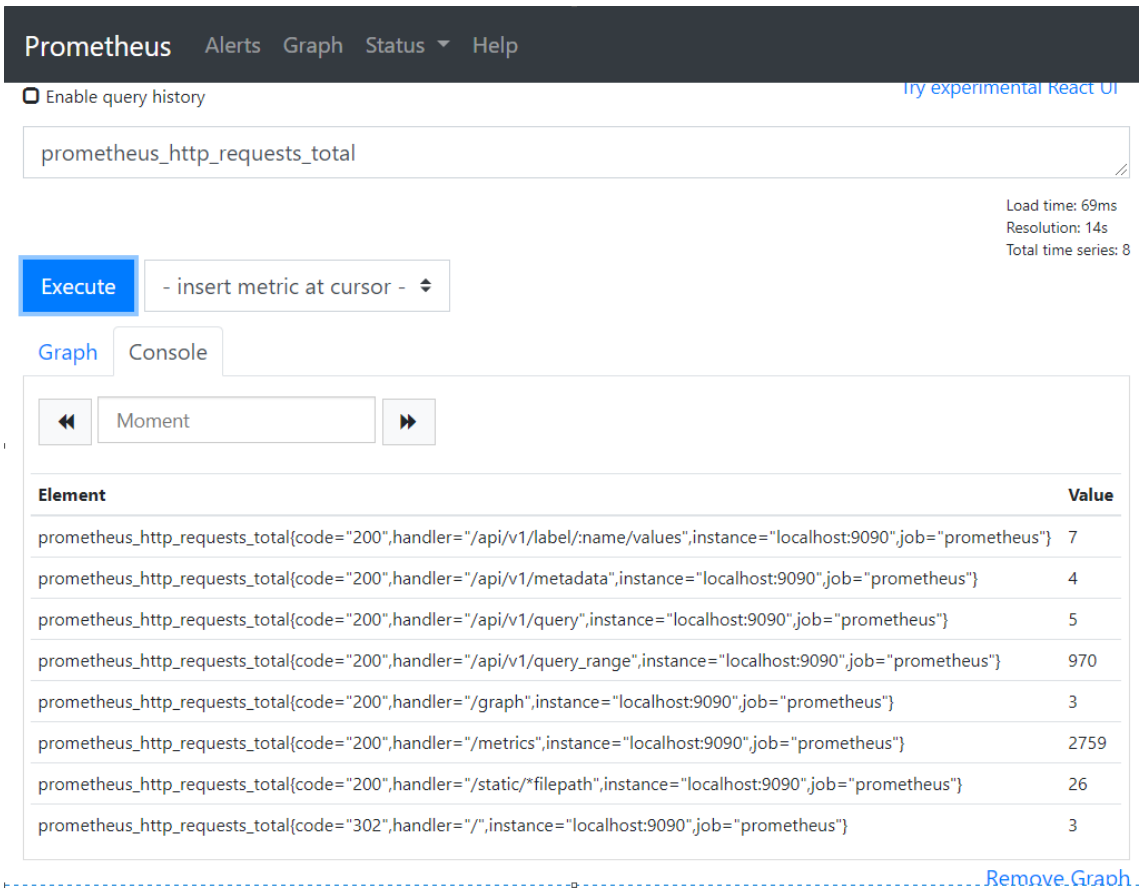


Figure 5. prometheus_http_request_total query results

Prometheus has separated the results by using multiple key/value pairs known as labels. The labels are given after the metric name inside the curly brackets.

5.3 Instant Vector Selector

In most cases, one does not want all these metrics but instead metrics with specific labels. That is why one can use instant vector selectors.

Instant vector selector is where one puts a label in the query after the curly brackets' metric name.

The following query:

```
prometheus_http_requests_total{handler="/api/v1/query"}
```

returns the number of HTTP requests for the “/api/v1/query” endpoint. (Figure 6)

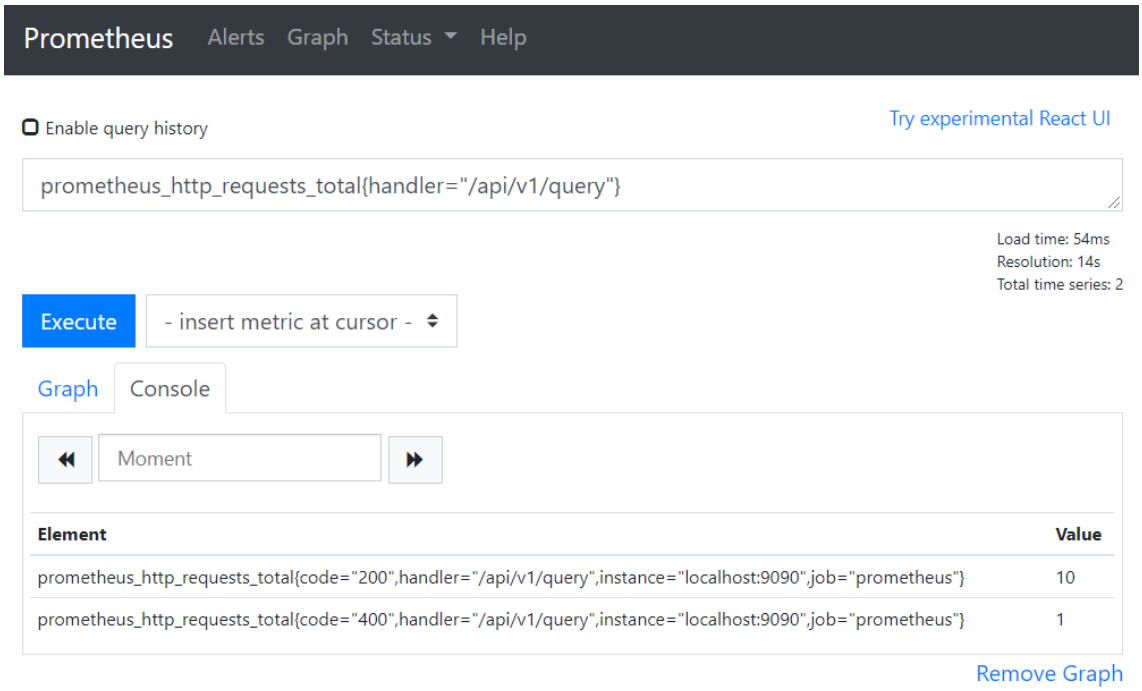


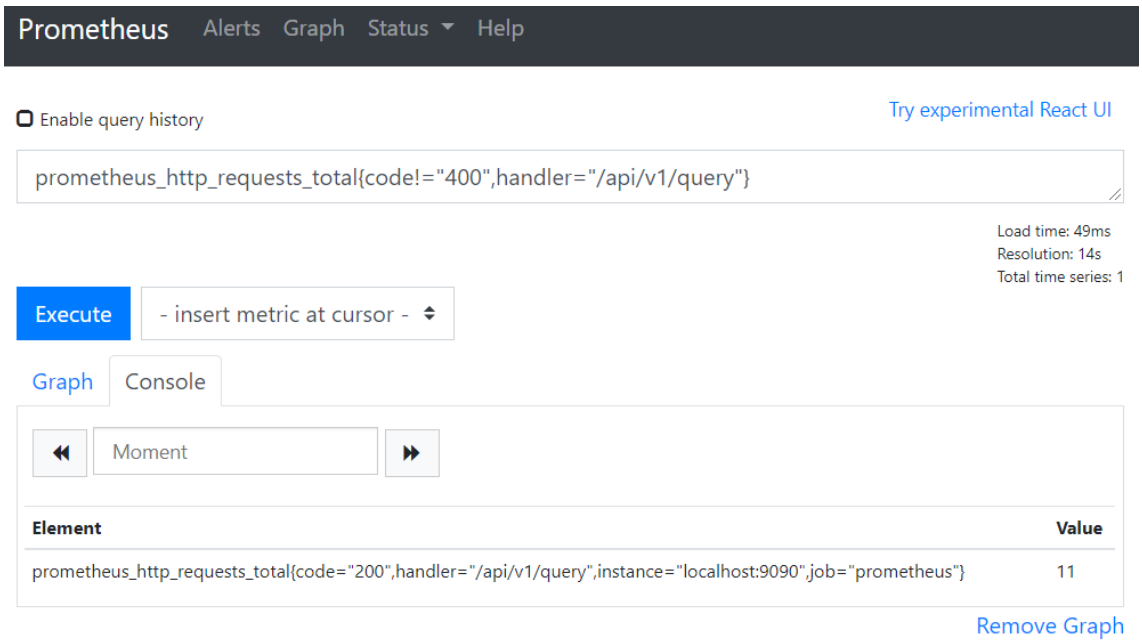
Figure 6. Results of prometheus_http_request_total query using handler label.

Instant vector selectors can also be used to exclude label values using the “!=” operator. [4]

The following query:

```
prometheus_http_requests_total{code!=”400”,handler="/api/v1/query"}
```

excludes the results where the HTTP code is 400. The results can be seen in Figure 7.



Result of the PromQL query excluding HTTP requests with HTTP code 400.

Figure 7.

In the two previous queries, Prometheus was looking for an exact match to either include or exclude them in the result. Prometheus can also match patterns by using regular expressions and the “=~” operator to include and the “!~” operator to exclude label values. [4]

The following query:

```
prometheus_http_requests_total{ handler=~"/api/v1/.*"}
```

returns every metric from every endpoint that starts with `/api/v1/` (Figure 8).

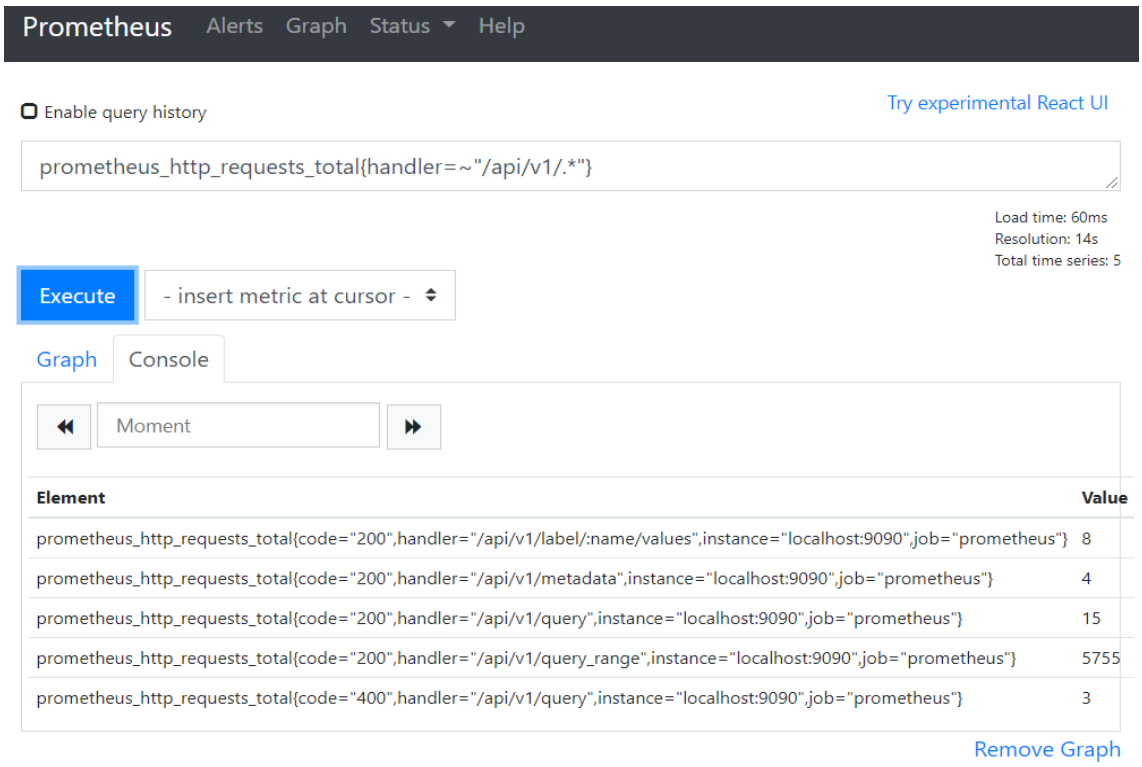


Figure 8. PromQL query results for HTTP requests from endpoints starting with "/api/v1/".

Figure 8 shows that instead of getting a result from the handler label with `/api/v1/` Prometheus returns the results where the handler starts with `/api/v1/`.

5.4 Range Vector Selector

Every query shown so far has used a time range from data collection beginning to the current time. However, this is not always wanted. To use a range vector selector, one simply adds a numeric value with a time unit inside square brackets to the query.

The following query

```
prometheus_http_requests_total{code="200",handler="/api/v1/query"}[10s]
```

shows the total number of HTTP requests with HTTP code 200 from endpoint "/api/v1/query" from 10 seconds ago to the current moment. (Figure 9)

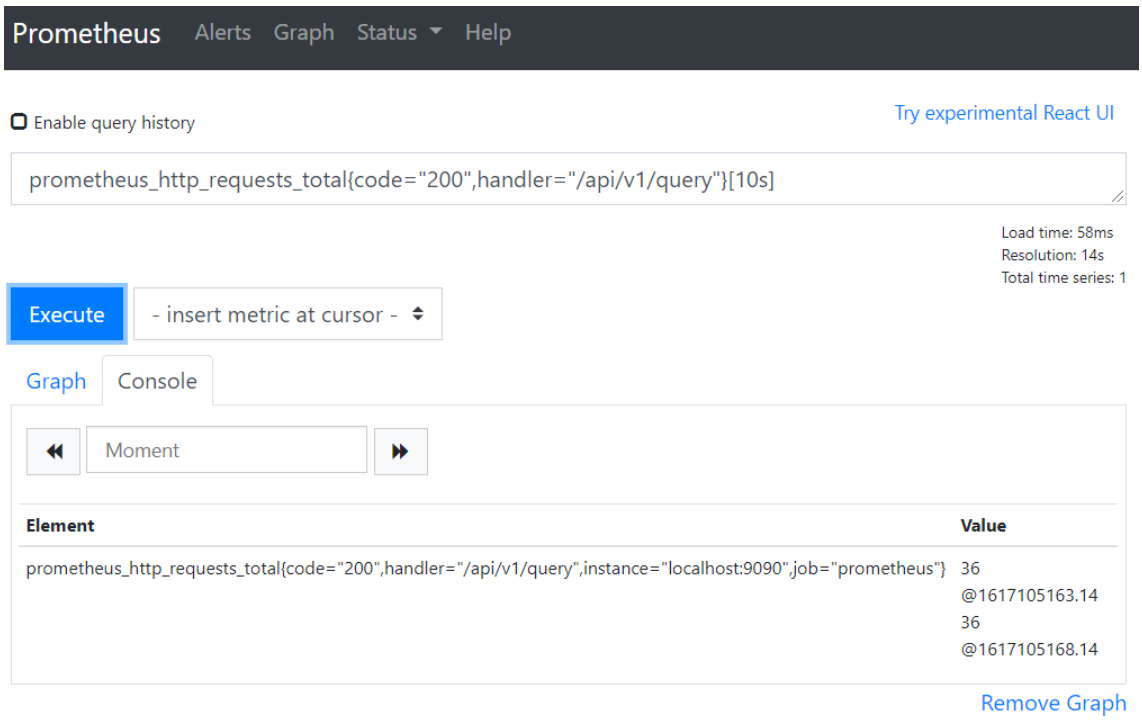


Figure 9. Example of range vector query results.

The query returns two different values, with the difference being after the @ sign. The first number is the number of HTTP requests and the second one is the timestamp in epoch time. Prometheus has been configured to collect data every 5 seconds, so it returns two different timestamps when the time range is set to looked 10 seconds in the past.

Range vectors have the following time units [4]

- y – years
- w – weeks
- d – days
- h – hours
- m – minutes
- s – seconds
- ms – milliseconds.

It is important to note that Prometheus query language always assumes that a year is 365 days, a week 7 days, and a day 24 hours.

These time units can combine instead of using bigger numeric values. For example, using [1h30m] instead of [90m}.

5.5 Offset Modifier

With the range vector selector, one can choose the desired the time range, but the end time is always the current time, which is not always what is wanted. For this reason, the offset modifier is used. The offset modifier allows one to change the time offset of the query relative to the current time by adding the word offset to the query and add a time value the same way one would add inside the square brackets when using a range vector.

The following query

```
prometheus_http_requests_total{handler="/api/v1/query"} offset 1d
```

checks the number of HTTP requests from the endpoint "/api/v1/query" one day ago (Figure 10).

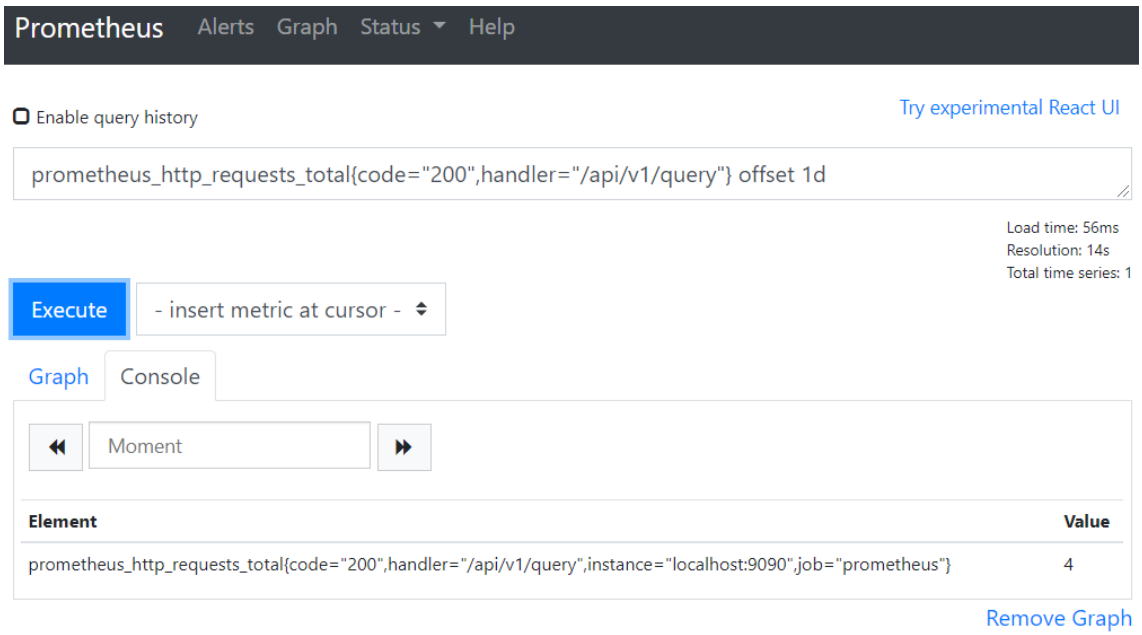


Figure 10. Offset modifier query result example.

Figure 10 shows that the value was 4. By contrast, in Figure 8 where the endpoint "/api/v1/query" was in the results, the value was 15.

5.6 Operators

The operators PromQL uses do not differ from the primary programming languages convention, + for addition, - for subtraction, == for equal, etc [16].

On top of regular operators, there are many built-in aggregation operations, for example, sum() to calculate the sum of a dimension and avg() to calculate the average of the dimension.

For example the following query

```
sum(prometheus_http_requests_total)
```

calculates the sum of all HTTP requests Prometheus has received (Figure 11).

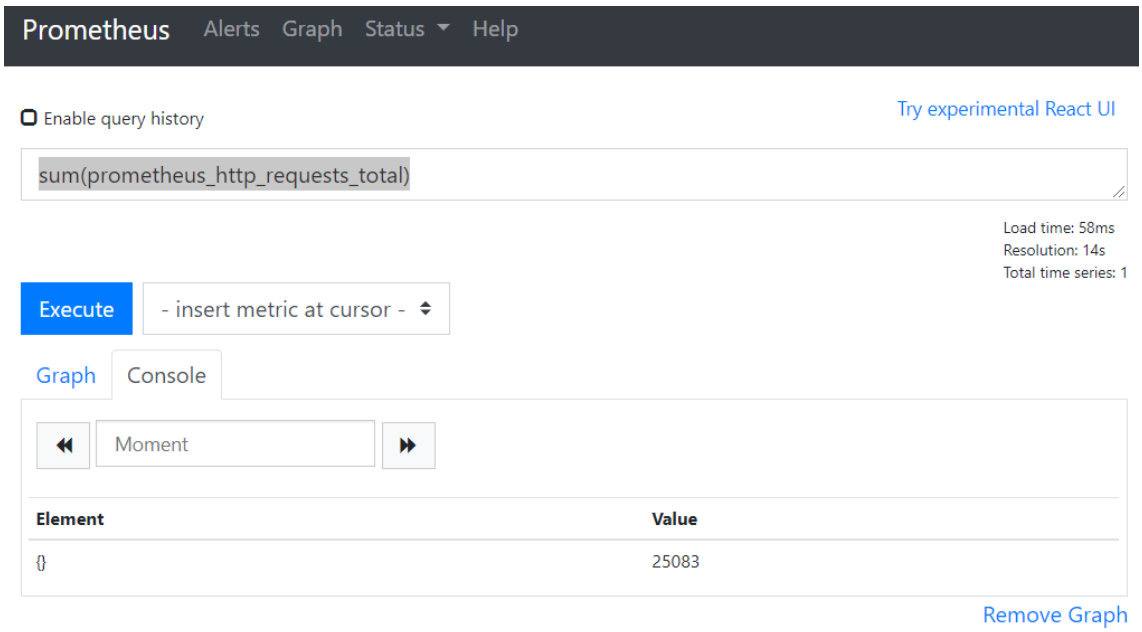


Figure 11 Results for the sum() aggregation operator example query.

By looking at Figure 11, one can note that instead of metric names and labels, the `Element` field has `{}` which is caused by the `sum` operator. The `sum` operator has calculated the sum of all the metrics and labels and combined them in a single value.

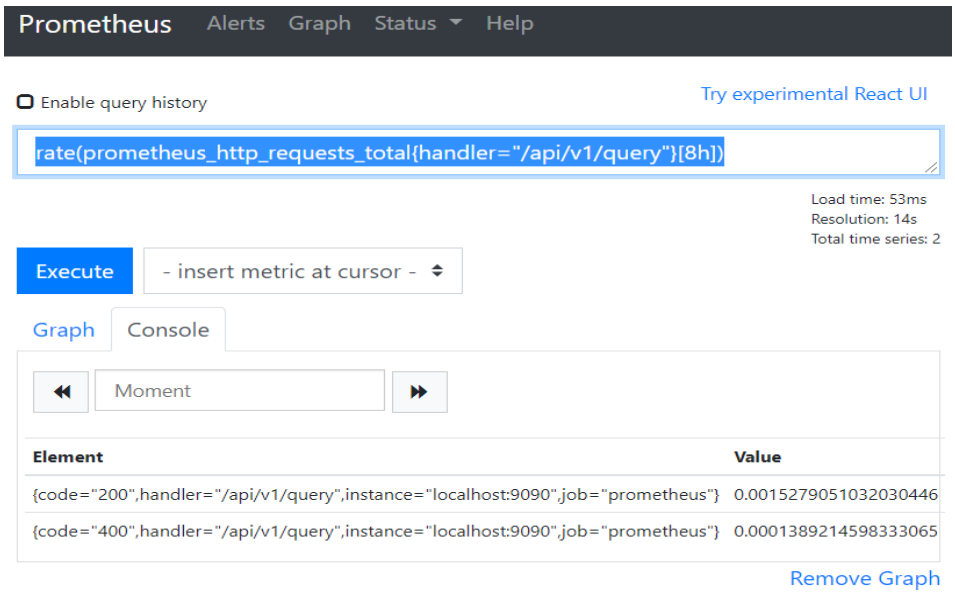
5.7 Functions

The Prometheus query language offers many functions to help to analyze the metric data, which are used in the same way one would use any function or method in programming languages[17]

For example the rate() function wants a range vector as a parameter [17]. The following query

```
rate(prometheus_http_requests_total{handler="/api/v1/query"}[8h])
```

returns the per-second rate of an HTTP request for the endpoint "/api/v1/query" over the last 8 hours (Figure 12).



Result of the rate function example query.

Figure 12.

In Figure 12 one can see the rate at the value column.

6 Configuring Grafana Dashboards

So far, this paper has discussed how to set up monitoring and introduce PromQL. The following chapter discusses how the metric data was visualized using Grafana dashboard and how the alerts were set up.

Grafana is entered by going into the browser and enter the server address and the port Grafana is running. For example, “localhost:3000”. First, Grafana wants the end-user to log in in order to access Grafana front page. Figure 13 shows the Grafana front page.

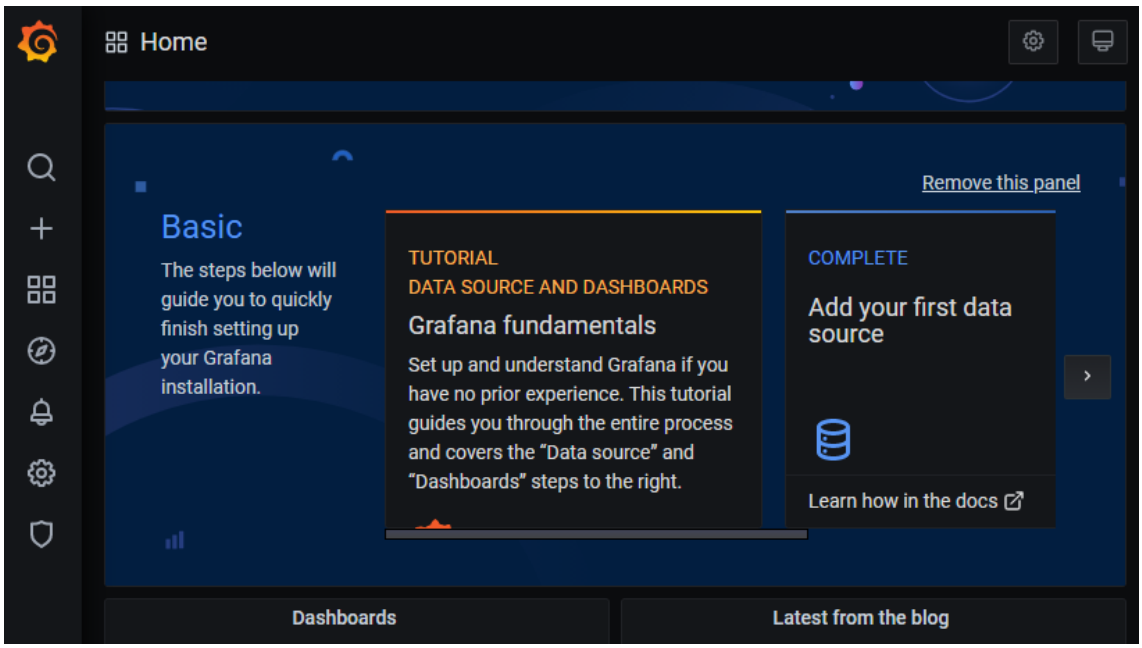


Figure 13. Grafana front page.

The menu bar on the left side of the screen is the primary way to navigate through Grafana.

6.1 Adding Data Source

Before configuring any Dashboards, a Data Source has to be added. Data sources are added through the "Add data source" page (Figure 14), which is accessed from the data source settings.

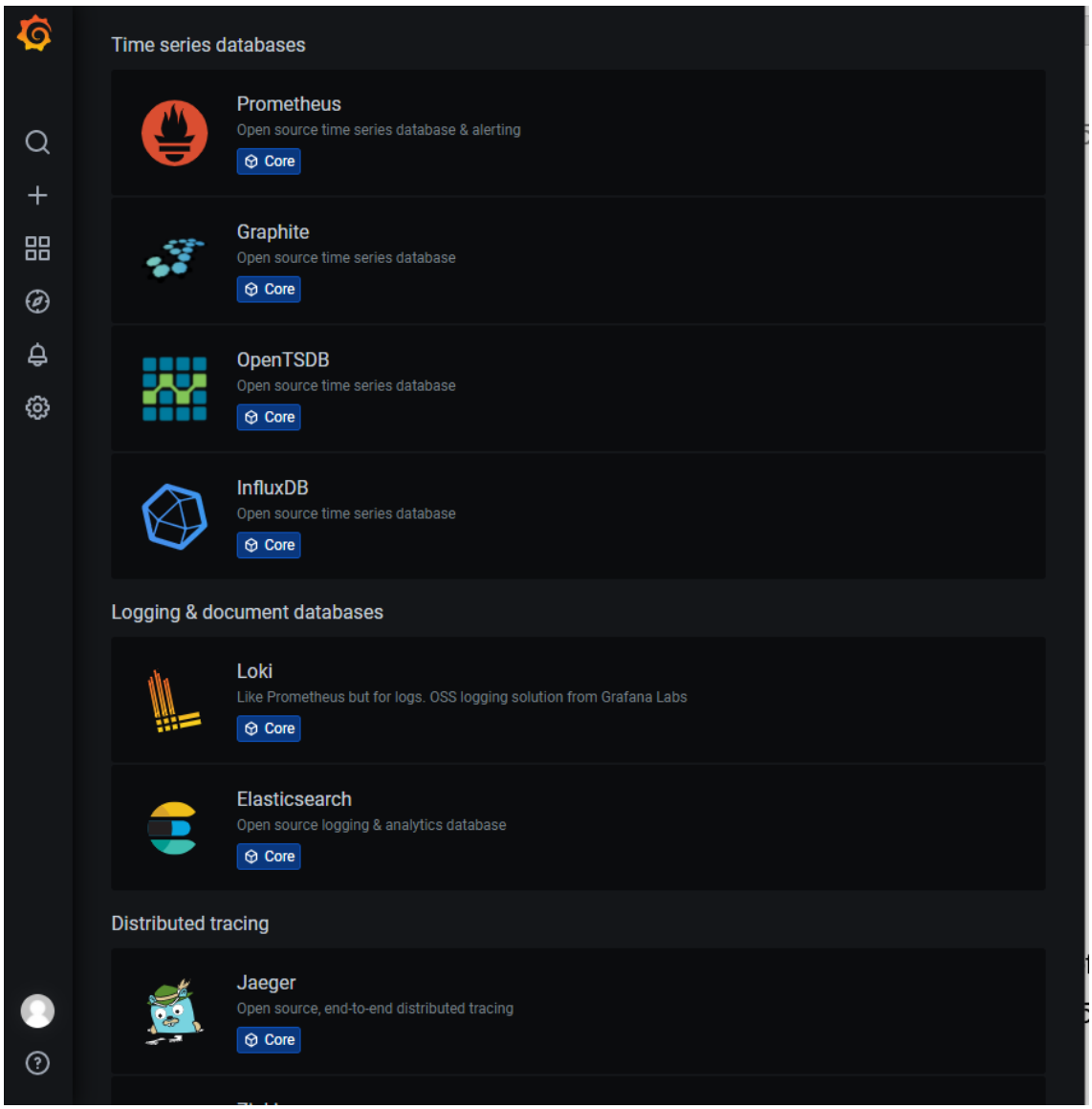


Figure 14.

Grafana Add data source page.

The “Add data source” page shows several different data sources Grafana supports out of the box and it is the page where one chooses which data source is configured. Figure 15 shows the Prometheus data sources configuration.

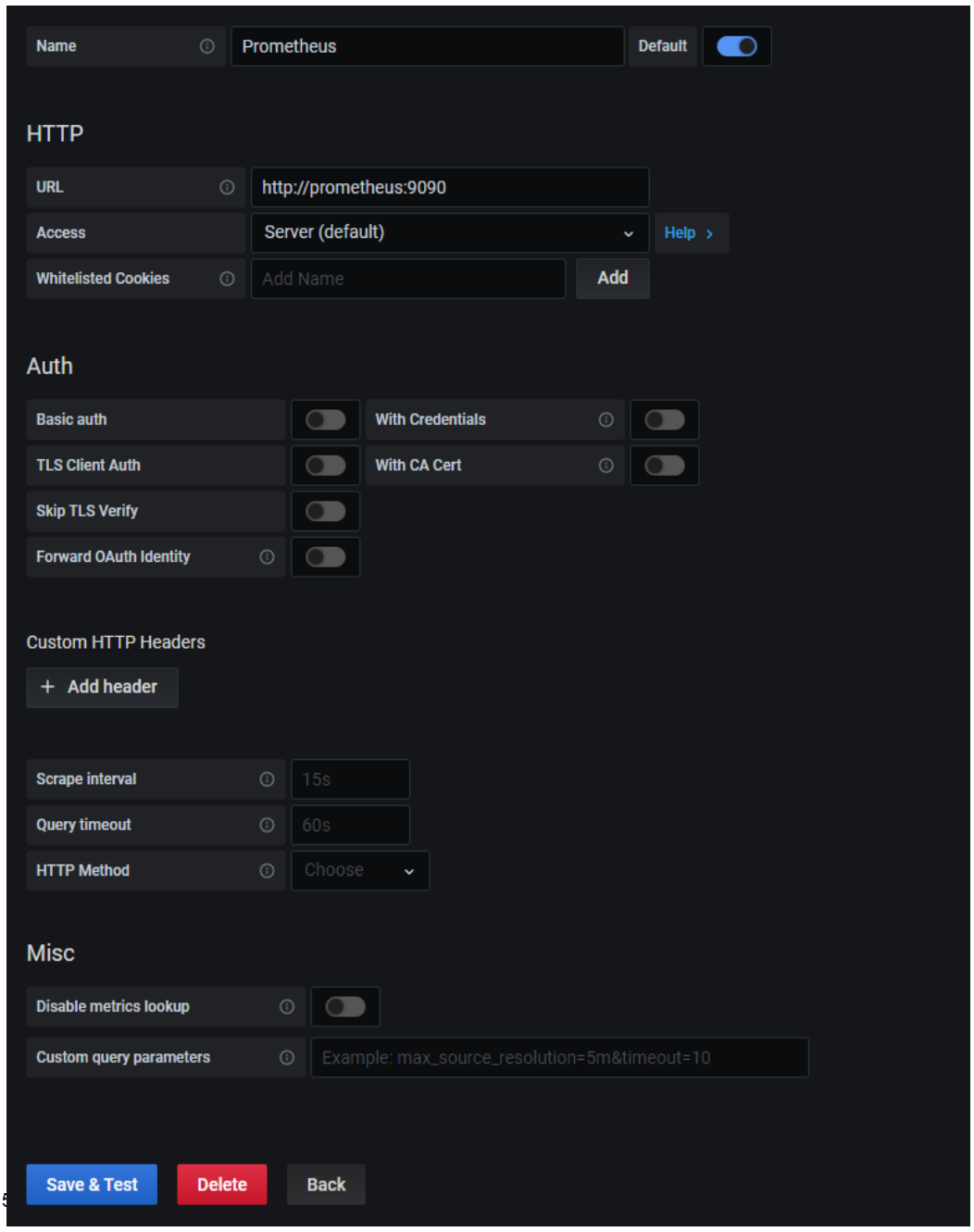


Figure 15. Configured Grafana Prometheus data source

In this context, the address is `http://prometheus:9090` since Docker was used to link the Prometheus to Grafana. Next, there are authorization options where one can, for example, use username and password with `Basic auth` and certification using `With CA Cert`. Prometheus and Grafana run on the same server. Therefore, there was no need for authorization. Custom HTTP Header options allow Grafana to send custom HTTP headers with the requests, which is helpful if Grafana has to send its message through

a load balancer. `Scrape interval` tells Grafana how often Prometheus scrapes its targets. The `Query timeout` tells Grafana how long it should wait to respond from Prometheus before timeout.

6.2 Configuring Dashboard

When the data source was configured, the configuration of the dashboard could start. On the menu bar, by hovering the moves over the plus image and clicking dashboard, a new empty dashboard is created. By clicking the “Add new panel” button, the edit panel view can be entered (Figure 16).

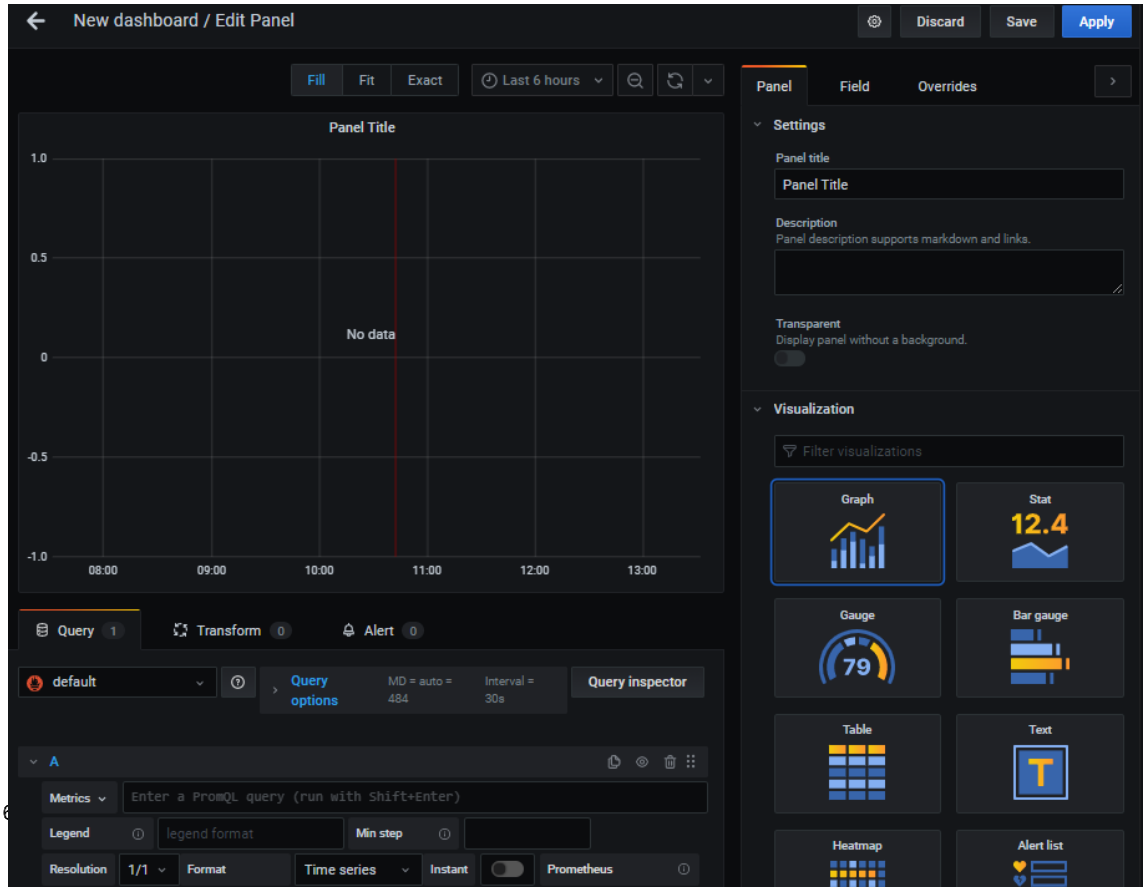


Figure 16. Grafana panel editor.

On the right are the visualization options. On the left, the view of the panel which is being configured, and below is the query editor.

6.2.1 Uptime Panel

The first panel configured shows the server uptime. To get the server uptime, the following query was used:

```
time()- node_boot_time_seconds{instance="node-exporter:9100"}
```

The `time()` is a Prometheus function that returns the current epoch time. The `node_boot_time_seconds` returns the last boot time as epoch time. Therefore by server uptime is the result of subtracting the `node_boot_time_seconds` value from the time() value. The instance label is set with the value of `node-exporter:9100` to tell Prometheus to get the data from the target `node-exporter:9100`.

Next, the panel was made visually easy to understand. The panel is made to look like the one in figure 17

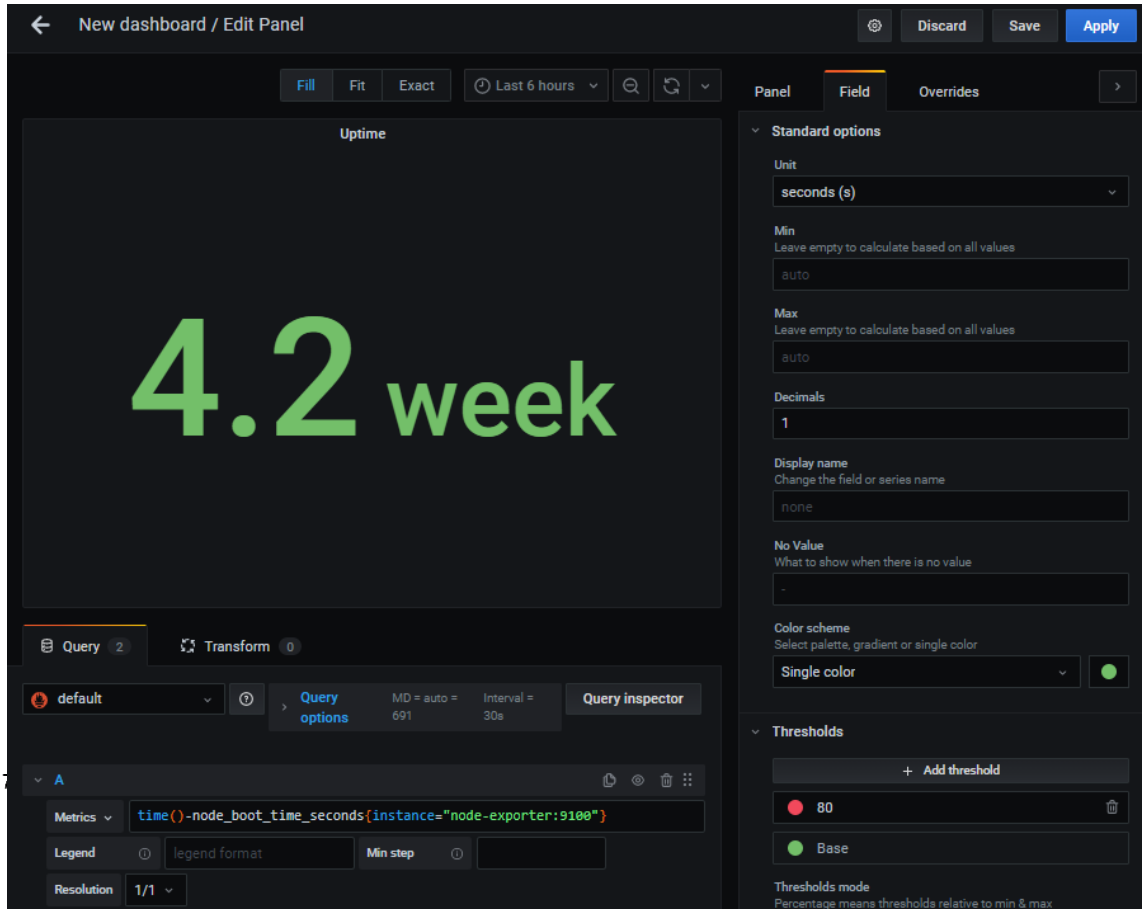


Figure 17

Configure Uptime panel.

The panel title was changed to “Uptime,” the visualization option chosen was Stat, and the Graph mode was set to “None.” Then in the Field tab the unit was set to seconds and decimals to one. The Color scheme was also changed to a single color of green.

6.2.2 CPU Usage Panel

The following query was used to get the CPU usage:

```
100 - (avg (irate(node_cpu_seconds_total{mode="idle",instance="node-ex-
porter:9100"}[5m])) * 100)
```

The query `node_cpu_seconds_total` returns the number of seconds the CPU has used to do different work types. The `mode` label was used with the value `idle` to get only the second when the CPU is idle.[5m] range vectore selector turns from instant vector to range vector, the variable type function `irate()` wants. The function irate calculates the per-second value for the query it is given. The avg is an aggregation operator that calculates the average over dimensions. In this case, it calculates the average rate for all the CPU cores. The average rate is between zero and one, multiplied by 100 to turn it into a percentage. The query calculates the percentage CPU is idle, so by subtracting the query from 100, the CPU usage comes out as a percentage.

Making the panel visually easy to understand, the panel was made to look like one in Figure 18.

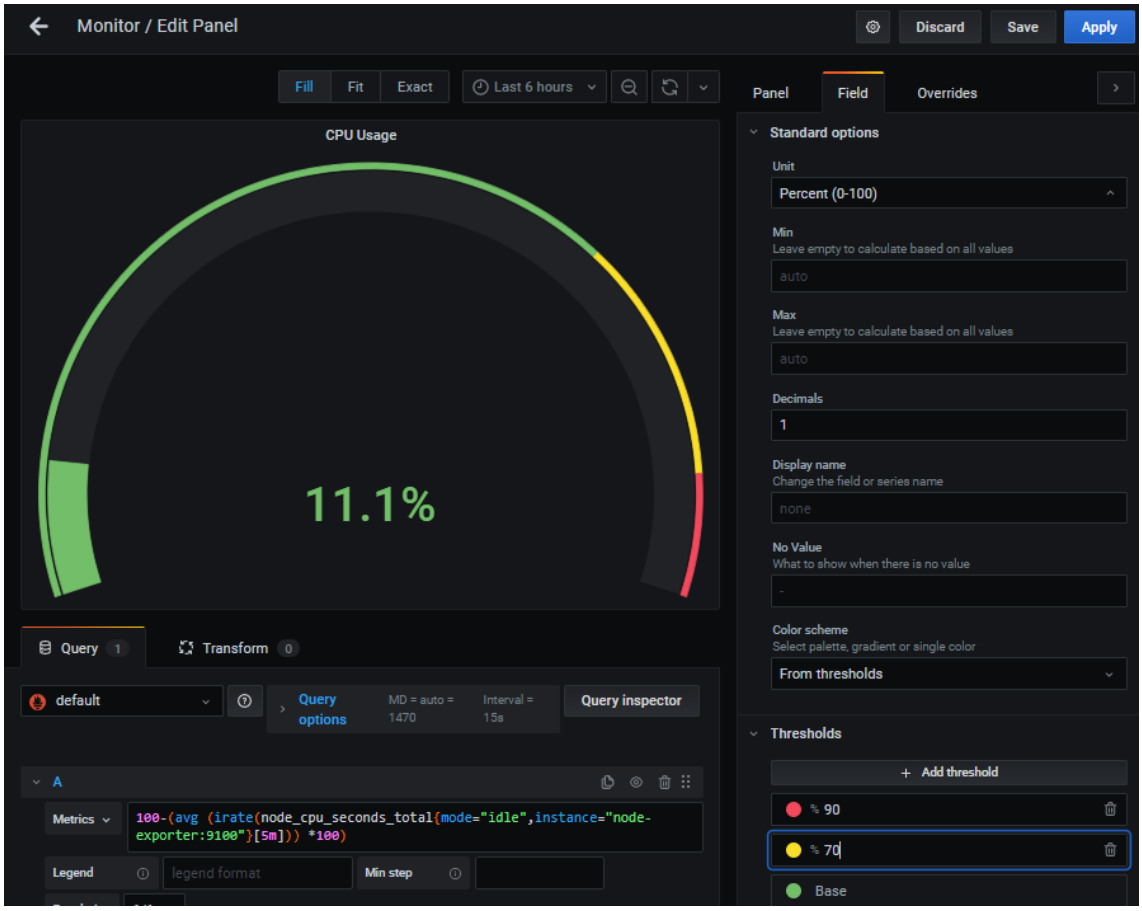


Figure 18. Configured CPU usage panel.

The panel was given the title “CPU Usage.” For the visualization option, the Gauge was chosen. On the Field tab, the unit was set to Percent (0-100), and decimals was set to one. The threshold values 90 and 70 were added and the threshold mode was set to a percentage. Threshold 90 was set to red and 70 to yellow. Grafana uses thresholds to change the color of the panel depending on the metric value.

6.2.3 Configuring Memory Usage Panel

The memory usage panel is visually similar to CPU Usage. Therefore the memory usage panel configuration started by duplicating the CPU Usage panel.

To calculate the memory usage percentage, the following equation was used:

$$(\frac{(total\ memory - memory\ in\ use)}{total\ memory} * 100)$$

In PromQL the equation looks as follows:

```
(((node_memory_MemTotal_bytes{instance="node-exporter:9100"} -
node_memory_MemAvailable_bytes{instance="node-exporter:9100"}) /
node_memory_MemTotal_bytes{instance="node-exporter:9100"}) * 100)
```

By replacing the CPU usage query with the query above, the panel looked as in Figure 19.

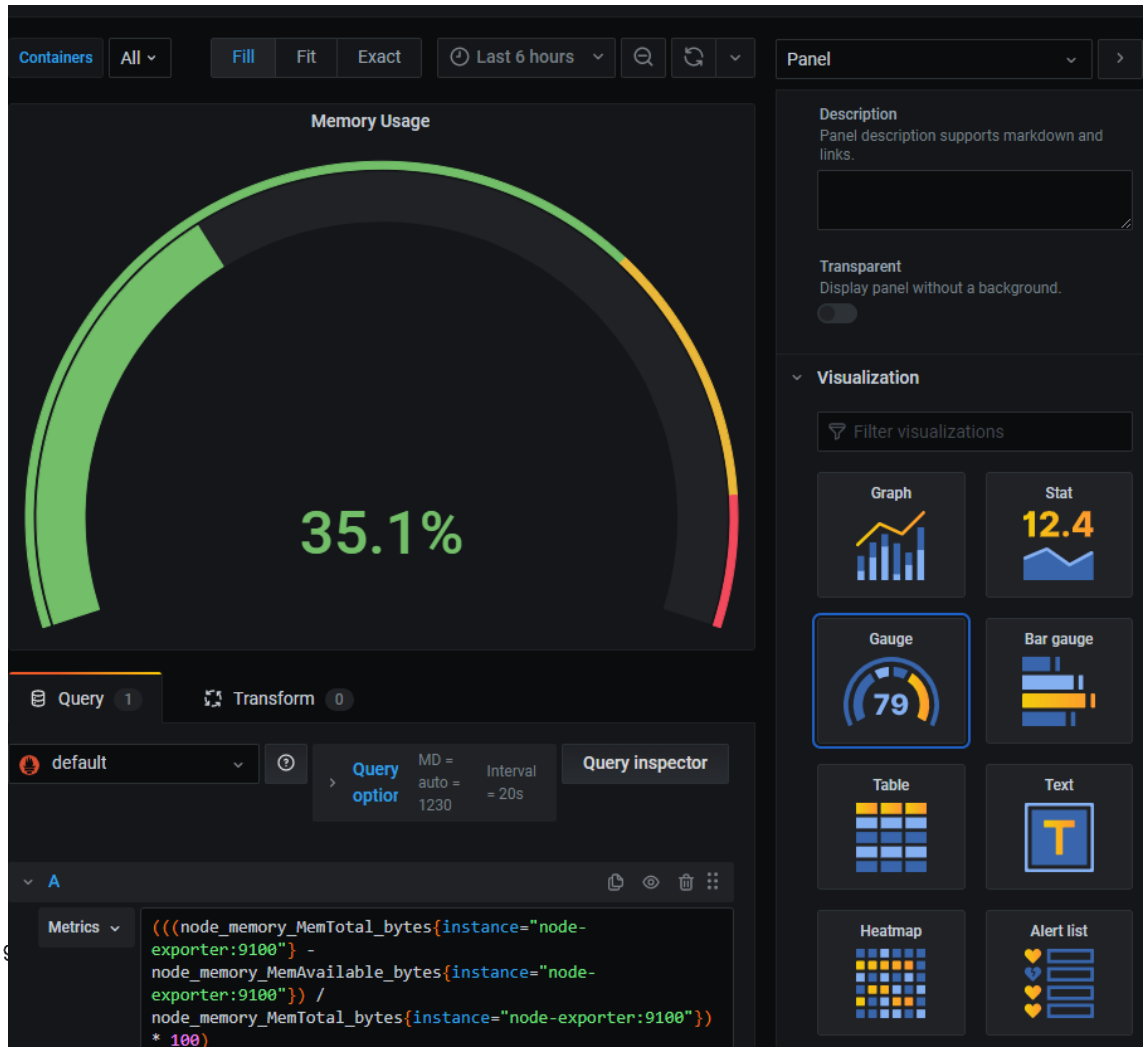


Figure 19 Configured Memory usage panel

By comparing Figure 19 and 18, one can note that difference with these panels are the query used and the panel title.

6.2.4 Network Traffic Panel

The network traffic panel shows both the received and transmitted traffic. All the applications in the environment are inside the containers. Therefore, the network metrics from cAdvisor were queried using the following query:

```
sum(rate(container_network_receive_bytes_total{id="/",instance="cadvisor:8080"}[5m]))
```

The query returns the sum of the incoming network of containers. The query was given a legend name “Received” in the legend field to distinguish this graph from the transmitted graph, which was done next.

A new query was added to the panel. “Sent” was written in the legend field and the following query was given to the new field:

```
- sum(rate(container_network_transmit_bytes_total{id="/", instance="cadvisor:8080"}[5m]))
```

By comparing the previous two queries, one can note the different metric names and the minus sign at the beginning. The minus sign in front of the query is there on purpose. This way, the incoming and outgoing traffic can be easily seen in the panel since incoming has a positive value and outgoing has a negative value.

The unit was set to bytes/sec(IEC), and the panel looks as in Figure 20.

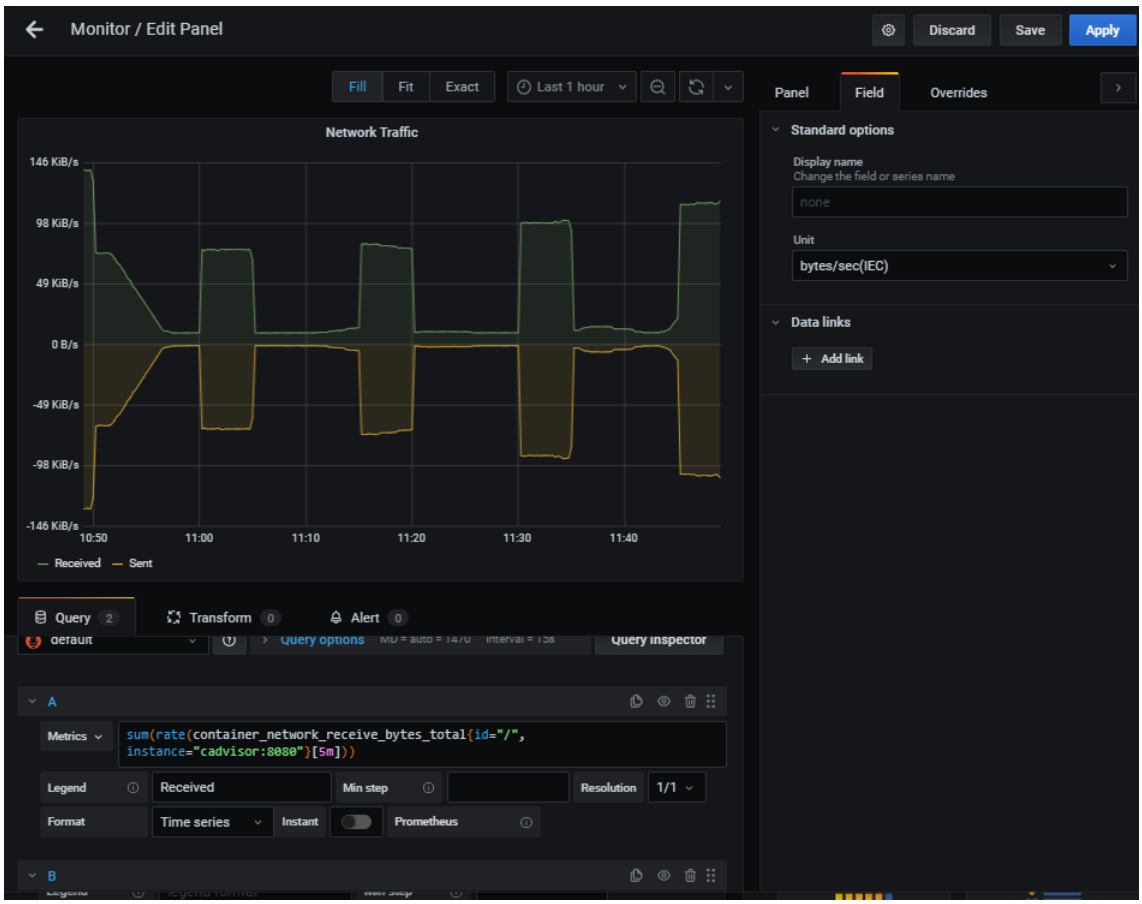


Figure 20. Configured Network traffic panel.

One can note how easily the incoming and outcoming traffic can be seen on the panel since the other one has positive value and other negatives.

6.3 Configuring Dashboard per Container Panels

For containers, the dashboard should visualize the network traffic, memory usage and CPU usage per container. Visually these panels are similar, with the main difference being the query. Therefore, creating one of the panels and duplicating it was the most efficient way to configure them. There was also a need to filter the data on these panels base on the containers. For this reason, a variable was created.

6.3.1 Variable

Variable is a value that can be changed quickly to filter what is seen on the panels. Variables are created through the dashboard settings by selecting variables on the left side menu and click the “Add variable” button. Figure 21 shows the configuration page.

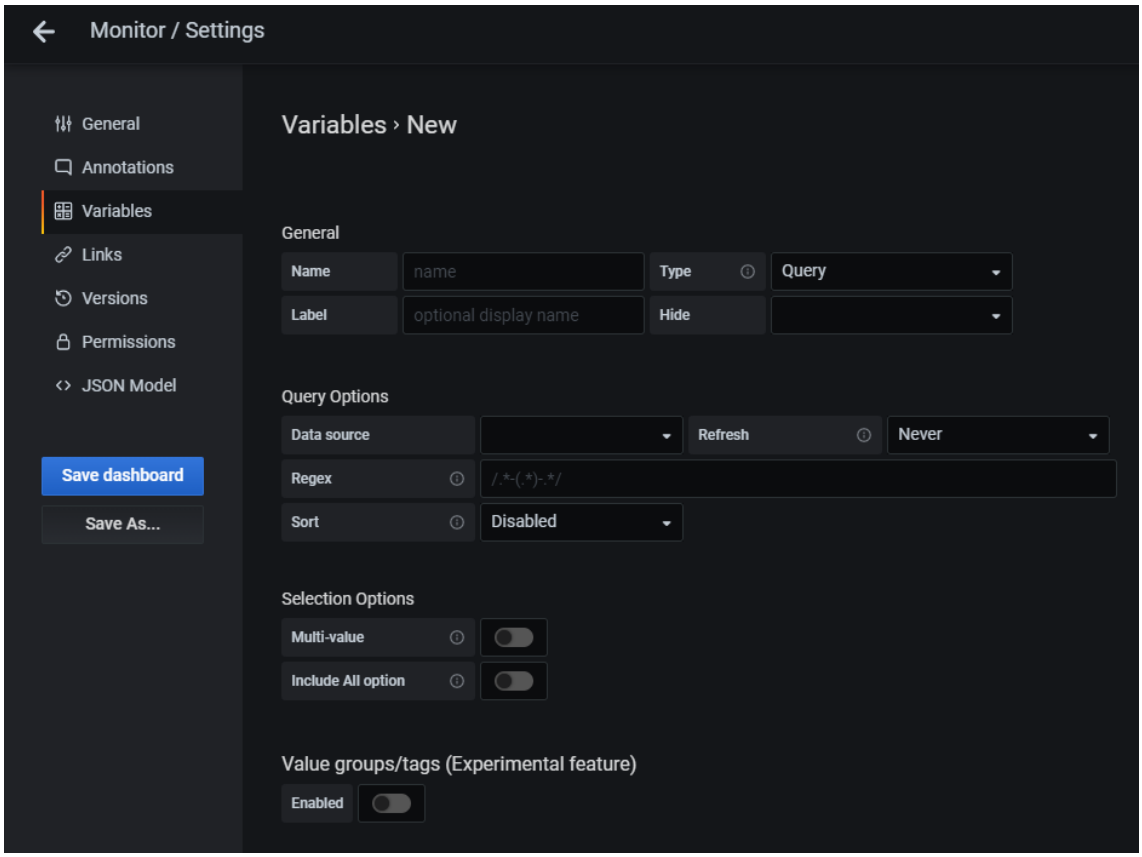


Figure 21. Variable configuration pages.

The ”Name” field is the variable name used in the queries. For this dashboard, it was set to “container”. The label is the display name for the variable. It was set to “Containers.” The data source was set to Prometheus, which enabled the Query field. For the query,: `label_values(container_label_com_docker_compose_service)` was used. This query returns every docker-compose service name cAdvisor knows. Refresh was set to “On Time Range Change”. “Multi-value” and “Include all options” were enabled in order to be able to select multiple containers at once and have one value the represents all the containers.

6.3.2 Received Network Traffic per Container Panel

A new panel with the name “Received Network Traffic per Container” was created. The following query was used to get receive network traffic metrics:

```
rate(container_network_receive_bytes_total{name!="", container_la-
bel_com_docker_compose_service=~"$container", instance="cadvisor:8080"}[5m])
```

The `container_label_com_docker_compose_service` label was used in the query and it was given the container variable value by using $ sign the same way one would in the shell script.

`{{name}}` was added to the legend field. It sets the container names as graph names. Finally, the panel unit was set to bytes/sec(IEC). The panel looks as in Figure 22.

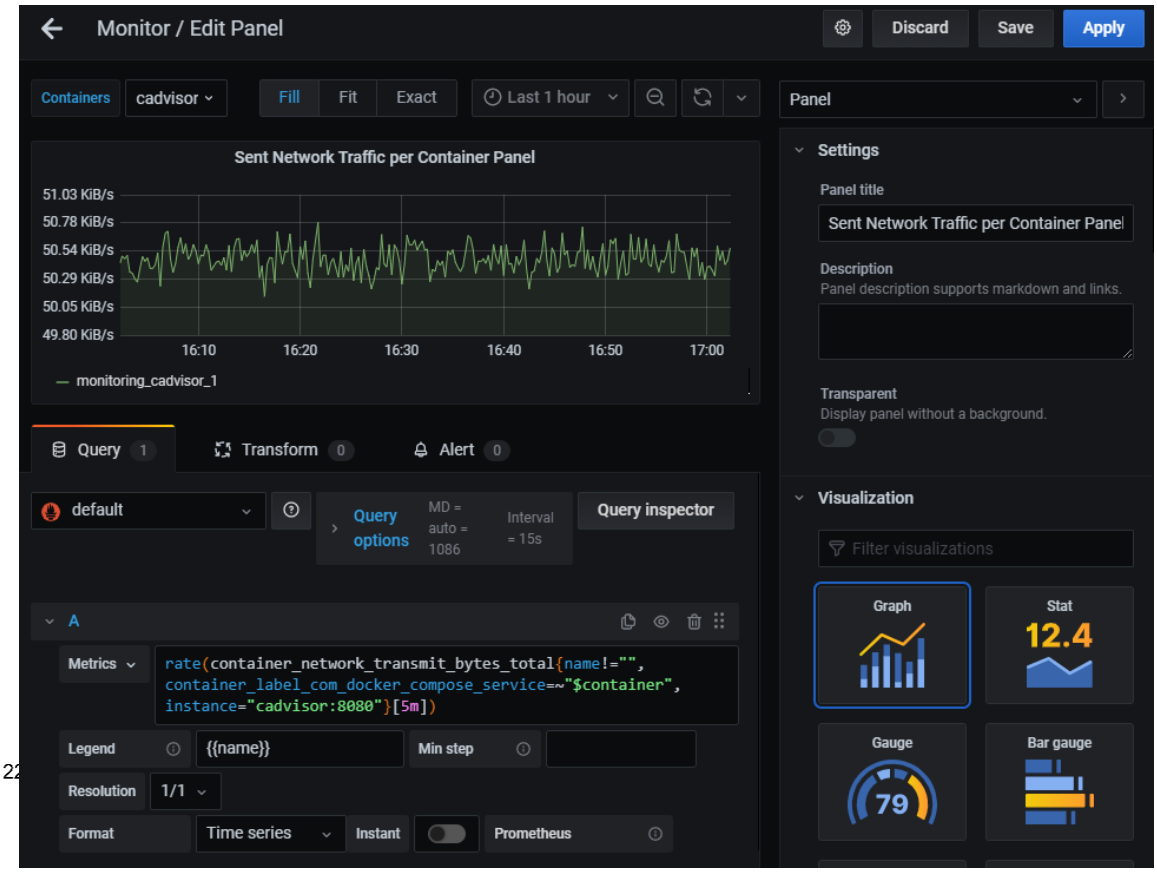


Figure 22

Configured Received Network Traffic per Container panel.

By looking at the Figure 22 it can be noted that the panel is similar to the “Network traffic” panel. The differences are in the queries, where this panel does not show the incoming, and instead of combining all the graphs in the one using the sum() operator the graphs are kept separated. The container variable is also used here to choose which graphs are seen.

6.3.3 Sent Network Traffic, CPU Usage and Memory Usage per Container Panels

For the “Sent Network Traffic per Container” the following query was used:

```
sum(rate(container_network_transmit_bytes_total{name!="", container_la-
bel_com_docker_compose_service=~"$container", instance="cadvisor:8080"}[5m]))
by (name)
```

By comparing the “Received Network Traffic per Container” panel query to the “Sent Network Traffic per Container” query one can note that the only difference with queries is the metric name.

“CPU Usage Per Container” panel units was set to “Percent(0-100)” and the following query was used for the container CPU usage metric:

```
sum(rate(container_cpu_usage_seconds_total{name=~".+", container_la-
bel_com_docker_compose_service=~"$container", instance="cadvisor:8080"}[5m]))
by (name) * 100
```

For the ”Memory Usage per Container” panel, the following query:

```
container_memory_usage_bytes{name!="", container_label_com_docker_compose_ser-
vice=~"$container", instance="cadvisor:8080"}
```

was used. The query returns memory usage in bytes. Thefore the unit of was set to bytes.

6.4 Configuring Alerts

Alerts are an essential part of any monitoring stack. Without alerts, one has to constantly check if there is an issue. With alerts, the ones who maintain the environment will get a

notification if there is an issue. Configuring alerts requires two things: configuring the notification channels and the alert rules.

6.4.1 Notification Channel

The notification channel tells Grafana where to send an alert notification. The configuration starts by creating a new notification channel on the notification channel page.

Grafana supports several different platforms, for example Emails, Google hangouts, Microsoft Teams and Discord.

Figure 23 shows an example configuration.

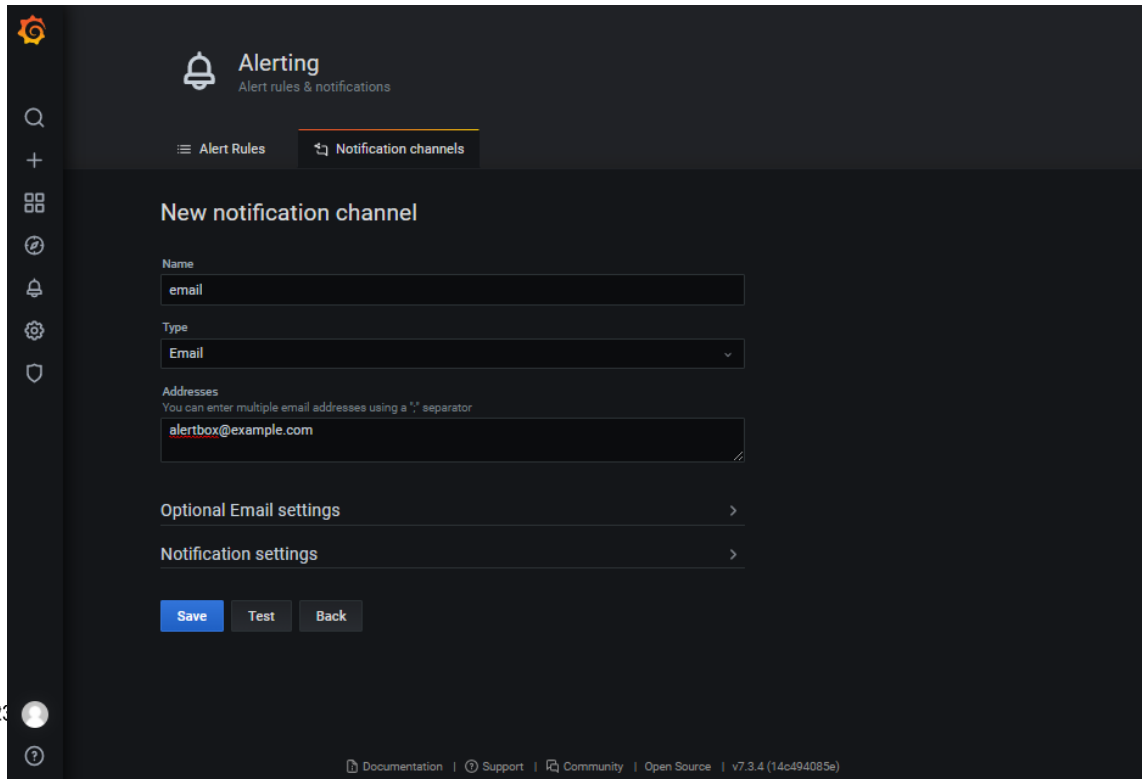


Figure 23

Notification channel configuration for emails.

The notification was wanted to go to an email inbox. Therefore the type was set to an email and an email address was given Alert Rules

Once the notification channel was configured the alerts were configured.

In the dashboard, a panel called “Container down alert” was created. This panel was used to configure an alert rule that alerts if any container goes down. The query

```
count(rate(container_last_seen{id=~"/system.slice/docker-.+",instance="cadvisor:8080"}[5m]))
```

returns the number of containers currently running. On the alert tab, a new alert rule was created. Figure 24 shows the default alert rule.

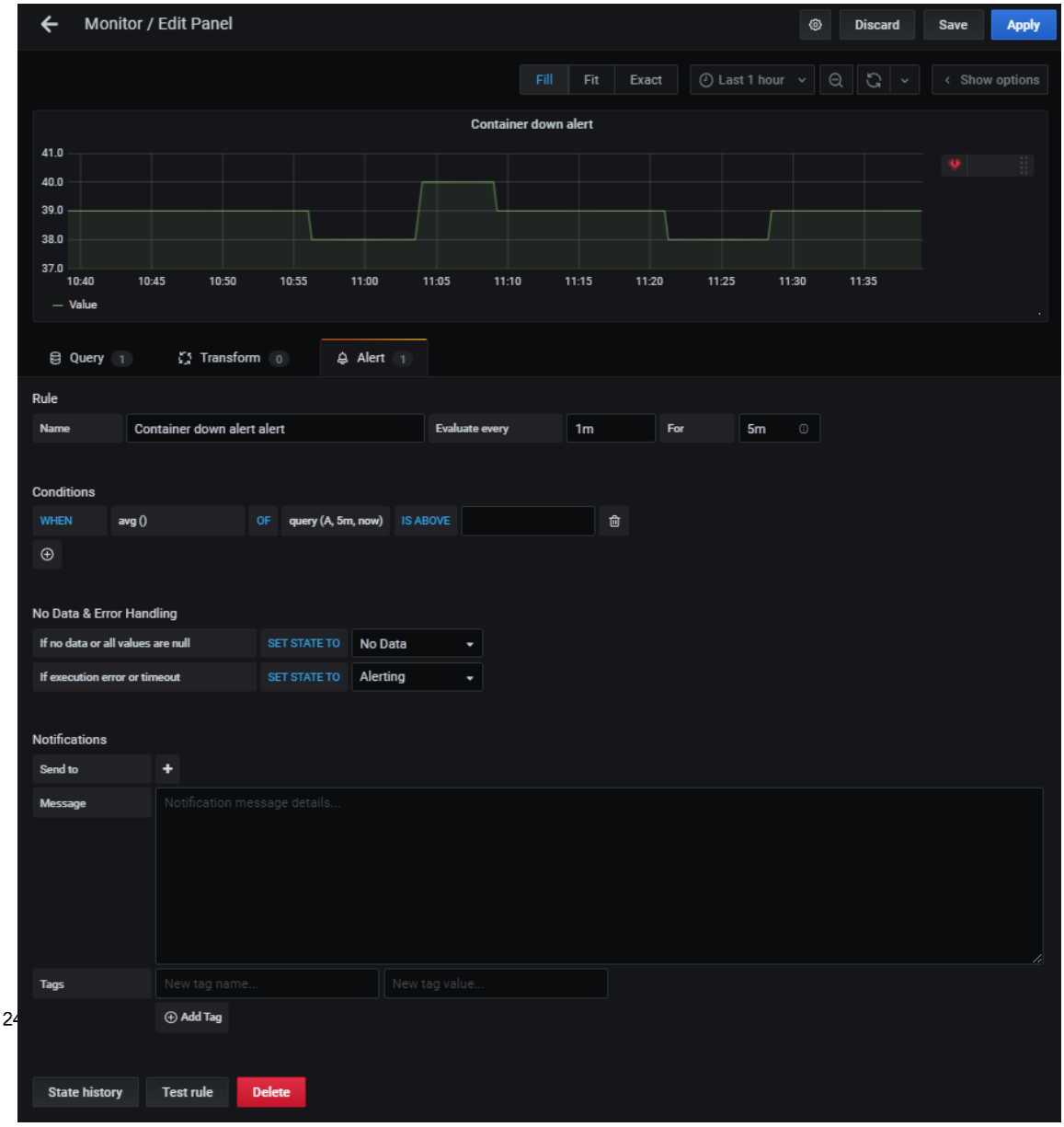


Figure 24. Alert rule configuration view.

The “Name” field is the alert name. The alert was given the name “Container alert.” “Evaluate for” fields tell Grafana how often it should evaluate the alert condition. The default one-minute was used for this alert. The “For” field tells Grafana how long the alert condition should have been met to cause the alert. When Grafana evaluates the alert condition to be true, the alert goes to a pending state, and if the pending state lasts the time set in the “For” field, the alert goes to an alert state. This feature is handy because when Grafana evaluates the alert condition to be true momentarily, it should not cause an alert if the system can recover on its own. For example, when one is changing the of the application configuration, the application has to restart, which causes the container to go down for less than one minute. The inbox should not be filled with useless alerts. Therefore, the “For” field is used to give the environment time to fix on its own. The default five minutes was used for this alert.

The “Conditions” is where the alert rules are set. The first field is the aggregation function field. The query evaluated for the alert rule may have many different series. Therefore, they are combined into one single value to compare them to the threshold value. For this alert rule, the median() function was used. After the aggregation function, there is the “query”. It takes three parameters. The first one is the query used.

Grafana indexes the queries in the panel by using alphabets. The first one is A, and the second one is B, and so on. There is only one query called A in this panel. The last two define the time range.

The default is 5m, and now, Grafana evaluates the query value from five minutes ago to the current time. The “5m” was changes to “1m”. After the query, there is the threshold and the threshold value. By clicking the “IS ABOVE,” one can see all the threshold availably. “IS BELOW” was the one used want. The threshold value depends on how many containers are running when everything is working correctly in the environment. For this environment, it was 39. Next, the alert rule was tested by clicking the “Test rule” button, which gave a successful result.

The “No data & Error Handling” section is used to tell what to do if there is no data or there is an error. For this alert, Grafana is set to start alerting.

The last part of the alert configuration is the “Notification” section. Here one tells Grafana which notification channel to send the notification and the message body of the notification. For the notification channel, the email channel created before was used.

The alert configuration looks as shown in Figure 25.

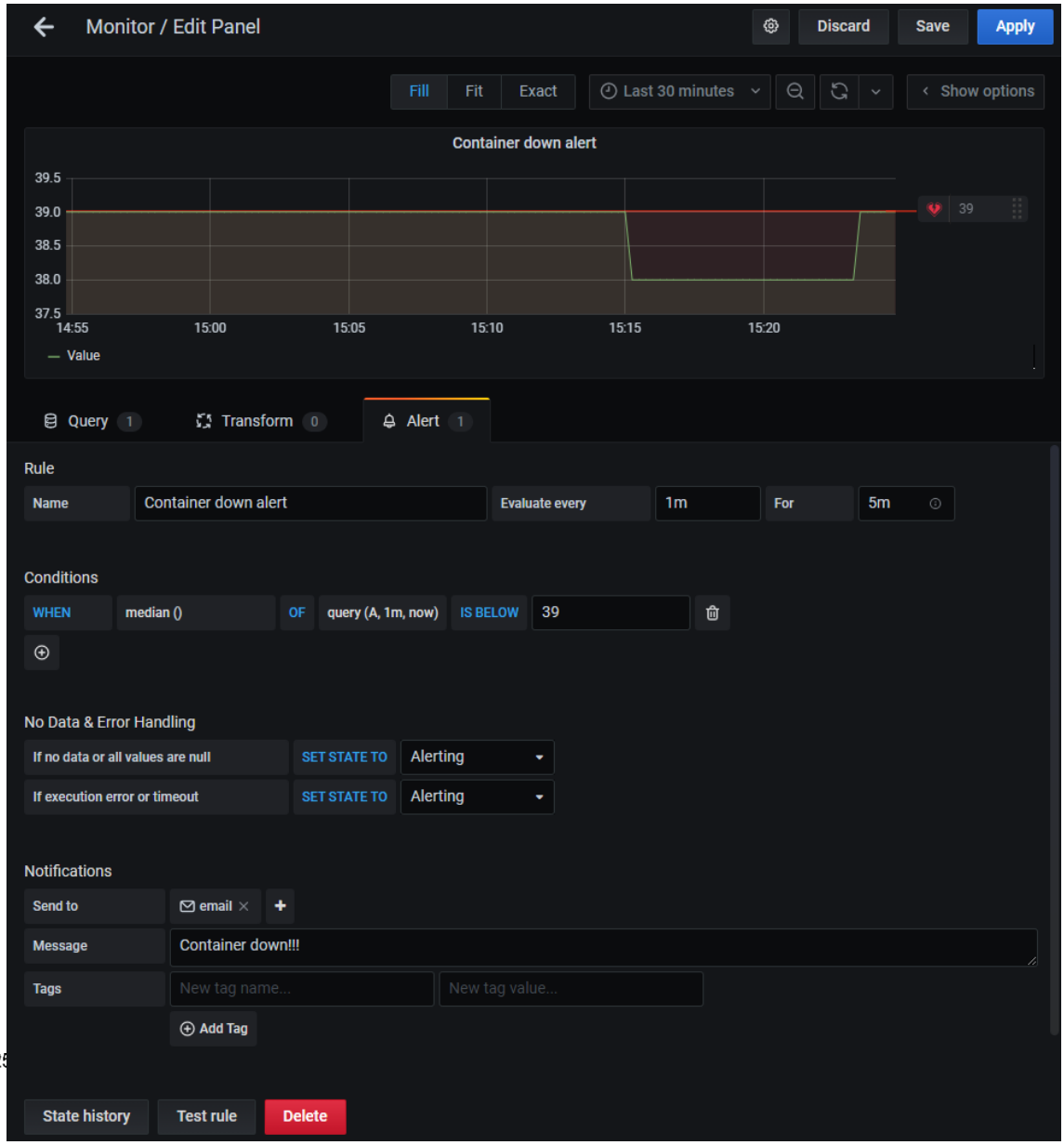


Figure 25

Configured alert rule.

Setting up the alert rules was easy. The hard part is to figure out what should cause an alert and what should the threshold values be, which is very dependent on the environment and the application running on the environment. The more one works with the environment, the more one knows what should cause an alert.

7 Analyzing Data

The metric stack collects the metric data over time, and the end-user can analyze it. Figure 26 shows the overview of the host panels.

Figure 26

General overview of the host panels.

At the top, there is the time range control. In Figure 26, the time scale is set to show the last 6 hours. The green heart next to `Container down alert` tells there is an alert configured in that panel and the color green tells the alerts is not firing. If the color is yellow, the alert is pending and red means it is alerting. The `Containers` is the variable used to control which containers are shown in the per container panels. The variable is set to "All" meaning the metrics from all the containers in the per container panels are shown (Figure 27).

Figure 27. General overview of the per container panels.

One should take note of the number of graphs there are in a single panel. In Figure 27, the metric data from every single container running is shown, which is great when one wants to see the big picture but not when one wants to focus on a single container. To see metric data from a single container or just a few containers, the `Container` variable

is used. For example, by selecting `prometheus`, `cadvisor`, `grafana`, `redis` and `node-exporter` from the drop-down menu, one can see the metric data from the monitoring stack (Figure 28).

Figure 28

Monitoring stack metrics.

By analyzing the monitoring stack metrics, it can be noted that cAdvisor and Node-exporter constantly send data, and Prometheus receives constantly data. By knowing the monitoring stack architecture, one knows Prometheus collects data from Node-exporter and cAdvisor. The monitoring stack shows this in action through the dashboard. When Grafana querying Prometheus can also be seen in action. When there is a spike in Prometheus CPU usage, there is also a spike in the amount of data Prometheus

sends. At the same time, there is a spike in Grafanas' Received and Sent network traffic graphs. These spikes look very similar to each other, indicating a connection. Of course since one knows monitoring stack architecture, one knows that Grafana queries Prometheus, and therefore what is seeing is a PromQL query is send from Grafana to Prometheus who executes it and returns the results.

This chapter showed how one could analyze the collected data. Showing how to analyze the data can be used to monitor the IT infrastructure

8 Improvements and Future Steps

By analyzing the data, it was possible to identify an issue with the memory usage in containers. While solving this issue, it was realized a need to monitor the swap usage of the host and containers. The stack already collects the swap usage metric, so the only thing needed was to configure the swap usage panel and swap usage per container panel.

The need to add the swap panel highlights that the Monitoring stack is not complete and the more one analyzes the data, the more one knows what panels are needed and if there is a need to expand the metrics the stack collects.

For possible improvements, more exporters could be added to get more metrics. The current stacks get the kernel, hardware, and container metrics. Nginx exporter could be added for Nginx metrics and a JVM exporter for JVM metrics. The stack has an issue where one can see what is happening in the server and containers but cannot see why it is happening. In other words, one does not see the events. To solve this issue, the stack could be given o a feature to collect and analyze logs. It could be by using Grafana Loki, which is a multi-tenant log aggregation system inspired by Prometheus [18].

9 Conclusion

The thesis shows how to set up a monitoring stack. A simple dashboard was created, which shows the host and container CPU and memory usage and network traffic. In the

improvement section, the need to monitor the swap use was noted. Showing that the dashboard is not complete, and new panels should be added when the need arises. It also showed that the stack already collects more metrics than one was interested in. In the improved section, went through ideas on expanding the stack by adding more exporters and monitoring logs. The stack monitors the container environment on a general level, but there are many improvements to make it more tailored for the environment and application running on the environment.

The purpose of the stack was to help troubleshoot and identify issues in the applications and IT infrastructure. The stack was used to fix an issue with container memory usage. Therefore, it can be concluded that this aim was successfully met.

Monitoring applications and IT infrastructure help to optimize, identify issues, and troubleshoot issues in them. Now, the basic monitor stack being ready the only thing left is to start adding a new panel when needed and expand the stack by adding more exporters to make the optimization, issue identification, and troubleshooting easier.

References

1 What is a Container?. Online. Docker. <https://www.docker.com/resources/what-container>. Accessed 21 April 2021

2 Overview Prometheus. Online. Prometheus. <https://prometheus.io/docs/introduction/overview/>. Accessed 15 April 2021

3 About SoundCloud. Online. SoundCloud. <https://soundcloud.com/pages/contact>. Accessed 15 April 2021

4 Querying Prometheus. Online. Prometheus. <https://prometheus.io/docs/prometheus/latest/querying/basics/>. Accessed 15 April 2021

5 GitHub Grafana. Online. GitHub. <https://github.com/grafana/grafana>. Accessed 15 April 2021

6 Dashboards. Online. Grafana Labs. <https://grafana.com/grafana/dashboards>. Accessed 15 April 2021

7 Plugins. Online. Grafana Labs. <https://grafana.com/grafana/plugins/>. Accessed 15 April 2021

8 cAdvisor. Online. GitHub. <https://github.com/google/cadvisor>. Accessed 15 April 2021

9 Redis. Online. Redis. <https://redis.io/>. Accessed 15 April 2021

10 Running cAdvisor. Online. GitHub <https://github.com/google/cadvisor/blob/master/docs/running.md>. Accessed 15 April 2021

11 depends_on. Online. Docker. <https://docs.docker.com/compose/compose-file/compose-file-v2/#depends_on>. Accessed 15 April 2021

12 Configuration. Online. Prometheus. <https://prometheus.io/docs/introduction/first_steps/>. Accessed 20 April 2021

13 Job and Instances. Online. Prometheus. <https://prometheus.io/docs/concepts/jobs_instances/>. Accessed 20 April 2021

14 New Docker Install with persistent storage, Permission problem. Online. Grafana community member. <https://community.grafana.com/t/new-docker-install-with-persistent-storage-permission-problem/10896>. Accessed 15 April 2021

15 grafana/defaultini. Online. GitHub. <https://github.com/grafana/grafana/blob/master/conf/defaults.ini>. Accessed 15 April 2021

16 Operators. Online. Prometheus. <https://prometheus.io/docs/prometheus/latest/querying/operators/>. Accessed 15 April 2021

17 Functions. Online. Prometheus. <https://prometheus.io/docs/prometheus/latest/querying/functions/>. Accessed 15 April 2021

18 Grafana Loki, Online. Grafana Labs. <https://grafana.com/oss/loki/>. Accessed 15 April 2021