Logging and Monitoring System of Docker Containers

Netti Marco
Mat: 892399

July 11, 2024

Contents

1 Introduction

1.1 Project Overview

In modern software development, containerization has become a pivotal technology for deploying and managing applications. Docker, a leading container platform, ensures consistent application operation across various environments by encapsulating applications and their dependencies. However, as applications grow in complexity and scale, effective monitoring and logging become essential to maintain performance and reliability.

This project aims to implement a comprehensive logging and monitoring system for Docker Containers. By leveraging state-of-the-art tools such as Prometheus, Node Exporter, Grafana, Alertmanager, cAdvisor, Loki, and Promtail, we will create a centralized platform to track the performance, health, and logs of Docker containers running on your machine. This setup will enhance visibility into your containerized environments, making it easier to manage and troubleshoot issues. The OS used for writing and building this project is Ubuntu 22.04 LTS.

For the complete code and configuration files, visit my Github repository.

1.2 Goals & Objectives

The primary objectives of this project are:

1. **Host metrics collection**: integrate Node Exporter to collect and monitor metrics from the host system where Docker containers are running;
2. **Container resource monitoring**: use cAdvisor to monitor resource usage and performance characteristics of running containers;
3. **Monitoring resources and services**: use Prometheus to gather metrics related to resource utilization (CPU, memory, network) and service health from Docker containers;
4. **Visualizing metrics**: employ Grafana to create intuitive and interactive dashboards for visualizing the metrics collected by Prometheus;
5. **Logging and analysis**: implement Loki for aggregating log data from Docker containers, and use Promtail to forward logs to Loki for analysis.

By the end of this project, users will have a fully functional logging and monitoring system tailored for Docker containers, capable of providing comprehensive insights and facilitating efficient management of containerized environments.

2 Installing Prerequisites

Before we can set up our monitoring and logging system, we need to install Docker and Docker Compose on our system. These tools will allow us to create and manage our containerized environment.

What is Docker?

Docker is an open-source platform tha automates the development, scaling, and management of applications. It uses containerization technology to package an application and its dependencies into a standardized unit known as *container*. This ensures that the applications run across different computing environments, from development and testing to production. The key features of Docker are:

- **Portability**: containers can run on any system that supports Docker, ensuring that applications behave consistently regardless of where they are deployed;
- **Isolation**: Docker containers encapsulate all the components an application need to run, including the code, libraries, and system tools, isolating them from the host system and other containers;
- **Efficiency**: containers are lightweight because they share the host system's kernel, allowing for efficient resource utilization and faster startup times compared to traditional virtual machines.

The main components of Docker are:

- **Docker Engine**: the runtime that allows you to build and run containers;

- **Docker Images**: read-only templates used to create containers. Images include the application code and its dependencies. These images can be stored and shared on Docker Hub, a cloud-based registry service that allows developers to distribute and access public or private container images.
- **Docker Containers**: instances of Docker images that run as isolated processes on the host system.

What is Docker Compose?

Docker Compose is a tool for defining and running multi-container Docker applications. This tool allows you to use a YAML file to configure your application's services, networks, and volumes. With a single command, you can create and start all the services from your configuration. The command to start services is:

```
docker compose up
```

To stop and remove all the containers, networks, and volumes created by `docker compose up`, you can use:

```
docker compose down
```

This command stops all running containers and removes them, along with the networks Docker Compose created at startup. If you want to stop the services without removing the containers, you can use:

```
docker compose stop service_name
```

These commands provide a streamlined way to manage the lifecycle of your multi-container applications.

2.1 Install Docker using the apt repository

Here is the installation guide provided by the official Docker website for Ubuntu/Debian systems.

Before installing the Docker Engine, you need to set up the Docker repository. Afterwards you can install and update Docker from repository.

```
# Setting up the Docker repository

# Add Docker's official GPG key
sudo apt-get update
sudo apt-get install ca-certificates curl
sudo install -m 0755 -d /etc/apt/keyrings
sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o
↪  /etc/apt/keyrings/docker.asc
sudo chmod a+r /etc/apt/keyrings/docker.asc

# Add the repository to apt sources
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
  ↪  https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

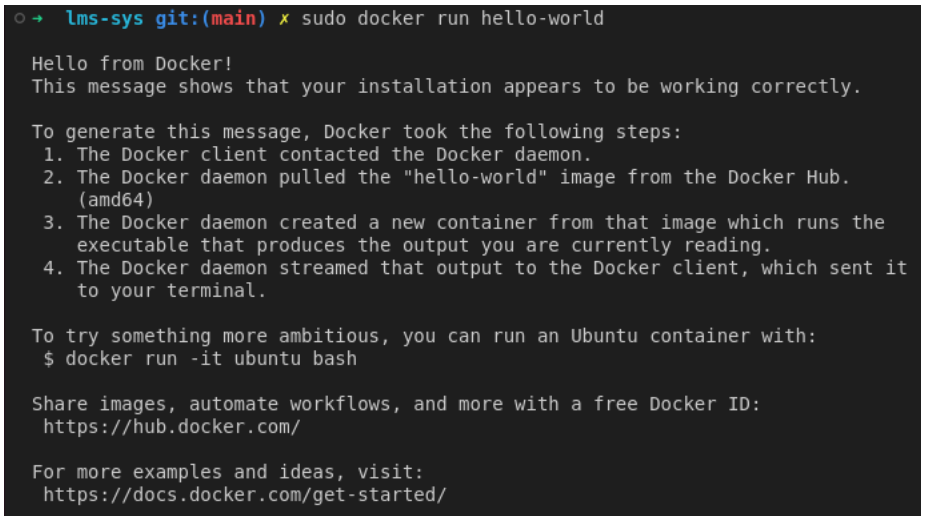
Install the Docker packages.

```
sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin
↪  docker-compose-plugin
```

To confirm the successful installation of Docker Engine, execute the `hello-world` container using the following command:

```
sudo docker run hello-world
```

This command downloads a test image and runs it in a container. When the container runs, it prints a confirmation message and exits.



```
○ ➜  lms-sys git:(main) ✗ sudo docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

You have now successfully installed and started Docker Engine.

2.2 Install Docker Compose using the apt repository

Let's install Docker Compose following the guide provided by the official Docker website.

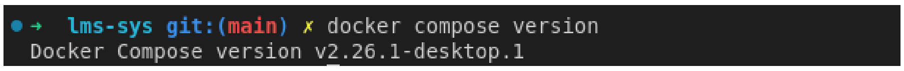
Update the package index, and install the latest version of the tool.

```
sudo apt-get update
sudo apt-get install docker-compose-plugin
```

Verify that Docker Compose is installed correctly by checking the version.

```
docker compose version
```

Expected output: `Docker Compose version v.N.N.N`, where `v.N.N` is placeholder text standing in for the latest version.



```
● ➜  lms-sys git:(main) ✗ docker compose version
 Docker Compose version v2.26.1-desktop.1
```

3 Setting Up Monitoring Services

3.1 Prometheus & Node Exporter

Prometheus is a robust, open-source monitoring and alerting toolkit that has revolutionized the way developers and operations teams approach system observability. Designed with reliability and scalability at its core, Prometheus offers a comprehensive solution for collecting, storing, and analyzing metrics from a wide array of systems and applications. At the heart of Prometheus lies its time series database, which efficiently records real-time metrics using a sophisticated HTTP pull model. This approach allows for highly accurate and timely data collection, ensuring that administrators have access to up-to-the-minute information about their systems' performance. The flexibility of Prometheus is evident in its data model, which employs a multidimensional approach, allowing metrics to be labeled with key-value pairs. This labeling system enables precise filtering and grouping of data, providing unparalleled granularity in metric analysis. One of Prometheus' standout features is its powerful query language, PromQL, which allows users to slice and dice collected metrics with ease, facilitating complex analyses and enabling the creation of detailed visualizations and alerts. The pull-based architecture of Prometheus sets it apart from many traditional monitoring systems, as it actively scrapes metrics from configured targets rather than passively waiting for data to be pushed to it. This approach not only enhances reliability but also provides greater control over the monitoring process. Furthermore, Prometheus excels in dynamic environments thanks to its service discovery capabilities, which allow it to automatically detect and monitor new instances as they come online in cloud or containerized setups. The alerting system in Prometheus is both flexible and powerful, enabling users to define complex alert conditions based on PromQL expressions. These alerts can then be routed through Alertmanager, which handles alert deduplication, grouping, and routing to various notification channels such as email, Slack, or PagerDuty.

3.1.1 Configuring Prometheus & Node Exporter

To begin setting up your monitoring and logging system, we first need to create a project directory. This directory will contain all the configuration file and resources necessary for deploying the services using Docker Compose.

Start by opening a terminal window on your system. Use the following commands to create a new directory for your project and navigate into it:

```
mkdir lms-sys
cd lms-sys
```

This command creates a directory named `lms-sys` and changes your current working directory to it. To ensure you are in the correct directory, you can run the `pwd` command, which should display the path to your newly created `lms-sys` directory. How, you have a dedicated directory for your monitoring and logging project. All subsequent configurations and files will be placed insede this directory.

Now we need to create a `docker-compose.yml` file: a configuration file used by Docker Compose to define and run multi-container Docker applications. It uses YAML syntax to describe services, networks and volumes required for the application.

To create the `docker-compose.yml` file, ensure you are inside the `lms-sys` directory:

```
touch docker-compose.yml
```

Open the file in you preferred text editor (or IDE) and add the following basic structure:

```
version: '3.8'

volumes:
    prometheus-data: {}

services:
    node-exporter:
        image: quay.io/prometheus/node-exporter:latest
        container_name: node-exporter
        restart: unless-stopped
        network_mode: host
        pid: host
        volumes:
            - '/proc:/host/proc'
        command:
            - '--path.rootfs=/host'

    prometheus:
        image: prom/prometheus:latest
        container_name: prometheus
        restart: unless-stopped
        volumes:
            - prometheus-data:/prometheus
        command:
            - '--web.enable-lifecycle'
        ports:
            - '9090:9090'
```

This configuration sets up the foundational structure for your monitoring system:

- **Version**: specifies the version of the Docker Compose file format;
- **Volumes**: defines a volume named `prometheus-data` to persist Prometheus data;
- **Services**:
  - **Node Exporter**: uses the latest `quay.io/prometheus/node-exporter` image. It esposes host metrics by mounting the `/proc` directory and runs with the `-path.rootfs=/host` commad. Node Exporter collects a variety of system-level metrics, including CPU usage, memory usage, disk I/O, network statistics, and filesystem metrics;
  - **Prometheus**: uses the latest `prom/prometheus` image, stores its data in the `prometheus-data` volume, and exposes port 9090.

Please note: Docker Compose defines a default network to facilitate communication between services. This means that Prometheus can scrape metrics from Node Exporter using the service name `node-exporter` as the hostname (or specifing a static ip). However, to archieve this, it is necessary to configure `prometheus.yml`.

Before configure `prometheus.yml`, create a directory for Prometheus configuration:

```
mkdir prometheus
```

Inside the `prometheus` directory, create a fine named `prometheus.yml` with the follwing content:

```
global:
  scrape_interval: 15s

scrape_configs:
  - job_name: 'node'
    static_configs:
      - targets: ['172.17.0.1:9100']
```

This configuration sets Prometheus to scrape metrics from Node Exporter running on the specified IP address and port.

Now, update the `docker-compose.yml` file to include the Prometheus configuration:

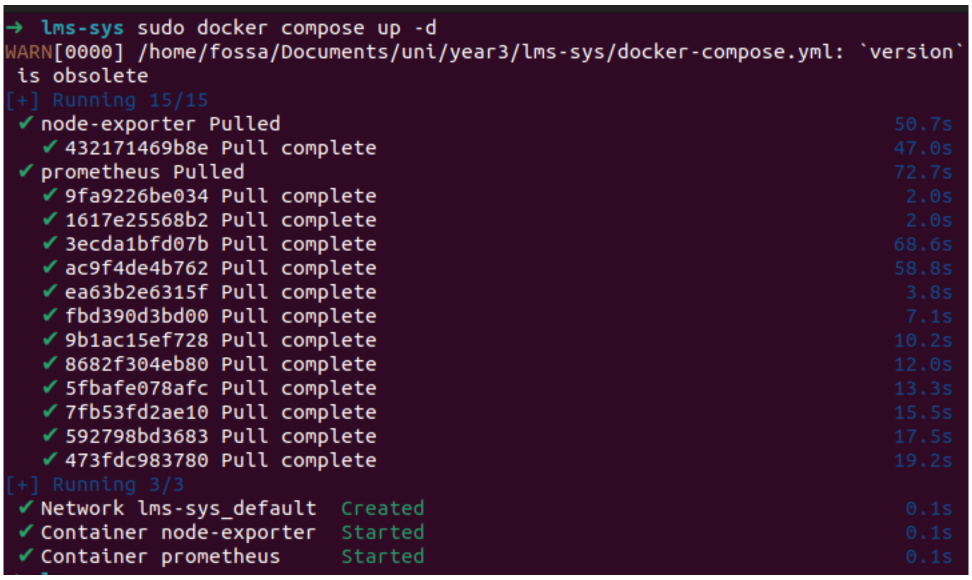
```
version: '3.8'

volumes:
  prometheus-data: {}

services:
  node_exporter:
    image: quay.io/prometheus/node-exporter:latest
    container_name: node_exporter
    restart: unless-stopped
    network_mode: host
    pid: host
    volumes:
      - '/proc:/host/proc'
    command:
      - '--path.rootfs=/host'

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml # mounted prometheus
    ↪ config directory
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--web.enable-lifecycle'
    ports:
      - '9090:9090'
```

The -config.file=/etc/prometheus/prometheus.yml command is used to specifiy the location of the Prometheus configuration file. When Prometheus starts, it uses this argument to find and load the configuration settings defined in the prometheus.yml file, which include global settings, scrape configurations, and any other directives necessary for Prometheus to function correctly. This ensures that Prometheus reads the correct configuration file and applies the specified settings during its operation.

Ensure you are in the project directory, the run the following command to start the services:

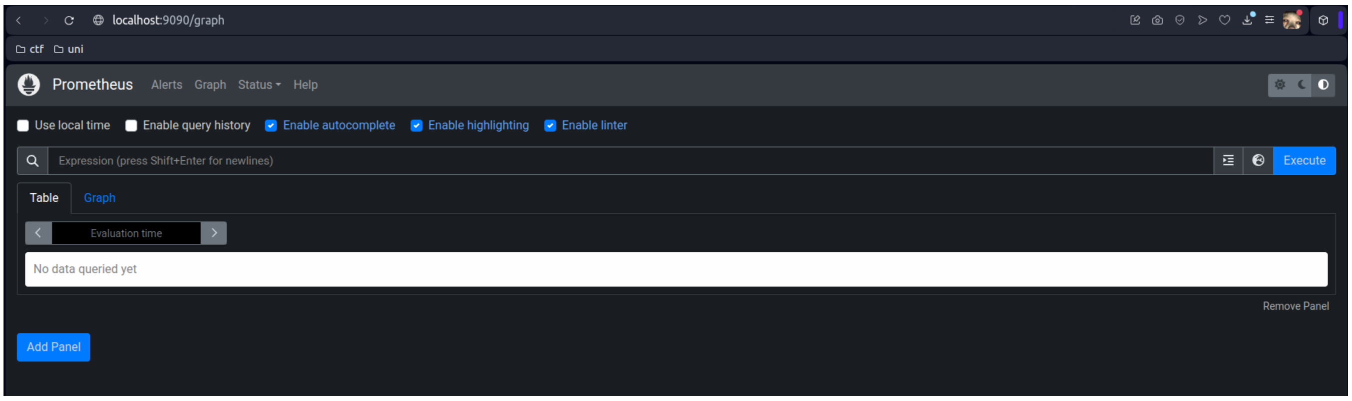
```
sudo docker compose up -d
```

The first time you run this command, it will pull the images specified on the Docker Compose file from Docker Hub:



```
➜  lms-sys sudo docker compose up -d
WARN[0000] /home/fossa/Documents/uni/year3/lms-sys/docker-compose.yml: `version`
 is obsolete
[+] Running 15/15
 ✔ node-exporter Pulled                                                    50.7s
   ✔ 432171469b8e Pull complete                                            47.0s
 ✔ prometheus Pulled                                                       72.7s
   ✔ 9fa9226be034 Pull complete                                             2.0s
   ✔ 1617e25568b2 Pull complete                                             2.0s
   ✔ 3ecda1bfd07b Pull complete                                            68.6s
   ✔ ac9f4de4b762 Pull complete                                            58.8s
   ✔ ea63b2e6315f Pull complete                                             3.8s
   ✔ fbd390d3bd00 Pull complete                                             7.1s
   ✔ 9b1ac15ef728 Pull complete                                            10.2s
   ✔ 8682f304eb80 Pull complete                                            12.0s
   ✔ 5fbafe078afc Pull complete                                            13.3s
   ✔ 7fb53fd2ae10 Pull complete                                            15.5s
   ✔ 592798bd3683 Pull complete                                            17.5s
   ✔ 473fdc983780 Pull complete                                            19.2s
[+] Running 3/3
 ✔ Network lms-sys_default  Created                                         0.1s
 ✔ Container node-exporter  Started                                         0.1s
 ✔ Container prometheus     Started                                         0.1s
```

To verify the setup, open a web browser and navigate to http://localhost:9090 to access to Prometheus web UI. You should see the Prometheus interface, confirming that Prometheus is up and running.

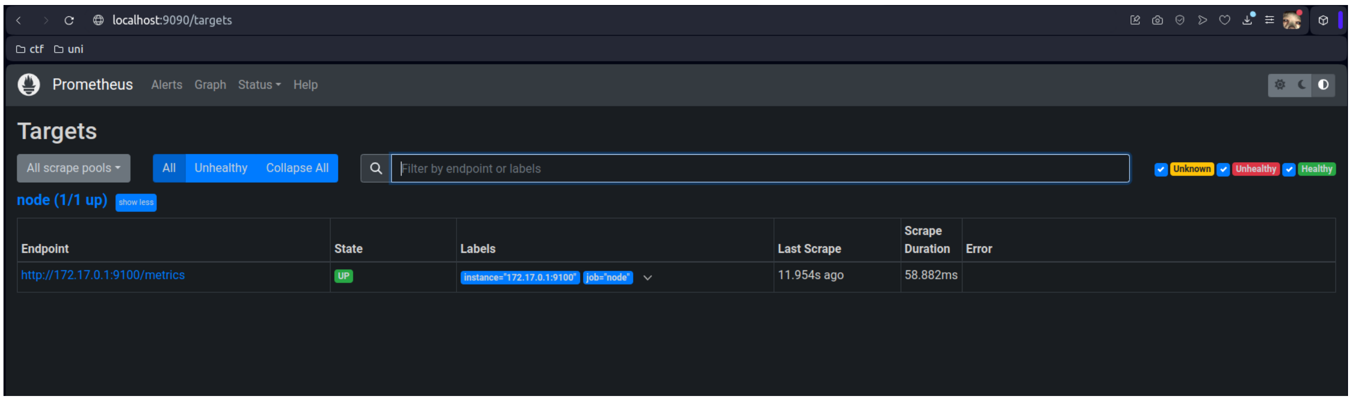


[ ] Use local time [ ] Enable query history [✓] Enable autocomplete [✓] Enable highlighting [✓] Enable linter

Expression (press Shift+Enter for newlines)

No data queried yet

Now, navigate to the "Targets" section under the "Status" menu. Here, you should see that Prometheus is successfully scraping metrics from the Node Exporter service, indicated by the target being listed with an "UP" status.



node (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.1:9100/metrics	UP	instance="172.17.0.1:9100" job="node"	11.954s ago	58.882ms	

To stop the running Containers, type the following command in the terminal:

```
sudo docker compose down
```

This command will stop and remove all the containers defined in your `docker-compose.yml` file, along with any networks created by Docker Compose. This is useful for halting your services when you need to make changes or updates to your setup.

3.2 Grafana

Grafana is an open-source platform widely used for monitoring, visualization, and alerting on time-series data. It allows users to create and share interactive and costumizable dashboards, offering powerful features metrics from various data sources such as Prometheus. The key features of Grafana are:

- **Multi-Source Data Integration**: Grafana supports numerous data sources, enabling users to create unified dashboards that combine metrics from multiple systems;
- **Interactive Dashboards**: users can create dynamic dashboards with a variety of visualization options, suche as graphs, heatmaps and table;
- **User Management**: Grafana includes comprehensive user management features, enabling administrators to control access to dashboards and data sources;
- **Plugin and Extensions**: a rich ecosystem of plugin extensions enhances Grafana's functionalities, providing additional data sources, panel types, and application integrations;

In this project, Grafana will serve as the primary tool for visualizing the metrics and logs collected from the Docker containers.

3.2.1 Configuring Grafana

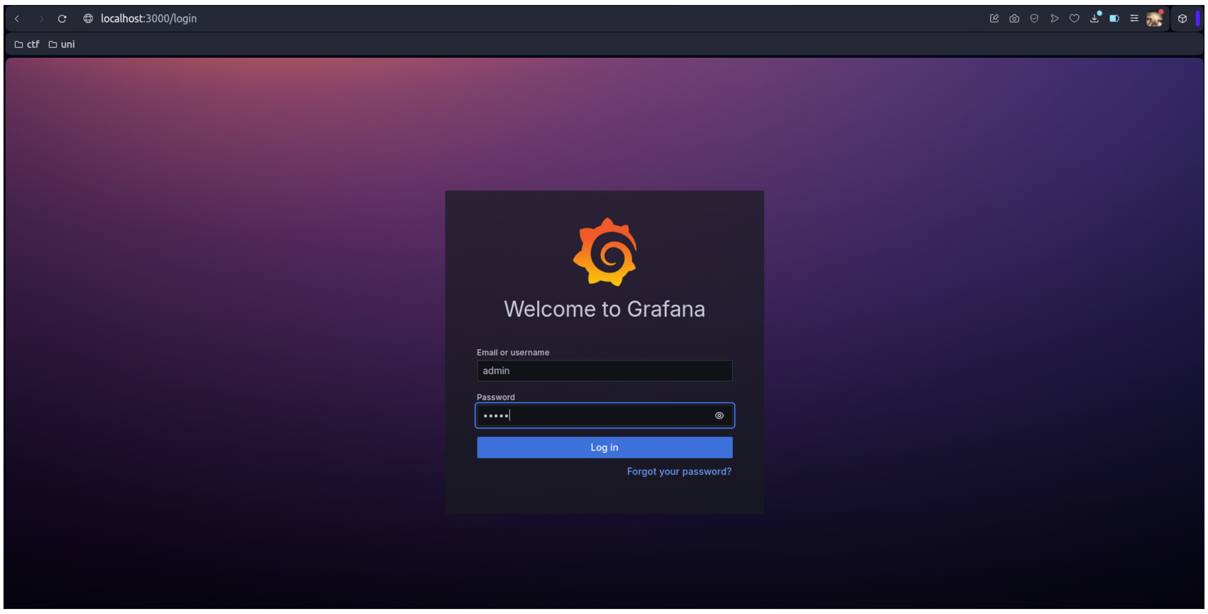
Extend your `docker-compose.yml` file to include the Grafana service. The updated file is:

```yaml
version: '3.8'

volumes:
  prometheus-data: {}
  grafana-data: {}

services:
  node-exporter:
    image: quay.io/prometheus/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    network_mode: host
    pid: host
    volumes:
      - '/proc:/host/proc'
    command:
      - '--path.rootfs=/host'

  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'
      - '--web.enable-lifecycle'
    ports:
      - '9090:9090'

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    restart: unless-stopped
    volumes:
      - grafana-data:/var/lib/grafana
    ports:
      - '3000:3000'
```

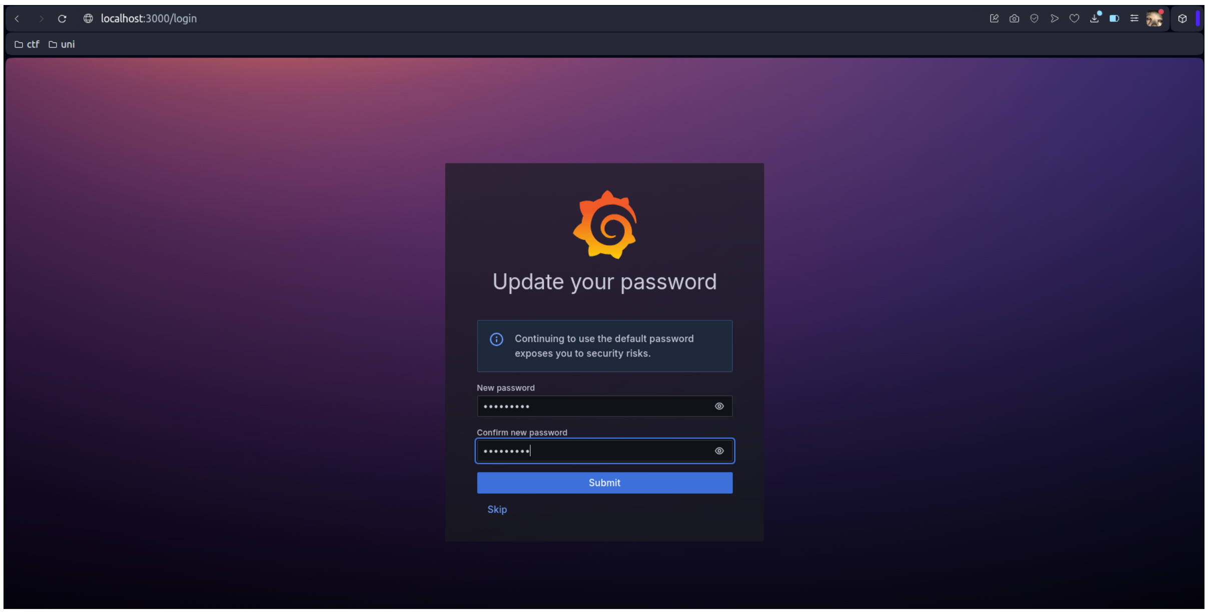
To compose file, ensure you are in the project directory and run:

```
sudo docker compose up -d
```

Open a web browser and navigate to `http://localhost:3000` and log in with the default username and password, which are both `admin` by default.

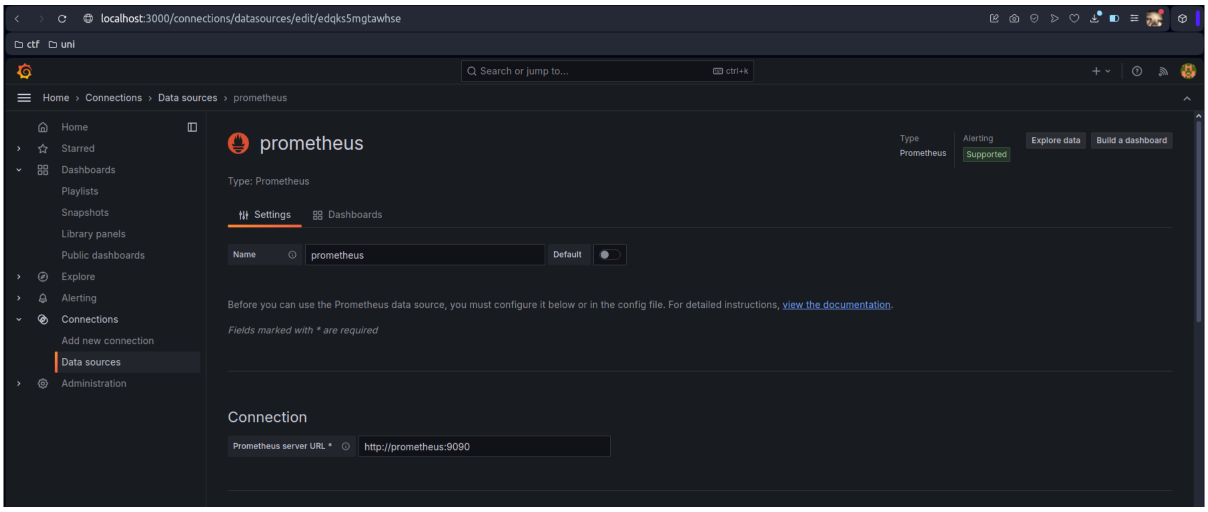


After your first login, you will be asked to change the default password to one that is more robust (but is not mandatory).



To add Prometheus as a data source in Grafana:

- Go to "Connections" > "Data Sources" > "Add data source".
- Select "Prometheus"
- Set the URL to `http://prometheus:9090`
- Click "Save & Test".



localhost:3000/connections/datasources/edit/edqks5mgtawhse

Home › Connections › Data sources › prometheus

prometheus

Type: Prometheus

Type Prometheus | Alerting Supported | Explore data | Build a dashboard

Settings | Dashboards

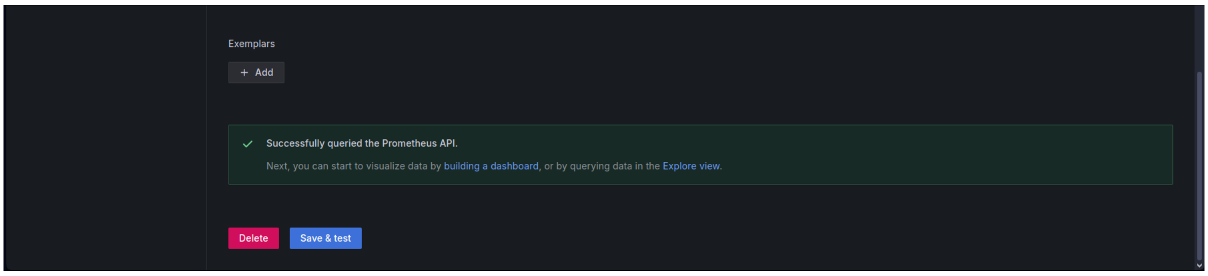
Name: prometheus | Default

Before you can use the Prometheus data source, you must configure it below or in the config file. For detailed instructions, view the documentation.

*Fields marked with * are required*

Connection

Prometheus server URL *: http://prometheus:9090



Exemplars

+ Add

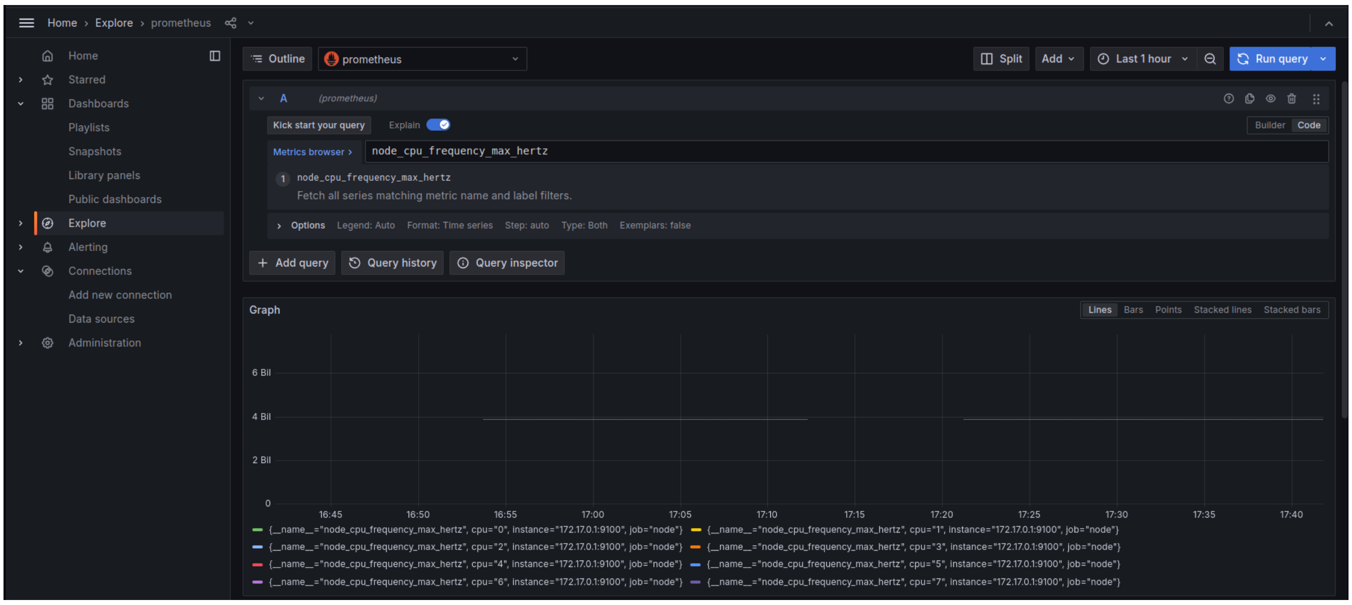
Successfully queried the Prometheus API.

Next, you can start to visualize data by building a dashboard, or by querying data in the Explore view.

Delete | Save & test

To explore and create queries on data scraped by Prometheus data source:

- Go to "Connections" > "Data Sources"
- Press "Explore" on Prometheus data source
- Use the query editor to create and run queries on your metrics (for example, the cpu frequecy of the node);



Home › Explore › prometheus

Outline | prometheus | Split | Add | Last 1 hour | Run query

A (prometheus)

Kick start your query | Explain | Builder | Code

Metrics browser › `node_cpu_frequency_max_hertz`

1 node_cpu_frequency_max_hertz
Fetch all series matching metric name and label filters.

Options Legend: Auto Format: Time series Step: auto Type: Both Exemplars: false

Add query | Query history | Query inspector

Graph

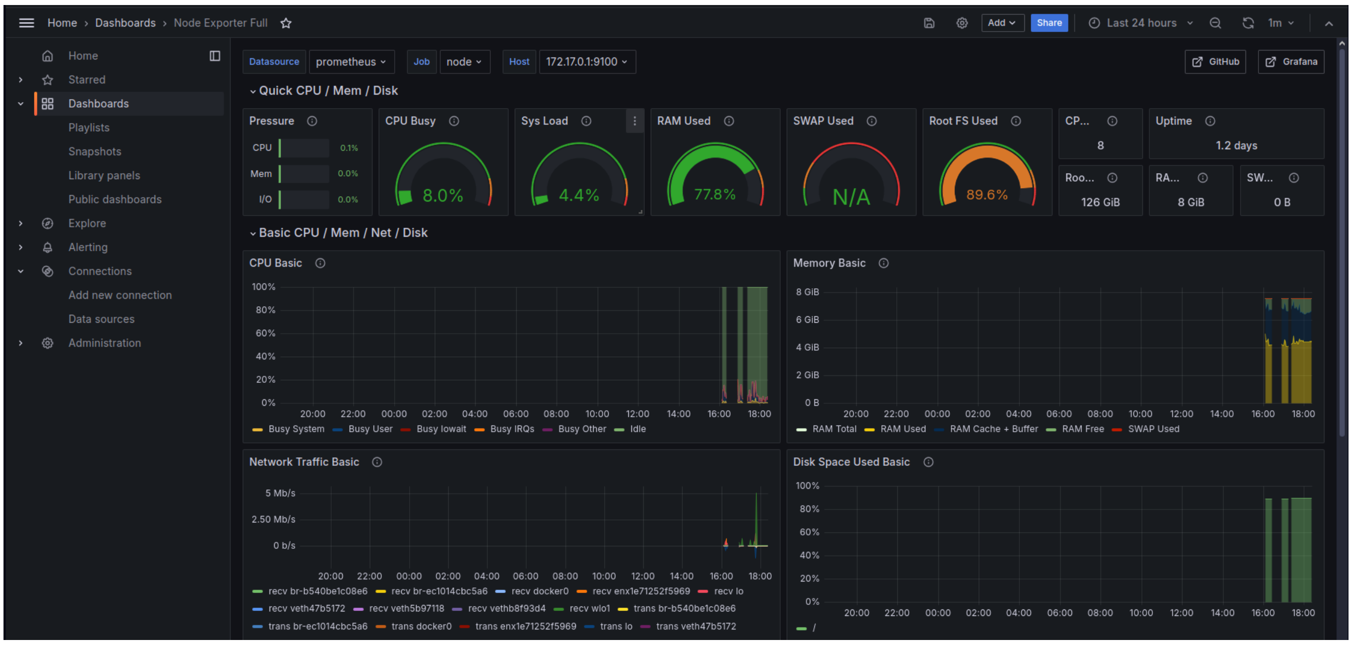
```
{__name__="node_cpu_frequency_max_hertz", cpu="0", instance="172.17.0.1:9100", job="node"}
{__name__="node_cpu_frequency_max_hertz", cpu="1", instance="172.17.0.1:9100", job="node"}
{__name__="node_cpu_frequency_max_hertz", cpu="2", instance="172.17.0.1:9100", job="node"}
{__name__="node_cpu_frequency_max_hertz", cpu="3", instance="172.17.0.1:9100", job="node"}
{__name__="node_cpu_frequency_max_hertz", cpu="4", instance="172.17.0.1:9100", job="node"}
{__name__="node_cpu_frequency_max_hertz", cpu="5", instance="172.17.0.1:9100", job="node"}
{__name__="node_cpu_frequency_max_hertz", cpu="6", instance="172.17.0.1:9100", job="node"}
{__name__="node_cpu_frequency_max_hertz", cpu="7", instance="172.17.0.1:9100", job="node"}
```

In Grafana, you can create costum dashboards or add pre-built dashboards created by other users. To add pre-built dashboards from Grafana Dashboards:

- Find a dashboard you want to use and copy its dashboard ID or download the JSON file.
- In Grafana, go to "Dashboards" > "New" > "New dashboard" > "Import"
- Paste the dashboard ID or upload the JSON file, and click "Load".

- Configure the data source (if required), and click "Import".

For example, I added the "Node Exporter Full" dashboard, which uses the metrics scraped by Prometheus from the Node Exporter service. This dashboard provides good-looking graphs and detailed insights into CPU usage, memory usage, disk I/O, network statistics, and filesystem metrics, helping you monitor the overall health and performance of the host system.



3.3 Alertmanager

Alertmanager is a crucial component of the Prometheus ecosystem designed to handle alerts sent by Prometheus server. It manages alerts by grouping, routing and sending notifications through various methods like email, Slack, and PagerDuty. Alertmanager also supports silencing and inhibition, which help in reducing alert noise by suppressing notifications for specific alerts under certain conditions. This ensures that only meaningful alerts reach the relevant personnel, thereby improvising the efficiency and effectiveness of monitoring and alerting systems.

In this guide, we will configure Alertmanager to send notifications via email.

3.3.1 Configuring Alertmanager

First, create the necessary directory for the Alertmanager configuration file (inside the project directory):

```
mkdir alertmanager
```

Now, create `alertmanager.yml` file inside the directory created with the following content:

```
global:
  resolve_timeout: 1m

route:
  receiver: 'mail_alert'
  group_by: [ alertname ]
  repeat_interval: 1m

receivers:
- name: 'mail_alert'
  email_configs:
  - smarthost: 'smtp.gmail.com:587'
```

```
      auth_username: 'monitorloga@gmail.com'
      auth_password: "<app Google key>" # secret
      from: 'monitorloga@gmail.com'
      to: 'monitorloga@gmail.com'
      headers:
        subject: 'Prometheus Mail Alert'
```

This configuration sets up Alertmanager to send alerts via email with the following settings:

- global: sets a global resolve_timeout of 1 minute, which is the time Alertmanager waits before considering an alert resolved;
- route: defines the primary route for alerts, sending them to the mail_alert receiver. Alerts are grouped by the alertname label, and notifications are reapeted every 1 minute;
- receivers: specifies the mail_alert receiver, which sends email notifications using the Gmail SMTP server. The email is sent form and to monitorloga@gmail.com, with a custom subject header "Prometheus Mail Alert".

Next, create the rules.yml file inside the prometheus directory (created in a previous section) with the following content:

```
groups:
  - name: alert_rules
    rules:
      - alert: InstanceDown
        expr: up == 0
        for: 1m
        labels:
          serverity: critical
        annotations:
          summary: "Instance {{ $labels.instance }} down"
          description: "{{ $labels.instance }} or job {{ $labels.job }} has been down
↪ for more than one minute."
```

This rules.yml file defines alerting rules for Prometheus. The example rule here raises an alert named "InstanceDown" if any instance is down (i.e., up == 0) for more than 1 minute. The alert is labeled as "critical" and includes annotations to provide a summary and description of the alert. Users can add more rules to these files as needed. Other scenarios could include:

- **High CPU Usage**: trigger an alert if CPU usage exceeds a certain threshold for a specified duration.
- **memory Usage**: alert if memory consumption goes beyond a set limit;
- **Disk Space**: raise an alert if disk space usage surpasses a defined percentage.

After creating rules.yml, update the prometheus.yml configuration file to include Alertmanager settings:

```
global:
  scrape_interval: 15s
  evaluation_interval: 15s # how often the Prometheus evaluates rules

rule_files: # specifies the path to the 'rules.yml' file that contains alerting rules
  - 'rules.yml'

alerting: # configures Prometheus to send alerts to the Alert-manager service
  alertmanagers:
    - static_configs:
        - targets: ['172.17.0.1:9093']

scrape_configs:
  - job_name: node
```

```
    static_configs:
    - targets: ['172.17.0.1:9100']
```

This configuration tells Prometheus to send alerts to the Alertmanager service running on port 9093.
Next, extend your `docker-compose.yml` file to include Alertmanager service. The updated file should look like this:

```
version: '3.8'

volumes:
  prometheus-data: {}
  alertmanager-data: {}
  grafana-data: {}

services:
  node_exporter:
    image: quay.io/prometheus/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    network_mode: host
    pid: host
    volumes:
      - '/proc:/host/proc'
    command:
      - '--path.rootfs=/host'

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - ./prometheus/rules.yml:/etc/prometheus/rules.yml
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--web.enable-lifecycle'
    ports:
      - '9090:9090'

  alertmanager:
    image: quay.io/prometheus/alertmanager:latest
    container_name: alertmanager
    restart: unless-stopped
    volumes:
      - ./alertmanager/alertmanager.yml:/etc/alertmanager/alertmanager.yml
      - alertmanager-data:/alertmanager
    command:
      - '--config.file=/etc/alertmanager/alertmanager.yml'
    ports:
      - '9093:9093'

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    restart: unless-stopped
    volumes:
      - 'grafana-data:/var/lib/grafana'
    ports:
      - '3000:3000'
```

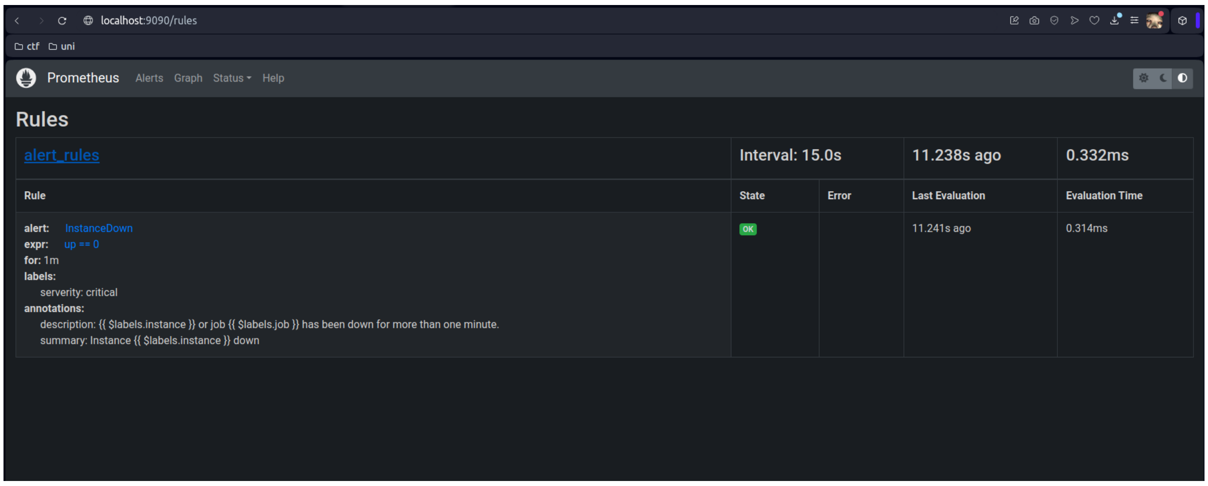
To compose the file, ensure you are in the project directory and run

```
docker compose up -d
```

To apply the changes to Prometheus, reload the configuration:

```
curl -X POST http://localhost:9090/-/reload
```

Now, if you navigate to `http://localhost:9090/rules`, you can see that the rules have been loaded correctly, indicating that the configuration of Alertmanager and Prometheus is correct.



Rules

alert_rules	Interval: 15.0s		11.238s ago	0.332ms
Rule	**State**	**Error**	**Last Evaluation**	**Evaluation Time**
alert: InstanceDown **expr:** up == 0 **for:** 1m **labels:** serverity: critical **annotations:** description: {{ $labels.instance }} or job {{ $labels.job }} has been down for more than one minute. summary: Instance {{ $labels.instance }} down	OK		11.241s ago	0.314ms

3.3.2 Testing Alertmanager

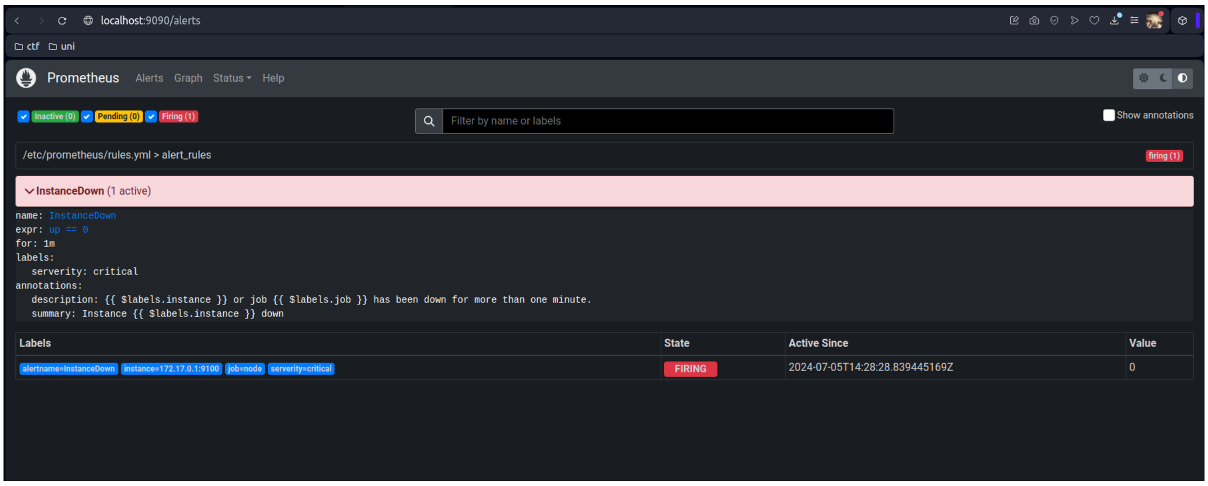
To test if the service works correctly, stop the Node Exporter service and observe the Prometheus web interface at `http://localhost:9090`:

1. Stop the Node Exporter service (it can take some time):

```
sudo docker compose stop node-exporter
```

2. Open the Prometheus web interface and navigate to the "Alerts" section at `http://localhost:9090/alerts`

3. You should see an alert for "InstanceDown" indicating that the Node Exporter is down.



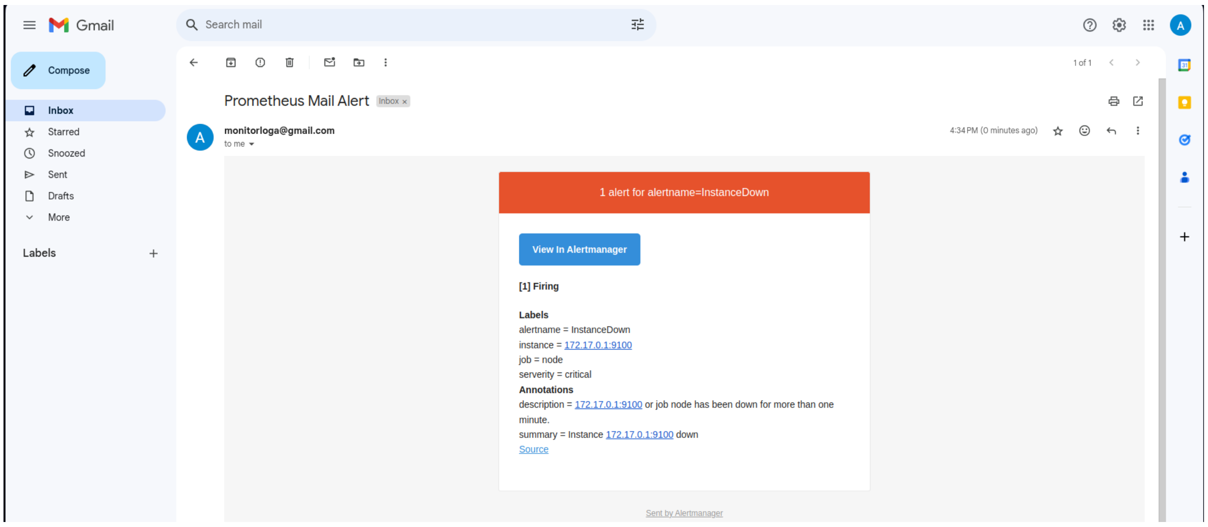
/etc/prometheus/rules.yml > alert_rules firing (1)

InstanceDown (1 active)

```
name: InstanceDown
expr: up == 0
for: 1m
labels:
  serverity: critical
annotations:
  description: {{ $labels.instance }} or job {{ $labels.job }} has been down for more than one minute.
  summary: Instance {{ $labels.instance }} down
```

Labels	State	Active Since	Value
alertname=InstanceDown instance=172.17.0.1:9100 job=node serverity=critical	FIRING	2024-07-05T14:28:28.839445169Z	0

4. You should also receive an email notification if the Alertmanager is correctly configured.



Prometheus Mail Alert Inbox

monitorloga@gmail.com
to me

4:34 PM (0 minutes ago)

1 alert for alertname=InstanceDown

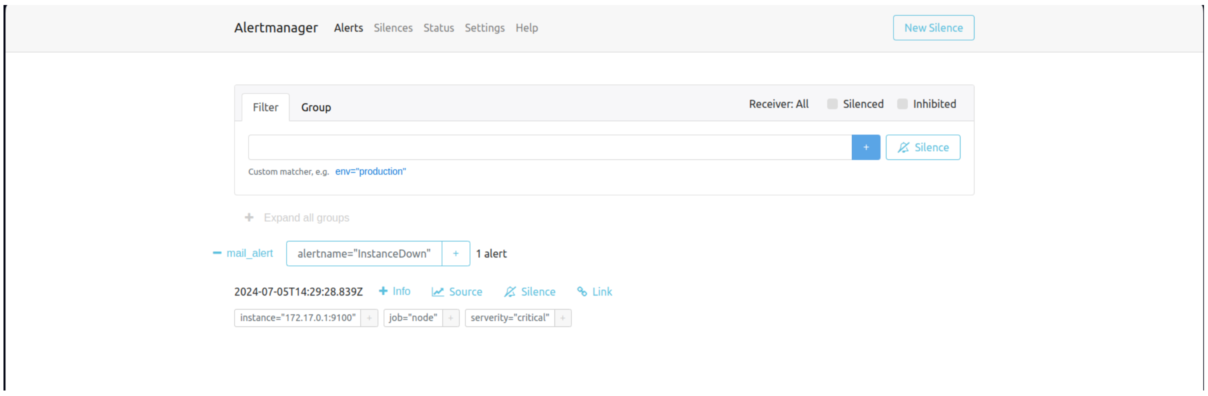
View In Alertmanager

[1] Firing

Labels
alertname = InstanceDown
instance = 172.17.0.1:9100
job = node
serverity = critical
Annotations
description = 172.17.0.1:9100 or job node has been down for more than one minute.
summary = Instance 172.17.0.1:9100 down
Source

Sent by Alertmanager

5. Additionally, you can verify the alert in the Alertmanager web interface at `http://locahost:9093`. The UI Will show the alert as beign active.



mail_alert alertname="InstanceDown" 1 alert

2024-07-05T14:29:28.839Z

instance="172.17.0.1:9100" job="node" serverity="critical"

3.4 cAdvisor

cAdvisor (Container Advisor) is an open-source container monitoring tool developed by Google. It provides detailed insights into the resource usage and performance characteristics of running containers. cAdvisor collects, aggregates, processes, and exports information about running containers, which includes CPU, memory, network, and filesystem usage. This data can be used to monitor and troubleshoot the performance of your containerized applications.

cAdvisor is lightweight and runs as a daemon. It is designed to support containers running on various platforms, including Docker. By integrating cAdvisor with Prometheus and Grafana, you can visualize the collected metrics and gain comprehensive insights into your containerized environment's health and performance.

3.4.1 Configuring cAdvisor

First, extend your `docker-compose.yml` file to include the cAdvisor service. The updated file should look like this:

```yaml
version: '3.8'

volumes:
  prometheus-data: {}
  alertmanager-data: {}
  grafana-data: {}

services:
  node_exporter:
    image: quay.io/prometheus/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    network_mode: host
    pid: host
    volumes:
      - '/proc:/host/proc'
    command:
      - '--path.rootfs=/host'

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - ./prometheus/rules.yml:/etc/prometheus/rules.yml
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--web.enable-lifecycle'
    ports:
      - '9090:9090'

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    restart: unless-stopped
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    ports:
      - '8080:8080'

  alertmanager:
    image: quay.io/prometheus/alertmanager:latest
    container_name: alertmanager
    restart: unless-stopped
    volumes:
      - ./alertmanager/alertmanager.yml:/etc/alertmanager/alertmanager.yml
      - alertmanager-data:/alertmanager
    command:
      - '--config.file=/etc/alertmanager/alertmanager.yml'
    ports:
      - '9093:9093'

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    restart: unless-stopped
    volumes:
      - ./grafana/datasources.yml:/etc/grafana/provisioning/datasources/datasources.yml
      - 'grafana-data:/var/lib/grafana'
    ports:
      - '3000:3000'
```

The cAdvisor service is responsible for collecting metrics from all running containers. It uses the following volumes to access the necessary information:

- `/:/rootfs:ro`: mounts the root filesystem in read-only mode, allowing cAdvisor to collect metrics from the host.
- `/var/run:/var/run:rw`: mounts the `/var/run` directory with read-write permissions, providing access to container runtime information.
- `/sys:/sys:ro`: mounts the `/sys` directory in read-only mode, enabling cAdvisor to gather system-level metrics.
- `/var/lib/docker/:/var/lib/docker:ro`: mounts the Docker data directory in read-only mode, giving cAdvisor access to container-specific data.

To monitor the metrics collected by cAdvisor with Prometheus, update the `prometheus.yml` configuration file to scrape metrics from cAdvisor:

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

rule_files:
  - 'rules.yml'

alerting:
  alertmanagers:
    - static_configs:
        - targets: ['172.17.0.1:9093']

scrape_configs:
  - job_name: node
    static_configs:
    - targets: ['172.17.0.1:9100']

  - job_name: cadvisor
    static_configs:
    - targets: ['172.17.0.1:8080']
```

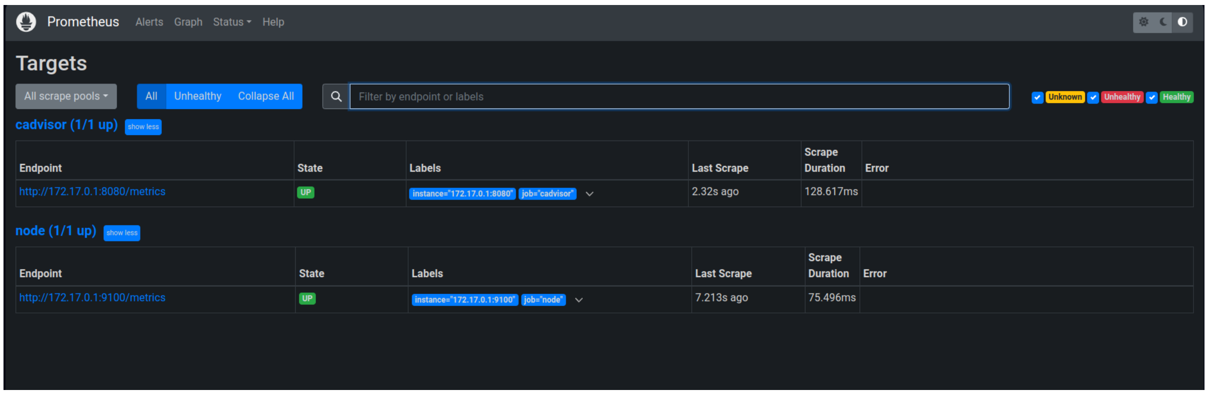
This configuration adds a new scrape job for cAdvisor, allowing Prometheus to collect metrics from the cAdvisor endpoint running on port 8080. To compose the file, ensure you are in the project directory and run

```
docker compose up -d
```

To apply the changes to Prometheus, reload the configuration:

```
curl -X POST http://localhost:9090/-/reload
```

Now you have two Prometheus targets:



Targets

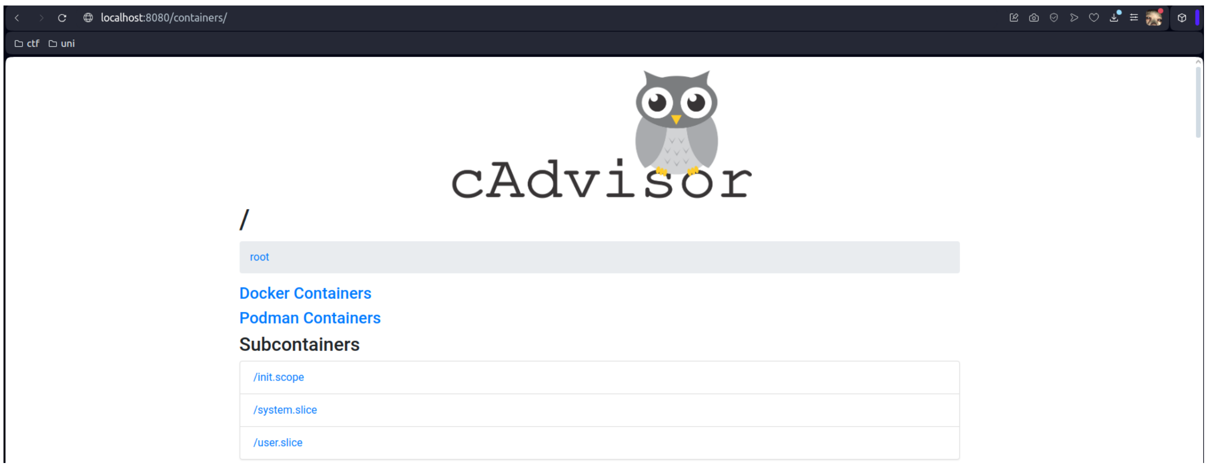
cadvisor (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.1:8080/metrics	UP	instance="172.17.0.1:8080" job="cadvisor"	2.32s ago	128.617ms	

node (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://172.17.0.1:9100/metrics	UP	instance="172.17.0.1:9100" job="node"	7.213s ago	75.496ms	

If you navigate to `http://localhost:8080`, you can see the cAdvisor UI, indicating that cAdvisor is running and collecting metrics.



/

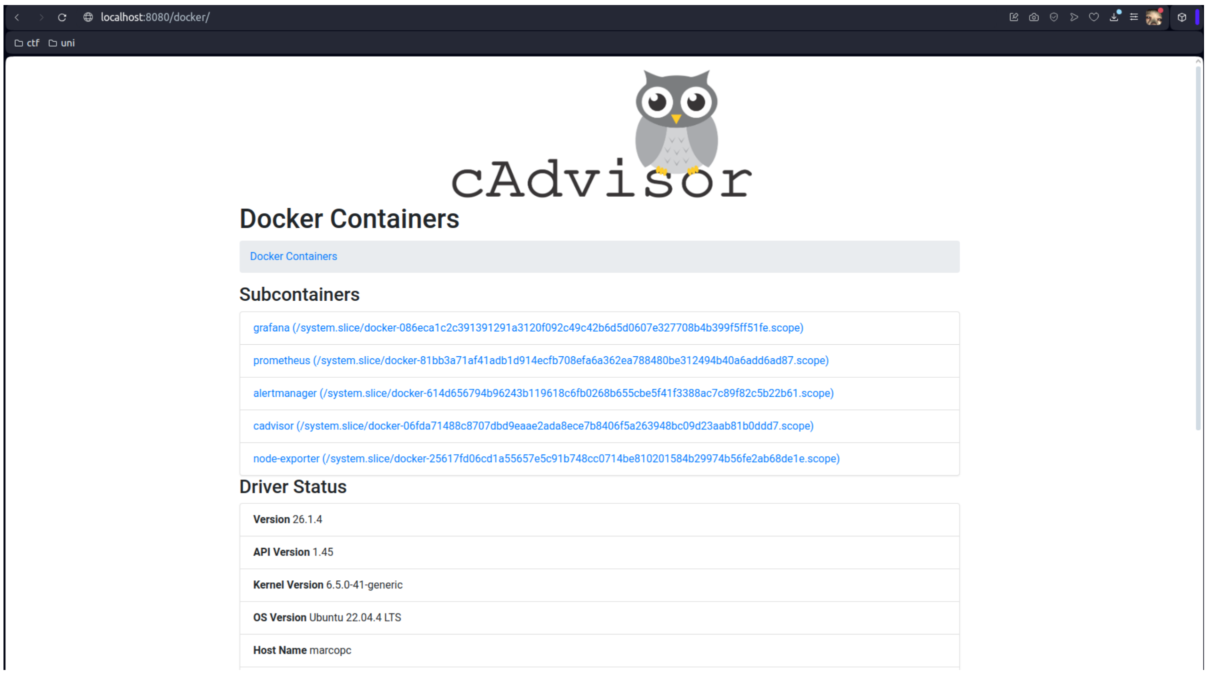
root

Docker Containers
Podman Containers

Subcontainers

/init.scope
/system.slice
/user.slice

If you click "Docker Containers" you get the following page:



Docker Containers

Docker Containers

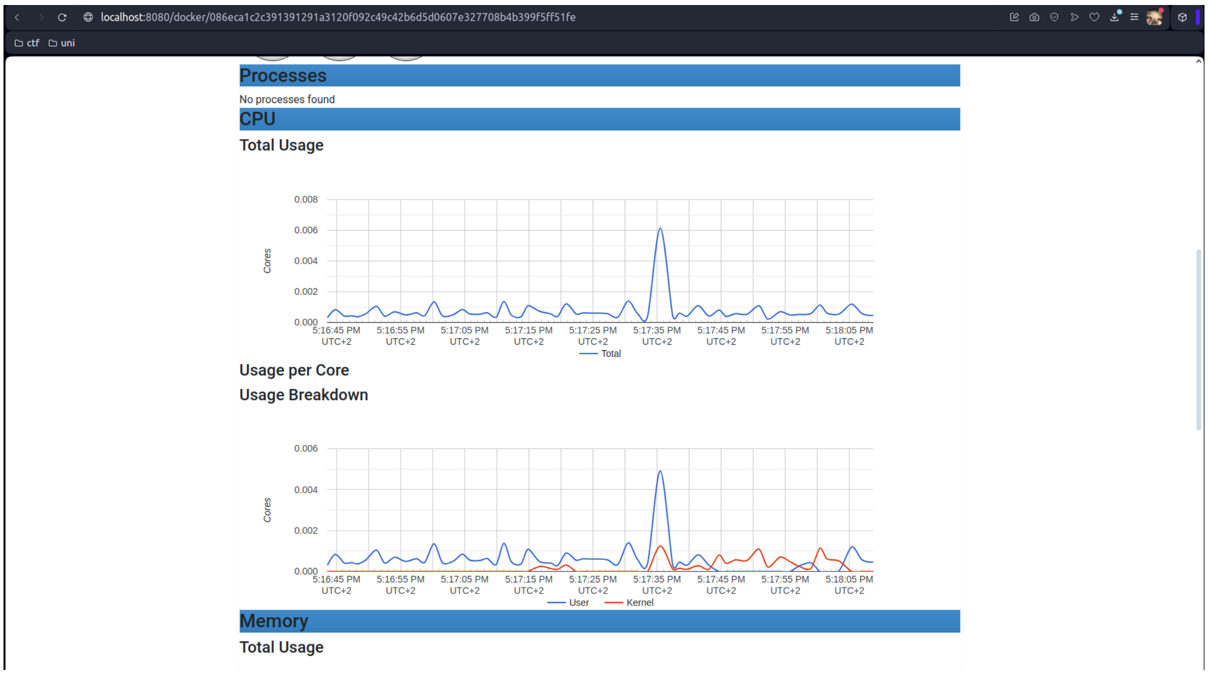
Subcontainers

grafana (/system.slice/docker-086eca1c2c391391291a3120f092c49c42b6d5d0607e327708b4b399f5ff51fe.scope)
prometheus (/system.slice/docker-81bb3a71af41adb1d914ecfb708efa6a362ea788480be312494b40a6add6ad87.scope)
alertmanager (/system.slice/docker-614d656794b96243b119618c6fb0268b655cbe5f41f3388ac7c89f82c5b22b61.scope)
cadvisor (/system.slice/docker-06fda71488c8707dbd9eaae2ada8ece7b8406f5a263948bc09d23aab81b0ddd7.scope)
node-exporter (/system.slice/docker-25617fd06cd1a55657e5c91b748cc0714be810201584b29974b56fe2ab68de1e.scope)

Driver Status

Version 26.1.4
API Version 1.45
Kernel Version 6.5.0-41-generic
OS Version Ubuntu 22.04.4 LTS
Host Name marcopc

If you click on "Grafana" container, you can monitor real-time CPU usage and other metrics for this container:



With cAdvisor set up and integrated with Prometheus, you can now monitor detailed metrics about your containers' resource usage and performance. This setup, combined with the previously configured tools (Prometheus, Node Exporter, Alertmanager), provides a comprehensive observability stack for your containerized environment.

Now that the monitoring tools are fully configured and operational, it's time to move on to the logging part of the setup. We will configure Loki and Promtail to handle logs from our Docker containers.

4 Setting Up Logging Services

4.1 Loki & Promtail

Loki is a highly efficient and scalable logging system designed to work seamlessly with Grafana and Prometheus, offering a unified observability stack that combines metrics and logs. Unlike traditional logging systems, Loki avoids full-text indexing and instead uses labels to index logs, which makes it highly efficient and cost-effective.

Promtail is an agent designed to ship the contents of local log files to a Loki instance. It discovers targets, scrape logs, and pushes them to Loki for storage and query logs efficiently, integrating them with your existing metrics infrastructure to provide comprehensive observability.

4.1.1 Configuring Loki

First, you need to configure Docker to use Loki as the logging driver. This is necessary because it allows Docker to send container logs directly to Loki, ensuring that all logs are captured and can be queried from a centralized location. This integration simplifies log management and makes it easier to correlate logs with metrics for troubleshooting and analysis.

Create or update the `daemon.json` file created at `/etc/docker/daemon.json` with the following content:

```json
{
    "log-driver": "loki",
    "log-opts": {
        "loki-url": "http://localhost:3100/loki/api/v1/push",
        "loki-batch-size": "400",
        "mode": "non-blocking"
    }
}
```

- `log-driver`: sets Loki as the default logging driver for Docker;

- `loki-url`: specifies the Loki URL for pushing logs;
- `loki-batch-size`: defines the number of log messages sent in each request, ensuring efficient log trasmissions;
- `mode`: uses `non-blocking` mode to allow the application to continue running even if a log trasmission is temporarily delayed, preventing log-related issues from affecting application performance.

After updating `daemon.json`, restart the Docker daemon to apply the changes:

```
sudo systemctl restart docker
```

Next, extend your `docker-compose.yml` file to include the Loki service. The updated file should look like this:

```
version: '3.8'

volumes:
  prometheus-data: {}
  alertmanager-data: {}
  grafana-data: {}

services:
  node_exporter:
    image: quay.io/prometheus/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    network_mode: host
    pid: host
    volumes:
      - '/proc:/host/proc'
    command:
      - '--path.rootfs=/host'

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - ./prometheus/rules.yml:/etc/prometheus/rules.yml
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--web.enable-lifecycle'
    ports:
      - '9090:9090'

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    restart: unless-stopped
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    ports:
      - '8080:8080'

  alertmanager:
    image: quay.io/prometheus/alertmanager:latest
```

```yaml
    container_name: alertmanager
    restart: unless-stopped
    volumes:
      - ./alertmanager/alertmanager.yml:/etc/alertmanager/alertmanager.yml
      - alertmanager-data:/alertmanager
    command:
      - '--config.file=/etc/alertmanager/alertmanager.yml'
    ports:
      - '9093:9093'

  loki:
    image: grafana/loki:latest
    container_name: loki
    volumes:
      - ./loki/loki.yml:/etc/loki/loki.yml
    command:
      - '--config.file=/etc/loki/loki.yml'
    ports:
      - "3100:3100"
```

Create a directory for Loki:

```
mkdir loki
```

Create the `loki.yml` configuration file inside `loki` directory with the following content:

```yaml
auth_enabled: false

server:
  http_listen_port: 3100
  grpc_listen_port: 9096
  log_level: debug

common:
  instance_addr: 127.0.0.1
  path_prefix: /tmp/loki
  storage:
    filesystem:
      chunks_directory: /tmp/loki/chunks
      rules_directory: /tmp/loki/rules
  replication_factor: 1
  ring:
    kvstore:
      store: inmemory

query_range:
  results_cache:
    cache:
      embedded_cache:
        enabled: true
        max_size_mb: 100

schema_config:
  configs:
    - from: 2020-10-24
      store: tsdb
      object_store: filesystem
      schema: v13
      index:
        prefix: index_
        period: 24h

ruler:
  alertmanager_url: http://localhost:9093
```

This configuration file initializes and sets up the Loki logging system. It specifies the server ports, log level, storage directories, schema configuration, and the Alertmanager URL for alerting.

4.1.2 Configuring Promtail

Extend your `docker-compose.yml` file to include the Promtail service:

```
version: '3.8'

volumes:
  prometheus-data: {}
  alertmanager-data: {}
  grafana-data: {}

services:
  node_exporter:
    image: quay.io/prometheus/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    network_mode: host
    pid: host
    volumes:
      - '/proc:/host/proc'
    command:
      - '--path.rootfs=/host'

  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    restart: unless-stopped
    volumes:
      - ./prometheus/prometheus.yml:/etc/prometheus/prometheus.yml
      - ./prometheus/rules.yml:/etc/prometheus/rules.yml
      - prometheus-data:/prometheus
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--web.enable-lifecycle'
    ports:
      - '9090:9090'

  cadvisor:
    image: gcr.io/cadvisor/cadvisor:latest
    container_name: cadvisor
    restart: unless-stopped
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
    ports:
      - '8080:8080'

  alertmanager:
    image: quay.io/prometheus/alertmanager:latest
```

```yaml
    container_name: alertmanager
    restart: unless-stopped
    volumes:
      - ./alertmanager/alertmanager.yml:/etc/alertmanager/alertmanager.yml
      - alertmanager-data:/alertmanager
    command:
      - '--config.file=/etc/alertmanager/alertmanager.yml'
    ports:
      - '9093:9093'

  loki:
    image: grafana/loki:latest
    container_name: loki
    volumes:
      - ./loki/loki.yml:/etc/loki/loki.yml
    command:
      - '--config.file=/etc/loki/loki.yml'
    ports:
      - "3100:3100"

  promtail:
    image: grafana/promtail:latest
    container_name: promtail
    volumes:
      - /var/log:/var/log
      - ./promtail/promtail.yml:/etc/promtail/promtail.yml
    command:
      - '--config.file=/etc/promtail/promtail.yml'

  grafana:
    image: grafana/grafana:latest
    container_name: grafana
    restart: unless-stopped
    volumes:
      - ./grafana/datasources.yml:/etc/grafana/provisioning/datasources/datasources.yml
      - 'grafana-data:/var/lib/grafana'
    ports:
      - '3000:3000'
```

Create a directory for Promtail:

```
mkdir promtail
```

Create the `promtail.yml` configuration file inside the `promtail` directory with following content:

```yaml
server:
  http_listen_port: 9080
  grpc_listen_port: 0

positions:
  filename: /tmp/positions.yaml

clients:
  - url: http://loki:3100/loki/api/v1/push

scrape_configs:
  - job_name: local
    static_configs:
    - targets:
        - localhost
      labels:
        job: varlogs
        __path__: /var/log/*log
        stream: stdout

  - job_name: docker
    pipeline_stages:
      - docker: {}
    static_configs:
      - labels:
          job: docker
          __path__: /var/lib/docker/containers/*/*-json.log
```

This configuration file sets up Promtail to scrape logs from the local system and Docker containers. It defines two jobs: one for local logs (`/var/log/log`) and one for Docker container logs (`/var/lib/docker/containers/*/*-json.log`).

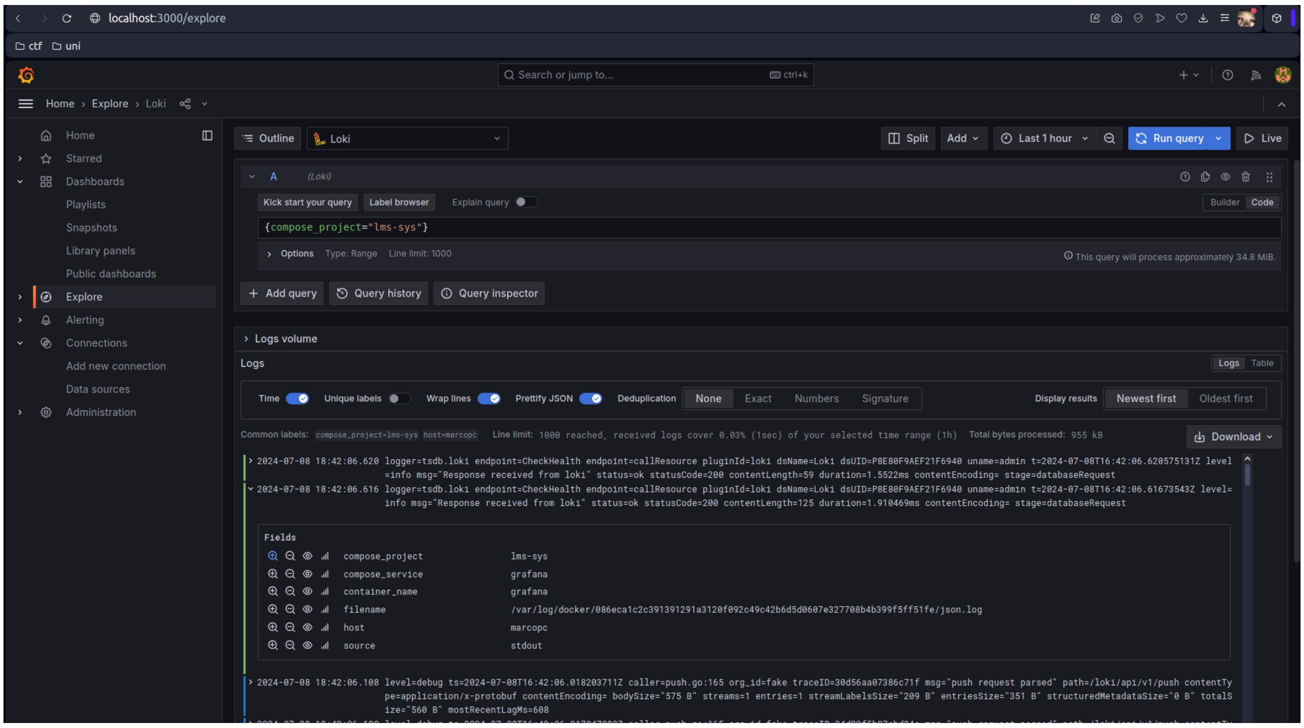
To compose the file, ensure you are in the project directory and run

```
docker compose up -d
```

Now, it's time to configure Grafana to use Loki (like we did with Prometheus). Open Grafana at `http://localhost:3000` and log in with your credentials:

- Go to "Connections" > "Data Sources" > "Add data source".
- Select "Loki"
- Set the URL to `http://loki:3100`
- Click "Save & Test".

This will add Loki as a data source, allowing you to query and visualize logs.



```
{compose_project="lms-sys"}
```

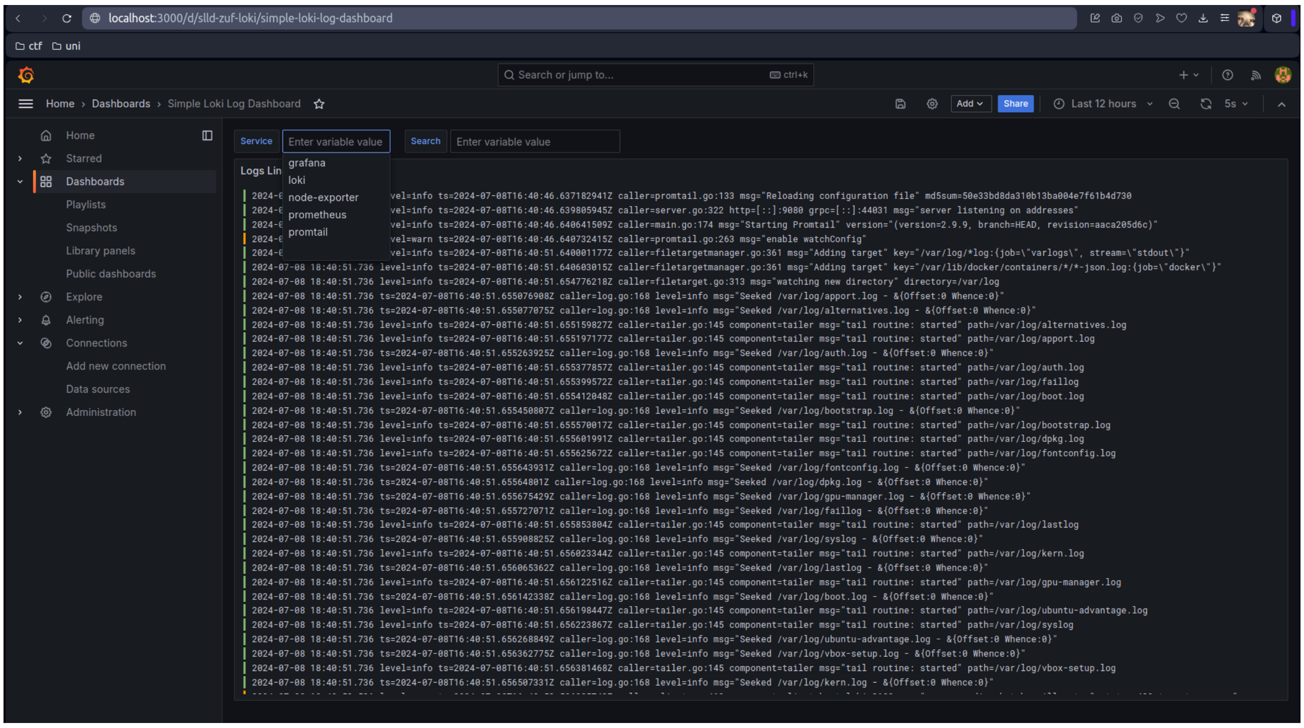
Options Type: Range Line limit: 1000   This query will process approximately 34.8 MiB.

Time ✓ Unique labels Wrap lines ✓ Prettify JSON ✓ Deduplication [✓ None] Exact Numbers Signature   Display results [✓ Newest first] Oldest first

Common labels: compose_project=lms-sys host=marcopc   Line limit: 1000 reached, received logs cover 0.03% (1sec) of your selected time range (1h)   Total bytes processed: 955 kB

```
> 2024-07-08 18:42:06.620 logger=tsdb.loki endpoint=CheckHealth endpoint=callResource pluginId=loki dsName=Loki dsUID=P8E80F9AEF21F6940 uname=admin t=2024-07-08T16:42:06.620575131Z level
                          =info msg="Response received from loki" status=ok statusCode=200 contentLength=59 duration=1.5522ms contentEncoding= stage=databaseRequest
v 2024-07-08 18:42:06.616 logger=tsdb.loki endpoint=CheckHealth endpoint=callResource pluginId=loki dsName=Loki dsUID=P8E80F9AEF21F6940 uname=admin t=2024-07-08T16:42:06.61673543Z level=
                          info msg="Response received from loki" status=ok statusCode=200 contentLength=125 duration=1.910469ms contentEncoding= stage=databaseRequest

  Fields
      compose_project          lms-sys
      compose_service          grafana
      container_name           grafana
      filename                 /var/log/docker/086eca1c2c391391291a3120f092c49c42b6d5d0607e327708b4b399f5ff51fe/json.log
      host                     marcopc
      source                   stdout

> 2024-07-08 18:42:06.108 level=debug ts=2024-07-08T16:42:06.018203711Z caller=push.go:165 org_id=fake traceID=30d56aa07386c71f msg="push request parsed" path=/loki/api/v1/push contentTy
                          pe=application/x-protobuf contentEncoding= bodySize="575 B" streams=1 entries=1 streamLabelsSize="209 B" entriesSize="351 B" structuredMetadataSize="0 B" totalS
                          ize="560 B" mostRecentLagMs=608
```

Additionally, I have added a simple pre-built Loki dashboard to visualize the logs:



```
2024-0         vel=info ts=2024-07-08T16:40:46.637182941Z caller=promtail.go:133 msg="Reloading configuration file" md5sum=50e33bd8da310b13ba004e7f61b4d730
2024-0         vel=info ts=2024-07-08T16:40:46.639805945Z caller=server.go:322 http=[::]:9080 grpc=[::]:44031 msg="server listening on addresses"
2024-0         vel=info ts=2024-07-08T16:40:46.640641509Z caller=main.go:174 msg="Starting Promtail" version="(version=2.9.9, branch=HEAD, revision=aaca205d6c)"
2024-0         vel=warn ts=2024-07-08T16:40:46.640732415Z caller=promtail.go:263 msg="enable watchConfig"
2024-0         vel=info ts=2024-07-08T16:40:51.640001177Z caller=filetargetmanager.go:361 msg="Adding target" key="/var/log/*log:{job=\"varlogs\", stream=\"stdout\"}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.640603015Z caller=filetargetmanager.go:361 msg="Adding target" key="/var/lib/docker/containers/*/*-json.log:{job=\"docker\"}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.654776218Z caller=filetarget.go:313 msg="watching new directory" directory=/var/log
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655076908Z caller=log.go:168 level=info msg="Seeked /var/log/apport.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655077075Z caller=log.go:168 level=info msg="Seeked /var/log/alternatives.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655159827Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/alternatives.log
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655197177Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/apport.log
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655263925Z caller=log.go:168 level=info msg="Seeked /var/log/auth.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655377857Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/auth.log
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655399572Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/faillog
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655412048Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/boot.log
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655450807Z caller=log.go:168 level=info msg="Seeked /var/log/bootstrap.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655570017Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/bootstrap.log
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655601991Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/dpkg.log
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655625672Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/fontconfig.log
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655643931Z caller=log.go:168 level=info msg="Seeked /var/log/fontconfig.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.65564801Z caller=log.go:168 level=info msg="Seeked /var/log/dpkg.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655675429Z caller=log.go:168 level=info msg="Seeked /var/log/gpu-manager.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655727071Z caller=log.go:168 level=info msg="Seeked /var/log/faillog - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.655853804Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/lastlog
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.655908825Z caller=log.go:168 level=info msg="Seeked /var/log/syslog - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.656023344Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/kern.log
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.656065362Z caller=log.go:168 level=info msg="Seeked /var/log/lastlog - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.656122516Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/gpu-manager.log
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.656142338Z caller=log.go:168 level=info msg="Seeked /var/log/boot.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.656198447Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/ubuntu-advantage.log
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.656223867Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/syslog
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.656268849Z caller=log.go:168 level=info msg="Seeked /var/log/ubuntu-advantage.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.656362775Z caller=log.go:168 level=info msg="Seeked /var/log/vbox-setup.log - &{Offset:0 Whence:0}"
2024-07-08 18:40:51.736 level=info ts=2024-07-08T16:40:51.656381468Z caller=tailer.go:145 component=tailer msg="tail routine: started" path=/var/log/vbox-setup.log
2024-07-08 18:40:51.736 ts=2024-07-08T16:40:51.656507331Z caller=log.go:168 level=info msg="Seeked /var/log/kern.log - &{Offset:0 Whence:0}"
```

5 Testing the final infrastructure

To ensure your logging and monitoring setup is functioning correctly, it's important to test it with a sample application. In this section we will deploy an Nginx service using Docker Compose to verify that logs and metrics are being collected and visualized as expected.

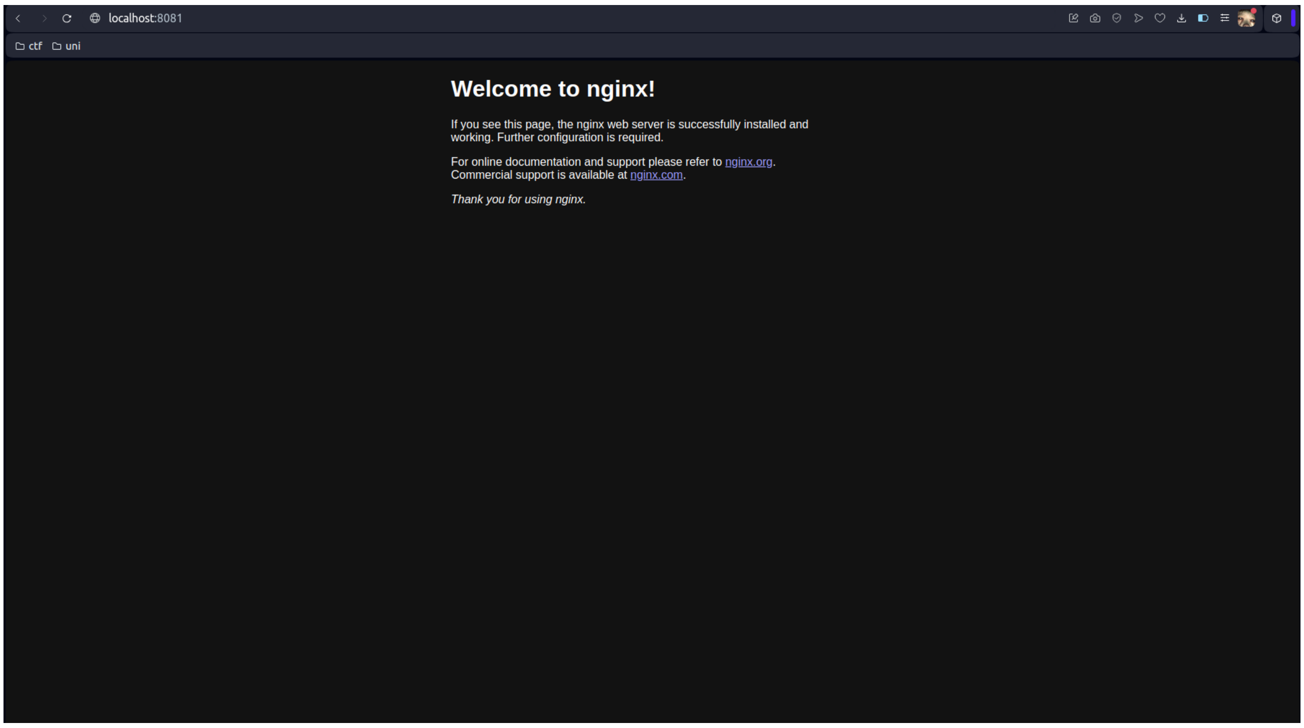
First, create a new `docker-compose.yml` file with the following content to include an Nginx service:

```yaml
version: '3.8'

services:
  nginx:
    image: nginx:latest
    container_name: nginx
    restart: unless-stopped
    ports:
      - '8081:80'
```

Ensure you are in the directory where the new docker compose file is located and run `docker compose up -d`.

Let's verify the setup:

1. **Verify Nginx**: open a web browser and navigate to `http://localhost:8081` to verify that the Nginx server is running



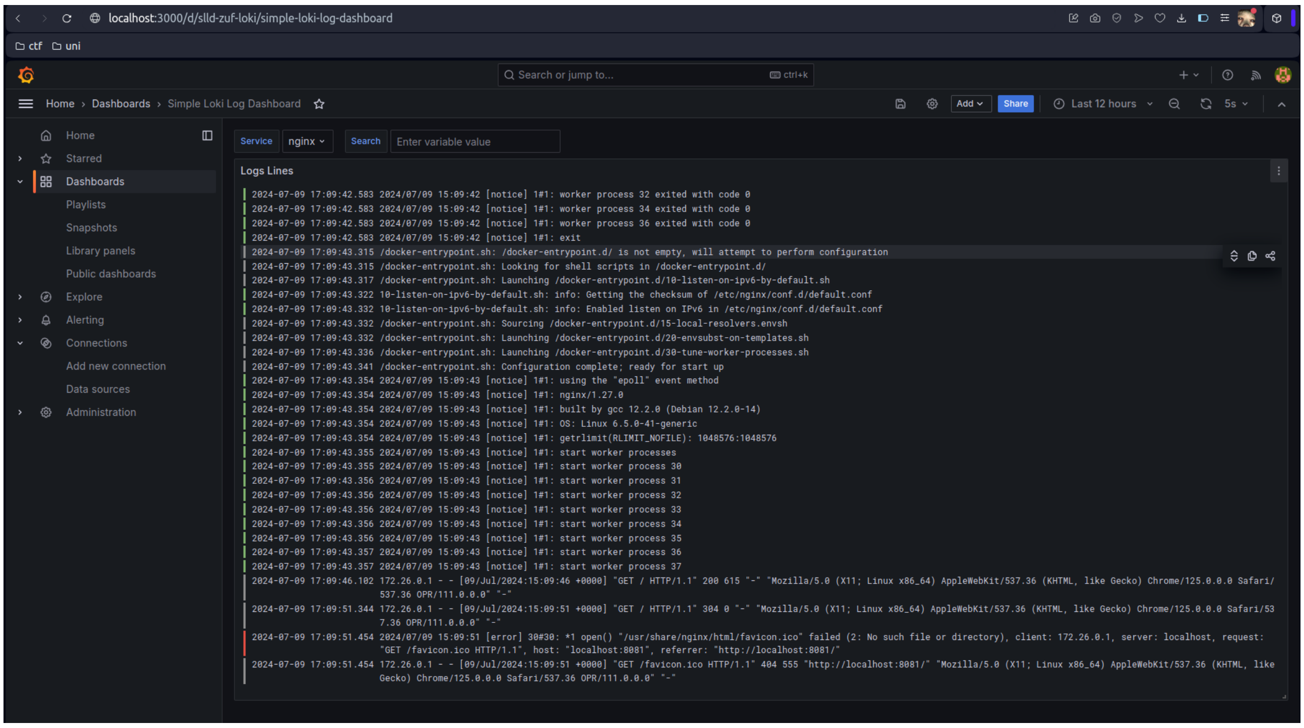
Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

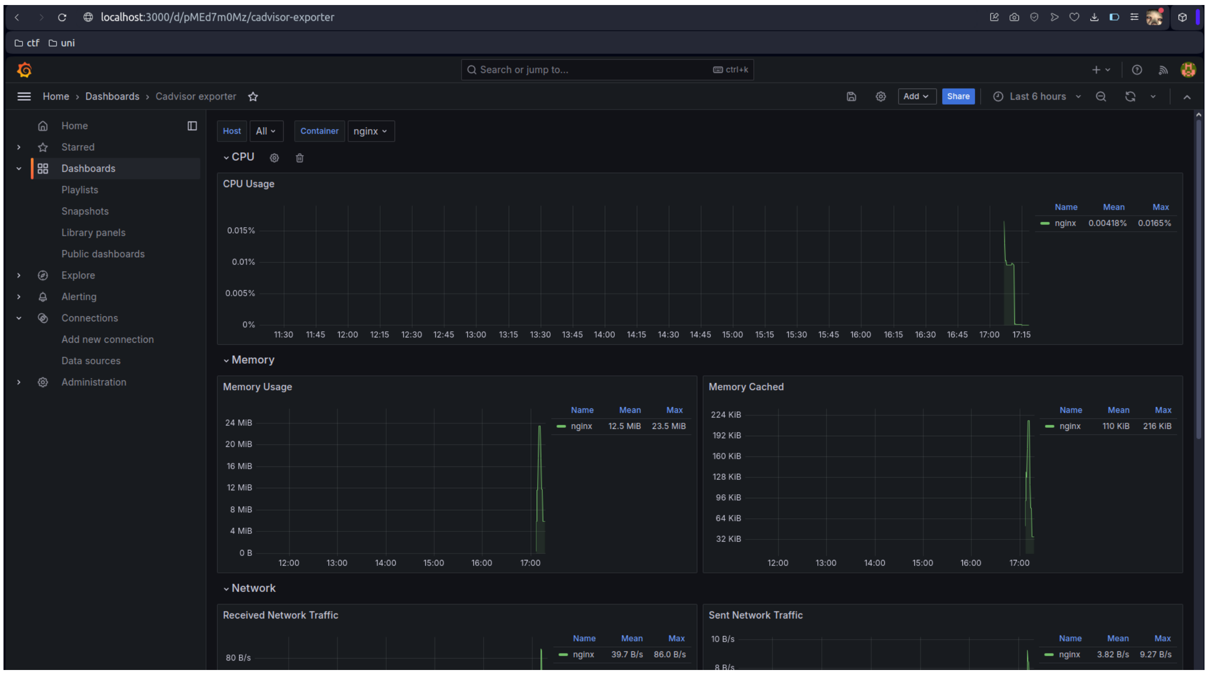
2. **Check metrics in Grafana**: open Grafana at `http://locahost:3000` and go to the dashboards dedicated to cAdvisor:



Logs Lines

```
2024-07-09 17:09:42.583 2024/07/09 15:09:42 [notice] 1#1: worker process 32 exited with code 0
2024-07-09 17:09:42.583 2024/07/09 15:09:42 [notice] 1#1: worker process 34 exited with code 0
2024-07-09 17:09:42.583 2024/07/09 15:09:42 [notice] 1#1: worker process 36 exited with code 0
2024-07-09 17:09:42.583 2024/07/09 15:09:42 [notice] 1#1: exit
2024-07-09 17:09:43.315 /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
2024-07-09 17:09:43.315 /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
2024-07-09 17:09:43.317 /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
2024-07-09 17:09:43.322 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
2024-07-09 17:09:43.332 10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
2024-07-09 17:09:43.332 /docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
2024-07-09 17:09:43.332 /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
2024-07-09 17:09:43.336 /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
2024-07-09 17:09:43.341 /docker-entrypoint.sh: Configuration complete; ready for start up
2024-07-09 17:09:43.354 2024/07/09 15:09:43 [notice] 1#1: using the "epoll" event method
2024-07-09 17:09:43.354 2024/07/09 15:09:43 [notice] 1#1: nginx/1.27.0
2024-07-09 17:09:43.354 2024/07/09 15:09:43 [notice] 1#1: built by gcc 12.2.0 (Debian 12.2.0-14)
2024-07-09 17:09:43.354 2024/07/09 15:09:43 [notice] 1#1: OS: Linux 6.5.0-41-generic
2024-07-09 17:09:43.354 2024/07/09 15:09:43 [notice] 1#1: getrlimit(RLIMIT_NOFILE): 1048576:1048576
2024-07-09 17:09:43.355 2024/07/09 15:09:43 [notice] 1#1: start worker processes
2024-07-09 17:09:43.355 2024/07/09 15:09:43 [notice] 1#1: start worker process 30
2024-07-09 17:09:43.356 2024/07/09 15:09:43 [notice] 1#1: start worker process 31
2024-07-09 17:09:43.356 2024/07/09 15:09:43 [notice] 1#1: start worker process 32
2024-07-09 17:09:43.356 2024/07/09 15:09:43 [notice] 1#1: start worker process 33
2024-07-09 17:09:43.356 2024/07/09 15:09:43 [notice] 1#1: start worker process 34
2024-07-09 17:09:43.356 2024/07/09 15:09:43 [notice] 1#1: start worker process 35
2024-07-09 17:09:43.357 2024/07/09 15:09:43 [notice] 1#1: start worker process 36
2024-07-09 17:09:43.357 2024/07/09 15:09:43 [notice] 1#1: start worker process 37
2024-07-09 17:09:46.102 172.26.0.1 - - [09/Jul/2024:15:09:46 +0000] "GET / HTTP/1.1" 200 615 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/
                        537.36 OPR/111.0.0.0" "-"
2024-07-09 17:09:51.344 172.26.0.1 - - [09/Jul/2024:15:09:51 +0000] "GET / HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/125.0.0.0 Safari/53
                        7.36 OPR/111.0.0.0" "-"
2024-07-09 17:09:51.454 2024/07/09 15:09:51 [error] 30#30: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.26.0.1, server: localhost, request:
                        "GET /favicon.ico HTTP/1.1", host: "localhost:8081", referrer: "http://localhost:8081/"
2024-07-09 17:09:51.454 172.26.0.1 - - [09/Jul/2024:15:09:51 +0000] "GET /favicon.ico HTTP/1.1" 404 555 "http://localhost:8081/" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like
                        Gecko) Chrome/125.0.0.0 Safari/537.36 OPR/111.0.0.0" "-"
```

3. **Check logs in Grafana**: check the Loki's dashboard previously loaded in Grafana:



Testing with the Nginx service ensures that your logging and monitoring infrastructure is capturing data from running services and displaying it correctly in Grafana.

6 Conclusions

Implementing this observability solution establishes a robust foundation for monitoring and logging containerized environments. The integrated toolkit presented in this guide offers deep visibility into your application ecosystem, empowering you with real-time insights into performance metrics, system health, and detailed log analysis. This sophisticated setup enables:

1. Capture system-wide metrics and container-specific data for a complete view of your environment;
2. Efficiently collect, index, and visualize logs from all your containerized applications in one place;
3. Leverage intelligent alerting to identify and address potential problems before they impact your services;
4. Easily adapt the monitoring stack as your containerized ecosystem grows and envolves.

By adopting observability framework, you're not just monitoring your containers: you're gaining a powerful ally in maintaining the health, performance, and reliability of your entire application infrastructure. Whether you're managing a handful of containers or orchestrating a complex microservices architecture, this solution provides the tools necessary for effective oversight and rapid troubleshooting.

The modular nature of this setup allows for future expansion and customization, ensuring that your monitoring capabilities can evolve alongside your containerized applications. With this foundation in place, you're well-equipped to tackle the challenges of modern, dynamic infrastructure management.

For access to the complete set of configuration files and code samples used in this guide, please visit my Github repository. This resource will help you implement and customize the solution to fit your specific needs.