**Cài đặt và cấu hình Prometheus server**

Hướng dẫn cài đặt Prometheus server trên CentOS 7 từ source

**Bước 1: Cài một số gói cần thiết**

yum install wget -y

**Bước 2: Tạo một user cho service prometheus**

useradd --no-create-home --shell /bin/false prometheus

**Bước 3: Tạo và phân quyền thư mục cấu hình và thư mục lưu trữ cho Prometheus**

mkdir /etc/prometheus

mkdir /var/lib/prometheus

chown prometheus:prometheus /etc/prometheus

chown prometheus:prometheus /var/lib/prometheus

**Bước 4: Tải source code prometheus**

cd /opt

wget https://github.com/prometheus/prometheus/releases/download/v2.27.1/prometheus-2.27.1.linux-amd64.tar.gz -O prometheus.tar.gz

tar xvf prometheus.tar.gz

mv prometheus-2.27.1.linux-amd64 prometheus

cp prometheus/prometheus /usr/local/bin/

cp prometheus/promtool /usr/local/bin/

chown prometheus:prometheus /usr/local/bin/prometheus

chown prometheus:prometheus /usr/local/bin/promtool

cp -r prometheus/consoles /etc/prometheus

cp -r prometheus/console\_libraries /etc/prometheus

chown -R prometheus:prometheus /etc/prometheus/consoles

chown -R prometheus:prometheus /etc/prometheus/console\_libraries

rm -rf prometheus\*

cd -

**Bước 5: Tạo một file cấu hình để tự scrape metric của chính prometheus server**

cat <<EOF > /etc/prometheus/prometheus.yml

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: 'prometheus'

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:9090']

EOF

Prometheus sẽ thực hiện scrape metric với chu kỳ là 5s lấy một lần, nếu một job được định nghĩa mà không có *scrape\_interval*thì chu kỳ sẽ được tính theo *scrape\_interval*trong phần *global* là 15s một lần.

**Bước 6: Quản lý service bằng systemd**

cat <<EOF > /etc/systemd/system/prometheus.service

[Unit]

Description=Prometheus

Wants=network-online.target

After=network-online.target

[Service]

User=prometheus

Group=prometheus

Type=simple

ExecStart=/usr/local/bin/prometheus \

--config.file /etc/prometheus/prometheus.yml \

--storage.tsdb.path /var/lib/prometheus/ \

--web.console.templates=/etc/prometheus/consoles \

--web.console.libraries=/etc/prometheus/console\_libraries

[Install]

WantedBy=multi-user.target

EOF

**Bước 7: Khởi động Prometheus service:**

systemctl daemon-reload

systemctl start prometheus

systemctl status prometheus

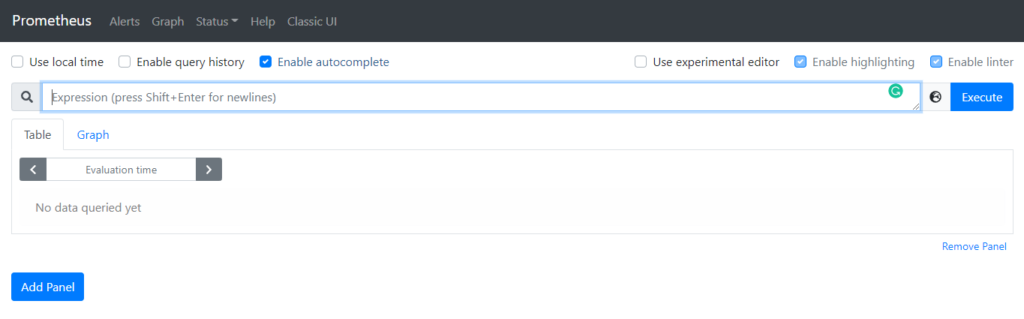
systemctl enable prometheus

**Bước 8: Hoàn tất cài đặt**

Truy cập vào giao diện web của Prometheus với link:

http://172.16.72.171:9090/graph

Nếu thành công bạn sẽ thấy giao diện như sau:



## Cài đặt và cấu hình giám sát hạ tầng máy chủ

### **3.1 Thực hiện trên Node 1**

**Bước 1: Tạo user cho prometheus**

useradd --no-create-home --shell /bin/false node\_exporter

**Bước 2: Tải source code**

cd /opt

wget https://github.com/prometheus/node\_exporter/releases/download/v0.18.1/node\_exporter-0.18.1.linux-amd64.tar.gz

tar xvf node\_exporter-0.18.1.linux-amd64.tar.gz

cp node\_exporter-0.18.1.linux-amd64/node\_exporter /usr/local/bin

chown node\_exporter:node\_exporter /usr/local/bin/node\_exporter

rm -rf node\_exporter-0.18.1.linux-amd64\*

cd -

**Bước 3: Chạy exporter dưới systemd**

cat <<EOF > /etc/systemd/system/node\_exporter.service

[Unit]

Description=Node Exporter

Wants=network-online.target

After=network-online.target

[Service]

User=node\_exporter

Group=node\_exporter

Type=simple

ExecStart=/usr/local/bin/node\_exporter

[Install]

WantedBy=multi-user.target

EOF

Khởi động service:

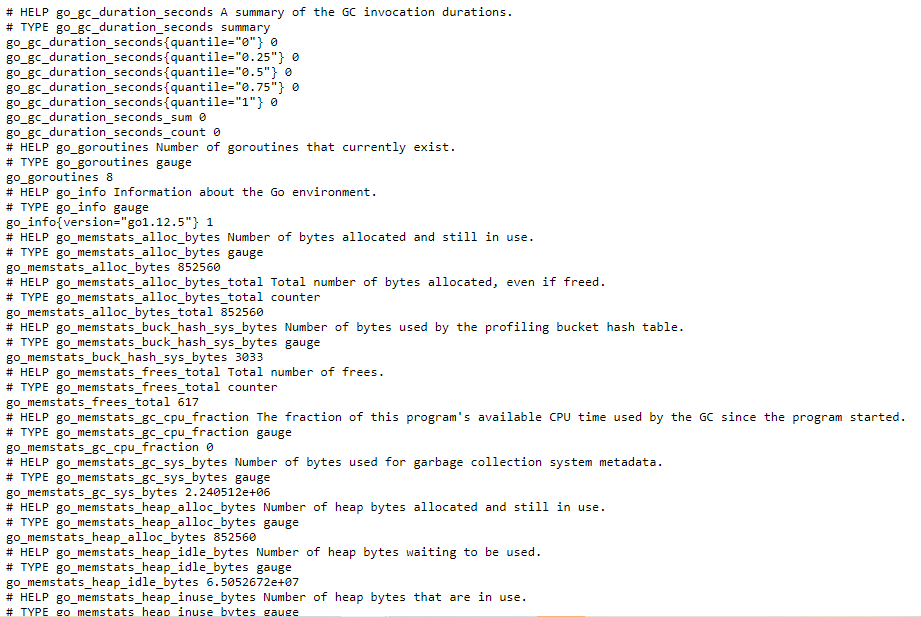
systemctl daemon-reload

systemctl start node\_exporter

systemctl enable node\_exporter

Truy cập vào đường dẫn sau để thấy các metric thu thập được trên node 1: http://172.16.72.173:9100/metrics

Trên giao diện web sẽ hiện thị metric dạng như sau:



### **3.2 Cấu hình trên Prometheus server**

**Bước 4: Add thêm jobs vào node Prometheus server**

Thêm cấu hình của node exporter mới cài đặt vào file /etc/prometheus/prometheus.yml như sau:

global:

scrape\_interval: 15s

scrape\_configs:

- job\_name: 'prometheus'

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:9090']

- job\_name: 'node\_1'

scrape\_interval: 5s

static\_configs:

- targets: ['172.16.72.173:9100']

Prometheus server sẽ thưc hiện scrape metric từ Node 1 mỗi 5s một lần (nếu muốn thay đổi scrape interval thì thay đổi tham số **scrape\_interval** bên trong mỗi jobs, nếu mỗi job không chỉ định tham số này thì sẽ sử dụng theo tham số trong phần **global**.

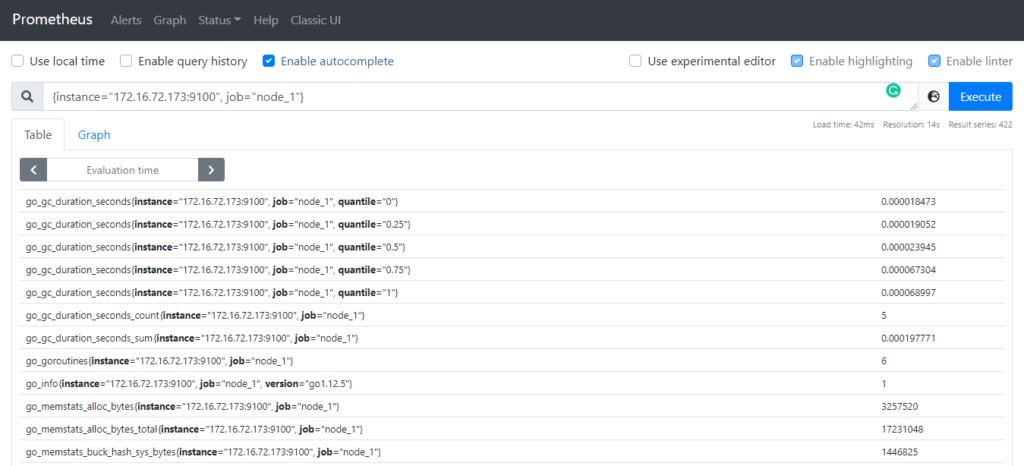
Khởi động lại service:

systemctl restart prometheus

Các metric sẽ được thu thập và lưu lại trên Prometheus server. Giờ ta có thể thực hiện query tới các node exporter từ Prometheus server.

Truy cập http://172.16.72.171:9090/graph để thực hiện query metrics thu thập được

Ví dụ:



# DHIS2 Monitoring

## General monitoring infrastructure

The DHIS2 monitoring infrastructure is designed to expose metrics related to the application runtime and other application-related information. Infrastucture related metrics (such as host metrics, Tomcat or Postgres) are not directly exposed by the application monitoring engine and they have to be collected separately.

The DHIS2 monitoring infrastructure is designed to work with Prometheus (<https://prometheus.io/>). Prometheus is, at its core, a time-series database that scrapes metrics from HTTP endpoints. A time series is a stream of timestamped values that belong to the same metric and the same labels. The labels cause the metrics to be multi-dimensional. Prometheus pulls data from a number of targets (as opposed to a push-based monitoring infrastructure). Prometheus is not an event-based system and this is very different from other time series databases. Prometheus is not designed to catch individual and punctual events in time (such as a service outage for example) but it is designed to **gather pre-aggregated metrics about one or more services**.

## DHIS2 application monitoring

DHIS2 exposes a series of metrics which can be scraped by Prometheus. Currently, the metrics exposed by the application are:

* DHIS2 API (response time, number of calls, etc.)
* JVM (Heap size, Garbage collection, etc.)
* Hibernate (Queries, cache, etc)
* C3P0 Database pool
* Application uptime
* CPU

Metrics are exported by micrometer.io (<http://micrometer.io/>). Micrometer is a Java based framework that acts as a facade over the instrumentation clients for the most popular monitoring systems. It supports many monitoring engines, including Prometheus.

The complete set of metrics is available from the following API endpoint:

/api/metrics

### API Monitoring

API monitoring works by intercepting incoming HTTP requests and records metrics about Spring MVC execution time and results.

The following metrics are exposed for each API endpoint exposed by DHIS2 (/api/\*).

| **name** | **type** | **description** |
| --- | --- | --- |
| seconds\_max | gauge | a moving window of the maximum recorded value in a client-side configurable interval |
| seconds\_count | summary | the number of times the API has been called |
| seconds\_sum | summary | the accumulated duration of the API call |

Each metric exposes is additionally disaggregated by the following tags:

| **tag name** | **description** |
| --- | --- |
| method | the HTTP method (for example, GET or PUT) |
| status | the numeric HTTP status code (for example, 200, 201, 500) |
| uri | the URI template prior to variable substitution (for example, /api/analytics/) |
| exception | the simple name of the exception class thrown (only if an exception is thrown) |
| outcome | request’s outcome based on the status code of the response. 1xx is INFORMATIONAL, 2xx is SUCCESS, 3xx is REDIRECTION, 4xx CLIENT\_ERROR, and 5xx is SERVER\_ERROR |

This is an example of metrics for the /29/analytics endpoint:

dhis2\_seconds\_max{exception="None",method="GET",outcome="SUCCESS",status="200",uri="/29/analytics",} 2.715992829

dhis2\_seconds\_sum{exception="None",method="GET",outcome="SUCCESS",status="200",uri="/29/analytics",} 15.404253163

dhis2\_seconds\_count{exception="None",method="GET",outcome="SUCCESS",status="200",uri="/29/analytics",} 40.0

### JVM Monitoring

JVM Monitoring exposes a set of metrics related to the JVM used by the application.

| **name** | **type** | **description** |
| --- | --- | --- |
| jvm\_memory\_used\_bytes | gauge | The amount of used memory |
| jvm\_memory\_committed\_bytes | gauge | The amount of memory in bytes that is committed for the Java virtual machine to use |
| jvm\_memory\_max\_bytes | gauge | The maximum amount of memory in bytes that can be used for memory management |
| jvm\_gc\_pause\_seconds | summary | Time spent in GC pause |
| jvm\_gc\_pause\_seconds\_max | gauge | Time spent in GC pause |
| jvm\_gc\_max\_data\_size\_bytes | gauge | Max size of old generation memory pool |
| jvm\_gc\_live\_data\_size\_bytes | gauge | Size of old generation memory pool after a full GC |
| jvm\_gc\_memory\_promoted\_bytes\_total | counter | Count of positive increases in the size of the old generation memory pool before GC to after GC |
| jvm\_gc\_memory\_allocated\_bytes | counter | Incremented for an increase in the size of the young generation memory pool after one GC to before the next |
| jvm\_classes\_loaded\_classes | gauge | The number of classes that are currently loaded in the Java virtual machine |
| jvm\_classes\_unloaded\_classes\_total | counter | The total number of classes unloaded since the Java virtual machine has started execution |
| jvm\_threads\_states\_threads | gauge | The current number of threads having NEW state |
| jvm\_threads\_peak\_threads | gauge | The peak live thread count since the Java virtual machine started or peak was reset |
| jvm\_threads\_live\_threads | gauge | The current number of live threads including both daemon and non-daemon threads |
| jvm\_threads\_daemon\_threads | gauge | The current number of live daemon threads |
| jvm\_buffer\_memory\_used\_bytes | gauge | An estimate of the memory that the Java virtual machine is using for this buffer pool |
| jvm\_buffer\_total\_capacity\_bytes | gauge | An estimate of the total capacity of the buffers in this pool |

### Uptime status monitoring

| **name** | **type** | **description** |
| --- | --- | --- |
| process\_uptime\_seconds | gauge | The uptime of the Java virtual machine |
| process\_start\_time\_seconds | gauge | Start time of the process since unix epoch. |

### CPU monitoring

| **name** | **type** | **description** |
| --- | --- | --- |
| system\_load\_average\_1m | gauge | The sum of the number of runnable entities queued to available processors and the number of runnable entities running on the available processors averaged over a period of time |
| system\_cpu\_count | gauge | The number of processors available to the Java virtual machine |
| process\_cpu\_usage | gauge | The "recent cpu usage" for the Java Virtual Machine process |
| system\_cpu\_usage | gauge | The "recent cpu usage" for the whole system |

### C3P0 Connection pool monitoring

| **name** | **type** | **description** |
| --- | --- | --- |
| jdbc\_connections\_idle | gauge | Number of idle connections |
| jdbc\_connections\_active | gauge | Number of active connections |
| jdbc\_connections\_max | gauge | Number of max connections available in the pool |
| jdbc\_connections\_min | gauge | Number of min connections in the pool |

### Hibernate monitoring

| **name** | **type** | **description** |
| --- | --- | --- |
| hibernate\_sessions\_open\_total | counter | Sessions opened |
| hibernate\_sessions\_closed\_total | counter | Session closed |
| hibernate\_transactions\_total | counter | The number of transactions we know to have been successful |
| hibernate\_transactions\_total | counter | The number of transactions we know to have failed |
| hibernate\_optimistic\_failures\_total | counter | The number of StaleObjectStateExceptions that have occurred |
| hibernate\_flushes\_total | counter | The global number of flushes executed by sessions (either implicit or explicit) |
| hibernate\_connections\_obtained\_total | counter | Get the global number of connections asked by the sessions (the actual number of connections used may be much smaller depending whether you use a connection pool or not) |
| hibernate\_statements\_total | counter | The number of prepared statements that were acquired |
| hibernate\_statements\_total | counter | The number of prepared statements that were released |
| hibernate\_second\_level\_cache\_requests\_total | counter | The number of cacheable entities/collections successfully retrieved from the cache |
| hibernate\_second\_level\_cache\_requests\_total | counter | The number of cacheable entities/collections not found in the cache and loaded from the database |
| hibernate\_second\_level\_cache\_puts\_total | counter | The number of cacheable entities/collections put in the cache |
| hibernate\_entities\_deletes\_total | counter | The number of entity deletes |
| hibernate\_entities\_fetches\_total | counter | The number of entity fetches |
| hibernate\_entities\_inserts\_total | counter | The number of entity inserts |
| hibernate\_entities\_loads\_total | counter | The number of entity loads |
| hibernate\_entities\_updates\_total | counter | The number of entity updates |
| hibernate\_collections\_deletes\_total | counter | The number of collection deletes |
| hibernate\_collections\_fetches\_total | counter | The number of collection fetches |
| hibernate\_collections\_loads\_total | counter | The number of collection loads |
| hibernate\_collections\_recreates\_total | counter | The number of collections recreated |
| hibernate\_collections\_updates\_total | counter | The number of collection updates |
| hibernate\_cache\_natural\_id\_puts\_total | counter | The number of cacheable naturalId lookups put in cache |
| hibernate\_cache\_natural\_id\_requests\_total | counter | The number of cached naturalId lookups successfully retrieved from cache |
| hibernate\_cache\_natural\_id\_requests\_total | counter | The number of cached naturalId lookups not found in cache |
| hibernate\_query\_natural\_id\_executions\_max\_seconds | gauge | The maximum query time for naturalId queries executed against the database |
| hibernate\_query\_natural\_id\_executions\_total | counter | The number of naturalId queries executed against the database |
| hibernate\_query\_executions\_total | counter | The number of executed queries |
| hibernate\_query\_executions\_max\_seconds | gauge | The time of the slowest query |

## DHIS2 Monitoring architecture

The monitoring subsytem revolves around the concept of Micrometer registry. A MeterRegistry collects metrics and exports them in the specified format defined by the chosen monitoring platform. Each supported monitoring system has an implementation of MeterRegistry. Since DHIS2 supports Prometheus, the configuration class org.hisp.dhis.monitoring.prometheus.config.PrometheusMonitoringConfig is responsible for configuring and initializing the PrometheusMeterRegistry.

DHIS2 exports the metrics described above through micrometer exporters, which are configured as Spring @Configuration classes.

All the exporters are located in the dhis-support-system module, under the monitoring package.

The API exporter is configured as a Servlet filter. The webMetricsFilter filter is declared in the web.xml file of the dhis-web-portal module.

Finally, the PrometheusScrapeEndpointController is the controller responsible for exposing the /api/metrics endpoint. The controller simply access the PrometheusMeterRegistry and renders the metrics using the appropriate Prometheus-friendly format.

## DHIS2 Monitoring configuration

The monitoring subsystem is disabled by default, and can be enabled by defining a set of properties in the dhis.conf DHIS 2 configuration file.

Each metrics cluster has to be explicitely enabled in order for the metrics to be exported. The metrics can be enabled by setting to on the following configuration keys (the default is off):

| **key name** | **value** | **metrics** |
| --- | --- | --- |
| monitoring.api.enabled | off | on | API |
| monitoring.jvm.enabled | off | on | JVM |
| monitoring.dbpool.enabled | off | on | Connection Pool |
| monitoring.hibernate.enabled | off | on | Hibernate |
| monitoring.uptime.enabled | off | on | Uptime |
| monitoring.cpu.enabled | off | on | CPU |

Prometheus’ configuration file is divided into three parts: global, rule\_files, and scrape\_configs.

In the global part we can find the general configuration of Prometheus: scrape\_interval defines how often Prometheus scrapes targets, evaluation\_interval controls how often the software will evaluate rules. Rules are used to create new time series and for the generation of alerts.

The rule\_files block contains information of the location of any rules we want the Prometheus server to load.

The last block of the configuration file is named scape\_configs and contains the information which resources Prometheus monitors.

A simple DHIS2 Prometheus monitoring file looks like this example:

global:

scrape\_interval: 15s

evaluation\_interval: 15s

scrape\_configs:

- job\_name: "dhis2"

metrics\_path: "/dhis/api/metrics"

basic\_auth:

username: admin

password: district

static\_configs:

- targets: ["localhost:80"]

The global scrape\_interval is set to 15 seconds which is enough for most use cases.

In the scrape\_configs part we have defined the DHIS2 exporter. The basic\_auth blocks contains the credentials required to access the metrics API: consider creating an ad-hoc user only for accessing the metrics endpoint.

Prometheus may or may not run on the same server as DHIS2: in the above configuration, it is assumed that Prometheus monitors only one DHIS2 instance, running on the same server as Prometheus, so we use localhost.