10. Stacklight LMA

Chapter Details	
Chapter Goal	Utilize Stacklight to monitor MCP
Chapter Sections	*10.1. Introuction to Stacklight* *10.2. Add a New Metric* *10.3. Grafana Monitoring UI* *10.4. Utilizing Prometheus*

10.1. Introuction to Stacklight ¶

Stacklight is the logging, monitoring, and alerting (LMA) toolchain of MCP. It leverages multiple open source components to monitor OpenStack, Kubernetes, and OpenContrail services / nodes and can notify users of critical conditions. In this section, we will explore various components of Stacklight LMA including its configurations and user interfaces.

Step 1 Log-in to the host and identify the monitoring node:

```
stack@kvm01:~$ virsh list
 Id     Name                              State
----------------------------------------------------
 1      cfg01.trainings.local             running
 2      mon03.trainings.local             running
 3      mon02.trainings.local             running
 4      mon01.trainings.local             running
 5      cid02.trainings.local             running
 6      cid01.trainings.local             running
 7      prx01.trainings.local             running
 8      cid03.trainings.local             running
 9      gtw01.trainings.local             running
 10     cmp001.trainings.local            running
 11     ctl02.trainings.local             running
 12     ctl03.trainings.local             running
 13     ctl01.trainings.local             running
```

Nodes with names starting with *mon* are the monitoring nodes.

Step 2 Become sudo and log-in to the monitoring node *mon01.trainings.local*:

```
stack@kvm01:~$ sudo -i
```

```
root@kvm01:~# ssh mon01
root@mon01:~#
```

Step 3 List the Docker containers running in the *mon01* node and take note of the *CONTAINER ID* of the *prometheus* container:

```
root@mon01:~# docker ps
a1d7f4b7045a        docker-prod-.../prometheus:stable
4aa1f4250eee        docker-prod-.../prometheus_relay:stable
9a2d1a6a8761        docker-prod-.../alertmanager:stable
5ed36d26c101        docker-prod-.../pushgateway:stable

# In this example, prometheus container ID is a1d7f4b7045a
```

Stacklight components run as highly available Docker containers via Docker Swarm.

Step 4 Execute the */bin/bash* process in the *prometheus* container using its *container-ID* (first column of the *docker ps* output):

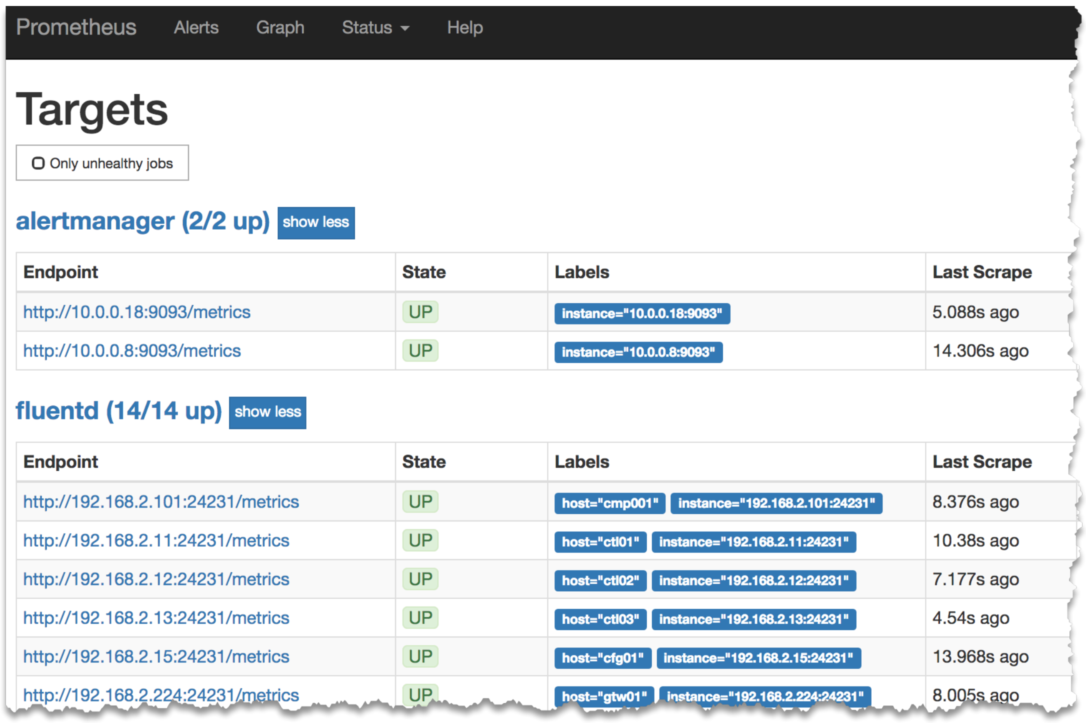
```
root@mon01:~# docker exec -it <container-ID> /bin/bash
root@<container-ID>:/opt/prometheus#
```

Step 5 View the Prometheus configuration within the container:

```
root@<container-ID>:/opt/prometheus# cat /srv/prometheus/prometheus.yml
global:
  evaluation_interval: 15s
  external_labels:
    region: region1
  scrape_interval: 15s
  scrape_timeout: 15s
alerting:
  alertmanagers:
    # docker_swarm_alertmanager
    - dns_sd_configs:
      - names: [tasks.monitoring_alertmanager]
        type: A
        port: 9093
...
```

This is the configuration file of Prometheus and defines the targets to scrape with intervals, timeouts, etc.

Step 6 Navigate to `http://192.168.2.70:15010/targets` to view this configuration on the Prometheus UI:



Targets

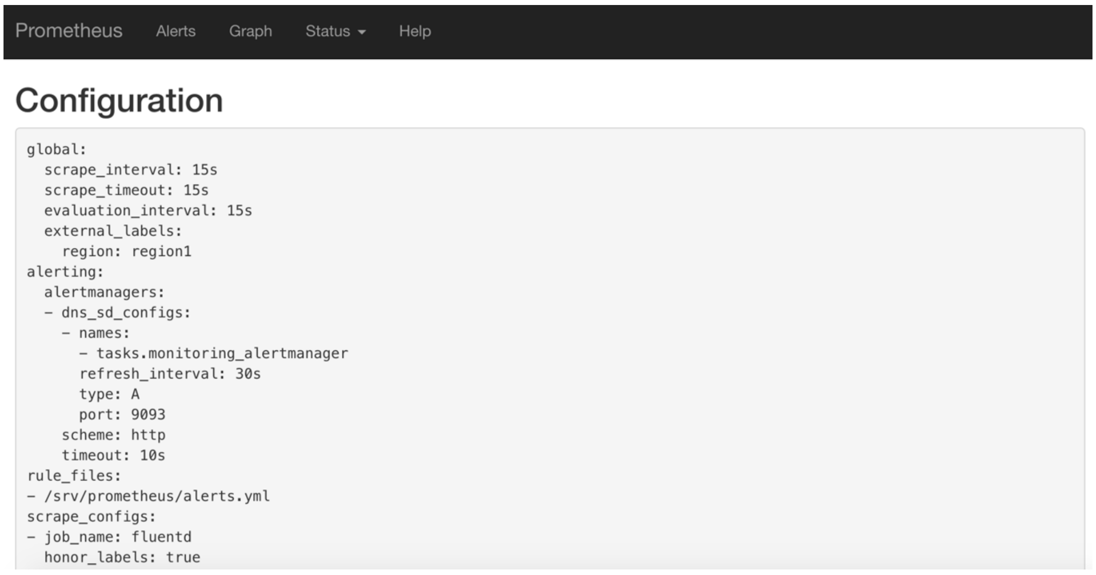
alertmanager (2/2 up)

Endpoint	State	Labels	Last Scrape
http://10.0.0.18:9093/metrics	UP	instance="10.0.0.18:9093"	5.088s ago
http://10.0.0.8:9093/metrics	UP	instance="10.0.0.8:9093"	14.306s ago

fluentd (14/14 up)

Endpoint	State	Labels	Last Scrape
http://192.168.2.101:24231/metrics	UP	host="cmp001" instance="192.168.2.101:24231"	8.376s ago
http://192.168.2.11:24231/metrics	UP	host="ctl01" instance="192.168.2.11:24231"	10.38s ago
http://192.168.2.12:24231/metrics	UP	host="ctl02" instance="192.168.2.12:24231"	7.177s ago
http://192.168.2.13:24231/metrics	UP	host="ctl03" instance="192.168.2.13:24231"	4.54s ago
http://192.168.2.15:24231/metrics	UP	host="cfg01" instance="192.168.2.15:24231"	13.968s ago
http://192.168.2.224:24231/metrics	UP	host="gtw01" instance="192.168.2.224:24231"	8.005s ago

Step 7 Navigate to **Status** > **Configuration** on the Prometheus UI to view the Prometheus configuration:



Configuration

```
global:
  scrape_interval: 15s
  scrape_timeout: 15s
  evaluation_interval: 15s
  external_labels:
    region: region1
alerting:
  alertmanagers:
  - dns_sd_configs:
    - names:
      - tasks.monitoring_alertmanager
      refresh_interval: 30s
      type: A
      port: 9093
    scheme: http
    timeout: 10s
rule_files:
- /srv/prometheus/alerts.yml
scrape_configs:
- job_name: fluentd
  honor_labels: true
```

10.2. Add a New Metric

Telegraf utilizes plugins to collect data from your infrastructure. This model makes it easy to add new Metrics at will!

The list of input plugins can be found here: https://docs.influxdata.com/telegraf/v1.8/plugins/inputs/

We will add the *http_response* plugin using our model. https://github.com/influxdata/telegraf/blob/release-1.8/plugins/inputs/http_response/README.md

Step 1 Log-in to the *cfg01* node:

```
root@kvm01:~# ssh cfg01
root@cfg01:~#
```

Step 2 Create a new directory called *telegraf* in the *stacklight* cluster model directory:

```
root@cfg01:~# mkdir /srv/salt/reclass/classes/cluster/lab28/stacklight/telegraf
```

Step 3 Create a new file called *http_response.yml* in the telegraf directory:

```
root@cfg01:~# touch /srv/salt/reclass/classes/cluster/lab28/stacklight/telegraf/http_re
```

Step 4 Open the *http_response.yml* file and populate it with the following content:

```
parameters:
  telegraf:
    agent:
      input:
        http_response:
          google:
            address: "http://www.google.com/"
            expected_code: 200
```

Presumably, this Telegraf plugin will make a request to *www.google.com/* and expect a response of 200 (OK). This means that in your environment you can add a number of HTTP endpoints here to test that a request sent to them returns an expected response code.

Step 5 Add the newly created file as a class in `stacklight/server.yml`:

```
root@cfg01:~# vim /srv/salt/reclass/classes/cluster/lab28/stacklight/server.yml

# Add the following line under classes:
- cluster.lab28.stacklight.telegraf.http_response
```

Save and quit the file.

Step 6 Let's apply the *telegraf* state to the *mon** nodes which will allow us to test to see if *www.google.com* is reachable from the *mon** nodes:

```
root@cfg01:~# salt 'mon*' state.apply telegraf
# Output should indicate that a new file is created called
# '/etc/telegraf/telegraf.d/input-http_response.conf`
```

Step 7 Log-out of the *cfg01* node and log-in to *mon01* node:

```
root@cfg01:~# logout
Connection to cfg01 closed.
```

```
root@kvm01:~# ssh mon01
root@mon01:~#
```

Step 8 View the `/etc/telegraf/telegraf.d/input-http_response.conf` file:

```
root@mon01:~# cat /etc/telegraf/telegraf.d/input-http_response.conf
[[inputs.http_response]]
  [[inputs.http_response.checks]]
    name = "google"
    expected_code = "200"
    address = "http://www.google.com/"
```

Step 9 Telegraf has a feature which allows you to test the configuration file. Run the following command to test the newly created file:

```
root@mon01:~# telegraf --test --config /etc/telegraf/telegraf.d/input-http_response.cor
* Plugin: inputs.http_response, Collection 1
> http_response,name=google,service=google,host=mon01 status=1i 1540504823000000000
```

The result is a sample output from Telegraf using the provided config file. These values will be scraped by Prometheus periodically.

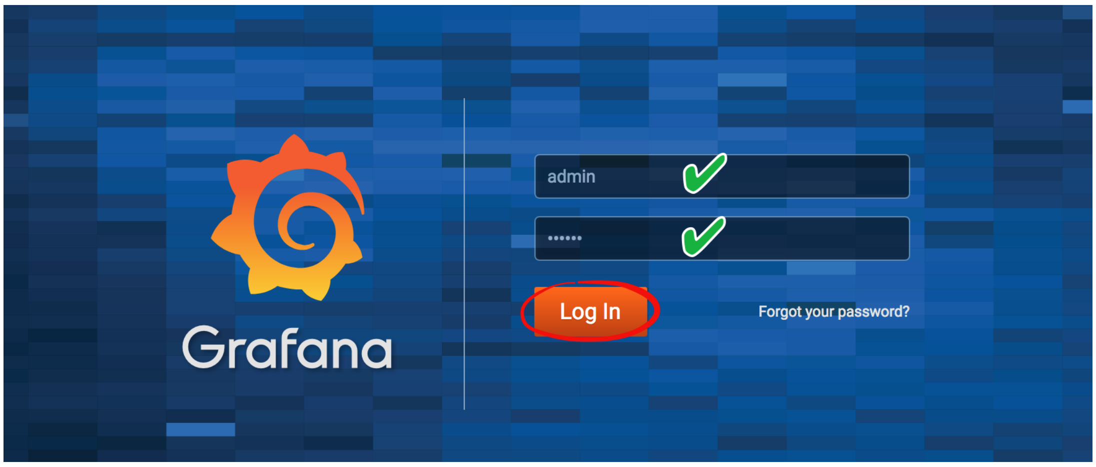
Great! In the upcoming sections, we will take a look at the new Telegraf input using the Prometheus UI.

10.3. Grafana Monitoring UI

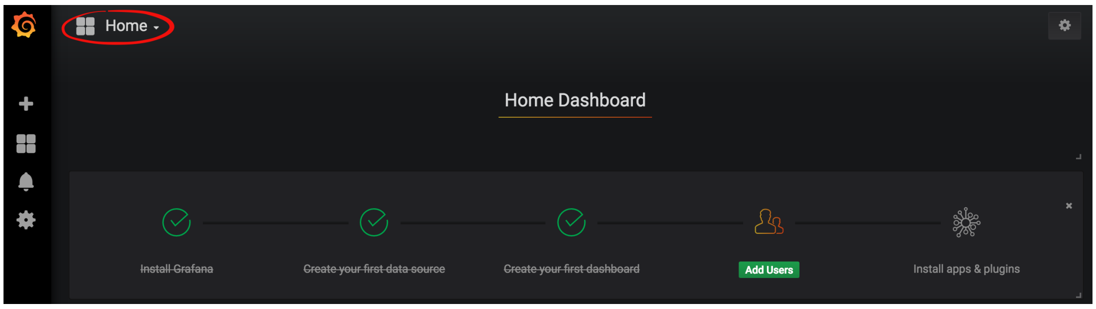
Let's take a look at the components covered by the default Grafana user interface. Grafana is an open platform for analytics and monitoring which supports multiple plugins - including Gnocchi, Prometheus, and Elasticsearch which are all commonly used in the MCP environment.

Step 1 Open a *VNC* connection or *SSH tunnel* to your environment

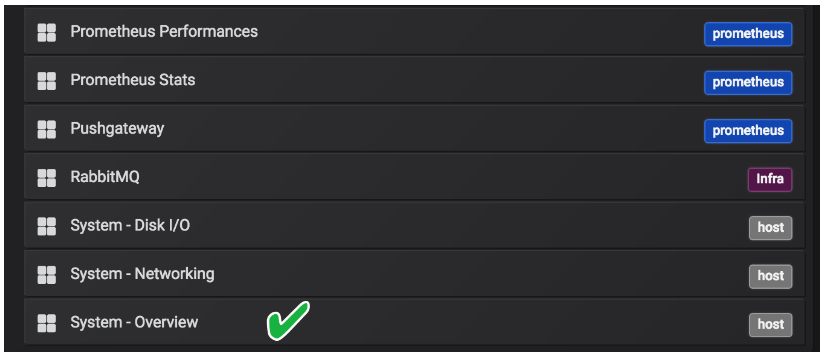
Step 2 Open the *Grafana* user interface by navigating to: `https://192.168.2.80:8084` and log-in using the credentials *admin* / *r00tme*:



Step 3 Click on the drop-down menu on the navigation bar to view the list of available preconfigured dashboards:

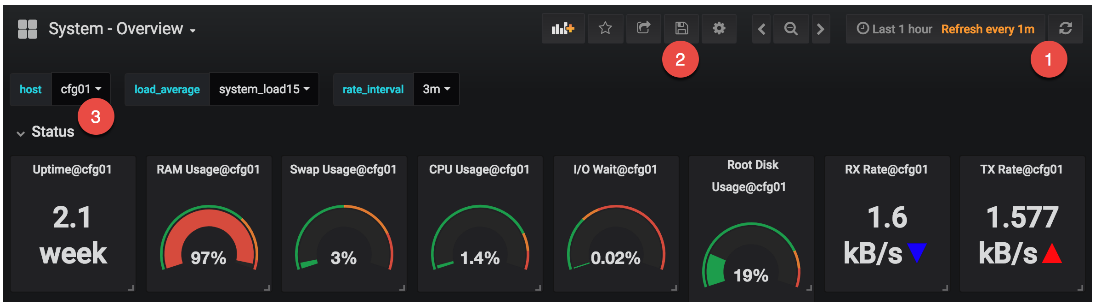


Step 4 Scroll down in the drop-down menu and click on *System - Overview*:



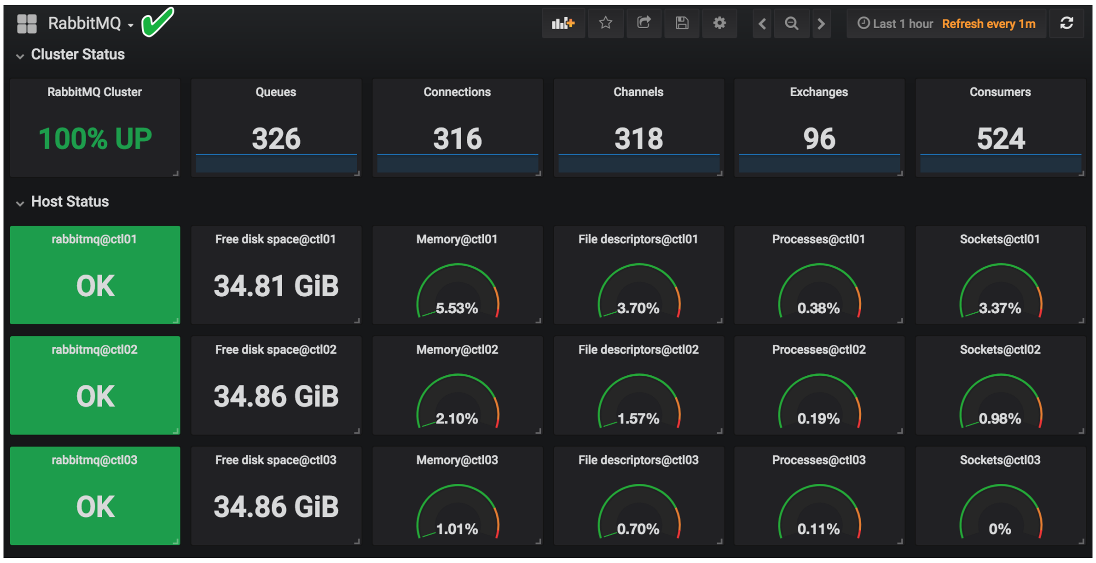
Prometheus Performances	prometheus
Prometheus Stats	prometheus
Pushgateway	prometheus
RabbitMQ	Infra
System - Disk I/O	host
System - Networking	host
System - Overview ✓	host

Here are some items worth noting on the dashboard:



1. Select the refresh rate of the metrics as well as time window to view
2. Save any changes made to the dashboard
3. Select one or more hosts to view side-by-side

Step 5 Navigate to the *RabbitMQ* dashboard:



RabbitMQ Cluster: 100% UP
Queues: 326
Connections: 316
Channels: 318
Exchanges: 96
Consumers: 524

rabbitmq	Free disk space	Memory	File descriptors	Processes	Sockets
rabbitmq@ctl01 OK	34.81 GiB	5.53%	3.70%	0.38%	3.37%
rabbitmq@ctl02 OK	34.86 GiB	2.10%	1.57%	0.19%	0.98%
rabbitmq@ctl03 OK	34.86 GiB	1.01%	0.70%	0.11%	0%

Let's simulate a failure of a RabbitMQ service on one of the nodes to see what it would look like on Grafana.

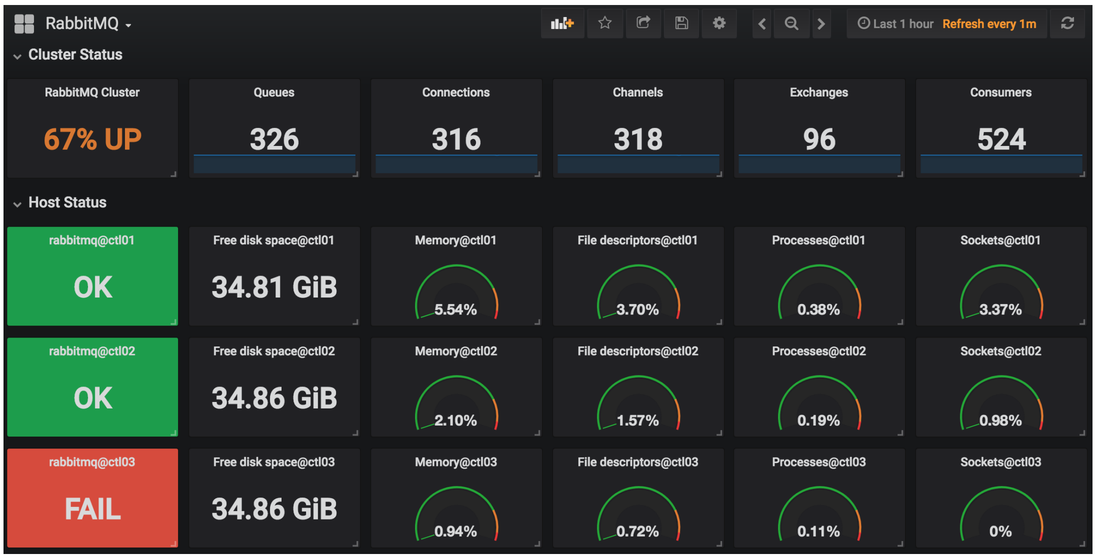
Step 6 Log-in to the *ctl03* node on the console in your environment:

```
root@kvm01:~# ssh ctl03
root@ctl03:~#
```

Step 7 Stop the *rabbitmq-server* service:

```
root@ctl03:~# service rabbitmq-server stop
```

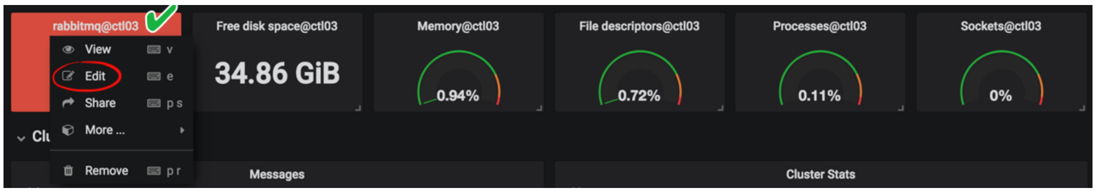
Step 8 Navigate back to the Grafana dashboard and view the *RabbitMQ* status (it may take a minute or so to show the latest status):



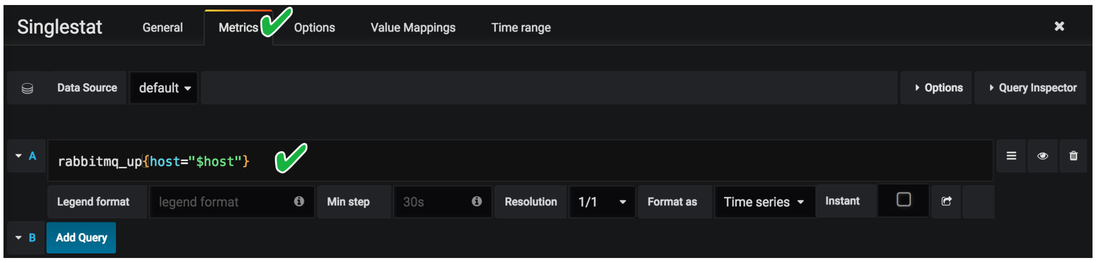
RabbitMQ Cluster: 67% UP
Queues: 326
Connections: 316
Channels: 318
Exchanges: 96
Consumers: 524

rabbitmq	Free disk space	Memory	File descriptors	Processes	Sockets
rabbitmq@ctl01 OK	34.81 GiB	5.54%	3.70%	0.38%	3.37%
rabbitmq@ctl02 OK	34.86 GiB	2.10%	1.57%	0.19%	0.98%
rabbitmq@ctl03 FAIL	34.86 GiB	0.94%	0.72%	0.11%	0%

Step 9 Click on the name of the status box labeled **rabbitmq@ctl03**, then click **Edit**:



Step 10 Scroll down and view the **Metrics** tab of the data:



```
rabbitmq_up{host="$host"}
```

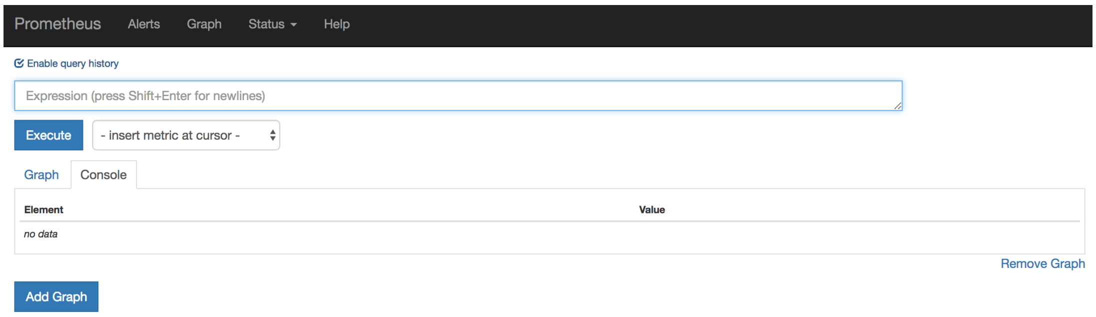
Our default Data Source is Prometheus and the query to Prometheus by Grafana can be viewed in this page (*rabbitmq_up{host="$host"}*)

Although RabbitMQ is a critical component of OpenStack, having one of the services down will not affect the functionality of OpenStack. Let's leave this service as-is for now and proceed to the next section to learn how to view this failure in Prometheus.

10.4. Utilizing Prometheus

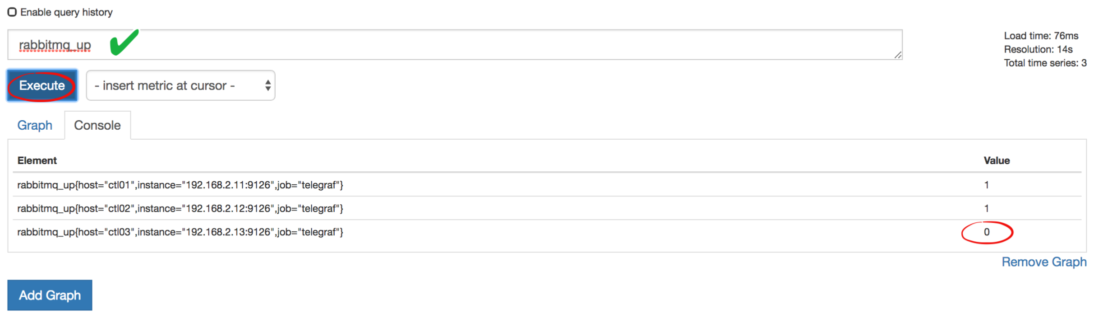
Prometheus is one of the key components of Stacklight which "scrapes" data from various sources and stores them as time series. This data is then retrieved via a flexible query language called PromQL which can be further analyzed for sending alerts or queried by GUI such as Grafana. In this section, we will go over how to utilize Prometheus so that you have more tools at your disposal when debugging your MCP environment.

Step 1 In your *VNC* session or utilizing *SSH tunnel*, open the Prometheus UI by navigating to: *http://192.168.2.70:15010*:



Element	Value
no data	

Step 2 In the *Expression* query field, type in **rabbitmq_up** then click **Execute**:



```
rabbitmq_up
```

Load time: 76ms
Resolution: 14s
Total time series: 3

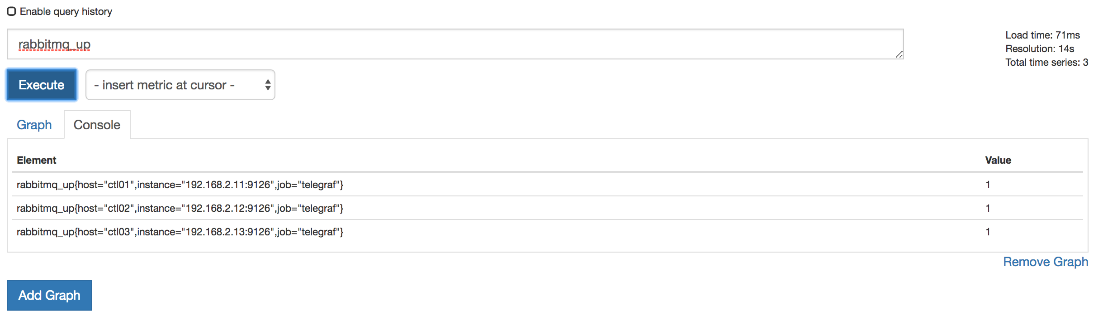
Element	Value
rabbitmq_up{host="ctl01",instance="192.168.2.11:9126",job="telegraf"}	1
rabbitmq_up{host="ctl02",instance="192.168.2.12:9126",job="telegraf"}	1
rabbitmq_up{host="ctl03",instance="192.168.2.13:9126",job="telegraf"}	0

As we have seen in Grafana, RabbitMQ service for **ctl03** node has a value of '0' which is equivalent to being down.

Step 3 Bring up the *rabbitmq-server* service again on **ctl03** node:

```
root@ctl03:~# service rabbitmq-server start
```

Step 4 Execute the **rabbitmq_up** query again on Prometheus UI:



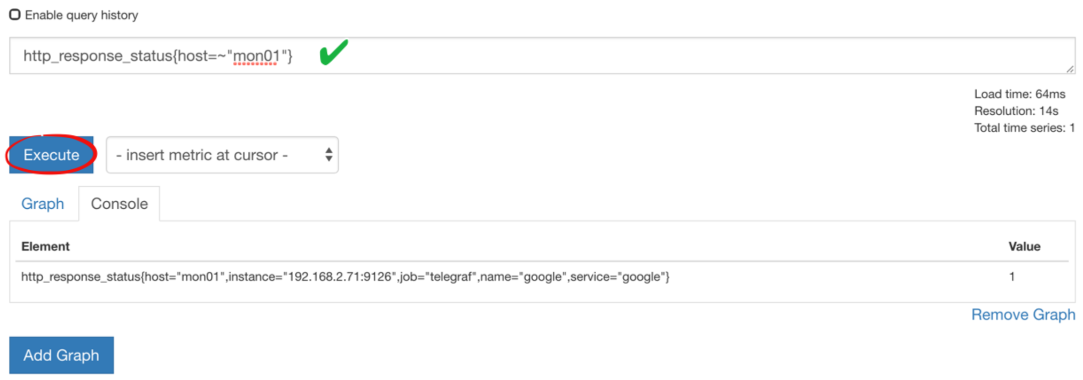
```
rabbitmq_up
```

Load time: 71ms
Resolution: 14s
Total time series: 3

Element	Value
rabbitmq_up{host="ctl01",instance="192.168.2.11:9126",job="telegraf"}	1
rabbitmq_up{host="ctl02",instance="192.168.2.12:9126",job="telegraf"}	1
rabbitmq_up{host="ctl03",instance="192.168.2.13:9126",job="telegraf"}	1

The changes are reflected immediately on the dashboard.

Step 5 Execute the **http_response_status{host=”mon01”}** query to view the HTTP response status from Google we created in earlier sections:



```
http_response_status{host=~"mon01"}
```

Load time: 64ms
Resolution: 14s
Total time series: 1

Element	Value
http_response_status{host="mon01",instance="192.168.2.71:9126",job="telegraf",name="google",service="google"}	1

Step 6 Lastly, you can execute more complex queries such as the following:

```
system_load5{host=~"ctl.*"} < 5 and mem_free{host=~"ctl.*"} > 10000000
```

For more information on queries, refer to the following Prometheus documentation page: https://prometheus.io/docs/prometheus/latest/querying/basics/

Congrautlations, you have finished the *10. Stacklight LMA* chapter!