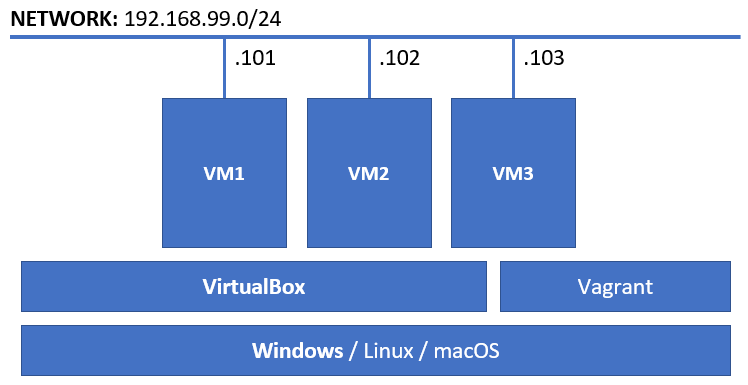
# Practice M6: Prometheus and Grafana

The infrastructure for this practice should look like:



We will create all machines upfront

*Note that we can use either* ***.yml*** *or* ***.yaml****. It is good to stick with one of them. Please keep in mind that sometimes one or the other is the preferred extension and used by default. So, this may lead to the need to adjust the configuration of the particular solution.*

## Part 1: Prometheus 101

Start the infrastructure with

**vagrant up**

### Start with Prometheus

Log on to the **node1** machine

**vagrant ssh node1**

Download **Prometheus** by executing the following

**wget https://github.com/prometheus/prometheus/releases/download/v2.42.0/prometheus-2.42.0.linux-amd64.tar.gz**

Then extract it

**tar xzvf prometheus-2.42.0.linux-amd64.tar.gz**

Enter the folder

**cd prometheus-2.42.0.linux-amd64**

Start **Prometheus** by executing the following

**./prometheus**

In a few seconds the server will be running and ready to serve

Open a browser tab and navigate to **http://192.168.99.101:9090/metrics**

Explore a bit the list of available metrics and their time series

You will notice the different prefixes – **go\_**, **net\_**, **prometheus\_**, **promhttp\_**, etc.

Each one of them applies for a set of time series for multiple metrics covering specific area. For example, the **go** **runtime**

Also, here you can see that all four types of metrics are utilized – **counter**, **gauge**, **histogram,** and **summary**

Check the values, for example, of the **promhttp\_** ones and refresh a few times. They will change

Open additional browser tab and navigate to **http://192.168.99.101:9090**

This should open the integrated web interface

Explore the tabs – **Alerts**, **Graph**, **Status**, **Help** and **Classic UI**

Pay particular attention to the options that are in the **Status** menu

And more specially to the **Command-Line Flags** and **Configuration** items

Once done, return to the terminal session and stop **Ptometheus** by pressing **Ctlr+C**

What will happen with the data? And where it is stored?

Don’t worry, the data will be there

It is stored in a sub-folder of the folder where we extracted **Prometheus**

Use the following command

**tree .**

And take a closer look at the **data** sub-folder hierarchy

In a while, when we start **Prometheus** again, it will continue to accumulate data on top what is already there

When on the command line, we can ask for the list of available options by executing the following

**./prometheus -h**

Explore the options

Check the two options - **--config.file** and **--web.enable-lifecycle**

Explore the file **prometheus.yml** (our current and default configuration, that we saw in the web interface)

Either copy the sample configuration file and modify it or create a new one with the following content

global:

  scrape\_interval: 20s

scrape\_configs:

  - job\_name: 'prometheus'

    static\_configs:

    - targets: ['node1:9090']

Stat it back again but this time with the following command

**./prometheus --config.file prometheus.yml --web.enable-lifecycle 2>> /tmp/prometheus.log &**

It will make it explicitly read the stated config file

At the same time, it will listen for restart signal sent via **HTTP** and will work in background mode

Finally, it will send all messages to a particular log file

Check in both browser tabs that everything is still okay

### Node Exporter

Now, go to the second machine (**node2**)

In another terminal session, execute:

**vagrant ssh node2**

Should we want to monitor machines, we must use node exporter (it works on **Linux**, **macOS**, and **BSD**)

One can be downloaded from here <https://prometheus.io/download/>

More information is available here <https://github.com/prometheus/node_exporter>

And more download artefacts, here <https://github.com/prometheus/node_exporter/releases>

For **Windows** based hosts, we must go here <https://github.com/prometheus-community/windows_exporter>

Download the node exporter by executing the following

**wget https://github.com/prometheus/node\_exporter/releases/download/v1.5.0/node\_exporter-1.5.0.linux-amd64.tar.gz**

Extract it

**tar xzvf node\_exporter-1.5.0.linux-amd64.tar.gz**

Navigate to the folder

**cd node\_exporter-1.5.0.linux-amd64/**

And start it (not in daemon mode)

**./node\_exporter**

Now, open a browser tab and navigate to **http://192.168.99.102:9100/metrics**

Again, we can see plenty of metrics grouped by categories – **go\_**, **node\_**, **process\_**, and **promhttp\_**

Explore a few. For example, **node\_uname\_info**, **node\_os\_info**, **node\_os\_version**, **node\_network\_info**, **node\_filesystem\_avail\_bytes**, **node\_cpu\_seconds\_total**, etc.

Repeat the same procedure on the third machine (**node3**)

In another terminal session, execute:

**vagrant ssh node3**

Download the node exporter by executing the following

**wget https://github.com/prometheus/node\_exporter/releases/download/v1.5.0/node\_exporter-1.5.0.linux-amd64.tar.gz**

Extract it

**tar xzvf node\_exporter-1.5.0.linux-amd64.tar.gz**

Navigate to the folder

**cd node\_exporter-1.5.0.linux-amd64/**

And start it (not in daemon mode)

**./node\_exporter**

Now, open a browser tab and navigate to **http://192.168.99.103:9100/metrics**

Again, we can see plenty of metrics grouped by categories – **go\_**, **node\_**, **process\_**, and **promhttp\_**

Explore a few. For example, **node\_uname\_info**, **node\_os\_info**, **node\_os\_version**, **node\_network\_info**, **node\_filesystem\_avail\_bytes**, **node\_cpu\_seconds\_total**, etc.

Besides the values, the set of metrics is the same (as the exporter and type of OS are the same)

Return on the first machine (**node1**)

Now, let’s extend our configuration to include the two stations

Change the **prometheus.yml** file to match the following

global:

  scrape\_interval: 20s

scrape\_configs:

  - job\_name: 'prometheus'

    static\_configs:

    - targets: ['node1:9090']

  - job\_name: 'debian'

    static\_configs:

    - targets: ['node2:9100']

  - job\_name: 'centos'

    static\_configs:

    - targets: ['node3:9100']

We could have included them as a single job, but we want them this way instead

Before we attempt a restart of **Prometheus**, let’s first check the configuration

**./promtool check config prometheus.yml**

Everything seems to be fine

Before, we reload, return to the web interface to check that there is still the old configuration

We can also check on the **/metrics** endpoint and look for the following metric

**prometheus\_config\_last\_reload\_success\_timestamp\_seconds**

*Note the value (you may convert it here* [*https://www.epochconverter.com/*](https://www.epochconverter.com/)*)*

Now, as we used the **--web.enable-lifecycle flag**, we can restart it by executing this command

**curl -X POST http://192.168.99.101:9090/-/reload**

Check on the **/metrics** endpoint again and look for the following metric

**prometheus\_config\_last\_reload\_success\_timestamp\_seconds**

*Note the value. It is different*

Now, return to the web interface to check that there is the new configuration

Explore the **Status** > **Service Discovery** and **Status** > **Targets** options in the menu

Then go to **Graph** and try with a few of the metrics we explored before

For example, **node\_uname\_info**, **node\_os\_info**, **node\_os\_version**, **node\_cpu\_seconds\_total**, etc.

### Custom Application

#### Start the app on node2

Now, let’s add a container application on the second node (**node2**)

It is with integrated **Prometheus** client library and exposes a set of metrics

First, we can stop the node exporter by pressing **Ctrl+C**

And then starting it back again but in background mode

**./node\_exporter &> /tmp/node-exporter.log &**

We are ready to start the application. Execute

**docker container run -d --name worker1 -p 8081:8080 shekeriev/goprom**

Check that the application is working

**docker container ls**

*If we do not want to use* ***Docker*** *for this, we can start it as a local process*

*Should we want to do this, we can use the files (source or binary ones) in the archive accompanying this document*

Now, open a browser tab and navigate to **http://192.168.99.102:8081/metrics**

There is not much to see here

Try to open the following URL **http://192.168.99.102:8081/**

Now return to the **/metrics** tab and refresh a few times

Aha, here are our application’s metrics

**There are counters, histogram,** and a **gauge**

Let’s start the **runner.sh script** (also part of the lecture’s files) to automate the application’s workload

**/vagrant/goprom/runner.sh http://192.168.99.102:8081**

Return to the **/metrics** tab and refresh a few times

We will start to see some slow requests and perhaps, some requests that led to errors

Stop the runner.sh script with **Ctrl+C** and re-run it as background job

**/vagrant/goprom/runner.sh http://192.168.99.102:8081 &> /tmp/runner8081.log &**

#### Start the app on node3

Repeat (almost) the same procedure on the third node (**node3**)

First, we can stop the node exporter by pressing **Ctrl+C**

And then starting it back again but in background mode

**./node\_exporter &> /tmp/node-exporter.log &**

We are ready to start the application. Execute

**docker container run -d --name worker1 -p 8081:8080 shekeriev/goprom**

Check that the application is working

**docker container ls**

Now, open a browser tab and navigate to **http://192.168.99.103:8081/metrics**

And then open the following URL **http://192.168.99.103:8081/**

Now return to the **/metrics** tab and refresh a few times

Let’s start the **runner.sh script** as background job

**/vagrant/goprom/runner.sh http://192.168.99.103:8081 &> /tmp/runner8081.log &**

Return to the **/metrics** tab and refresh a few times

We will start to see some slow requests and perhaps, some requests that led to errors

#### Change Prometheus configuration

Now, we can include both instances of the application in **Prometheus**

Return on the first machine (**node1**)

Change the configuration to match this

global:

  scrape\_interval: 20s

scrape\_configs:

  - job\_name: 'prometheus'

    static\_configs:

    - targets: ['node1:9090']

  - job\_name: 'debian'

    static\_configs:

    - targets: ['node2:9100']

  - job\_name: 'centos'

    static\_configs:

    - targets: ['node3:9100']

  - job\_name: 'application'

    static\_configs:

    - targets: ['node2:8081']

    - targets: ['node3:8081']

This time, instead of two additional jobs, we add one with two instances

Check the configuration

**./promtool check config prometheus.yml**

And signal **Prometheus** to restart

**curl -X POST http://192.168.99.101:9090/-/reload**

Now, return to the web interface to check that there is the new configuration

Explore the **Status** > **Service Discovery** and **Status** > **Targets** options in the menu

We can see our new job with its two instances

Then go to **Graph** and try with few of the application metrics

For example, check **http\_requests\_total**, **jobs\_processed\_total**, and **jobs\_active**

For each one of them test both the **Table** and **Graph** view

Now, return to the **Status** > **Targets** option in the menu

Both application instances are up

Switch to the terminal that is connected to **node3**

Stop the application by executing this

**docker container rm --force worker1**

Now, return to **Prometheus** UI and check what will happen

The second instance (**node3:8081**) is shown as **DOWN**

Wait for at least 60 seconds and meanwhile check the metrics again

Go to **Graph** and check **http\_requests\_total**, **jobs\_processed\_total**, and **jobs\_active**

You will notice that only data for the first instance (**node2:8081**) is returned

Switch to the terminal that is connected to **node3**

Start the application again by executing this

**docker container run -d --name worker1 -p 8081:8080 shekeriev/goprom**

Return to **Prometheus** UI and check what will happen

The second instance should appear as **UP** and data for its metrics should be shown

### File-based Discovery

Let’s do one more configuration change (for now)

Instead of storing the target statically in the main configuration file, we can use external files and reference them instead

Create an **application.json** file with the following content

[

  {

    "targets": [ "node2:8081" ]

  }

]

Now, adjust the main configuration file (**prometheus.yml**) to match the following

global:

  scrape\_interval: 20s

scrape\_configs:

  - job\_name: 'prometheus'

    static\_configs:

    - targets: ['node1:9090']

  - job\_name: 'debian'

    static\_configs:

    - targets: ['node2:9100']

  - job\_name: 'centos'

    static\_configs:

    - targets: ['node3:9100']

  - job\_name: 'application'

    file\_sd\_configs:

    - files:

      - 'application.json'

This time, instead of hardcoding the application job’s targets, we are reading them from a file

In fact, we are reading just one target (for now)

Check the configuration

**./promtool check config prometheus.yml**

And signal **Prometheus** to reload the configuration

**curl -X POST http://192.168.99.101:9090/-/reload**

Now, return to the web interface to check that there is the new configuration

Explore the **Status** > **Service Discovery** and **Status** > **Targets** options in the menu

We can see our new job with its single instance

Now, return to the terminal and adjust the **application.json** file to include the second instance as well

[

  {

    "targets": [ "node2:8081" ]

  },

  {

    "targets": [ "node3:8081" ]

  }

]

The following is an alternative version of the above

[

  {

    "targets": [ "node2:8081", "node3:8081" ]

  }

]

Now, return to the web interface to check that there is the new configuration

Explore the **Status** > **Service Discovery** and **Status** > **Targets** options in the menu

We should see our application job with its two instances without the need to restart **Prometheus**

## Part 2: Prometheus 102

### Labels and Metrics Manipulation

#### Drop Metrics

Let’s first return to one of the nodes’ metrics

Navigate to **http://192.168.99.102:9100/metrics**

There are plenty of metrics related to the **Go** runtime

We do not want these

Return on the **Prometheus** node

Change the main configuration file to match this

global:

  scrape\_interval: 20s

scrape\_configs:

  - job\_name: 'prometheus'

    static\_configs:

    - targets: ['node1:9090']

  - job\_name: 'debian'

    static\_configs:

    - targets: ['node2:9100']

    metric\_relabel\_configs:

      - source\_labels: [\_\_name\_\_]

        regex: 'go\_.\*'

        action: drop

  - job\_name: 'centos'

    static\_configs:

    - targets: ['node3:9100']

  - job\_name: 'application'

    file\_sd\_configs:

    - files:

      - 'application.json'

Check the configuration

**./promtool check config prometheus.yml**

And signal **Prometheus** to reload the configuration

**curl -X POST http://192.168.99.101:9090/-/reload**

Now, first navigate to **http://192.168.99.102:9100/metrics** to see if they are still there

Yes, they are. The exporter still produces them

Now, return to the web interface to check that there is the new configuration

Explore the **Status** > **Service Discovery** and **Status** > **Targets** options in the menu

No changes here

Switch to **Graph** and look for some of the **go\_** metrics

For example, check **go\_gc\_duration\_seconds**

We can see that no data is returned for **node2**

This is what we expected

If we switch from **Table** to **Graph** view, we will see that there is data but up to a certain point in time

We can change the configuration file to remove **Go** related metrics from all the nodes

global:

  scrape\_interval: 20s

scrape\_configs:

  - job\_name: 'prometheus'

    static\_configs:

    - targets: ['node1:9090']

    metric\_relabel\_configs:

      - source\_labels: [\_\_name\_\_]

        regex: 'go\_.\*'

        action: drop

  - job\_name: 'debian'

    static\_configs:

    - targets: ['node2:9100']

    metric\_relabel\_configs:

      - source\_labels: [\_\_name\_\_]

        regex: 'go\_.\*'

        action: drop

  - job\_name: 'centos'

    static\_configs:

    - targets: ['node3:9100']

    metric\_relabel\_configs:

      - source\_labels: [\_\_name\_\_]

        regex: 'go\_.\*'

        action: drop

  - job\_name: 'application'

    file\_sd\_configs:

    - files:

      - 'application.json'

Then check and apply the changes

**./promtool check config prometheus.yml**

And signal **Prometheus** to reload the configuration

**curl -X POST http://192.168.99.101:9090/-/reload**

Return to the web interface to check that there is the new configuration

Switch to **Graph** and look for some of the **go\_** metrics

For example, check **go\_gc\_duration\_seconds**

Refresh a few times

After a while an empty result set will be returned as **Prometheus** is ignoring this metric for all our nodes

This is what we expected

#### Add Target Labels

We can add target labels as well. Let’s add a few

For example, we will assign a label named **role** to each one of the targets (only for the nodes for now)

Change the configuration to match this

global:

  scrape\_interval: 20s

scrape\_configs:

  - job\_name: 'prometheus'

    static\_configs:

    - targets: ['node1:9090']

      labels:

        role: main

    metric\_relabel\_configs:

      - source\_labels: [\_\_name\_\_]

        regex: 'go\_.\*'

        action: drop

  - job\_name: 'debian'

    static\_configs:

    - targets: ['node2:9100']

      labels:

        role: node

    metric\_relabel\_configs:

      - source\_labels: [\_\_name\_\_]

        regex: 'go\_.\*'

        action: drop

  - job\_name: 'centos'

    static\_configs:

    - targets: ['node3:9100']

      labels:

        role: node

    metric\_relabel\_configs:

      - source\_labels: [\_\_name\_\_]

        regex: 'go\_.\*'

        action: drop

  - job\_name: 'application'

    file\_sd\_configs:

    - files:

      - 'application.json'

Then check and apply the changes

**./promtool check config prometheus.yml**

And signal **Prometheus** to reload the configuration

**curl -X POST http://192.168.99.101:9090/-/reload**

Return to the web interface (**Status** > **Targets**) to check that the changes are reflected

Now, also change the **application.json** file to match this

[

  {

    "targets": [ "node2:8081" ],

      "labels": {

        "role": "container"

      }

  },

  {

    "targets": [ "node3:8081" ],

      "labels": {

        "role": "cont"

    }

  }

]

Yes, it is not a mistake. It is on purpose that the two have different values for the role

Return to the web interface (**Status** > **Targets**) to check that the changes are reflected

All our targets have the new label

Now, we can use it to filter them

Enter **main**, then **node**, and finally **cont** to see what will happen

Obviously, the **node** value doesn’t lead to filtering, as it is seen in other places beside the label value

#### Change (Relabel) Labels

We can change label values for example to consolidate the data

In our case, we will imagine for a moment that we do not have any other option to influence the role label values of the applications beside changing them in the configuration (not the **application.json** file where we declared them)

Add the following block to the end of the **prometheus.yml** file

    relabel\_configs:

      - source\_labels: [role]

        regex: cont

        target\_label: role

        replacement: container

Then check and apply the changes

**./promtool check config prometheus.yml**

And signal **Prometheus** to reload the configuration

**curl -X POST http://192.168.99.101:9090/-/reload**

Return to the web interface (**Status** > **Targets**) to check that the changes are reflected

Refresh a few times

Now, our targets are with aligned (consolidated) labels

#### Remove a Label from a Metric

Return to the UI (**Graph**) and check this metric - **node\_filesystem\_avail\_bytes**

There appears to be a **mountpoint** label

Let’s imagine that we want to remove it

Return on **Prometheus** machine

Open the configuration file (**prometheus.yml**) and add the following **source\_labels** block as second relabel configuration both for **node2** and **node3**

      - source\_labels: [\_\_name\_\_, mountpoint]

        regex: 'node\_filesystem\_avail\_bytes;.\*'

        action: replace

        replacement: ""

        target\_label: mountpoint

Then check and apply the changes

**./promtool check config prometheus.yml**

And signal **Prometheus** to reload the configuration

**curl -X POST http://192.168.99.101:9090/-/reload**

Return to the web interface (**Graph**)

Check again the **node\_filesystem\_avail\_bytes** metric

Refresh a few times and will notice that the list of time series shrinks as the mountpoint label is no longer used

Even more complicated manipulations could be made but they are outside our scope

### PromQL

It is time to do try a few queries and functions

#### Preparation

Go to **node2** and start one more container

**docker container run -d --name worker2 -p 8082:8080 shekeriev/goprom**

And a **runner.sh** script

**/vagrant/goprom/runner.sh http://192.168.99.102:8082 &> /tmp/runner8082.log &**

Go to **node3** and do the same – first the container

**docker container run -d --name worker2 -p 8082:8080 shekeriev/goprom**

And a **runner.sh** script

**/vagrant/goprom/runner.sh http://192.168.99.103:8082 &> /tmp/runner8082.log &**

Now, return to **node1** and change the **application.json** file by adding these two targets

  {

    "targets": [ "node2:8082" ],

      "labels": {

        "role": "secondary"

      }

  },

  {

    "targets": [ "node3:8082" ],

      "labels": {

        "role": "secondary"

      }

  }

Save and close the file

Check in the web UI if the changes are reflected

#### Experimenting

Return to the web interface (**Graph**)

First, did you know that you can click on the button before the Execute button to see a list of all metrics?

No. Okay, now you know it 😉

Click the button to see the full list

##### Gauge

Select the **jobs\_active** metric and click **Execute**

We will see all for sets of data for the four application instances

Should we want to filter them and select particular one, we can add a selector

For example, let’s imagine that we want to see only data for the **secondary** containers

Modify the formula to

**jobs\_active{role="secondary"}**

It works 😉

What if we want to see all data coming from **node3**?

We should apply a simple regular expression

Let’s try this formula

**jobs\_active{instance=~"node3.\*"}**

Okay, this is as of now (the latest generation of data)

Should we want to see data from a past point of time, we can use the offset construction

For example, to see a measurement as 2 minutes ago, we must use the following formula

**jobs\_active{instance=~"node3.\*"} offset 2m**

Of course, as with previous ones, if we refresh a few times, the data will change. Why?

Now, let’s show all again

**jobs\_active**

And try to sum them

**sum(jobs\_active)**

It works. But what if want to see the sum per role for example? This would mean that all other labels are not needed

And we can remove them from the calculation with this formula

**sum without(instance, job) (jobs\_active)**

Wow, we are getting better and better 😉

Alternative version of the above is this one

**sum by(role) (jobs\_active)**

Here we are stating what we are interested in instead of stating what to omit

We can switch to **Graph** to see our data graphically

Of course, we can apply other aggregation functions like

**max without(instance, job) (jobs\_active)**

**min without(instance, job) (jobs\_active)**

**avg without(instance, job) (jobs\_active)**

##### Counter

Now, let’s switch to another metric - **jobs\_processed\_total**

We can see amounts of processed jobs per application instance with all its dimensions

Here, we can ask for example, how the processed jobs changed in the last 5 minutes

Change the formula to

**jobs\_processed\_total [5m]**

The above returns a range vector

We can filter using a selector, for example only for the failed jobs

**jobs\_processed\_total {result="fail"} [5m]**

Now, we could ask for the rate of change (per second increase for the counters)

**rate(jobs\_processed\_total [5m])**

Of course, this one can be filtered also

**rate(jobs\_processed\_total {result="fail"} [5m])**

We can apply aggregation

For example, to see the average change rate in the last 5 minutes

**avg(rate(jobs\_processed\_total [5m]))**

Or summarize it

**sum(rate(jobs\_processed\_total [5m]))**

We can apply selectors within the aggregation function. For example, see the max rate for filed jobs

**max(rate(jobs\_processed\_total {result="fail"} [5m]))**

Or remove labels from the equation

**sum without(instance, result) (rate(jobs\_processed\_total [5m]))**

Or even remove and filter labels

**sum without(instance, result) (rate(jobs\_processed\_total {result="fail"} [5m]))**

##### Complex Expressions

Select the **http\_requests\_total** metric

Now, we can summarize it by code

**sum without (instance, job, method, role) (http\_requests\_total)**

Or

**sum by (code) (http\_requests\_total)**

Obviously, the second one is shorter compared to the first one

And we can see amount of **404** compared to **200**

**sum without (instance, job, method, role) (http\_requests\_total {code="404"}) / sum without (instance, job, method, role) (http\_requests\_total {code="200"})**

Hm, empty result set. Why? *Perhaps, it has something to do with vector operations*

Let’s correct the expression to match this

**sum without (instance, job, method, role, code) (http\_requests\_total{code="404"}) / sum without (instance, job, method, role, code) (http\_requests\_total{code="200"})**

Or go with a simplified version (as we are removing all labels):

**sum (http\_requests\_total {code="404"}) / sum (http\_requests\_total {code="200"})**

No matter which one we select, the effect is that we ensured that both vectors match

We can change the formula to see for example, what part of all codes are the **200** or **404**

**sum without (instance, job, method, role, code) (http\_requests\_total {code="200"}) / sum without (instance, job, method, role, code) (http\_requests\_total)**

**sum without (instance, job, method, role, code) (http\_requests\_total {code="404"}) / sum without (instance, job, method, role, code) (http\_requests\_total)**

##### Histograms

Now, let’s try how histograms work

Check what our **http\_request\_duration\_seconds\_bucket** metric will return

It is a complex metric compared to the previous ones

Let’s filter is for one instance

**http\_request\_duration\_seconds\_bucket {instance="node2:8081"}**

We can switch to **Graph** to see how each bucket develops over time

Return back to **Table**

We can see the change rate for the last 5 minutes

**rate(http\_request\_duration\_seconds\_bucket {instance="node2:8081"} [5m])**

Okay, let’s include one other instance as well

**rate(http\_request\_duration\_seconds\_bucket {instance=~"node2:.\*"} [5m])**

We can aggregate this on a higher level

**sum without (code, handler, instance, job, method) (rate(http\_request\_duration\_seconds\_bucket {instance=~"node2.\*"} [5m]))**

To see up to how many seconds we need to deal with 90% of the requests over the last 5 minutes, we can execute this

**histogram\_quantile(0.90, sum without (code, handler, instance, job, method) (rate(http\_request\_duration\_seconds\_bucket {instance=~"node2.\*"} [5m])))**

And then switch to **Graph**

### Alert Manager

Now, let’s deploy **Alert Manager** and see it in (simple) action

#### Installation

First, we must install it

Navigate to this URL <https://prometheus.io/download/#alertmanager>

Go to the **Prometheus** server

Execute the following to download it

**wget https://github.com/prometheus/alertmanager/releases/download/v0.25.0/alertmanager-0.25.0.linux-amd64.tar.gz**

Now, extract it

**tar xzvf alertmanager-0.25.0.linux-amd64.tar.gz**

Enter the folder

**cd alertmanager-0.25.0.linux-amd64/**

And start it in background mode

**./alertmanager &> /tmp/am.log &**

It will start with the default configuration (**alertmanager.yml**) and will listen for connections on port **9093**

#### Configuration

The default configuration of **Alert Manager** is just fine for our scenario

We should however configure our **Prometheus** instance

Open the main configuration file (**prometheus.yml**) and add the following block between the **global** and **scrape\_configs** sections

alerting:

  alertmanagers:

    - static\_configs:

      - targets:

        - node1:9093

Then check and apply the changes

**./promtool check config prometheus.yml**

And signal **Prometheus** to restart

**curl -X POST http://192.168.99.101:9090/-/reload**

Go to the web interface and check the information in **Status** > **Configuration**

#### Alert Rule

Now, let’s assume that we want to emit alerts if any of our targets is down

We can use the up metric for this

Go to the **Prometheus** server

And create a **rules.yml** file with the following content

groups:

- name: MonitorAllInstances

  rules:

  - alert: InstanceIsDown

    expr: up == 0

    for: 1m

    annotations:

      title: 'Instance {{ $labels.instance }} is down'

      description: '{{ $labels.instance }} of job {{ $labels.job }} has been down for more than 1 minute.'

    labels:

      severity: 'critical'

Now, let’s include the **rules.yml** file in the main configuration file

Add the following between the **global** and **altering** sections

rule\_files:

  - rules.yml

Then check and apply the changes

**./promtool check config prometheus.yml**

And signal **Prometheus** to restart

**curl -X POST http://192.168.99.101:9090/-/reload**

Go to the web interface and check the information in **Status** > **Configuration**

And then return to the **Alerts** section and explore there

#### Test the Alert

Now, go to **node2** and stop and remove one of the containers (for example, **worker2**)

**docker container rm --force worker2**

Go to **node3** and stop both

**docker container rm --force worker1 worker2**

Return to the UI (**Status** > **Targets**) and refresh

Go to **Alerts**

Our alert is in **pending** state

After a while it will transition to **firing** state

You can check the **Alert Manager UI** as well at **http://192.168.99.101:9093/**

Go to **node2** and start **worker2**

**docker container run -d --name worker2 -p 8082:8080 shekeriev/goprom**

Go to **node3** and start **worker1** and **worker2**

**docker container run -d --name worker1 -p 8081:8080 shekeriev/goprom**

**docker container run -d --name worker2 -p 8082:8080 shekeriev/goprom**

Okay, our alerting is working but doesn’t notify anyone (yet)

#### Link to Slack

Log in to **Slack** and go to your workspace

Go to **Settings & administration** > **Manage Apps**

Enter **hooks** in the **Search App Directory** field

Select the **Incoming WebHooks**

Once in the page of the app, click **Add to Slack**

Then either select the channel to be used, or click the create a new channel link

Let’s create a new one

Enter **prometheus** for **Name**

And **Prometheus Alerts** for **Description**

Click the **Create** button

Click the **Skip for now** button to avoid adding people now

Now, click the **Add Incoming WebHooks integration** button

Then copy the **Webhook URL**

Return to the **Prometheus** server

Navigate to the folder for **Alert Manager**

Open the **alertmanager.yml** file and make sure that it looks like this

global:

  resolve\_timeout: 1m

  slack\_api\_url: 'https://hooks.slack.com/services/<hook-id>'

route:

  receiver: 'slack-notifications'

receivers:

- name: 'slack-notifications'

  slack\_configs:

  - channel: '#prometheus'

    send\_resolved: true

Of course, you should adapt it (**url** and **channel**) to your values

Then check the configuration with

**./amtool check-config alertmanager.yml**

And restart it by executing this

**curl -X POST http://node1:9093/-/reload**

Go to **node3** and stop both containers

**docker container rm --force worker1 worker2**

Return to **Prometheus** UI and refresh and wait a bit

Do not forget to check the **Alert Manager UI** as well at **http://192.168.99.101:9093/**

After a while a notification will appear in the **Slack** application

Is not the best looking one, but it works

Of course, this can be improved by using templating

Go to **node3** and start both containers

**docker container run -d --name worker1 -p 8081:8080 shekeriev/goprom**

**docker container run -d --name worker2 -p 8082:8080 shekeriev/goprom**

After a while it will be detected that the issue has been resolved and we will receive a message

## Part 3: Grafana 101

### Run Grafana

There are multiple options as we know, but we will go with the **Docker** way

Go to the **Prometheus** node first

Then, create a volume for **Grafana**

**docker volume create grafana**

And finally, start **Grafana**

**docker run -d -p 3000:3000 --name grafana --rm -v grafana:/var/lib/grafana grafana/grafana-oss**

After a while the instance will be ready

Open a browser tab and navigate to <http://192.168.99.101:3000/login>

The default credentials are **admin** / **admin**

We will be asked if we want to change the password

Why not? Let’s do it. Set a password you like and hit **Submit**

Alternatively, you can always click **Skip** and go with the default credentials

### Explore the UI

As we just started the instance, there is not much to do here

By default, we are presented with the **Home** view (a special “dashboard”)

On the left, there is the toolbar

Here, we can do the following (from top to bottom):

* Go to the **Home** dashboard
* Search for dashboards
* Create new artefact – dashboard, folder, import a dashboard, or create an alert rule
* Browse dashboards, playlists, snapshots, or panels
* Explore the data sources
* Manage the alerting
* Instance configuration
* Server configuration
* Profile settings
* Help

As we do not have any data sources, there is not much we can do

### Connect to Prometheus

Let’s go to **Configuration** > **Data sources**

Click the **Add data source** button

Select **Prometheus**

Leave the name as it is (or change it if you like)

Set the **URL** to **http://192.168.99.101:9090/**

Hit the **Save & test** button

Now, we have our data source

### Explore the Data Source

Click the **Explore** option in the left menu

Switch to the data source we registered earlier (for example, **Prometheus**)

Click the **Metrics browser** link (on the left of the text box)

We can see many familiar metrics here

Select **jobs\_processed\_total**

Modify the **Label filters** section to filter the **instance** label and select the **node2:8081** and **node2:8082** values

We can see the query that was built

Click the **Run query** button (top-right)

And bam, we have a chart 😊

We can experiment with the graph types

We can then click **Inspector** to explore what is happening

Then, we can click the **Query history** button to explore this as well

Return to the **Home** screen

### Install Dashboards

Navigate to **Configuration** > **Data sources**

Select our data source

Then, switch to **Dashboards** view (top of the screen)

Click the **Import** button next to the **Prometheus 2.0 Stats**

Then click on **Dashboards** > **Browse** in the left menu

Select the imported dashboard

It will load

There are plenty of items here

Click on the title of one of them. For example, the **Scrape Duration** chart and select **View**

This will open the chart in full size

Explore the data

You can click on the title again and select, for example **Explore**

Here, you can see the query as well

If you enable the **Explain** mode you will see even more details

Click the title again and select **Share**

Select **Link** and click **Copy**

Then open another browser window or tab and paste the link

You will see the panel as it was at the time of the share

*Please note, that if you pasted the link in another browser, you may be required to log in again*

Return to the initial window

### Create Library Items

While still in the **share window** (if closed, reopen it) of the **Scrape Duration** panel, switch to **Library panel**

There leave the proposed name and folder and click **Create library panel**

Click the back arrow to return to the dashboard

Click on the title of another panel, for example the **Head Chunks** one

There, in the menu, select **Share**

Switch to **Library** **panel** and click the **Create library panel** button

Now, we have two shared panel which could be included in dashboards and playlists

### Create Own Dashboard #1

Let’s create one simple dashboard

Click **Dashboards** > **New dashboard** (in the left menu)

*If you are asked to save the changes, then click the* ***Discard*** *button*

Click the **Add a panel from the panel library** link

Select **Scrape Duration**

It will appear on the new dashboard

Resize it to fill the whole area

Click the **Save dashboard** button (top right)

Enter **Dash: Scrape Duration** in the **Dashboard name** field

Click **Save**

Let’s create another one

Click **Dashboards** > **New dashboard** (in the left menu)

If you are asked to save the changes, then click the **Discard** button

Click the **Add a panel from the panel library** link

Select **Head Chunks**

It will appear on the new dashboard

Resize it to fill the whole area

Click the **Save dashboard** button (top right)

Enter **Dash: Head Chunks** in the **Dashboard name** field

Click **Save**

### Create Playlist

Click on **Dashboards** > **Playlists** item in the left menu

Click **Create Playlist**

Enter **Prometheus Stats** in the **Name** field

Change the value of the **Interval** field to **1m** or something else like 15s (for 15 seconds interval)

In the **Add dashboards** section

Open the **Add by title** drop-down list

Select **General/Dash: Scrape Duration** item

Open the **Add by title** drop-down list again

Select **General/Dash: Head Chunks** item

Now the two dashboards are part of the playlist

Click the **Save** button

Once back in the **Playlists** tab, click the **Start playlist** button to see our new playlist in action

A new dialog window will appear

Click the **Autofit** option and then click on the **Start Prometheus Stats** button

Sit back and relax while watching our playlist running

Wait a while and you will see the switch between the two dashboards

Once if have seen enough, click the **Dashboards** > **Home** item in the left menu

### Create Own Dashboard #2

Go to **Dashboards** > **New dashboard** in the left menu

Then click **Add a new panel**

Switch the visualization from **Timeseries** to **Stat** (top right corner)

Click the **Metrics browser**

Select **jobs\_active** item

Click **Run queries**

A visualization will appear

Click the **Save** button (top right corner)

Enter **Active Jobs** in the **Dashboard name** field

Click the **Save** button

Now, click the **Dashboard settings** button (top right corner)

Click on **Variables**

Then click the **Add variable** button

Enter **instancename** in the **Name** field

Enter **label\_values(jobs\_active, instance)** in **Query**

Select the **Include All option**

A preview of the values will appear at the bottom

Click the **Apply** button

Click the **Save dashboard** button

Enter **A variable added** in the comment box of the **Save dashboard** dialog and click **Save**

Now, click the **Back** arrow (top left corner) to return to the dashboard

You will see the variable as a drop-down list just above the panel

Select the **All** option in it

Click the **Panel Title** and select **Edit**

Now, in the right panel, go to the **Panel options** section

Change the **Title** to **Active Jobs**

Scroll down to the **Repeat options** section

In the dropdown list select **instancename** (our variable)

Change the **Max per row** value to **2**

Now, change the expression (in **Code** view) to **jobs\_active{instance="$instancename"}**

Change the time period to Last 15 minutes (top part of the screen)

Click **Apply** (top right corner)

Change the value in the drop-down **instancename** list to one of the instances and then to **All**

Four visualizations will appear

Click the **Save dashboard** button

Select the **Save current time range as dashboard default** option and click **Save**

Go to dashboard settings

Go to **Variables**

Select our variable

Select **Nothing** in the **Show on dashboard** section

Click **Apply**

Click **Save dashboard**

Click again **Save**

Click the **Back** arrow

Now, everything seems to be just fine

We’ve made it 😊

Click on the Grafana icon (top left) to return to the **Home** page

Select **Dashboards** > **Browse**

Click on our dashboard

It opens and looks perfect

Now, we can add it to a playlist (the existing one or new one) if we want