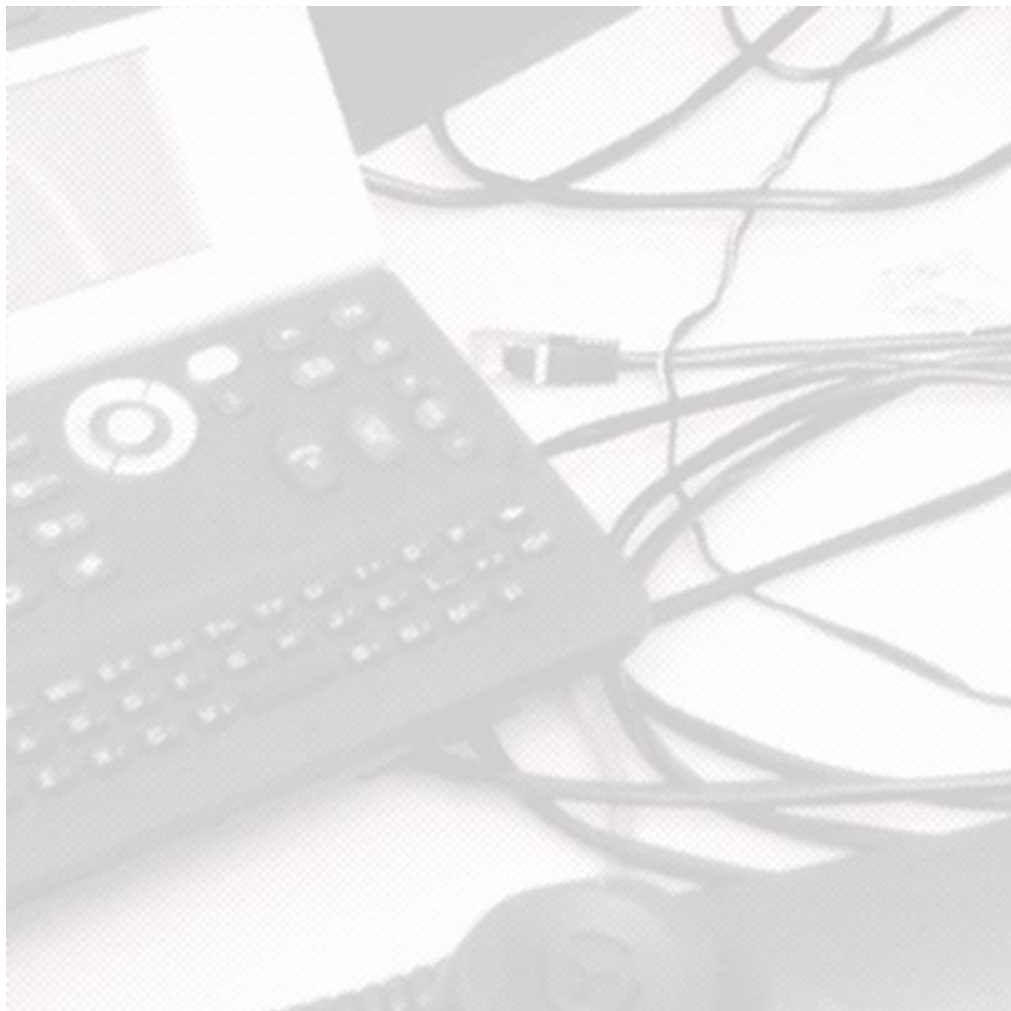


ZSUT

Zakład Sieci i Usług Teleinformatycznych

Kubernetes laboratory



Lab part 2: Kubernetes monitoring with Prometheus and Grafana

Prepared by: dr inż. Dariusz Bursztynowski

Acknowledgements: The author is grateful to the following students for their help in preparing the lab (in alphabetical order): Hubert Daniłowicz, Franciszek Dec, Jerzy Jastrzębiec-Jankowski, Maciej Maliszewski, Miłosz Marchewka, Adrian Osędowski, Cezary Osuchowski, Piotr Polnau, Jan Sosulski, Filip Wrześniewski, Marta Zielińska

ZSUT. Zakład Sieci i Usług Teleinformatycznych
Instytut Telekomunikacji
Wydział Elektroniki i Technik Informacyjnych
Politechnika Warszawska

Last update: May 2023

WARNING: Before powering the devices make sure you are using the right power supply. The power supplies in your set have the same type of DC plug but they **DIFFER SIGNIFICANTLY in the output voltage** (Linksys device powers from 12V DC while TP-Link switch needs 53V DC). Powering Linksys device from power supply of TP-Link switch results in instantaneously damaging the Linksys.

Table of contents

1. Installing Prometheus stack.....	3
1.1. Installation	3
1.2. Main components of the stack.....	5
1.2.1. Prometheus operator	5
1.2.2. Selected service monitors.....	5
1.2.2.1. Node-exporter service monitor.....	6
1.2.2.2. Kube State Metrics.....	6
1.2.2.3. Kubelet service monitor.....	6
1.3. Prometheus server (the main application)	6
1.3.1. Final configurations of Prometheus server.....	6
1.3.1.1. Enabling access to all required resources	6
1.3.1.2. Assigning persistent storage.....	7
1.3.1.3. Exposing Prometheus dashboard as LoadBalancer service.....	9
1.3.2. Deploying the updates to a running Prometheus server	10
1.4. Optional: Monitoring extensions for our cluster	14
1.4.1. Traefik service monitor	14
2. Grafana.....	16
2.1. Manifest preparation.....	16
2.1.1. Downloading the manifests.....	16
2.1.2. Deploying Grafana, checking main components and using the first dashboard	18
2.2. Checking Grafana – first graphs.....	19
3. Useful readings and videos.....	20

1. Installing Prometheus stack

Prometheus is the collector of metrics popular in Kubernetes deployments to monitor clusters including server infrastructure and the services are run in Kubernetes clusters. It uses so-called service monitors that provide information Prometheus can scrape and present to analytical applications. Prometheus will use persistent storage, and we will specify for how long it will keep the data. You can have more than one instance of Prometheus, but to save resources in our case we will use one for everything we want to monitor.

Our Prometheus instance will be controlled by Prometheus Operator – a dedicated Kubernetes controller run in the cluster responsible for the deployment and life cycle management of all components related to the Prometheus instance. After Prometheus components are installed, Prometheus Operator will monitor the Kubernetes API server for changes to Prometheus-related objects and ensure that the current Prometheus deployments (there can be multiple Prometheus deployments in one cluster) match these objects.

In principle, we will install Prometheus stack according to kube-prometheus project following the instructions available here: <https://github.com/prometheus-operator/kube-prometheus>. This project provides example configurations for a complete cluster monitoring stack based on Prometheus and the Prometheus Operator. This includes deployment of multiple Prometheus and Alertmanager instances, metrics exporters such as the node_exporter for gathering node metrics, scrape target configuration linking Prometheus to various metrics endpoints, and example alerting rules for notification of potential issues in the cluster. It also provides Grafana and Grafana dashboards. For more info in kube-prometheus, please refer to their web page.

One can save a little bit of resources deploying only those modules of the full kube-prometheus stack that will be mandatory for her/his use case (e.g., in our case we will not need Alertmanager, PrometheusRule, PrometheusAdapter and BlackboxExporter). Those missing CRDs can be included at a later time if needed. However, to make our work easier we'll deploy the full stack leaving experimenting with possible adjustments to your own decision.

In the remainder of this section, we first run a pure installation procedure of the whole stack (subsection 1.1) and then describe selected components of the platform and fine-tune a couple of settings to according to our needs (subsection 1.2).

1.1. Installation

As explained above, we will kube-prometheus project to install Prometheus stack in our cluster:
<https://github.com/prometheus-operator/kube-prometheus>.

Note 1: If you happen to check in more detail the manifests of kube-prometheus you will often see RBAC (role based control) declaration for various platform components. More on RBAC in Kubernetes can be found, e.g., in [this document](#). Also worth noting is the following [link here](#).

Note 2: If you are working with a fresh k3s cluster then by default it contains the right settings to accommodate kube-prometheus installation (or parts of it). In case of doubts, check the **Prerequisites** for installing kube-prometheus as described [here](#); run the command `kubeadm config print init-defaults --component-configs KubeletConfiguration` and inspect it for actual values of `authentication.webhook.enabled` and `authorization.mode` (defaults are listed [here](#)).

Note 3: As mentioned in our *Kubernetes Lab Part 1* guide, the compatibility matrix <https://github.com/prometheus-operator/kube-prometheus#compatibility> between Kubernetes and kube-prometheus limits the use of kubernetes up to version to 1.25. That is why we recommended for installation in Part 1.

We install kube-prometheus according to the [quickstart](#) procedure. To this end follow the steps listed below:

1. To store only the right stuff and keep it in order it is recommended to create a directory named `monitoring` on the management host where we'll hold all data related to the monitoring part of our lab. You can create it in a directory of your choice. Then `git clone kube-prometheus` to this directory, copy relevant manifests to it, and then delete the whole (already unnecessary) kube-prometheus clone¹ (below, commands to be executed are in **bold**).

```
xubuntu@xubulab:~/k3s-taskforce/manifests$ mkdir monitoring
xubuntu@xubulab:~/k3s-taskforce/manifests$ cd monitoring

# clone complete kube-prometheus, copy relevant manifests and delete unnecessary stuff
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ git clone https://github.com/prometheus-
operator/kube-prometheus.git
Cloning into 'kube-prometheus'...
remote: Enumerating objects: 17669, done.
remote: Counting objects: 100% (218/218), done.
remote: Compressing objects: 100% (114/114), done.
remote: Total 17669 (delta 150), reused 140 (delta 94), pack-reused 17451
Receiving objects: 100% (17669/17669), 9.28 MiB | 25.48 MiB/s, done.
Resolving deltas: 100% (11652/11652), done.
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cp -r kube-prometheus/manifests/* ./
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ sudo rm -r kube-prometheus
```

2. Execute the installation steps from [here](#) (below, commands to be executed are in **bold**):

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cd ..
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply --server-side -f monitoring/setup
customresourcedefinition.apiextensions.k8s.io/alertmanagerconfigs.monitoring.coreos.com
serverside-applied
...
namespace/monitoring serverside-applied
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl wait \
> --for condition=Established \
> --all CustomResourceDefinition \
> --namespace=monitoring
customresourcedefinition.apiextensions.k8s.io/addons.k3s.cattle.io condition met
customresourcedefinition.apiextensions.k8s.io/helmcharts.helm.cattle.io condition met
...
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply -f monitoring/
alertmanager.monitoring.coreos.com/main created
networkpolicy.networking.k8s.io/alertmanager-main created
...
# check all pods from monitoring namespace are running
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get pods -n monitoring
```

NAME	READY	STATUS	RESTARTS	AGE
node-exporter-5h486	2/2	Running	0	2m41s
node-exporter-9xzpt	2/2	Running	0	2m41s
node-exporter-hn7df	2/2	Running	0	2m41s
kube-state-metrics-787856b689-rqx2n	3/3	Running	0	2m42s
prometheus-operator-65ff8b668d-712f4	2/2	Running	0	2m39s
blackbox-exporter-6fd586b445-lwkkx	3/3	Running	0	2m44s
node-exporter-9z7qr	2/2	Running	0	2m41s
alertmanager-main-2	2/2	Running	0	2m28s

¹ Methods as `wget -r` that are often used for bulk file downloads from webpages do not work well for github.

alertmanager-main-0	2/2	Running	0	2m28s
prometheus-adapter-757f9b4cf9-92jlk	1/1	Running	0	2m39s
alertmanager-main-1	2/2	Running	1 (2m ago)	2m28s
prometheus-adapter-757f9b4cf9-x9zc7	1/1	Running	0	2m39s
prometheus-k8s-0	2/2	Running	0	2m24s
prometheus-k8s-1	2/2	Running	0	2m24s
grafana-58bbb48f66-lspdk	1/1	Running	0	2m42s
xubuntu@xubulab:~/k3s-taskforce/manifests\$				

1.2. Main components of the stack

In this subsection, selected components from the monitoring stack installed in the previous section are briefly described and slightly configured to meet our needs.

1.2.1. Prometheus operator

Beginner's guide to using Prometheus Operator: <https://blog.container-solutions.com/prometheus-operator-beginners-guide>. Prometheus operator is a kind of Kubernetes controller that manages the lifecycle of Prometheus application (all the microservices Prometheus is composed of) using the models of all needed components. Those models are provided in the form of custom resource descriptors (CRD). We will create all the CRDs that define the Prometheus, Alertmanager, and ServiceMonitor abstractions used to configure the monitoring stack, as well as the Prometheus Operator controller and Service.

1.2.2. Selected service monitors

General architecture of Prometheus in Kubernetes including the role of service monitor is described, e.g., at <https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/user-guides/getting-started.md>. Prometheus operator uses ServiceMonitors to define a set of targets to be monitored by Prometheus. It uses `matchLabel` selectors in ServiceMonitor definition to define which Services to monitor, the namespaces to look for, and the port on which the metrics are exposed. ServiceMonitor CRD are used by the operator to configure Prometheus server to scrap desired services (those indicated in ServiceMonitor). More explanations regarding Prometheus operator and the role of service monitor custom resource can be found in several other places, e.g., in <https://blog.container-solutions.com/prometheus-operator-beginners-guide>.

The kube-prometheus project we use here provides service monitors for all control plane components of Kubernetes platform, i.e. (see the link above and the one given in the intro to section 1):

- `kubernetesControlPlane-serviceMonitorApiserver`
- `kubernetesControlPlane-serviceMonitorCoreDNS`
- `kubernetesControlPlane-serviceMonitorKubeControllerManager`
- `kubernetesControlPlane-serviceMonitorKubeScheduler`
- `kubernetesControlPlane-serviceMonitorKubelet`

Actually, all components of the kube-prometheus stack have their service monitor that is scraped by Prometheus (so even more monitors that those listed above are present). In the following, we describe only a couple of selected monitors and for more information the reader is referred to the original documentation of the kube-prometheus project.

1.2.2.1. Node-exporter service monitor

Node-exporter is a service monitor for Prometheus that is designed to provide only hardware and OS metrics exposed by the cluster nodes running Linux kernel. It is deployed as a Kubernetes Daemon set. Other metrics as service level metrics for microservices or for Kubernetes level components/resources (API server, pods, deployments, etc.) have to be gathered by Prometheus using other exporters than node-exporter.

Note: node-exporter deployed in our stack is similar to this <https://github.com/carlosedp/cluster-monitoring/blob/master/manifests/node-exporter-daemonset.yaml> (link daemonset only, the rest is straightforward to locate). There are small differences between them (e.g., for the daemonset there's updateStrategy included in "our" version contrary to the linked one, different container repos are referenced, etc.), but the overall shape is the same including additional flags setup for docker container according to this https://github.com/prometheus/node_exporter#docker. We will use the RedHat repo.

1.2.2.2. Kube State Metrics

According to the original documentation, *kube-state-metrics (KSM) is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects. It is not focused on the health of the individual Kubernetes components, but rather on the health of the various objects inside, such as deployments, nodes and pods. kube-state-metrics is about generating metrics from Kubernetes API objects **without modification**. This ensures that features provided by kube-state-metrics have the same grade of stability as the Kubernetes API objects themselves. In turn, this means that kube-state-metrics in certain situations may not show the exact same values as kubectl, as kubectl applies certain heuristics to display comprehensible messages. kube-state-metrics **exposes raw data unmodified from the Kubernetes API**, this way users have all the data they require and perform heuristics as they see fit.*

1.2.2.3. Kubelet service monitor

Kubelet is an important part of Kubernetes control plane, and it also exposes Prometheus metrics by default on port 10255. Actually, kube-state-metrics available in kube-prometheus (see above) collects lots of data, and some of them overlap with Kubelet provided metrics, nevertheless some information can be collected only from Kubelet.

1.3. Prometheus server (the main application)

Prometheus is the core application of the stack responsible for scraping, storing and providing access to the monitoring information (see, e.g., [here](#) for a short intro to Prometheus server).

1.3.1. Final configurations of Prometheus server

This subsection describes how to adjust Prometheus CRD to: 1) enable Prometheus access to all required resources, 2) assign persistent storage to Prometheus server, and 3) enable external access to Prometheus dashboard using service type LoadBalancer. Updates to kube-prometheus manifests stored according to description in section 1.1 will be needed.

1.3.1.1. Enabling access to all required resources

Update the manifest file `prometheus-clusterRole.yaml` according to [this](#). This cluster role defines which apiGroups of Kubernetes API can be accessed by the Prometheus server in our cluster. Take care to preserve

the values of all labels in `prometheus-clusterRole.yaml` as in the original version of this file. A short intro to Kubernetes RBAC API can be found, e.g., [here](#).

```
$ nano prometheus-clusterRole.yaml
# edit your file/save; don't blindly copy-paste everything from here
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0 # keep version the same as in the original from kube-prometheus
  name: prometheus-k8s
# the following should be filled in according to
# https://github.com/prometheus-operator/prometheus-
operator/blob/main/Documentation/rbac.md#prometheus-rbac
rules:
- apiGroups: [""]
  resources:
    - nodes
    - nodes/metrics
    - services
    - endpoints
    - pods
  verbs: ["get", "list", "watch"]
- apiGroups: [""]
  resources:
    - configmaps
  verbs: ["get"]
- apiGroups:
  - networking.k8s.io
  resources:
    - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
```

1.3.1.2. Assigning persistent storage

Assigning persistent storage to Prometheus server will need updating kind *Prometheus* to configure affinity rules, replica number, and PVC (persistent volume claim) for Prometheus server storage.

Persistent storage is needed to preserve historical monitoring data even across system shutdowns. One would want to declare PVC (Persistent Volume Claim) for Prometheus server directly following Rancher recommendations: <https://github.com/rancher/local-path-provisioner#usage>. However, the guidelines for using *Prometheus operator* provided [here](#) recommend embedding the PVC in Prometheus CRD. Additionally, in case of using *kube-prometheus* (our case) the recommended form of such a PVC specification is given [here](#) (we do not need a long retention time). We will stick to the guidelines outlined in both of those documents.

- **Persistent volume claim explanation:** When deploying an application that needs to retain its data (in case of node/cluster downtimes), you'll need to create persistent storage. Persistent storage allows you to store application data external from the pod running your application and maintain application data even if the application's pod fails. A **persistent volume (PV)** is a piece of storage in the Kubernetes cluster located, while a **persistent volume claim (PVC)** is a request for storage. For details on how PVs and PVCs work and interrelate, refer to the official Kubernetes documentation on storage. Below we use the simplest way of setting up *dynamic* persistent storage with a local-storage provider for Prometheus. More advanced methods using explicit definition of persistent volume claim and persistent volume (also *static*) can be found in:

- <https://docs.k3s.io/storage>

- Two recommended interesting discussions:
 - <https://blog.differentpla.net/blog/2021/12/17/k3s-dynamic-pv/>
 - <https://blog.differentpla.net/blog/2021/12/17/k3s-static-pv/>
- <https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/user-guides/storage.md>
- **Affinity rules explanation.** We want Prometheus not to run on control (master) node(s) to limit the load put onto the control plane in the cluster. This can be achieved in many ways. We already limited the access to the control node(s) for non-critical workloads by setting appropriate flag in k3s install script that resulted in setting taint `CriticalAddonsOnly=true:NoExecute` on the master node. We saw this flag in operation when we discussed why MetalLB speakers run only on worker nodes. Basically, this taint should be sufficient to exclude the master from running Prometheus. If this taint were not set then alternative options could be used to achieve the same effect and they are shown/commented in the frame given below. Use your preferred option, carefully commenting/uncommenting respective parts of the manifest if needed. A quick guide on nodeSelectors and affinity rules can be found [here](#).
- **Assigning the storage:** Open the file `prometheus-prometheus.yaml` and change its contents according to the frame presented below. Notice we set `replicas=1` (originally there is 2 replicas, but one is sufficient for us and it also saves resources in our cluster). Watch blank indents when copy-pasting.

```
# file prometheus-prometheus.yaml
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0      # keep version the same as in the original file
  name: k8s
  namespace: monitoring
spec:
  alerting:
    alertmanagers:
      - apiVersion: v2
        name: alertmanager-main
        namespace: monitoring
        port: web
  enableFeatures: []
  externalLabels: {}
  image: quay.io/prometheus/prometheus:v2.42.0
  # We would like Prometheus to run on worker nodes only, not on the control node. In the following, we
  # describe two options that could be used if we had not set CriticalAddonsOnly=true:NoExecute during
  # cluster installation.
  # Originally, only node selector "kubernetes.io/os: linux" is set for Prometheus (see 3 lines below)
  # and to force Prometheus run on workers one could add "node-role.kubernetes.io/worker: true" as shown
  # in the commented line that follows "kubernetes.io/os: linux"
  nodeSelector:
    kubernetes.io/os: linux
  #   node-role.kubernetes.io/worker: true
  # As mentioned before, we did set taint CriticalAddonsOnly for the master so effectively we already
  # disabled Prometheus from running on the master and basically we achieve our goal with the default
  # nodeSelector settings.
  #
  # Alternatively, we can use the affinity mechanism instead of nodeSelector.
  # Affinity mechanism in general is more expressive than nodeSelector and offers more flexibility in
  # defining placement constraints for Kubernetes workloads; to get acquainted with this option. In the
  # following we configure affinity rules for Prometheus that give the same effect as the ones based on
  # nodeSelector discussed above.
  # affinity:
  #   nodeAffinity:
```



```

#   requiredDuringSchedulingIgnoredDuringExecution:
#   nodeSelectorTerms:
#     - matchExpressions:
#       - key: kubernetes.io/os
#         operator: In
#         values:
#           - linux
#       - key: node-role.kubernetes.io/worker
#         operator: Equal
#         value:
#           - true
podMetadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0 # keep version the same as in the original file
podMonitorNamespaceSelector: {}
podMonitorSelector: {}
probeNamespaceSelector: {}
probeSelector: {}
# one replica and retention to 3 days is sufficient for us
replicas: 1 # one replica will be sufficient in our case
retention: 3d # we probably do not need longer retention time
resources:
  requests:
    memory: 400Mi
ruleNamespaceSelector: {}
ruleSelector: {}
securityContext:
  fsGroup: 2000
  runAsNonRoot: true
  runAsUser: 1000
serviceAccountName: prometheus-k8s
# (blank braces {} mean "all") All service monitors from all namespaces can be scraped by Prometheus
serviceMonitorNamespaceSelector: {}
serviceMonitorSelector: {}
version: 2.42.0
# SPIW addition: define storage which suits our basic needs and allows to control the amount of disk
# space used; # unfortunately, at the time of this writing (Oct. 2022) the declaration of 2Gi sharp
# will be ignored in k3s and as a workaround we use a soft control in the form of retention property
# set to 3 days (see above)
storage:
  volumeClaimTemplate:
    spec:
      storageClassName: local-path
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 2Gi

```

Notice that according to current (Nov. 2022) Rancher documentation, the resource request declaration for storage (2GB in our case) will be ignored. In Prometheus CRD, however, we can indicate the durability of the monitoring data stored (default duration is 24h) thus preserving a level of control over the disk space used by Prometheus. As can be seen above, we have set it to 3 days with the `retention` attribute. In case of facing storage shortages the retention time can be decreased. Resizing a volume can be done following the guidelines under this [link](#).

1.3.1.3. Exposing Prometheus dashboard as LoadBalancer service

To expose Prometheus dashboard as a service of type LoadBalancer requires some updates to service manifests stored in our `monitoring/` directory. One option is to rename the existing manifest of the local (default) Service (possibly also renaming the service in the manifest, e.g., `name: prometheus-k8s-int`)

and create a new manifest for a Service of type LoadBalancer as shown below. In the example we make MetalLB assign a specific external IP address (chosen from the MetalLB pool) to the Prometheus service (loadBalancerIP: 192.168.1.201). In your case this address can be different.

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cp prometheus-service.yaml prometheus-
service-local.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano prometheus-service-ext.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cat prometheus-service-ext.yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0      # keep version the same as in the original file
  name: prometheus-k8s-ext
  namespace: monitoring
spec:
  ports:
    - name: web
      port: 9090
      targetPort: web
    - name: reloader-web
      port: 8080
      targetPort: reloader-web
  selector:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
  type: LoadBalancer
  loadBalancerIP: 192.168.1.201
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$
```

1.3.2. Deploying the updates to a running Prometheus server

- a) Run the `apply` command, check the location of Prometheus pod and the PV (actual volume) (in our case it is worker kpi094).

```

xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cd ..
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply -f monitoring/
# ... skipping the output from command execution
# checking the created Pod, PVC and PV and their placement

xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl get pods -n monitoring -o wide
NAME                                READY   STATUS    RESTARTS   AGE   IP            NODE
prometheus-k8s-0                    2/2     Running   0           63m   10.42.1.58    kpi094
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl get pvc -n monitoring
NAME                                STATUS    VOLUME                                     CAPACITY
ACCESS MODES   STORAGECLASS   AGE
prometheus-k8s-db-prometheus-k8s-0  Bound     pvc-b40a14ef-5a6b-49ce-bb6e-53bc56a489ca  2Gi
RWO            local-path     68m
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl describe -n monitoring pvc prometheus-
k8s-db-prometheus-k8s-0
Name:          prometheus-k8s-db-prometheus-k8s-0
Namespace:     monitoring
StorageClass:  local-path
Status:        Bound
Volume:        pvc-b40a14ef-5a6b-49ce-bb6e-53bc56a489ca
Labels:        app.kubernetes.io/instance=k8s
               app.kubernetes.io/managed-by=prometheus-operator
               app.kubernetes.io/name=prometheus
               operator.prometheus.io/name=k8s
               operator.prometheus.io/shard=0
               prometheus=k8s
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
               volume.beta.kubernetes.io/storage-provisioner: rancher.io/local-path
               volume.kubernetes.io/selected-node: kpi094
               volume.kubernetes.io/storage-provisioner: rancher.io/local-path
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      2Gi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       prometheus-k8s-0
Events:        <none>
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl describe -n monitoring pv pvc-
b40a14ef-5a6b-49ce-bb6e-53bc56a489ca
Name:          pvc-b40a14ef-5a6b-49ce-bb6e-53bc56a489ca
Labels:        <none>
Annotations:   pv.kubernetes.io/provisioned-by: rancher.io/local-path
Finalizers:    [kubernetes.io/pv-protection]
StorageClass:  local-path
Status:        Bound
Claim:         monitoring/prometheus-k8s-db-prometheus-k8s-0
Reclaim Policy: Delete
Access Modes:  RWO
VolumeMode:    Filesystem
Capacity:      2Gi
Node Affinity:
  Required Terms:
    Term 0:      kubernetes.io/hostname in [kpi094]
Message:
Source:
  Type:         HostPath (bare host directory volume)
  Path:         /var/lib/rancher/k3s/storage/pvc-b40a14ef-5a6b-49ce-bb6e-
53bc56a489ca_monitoring_prometheus-k8s-db-prometheus-k8s-0
  HostPathType: DirectoryOrCreate
Events:        <none>

```

It can be seen that the pod and the PV (the volume) are placed in the same node – kpi094. Interesting fact is that the PV has the property Node-Affinity set to `kubernetes.io/hostname in [kpi094]`; the PVC also has annotation `volume.kubernetes.io/selected-node: kpi094`. That means that if node kpi094 fails then even if the Prometheus pod will be rescheduled to another node, the volume will not due to being bound to node kpi094.

b) Check the access to the Prometheus dashboard

External (i.e., from your local network) access to Prometheus dashboard can be checked via browser or curl trying to reach <prometheus-service-ip-add>:9090. With all installation details as outlined above, you will get HTTP 404 error. This is because the default NetworkPolicy for Prometheus disallows such traffic and the policy needs to be updated. In the following frame the steps needed to check the reachability and update NetworkPolicy are documented. Notice that Kubernetes NetworkPolicy mechanism is in essence similar to Security Groups known from OpenStack, although functionally richer. Nevertheless, it seems to be a bit loose in regard to address blocks (*ipBlock*) in the sense that it is not unequivocally defined (known) which exactly address will be handled by the rule. The reason is that cluster ingress and egress mechanisms often require rewriting the source or destination IP of packets. In effect at the moment of writing the manifest it is unknown whether the rule will be applied to the address set by the application (as we would like it to be the case) or on address substituted by Kubernetes (ingress controller/kube-proxy, ...). This results in that NetworkPolicy mechanism should be used with great care for *ipBlocks*. A relevant note on that can be found [here](#), and some useful NetworkPolicy recipes can be found [here](#).

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl get services -n monitoring
NAME                                TYPE            CLUSTER-IP      EXTERNAL-IP      PORT(S)
prometheus-operator                 ClusterIP        None             <none>            8080/TCP
node-exporter                       ClusterIP        None             <none>            9100/TCP
kube-state-metrics                  ClusterIP        None             <none>            8443/TCP,9443/TCP
prometheus-k8s-external              LoadBalancer    10.43.253.155    192.168.1.201    9090:30922/TCP
prometheus-k8s                      ClusterIP        10.43.177.245    <none>            9090/TCP,8080/TCP
prometheus-k8s-nodeport              NodePort         10.43.56.246     <none>            9090:30090/TCP
prometheus-operated                  ClusterIP        None             <none>            9090/TCP

xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ curl 192.168.1.201:9090
curl: (7) Failed to connect to 192.168.1.202 port 9090: No route to host

xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ sudo nano prometheus-networkPolicy.yaml
# update the file adding a rule to allow traffic from the local network
# ipBlock cidr: 10.42.0.0/16 has been added as an addition to cope with NetworkPolicy uncertainties
# explained above and observed in practice.

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  labels:
    app.kubernetes.io/component: prometheus
    app.kubernetes.io/instance: k8s
    app.kubernetes.io/name: prometheus
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 2.42.0
  name: prometheus-k8s
  namespace: monitoring
spec:
  egress:
  - {}
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app.kubernetes.io/name: prometheus
      ports:
      - port: 9090
        protocol: TCP
      - port: 8080
        protocol: TCP
    - from:
      - podSelector:
          matchLabels:
            app.kubernetes.io/name: grafana
          ports:
          - port: 9090
            protocol: TCP
      - from:
      - ipBlock:
          cidr: 192.168.1.0/24 #with cidr: 10.42.0.0/16 as below this ipBlock may not be necessary
      - ipBlock:
          cidr: 10.42.0.0/16
    podSelector:
      matchLabels:
        app.kubernetes.io/component: prometheus
        app.kubernetes.io/instance: k8s
        app.kubernetes.io/name: prometheus
        app.kubernetes.io/part-of: kube-prometheus
  policyTypes:
  - Egress
  - Ingress

# check the reachability of Prometheus dashboard again
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ kubectl apply -f prometheus-networkPolicy.yaml
networkpolicy.networking.k8s.io/prometheus-k8s configured
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ curl 192.168.1.201:9090
<a href="/graph">Found</a>.
```

c) (Optional) Check if Prometheus is monitoring selected services

After achieving reachability you can access Prometheus dashboard with browser to see its main panels and check the health of installed monitors. To this end enter 192.168.1.201:9090 in your browser.

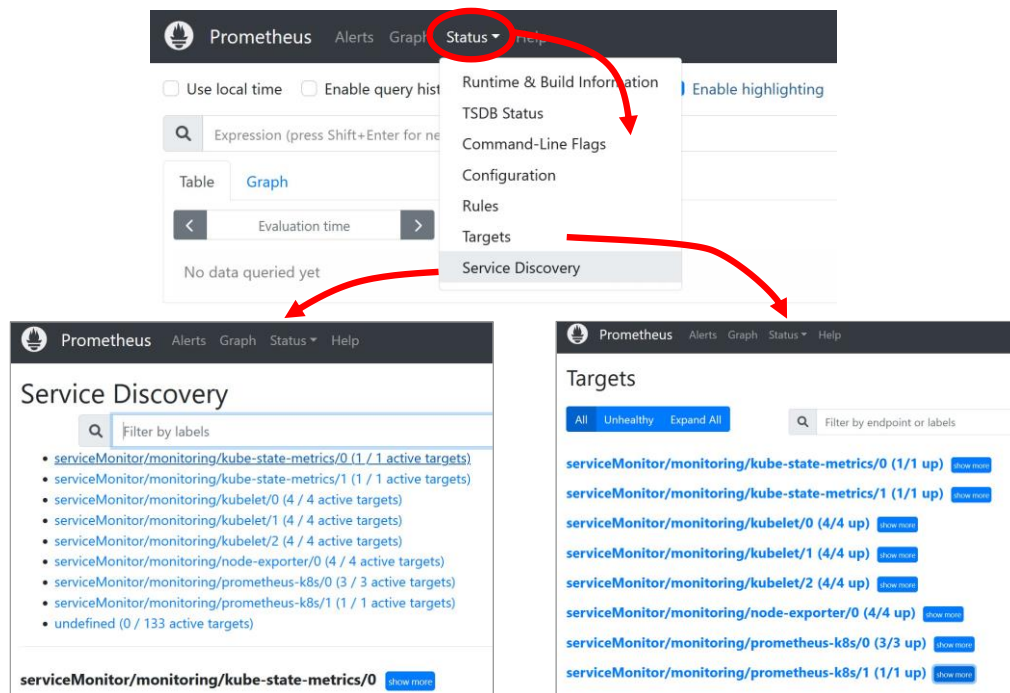


Figure 1 First touch of Prometheus dashboard – just checking how it looks like.

Hints - How to query with PromQL: <https://www.opsramp.com/guides/prometheus-monitoring/promql/>

1.4. Optional: Monitoring extensions for our cluster

Other monitors for our platform components² can be added, for example one may want to monitor Traefik or MetalLB. In the following, instructions are provide to include such extensions.

1.4.1. Traefik service monitor

Traefik comes with its Prometheus service monitor. If so, it can be good to also have Traefik metrics covered in our monitoring stack. Below is a list of reference links for Traefik service monitor we will use:

<https://github.com/cablespaghetti/k3s-monitoring>

<https://github.com/cablespaghetti/k3s-monitoring/blob/master/traefik-servicemonitor.yaml>

<https://raw.githubusercontent.com/cablespaghetti/k3s-monitoring/master/traefik-servicemonitor.yaml>

Follow the steps from the table below to deploy Traefik service monitor for Prometheus.

² Of course, Prometheus monitors for „regular” application services can also be installed but such applications are not covered in this part of the lab and can be included in the future as an extension of our lab series.

```
# make directory and download the manifests
```

```
xubuntu@xubulab:~/k3s-taskforce/manifests$ mkdir traefik-service-monitor
```

```
xubuntu@xubulab:~$ cd traefik-service-monitor
```

```
xubuntu@xubulab:~/k3s-taskforce/manifests/traefik-service-monitor$ wget  
https://raw.githubusercontent.com/cablespaghetti/k3s-monitoring/master/traefik-  
servicemonitor.yaml
```

```
#update the manifest to add namespace: monitoring if it is missing:
```

```
apiVersion: monitoring.coreos.com/v1
```

```
kind: ServiceMonitor
```

```
metadata:
```

```
  labels:
```

```
    app: traefik
```

```
    release: prometheus
```

```
  name: traefik
```

```
  namespace: monitoring
```

```
spec:
```

```
  endpoints:
```

```
  - port: metrics
```

```
  namespaceSelector:
```

```
    matchNames:
```

```
    - kube-system
```

```
  selector:
```

```
    matchLabels:
```

```
      app: traefik
```

```
# deploy traefik service monitor
```

```
xubuntu@xubulab:~/k3s-taskforce/manifests/traefik-service-monitor$ kubectl apply -f traefik-  
servicemonitor.yaml
```

```
# check service monitors deployed so far
```

```
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get servicemonitors -A
```

NAMESPACE	NAME	AGE
monitoring	traefik	16s
monitoring	kube-state-metrics	9m49s
monitoring	kubelet	6m41s
monitoring	node-exporter	12m

```
xubuntu@xubulab:~/k3s-taskforce/manifests/traefik-service-monitor$
```


2. Grafana

Grafana provides a rich set of dashboards to visualize the metrics scraped by Prometheus from the services monitored.

2.1. Manifest preparation

2.1.1. Downloading the manifests

- a) Create LoadBalancer service for Grafana and update NetworkPolicy to enable external traffic to Grafana. These steps are similar to the ones from the previous section for Prometheus. Remember to set an unused IP address if you want to choose the address yourself (otherwise MetalLB will assign one from it's pool).

```
# create a new file, say grafana-service-ext.yaml, defining LoadBalancer service for Grafana
dashboard
apiVersion: v1
kind: Service
metadata:
  labels:
    app.kubernetes.io/component: grafana
    app.kubernetes.io/name: grafana
    app.kubernetes.io/part-of: kube-prometheus
    app.kubernetes.io/version: 9.1.7
  name: grafana-ext
  namespace: monitoring
spec:
  type: LoadBalancer
  loadBalancerIP: 192.168.1.202
  ports:
    - name: http
      port: 3000
      targetPort: http
  selector:
    app.kubernetes.io/component: grafana
    app.kubernetes.io/name: grafana
    app.kubernetes.io/part-of: kube-prometheus

# update file grafana-networkPolicy.yaml adding the following the same way as for Prometheus:
- from:
  - ipBlock:
      cidr: 192.168.1.0/24
  - ipBlock:
      cidr: 10.42.0.0/16

# NOTE: this part was not needed when installing the original version of kube-prometheus (as
# described in the github).
# in manifest Grafana-deployment.yaml update the description of readiness probe and add liveness
# probe as below; you first may want to check what happens without updating the readiness probe
# settings (you will need to run kubectl describe pod -n monitoring <grafana-pod-current-name> )
  - containerPort: 3000
    name: http
  readinessProbe:
    httpGet:
      path: /api/health
      port: http
#----
#
  readinessProbe:
    failureThreshold: 3
    initialDelaySeconds: 10
    periodSeconds: 30
    successThreshold: 1
    timeoutSeconds: 2
#----
```

```
#----
  livenessProbe:
    failureThreshold: 3
    initialDelaySeconds: 30
    periodSeconds: 10
    successThreshold: 1
    tcpSocket:
      port: 3000
    timeoutSeconds: 1
#----
```

b) Configure persistent storage for Grafana

Grafana does not store monitoring data from Prometheus, but it uses dashboards and their JSON templates must be available to Grafana in run time for use. If you do not designate a location for information storage, then all your Grafana data including dashboards disappears as soon as you stop your container. To save your data, you need to set up persistent storage for your Grafana. To this end follow the steps in the frame below.

```
# create and edit file grafana-pvc.yaml to define a volume claim for grafana
xubuntu@xubulab:~/k3s-taskforce/manifests/grafana$ nano grafana-pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: grafana-local-path-pvc
  namespace: monitoring
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: local-path
  resources:
    requests:
      storage: 500Mi

# update grafana deployment file to create volume from the the pvc defined above
  serviceAccountName: grafana
  volumes:
# grafana-storage will be persisting to store dashboard templates; remaining volumes
# should remain ephemeral (emptyDir, secret, configMap)
#   - emptyDir: {}
#   name: grafana-storage
- name: grafana-storage
  persistentVolumeClaim:
    claimName: grafana-local-path-pvc
```

2.1.2. Deploying Grafana, checking main components and using the first dashboard

a) Deploy Grafana stack. Check basic elements to be present (pod, PVC, services)

```
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano prometheus-networkPolicy.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano grafana-pvc.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ nano grafana-pvc.yaml
xubuntu@xubulab:~/k3s-taskforce/manifests/monitoring$ cd ..
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl apply -f monitoring/
alertmanager.monitoring.coreos.com/main unchanged
. . .
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get pods -n monitoring
NAME                                READY   STATUS    RESTARTS   AGE
node-exporter-r454m                 2/2     Running   2 (22h ago) 27h
node-exporter-xlhbs                 2/2     Running   2 (22h ago) 27h
prometheus-operator-7b4d465f47-785hb 1/1     Running   1 (22h ago) 27h
node-exporter-th6k7                 2/2     Running   2 (22h ago) 27h
node-exporter-qgxr5                 2/2     Running   2 (22h ago) 27h
kube-state-metrics-8658546b69-c6wnk 3/3     Running   3 (22h ago) 27h
prometheus-k8s-0                    2/2     Running   2 (22h ago) 23h
grafana-96d64cd76-jt82p             1/1     Running   0           88s
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get pvc -n monitoring
NAME                                STATUS   VOLUME
CAPACITY  ACCESS MODES  STORAGECLASS  AGE
prometheus-k8s-db-prometheus-k8s-0  Bound    pvc-a4da82ff-b549-48a8-8e62-6e9a35e864d9  2Gi
RWO      local-path    23h
grafana-local-path-pvc              Bound    pvc-ad9d7a13-75cb-4522-a8d0-fe9627cb0cce  500Mi
RWO      local-path    2m42s
xubuntu@xubulab:~/k3s-taskforce/manifests$
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl describe -n monitoring pvc grafana-local-path-pvc
Name:          grafana-local-path-pvc
Namespace:     monitoring
StorageClass:  local-path
Status:        Bound
Volume:        pvc-ad9d7a13-75cb-4522-a8d0-fe9627cb0cce
Labels:        <none>
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
               volume.beta.kubernetes.io/storage-provisioner: rancher.io/local-path
               volume.kubernetes.io/selected-node: kpi093
               volume.kubernetes.io/storage-provisioner: rancher.io/local-path
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      500Mi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       grafana-96d64cd76-jt82p
Events:
  Type     Reason          Age          From          Message
  . . .
xubuntu@xubulab:~/k3s-taskforce/manifests$ kubectl get services -n monitoring
NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
prometheus-operator                 ClusterIP    None             <none>            8080/TCP          28h
node-exporter                       ClusterIP    None             <none>            9100/TCP          27h
kube-state-metrics                  ClusterIP    None             <none>            8443/TCP, 9443/TCP 27h
prometheus-k8s-external              LoadBalancer 10.43.253.155    192.168.1.201    9090:30922/TCP    24h
prometheus-k8s                      ClusterIP    10.43.177.245    <none>            9090/TCP, 8080/TCP 24h
prometheus-k8s-nodeport              NodePort     10.43.56.246     <none>            9090:30090/TCP    24h
prometheus-operated                  ClusterIP    None             <none>            9090/TCP          24h
grafana-external                     LoadBalancer 10.43.125.169    192.168.1.202    3000:31921/TCP    5m39s
grafana-local                       ClusterIP    10.43.56.206     <none>            3000/TCP          5m39s
xubuntu@xubulab:~/k3s-taskforce/manifests$
```

2.2. Checking Grafana – first graphs

a) Checking the access to Grafana dashboard

Enter 192.168.1.202:3000 in your browser and you will be presented the invitation panel of Grafana. Default user:passwd is admin:admin, and you can set your own credentials. Then follow the steps (arrows) from the picture in Figure 2 for setting up an exemplary dashboard for our cluster. Additional dashboards can be set in a similar way.

As can be noticed, the exemplary dashboard provides info about servers only which may seem too little in many applications. Other dashboards providing more information about the cluster (particular workloads, etc.) should be found/downloaded from elsewhere and/or configured on your own. Here go another two useful dashboards that should work out of the box:

- Kubernetes Cluster (cluster level overview of workloads deployed, based on Prometheus metrics exposed by kubelet, node-exporter, nginx ingress controller)
<https://grafana.com/grafana/dashboards/7249-kubernetes-cluster/>
- Kubernetes Cluster (Prometheus) (summary metrics for containers running on Kubernetes nodes)
<https://grafana.com/grafana/dashboards/6417-kubernetes-cluster-prometheus/>

entering IP addr 192.168.1.201 in place of prometheus-k8s-external would work as well

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
prometheus-operator	ClusterIP	None	<none>	8080/TCP
node-exporter	ClusterIP	None	<none>	9100/TCP
kube-state-metrics	ClusterIP	None	<none>	8443/TCP,9443/TCP,443/TCP
prometheus-k8s-external	LoadBalancer	10.43.253.155	192.168.1.202	9090,30922/TCP
prometheus-k8s	ClusterIP	10.43.177.245	<none>	9090/TCP,8080/TCP
prometheus-k8s-nodeport	NodePort	10.43.56.246	<none>	9090:30090/TCP
prometheus-operated	ClusterIP	None	<none>	9090/TCP
grafana-external	LoadBalancer	10.43.238.206	192.168.1.202	3000:31589/TCP

<https://grafana.com/grafana/dashboards/8171-kubernetes-nodes/>

find this numer in the page linked

Unique identifier (uid) of a dashboard can be used for uniquely identify a dashboard between multiple dashboards. The uid allows having consistent URLs for accessing dashboards without changing the title of a dashboard will not break any bookmarked links to that dashboard.

Prometheus

You need to have node_exporter > 0.16 that gets scraped by prometheus

Prometheus-cluster (default)

General / Kubernetes Nodes

Idle CPU

System Load

Memory Usage

27.6%

Disk I/O

Disk Space Usage

Figure 2 Exemplary Grafana dashboard for the cluster.

3. Useful readings and videos

Node/pod Affinity/antiaffinity explained: <https://medium.com/kubernetes-tutorials/learn-how-to-assign-pods-to-nodes-in-kubernetes-using-nodeselector-and-affinity-features-e62c437f3cf8>

topologyKey: `kubernetes.io/hostname` – scope of single host

topologyKey: `kubernetes.io/zone` – scope of single zone

`node-role.kubernetes.io/worker=true`

`node-role.kubernetes.io/controlplane=true`

`kubectl label node <node name> node-role.kubernetes.io/<role name>=<value>`

`kubectl get nodes --show-labels`

- kubectl cheat-sheet

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

- delete all resources from namespace

<https://stackoverflow.com/questions/47128586/how-to-delete-all-resources-from-kubernetes-one-time>

- How to Monitor a Kubernetes Cluster in 2022 with Prometheus & Grafana

<https://www.youtube.com/watch?v=YDtuwINTzRc>

- Modern Grafana dashboards for kubernetes

<https://medium.com/@dotdc/a-set-of-modern-grafana-dashboards-for-kubernetes-4b989c72a4b2>

- Adding Grafana dashboard using configMap

<https://fabianlee.org/2022/07/06/prometheus-adding-a-grafana-dashboard-using-a-configmap/>

- Simple Prometheus & Grafana setup with PVC

<https://medium.com/@bmbvfx/kubernetes-persistent-volume-claim-prometheus-grafana-4e821e283edc>

- Accessing Kubernetes services

OK: <https://blog.alexellis.io/primer-accessing-kubernetes-services/>

<https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster-services/>

Prometheus/AlertManager/Grafana/Loki - ciekawe, do pooglądania, wykorzystanie AlertManager

<https://www.youtube.com/watch?v=NABZqKq1McE>

przy okazji: strona z opisami helm-chartów dla aplikacji - B. CIEKAWA

<https://artifacthub.io/>

CNCF landscape

<https://landscape.cncf.io/>

Kubernetes API Basics - Resources, Kinds, and Objects

<https://iximiuz.com/en/posts/kubernetes-api-structure-and-terminology/>