Obserwowalność klastra: metryki, logi i alerty bez bólu

Poziom: średnie

Dotyczy: kube-prometheus-stack 91.8.2 (Prometheus Operator 0.94.1), nginx-prometheus-exporter 1.5.3, k3s 1.36, Zabbix 7.0 LTS, wykres kolorowy-nginx 0.3.0 (wersje sprawdzone 3.10.2026)

Czas: ~75 min

Stawiamy w laboratorium okrojony Prometheus z Grafaną przez Helm, dokładamy do kolorowego nginksa sidecar z eksporterem metryk i opisujemy go obiektem ServiceMonitor. Na końcu pokazujemy, jak się to ma do Zabbixa na `mon01`, który w Zielonej zostaje głównym miejscem alarmów.

Skróty użyte na tej stronie (9)

Skrót	Co znaczy
API	Application Programming Interface: sposób, w jaki programy rozmawiają ze sobą
CLI	Command Line Interface: praca przez wpisywanie komend w terminalu
HPA	HorizontalPodAutoscaler: obiekt Kubernetesa, który sam zwiększa albo zmniejsza liczbę podów Deploymentu, gdy zużycie procesora lub pamięci odbiega od celu
HTTP	Hypertext Transfer Protocol: protokół stron WWW
IP	Internet Protocol: adres komputera w sieci; w elektryce: stopień ochrony obudowy (Ingress Protection)
JSONPath	Zapis ścieżki, która wskazuje jedną wartość w dokumencie JSON, np. $.value to pole value na samym wierzchu
LTS	Long Term Support: wersja z długim wsparciem poprawkami
PromQL	Prometheus Query Language: język zapytań do metryk zapisanych w Prometheusie, używany w panelach Grafany i w regułach alertów
README	Read Me: plik tekstowy z opisem projektu, wykresu lub szablonu, czytany jako pierwszy

Fiszka

Komendy wykonujemy z CLI bash na stacji administratora albo na `k3s01`. Zakładamy wydanie `kolorowy-nginx` z poprzednich stron, z wykresem w wersji 0.3.0 i włączonym autoskalowaniem, oraz repozytorium `prometheus-community` dodane na stronie Helm: kolorowy nginx jako pakiet. Plik `kps-values-lab.yaml` pokazujemy w sekcji o instalacji stosu.

◫◫ ZMIENIA SYSTEM

Założenie przestrzeni nazw i hasła do Grafany: kubectl create secret

```
kubectl create namespace monitoring
kubectl create secret generic grafana-admin -n monitoring \
  --from-literal=admin-user=admin --from-literal=admin-
password='HasloAdministratoraGrafany'
```

Przykład:

```
namespace/monitoring created
secret/grafana-admin created
```

Hasło trzymamy w Secret utworzonym osobno, a nie w pliku wartości, bo wartości Helma zostają w Secret wydania i w repozytorium Gita.

◫◫ **kubectl** = Kubernetes control: główne narzędzie Kubernetesa

◫◫ ZMIENIA SYSTEM

Instalacja stosu monitoringu: helm install kube-prometheus-stack

```
helm install kps prometheus-community/kube-prometheus-stack --version 91.8.2 \
  --namespace monitoring -f kps-values-lab.yaml --wait --timeout 10m
```

Przykład:

```
NAME: kps
LAST DEPLOYED: Tue Oct  6 10:02:11 2026
NAMESPACE: monitoring
STATUS: deployed
```

```
REVISION: 1
DESCRIPTION: Install complete
```

Wydanie nazywamy krótko, `kps`, bo nazwa wchodzi do nazw wszystkich obiektów stosu. Przypięta wersja wykresu 91.8.2 to wersja sprawdzona 3.10.2026.

helm = menedżer paczek aplikacji dla Kubernetesa

TYLKO ODCZYT

Sprawdzenie podów stosu: kubectl get pods

```
kubectl get pods -n monitoring
```

Przykład:

```
NAME                                                     READY   STATUS    RESTARTS   AGE
alertmanager-kps-kube-prometheus-stack-alertmanager-0    2/2     Running   0          3m
kps-grafana-6b7f9c5d8f-x2jqk                             3/3     Running   0          3m
kps-kube-prometheus-stack-operator-5d9c7b6f74-9wq2z      1/1     Running   0          3m
kps-kube-state-metrics-7c8d9f5b6-hk4tp                   1/1     Running   0          3m
kps-prometheus-node-exporter-7zx8d                       1/1     Running   0          3m
prometheus-kps-kube-prometheus-stack-prometheus-0        2/2     Running   0          3m
```

Liczba kontenerów w kolumnie `READY` zależy od wersji wykresu, ważne, żeby wszystkie pody miały `Running` i pełną gotowość.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie dodanych typów obiektów: kubectl get crd

```
kubectl get crd | grep monitoring.coreos.com
```

Przykład:

```
alertmanagerconfigs.monitoring.coreos.com   2026-10-06T08:02:14Z
alertmanagers.monitoring.coreos.com         2026-10-06T08:02:14Z
podmonitors.monitoring.coreos.com           2026-10-06T08:02:14Z
prometheusrules.monitoring.coreos.com       2026-10-06T08:02:14Z
```

```
servicemonitors.monitoring.coreos.com       2026-10-06T08:02:15Z
```

Wykres dodał do klastra własne typy obiektów, w tym `ServiceMonitor` i `PrometheusRule`, których używamy niżej. Lista w klastrze jest dłuższa.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

ZMIENIA SYSTEM

Włączenie metryk kolorowego nginksa: helm upgrade --set metrics.enabled

```
helm upgrade kolorowy-nginx ./kolorowy-nginx --namespace kolory --reuse-values \
  --set metrics.enabled=true --set metrics.serviceMonitor.enabled=true --set
metrics.alerts.enabled=true --wait
```

Przykład:

```
Release "kolorowy-nginx" has been upgraded. Happy Helming!
NAME: kolorowy-nginx
STATUS: deployed
REVISION: 3
```

Pody dostają drugi kontener, `exporter`, a w przestrzeni nazw `kolory` pojawiają się ConfigMap, ServiceMonitor i PrometheusRule. Pody przechodzą aktualizację kroczącą, bo zmienił się ich szablon.

helm = menedżer paczek aplikacji dla Kubernetesa

TYLKO ODCZYT

Sprawdzenie nowych obiektów: kubectl get servicemonitor

```
kubectl get servicemonitor,prometheusrule,pods -n kolory
```

Przykład:

```
NAME                                                AGE
servicemonitor.monitoring.coreos.com/kolorowy-nginx   40s
```

```
NAME                                                AGE
prometheusrule.monitoring.coreos.com/kolorowy-nginx   40s

NAME                                 READY   STATUS    RESTARTS   AGE
pod/kolorowy-nginx-7d9c6b5f8d-4tkq2   2/2     Running   0          38s
pod/kolorowy-nginx-7d9c6b5f8d-m8zr5   2/2     Running   0          52s
```

Wartość `2/2` w kolumnie `READY` oznacza, że w każdym podzie działają teraz dwa kontenery: nginx i exporter.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Odczyt metryk wprost z eksportera: curl przez port-forward

```
kubectl port-forward -n kolory deploy/kolorowy-nginx 9113:9113 &
curl -s localhost:9113/metrics | grep -E
'^nginx_(up|http_requests_total|connections_active)'
```

Przykład:

```
nginx_connections_active 2
nginx_http_requests_total 1301
nginx_up 1
```

Wartość `nginx_up 1` znaczy, że eksporter dosięgnął licznika `stub_status` nginksa. Zamykamy przekierowanie poleceniem `kill %1`.

kubectl = Kubernetes control: główne narzędzie Kubernetesa · **curl** = client URL: pobieranie i wysyłanie przez HTTP z terminala

TYLKO ODCZYT

Podejrzenie Prometheusa z komputera: kubectl port-forward

```
kubectl port-forward -n monitoring svc/kps-kube-prometheus-stack-prometheus 9090:9090
```

Przykład:

```
Forwarding from 127.0.0.1:9090 -> 9090
Forwarding from [::1]:9090 -> 9090
```

Po ustawieniu przekierowania Prometheus jest dostępny pod `http://localhost:9090`, a w tym samym oknie przeglądarki widzimy zakładkę Status z celami zbierania danych.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Sprawdzenie, czy Prometheus widzi cele nginksa: zapytanie up

```
curl -s 'http://localhost:9090/api/v1/query?query=up%7Bnamespace%3D%22kolory%22%7D' |
python3 -m json.tool | grep -E '"(pod|value)"|"1"' | head
```

Przykład:

```
                    "pod": "kolorowy-nginx-7d9c6b5f8d-4tkq2"
                    "1"
                    "pod": "kolorowy-nginx-7d9c6b5f8d-m8zr5"
                    "1"
```

Wartość `1` oznacza, że ostatnie zbieranie z danego poda się udało.

curl = client URL: pobieranie i wysyłanie przez HTTP z terminala

KLIKAMY W INTERFEJSIE

Podejrzenie celów zbierania w interfejsie Prometheusa

```
Prometheus → Status → Target health → serviceMonitor/kolory/kolorowy-nginx/0
```

To samo, co daje zapytanie `up`, widzimy w przeglądarce razem z czasem ostatniego zbierania i ewentualnym błędem. Po przekierowaniu portu z poprzedniej fiszki adres to `http://localhost:9090`.

TYLKO ODCZYT

Zapytanie o liczbę zapytań na sekundę: rate

```
curl -s
'http://localhost:9090/api/v1/query?query=sum(rate(nginx_http_requests_total%5B1m%5D))'
```

Przykład:

```
{"status":"success","data":{"resultType":"vector","result":[{"metric":{},"value":[179000
0000.123,"0.9833"]}]}}
```

Wynik to średnia liczba zapytań na sekundę ze wszystkich podów w ostatniej minucie. Ten sam wyraz wklejamy w panel w Grafanie.

curl = client URL: pobieranie i wysyłanie przez HTTP z terminala

TYLKO ODCZYT

Podgląd logów kontenera nginx we wszystkich podach: kubectl logs --prefix

```
kubectl logs -n kolory -l app.kubernetes.io/name=kolorowy-nginx -c nginx --prefix --
tail=2 --since=5m
```

Przykład:

```
[pod/kolorowy-nginx-7d9c6b5f8d-4tkq2/nginx] 10.42.0.1 - - [06/Oct/2026:08:14:07 +0000]
"GET / HTTP/1.1" 200 148 "-" "curl/8.9.1"
[pod/kolorowy-nginx-7d9c6b5f8d-m8zr5/nginx] 10.42.0.1 - - [06/Oct/2026:08:14:09 +0000]
"GET / HTTP/1.1" 200 148 "-" "curl/8.9.1"
```

Flaga `--prefix` dopisuje nazwę poda i kontenera na początku każdej linii, więc widzimy, do którego poda trafiło żądanie.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

TYLKO ODCZYT

Podgląd logów eksportera: kubectl logs -c exporter

```
kubectl logs -n kolory deploy/kolorowy-nginx -c exporter --tail=5
```

Wypisuje ostatnie linie dziennika kontenera `exporter`. Przy problemach z metrykami to pierwsze miejsce, w którym szukamy komunikatów o połączeniu z `stub_status`.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

OSTROŻNIE · NIEODWRACALNE

Wywołanie alertu na próbę: usunięcie ServiceMonitora

```
kubectl delete servicemonitor kolorowy-nginx -n kolory
```

Po usunięciu ServiceMonitora Prometheus przestaje zbierać metryki nginksa, a po pięciu minutach reguła `KolorowyNginxBrakMetryk` przechodzi ze stanu `Pending` do `Firing`. Przywracamy obiekt poleceniem `helm upgrade` z tymi samymi wartościami, które odtwarza brakujące obiekty wykresu.

kubectl = Kubernetes control: główne narzędzie Kubernetesa

Z życia wzięte

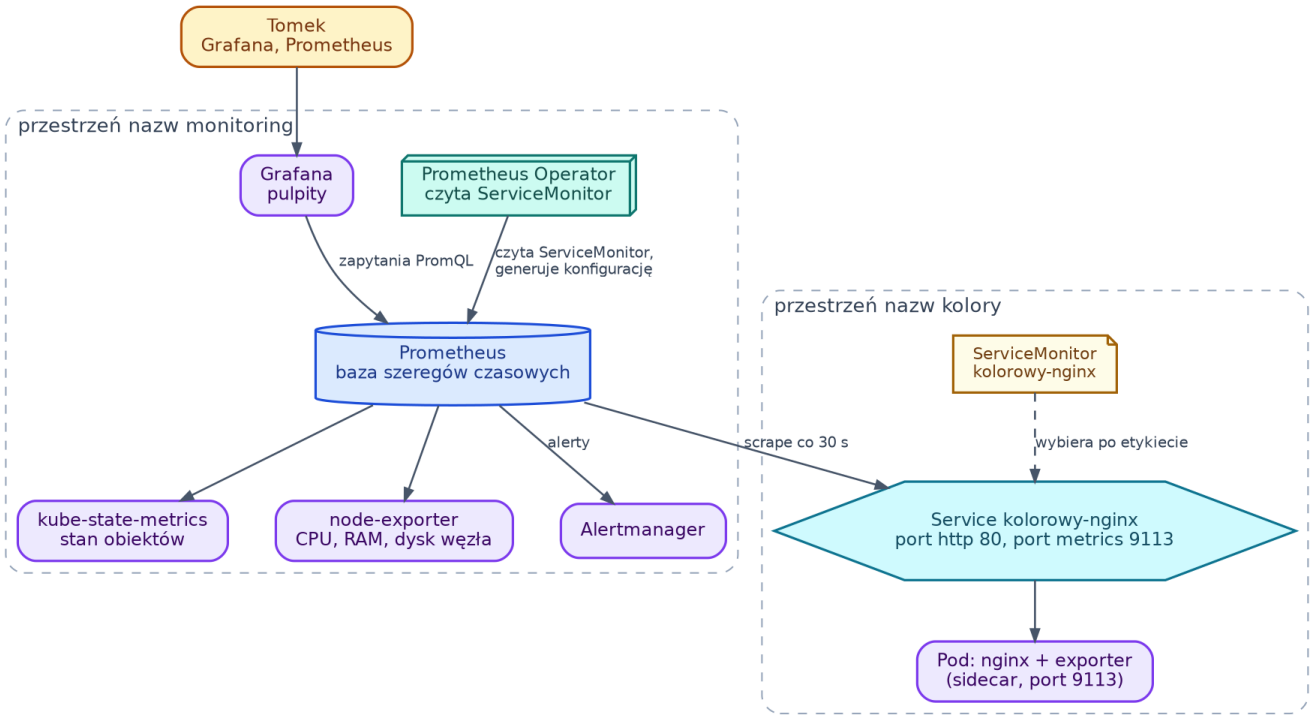
Z ŻYCIA WZIĘTE

Czwartek, 11:25. Ola pyta, czy strona kolorów wytrzymała poniedziałkowy szczyt, bo na wykresie w Grafanie widzi skok do 40 zapytań na sekundę, a Tomek pamięta tylko, że „coś się skalowało”. Tomek otwiera pulpit, nakłada na siebie liczbę podów z kube-state-metrics i zapytania na sekundę z eksportera, a wykresy pokrywają się co do minuty. Bez Prometheusa miałby w głowie tylko jedno `kubectl get hpa` z chwili, w której akurat patrzył.

Jak to działa

Rozwiń: jak to działa pod spodem

`kubectl top` z poprzedniej strony pokazuje, ile podów zużywa teraz. Nie pamięta niczego sprzed minuty, nie wie, ile żądań obsłużył nginx, i nie powiadomi nikogo w nocy. Do tego potrzebujemy trzech rzeczy: kogoś, kto zbiera liczby w czasie, kogoś, kto je rysuje, i kogoś, kto się odzywa, gdy liczby wyglądają źle.



W przestrzeni nazw monitoring Prometheus zbiera metryki z kube-state-metrics, node-exporter i z Service kolorowego nginksa wskazanego przez ServiceMonitor, a Grafana i Alertmanager korzystają z jego danych (kliknięcie otwiera pełny rozmiar)

Pojęcie: Prometheus. System, który co kilkadziesiąt sekund sam odpytuje wskazane adresy HTTP o ich metryki i zapisuje wyniki jako szeregi czasowe: nazwa metryki, etykiety i wartość z czasem. Model polega na tym, że to Prometheus idzie po dane, a aplikacja tylko wystawia stronę `/metrics`. Zapytania zapisujemy w języku PromQL.

Pojęcie: eksporter (exporter). Mały program, który tłumaczy to, co potrafi dana aplikacja, na format metryk Prometheusa. Nginx sam nie wystawia `/metrics`, ale potrafi pokazać prosty licznik `stub_status`. Eksporter czyta ten licznik i wystawia go jako metryki, na przykład `nginx_http_requests_total`.

Pojęcie: ServiceMonitor. Obiekt Kubernetesa dodany przez Prometheus Operator, który mówi Prometheusowi, które Service i który port ma odpytywać. Zamiast edytować konfigurację Prometheusa, dopisujemy ServiceMonitor obok aplikacji, w jej wykresie. Operator zauważa nowy obiekt i sam przebudowuje konfigurację.

> **Pojęcie: kube-state-metrics.** Program, który odczytuje obiekty z API Kubernetesa i zamienia ich stan na metryki: ile replik ma Deployment, czy pod jest gotowy, ile maksymalnie pozwala HPA. To inne dane niż `kubectl top`, bo dotyczą stanu obiektów, a nie zużycia zasobów.

Z czego składa się stos

Wykres `kube-prometheus-stack` instaluje za jednym razem wszystko, co widać na schemacie: Prometheus Operator, Prometheus, Alertmanager, Grafanę, kube-state-metrics i node-exporter, razem z gotowymi pulpitami i regułami alertów dla klastra. Wszystko jest w jednym wykresie, więc instalujemy je jednym `helm install` i aktualizujemy jednym `helm upgrade`. Cena to rozmiar: bez ograniczeń Prometheus i Grafana potrafią zająć kilka gigabajtów pamięci, a laboratorium na `k3s01` ma jedną maszynę.

Dlatego instalujemy wersję okrojoną plikiem wartości. Zostawiamy retencję trzech dni, wyraźne limity pamięci, trwały wolumen na `local-path` o rozmiarze 5Gi i wyłączamy cztery cele, których k3s nie wystawia. Przed instalacją sprawdzamy `free -h` na `k3s01` i dla pewności zmniejszamy `maxReplicas` w HPA, żeby pody kolorów nie walczyły o pamięć z Prometheusem.

```
# Okrojony kube-prometheus-stack dla laboratorium na k3s01
defaultRules:
  create: true

alertmanager:
  enabled: true
  alertmanagerSpec:
    resources:
      requests:
        cpu: 10m
        memory: 32Mi
      limits:
        memory: 64Mi

grafana:
  enabled: true
  admin:
    existingSecret: grafana-admin
    userKey: admin-user
```

```
    passwordKey: admin-password
  ingress:
    enabled: true
    ingressClassName: traefik
    hosts:
      - grafana.zielona.example
  resources:
    requests:
      cpu: 50m
      memory: 128Mi
    limits:
      memory: 256Mi

prometheus:
  ingress:
    enabled: true
    ingressClassName: traefik
    hosts:
      - prometheus.zielona.example
  prometheusSpec:
    retention: 3d
    resources:
      requests:
        cpu: 100m
        memory: 512Mi
      limits:
        memory: 1Gi
    # Bez tych trzech linii Prometheus widzi tylko obiekty z etykietą wydania Helm
    serviceMonitorSelectorNilUsesHelmValues: false
    podMonitorSelectorNilUsesHelmValues: false
    ruleSelectorNilUsesHelmValues: false
    storageSpec:
      volumeClaimTemplate:
        spec:
          storageClassName: local-path
          accessModes: ["ReadWriteOnce"]
          resources:
            requests:
```

```
              storage: 5Gi

# k3s uruchamia te komponenty w jednym procesie i nie wystawia ich metryk na zewnątrz
kubeEtcd:
  enabled: false
kubeScheduler:
  enabled: false
kubeControllerManager:
  enabled: false
kubeProxy:
  enabled: false
```

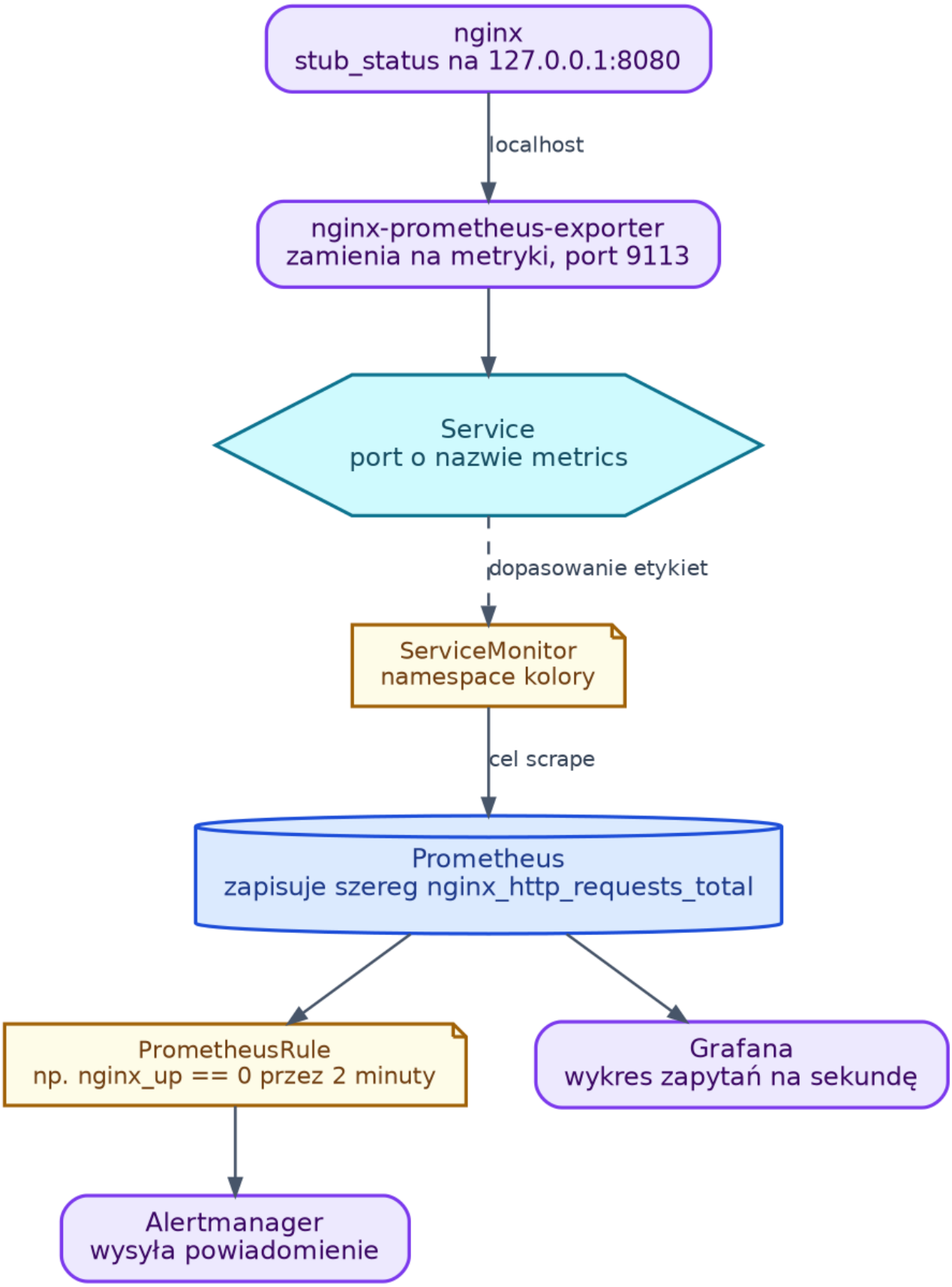
Trzy linie `...NilUsesHelmValues: false` rozwiązują najczęstszy kłopot nowicjuszy. Domyślnie Prometheus wybiera tylko ServiceMonitory z etykietą wydania `kps`, więc ServiceMonitor z wykresu kolorowego nginksa, który tej etykiety nie ma, jest dla niego niewidoczny. Wartość `false` mówi Prometheusowi, żeby brał pod uwagę wszystkie obiekty w klastrze.

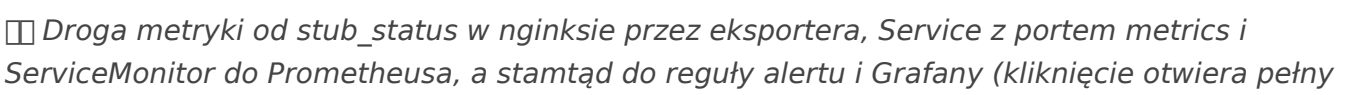
> Prometheus i Grafana wystawione przez Ingress bez uwierzytelniania są dostępne dla całej sieci, która dosięgnie `k3s01`. W laboratorium ograniczamy dostęp po adresach IP stacji administratorów albo zostawiamy samo przekierowanie portu, a w produkcji dokładamy logowanie przez Traefika.

Sidecar z eksporterem w wykresie

> **Pojęcie: sidecar.** Dodatkowy kontener w tym samym podzie co aplikacja, opisany na stronie Pod: najmniejsza jednostka. Nasz eksporter dzieli z nginksem sieć poda, więc dosięga licznika przez `127.0.0.1`, a nie przez sieć klastra.

Droga metryki zaczyna się w nginksie, który wystawia licznik tylko dla własnego poda, a kończy w Prometheusie, który zapisuje wartość i porównuje ją z regułą alertu.


nginx
stub_status na 127.0.0.1:8080
localhost
nginx-prometheus-exporter
zamienia na metryki, port 9113
Service
port o nazwie metrics
dopasowanie etykiet
ServiceMonitor
namespace kolory
cel scrape
Prometheus
zapisuje szereg nginx_http_requests_total
PrometheusRule
np. nginx_up == 0 przez 2 minuty
Grafana
wykres zapytań na sekundę
Alertmanager
wysyła powiadomienie



Droga metryki od stub_status w nginksie przez eksportera, Service z portem metrics i ServiceMonitor do Prometheusa, a stamtąd do reguły alertu i Grafany (kliknięcie otwiera pełny rozmiar)

Wykres, który nazywamy teraz 0.3.0, dostaje cztery zmiany:

- blok `metrics` w `values.yaml`, wyłączony domyślnie, tak że wykres działa także w klastrach bez Prometheusa
- ConfigMap z konfiguracją nginksa, w której osobny serwer na porcie 8080 pokazuje `stub_status` wyłącznie dla adresu `127.0.0.1`
- drugi port w Service i drugi kontener w podzie
- ServiceMonitor i PrometheusRule

```
# Dopisek do values.yaml (przed nameOverride)
metrics:
  enabled: false
  image:
    repository: nginx/nginx-prometheus-exporter
    tag: "1.5.3"
  resources:
    requests:
      cpu: 10m
      memory: 16Mi
    limits:
      cpu: 50m
      memory: 32Mi
  serviceMonitor:
    enabled: false
    interval: 30s
  alerts:
    enabled: false
```

Konfiguracja nginksa trafia do ConfigMap, który kontener nginx montuje w katalogu `/etc/nginx/conf.d`, zasłaniając domyślny plik obrazu. Serwer na porcie 80 serwuje stronę z kolorem, jak dotąd.

```
{{- if .Values.metrics.enabled }}
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: {{ include "kolorowy-nginx.fullname" . }}-nginx
  labels:
    {{- include "kolorowy-nginx.labels" . | nindent 4 }}
data:
  default.conf: |
    server {
      listen 80;
      server_name _;
      root /usr/share/nginx/html;
      location / {
        index index.html;
      }
    }
    # Licznik stub_status tylko dla eksportera z tego samego poda
    server {
      listen 8080;
      server_name localhost;
      allow 127.0.0.1;
      deny all;
      location = /stub_status {
        stub_status;
      }
    }
{{- end }}
```

Service dostaje drugi port o nazwie `metrics`. Nazwa jest ważna, bo ServiceMonitor odwołuje się do portu po nazwie, nie po numerze.

```
apiVersion: v1
kind: Service
metadata:
  name: {{ include "kolorowy-nginx.fullname" . }}
  labels:
    {{- include "kolorowy-nginx.labels" . | nindent 4 }}
spec:
  type: ClusterIP
  selector:
    {{- include "kolorowy-nginx.selectorLabels" . | nindent 4 }}
```

```
  ports:
    - name: http
      port: {{ .Values.service.port }}
      targetPort: http
      protocol: TCP
    {{- if .Values.metrics.enabled }}
    - name: metrics
      port: 9113
      targetPort: metrics
      protocol: TCP
    {{- end }}
```

Deployment zyskuje warunkowy kontener `exporter`, montowanie ConfigMap i adnotację z sumą kontrolną konfiguracji. Suma kontrolna sprawia, że zmiana w ConfigMap zmienia szablon poda, więc pody uruchamiają się od nowa z nową konfiguracją. Bez niej nginx dalej czytałby starą wersję pliku.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: {{ include "kolorowy-nginx.fullname" . }}
  labels:
    {{- include "kolorowy-nginx.labels" . | nindent 4 }}
spec:
  {{- if not .Values.autoscaling.enabled }}
  replicas: {{ .Values.replicaCount }}
  {{- end }}
  selector:
    matchLabels:
      {{- include "kolorowy-nginx.selectorLabels" . | nindent 6 }}
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 0
  template:
    metadata:
      {{- if .Values.metrics.enabled }}
      annotations:
```

```
        checksum/konfiguracja-nginx: {{ include (print $.Template.BasePath
"/configmap.yaml") . | sha256sum }}
      {{- end }}
      labels:
        {{- include "kolorowy-nginx.selectorLabels" . | nindent 8 }}
    spec:
      initContainers:
        - name: generator-koloru
          image: "{{ .Values.generator.repository }}:{{ .Values.generator.tag }}"
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_IP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
            - name: KOLORY
              value: {{ include "kolorowy-nginx.paleta" . | quote }}
          command:
            - sh
            - -c
            - |
              set -e
              LICZBA=$(echo $KOLORY | wc -w)
              HASH=$(echo -n "$POD_NAME" | cksum | cut -d' ' -f1)
              INDEKS=$((HASH % LICZBA))
              WYBRANY=$(echo $KOLORY | cut -d' ' -f$((INDEKS + 1)))
              NAZWA=$(echo $WYBRANY | cut -d':' -f1)
              HEX=$(echo $WYBRANY | cut -d':' -f2)
              echo "<html><body style='background-color:$HEX; font-family:sans-serif;
text-align:center; padding-top:15%;'>" > /strona/index.html
              echo "<h1>Pod: $POD_NAME</h1>" >> /strona/index.html
              echo "<h2>Kolor: $NAZWA</h2>" >> /strona/index.html
              echo "<h3>Adres IP: $POD_IP</h3>" >> /strona/index.html
              echo "</body></html>" >> /strona/index.html
          resources:
```

```
            {{- toYaml .Values.generator.resources | nindent 12 }}
          volumeMounts:
            - name: strona
              mountPath: /strona
      containers:
        - name: nginx
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag | default
.Chart.AppVersion }}"
          imagePullPolicy: {{ .Values.image.pullPolicy }}
          ports:
            - name: http
              containerPort: 80
          readinessProbe:
            httpGet:
              path: /
              port: http
            periodSeconds: 5
            failureThreshold: 3
          livenessProbe:
            httpGet:
              path: /
              port: http
            periodSeconds: 10
            failureThreshold: 3
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
          volumeMounts:
            - name: strona
              mountPath: /usr/share/nginx/html
              readOnly: true
            {{- if .Values.metrics.enabled }}
            - name: konfiguracja-nginx
              mountPath: /etc/nginx/conf.d
              readOnly: true
            {{- end }}
        {{- if .Values.metrics.enabled }}
        - name: exporter
          image: "{{ .Values.metrics.image.repository }}:{{ .Values.metrics.image.tag }}"
```

```
          args:
            - --nginx.scrape-uri=http://127.0.0.1:8080/stub_status
          ports:
            - name: metrics
              containerPort: 9113
          resources:
            {{- toYaml .Values.metrics.resources | nindent 12 }}
        {{- end }}
      volumes:
        - name: strona
          emptyDir: {}
        {{- if .Values.metrics.enabled }}
        - name: konfiguracja-nginx
          configMap:
            name: {{ include "kolorowy-nginx.fullname" . }}-nginx
        {{- end }}
```

Eksporter ma własne `requests`, i jest to konieczne. Autoskaler z poprzedniej strony liczy zużycie względem sumy `requests` wszystkich kontenerów poda, więc sidecar bez `requests` unieruchomiłby HPA komunikatem `missing request for cpu`.

ServiceMonitor wybiera Service po tych samych etykietach, po których Service wybiera pody. Wpis `endpoints` wskazuje port po nazwie i interwał zbierania.

```
{{- if and .Values.metrics.enabled .Values.metrics.serviceMonitor.enabled }}
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: {{ include "kolorowy-nginx.fullname" . }}
  labels:
    {{- include "kolorowy-nginx.labels" . | nindent 4 }}
spec:
  selector:
    matchLabels:
      {{- include "kolorowy-nginx.selectorLabels" . | nindent 6 }}
  endpoints:
    - port: metrics
      interval: {{ .Values.metrics.serviceMonitor.interval }}
{{- end }}
```

Alerty: PrometheusRule

Reguła alertu to zapytanie PromQL razem z czasem, przez jaki ma być prawdziwe (`for`). Zbyt krótkie `for` daje alarmy po każdym krótkim zacięciu, zbyt długie spóźnia powiadomienie. Nasze reguły pokrywają trzy różne sytuacje: pojedynczy pod bez metryk nginksa, brak jakichkolwiek metryk i autoskalowanie na maksimum. Trzecia reguła pojawia się tylko wtedy, gdy włączone jest autoskalowanie.

```
{{- if and .Values.metrics.enabled .Values.metrics.alerts.enabled }}
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  name: {{ include "kolorowy-nginx.fullname" . }}
  labels:
    {{- include "kolorowy-nginx.labels" . | nindent 4 }}
spec:
  groups:
    - name: kolorowy-nginx
      rules:
        - alert: KolorowyNginxPodNieOdpowiada
          expr: nginx_up{namespace="{{ .Release.Namespace }}", service="{{ include
"kolorowy-nginx.fullname" . }}"} == 0
          for: 2m
          labels:
            severity: warning
          annotations:
            summary: "Eksporter nie widzi nginksa w podzie"
            description: 'Pod {{ "{{ $labels.pod }}" }} nie odpowiada na stub_status od 2
minut'
        - alert: KolorowyNginxBrakMetryk
          expr: absent(nginx_up{namespace="{{ .Release.Namespace }}", service="{{ include
"kolorowy-nginx.fullname" . }}"})
          for: 5m
          labels:
            severity: critical
          annotations:
            summary: "Prometheus nie dostaje metryk kolorowego nginksa"
{{- if .Values.autoscaling.enabled }}
```

```
        - alert: KolorowyNginxHpaNaMaksimum
          expr: kube_horizontalpodautoscaler_status_current_replicas{namespace="{{
.Release.Namespace }}", horizontalpodautoscaler="{{ include "kolorowy-nginx.fullname" .
}}"} >= on(namespace, horizontalpodautoscaler)
kube_horizontalpodautoscaler_spec_max_replicas{namespace="{{ .Release.Namespace }}",
horizontalpodautoscaler="{{ include "kolorowy-nginx.fullname" . }}"}
          for: 10m
          labels:
            severity: warning
          annotations:
            summary: "Autoskalowanie stoi na maksimum od 10 minut"
{{- end }}
{{- end }}
```

Adnotacja z `{{ "{{ $labels.pod }}" }}` wygląda dziwnie, bo Helm i Prometheus używają takiej samej składni podwójnych nawiasów klamrowych. Zapis w cudzysłowie każe Helmowi wypisać ten tekst dosłownie, a wyrażenie wykona dopiero Prometheus.

Alertmanager przyjmuje alerty od Prometheusa i decyduje, komu je wysłać. Dla laboratorium zostawiamy go z konfiguracją domyślną, więc alerty widzimy w jego interfejsie, ale nigdzie nie wychodzą. Powiadomienia e-mail przez serwer pocztowy firmy konfigurujemy w osobnym kroku, w sekcji `alertmanager.config` pliku wartości.

Logi: kubectl logs, a potem Loki

Dla jednego poda i jednej awarii wystarczy `kubectl logs`, jak w fiszkach wyżej i na stronie Diagnostyka: objaw, przyczyna, pierwsze kroki. Ma to dwie granice. Logi znikają razem z podem, a kubelet rotuje je na węźle, domyślnie zostawiając pięć plików po 10Mi na kontener.

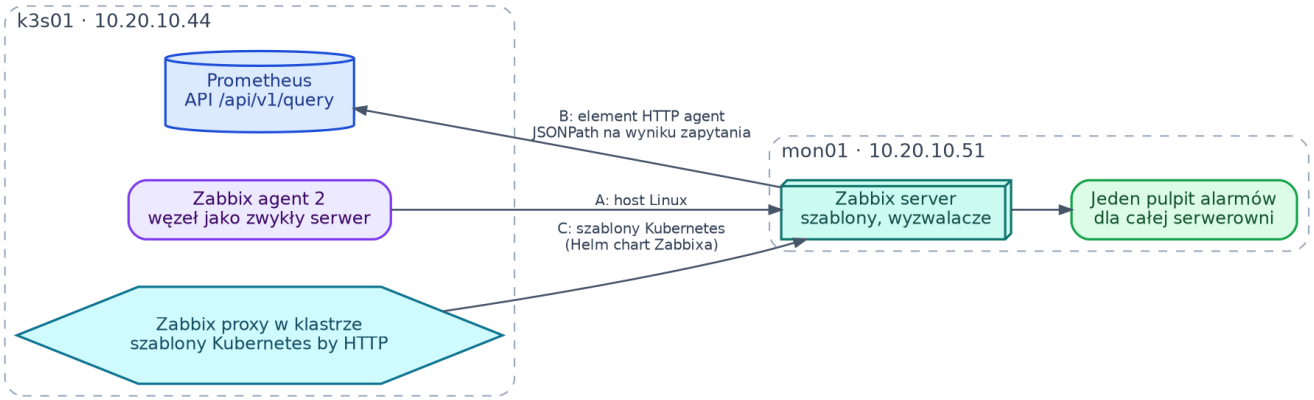
> **Pojęcie: Loki.** System do przechowywania logów z Kubernetesa od twórców Grafany, który indeksuje tylko etykiety, a treść linii zostawia nieindeksowaną. Dzięki temu jest tani w utrzymaniu i dobrze współpracuje z Grafaną, w której logi widać obok metryk. Logi do Lokiego wysyła agent, rekomendowany w dokumentacji to Grafana Alloy.

W Zielonej mamy już Graylog na `docker01`, opisany na stronie Po co centralne logowanie i jak to działa. Dla laboratorium nie dokładamy więc drugiego systemu logów bez potrzeby. Loki warto dołożyć, gdy chcemy zobaczyć logi w tej samej Grafanie co metryki. W przeciwnym razie kierujemy logi z `k3s01` do Grayloga, który zespół już zna.

Instalację Lokiego opisuje dokumentacja Grafany dla trybu monolitycznego (single binary) z magazynem plikowym. Wykres ma osobne repozytorium, więc wersję wykresu wyszukujemy poleceniem `helm search repo` i przypinamy, tak jak wcześniej wykres stosu monitoringu. Szczegółowych wartości tu nie przepisujemy, bo zmieniają się między wersjami i sprawdzamy je w dokumentacji instalacji.

Zabbix na mon01 a Prometheus

Zielona ma już Zabbixa na `mon01`, opisanego na stronie Architektura Zabbixa: serwer, proxy, agent, frontend, i nie zamierza prowadzić dwóch miejsc z alarmami. Prometheus w klastrze zostaje więc narzędziem do oglądania i diagnozy, a decyzje o tym, kogo budzić, zostają w Zabbixie. Mamy trzy sposoby połączenia, które stosujemy razem.



Zabbix na mon01 dostaje dane z agenta na węźle k3s01 (A), pyta API Prometheusa elementem HTTP agent (B) i korzysta z szablonów Kubernetes by HTTP przez proxy w klastrze (C) (kliknięcie otwiera pełny rozmiar)

Sposób	Co daje	Wymaga	Nasz wybór
A: host `k3s01` z agentem 2	procesor, pamięć, dysk i usługi maszyny, na której stoi k3s	agent Zabbixa na węźle, jak na stronie Agent: instalacja i konfiguracja	tak, od razu
B: element HTTP agent do API Prometheusa	pojedyncze wartości z Prometheusa, na przykład zapytania na sekundę	adres Prometheusa osiągalny z `mon01`, na przykład przez Ingress ograniczony do adresu `mon01`	tak, dla kilku kluczowych liczb

Sposób	Co daje	Wymaga	Nasz wybór
C: szablony Kubernetes by HTTP	stan węzłów, podów i komponentów klastra w Zabbixie	Zabbix proxy i agent w klastrze z wykresu Helm Zabbixa, token konta usługi	później, gdy klaster dorośnie

Zabbix 7.0 ma sześć oficjalnych szablonów do Kubernetesa: Kubernetes nodes by HTTP, Kubernetes cluster state by HTTP, Kubernetes API server by HTTP, Kubernetes Controller manager by HTTP, Kubernetes Scheduler by HTTP i Kubernetes kubelet by HTTP. Pracują przez zapytania HTTP do punktów `/metrics` komponentów klastra, a dokumentacja wymaga wdrożenia w klastrze wykresu Helm Zabbixa z proxy. Nazwy makr, token konta usługi i kolejność kroków sprawdzamy w pliku README szablonu `Kubernetes cluster state by HTTP` dla Zabbixa 7.0, bo ich nie przepisujemy z pamięci.

Sposób B ustawiamy bez niczego w klastrze poza dostępem do Prometheusa. W interfejsie Zabbixa tworzymy dla hosta `k3s01` element danych, który co minutę wykonuje zapytanie o liczbę żądań na sekundę.

```
Zabbix → Data collection → Hosts → k3s01 → Items → Create item
```

Pole elementu	Wartość
Name	Kolorowy nginx: zapytania na sekundę
Type	HTTP agent
Key	`kolory.nginx.rps`
URL	`http://prometheus.zielona.example/api/v1/query`
Query fields	nazwa `query`, wartość `sum(rate(nginx_http_requests_total{namespace="kolory"}[1m]))`
Request type	GET
Type of information	Numeric (float)
Update interval	1m
Preprocessing	JSONPath `$.data.result[0].value[1]`

> □ Zrzut do uzupełnienia: ekran Zabbixa z formularzem elementu danych dla hosta `k3s01`, z widocznymi typem „HTTP agent”, adresem Prometheusa, polem „Query fields” z zapytaniem PromQL i zakładką „Preprocessing” z krokiem JSONPath.

Na tym elemencie budujemy wyzwalacz, na przykład ostrzeżenie, gdy wartość przekracza z góry ustaloną liczbę. Jak się robi wyzwalacze i akcje, opisuje strona Host, element danych, wyzwalacz, akcja: jak to się składa. Wynik dla Zabbixa jest wtedy taki sam jak dla każdego innego hosta: jedno miejsce z alarmami, mimo że za liczbą stoi cały stos z klastra.

Częste błędy

Objaw	Przyczyna	Co robimy
`helm install kps` kończy się `context deadline exceeded`, a pody są `Pending`	za mało pamięci albo procesora na `k3s01`	`kubectl describe pod -n monitoring`, `free -h`, mniejsze `requests` w `kps-values-lab.yaml`
W Prometheusie alerty `KubeSchedulerDown`, `KubeControllerManagerDown`, `KubeProxyDown`	k3s nie wystawia metryk tych komponentów na adresach, które znają domyślne ServiceMonitory	w `kps-values-lab.yaml` wyłączamy `kubeScheduler`, `kubeControllerManager`, `kubeProxy` i `kubeEtcd`
ServiceMonitor istnieje, ale nie ma go w Status → Targets	Prometheus wybiera tylko ServiceMonitory z etykietą swojego wydania	w `prometheusSpec` ustawiamy `serviceMonitorSelectorNilUsesHelmValues: false` albo dodajemy etykietę `release: kps`
Cel istnieje, ale ma status `DOWN` i błąd `connection refused`	port Service nie ma nazwy `metrics` albo `targetPort` nie pasuje do portu kontenera	`kubectl get svc -n kolory kolorowy-nginx -o yaml`, nazwa portu musi się zgadzać z `endpoints.port` w ServiceMonitorze
`nginx_up 0`, w logach eksportera `connection refused` albo `403`	serwer na porcie 8080 nie działa albo `allow` nie pasuje do adresu `127.0.0.1`	`kubectl exec ... -c nginx -- cat /etc/nginx/conf.d/default.conf`, sprawdzamy ConfigMap
Pod nginksa w logach pisze `can not modify /etc/nginx/conf.d/default.conf (read-only file system?)`	skrypt startowy obrazu próbuje zmienić plik z ConfigMap zamontowanego tylko do odczytu	ignorujemy: to komunikat informacyjny, nginx startuje z naszą konfiguracją
Po `helm upgrade` HPA pokazuje `<unknown>`	sidecar bez `requests.cpu`	dopisujemy `requests` w `metrics.resources`
`Error: ... undefined variable "$labels"` w PrometheusRule	wyrażenie `{{ $labels.pod }}` zostało odczytane przez Helma	zapisujemy `{{ "{{ $labels.pod }}" }}`
Zmiana ConfigMap nie zmienia zachowania nginksa	pody nie zostały zrestartowane	adnotacja `checksum/...` w szablonie poda, jak w wykresie 0.3.0
Prometheus traci dane po restarcie poda	brak `storageSpec`, dane w `emptyDir`	wolumen `local-path` w `prometheusSpec.storageSpec`
Zabbix nie widzi Prometheusa pod adresem z Ingressa	Traefik nie ma reguły dla hosta albo adres jest zablokowany dla `mon01`	`curl -I` z `mon01`, `kubectl get ingress -n monitoring`

Ćwiczenia

1. Wyjaśniamy, dlaczego eksporter widzi `stub_status` przez `127.0.0.1`, a Prometheus nie widzi go bezpośrednio.

Odpowiedź

Kontenery jednego poda dzielą jedną przestrzeń sieciową, więc eksporter dosięga nginksa przez `127.0.0.1:8080`. Prometheus jest w innym podzie, więc zapytałby adres poda, a tam serwer na 8080 odpowiada kodem 403, bo `allow` przepuszcza tylko `127.0.0.1`. Prometheus odpytuje więc port 9113 eksportera, a nie `stub_status`.

1. Usuwamy ServiceMonitor poleceniem z fiszki. Kiedy alert `KolorowyNginxBrakMetryk` zmienia stan na Firing i dlaczego nie od razu?

Odpowiedź

Po około pięciu minutach. Wyrażenie `absent(...)` jest prawdziwe od razu po zniknięciu celu, ale reguła ma `for: 5m`, więc przez ten czas alert ma stan Pending. Okres oczekiwania chroni przed alarmami po krótkim zacięciu.

1. Dlaczego do ServiceMonitora wpisaliśmy `port: metrics`, a nie `port: 9113`?

Odpowiedź

ServiceMonitor odwołuje się do portu Service po jego nazwie. Nazwa przeżyje zmianę numeru portu w kontenerze, a numer w manifeście wymusiłby poprawki w dwóch miejscach naraz.

1. Czym różni się `kubectl top pods` od metryki `container_cpu_usage_seconds_total` w Prometheusie?

Odpowiedź

`kubectl top` pokazuje jedną migawkę z metrics-server, bez historii. `container_cpu_usage_seconds_total` to licznik zbierany co kilkadziesiąt sekund i zapamiętywany na trzy dni zgodnie z retencją. Dzięki temu w Prometheusie możemy policzyć średnie i szczyty z przeszłości, a w `kubectl top` tylko stan z ostatniej chwili.

1. Który sposób połączenia z Zabbixem wybieramy dla jednego alarmu „kolorowy nginx obsługuje mniej niż jedno zapytanie na minutę w godzinach pracy” i dlaczego?

Odpowiedź

Sposób B, element HTTP agent z zapytaniem PromQL. To jedna wartość biznesowa z metryki aplikacji. Szablony C pokazują stan klastra, a agent A stan węzła, więc żaden z nich nie zna liczby zapytań nginksa.

Powiązane: Helm: kolorowy nginx jako pakiet · Autoskalowanie: HPA i requests, limits na kolorowym nginx · Diagnostyka: objaw, przyczyna, pierwsze kroki · Architektura Zabbixa: serwer, proxy, agent, frontend · Po co centralne logowanie i jak to działa · GitOps z Argo CD: klaster sam pobiera zmiany z Gitea

Źródła: artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack (91.8.2, aplikacja Prometheus Operator 0.94.1) · github.com/nginx/nginx-prometheus-exporter (v1.5.3 z 14.08.2026, obraz `nginx/nginx-prometheus-exporter:1.5.3`) · prometheus-operator.dev, rozdział o ServiceMonitor i PrometheusRule · zabbix.com/integrations/kubernetes (szablony Kubernetes by HTTP dla Zabbixa 7.0) · grafana.com/docs/loki, rozdziały „Send data” i instalacja Helm w trybie monolitycznym · docs.k3s.io · kubernetes.io/docs/concepts/cluster-administration/logging

---

Revision #17
Created 2026-10-03 09:39:09 UTC by Tomasz Konieczny
Updated 2026-10-26 14:17:22 UTC by Tomasz Konieczny