t

➜ prometheus ccat nginx-ingress.yaml

controller:

kind: DaemonSet

reportNodeInternalIp: true

hostPort:

😊 е🙄 enabled: true

ports:

http: 80

https: 443

service:

type: NodePort

➜ prometheus ccat prometheus.yaml

prometheus:

prometheusSpec:

serviceMonitorSelectorNilUsesHelmValues: false

serviceMonitorSelector: {}

➜ prometheus

➜ prometheus

🌑 "The 'ingress' addon is disabled

➜ prometheus

➜ prometheus kubectl create namespace monitoring

➜ prometheus git:(master) ✗ kubectl config set-context --current --namespace=monitoring

Context "minikube" modified.

Ставим прометей с графаной в качестве оператора.

Добавляем helm репозитории:

➜ prometheus helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

"prometheus-community" has been added to your repositories

➜ prometheus helm repo add stable <https://charts.helm.sh/stable>

"stable" has been added to your repositories

➜ prometheus ✗ helm repo update

Hang tight while we grab the latest from your chart repositories...

...Successfully got an update from the "stable" chart repository

...Successfully got an update from the "prometheus-community" chart repository

Update Complete. ⎈ Happy Helming!⎈

➜ prometheus

Устанавливаем с помощью хелма стек прометеуса

➜ prometheus helm install prom prometheus-community/kube-prometheus-stack -f prometheus.yaml --atomic

manifest\_sorter.go:192: info: skipping unknown hook: "crd-install"

manifest\_sorter.go:192: info: skipping unknown hook: "crd-install"

manifest\_sorter.go:192: info: skipping unknown hook: "crd-install"

manifest\_sorter.go:192: info: skipping unknown hook: "crd-install"

manifest\_sorter.go:192: info: skipping unknown hook: "crd-install"

manifest\_sorter.go:192: info: skipping unknown hook: "crd-install"

manifest\_sorter.go:192: info: skipping unknown hook: "crd-install"

NAME: prom

LAST DEPLOYED: Fri Oct 30 09:08:54 2020

NAMESPACE: monitoring

STATUS: deployed

REVISION: 1

NOTES:

kube-prometheus-stack has been installed. Check its status by running:

kubectl --namespace monitoring get pods -l "release=prom"

Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to create & configure Alertmanager and Prometheus instances using the Operator.

➜ prometheus

Установка обычно занимается некоторое время. После того, как все поставится, можно посмотреть на установленные сущности внутри куба:

➜ prometheus kubectl get all

NAME READY STATUS RESTARTS AGE

pod/alertmanager-prom-kube-prometheus-stack-alertmanager-0 2/2 Running 0 4m52s

pod/prom-grafana-79954d487-czpxc 2/2 Running 0 5m34s

pod/prom-kube-prometheus-stack-operator-686659df96-bmdfz 2/2 Running 0 5m34s

pod/prom-kube-state-metrics-7988bdcf7b-9zq5c 1/1 Running 0 5m34s

pod/prom-prometheus-node-exporter-4bntm 1/1 Running 0 5m34s

pod/prometheus-prom-kube-prometheus-stack-prometheus-0 3/3 Running 1 4m51s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/alertmanager-operated ClusterIP None <none> 9093/TCP,9094/TCP,9094/UDP 4m52s

service/prom-grafana ClusterIP 10.102.15.148 <none> 80/TCP 5m34s

service/prom-kube-prometheus-stack-alertmanager ClusterIP 10.100.176.123 <none> 9093/TCP 5m34s

service/prom-kube-prometheus-stack-operator ClusterIP 10.107.78.107 <none> 8080/TCP,443/TCP 5m34s

service/prom-kube-prometheus-stack-prometheus ClusterIP 10.99.40.112 <none> 9090/TCP 5m34s

service/prom-kube-state-metrics ClusterIP 10.105.66.75 <none> 8080/TCP 5m34s

service/prom-prometheus-node-exporter ClusterIP 10.103.54.136 <none> 9100/TCP 5m34s

service/prometheus-operated ClusterIP None <none> 9090/TCP 4m51s

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

daemonset.apps/prom-prometheus-node-exporter 1 1 1 1 1 <none> 5m34s

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/prom-grafana 1/1 1 1 5m34s

deployment.apps/prom-kube-prometheus-stack-operator 1/1 1 1 5m34s

deployment.apps/prom-kube-state-metrics 1/1 1 1 5m34s

NAME DESIRED CURRENT READY AGE

replicaset.apps/prom-grafana-79954d487 1 1 1 5m34s

replicaset.apps/prom-kube-prometheus-stack-operator-686659df96 1 1 1 5m34s

replicaset.apps/prom-kube-state-metrics-7988bdcf7b 1 1 1 5m34s

NAME READY AGE

statefulset.apps/alertmanager-prom-kube-prometheus-stack-alertmanager 1/1 4m52s

statefulset.apps/prometheus-prom-kube-prometheus-stack-prometheus 1/1 4m51s

➜ prometheus

Устанавливаем через хелм ингресс.

➜ prometheus helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

"ingress-nginx" has been added to your repositories

➜ prometheus helm repo update

Hang tight while we grab the latest from your chart repositories...

...Successfully got an update from the "ingress-nginx" chart repository

Update Complete. ⎈ Happy Helming!⎈

➜ prometheus helm install nginx ingress-nginx/ingress-nginx -f nginx-ingress.yaml --atomic

NAME: nginx

LAST DEPLOYED: Fri Oct 30 09:22:19 2020

NAMESPACE: monitoring

STATUS: deployed

REVISION: 1

TEST SUITE: None

NOTES:

The ingress-nginx controller has been installed.

Get the application URL by running these commands:

export HTTP\_NODE\_PORT=$(kubectl --namespace monitoring get services -o jsonpath="{.spec.ports[0].nodePort}" nginx-ingress-nginx-controller)

export HTTPS\_NODE\_PORT=$(kubectl --namespace monitoring get services -o jsonpath="{.spec.ports[1].nodePort}" nginx-ingress-nginx-controller)

export NODE\_IP=$(kubectl --namespace monitoring get nodes -o jsonpath="{.items[0].status.addresses[1].address}")

echo "Visit http://$NODE\_IP:$HTTP\_NODE\_PORT to access your application via HTTP."

echo "Visit https://$NODE\_IP:$HTTPS\_NODE\_PORT to access your application via HTTPS."

An example Ingress that makes use of the controller:

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

annotations:

kubernetes.io/ingress.class: nginx

name: example

namespace: foo

spec:

rules:

- host: www.example.com

http:

paths:

- backend:

serviceName: exampleService

servicePort: 80

path: /

# This section is only required if TLS is to be enabled for the Ingress

tls:

- hosts:

- www.example.com

secretName: example-tls

If TLS is enabled for the Ingress, a Secret containing the certificate and key must also be provided:

apiVersion: v1

kind: Secret

metadata:

name: example-tls

namespace: foo

data:

tls.crt: <base64 encoded cert>

tls.key: <base64 encoded key>

type: kubernetes.io/tls

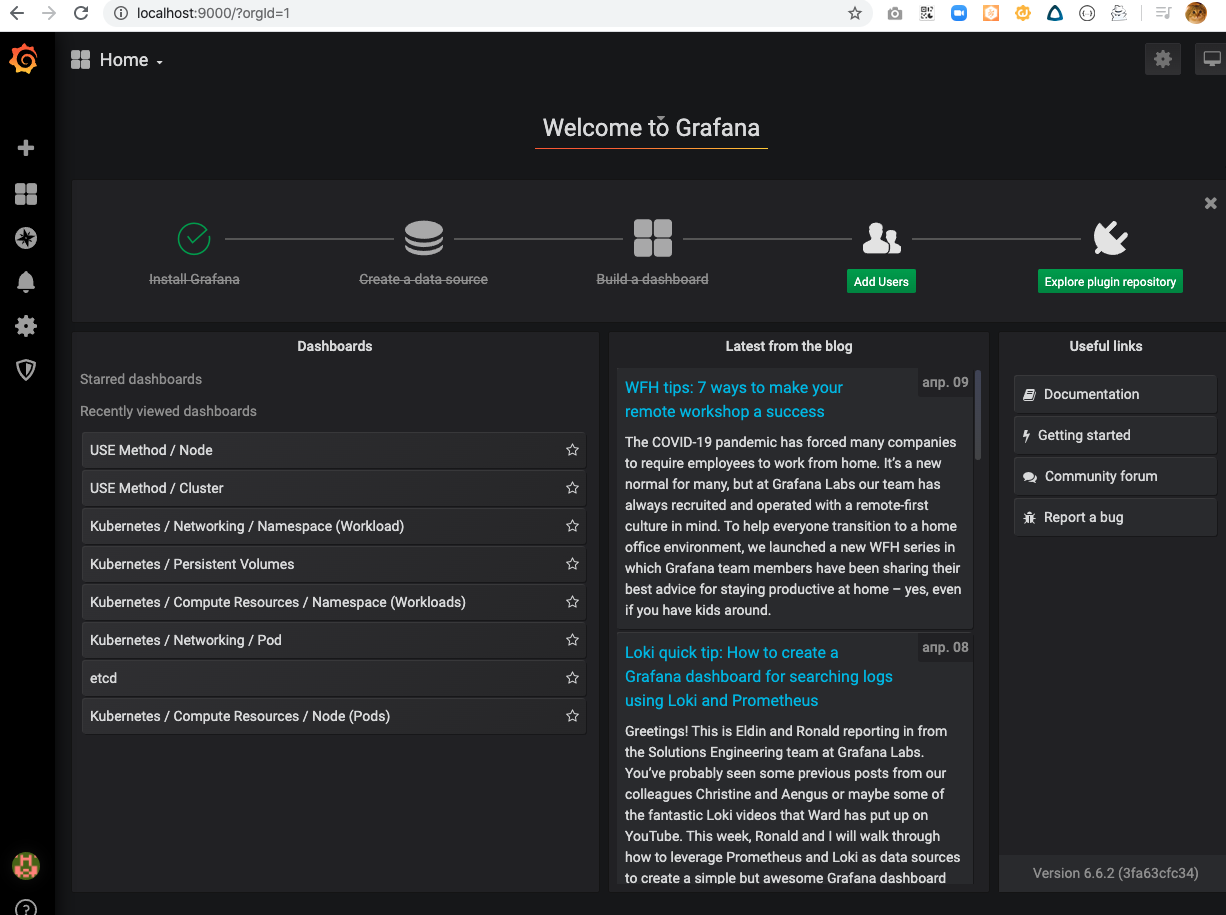
➜ prometheus

Запускаем графану

➜ ~ kubectl port-forward service/prom-grafana 9000:80

Forwarding from 127.0.0.1:9000 -> 3000

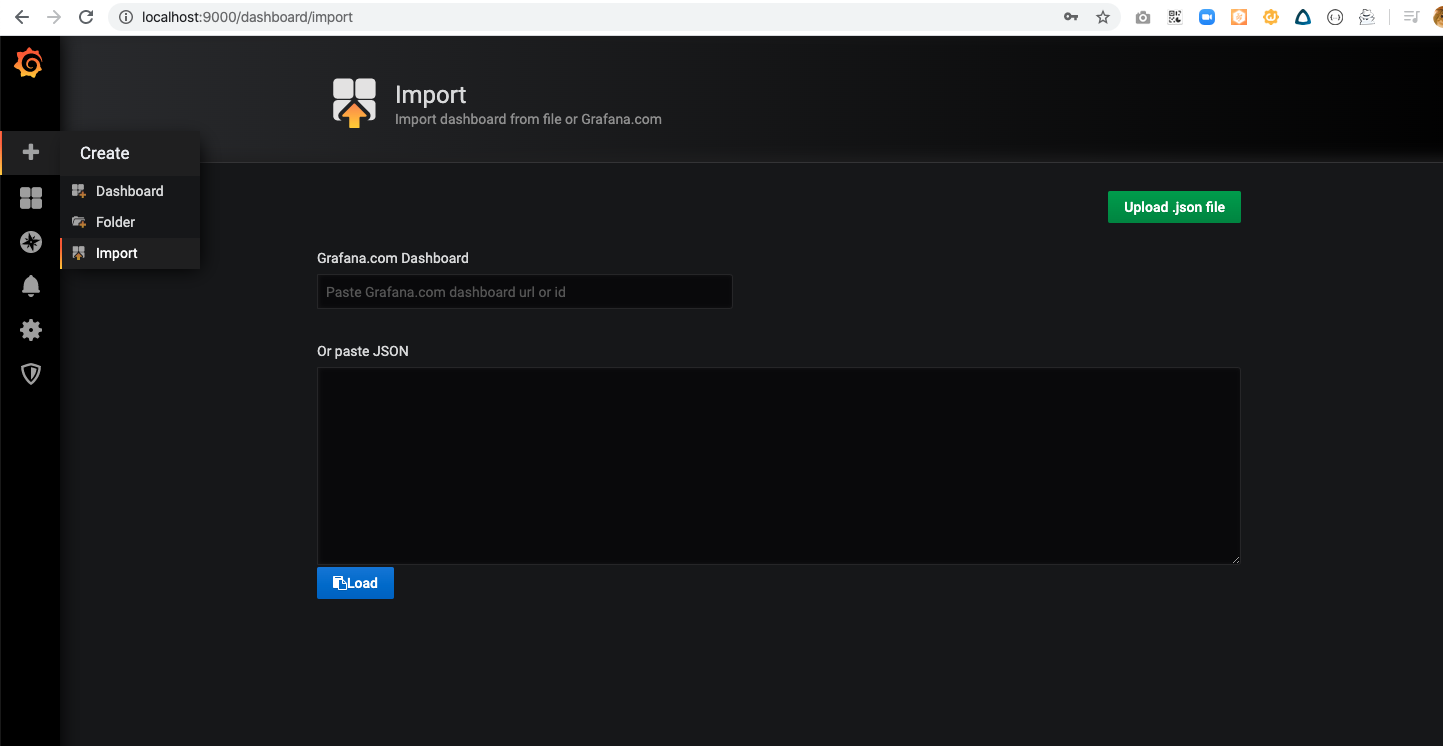
Forwarding from [::1]:9000 -> 3000

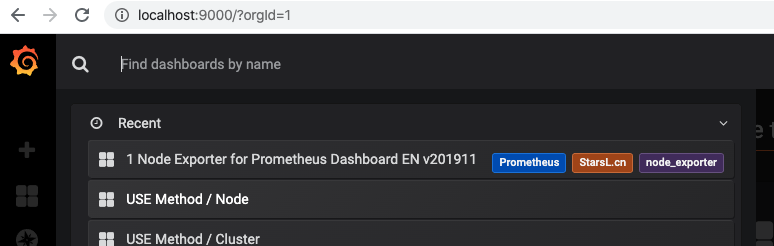


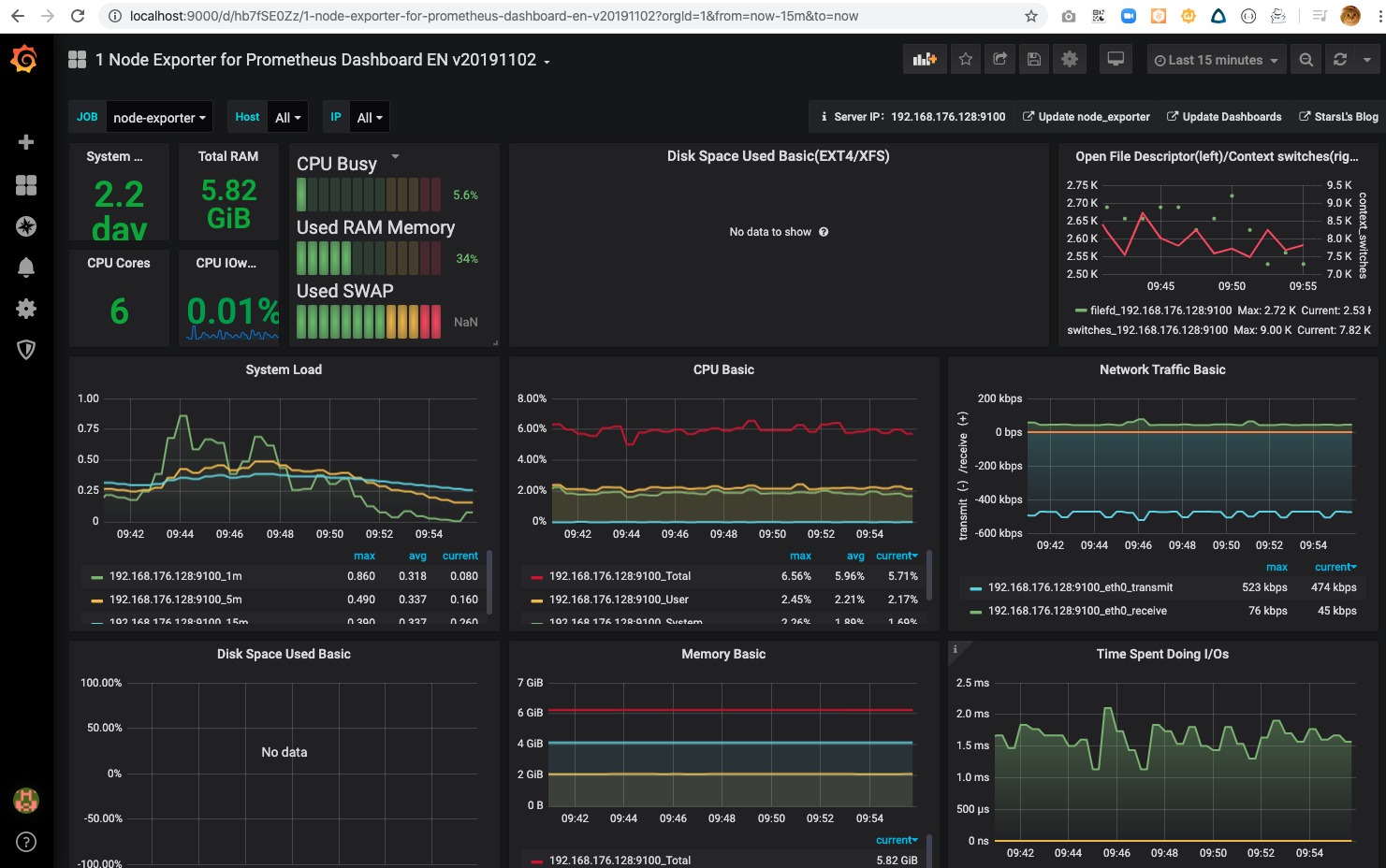
В поставке приложения, уже идут встроенные дашборды, которые мониторят ресурсы кубика.

Но можно симпортировать борду. Давайте симпортируем какую-нибудь. Например, <https://grafana.com/grafana/dashboards/11074>

Симпортируем json борды.





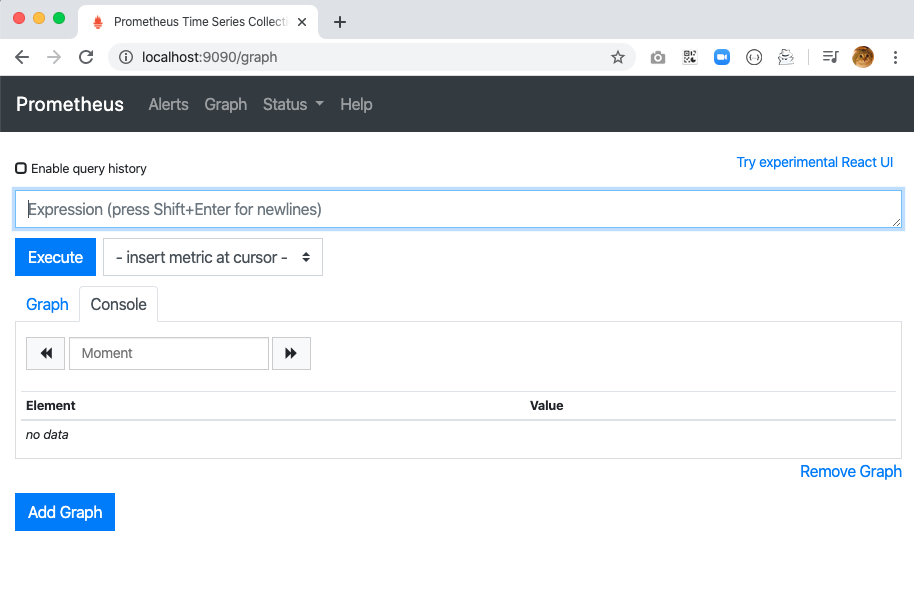


Посмотрим на промтеус. И откуда данные собираются.

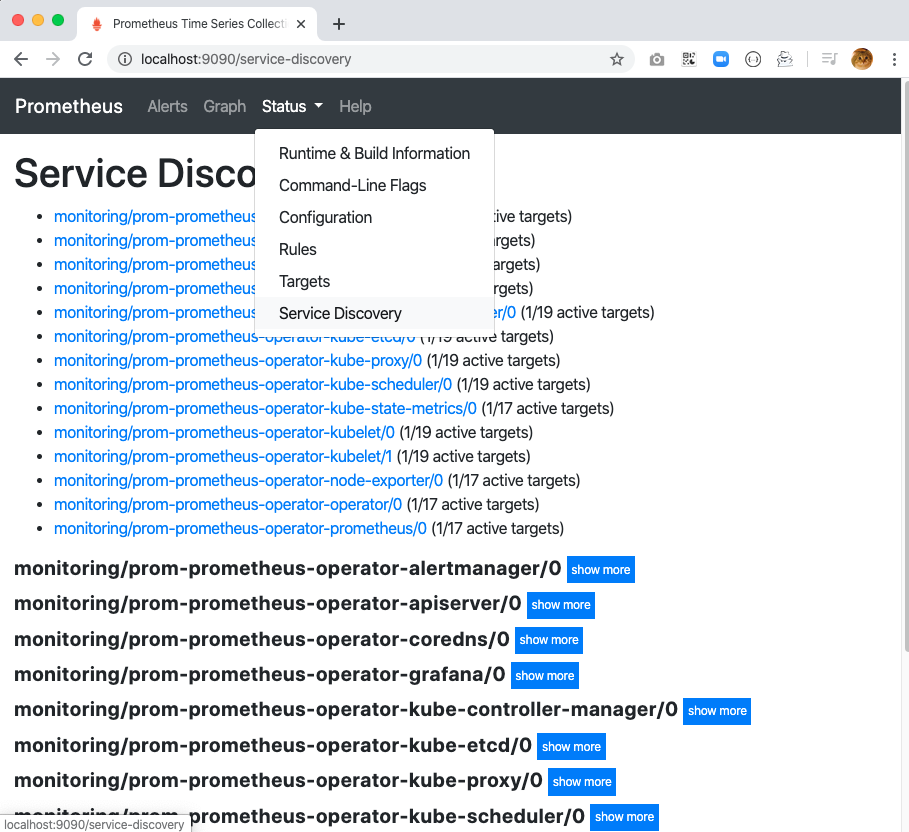
➜ ~ kubectl port-forward service/prom-kube-prometheus-stack-prometheus 9090

Forwarding from 127.0.0.1:9090 -> 9090

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



В service discovery можно посмотреть, какие сервисы мониторятся.



<https://prometheus.io/docs/prometheus/latest/querying/basics/> - тут про promql

<https://robustperception.io/how-does-a-prometheus-counter-work> - про каунтеры почитать

<https://www.youtube.com/watch?v=67Ulrq6DxwA> - видео одного из контрибьюторов prometheus

ДОбавим метрики в наше приложение

Определим, как и какие мы метрики считаем, использую стандартную библиотеку для прометеуса. На каждый запрос считаем LATENCY и REQUEST\_COUNT.

LATENCY - имеет тип гистограмма

REQUEST\_COUNT - это счетчик (counter)

➜ hello-py git:(master) ✗ ccat metrics.py

import time

from prometheus\_client import Counter, Histogram, Info

from flask import request

METRICS\_REQUEST\_LATENCY = Histogram(

"app\_request\_latency\_seconds", "Application Request Latency", ["method", "endpoint"]

)

METRICS\_REQUEST\_COUNT = Counter(

"app\_request\_count",

"Application Request Count",

["method", "endpoint", "http\_status"],

)

METRICS\_INFO = Info("app\_version", "Application Version")

def before\_request():

request.\_prometheus\_metrics\_request\_start\_time = time.time()

def after\_request(response):

request\_latency = time.time() - request.\_prometheus\_metrics\_request\_start\_time

METRICS\_REQUEST\_LATENCY.labels(request.method, request.path).observe(

request\_latency

)

METRICS\_REQUEST\_COUNT.labels(

request.method, request.path, response.status\_code

).inc()

return response

def register\_metrics(app, app\_version=None, app\_config=None):

app.before\_request(before\_request)

app.after\_request(after\_request)

METRICS\_INFO.info({"version": "1", "config": "develop"})

➜ hello-py git:(master) ✗

В самом приложении добавляем путь /metrics, который будет отдавать эти метрики.

➜ hello-py git:(master) ✗ ccat app.py

import os

import json

from flask import Flask

from metrics import register\_metrics

app = Flask(\_\_name\_\_)

config = {

'DATABASE\_URI': os.environ.get('DATABASE\_URI', ''),

'HOSTNAME': os.environ['HOSTNAME'],

'GREETING': os.environ.get('GREETING', 'Hello'),

}

@app.route("/")

def hello():

return config['GREETING'] + ' from ' + config['HOSTNAME'] + '!'

@app.route("/config")

def configuration():

return json.dumps(config)

@app.route('/db')

def db():

from sqlalchemy import create\_engine

engine = create\_engine(config['DATABASE\_URI'], echo=True)

rows = []

with engine.connect() as connection:

result = connection.execute("select id, name from client;")

rows = [dict(r.items()) for r in result]

return json.dumps(rows)

@app.route('/metrics')

def metrics():

from prometheus\_client import generate\_latest

return generate\_latest()

if \_\_name\_\_ == "\_\_main\_\_":

register\_metrics(app)

app.run(host='0.0.0.0', port='80', debug=True)

➜ hello-py git:(master) ✗

Ну и добавим в зависимости библиотеку прометеуса.

➜ hello-py git:(master) ✗ ccat requirements.txt

Flask==1.1.2

Flask-SQLAlchemy

Flask-Migrate

psycopg2

Flask-Script

# Prometheus Python client

prometheus-client==0.7.1

➜ hello-py git:(master) ✗

Собираем новый image

➜ hello-py git:(master) ✗ docker build -t hello-py:0.10.0 .

Sending build context to Docker daemon 67.07kB

Step 1/3 : FROM python:3.5-onbuild

# Executing 3 build triggers

---> Using cache

---> Using cache

---> 2520b7b19a33

Step 2/3 : EXPOSE 8000

---> Running in 513fd2cfe724

Removing intermediate container 513fd2cfe724

---> f33f08362149

Step 3/3 : ENTRYPOINT ["python", "/usr/src/app/app.py"]

---> Running in 07b6c463dea7

Removing intermediate container 07b6c463dea7

---> e26043352c69

Successfully built e26043352c69

Successfully tagged hello-py:0.10.0

➜ hello-py git:(master) ✗

Обновляем helm chart

➜ prometheus ccat hello-chart/Chart.yaml

apiVersion: v2

name: hello-chart

description: A Helm chart for Kubernetes

type: application

version: 0.4.0

appVersion: 0.10.0

dependencies:

- name: postgresql

version: 8.x.x

repository: https://charts.bitnami.com/bitnami

condition: postgresql.enabled

tags:

- myapp-database

➜ prometheus

И устанавливаем приложение

➜ prometheus helm install myapp ./hello-chart --atomic

NAME: myapp

LAST DEPLOYED: Sat Apr 11 14:41:02 2020

NAMESPACE: monitoring

STATUS: deployed

REVISION: 1

NOTES:

TO BE DONE

Посмотрим, какие создались ресурсы внутри куба

➜ prometheus kubectl get all -l app.kubernetes.io/instance=myapp

NAME READY STATUS RESTARTS AGE

pod/myapp-hello-chart-5cdf58458c-7q8c5 1/1 Running 0 102s

pod/myapp-hello-chart-5cdf58458c-gslzs 1/1 Running 0 102s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

service/myapp-hello-chart NodePort 10.102.54.151 <none> 9000:32033/TCP 102s

NAME READY UP-TO-DATE AVAILABLE AGE

deployment.apps/myapp-hello-chart 2/2 2 2 102s

NAME DESIRED CURRENT READY AGE

replicaset.apps/myapp-hello-chart-5cdf58458c 2 2 2 102s

➜ prometheus

➜ prometheus helm list | grep myapp

myapp monitoring 1 2020-04-11 14:41:02.101815 +0300 MSK deployed hello-chart-0.4.0 0.10.0

➜ prometheus

Проверим, что приложение корректно работает

➜ ~ curl -s http://192.168.176.128:32033/db | jq

[

{

"id": 1,

"name": "Konstantin"

}

]

➜ ~

Посмотрим, в каком формате отдаются метрики:

➜ ~ curl -s http://192.168.176.128:32033/metrics

# HELP app\_request\_count\_total Application Request Count

# TYPE app\_request\_count\_total counter

app\_request\_count\_total{endpoint="/metrics",http\_status="200",method="GET"} 20.0

app\_request\_count\_total{endpoint="/db",http\_status="200",method="GET"} 1.0

# TYPE app\_request\_count\_created gauge

app\_request\_count\_created{endpoint="/metrics",http\_status="200",method="GET"} 1.5866052720351672e+09

app\_request\_count\_created{endpoint="/db",http\_status="200",method="GET"} 1.5866054782892115e+09

# HELP python\_info Python platform information

# TYPE python\_info gauge

python\_info{implementation="CPython",major="3",minor="5",patchlevel="5",version="3.5.5"} 1.0

# HELP python\_gc\_objects\_collected\_total Objects collected during gc

# TYPE python\_gc\_objects\_collected\_total counter

python\_gc\_objects\_collected\_total{generation="0"} 458.0

python\_gc\_objects\_collected\_total{generation="1"} 310.0

python\_gc\_objects\_collected\_total{generation="2"} 0.0

# HELP python\_gc\_objects\_uncollectable\_total Uncollectable object found during GC

# TYPE python\_gc\_objects\_uncollectable\_total counter

python\_gc\_objects\_uncollectable\_total{generation="0"} 0.0

python\_gc\_objects\_uncollectable\_total{generation="1"} 0.0

python\_gc\_objects\_uncollectable\_total{generation="2"} 0.0

# HELP python\_gc\_collections\_total Number of times this generation was collected

# TYPE python\_gc\_collections\_total counter

python\_gc\_collections\_total{generation="0"} 108.0

python\_gc\_collections\_total{generation="1"} 9.0

python\_gc\_collections\_total{generation="2"} 0.0

# HELP process\_virtual\_memory\_bytes Virtual memory size in bytes.

# TYPE process\_virtual\_memory\_bytes gauge

process\_virtual\_memory\_bytes 3.30797056e+08

# HELP process\_resident\_memory\_bytes Resident memory size in bytes.

# TYPE process\_resident\_memory\_bytes gauge

process\_resident\_memory\_bytes 3.883008e+07

# HELP process\_start\_time\_seconds Start time of the process since unix epoch in seconds.

# TYPE process\_start\_time\_seconds gauge

process\_start\_time\_seconds 1.58660526249e+09

# HELP process\_cpu\_seconds\_total Total user and system CPU time spent in seconds.

# TYPE process\_cpu\_seconds\_total counter

process\_cpu\_seconds\_total 1.73

# HELP process\_open\_fds Number of open file descriptors.

# TYPE process\_open\_fds gauge

process\_open\_fds 10.0

# HELP process\_max\_fds Maximum number of open file descriptors.

# TYPE process\_max\_fds gauge

process\_max\_fds 1.048576e+06

# HELP app\_request\_latency\_seconds Application Request Latency

# TYPE app\_request\_latency\_seconds histogram

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.005",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.01",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.025",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.05",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.075",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.1",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.25",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.5",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="0.75",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="1.0",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="2.5",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="5.0",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="7.5",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="10.0",method="GET"} 20.0

app\_request\_latency\_seconds\_bucket{endpoint="/metrics",le="+Inf",method="GET"} 20.0

app\_request\_latency\_seconds\_count{endpoint="/metrics",method="GET"} 20.0

app\_request\_latency\_seconds\_sum{endpoint="/metrics",method="GET"} 0.03050398826599121

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.005",method="GET"} 0.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.01",method="GET"} 0.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.025",method="GET"} 0.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.05",method="GET"} 0.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.075",method="GET"} 0.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.1",method="GET"} 0.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.25",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.5",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="0.75",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="1.0",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="2.5",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="5.0",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="7.5",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="10.0",method="GET"} 1.0

app\_request\_latency\_seconds\_bucket{endpoint="/db",le="+Inf",method="GET"} 1.0

app\_request\_latency\_seconds\_count{endpoint="/db",method="GET"} 1.0

app\_request\_latency\_seconds\_sum{endpoint="/db",method="GET"} 0.11905932426452637

# TYPE app\_request\_latency\_seconds\_created gauge

app\_request\_latency\_seconds\_created{endpoint="/metrics",method="GET"} 1.5866052720350592e+09

app\_request\_latency\_seconds\_created{endpoint="/db",method="GET"} 1.5866054782891264e+09

# HELP app\_version\_info Application Version

# TYPE app\_version\_info gauge

app\_version\_info{config="develop",version="1"} 1.0

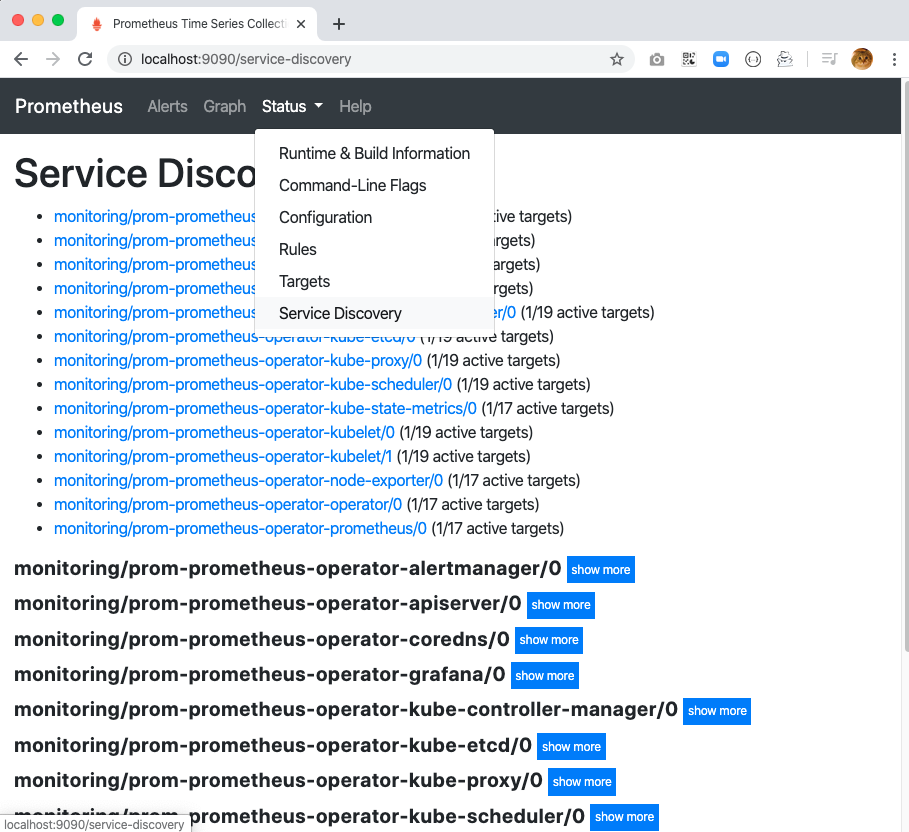
➜ ~

Если посмотреть в прометеус, то там метрик не будет.

➜ ~ kubectl port-forward service/prom-prometheus-operator-prometheus 9090

Forwarding from 127.0.0.1:9090 -> 9090

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



Чтобы метрики появились, мы должны добавить servicemonitor, который бы подсказал прометеусу что надо мониторить наше приложение и собирать метрики, и также рассказал куда ходить за метриками.

Напишем манифест сервис-монитора сразу в шаблоны

В нем важно указать метрики, по которым он будет искать сервис и по какому эндпоинту их искать и с какой частотой забирать данные.

➜ prometheus ccat hello-chart/templates/servicemonitor.yaml

{{- if .Values.metrics.serviceMonitor.enabled }}

apiVersion: monitoring.coreos.com/v1

kind: ServiceMonitor

metadata:

name: {{ include "hello-chart.fullname" . }}

labels:

{{- include "hello-chart.labels" . | nindent 4 }}

spec:

jobLabel: {{ include "hello-chart.fullname" . }}

namespaceSelector:

matchNames:

- "{{ $.Release.Namespace }}"

selector:

matchLabels:

{{- include "hello-chart.selectorLabels" . | nindent 6 }}

endpoints:

- interval: 15s

port: web

path: /metrics

{{- end }}

➜ prometheus

Апгрейдим релиз

➜ prometheus helm upgrade myapp ./hello-chart --atomic

Release "myapp" has been upgraded. Happy Helming!

NAME: myapp

LAST DEPLOYED: Sat Apr 11 14:56:29 2020

NAMESPACE: monitoring

STATUS: deployed

REVISION: 2

NOTES:

TO BE DONE

➜ prometheus

Смотрим, что сервис-монитор создался

➜ prometheus kubectl get servicemonitors.monitoring.coreos.com

NAME AGE

myapp-hello-chart 16m

prom-prometheus-operator-alertmanager 29h

prom-prometheus-operator-apiserver 29h

prom-prometheus-operator-coredns 29h

prom-prometheus-operator-grafana 29h

prom-prometheus-operator-kube-controller-manager 29h

prom-prometheus-operator-kube-etcd 29h

prom-prometheus-operator-kube-proxy 29h

prom-prometheus-operator-kube-scheduler 29h

prom-prometheus-operator-kube-state-metrics 29h

prom-prometheus-operator-kubelet 29h

prom-prometheus-operator-node-exporter 29h

prom-prometheus-operator-operator 29h

prom-prometheus-operator-prometheus 29h

➜ prometheus

И даже можем посмотреть, что внутри

➜ prometheus kubectl describe servicemonitors.monitoring.coreos.com myapp-hello-chart

Name: myapp-hello-chart

Namespace: monitoring

Labels: app.kubernetes.io/instance=myapp

app.kubernetes.io/managed-by=Helm

app.kubernetes.io/name=hello-chart

app.kubernetes.io/version=0.10.0

helm.sh/chart=hello-chart-0.4.0

Annotations: <none>

API Version: monitoring.coreos.com/v1

Kind: ServiceMonitor

Metadata:

Creation Timestamp: 2020-04-11T11:41:02Z

Generation: 1

Managed Fields:

API Version: monitoring.coreos.com/v1

Fields Type: FieldsV1

fieldsV1:

f:metadata:

f:labels:

.:

f:app.kubernetes.io/instance:

f:app.kubernetes.io/managed-by:

f:app.kubernetes.io/name:

f:app.kubernetes.io/version:

f:helm.sh/chart:

f:spec:

.:

f:endpoints:

f:jobLabel:

f:namespaceSelector:

.:

f:matchNames:

f:selector:

.:

f:matchLabels:

.:

f:app.kubernetes.io/instance:

f:app.kubernetes.io/name:

Manager: Go-http-client

Operation: Update

Time: 2020-04-11T11:41:02Z

Resource Version: 575739

Self Link: /apis/monitoring.coreos.com/v1/namespaces/monitoring/servicemonitors/myapp-hello-chart

UID: 7a929453-f765-44c8-a4e3-e9c955026778

Spec:

Endpoints:

Interval: 15s

Path: /metrics

Port: web

Job Label: myapp-hello-chart

Namespace Selector:

Match Names:

monitoring

Selector:

Match Labels:

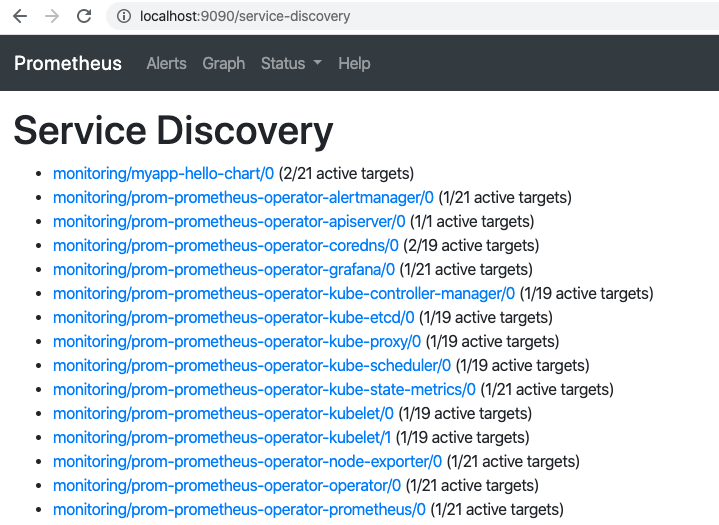
app.kubernetes.io/instance: myapp

app.kubernetes.io/name: hello-chart

Events: <none>

➜ prometheus

Через некоторое время (минуту или около того) прометей подхватывает наше приложение и мы видим цели



И метрики приложения появились

Создадим нагрузку на сервис с помощью ab Будем делать по 50 запросов в 5 одновременных соединений примерно раз в 3 секунды. И оставим на некоторое время.

➜ ~ while 1; do ab -n 50 -c 5 http://192.168.176.128:32033/db ; sleep 3; done

~

This is ApacheBench, Version 2.3 <$Revision: 1843412 $>

Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/

Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking 192.168.176.128 (be patient).....done

Server Software: Werkzeug/1.0.1

Server Hostname: 192.168.176.128

Server Port: 32033

Document Path: /db

Document Length: 33 bytes

Concurrency Level: 5

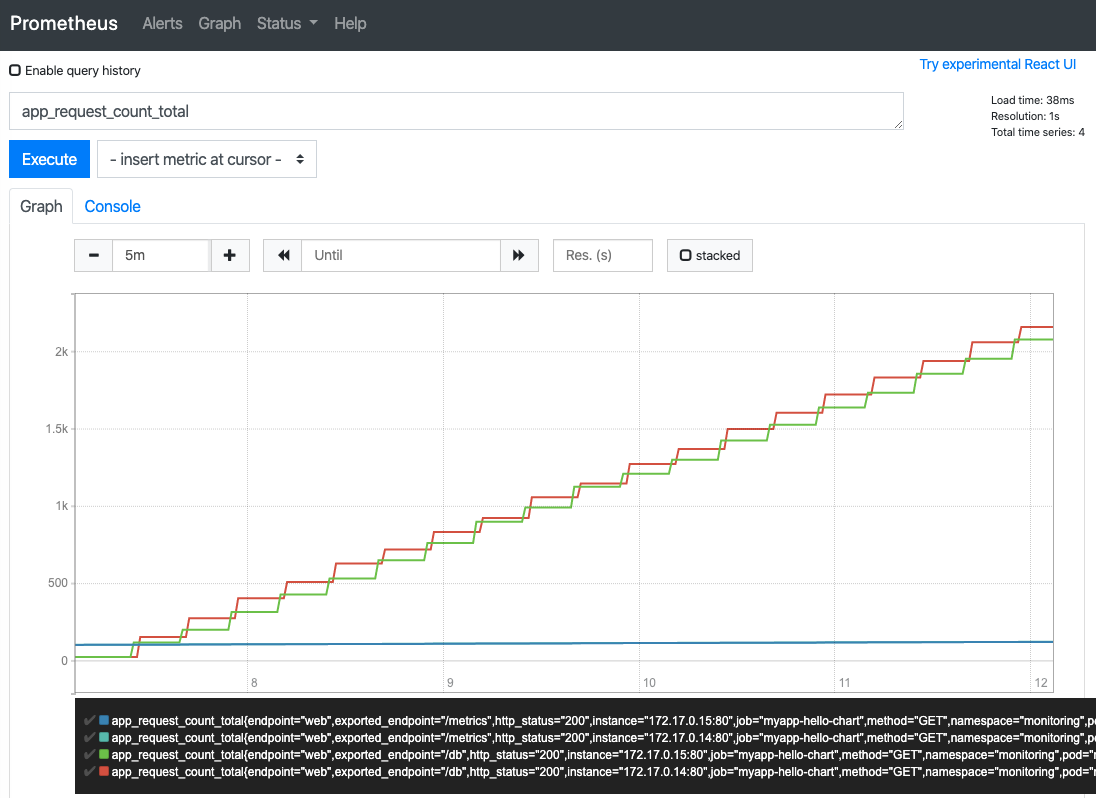
Time taken for tests: 0.303 seconds

Complete requests: 50

Посмотрим, как выглядят timeseries в моменте

Можно увидеть, что прометей добавил еще и своих лейблов, помимо тех, которые мы в самом приложении прописали: namespace, service и т.д. - это бывает полезно. А label endpoint он переименовал в exported\_endpoint.

Давайте посмотрим на графике, как будет выглядеть все

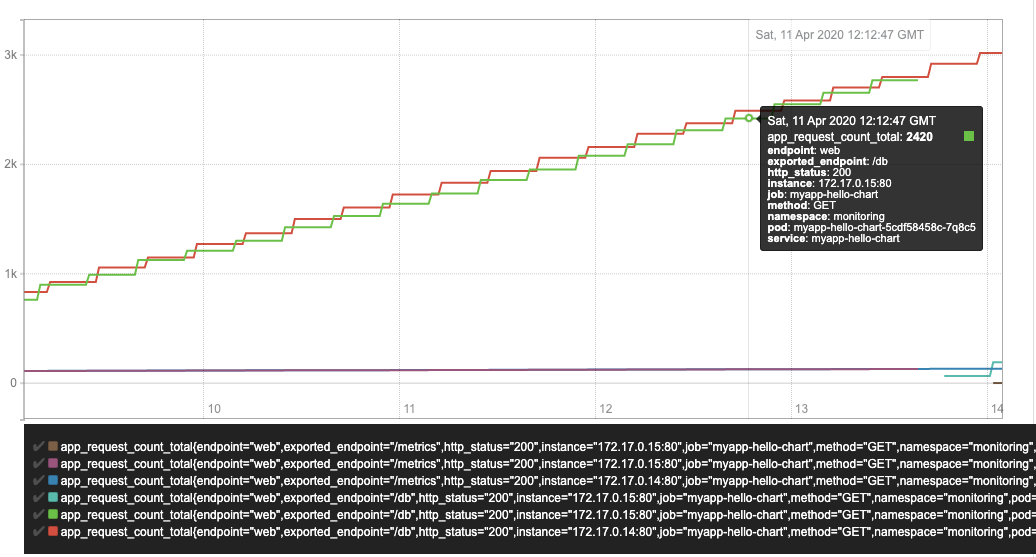


Так выглядит counter. Давайте убьем один из подов.

➜ prometheus kubectl delete pod/myapp-hello-chart-5cdf58458c-7q8c5

+pod "myapp-hello-chart-5cdf58458c-7q8c5" deleted

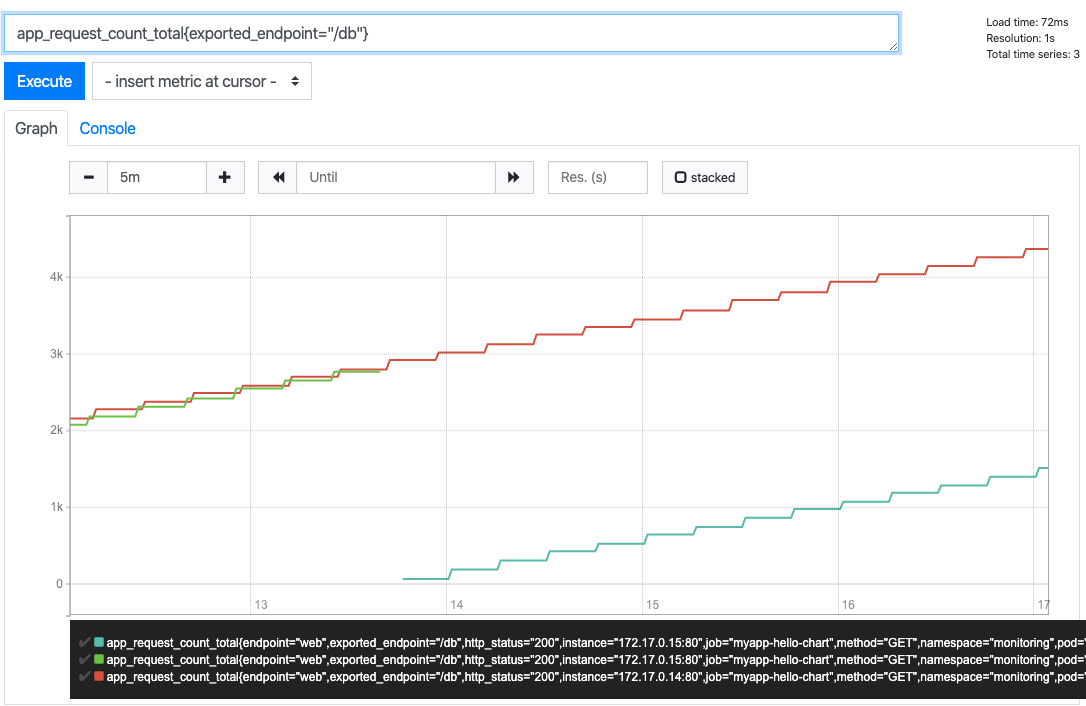
➜ prometheus



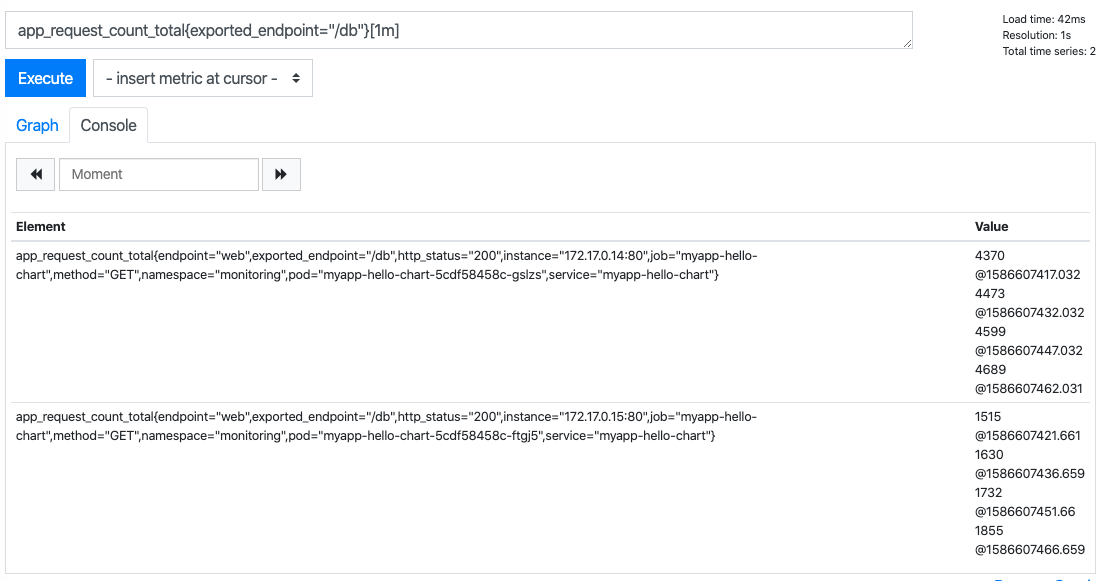
Появились новые графики, cardinality увеличился.

Давайте посмотрим запросы только по db.

app\_request\_count\_total{exported\_endpoint="/db"}



А теперь посмотрим, как выглядит range vector: это просто ts, у которого значение - не последнее значение, а набор из значение@timestamp.



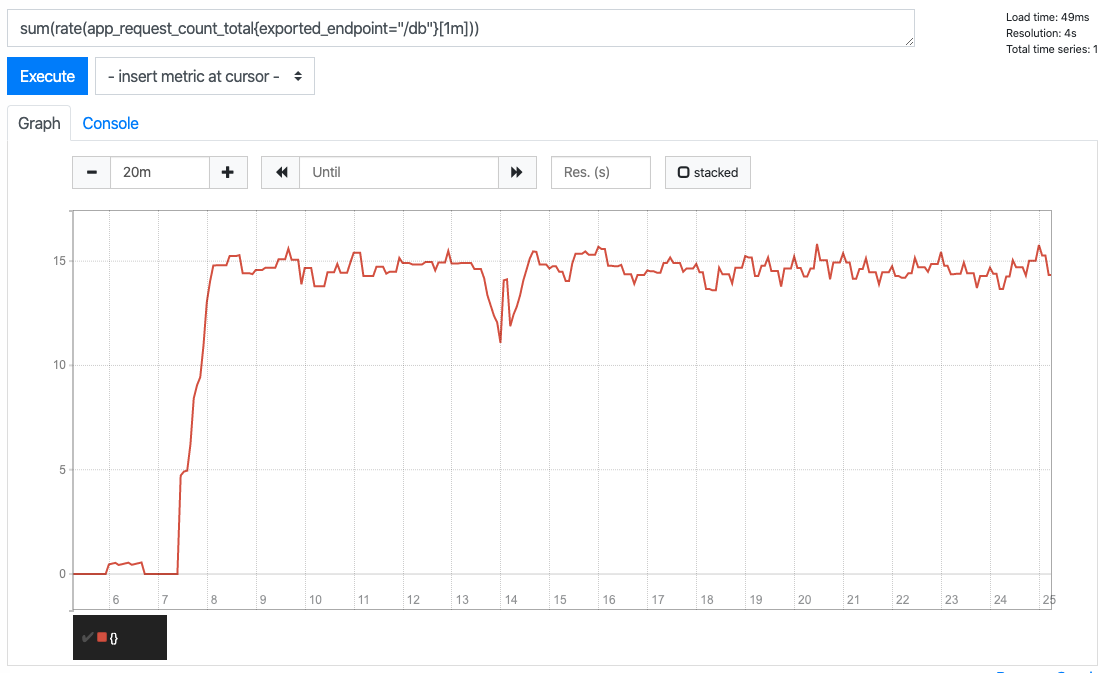
Нарисовать такой график нельзя, но можно посчитать например rps c помощью функции rate - она посчитает среднее значение за указанный период в range vector

rate(app\_request\_count\_total{exported\_endpoint="/db"}[1m])



Мы видим rps-ы по каждому инстансу, чтобы получить общий rps их нужно сложить:

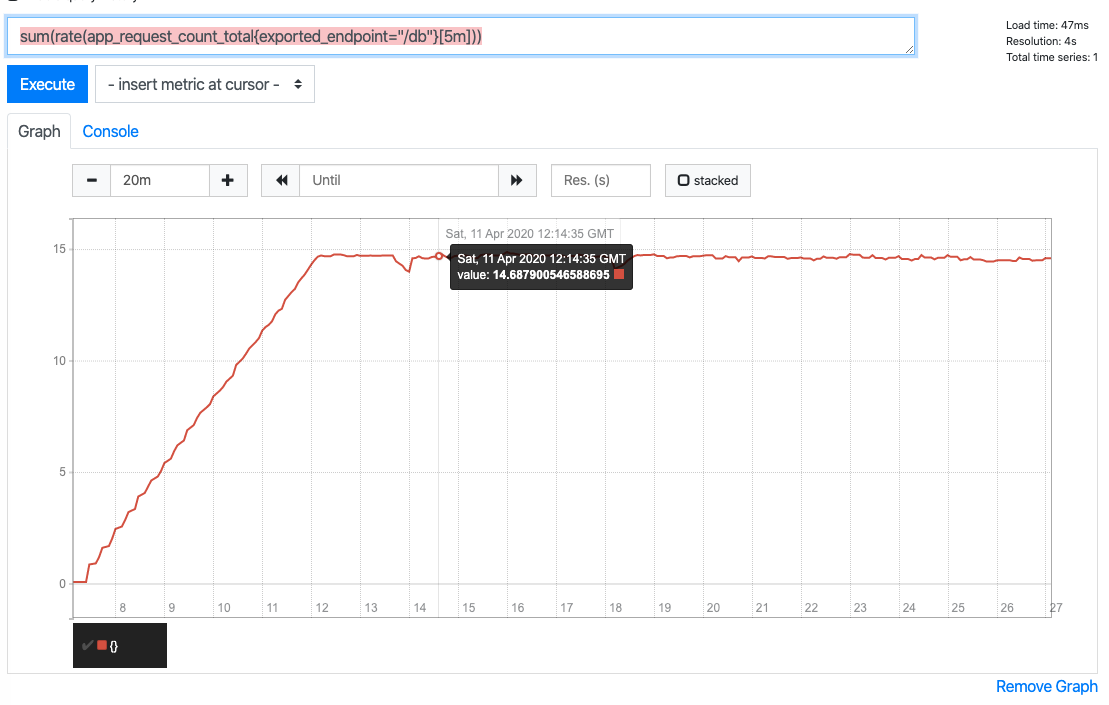
sum(rate(app\_request\_count\_total{exported\_endpoint="/db"}[1m]))



Значение в районе 15 очень похоже на правду (50 / 3.3) 3с секунды ожидание + 300ms на совершение запросов.

Если возьмем range побольше, график сгладится

sum(rate(app\_request\_count\_total{exported\_endpoint="/db"}[5m]))



Важно понимать, что значение точки на графике - это не моментальное значение rps - а среднее значение за какой-то период (1m, 5m и т.д.)

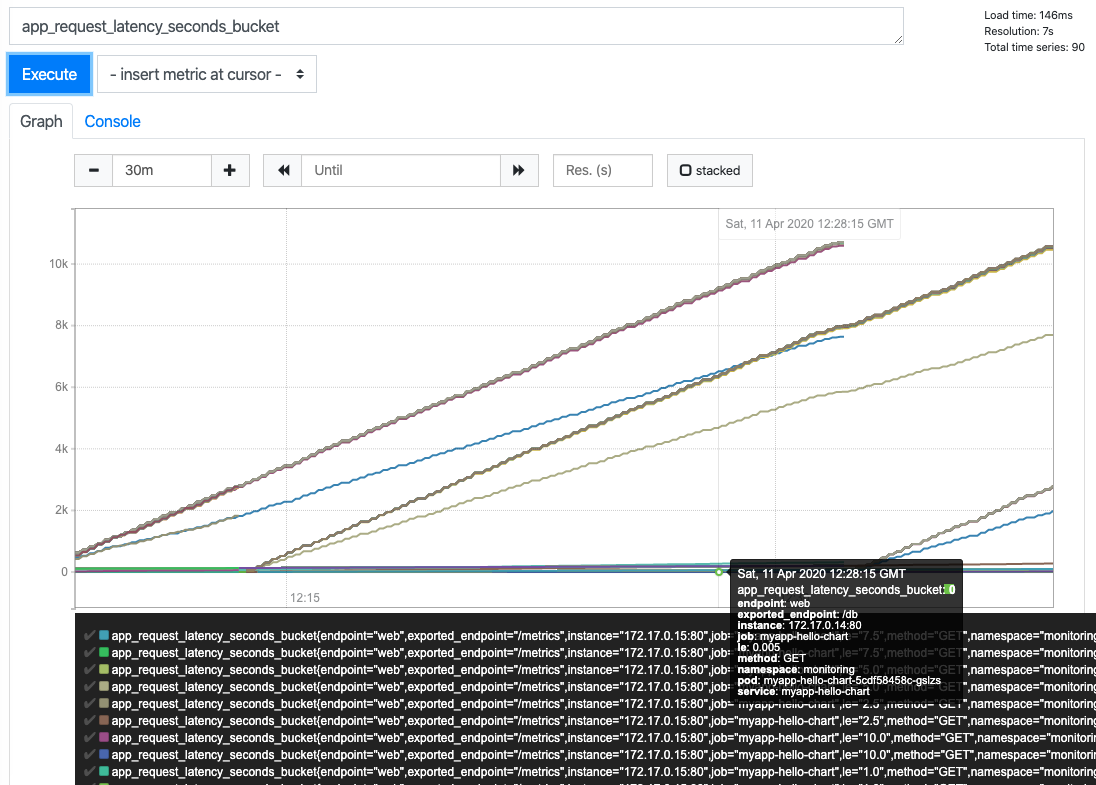
Если мы хотим значение в моменте, можем использовать irate - он берет среднее из последних двух точек в range vector. Тогда график получается больше похожим на забор

irate(app\_request\_count\_total{exported\_endpoint="/db"}[5m])



Посмотрим, как посчитать квантили по временам запроса.

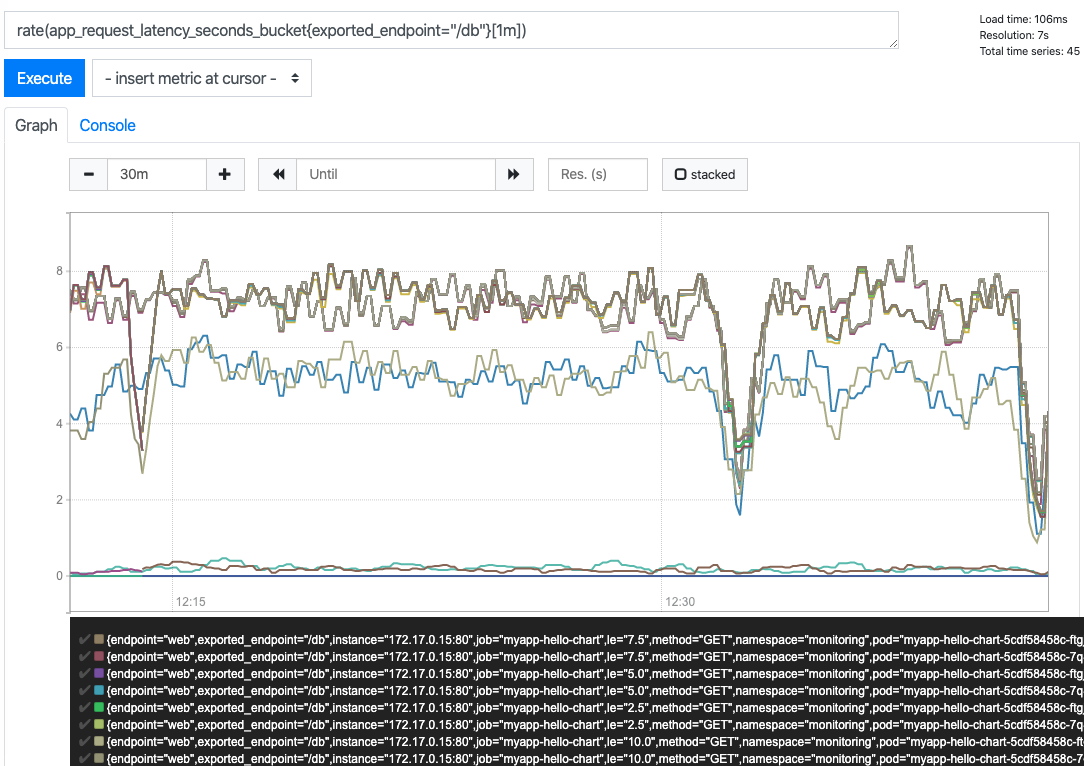
app\_request\_latency\_seconds\_bucket



Видим, что бакеты - это лишь набор сounter-ов, с разными лейблами (le, например).

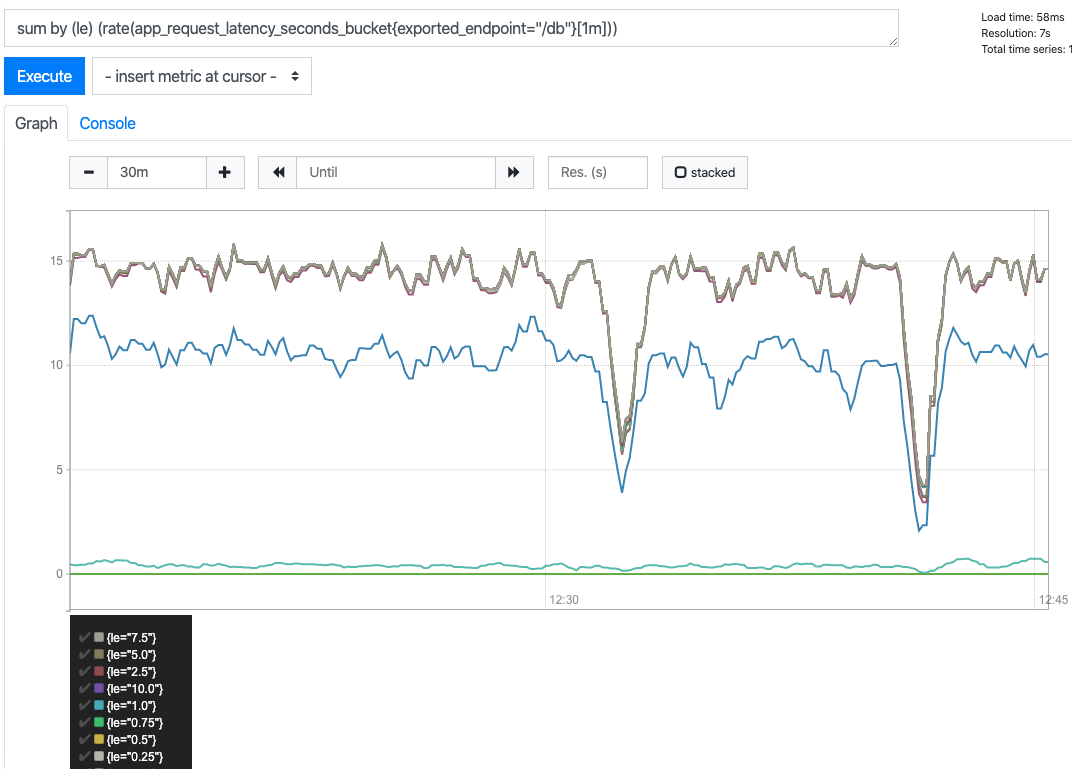
Давайте посчитаем, с какой частотой в какой бакет попадют значения. По факту мы будем считать сколько в среднем за секунду появляется запросов, которые меньше 100мс, меньше 1с, и т.д.

rate(app\_request\_latency\_seconds\_bucket{exported\_endpoint="/db"}[1m])



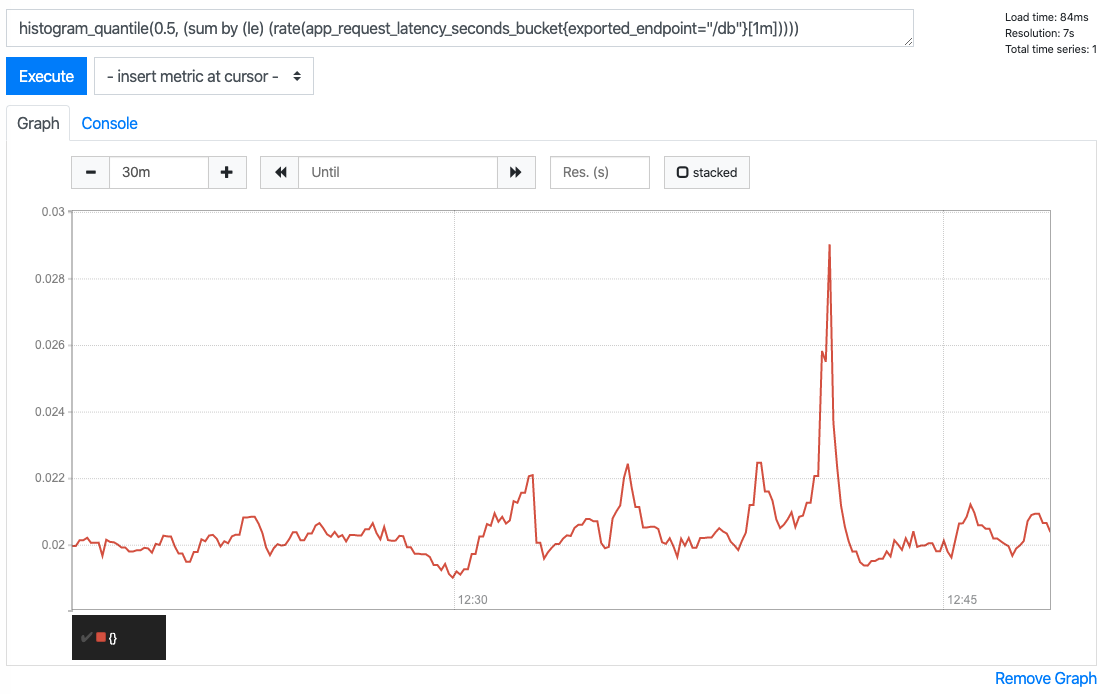
Просуммируем и сгруппируем по le и получим “rps” по каждому из бакетов

sum by (le) (rate(app\_request\_latency\_seconds\_bucket{exported\_endpoint="/db"}[1m]))

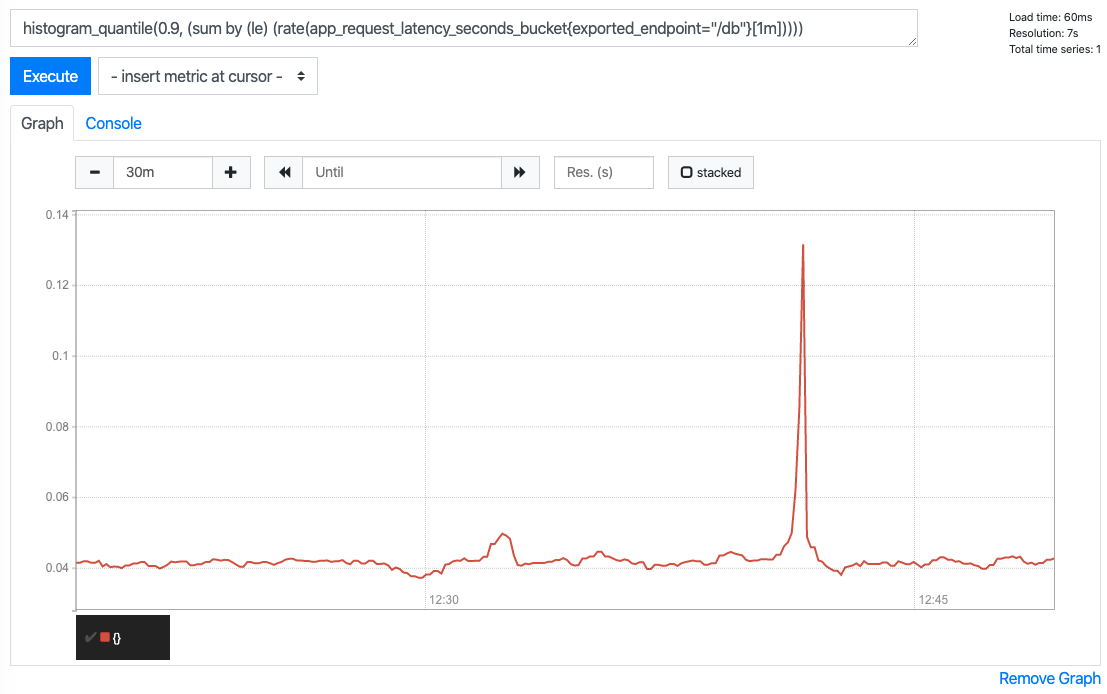


Посчитаем медианное значение с помощью функции

histogram\_quantile(0.5, (sum by (le) (rate(app\_request\_latency\_seconds\_bucket{exported\_endpoint="/db"}[1m]))))



Или давайте посмотрим 90 квантиль



Видим, что латенси ведет себя не очень хорошо. Давайте посмотрим еще раз на rps



Посмотрим, все ли хорошо с подами. Видим что поды уже по 2 раза рестартились.

➜ prometheus kubectl get pods -l app.kubernetes.io/instance=myapp

NAME READY STATUS RESTARTS AGE

myapp-hello-chart-5cdf58458c-ftgj5 1/1 Running 2 51m

myapp-hello-chart-5cdf58458c-gslzs 1/1 Running 2 84m

➜ prometheus

➜ prometheus kubectl describe pod/myapp-hello-chart-5cdf58458c-ftgj5

Name: myapp-hello-chart-5cdf58458c-ftgj5

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal Scheduled <unknown> default-scheduler Successfully assigned monitoring/myapp-hello-chart-5cdf58458c-ftgj5 to minikube

Normal Pulled 8s (x3 over 50m) kubelet, minikube Container image "hello-py:0.10.0" already present on machine

Normal Created 8s (x3 over 50m) kubelet, minikube Created container hello-chart

Normal Started 8s (x3 over 50m) kubelet, minikube Started container hello-chart

Падение рпс происходит из-за рестартов поды.

Как посмотреть рестарты поды? При рестарте поды происходит reset counter-а он начинается с нуля, и есть специальная функция, которая может это отслеживать.

resets(app\_request\_count\_total{exported\_endpoint="/db"}[1m])



Видно, что рестарты совпадают с падениями рпс



## Давайте теперь выведем эти графики в графану.

➜ ~ kubectl port-forward prom-grafana-7ccf6577f8-8dnm2 9000

Forwarding from 127.0.0.1:9000 -> 9000

Forwarding from [::1]:9000 -> 9000

Добавляем дашборд

Выведем RPS по методам:

sum by (exported\_endpoint) (rate(app\_request\_count\_total[1m]))

В легендах можно использовать переменные из запросов RPS {{exported\_endpoint}}



Видим, что есть трафик по ручке /metrics, в которую ходит прометеус.

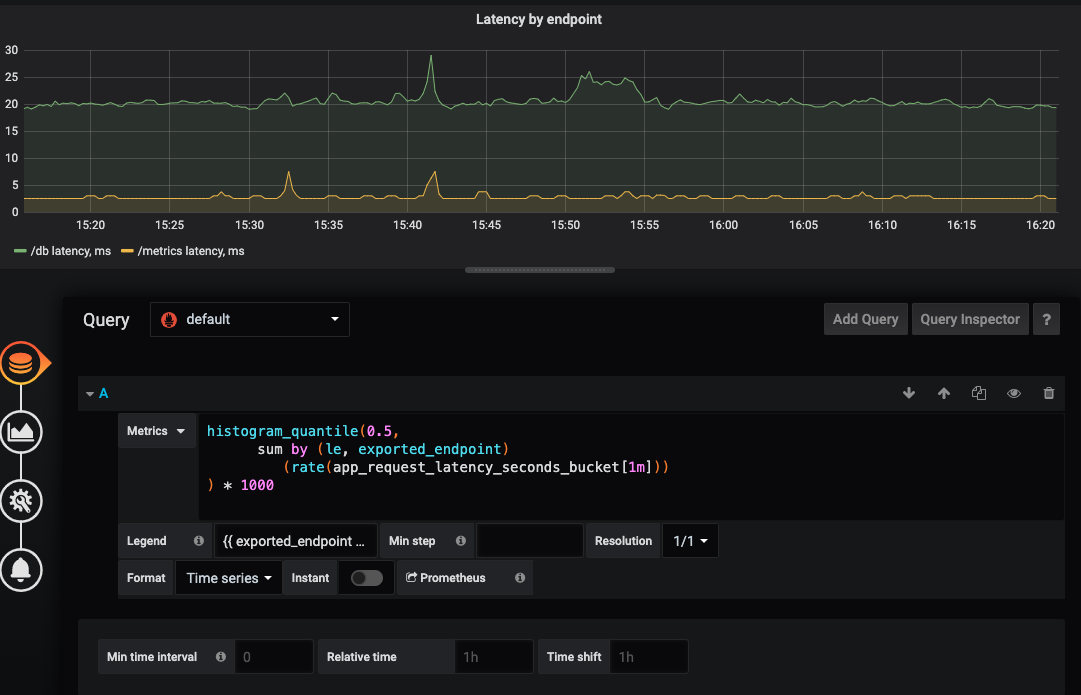
Построим график медианного latency по методам

histogram\_quantile(0.5,

sum by (le, exported\_endpoint)

(rate(app\_request\_latency\_seconds\_bucket[1m]))

) \* 1000



Кстати это время хорошо согласуется с тем, что отдает ab

Connection Times (ms)

min mean[+/-sd] median max

Connect: 0 0 0.2 0 1

Processing: 9 23 8.5 21 50

Waiting: 9 23 8.4 20 50

Total: 9 24 8.5 21 50

Т.е. даже приближенные вычисления квантиля хорошо согласуются с реальным положением дел.

## Давайте сделаем импорт дашборд с помощью конфигмапа

➜ prometheus ccat prometheus.yaml

prometheus:

prometheusSpec:

serviceMonitorSelectorNilUsesHelmValues: false

serviceMonitorSelector: {}

grafana:

sidecar:

dashboards:

enabled: true

label: grafana-dashboard

➜ prometheus ccat grafana.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: grafana-import-dashboards

labels:

grafana-dashboard: "1"

data:

grafana-net-2-dashboard.json: |

{

"annotations": {

"list": [

{

"builtIn": 1,

"datasource": "-- Grafana --",

"enable": true,

…

➜ prometheus kubectl apply -f grafana.yaml

configmap/grafana-import-dashboards created

После этого дашборд появится магическим образом в Графане

## Метрики с Ingress-ов

Давайте добавим в шаблон ingress

➜ prometheus ccat hello-chart/templates/ingress.yaml

{{- if .Values.ingress.enabled -}}

{{- $fullName := include "hello-chart.fullname" . -}}

{{- $svcPort := .Values.service.port -}}

{{- if semverCompare ">=1.14-0" .Capabilities.KubeVersion.GitVersion -}}

apiVersion: networking.k8s.io/v1beta1

{{- else -}}

apiVersion: extensions/v1beta1

{{- end }}

kind: Ingress

metadata:

name: {{ $fullName }}

labels:

{{- include "hello-chart.labels" . | nindent 4 }}

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /$2

spec:

rules:

{{- range .Values.ingress.hosts }}

- host: {{ .host | quote }}

http:

paths:

{{- range .paths }}

- path: {{ . }}($|/)(.\*)

backend:

serviceName: {{ $fullName }}

servicePort: {{ $svcPort }}

{{- end }}

{{- end }}

{{- end }}

➜ prometheus

Обновим values и версию. Повесим на всякий случай на урл /app, а не /myapp.

➜ prometheus ccat hello-chart/values.yaml

replicaCount: 2

image:

repository: hello-py

service:

type: NodePort

port: 9000

postgresql:

enabled: true

postgresqlUsername: myuser

postgresqlPassword: passwd

postgresqlDatabase: myapp

service:

port: "5432"

metrics:

serviceMonitor:

enabled: true

ingress:

enabled: true

hosts:

- host: hello.world

paths: ["/app"]

➜ prometheus

➜ prometheus ccat hello-chart/Chart.yaml

apiVersion: v2

name: hello-chart

description: A Helm chart for Kubernetes

type: application

version: 0.5.0

appVersion: 0.10.0

dependencies:

- name: postgresql

version: 8.x.x

repository: https://charts.bitnami.com/bitnami

condition: postgresql.enabled

tags:

- myapp-database

➜ prometheus

Обновим инсталляцию приложения

➜ prometheus helm upgrade myapp ./hello-chart --atomic

Release "myapp" has been upgraded. Happy Helming!

NAME: myapp

LAST DEPLOYED: Sat Apr 11 18:56:22 2020

NAMESPACE: monitoring

STATUS: deployed

REVISION: 3

NOTES:

TO BE DONE

➜ prometheus

Проверим, что ингресс завелся

➜ ~ curl -s -H'Host: hello.world' http://192.168.176.128/app/db | jq

[

{

"id": 1,

"name": "Konstantin"

}

]

Для того, что nginx начал отдавать метрики, необходимо ему это прописать в настройках.

➜ prometheus git:(master) ✗ ccat nginx-ingress.yaml

controller:

kind: DaemonSet

reportNodeInternalIp: true

hostPort:

enabled: true

ports:

http: 80

https: 443

service:

type: NodePort

metrics:

enabled: true

serviceMonitor:

enabled: true

metrics.serviceMonitor.enabled=true

Теперь обновим релиз nginx-ingress через helm

➜ prometheus helm upgrade nginx stable/nginx-ingress -f nginx-ingress.yaml

Release "nginx" has been upgraded. Happy Helming!

NAME: nginx

LAST DEPLOYED: Sat Apr 11 19:20:18 2020

NAMESPACE: monitoring

STATUS: deployed

REVISION: 2

TEST SUITE: None

NOTES:

The nginx-ingress controller has been installed.

Get the application URL by running these commands:

export HTTP\_NODE\_PORT=$(kubectl --namespace monitoring get services -o jsonpath="{.spec.ports[0].nodePort}" nginx-nginx-ingress-controller)

export HTTPS\_NODE\_PORT=$(kubectl --namespace monitoring get services -o jsonpath="{.spec.ports[1].nodePort}" nginx-nginx-ingress-controller)

export NODE\_IP=$(kubectl --namespace monitoring get nodes -o jsonpath="{.items[0].status.addresses[1].address}")

echo "Visit http://$NODE\_IP:$HTTP\_NODE\_PORT to access your application via HTTP."

echo "Visit https://$NODE\_IP:$HTTPS\_NODE\_PORT to access your application via HTTPS."

An example Ingress that makes use of the controller:

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

annotations:

kubernetes.io/ingress.class: nginx

name: example

namespace: foo

spec:

rules:

- host: www.example.com

http:

paths:

- backend:

serviceName: exampleService

servicePort: 80

path: /

# This section is only required if TLS is to be enabled for the Ingress

tls:

- hosts:

- www.example.com

secretName: example-tls

If TLS is enabled for the Ingress, a Secret containing the certificate and key must also be provided:

apiVersion: v1

kind: Secret

metadata:

name: example-tls

namespace: foo

data:

tls.crt: <base64 encoded cert>

tls.key: <base64 encoded key>

type: kubernetes.io/tls

➜ prometheus

Смотрим, что сервис монитор добавился

➜ prometheus kubectl get servicemonitors.monitoring.coreos.com

NAME AGE

myapp-hello-chart 4h43m

nginx-nginx-ingress-controller 61s

prom-prometheus-operator-alertmanager 33h

prom-prometheus-operator-apiserver 33h

prom-prometheus-operator-coredns 33h

prom-prometheus-operator-grafana 33h

prom-prometheus-operator-kube-controller-manager 33h

prom-prometheus-operator-kube-etcd 33h

prom-prometheus-operator-kube-proxy 33h

prom-prometheus-operator-kube-scheduler 33h

prom-prometheus-operator-kube-state-metrics 33h

prom-prometheus-operator-kubelet 33h

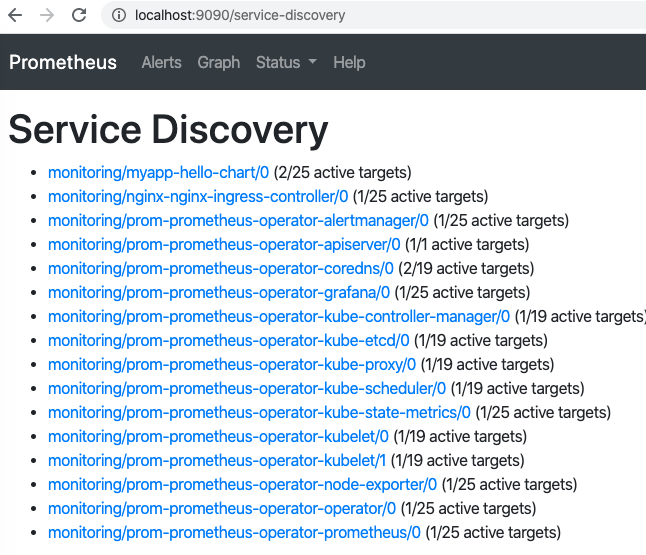
prom-prometheus-operator-node-exporter 33h

prom-prometheus-operator-operator 33h

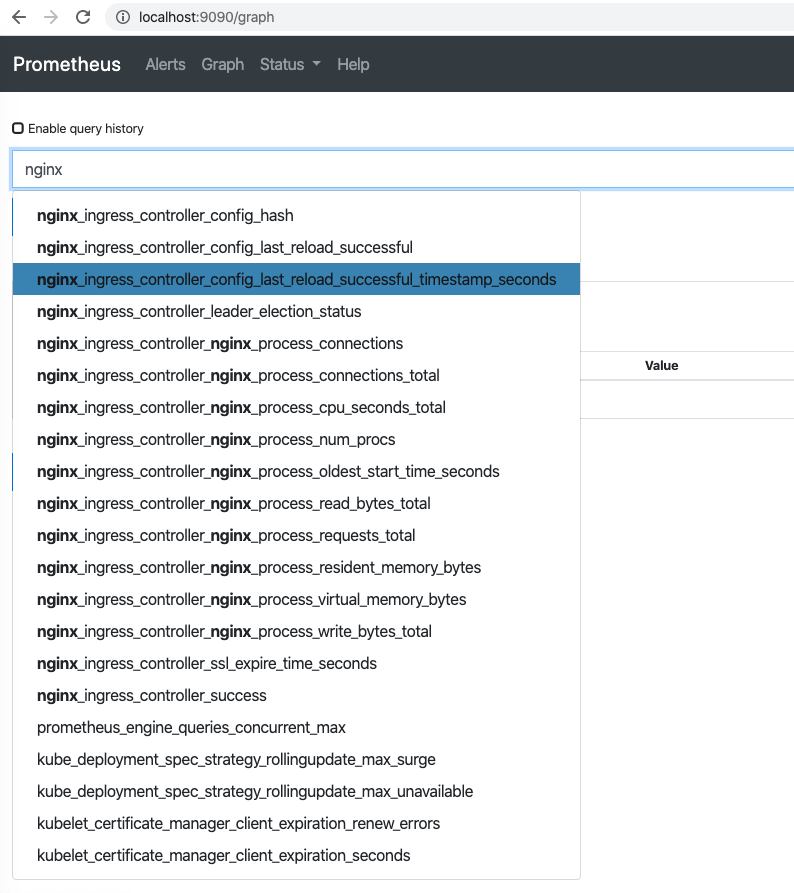
prom-prometheus-operator-prometheus 33h

➜ prometheus

Через некоторое время можем увидеть, что ингресс в таргетах прометея появился



Метрики nginx появились



Сделаем несколько запросов через nginx ingress, в которых есть указание хоста (это важно)

➜ ~ curl -s -H'Host: hello.world' http://192.168.176.128/app/db

[{"id": 1, "name": "Konstantin"}]% ➜ ~ curl -s -H'Host: hello.world' http://192.168.176.128/app/db

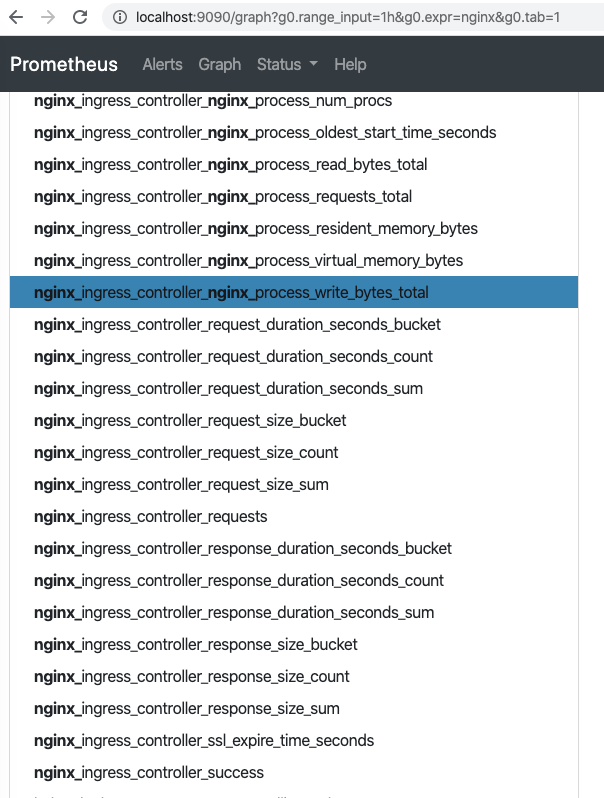
[{"id": 1, "name": "Konstantin"}]% ➜ ~ curl -s -H'Host: hello.world' http://192.168.176.128/app/db

[{"id": 1, "name": "Konstantin"}]% ➜ ~ curl -s -H'Host: hello.world' http://192.168.176.128/app/db

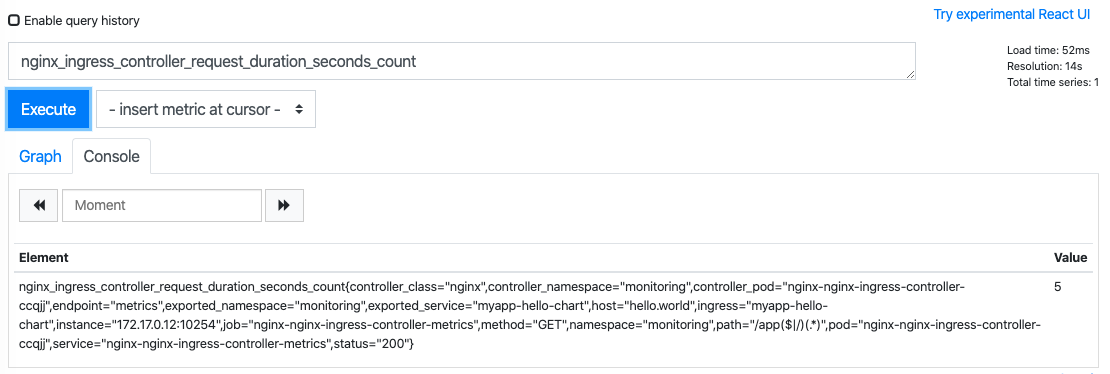
[{"id": 1, "name": "Konstantin"}]% ➜ ~ curl -s -H'Host: hello.world' http://192.168.176.128/app/db

[{"id": 1, "name": "Konstantin"}]% ➜ ~

И через некоторое время (зависит, как часто собирается с nginx cтата), появятся еще метрики, связанные с параметрами запросов и ответов.



Можно увидеть, что у нас всего 5 запросов было ( и это правда)



Самое интересное, это какие есть метки (labels)

nginx\_ingress\_controller\_request\_duration\_seconds\_count{controller\_class="nginx",controller\_namespace="monitoring",controller\_pod="nginx-nginx-ingress-controller-ccqjj",endpoint="metrics",exported\_namespace="monitoring",exported\_service="myapp-hello-chart",host="hello.world",ingress="myapp-hello-chart",instance="172.17.0.12:10254",job="nginx-nginx-ingress-controller-metrics",method="GET",namespace="monitoring",path="/app($|/)(.\*)",pod="nginx-nginx-ingress-controller-ccqjj",service="nginx-nginx-ingress-controller-metrics",status="200"}

Давайте построим rps-ы по ингрессам

Сначала дадим нагрузку на сервис через ингресс

➜ ~ while 1; do ab -n 50 -c 5 -H'Host: hello.world' http://192.168.176.128/app/db >> /dev/null ; sleep 3; done

~

И посмотрим на графики в прометее

sum by (status) (rate(nginx\_ingress\_controller\_request\_duration\_seconds\_count[1m]))



Как видим, появились 500ки.

Сравним с тем, что мы видим внутри приложения.



Приложение 500ки не видит. Потому что это 500 СНАРУЖИ приложения - сеть тупит или происходит рестарт пода не видит. Плюс внутренние ошибки могут таковы, что приложение их не перехватывает, и поэтому эта информация теряется.

Вообще, 500 лучше смотреть по количеству, а не по rate. И отслеживать их максимально близко к пользователю, а не к приложению.

sum by (status) (increase(nginx\_ingress\_controller\_request\_duration\_seconds\_count{status=~"5.+"}[1m]))



Для того, чтобы понимать, какие ошибки есть необходимо использовать Exception Handling инструменты типа sentry.