**Monitoring and Logging in Kubernetes**

* Metrics vs Logs:
  + Metric: A series of numbers over a period of time
  + Logs: Log represents some activity or information occured in the application/system. Logs have data in the form of text (majorly) in some format
* Monitoring Techniques:
  + Black-box monitoring focuses on monitoring from outside of an application and is what has been used traditionally when monitoring systems for components like CPU, memory, storage and so on. This can be still useful for monitoring at infrastructure level, but it lacks insights and context to how application is operating.
  + White-box monitoring focuses on details in the context of application state, such as total HTTP requests, number of 500 error, latency of request and so on. With this monitoring we can understand why of our system state.

**Monitoring Patterns**

* Kubernetes is much more dynamic and transient, and we need to change the way how we monitor these environments
* There are couple of different monitoring patterns to focus on when monitoring distributed systes
  + The USE method (Bredan Gregg) :
    - Focuses on
      * U – Utilization
      * S – Saturation
      * E – Errors
    - This method is focused on infra monitoring. The USE Method can be described as "For every resource, check utilization, saturation and error rates"
    - This method helps in quickly identifying resource contstraints and error rates of systems
  + The RED Method (Tom Wilke):
    - Focuses on
      * R – Rate
      * E – Errors
      * D – Duration
    - This philosophy was taken from Google Four Golden signals
      * Latency (How long it takes to serve a request)
      * Traffic (how much demand is place on system)
      * Error (rate of requests that are failing)
      * Saturation (How utilized your service)

**Kubernetes Metrics Overview** Let’s look at what components we should be monitoring in k8s cluster.

* In K8s we need to monitor
  + Control-Plane Components
    - API Server
    - etcd
    - Scheduler
    - Controller
  + Worker Node Components
    - Kubelet
    - Container Runtime
    - Kube-proxy
    - Kube-dns
    - Pods
* Kubernetes exposes the above metrics in variety of ways, so lets look at different components that we can use to collect metrics in your cluster
  + cAdvisor:
    - Container Advisor or cAdvisor is an open source project that collects resources and metrics for containers running on a node. cAdvisor is built into the k8s kubelet, which runs on every node in Cluster
    - It collects memory and CPU metrics using cgroup.
    - It also collects disk metrics through statfs (which is built into the Linux Kernel)
    - We should be considering cAdvisor as source of truth for all container metrics
  + Metrics Server:
    - The k8s metrics server and Metrics Server API are replacement for deprecated Heapster.
    - There are two aspects to understand in Metrics Server API and metrics server
      * The implementation of Resource Metrics API is the metrics server. This metrics server gathers resource metrics such as CPU and memory. It gathers these metrics from kubelet’s API and stores in memory. K8s uses these resource metrics in the scheduler,Horizontal Pod AutoScaler (HPA) and Vertical Pod AutoScaler (VPA)
      * The Custom Metrics API allows monitoring systems to collect arbitrary metrics. This allows monitoring solutions to build custom adapters that will allow for extending outside the core resource metrics. For example Prometheus built one of the first custom metrics adapters, which allows us to use the HPA based on a custom metric.
  + kube-state-metrics:
    - This is k8s add-on that monitors the object stored in k8s.
    - Where cAdvisor and metrics server are used to provide the detailed metrics on resource usage, kube-state-metrics is focused on identifying conditions on k8s objects deployed on the cluster
    - Following are some questions that kube-state-metrics can answer
      * Pods
        + How many Pods are deployed to the cluster?
        + How many Pods are in Pending State?
        + Are there enough resoruces to serve a pod request?
      * Deployments
        + How many pods are in running state vs desired state?
        + How many replicas are available?
        + What deployements have been update?
      * Nodes:
        + Whats the status of the Worker Nodes?
        + WHat are allocatable CPU cores in my cluster?
        + Are there any nodes that are unschedulable?

**Monitoring Tools**

* There are many monitoring tools that can be integrated with K8s. Following are few popular tools
  + Prometheus
  + Influx DB
  + Datadog
  + Sysdig
  + Cloud Provider Tools:
    - GCP Stackdriver
    - Microsoft Azure Monitor for containers
    - AWS Container Insights

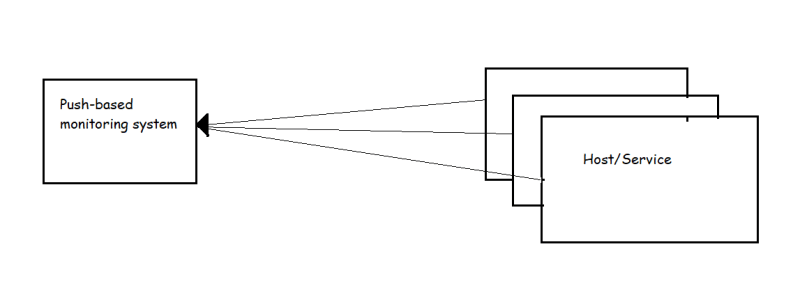
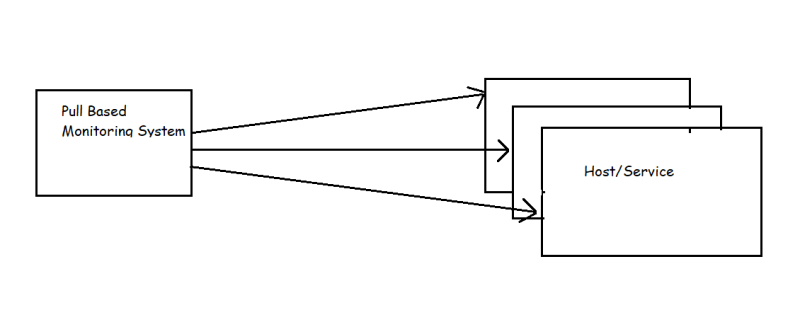
**Logging Overview**

* In k8s cluster, there are multiple components to log
* Following is a list of components from which you should be collecting metrics
  + Node logs
  + K8s Control-Plane logs
    - API Server
    - Controller Manager
    - Scheduler
  + K8s audit logs
  + Applciation Container logs

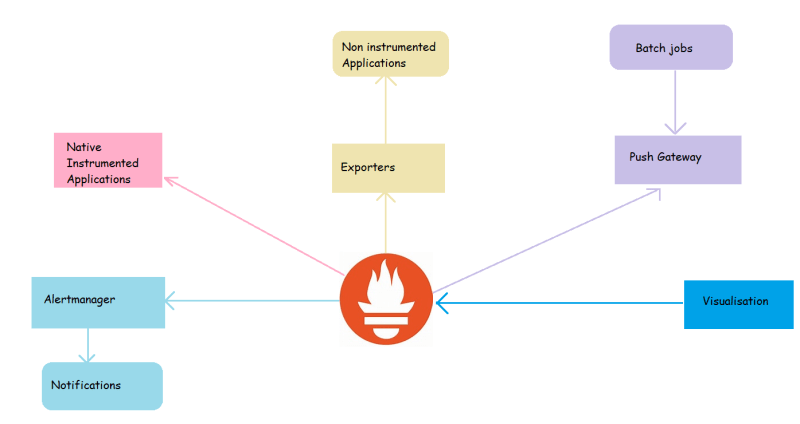
**Tools for Logging**

* Some of more popular tools with k8s integration
  + Elastic Stack
  + Data dog
  + Sumo Logic
  + Sys dig
  + Cloud Provider Services
    - GCP Stack Driver
    - Azure Monitor for Containers
    - AWS Cloud Watch

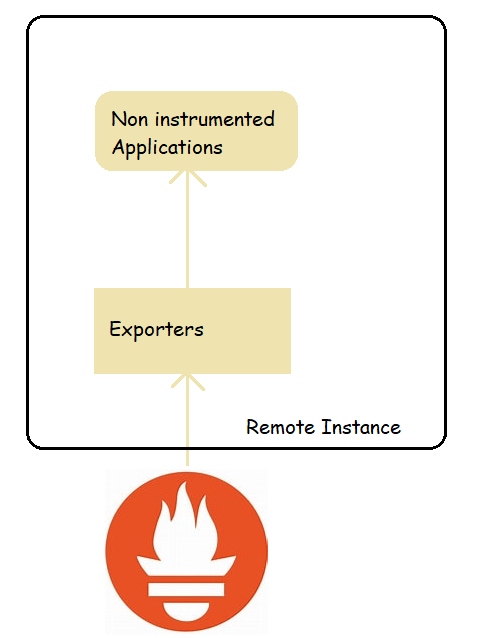
**An Overview of Metric Collection approaches (Push and Pull)**

* In Push Based monitoring systems, emitted metrics or events are sent directly from producing application or from a logical agent to the collecting server 
* Some examples of this approach is used in Elastic Search, LogStash and Kibana (Elastic Stack)
* In contrast, pull based monitoring systems collect metrics directly from applications or from proxy processes that makes those metrics available 
* Prometheus is a pull-based monitoring system & it also provides a way of ingesting pushed metrics by using a gateway that converts from push to pull.

**Prometheus**

* Prometheus is time series based open source monitoring system.
* It collects data by sending HTTP requests to hosts and services on metric endpoints, which it makes available for analysis and alerting using a powerful query language
* <https://prometheus.io/docs/introduction/overview/> for official docs
* Prometheus has joined Cloud Native Computing Foundation (CNCF) in year 2016.
* The Prometheus ecosystem is composed of several components
* High level overview of main components of Prometheus eco system 
* As we can see in the image above,
  + Prometheus Server collects time series data, stores it and makes it available for querying and send alerts based on it
  + The Alert Manager recieves alert triggers from Prometheus and handling routing and dispatching of events
  + The Push gatewayy handles the metrics that have been pushed from short-lived jobs (cron jobs or batch jobs)
  + Applications that support the Prometheus exposition format make internal state available through and HTTP endpoint
  + Community driven exporters expose metrics from applications that do not support prometheus natively
  + First-Party and Third-party dashboarding provide a visualization of collected
* Prometheus when it was originally created at SoundCloud was influenced from Google’s Borgmon.
  + Scraping plain text from metrics endpoints
  + exporters as proxies for metrics collections
  + time series as multi dimensional vectors
  + use of ruleset evaluations

**Exposing Internal State with exporters**

* Not all applications are built with Prometheus compatible instrumentations, Sometimes no metrics are exposed at all, In these case we can rely of exporter. 
* Exporter is nothing more that a piece of software that collects data from service or application and exposes via HTTP in the Prometheus
* Node Exporter is one of most commonly used exporters, which presents number of kernel statistics such as disk I/O, CPU, Memory, network, filesystem usage and much more.
* We have exporters for pretty much everything <https://prometheus.io/docs/instrumenting/exporters/>
* Terminology:
  + Scrape: The HTTP GET request made by the Prometheus server to the observed system for metric collection is called as scrape.
* Guidelines:
  + If you are one writing the service, the best option is to instrument the code directly using a Prometheus client library.
  + There are official libraries for
    - Go
    - Java
    - Python
    - Ruby
  + There are community driven client libraries for almost all the programming languages <https://prometheus.io/docs/instrumenting/clientlibs/>
  + If you want to develop exporters <https://prometheus.io/docs/instrumenting/writing_exporters/>
* Alerting routes: There are multiple out of the box integrations available for most common use case such as
  + email
  + HipChat
  + Slack
  + OpsGenie
  + PagerDuty

**Visualization Data collected**

* Prometheus exposes a well defined AP where PromQL queries can provide raw data for visualizations
* As of now the best external software for visualization used along with Prometheus is Grafana.
* The Prometheus server also ships with two internal visualization components
  + Experssion browser: Here we can run PromQL directly to quickly query and visualize data instantly
  + Consoles: These are web pages that ar built using the Golang templating language and served by Prometheus server itself.

**Local Environment**

* Windows 10:
  + Download Docker
  + Install kubectl
  + Install Helm

**Minikube installation**

* <https://minikube.sigs.k8s.io/docs/start/>
* Now start the local k8s cluster

minikube start

**Install Prometheus**

* We will be installing prometheus using Helm

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

helm install prometheus prometheus-community/prometheus

kubectl expose service prometheus-server --type=NodePort --target-port=9090 --name=prom-server

minikube service prom-server

* To setup the prometheus follow steps <https://www.redhat.com/sysadmin/installing-prometheus>

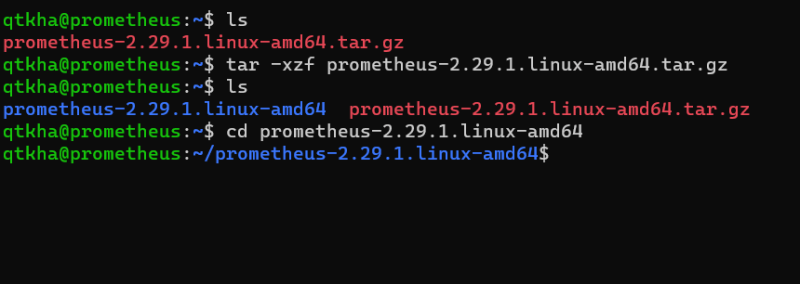
**Install Grafana**

helm repo add grafana https://grafana.github.io/helm-charts

helm search repo grafana

helm install grafana bitnami/grafana

**Running Prometheus**

* Lets try to run Prometheus on a Linux machine
* Lets create a vm
* Download the linux tar file <https://prometheus.io/download/> 
* Now in the prometheus.yml file ensure you have the following content

# my global config

global:

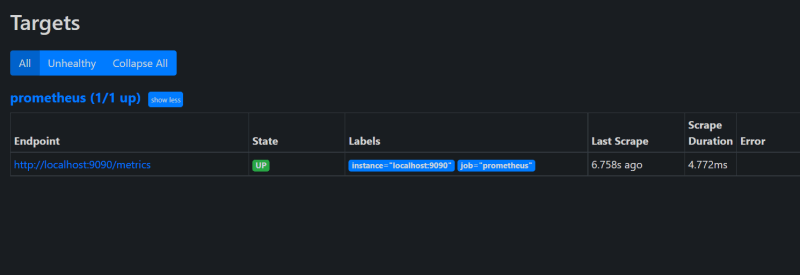
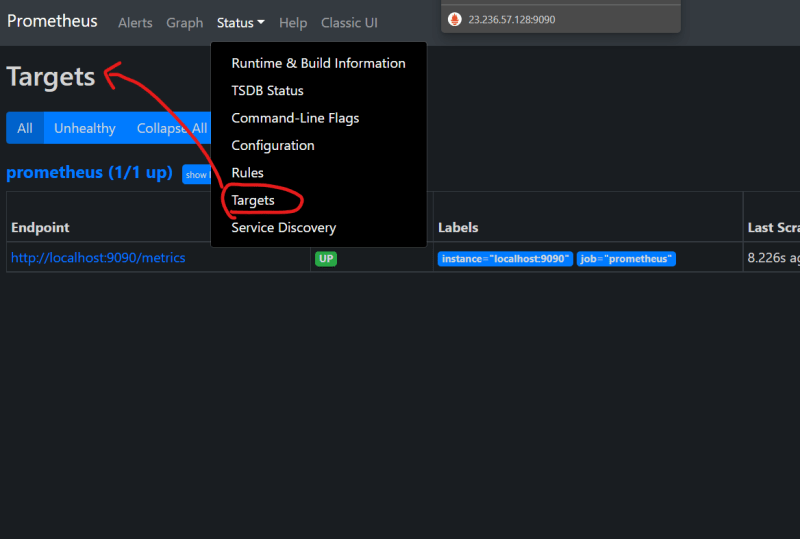
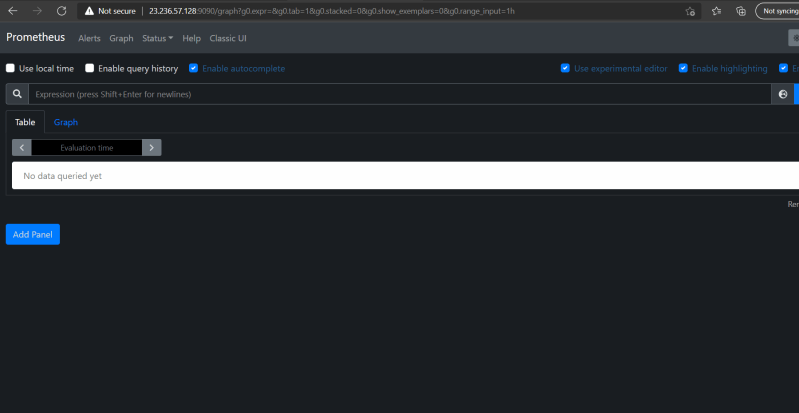
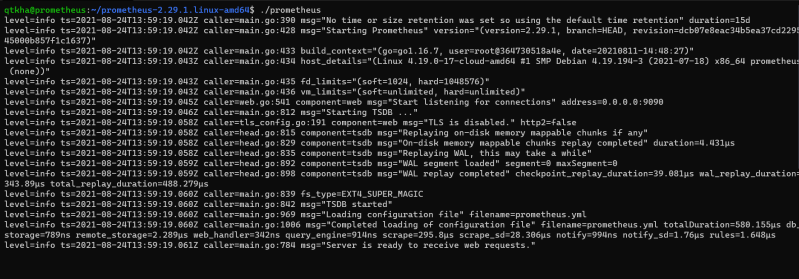
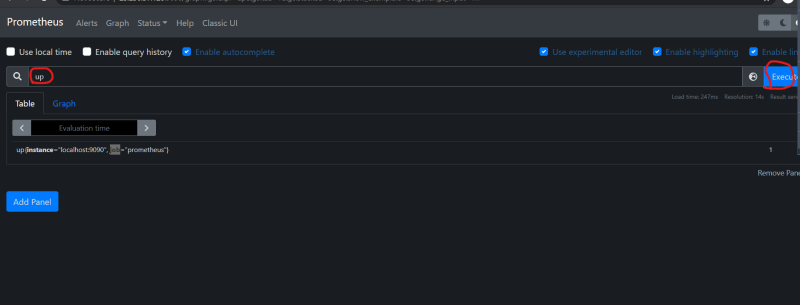
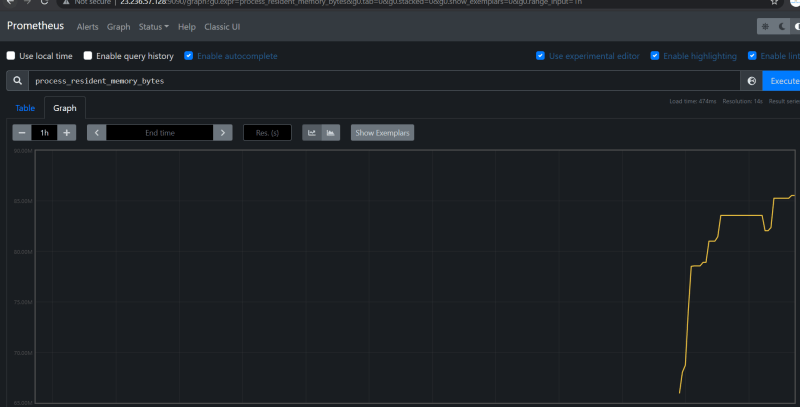
scrape\_interval: 10s

scrape\_configs:

- job\_name: "prometheus"

static\_configs:

- targets: ["localhost:9090"]

* Now lets try to run prometheus 
* Open the prometheus expression browser and execute up 
* There is a single result with value 1 and the name up{instance="localhost:9090", job="prometheus"}.
* up is a special metric added by Prometheus when it performsa scrape and 1 indicates that the scrape was succesful. Then instance is a label, indicating the target wthat was scrapted and the job label here comes from job\_name in the prometheus.yml file 
* Lets try to download a Node exporter <https://prometheus.io/download/>
  + untar the exporter and try to run the node exporter with ./node\_exporter
  + Now access the metrics from node exporter
* Now lets add some more info to the prometheus.yml to scrape node metrics exported by node exporter

global:

scrape\_interval: 10s

scrape\_configs:

- job\_name: "prometheus"

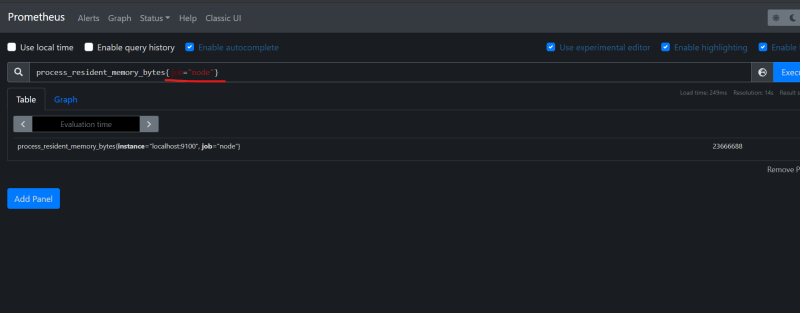
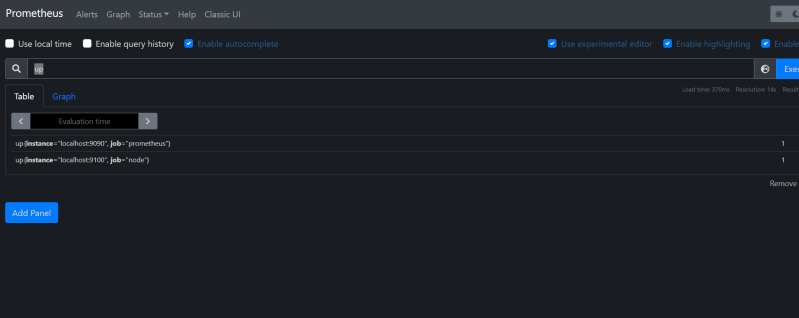
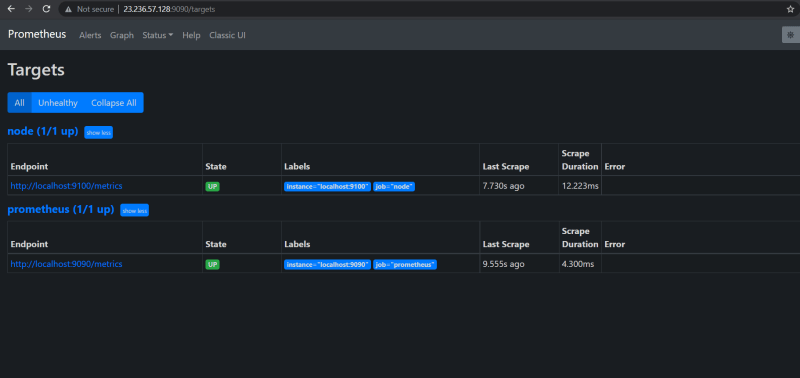
static\_configs:

- targets: ["localhost:9090"]

- job\_name: node

static\_configs:

- targets: ["localhost:9100"]

* and restart prometheus (./prometheus) 
* Now lets try to add some alert manager configuration

global:

scrape\_interval: 10s

alerting:

alertmanagers:

- static\_configs:

- localhost:9093

scrape\_configs:

- job\_name: "prometheus"

static\_configs:

- targets: ["localhost:9090"]

- job\_name: node

static\_configs:

- targets: ["localhost:9100"]

* Now create a new file called as rules.yaml

groups:

- name: example

rules:

- alert: InstanceDown

expr: up == 0

for: 1m

**Types of Metrics**

* Counter:
  + These are type of metrics which will use track either the number or size of the events
  + Examples
    - Request Count
    - Exception Count
    - Size of records processed
* Gauge:
  + These are snapshot of some current state
  + While for counters how fast it is increase is what we care about, for gauge it is actual value of gauge
  + Examples
    - number of items in queue
    - memory usage of a cache
    - number of active threads
* Histogram:
  + A summary while provide the average latency, but waht if you want quanitile?
  + Quantile tells you that a certain proportion of events had a size belo a give value.
  + For example: 0.95 quantile being 300 ms meants that 95% of requests took less than 300 ms

**Instrumentation**

* The largest payoffs you will get from prometheus are through instrumenting your own applications using direct instrumentation and client library
* <https://github.com/asquarezone/ExpertKubernetes/commit/b374d0f3ecb478739d6b76665d9f17d186bb559f> for the sample instrumentation code

**PromQL**

* PromQL is the Prometheus Query Language
* Labels are key part of PromQL and you can use them not only do arbitrary aggregations but also to join different metrics together for arthimetic operations against them.

**Aggregation Basics**

* Gauge: These are snapshots of state and usually when you are aggregating them you want to take a sum, average, minimum or maximum.
  + Consider the metric node\_filesystem\_size\_bytes (Node Exporter) which reports the size of each of your mounted filesystems and has device, fstype and mountpoint labels
  + Consider this query

sum without(device, fstype, mountpoint)(node\_filesystem\_size\_bytes)

* This works as without tells the sum aggregator to sum everything up with the same labels and ignoring these three
* Consider this query

max without(device, fstype, mountpoint)(node\_filesystem\_size\_bytes)

* This would return the biggest mounted filesystem on each device.
* Consider the expression avg without(instance, job)(process\_open\_fds)
* Counter: Counter tracks the number or size of events and the value your applications expose on their metrics.
  + When we use counter we would usually want to know how counter is increasing/decreasing over time
  + This can be done by rate function
* rate(node\_network\_receive\_bytes\_total[5m])
  + The above expression/query calculates amount of network traffic received per second and [5m] provides the rate function with 5 minutes of data
  + The output of rate function is a gauge, so we can use aggregations
* sum without(device)(rate(node\_network\_receive\_bytes\_total[5m]))
* Summary: Summary metric usually contains both \_sum and \_count and sometimes a time series with no suffix with a quantile lablel. \_sum and \_count are both counters
  + Prometheus exposes http\_response\_size\_bytes summary and http\_response\_size\_bytes\_count tracks number of user requests
  + Consider the expression sum without(handler)(rate(http\_response\_size\_bytes\_count[5m]))
* Histogram: Histogram metrics allows you to track the distribution of the size of the events, which allows you to calculate quantiles
  + Prometheus exposes a histogram prometheus\_tsdb\_compaction\_duration\_seconds that tracks how many seconds compaction takes for time series database
  + histogram\_quantile function takes catre of calculating quantiles
* histogram\_quantile(0.9, rate(prometheus\_tsdb\_compaction\_duration\_seconds[1d]))
* Selectors: working with all the different time series with different label values for a metric can be overwhelming and confusing. Usually you will want to narrow down which time series you are working on
  + process\_resident\_memory\_bytes{job="node"}
  + `job="node" is called a matcher and we have many matcher
  + Matchers: There are four matchers
    - =: this is equality matcher
    - !=: this is negative equality matcher
    - =~: This is regular expression mathcher job=~"n.\*"
    - !~: This is negative regular expression matcher instance!~"prod\*"
* Durations:
  + ms: Milliseconds
  + s: seconds
  + m: minutes
  + h: hours
  + d: days
  + w: weeks
  + y: year
  + While using durations write duration as 1 unit
* 100m (valid)
* 1h40m (invalid)
* Offset: There is a modifier we can use called as offset, which allows you take evaluation time for a query on a per-selector basis
  + process\_resident\_memory\_bytes{job="node"} offset 1h this would get memory usage an hour before the query evaluation time.
  + `rate(process\_cpu\_seconds\_total{job="node"}[5m] offset 1h )
* by: In addition to without ther s also a by clause. Where without specifies the labels to remove by specifies labesls to keep. you cannot use both by and without in same aggregation
  + sum by(job, instane, device)(node\_filesystem\_size\_bytes)
  + count by(release)(node\_uname\_info)
* Operators:
  + sum
  + count
  + avg
  + stddev
  + stdvar
  + min
  + max
  + topk
  + bottomk
  + quantile
  + count\_values
* Arithmetic Operators:
  + / devision
  + % : modulation
  + ^: exponentiation
* Comparision Operators
  + == equals
  + != not equals
  + <
  + >
  + >=
  + <=

**Setting up Prometheus on Kubernetes**

* <https://cloud.google.com/architecture/monitoring-apps-running-on-multiple-gke-clusters-using-prometheus-and-stackdriver#configure_cloud_identity_and_access_management_(iam)> for the article for setting up prometheus on gke
* Steps to execute from windows terminal /powershell

git clone https://github.com/GoogleCloudPlatform/prometheus-stackdriver-gke

cd prometheus-stackdriver-gke

gcloud iam service-accounts create prometheus --display-name prometheus-service-account

$PROJECT\_ID=$(gcloud info --format='value(config.project)')

$PROMETHEUS\_SA\_EMAIL=$(gcloud iam service-accounts list --filter="displayName:prometheus-service-account" --format='value(email)')

gcloud projects add-iam-policy-binding $PROJECT\_ID --role roles/monitoring.metricWriter --member serviceAccount:$PROMETHEUS\_SA\_EMAIL

gcloud iam service-accounts keys create prometheus-service-account.json --iam-account $PROMETHEUS\_SA\_EMAIL

gcloud container clusters create hello-cluster --num-nodes=1

gcloud container clusters get-credentials hello-cluster

kubectl create namespace prometheus

* Make changes in gke-prometheus-deployment.yaml

#Copyright 2019 Google LLC

#Licensed under the Apache License, Version 2.0 (the "License");

#you may not use this file except in compliance with the License.

#You may obtain a copy of the License at

#https://www.apache.org/licenses/LICENSE-2.0

#Unless required by applicable law or agreed to in writing, software

#distributed under the License is distributed on an "AS IS" BASIS,

#WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

#See the License for the specific language governing permissions and

#limitations under the License.

apiVersion: apps/v1

kind: Deployment

metadata:

name: prometheus-deployment

namespace: prometheus

labels:

app: prometheus-server

spec:

replicas: 1

selector:

matchLabels:

app: prometheus-server

template:

metadata:

labels:

app: prometheus-server

spec:

containers:

- name: prometheus

image: prom/prometheus:v2.19.3

args:

- "--config.file=/etc/prometheus/prometheus.yml"

- "--storage.tsdb.path=/prometheus/"

ports:

- containerPort: 9090

volumeMounts:

- name: prometheus-config-volume

mountPath: /etc/prometheus/

- name: prometheus-storage-volume

mountPath: /prometheus/

- name: sidecar

image: gcr.io/stackdriver-prometheus/stackdriver-prometheus-sidecar:0.8.2

imagePullPolicy: Always

args:

- --stackdriver.project-id=expertkubernetes

- --prometheus.wal-directory=/prometheus/wal

- --stackdriver.kubernetes.location=us-central1-a

- --stackdriver.kubernetes.cluster-name=hello-cluster

#- \"--stackdriver.generic.location=${GCP\_LOCATION}\"

#- \"--stackdriver.generic.namespace=${KUBE\_CLUSTER}\"

ports:

- name: sidecar

containerPort: 9091

volumeMounts:

- name: prometheus-storage-volume

mountPath: /prometheus

volumes:

- name: prometheus-config-volume

configMap:

defaultMode: 420

name: prometheus-server-conf

- name: prometheus-storage-volume

emptyDir: {}

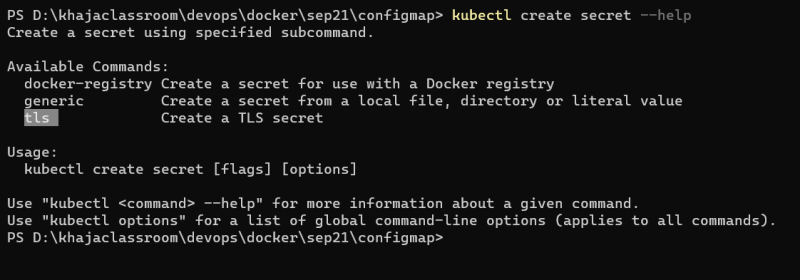
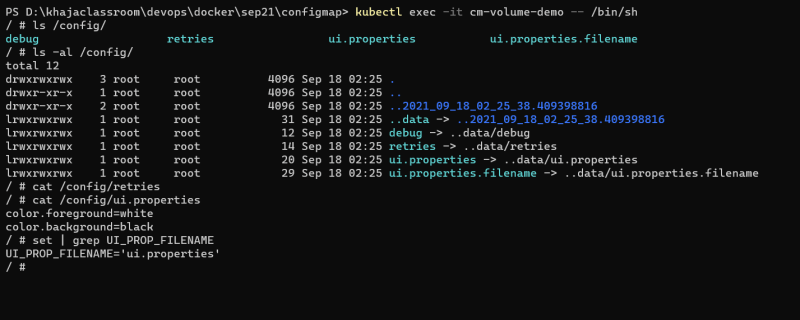
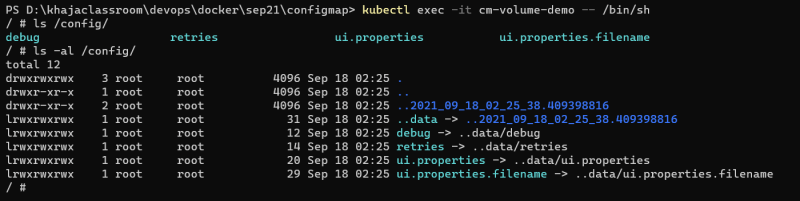
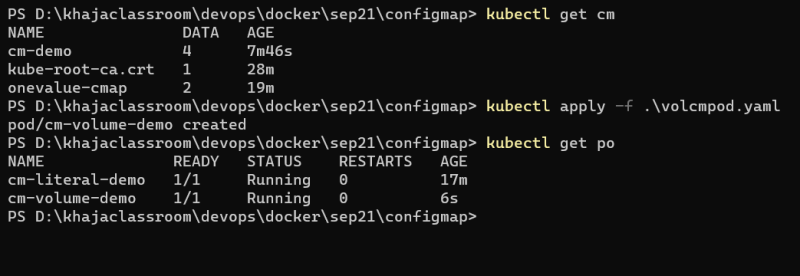
* Now apply

kubectl apply -f .\gke-prometheus-deployment.yaml

kubectl get pods -n prometheus

$PROMETHEUS\_POD\_GKE=$(kubectl get pods --namespace prometheus -l "app=prometheus-server" -o jsonpath="{.items[0].metadata.name}" )

kubectl port-forward --namespace prometheus $PROMETHEUS\_POD\_GKE 9090:9090

* Now work with Prometheus Expression Browser 
* Now lets create a postgres deployment into the k8s cluster

helm repo add bitnami https://charts.bitnami.com/bitnami

helm repo update

helm install gke bitnami/postgresql --set metrics.enabled=true --set postgresqlDatabase=pro

**Alert Manager Installation**

* Lets install alert manager on the same server where prometheus is running
* create a user called as alertmanager

sudo useradd --no-create-home --shell /bin/false alertmanager

* Lets create some folders for holding alert manager

sudo mkdir /etc/alertmanager

sudo mkdir -p /data/alertmanager

* Now Download & untar alertmanager

wget https://github.com/prometheus/alertmanager/releases/download/v0.23.0/alertmanager-0.23.0.linux-amd64.tar.gz

tar xzf alertmanager-0.23.0.linux-amd64.tar.gz

* Lets copy alertmanager amtool to /usr/local/bin

sudo cp amtool /usr/local/bin/

sudo cp alertmanager /usr/local/bin/

* copy the alert manager yaml file /etc/alertmanager/

sudo cp alertmanager.yml /etc/alertmanager

* Lets give permissions to alertmanager user

sudo chown alertmanager:alertmanager /usr/local/bin/{amtool,alertmanager}

sudo chown -R alertmanager:alertmanager /data/alertmanager /etc/alertmanager/\*

* Lets create a systemd unitfile at /etc/systemd/system/alertmanager.service

[Unit]

Description=AlertManager

Wants=network-online.target

After=network-online.target

[Service]

User=alertmanager

Group=alertmanager

Type=simple

ExecStart=/usr/local/bin/alertmanager \

--config.file /etc/alertmanager/alertmanager.yml \

--storage.path /data/alertmanager

[Install]

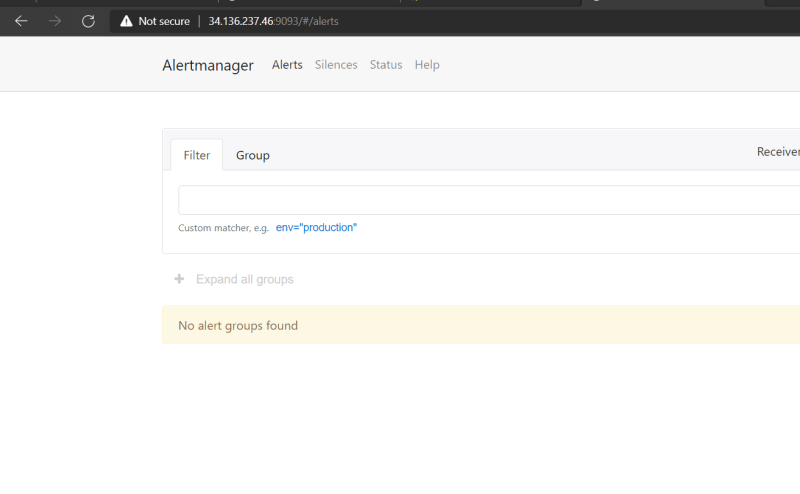
WantedBy=multi-user.target

* Now enable and start the alertmanager

sudo systemctl enable alertmanager.service

sudo systemctl start alertmanager.service

sudo systemctl status alertmanager.service

* 
* We need to bind the alert manager to prometheus. Change /etc/prometheus/prometheus.yml to add the following section of alerts

global:

scrape\_interval: 15s

alerting:

alertmanagers:

- static\_configs:

- targets: ['localhost:9093']

scrape\_configs:

- job\_name: 'prometheus'

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:9090']

- job\_name: 'kubernetes'

static\_configs:

- targets: ['10.128.0.31:30000']

* Now restart prometheus

sudo systemctl restart prometheus

* Now change the alert manager to add the email reciever

GNU nano 4.8 /etc/alertmanager/alertmanager.yml Modified route:

group\_by: ['alertname']

group\_wait: 30s

group\_interval: 5m

repeat\_interval: 1h

receiver: 'web.hook'

receivers:

- name: 'email'

email\_configs:

- to: 'devops@qt.com'

from: 'alerts@qt.com'

smarthost: smtp.mailtrap.io:587

auth\_username: 'jdsflksd'

auth\_password: 'test'

- name: 'web.hook'

webhook\_configs:

- url: 'http://127.0.0.1:5001/'

inhibit\_rules:

- source\_match:

severity: 'critical'

target\_match:

severity: 'warning'

equal: ['alertname', 'dev', 'instance']

**Recording Rules**

* We can use recording rules to have prometheus evaluate PromQL Exressions regularly and ingest their results.
* This is useful to speed up your dashboards and provide aggregrated results for use elsewhere
* Recording rules got in seperate files from prometheus.yaml which can be specified in rule\_files top\_level filed in prometheus.yml file

global:

scrape\_interval: 15s

rule\_files:

- rules.yaml

alerting:

alertmanagers:

- static\_configs:

- targets: ['localhost:9093']

scrape\_configs:

- job\_name: 'prometheus'

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:9090']

- job\_name: 'kubernetes'

static\_configs:

- targets: ['10.128.0.31:30000']

* Sample rules.yaml

groups:

- name: example

rules:

- record: job:process\_cpu\_seconds:rate5m

expr: sum without(instance)(rate(process\_cpu\_seconds\_total[5m]))

**Alerting**

* There is a set of community alerts created and hosted over here
* Let’s create a alert-k8s.yaml in /etc/prometheus/alerts/k8s.yaml

groups:

- name: LearningK8s

rules:

- alert: KubernetesNodeReady

expr: kube\_node\_status\_condition{condition="Ready",status="true"} == 0

for: 1m

labels:

severity: critical

annotations:

summary: Kubernetes Node ready (instance {{ $labels.instance }})

description: "Node {{ $labels.node }} has been unready for a long time\n VALUE = {{ $value }}\n LABELS = {{ $labels }}"

* prometheus.yaml

global:

scrape\_interval: 15s

rule\_files:

- 'alerts/k8s.yml'

alerting:

alertmanagers:

- static\_configs:

- targets: ['localhost:9093']

scrape\_configs:

- job\_name: 'prometheus'

scrape\_interval: 5s

static\_configs:

- targets: ['localhost:9090']

- job\_name: 'kubernetes'

static\_configs:

- targets: ['10.128.0.31:30000']

* As of now we are able to get the alert in prometheus which gets forwarded to alert manager but we are not recieving email
* Note: I will try to look into this issue and we will create some alerts