Defining Alert Rules

In this lesson, we will install and see the usage of Prometheus and it's screens.

WE'LL COVER THE FOLLOWING

Installation of `Prometheus`

We'll continue the trend of using *Helm* as the installation mechanism. `Prometheus` 's Helm Chart is maintained as one of the official Charts. You can find more info in the project's README. If you focus on the variables in the Configuration section, you'll notice that there are quite a few things we can tweak. We won't go through all the variables. You can check the official documentation for that. Instead, we'll start with a basic setup, and extend it as our needs increase.

Let's take a look at the variables we'll use as a start.

```
cat mon/prom-values-bare.yml
```

The **output** is as follows.

```
server:
  ingress:
    enabled: true
    annotations:
      ingress.kubernetes.io/ssl-redirect: "false"
```

```
      nginx.ingress.kubernetes.io/ssl-redirect: "false"
  resources:

    limits:
      cpu: 100m
      memory: 1000Mi
    requests:
      cpu: 10m
      memory: 500Mi
alertmanager:
  ingress:
    enabled: true
    annotations:
      ingress.kubernetes.io/ssl-redirect: "false"
      nginx.ingress.kubernetes.io/ssl-redirect: "false"
  resources:
    limits:
      cpu: 10m
      memory: 20Mi
    requests:
      cpu: 5m
      memory: 10Mi
kubeStateMetrics:
  resources:
    limits:
      cpu: 10m
      memory: 50Mi
    requests:
      cpu: 5m
      memory: 25Mi
nodeExporter:
  resources:
    limits:
      cpu: 10m
      memory: 20Mi
    requests:
      cpu: 5m
      memory: 10Mi
pushgateway:
  resources:
    limits:
      cpu: 10m
      memory: 20Mi
    requests:
      cpu: 5m
      memory: 10Mi
```

All we're doing, for now, is defining `resources` for all five applications we'll install and enabling Ingress with a few annotations that will make sure that we are not redirected to the HTTPS version since we do not have certificates for our ad-hoc domains. We'll dive into the applications that'll be installed later.

For now, we'll define the addresses for `Prometheus` and `Alertmanager` UIs.

```
PROM_ADDR=mon.$LB_IP.nip.io

AM_ADDR=alertmanager.$LB_IP.nip.io
```

Let's install the Chart.

```
kubectl create namespace metrics

helm install prometheus \
    stable/prometheus \
    --namespace metrics \
    --version 9.5.2 \
    --set server.ingress.hosts={$PROM_ADDR} \
    --set alertmanager.ingress.hosts={$AM_ADDR} \
    -f mon/prom-values-bare.yml
```

The command we just executed should be self-explanatory.

The Chart installed one DeamonSet and four Deployments.

The **DeamonSet** is a Node Exporter, and it'll run a Pod on every node of the cluster. It provides node-specific metrics that will be pulled by `Prometheus`. The second exporter (Kube State Metrics) runs as a single replica Deployment. It fetches data from Kube API and transforms them into the Prometheus-friendly format. The two will provide most of the metrics we'll need. Later on, we might choose to expand them with additional exporters. For now, those two together with metrics fetched directly from Kube API should provide more metrics than we can absorb in a single chapter.

Further on, we got the Server, which is `Prometheus` itself. `Alertmanager` will forward alerts to their destination. Finally, there is *Pushgateway* that we might explore in one of the following chapters.

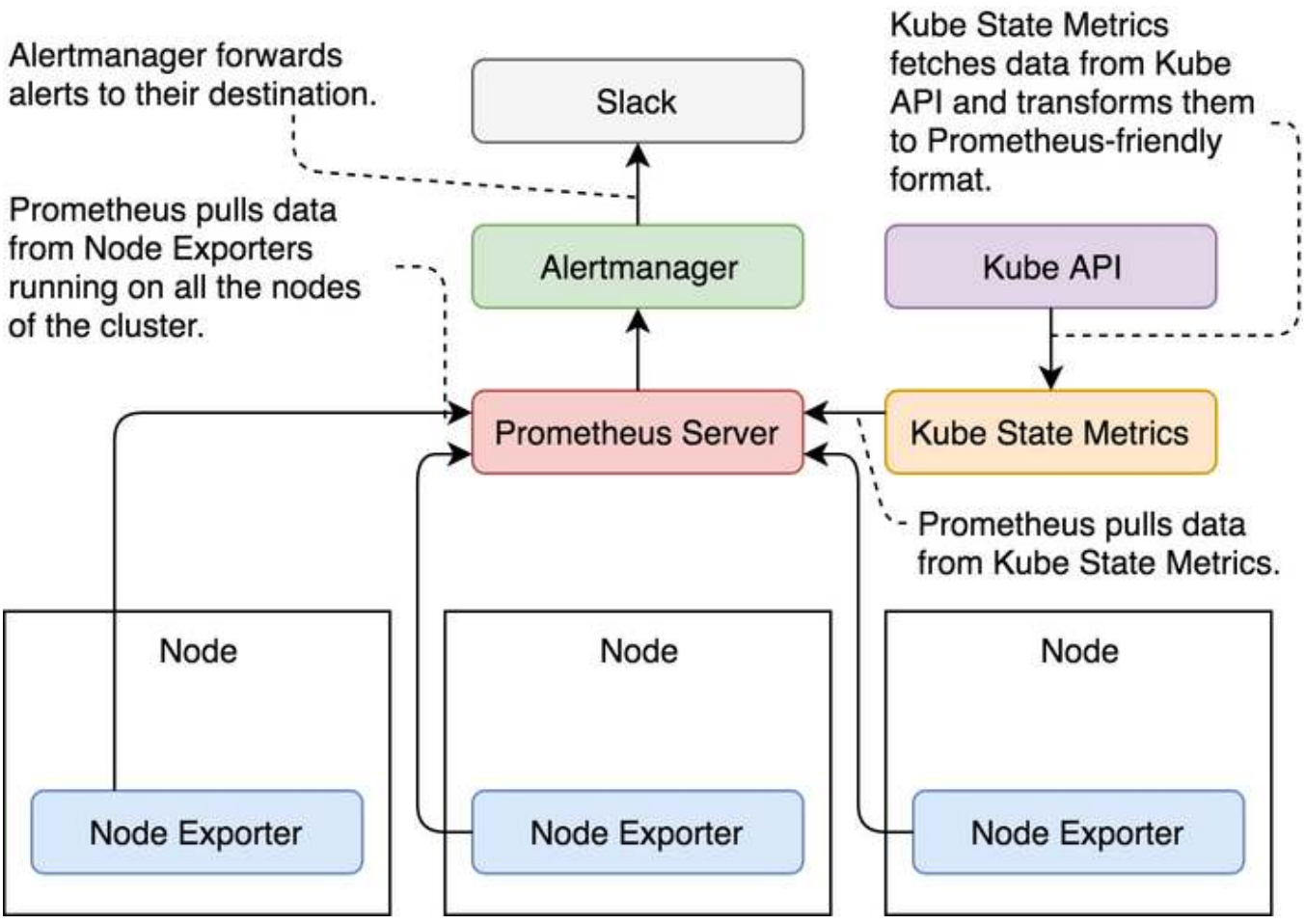
While waiting for all those apps to become operational, we might explore the

flow between them.

Flow of data to and from `Prometheus`

`Prometheus Server` pulls data from exporters. In our case, those are Node Exporter and Kube State Metrics. The job of those exporters is to fetch data from the source and transform it into the Prometheus-friendly format. Node Exporter gets the data from `/proc` and `/sys` volumes mounted on the nodes, while Kube State Metrics gets it from Kube API. Metrics are stored internally in `Prometheus`.

Apart from being able to query that data, we can define alerts. When an alert reaches its threshold, it is forwarded to `Alertmanager` which acts as a crossroad. Depending on its internal rules, it can forward those alerts further to various destinations like Slack, email, and HipChat (only to name a few).



The flow of data to and from Prometheus (arrows indicate the direction)

Q **Daemon Set** runs as a single replica Deployment. It fetches data from Kube API and transforms them into the Prometheus-friendly format.

`Prometheus Server` rolled out

By now, `Prometheus Server` probably rolled out. We'll confirm that just in case.

```
kubectl -n metrics \
    rollout status \
    deploy prometheus-server
```

Let's take a look at what is inside the Pod created through the `prometheus-server` Deployment.

```
kubectl -n metrics \
    describe deployment \
    prometheus-server
```

The **output**, limited to the relevant parts, is as follows.

```
  Containers:
   prometheus-server-configmap-reload:
    Image: jimmidyson/configmap-reload:v0.2.2
    ...
   prometheus-server:
    Image: prom/prometheus:v2.4.2
    ...
```

Besides the container based on the `prom/prometheus` image, we got another one created from `jimmidyson/configmap-reload`. The job of the latter is to reload `Prometheus` whenever we change the configuration stored in a ConfigMap.

Next, we might want to take a look at the `prometheus-server` ConfigMap, since it stores all the configuration `Prometheus` needs

It stores all the configuration `Prometheus` needs.

```
kubectl -n metrics \
    describe cm prometheus-server
```

The **output**, limited to the relevant parts, is as follows.

```
...
Data
====
alerts:
----
{}

prometheus.yml:
----
global:
  evaluation_interval: 1m
  scrape_interval: 1m
  scrape_timeout: 10s

rule_files:
- /etc/config/rules
- /etc/config/alerts
scrape_configs:
- job_name: prometheus
  static_configs:
  - targets:
    - localhost:9090
- bearer_token_file: /var/run/secrets/kubernetes.io/serviceaccount/token
  job_name: kubernetes-apiservers
  kubernetes_sd_configs:
  - role: endpoints
  relabel_configs:
  - action: keep
    regex: default;kubernetes;https
    source_labels:
    - __meta_kubernetes_namespace
    - __meta_kubernetes_service_name
    - __meta_kubernetes_endpoint_port_name
  scheme: https
  tls_config:
    ca_file: /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    insecure_skip_verify: true
...
```

We can see that the `alerts` are still empty. We'll change that soon.

Further down is the `prometheus.yml` config with `scrape_configs` taking most of the space. We could spend a whole chapter explaining the current config and the ways we could modify it. We will not do that because the config in front of you is bordering insanity. It's the prime example of how something can be made more complicated than it should be. In most cases, you should keep it as-is. If you do want to fiddle with it, please consult the official documentation.

Looking into `Prometheus`'s screens

Next, we'll take a quick look at `Prometheus`'s screens.

> ### A note to Windows users
>
> 🔍 Git Bash might not be able to use the `open` command. If that's the case, replace `open` with `echo`. As a result, you'll get the full address that should be opened directly in your browser of choice.

```
open "http://$PROM_ADDR/config"
```

The config screen reflects the same information we already saw in the `prometheus-server` ConfigMap, so we'll move on.

Next, let's take a look at the targets.

```
open "http://$PROM_ADDR/targets"
```

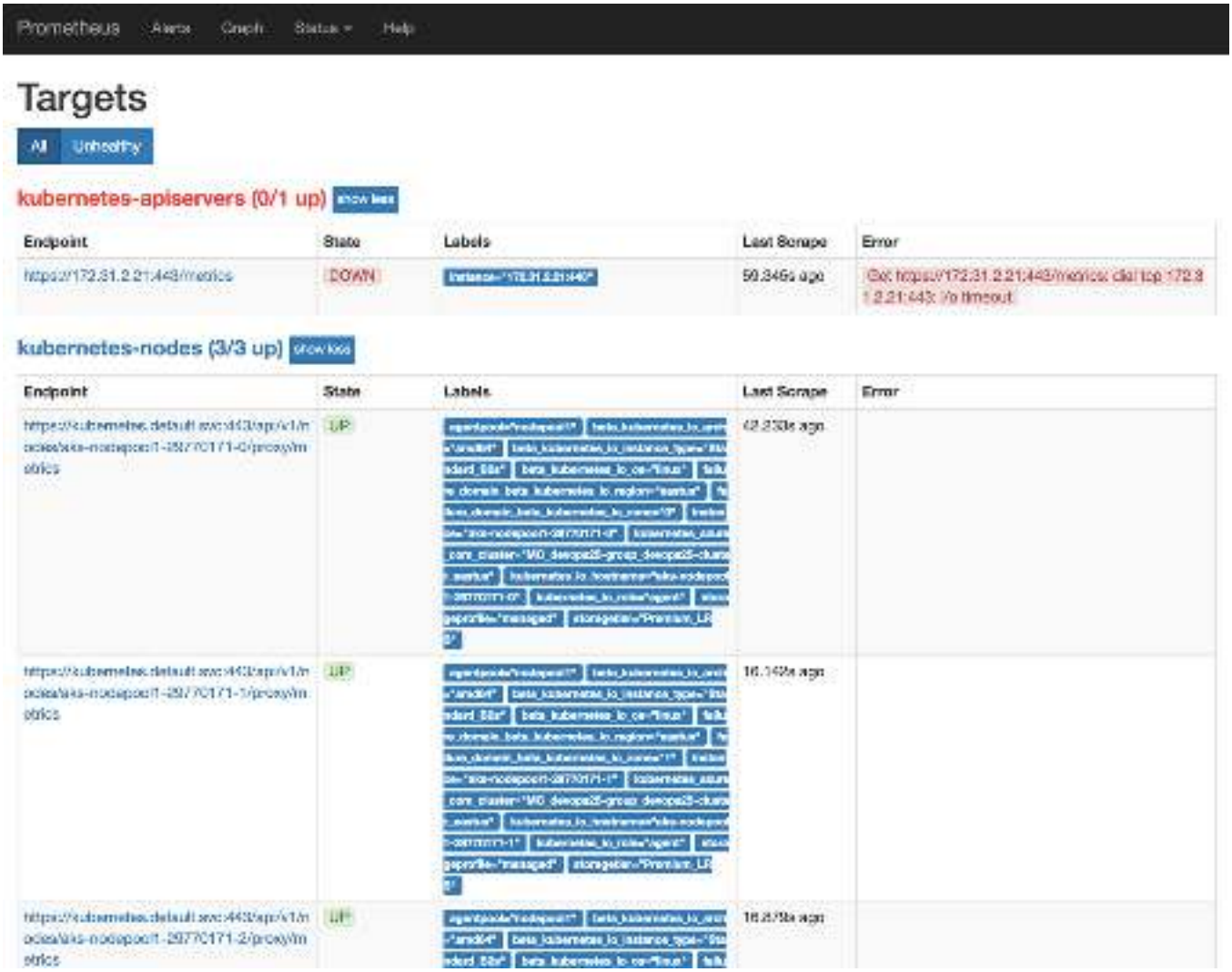
That screen contains seven targets, each providing different metrics. `Prometheus` is periodically pulling data from those targets.

> 🔍 All the outputs and screenshots in this chapter are taken from AKS. You might see some differences depending on your Kubernetes flavor.

> 🔍 You might notice that this chapter contains many more screenshots than any other. Even though it might look like there are too many, I

wanted to make sure that you can compare your results with mine, since

there will be inevitable differences that might sometimes look confusing if you do not have a reference (my screenshots).



Prometheus' targets screen

A note to AKS users

🔍 The *kubernetes-apiservers* target might be red indicating that `Prometheus` cannot connect to it. That's OK since we won't use its metrics.

A note to minikube users

🔍 The *kubernetes-service-endpoints* target might have a few sources in red. There's no reason for alarm. Those are not reachable, but that won't affect our exercises.

Query the Exporters

We cannot find out what each of those targets provides from that screen. We'll try to query the exporters in the same way as `Prometheus` pulls them. To do that, we'll need to find out the Services through which we can access the exporters.

```
kubectl -n metrics get svc
```

The **output**, from AKS, is as follows.

```
NAME                            TYPE        CLUSTER-IP     EXTERNAL-IP PORT(S)
  AGE
prometheus-alertmanager         ClusterIP 10.23.245.165 <none>      80/TCP
   41d
prometheus-kube-state-metrics ClusterIP None          <none>      80/TCP
   41d
prometheus-node-exporter        ClusterIP None          <none>      9100/TCP
 41d
prometheus-pushgateway          ClusterIP 10.23.244.47  <none>      9091/TCP
 41d
prometheus-server               ClusterIP 10.23.241.182 <none>      80/TCP
   41d
```

We are interested in `prometheus-kube-state-metrics` and `prometheus-node-exporter` since they provide access to data from the exporters we'll use in this chapter.

Next, we'll create a temporary Pod through which we'll access the data available through the exporters behind those Services.

```
kubectl -n metrics run -it test \
    --image=appropriate/curl \
    --restart=Never \
    --rm \
    -- prometheus-node-exporter:9100/metrics
```

We created a new Pod based on `appropriate/curl`. That image serves a single purpose of providing `curl`. We specified `prometheus-node-exporter:9100/metrics` as the command, which is equivalent to running `curl` with that address. As a result, a lot of metrics were output. They are all in the same key/value format with optional labels surrounded by curly braces (`{`

same key/value format with optional labels surrounded by curly braces (`{` and `}`). On top of each metric, there is a `HELP` entry that explains its function as well as `TYPE` (.e.g, `gauge`). One of the metrics is as follows.

```
# HELP node_memory_MemTotal_bytes Memory information field MemTotal_bytes.
# TYPE node_memory_MemTotal_bytes gauge
node_memory_MemTotal_bytes 3.878477824e+09
```

We can see that it provides `Memory information field MemTotal_bytes` and that the type is `gauge`. Below the `TYPE` is the actual metric with the key (`node_memory_MemTotal_bytes`) and value `3.878477824e+09`.

Looking in the `prometheus-kube-state-metrics` exporter

Most Node Exporter metrics are without labels. So, we'll have to look for an example in the `prometheus-kube-state-metrics` exporter.

```
kubectl -n metrics run -it test \
    --image=appropriate/curl \
    --restart=Never \
    --rm \
    -- prometheus-kube-state-metrics:8080/metrics
```

As you can see, the Kube State metrics follow the same pattern as those from the Node Exporter. The major difference is that most of them do have labels. An example is as follows.

```
kube_deployment_created{deployment="prometheus-server",namespace="metrics"
} 1.535566512e+09
```

That metric represents the time the Deployment `prometheus-server` was created inside the `metrics` Namespace.

I'll leave it to you to explore those metrics in more detail. We'll use quite a few of them soon.

For now, just remember that with the combination of the metrics coming from the Node Exporter, Kube State Metrics, and those coming from Kubernetes itself, we can cover most of our needs. Or, to be more precise, those provide data required for most of the basic and common use cases.