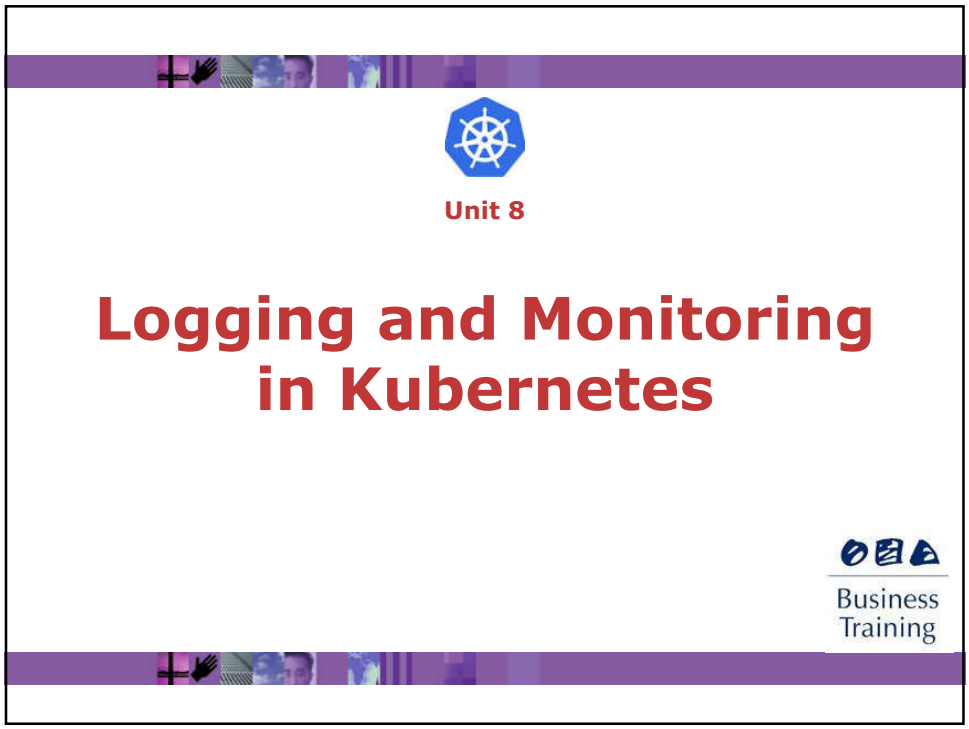


1

Outline

- Container logging
- Kubernetes component logging
- Monitoring with Prometheus and Grafana
- Troubleshooting

2

Container logging

3

Types of Kubernets Logging

- **Within a Kubernetes system, we can name three types of logs:**
  1. Container logs,
  2. Node logs, and
  3. Cluster (or system component) logs.
- **Container logs are logs generated by the containerized applications.**
- **Node logs are collected by the Kubelet running on each Kubernetes node from the stdout and stderr of each running pod . They are combined them into a log file that is managed by Kubernetes.**
- **Cluster logs refer to Kubernetes itself and all of its system component logs, and we can differentiate between components that run in a container and components that do not run in a container.**

4

Container Logging

- Container logging are the first layer of logs that can be collected from a Kubernetes cluster are those being generated by your containerized applications.
- The easiest method for logging containers is to write to the standard output (stdout) and standard error (stderr) streams.
- Let's take a look at an example pod manifest that will result in running one container logging to stdout:

```
apiVersion: v1
kind: Pod
metadata:
name: example
spec:
containers:
- name: example
image: busybox
args: [/bin/sh, -
c, 'while true; do echo $(date); sleep 1;
 done']
```


5

Using a Sidecar Container

- For persisting container logs, the common approach is to write logs to a log file and then use a sidecar container
  - A sidecar container will run in the same pod along with the application container, mounting the same volume and processing the logs separately.
- Viewing the logs

```
kubectl logs POD
kubectl logs --since=1h POD
kubectl logs POD –previous // For
     // for a previous instantiation
     // of a container
kubectl logs -f POD // Follow
                  //the stream
kubectl -n NAMESPACE logs
        POD -c CONTAINER
kubectl logs --since=1h -f POD |
grep Exception
```

```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
  - name: example
    image: busybox
    args:
    - /bin/sh
    - -c
    - >
      while true;  do
        echo "$(date)\n" >> /var/log/example.log;
        sleep 1;
      done
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  - name: sidecar
    image: busybox
    args: [/bin/sh, -c, 'tail -f /var/log/example.log']
    volumeMounts:
    - name: varlog
      mountPath: /var/log
  volumes:
  - name: varlog
    emptyDir: {}
```


6

Kubernetes component logging

7

Node Logging

- Everything that a containerized application writes to stdout or stderr is streamed somewhere by the container engine – in Docker's case, for example, to a logging driver.
- These logs are usually located in the /var/log/containers directory on your host.
- To prevent logs from filling up all the available space on the node, Kubernetes has a log rotation policy set in place.
- Depending on your operating system and services, there are various node-level logs you can collect, such as kernel logs or systemd logs.
  - You can access systemd logs with the `journalctl` command.

8

Cluster Logging

- **Kubernetes cluster logs refer to Kubernetes itself and all of its system component logs, and we can differentiate between components that run in a container and components that do not run in a container.**
  - For example, kube-scheduler, kube-apiserver, etcd, and kube-proxy run inside a container, while kubelet and the container runtime run on the operating system level, usually, as a systemd service.
- **By default, system components outside a container write files to journald, while components running in containers write to /var/log directory.**
  - However, there is the option to configure the container engine to stream logs to a preferred location.
- **Kubernetes doesn't provide a native solution for logging at cluster level. However, there are other approaches available to you:**
  - Use a node-level logging agent that runs on every node
  - Add a sidecar container for logging within the application pod
  - Expose logs directly from the application.

9

Kubernetes Events Logging

- **Kubernetes events are objects that show you what is happening inside a cluster, such as what decisions were made by the scheduler or why some pods were evicted from the node. All core components and extensions (operators) may create events through the API Server.**
  - Events are stored on the master. Similar to node logging, there is a removal mechanism set in place to avoid using all the master's disk space. Therefore, Kubernetes removes events an hour after the last occurrence. If you want to capture events over a longer period, you need to install a third-party solution.
- **Access event logs**
  - You can quickly check the events in a namespace by running:

    ```
    kubectl get events -n <namespace>
    ```
  - You can use describe command as well (the output contains the events)

    ```
    kubectl describe pod <pod-name>
    ```


10

Kubernetes logging tools

- **Fluentd**
  - Fluentd is a popular open-source log aggregator that allows you to collect various logs from your Kubernetes cluster, process them, and then ship them to a data storage backend of your choice.
  - Fluentd is Kubernetes-native and integrates seamlessly with Kubernetes deployments. The most common method for deploying fluentd is as a daemonset which ensures a fluentd pod runs on each pod.
- **ELK Stack**
  - The ELK Stack (Elasticsearch, Logstash and Kibana) is another very popular open-source tool used for logging Kubernetes
- **Google Stackdriver**
  - Stackdriver is another Kubernetes-native logging tool that provides users with a centralized logging solution.

11

Monitoring with Prometheus and Grafana

12

Introduction to application monitoring

- **Monitoring an application's health and metrics helps us manage it better, notice unoptimized behavior and get closer to its performance.**
  - This especially holds true when we're developing a system with many microservices, where monitoring each service can prove to be crucial when it comes to maintaining our system.
- **Based on this real-time monitoring information, we can draw conclusions and decide which microservice needs to scale if further performance improvements can't be achieved with the current setup.**

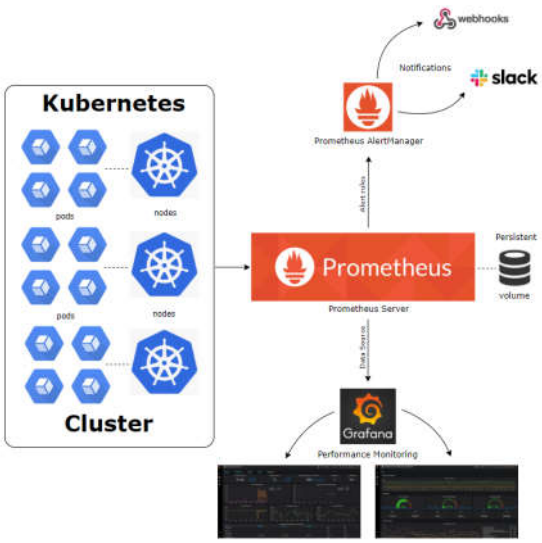
Kubernetes Metrics API Server

- **Metric server collects metrics such as CPU and Memory by each pod and node from the Summary API, exposed by Kubelet on each node.**
- **Metrics Server registered in the main API server through Kubernetes aggregator, which was introduced in Kubernetes 1.7**
- **Used with horizontal pod autoscaler, kube-dashboard, kubectl etc**

What is Prometheus ? What is Grafana ?

- **Prometheus is an open-source monitoring system that was originally built by SoundCloud. It consists of the following core components -**
  - **A data scraper** that pulls metrics data over HTTP periodically at a configured interval.
  - A **time-series database** to store all the metrics data.
  - A **simple user interface** where you can visualize, query, and monitor all the metrics.
- **Grafana Grafana allows you to bring data from various data sources like Elasticsearch, Prometheus, Graphite, InfluxDB etc, and visualize them with beautiful graphs.**
  - It also lets you set alert rules based on your metrics data. When an alert changes state, it can notify you over email, slack, or various other channels. Prometheus dashboard also has simple graphs. But Grafana's graphs are way better.

Prometheus integration in Kubernetes



Installing Prometheus and Grafana with Helm 3

- **Add repository of stable charts. Helm3 has not default repository**
  - helm repo add stable https://kubernetes-charts.storage.googleapis.com
- **Install prometheus-operator. The Prometheus-operator char contains Grafana.**

```
helm install my-prometheus-operator stable/prometheus-operator
```

- **Show pods**

```
kubectl --namespace default get pods -l
       "release=my-prometheus-operator "
```

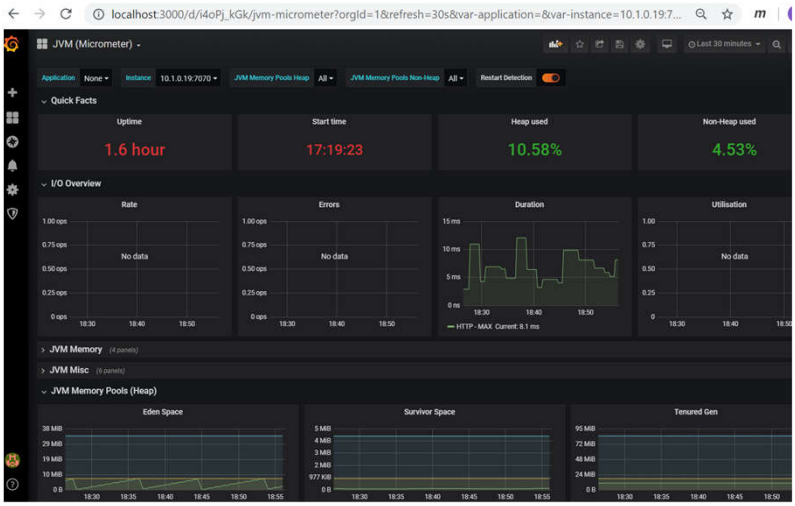
- **Show Grafana UI**

```
kubectl port-forward $(kubectl get pods --selector=app=grafana
        --output=jsonpath="{.items..metadata.name}") 3000
```

- **Open the Grafana dashboard** http://localhost:3000/
  - Default credentials for Grafana: login/pass: admin/prom-operator

17

Monitoring a Java/Spring Application using Grafana



18

Troubleshooting

19

Diagnosing the problem

- **The first step in troubleshooting is triage. What is the problem? Is it your Pods, your Replication Controller or your Service?**
  - Debugging Pods
  - Debugging Replication Controllers
  - Debugging Services

20

Debugging Pods

- **The first step in debugging a Pod is taking a look at it. Check the current state of the Pod and recent events with the following command:**
  - `kubectl describe pods ${POD_NAME}`
- **Look at the state of the containers in the pod. Are they all Running? Have there been recent restarts?**
  - Continue debugging depending on the state of the pods.
- **My pod stays pending**
  - If a Pod is stuck in Pending it means that it can not be scheduled onto a node. Generally this is because there are **insufficient resources** of one type or another that prevent scheduling. Look at the output of the **kubectl describe ... command** above. There should be messages from the scheduler about why it can not schedule your pod.
- **My pod stays waiting**
  - If a Pod is stuck in the Waiting state, then it has been scheduled to a worker node, but it can't run on that machine. Again, the information from kubectl describe ... should be informative. The most common cause of Waiting pods is a failure to pull the image.

Debugging Pods

- **My pod is crashing or otherwise unhealthy**
  - Once your pod has been scheduled, the methods described in Debug Running Pods are available for debugging.
    - First, look at the logs of the affected container:
      `kubectl logs ${POD_NAME} ${CONTAINER_NAME}`
    - If your container has previously crashed, you can access the previous container's crash log with:
      `kubectl logs --previous ${POD_NAME} ${CONTAINER_NAME}`
- **My pod is running but not doing what I told it to do**
  - If your pod is not behaving as you expected, it may be that there was an error in your pod description (e.g. mypod.yaml file on your local machine), and that the error was silently ignored when you created the pod.
  - Often a section of the pod description is **nested incorrectly**, or a **key name is typed incorrectly**, and so the key is ignored. For example, if you misspelled command as **commnd** then the pod will be created but will not use the command line you intended it to use.
    - The first thing to do is to delete your pod and try creating it again with the --validate option. For example, run `kubectl apply --validate -f mypod.yaml`. If you misspelled command as commnd then will give an error.
    - The next thing to check is whether the pod on the apiserver matches the pod you meant to create (e.g. in a yaml file on your local machine). For example, run kubectl get pods/mypod -o yaml > mypod-on-apiserver.yaml and then manually compare the descriptions..

Debugging Replication Controllers

- **Replication controllers are fairly straightforward.**
  - They can either create Pods or they can't. If they can't create pods, then please refer to the instructions above to debug your pods.
  - You can also use `kubectl describe rc ${CONTROLLER_NAME}` to introspect events related to the replication controller.

Debugging Services

- **Services provide load balancing across a set of pods. There are several common problems that can make Services not work properly.**
  - First, verify that there are endpoints for the service. For every Service object, the apiserver makes an endpoints resource available.
    - You can view this resource with: `kubectl get endpoints ${SERVICE_NAME}`
  - Make sure that the endpoints match up with the number of containers that you expect to be a member of your service. For example, if your Service is for an nginx container with 3 replicas, you would expect to see three different IP addresses in the Service's endpoints.

- **My service is missing endpoints**
  - If you are missing endpoints, try listing pods using the labels that Service uses. Imagine that you have a Service where the labels are:

```
...
spec:
  - selector:
      name: nginx
      type: frontend
```

    - You can use: `kubectl get pods --selector=name=nginx,type=frontend`
    - to list pods that match this selector. Verify that the list matches the Pods that you expect to provide your Service.
    - If the list of pods matches expectations, but your endpoints are still empty, it's possible that you don't have the right ports exposed.

Debugging Services

- **Network traffic is not forwarded**
  - If you can connect to the service, but the connection is immediately dropped, and there are endpoints in the endpoints list, it's likely that the proxy can't contact your pods.
  - There are three things to check:
    - Are your pods working correctly? Look for restart count, and debug pods.
    - Can you connect to your pods directly? Get the IP address for the Pod, and try to connect directly to that IP.
    - Is your application serving on the port that you configured? Kubernetes doesn't do port remapping, so if your application serves on 8080, the containerPort field needs to be 8080.
- **If none of the above solves your problem, follow the instructions in Debugging Service document to make sure that your Service is running, has Endpoints, and your Pods are actually serving; you have DNS working, iptables rules installed, and kube-proxy does not seem to be misbehaving.**