Learning Objectives

By the end of this chapter, you should be able to:

- Understand that Kubernetes does not yet have integrated logging.
- Learn which external products are often used to aggregate logs.
- Examine the basic flow of troubleshooting.
- Discuss the use of a sidecar for in-Pod logs.

LOGGING AND TROUBLESHOOTING

Overview

Kubernetes relies on API calls and is sensitive to network issues. Standard Linux tools and processes are the best method for troubleshooting your cluster. If a shell, such as bash, is not available in an affected Pod, consider deploying another similar pod with a shell, like **busybox**. DNS configuration files and tools like **dig** are a good place to start. For more difficult challenges, you may need to install other tools, like **tcpdump**.

Large and diverse workloads can be difficult to track, so monitoring of usage is essential. Monitoring is about collecting key metrics, such as CPU, memory, and disk usage, and network bandwidth on your nodes, as well as monitoring key metrics in your applications. These features are being ingested into Kubernetes with the Metric Server, which is a cut-down version of the now deprecated Heapster. Once installed, the Metrics Server exposes a standard API which can be consumed by other agents, such as autoscalers. Once installed, this endpoint can be found here on the master server: **/apis/metrics/k8s.io/**.

Logging activity across all the nodes is another feature not part of Kubernetes. Using Fluentd can be a useful data collector for a unified logging layer. Having aggregated logs can help visualize the issues, and provides the ability to search all logs. It is a good place to start when local network troubleshooting does not expose the root cause. It can be downloaded from the Fluentd website.

Another project from CNCF combines logging, monitoring, and alerting and is called Prometheus - you can learn more from the Prometheus website. It provides a time-series database, as well as integration with Grafana for visualization and dashboards.

We are going to review some of the basic **kubectl** commands that you can use to debug what is happening, and we will walk you through the basic steps to be able to debug your containers, your pending containers, and also the systems in Kubernetes.

Basic Troubleshooting Steps

The troubleshooting flow should start with the obvious. If there are errors from the command line, investigate them first. The symptoms of the issue will probably determine the next step to check. Working from the application running inside a container to the cluster as a whole may be a good idea. The application may have a shell you can use, for example:

```
$ kubectl create deploy busybox --image=busybox --command sleep 3600

$ kubectl exec -ti <busybox_pod> -- /bin/sh
```

If the Pod is running, use **kubectl logs pod-name** to view the standard out of the container. Without logs, you may consider deploying a sidecar container in the Pod to generate and handle logging. The next place to check is networking, including DNS, firewalls and general connectivity, using standard Linux commands and tools.

Security settings can also be a challenge. RBAC, covered in the security chapter, provides mandatory or discretionary access control in a granular manner. SELinux and AppArmor are also common issues, especially with network-centric applications.

A newer feature of Kubernetes is the ability to enable auditing for the kube-apiserver, which can allow a view into actions after the API call has been accepted.

The issues found with a decoupled system like Kubernetes are similar to those of a traditional datacenter, plus the added layers of Kubernetes controllers:

- Errors from the command line
- Pod logs and state of Pods
- Use shell to troubleshoot Pod DNS and network
- Check node logs for errors, make sure there are enough resources allocated
- RBAC, SELinux or AppArmor for security settings
- API calls to and from controllers to kube-apiserver
- Enable auditing
- Inter-node network issues, DNS and firewall
- Master server controllers (control Pods in pending or error state, errors in log files, sufficient resources, etc).

Ephemeral Containers

A feature new to the 1.16 version is the ability to add a container to a running pod. This would allow a feature-filled container to be added to an existing pod without having to terminate and re-create. Intermittent and difficult to determine problems may take a while to reproduce, or not exist with the addition of another container.

As an alpha stability feature, it may change or be removed at any time. As well, they will not be restarted automatically, and several resources such as ports or resources are not allowed.

These containers are added via the **ephemeralcontainers** handler via an API call, not via the **podSpec**. As a result, the use of **kubectl edit** is not possible.

You may be able to use the **kubectl attach** command to join an existing process within the container. This can be helpful instead of **kubectl exec**, which executes a new process. The functionality of the attached process depends entirely on what you are attaching to.

```
kubectl debug buggypod --image debian --attach
```

Cluster Start Sequence

The cluster startup sequence begins with systemd if you built the cluster using **kubeadm**. Other tools may leverage a different method. Use **systemctl status kubelet.service** to see the current state and configuration files used to run the kubelet binary.

The cluster startup sequence begins with systemd if you built the cluster using **kubeadm**. Other tools may leverage a different method. Use **systemctl status kubelet.service** to see the current state and configuration files used to run the kubelet binary.

- Uses **/etc/systemd/system/kubelet.service.d/10-kubeadm.conf**

Inside of the **config.yaml** file you will find several settings for the binary, including the **staticPodPath** which indicates the directory where kubelet will read every yaml file and start every pod. If you put a yaml file in this directory, it is a way to troubleshoot the scheduler, as the pod is created with any requests to the scheduler.

- Uses **/var/lib/kubelet/config.yaml** configuration file
- **staticPodPath** is set to **/etc/kubernetes/manifests/**

The four default yaml files will start the base pods necessary to run the cluster:

- kubelet creates all pods from *.yaml in directory: kube-apiserver, etcd, kube-controller-manager, kube-scheduler.

Once the watch loops and controllers from kube-controller-manager run using etcd data, the rest of the configured objects will be created.

Monitoring

Monitoring is about collecting metrics from the infrastructure, as well as applications.

The long used and now deprecated Heapster has been replaced with an integrated Metrics Server. Once installed and configured, the server exposes a standard API which other agents can use to determine usage. It can also be configured to expose custom metrics, which then could also be used by autoscalers to determine if an action should take place.

Plugins

We have been using the **kubectl** command throughout the course. The basic commands can be used together in a more complex manner extending what can be done. There are over seventy and growing plugins available to interact with Kubernetes objects and components.

At the time this course was written, plugins cannot overwrite existing **kubectl** commands, nor can it add sub-commands to existing commands. Writing new plugins should take into account the command line runtime package and a Go library for plugin authors.

As a plugin the declaration of options such as namespace or container to use must come after the command.

```
$ kubectl sniff bigpod-abcd-123 -c mainapp -n accounting
```

Plugins can be distributed in many ways. The use of **krew** (the **kubectl** plugin manager) allows for cross-platform packaging and a helpful plugin index, which makes finding new plugins easy.

Install the software using steps available in krew's GitHub repository.

```
$ kubectl krew help
```

You can invoke **krew** through kubectl:

```
kubectl krew [command]...
```

Usage:

```
krew [command]
```

Available Commands

COMMAND	DESCRIPTION
help	Help about any command
info	Show information about kubectl plugin
install	Install kubectl plugins
list	List installed kubectl plugins
search	Discover kubectl plugins
uninstall	Uninstall plugins
update	Update the local copy of the plugin index
upgrade	Upgrade installed plugins to newer versions
version	Show krew version and diagnostics

Managing Plugins

The **help** option explains basic operation. After installation ensure the $PATH includes the plugins. **krew** should allow easy installation and use after that.

```
$ export PATH="${KREW_ROOT:-$HOME/.krew}/bin:$PATH"
$ kubectl krew search

NAME              DESCRIPTION
INSTALLED
access-matrix     Show an RBAC access matrix for server resources       no
advise-psp        Suggests PodSecurityPolicies for cluster.             no
....

$ kubectl krew install tail

Updated the local copy of plugin index.
Installing plugin: tail
Installed plugin: tail
\
 | Use this plugin:

....

 | | Usage:
 | |
 | | # match all pods
 | | $ kubectl tail
 | |
 | | # match pods in the 'frontend' namespace
 | | $ kubectl tail --ns staging
....
```

In order to view the current plugins use:

```
kubectl plugin list
```

To find new plugins use:

```
kubectl krew search
```

To install use:

```
kubectl krew install new-plugin
```

Once installed use as **kubectl** sub-command. You can also upgrade and uninstall.

Sniffing Traffic With Wireshark

Cluster network traffic is encrypted making troubleshooting of possible network issues more complex. Using the sniff plugin you can view the traffic from within. sniff requires Wireshark and ability to export graphical display.

The **sniff** command will use the first found container unless you pass the **-c** option to declare which container in the pod to use for traffic monitoring.

```
$ kubectl krew install sniff nginx-123456-abcd -c webcont
```

Logging Tools

Logging, like monitoring, is a vast subject in IT. It has many tools that you can use as part of your arsenal.

Typically, logs are collected locally and aggregated before being ingested by a search engine and displayed via a dashboard which can use the search syntax. While there are many software stacks that you can use for logging, the Elasticsearch, Logstash, and Kibana Stack (ELK) has become quite common.

In Kubernetes, the kubelet writes container logs to local files (via the Docker logging driver). The **kubectl logs** command allows you to retrieve these logs.

Cluster-wide, you can use Fluentd to aggregate logs. Check out the cluster administration logging concepts for a detailed description.

Fluentd is part of the Cloud Native Computing Foundation and, together with Prometheus, they make a nice combination for monitoring and logging. You can find a detailed walk-through of running Fluentd on Kubernetes in the Kubernetes documentation.