LFS242 - Cloud Native Logging with Fluentd

Fluentd is a cross platform open-source data collector. It is written primarily in the Ruby programming language with several components developed in C for performance. Fluentd can be used to create a unified logging layer, allowing users to unify data collection and consumption across a wide array of systems and services.

Fluentd can be installed and configured for a variety of environments. Lab 1 is organized into three separate parts, each involves the installation and configuration steps necessary to get started with Fluentd in a different environment:

- A. Linux
- B. Docker
- C. Kubernetes

Lab 1-C – Running Fluentd in a Kubernetes environment

In this lab you will get a chance to run Fluentd using the Kubernetes container orchestration system. This lab is designed to be independent of Lab 1-A and Lab 1-B and can be run on a separate virtual machine, though you can run these steps on same machine as Lab 1-B.

Kubernetes (k8s) is an open-source system for automating deployment, scaling, and management of containerized applications. It groups containers that make up an application into logical units for easy management and discovery. Kubernetes builds upon 15 years of experience of running production workloads at Google, combined with best-of-breed ideas and practices from the Kubernetes community.

In this lab we will see how Fluentd can play a key role in the log management of containerized applications running under Kubernetes.

Objectives

- Learn how to install a simple Kubernetes test cluster.
- Learn how to run Fluentd on Kubernetes.
- Learn how to process logs from containerized applications with Fluentd on Kubernetes.

1. Prepare the lab system

If you have not already, log in to your lab system and ensure that any Fluentd instances or other programs started in prior labs are stopped. Also stop any other containers you have on the system.

In order to work with Fluentd on Kubernetes we will need to install Kubernetes. Kubernetes requires a container runtime like Docker to operate. If you have not yet installed Docker on your lab system, do so now using the Docker quick-start script:

```
ubuntu@labsys:~$ wget -O - https://get.docker.com | sh

...

ubuntu@labsys:~$
```

If your lab system has less than 4GB of memory you will get faster performance if you increase it to 4GB or more. To increase the memory you will need to shut down the virtual machine (`$ sudo shutdown -h now`), increase the memory through the hypervisor (VMwarePlayer/Workstation/Fusion, Virtual Box, etc.) and then restart the system.

Computer operating systems have traditionally used disk space as a place to swap out less used pages of memory when the system is low on memory. These swap/page files/volumes worked well in older computers which operated with very small amounts of memory. Modern systems are not typically as constrained.

Kubernetes is designed to make efficient use of a computer cluster, so adding more memory is as easy as adding another computer. Disk

access is much slower than actual memory access, so Kubernetes expects memory swap to be disabled.

Turn off memory swapping on your system:

> N.B If you are running this lab on a cloud instance, this step may not be necessary:

```
ubuntu@labsys:~$ sudo swapoff -a

ubuntu@labsys:~$
```

The `swapoff` command disables memory swapping to a particular file or device and the `-a` switch disables "all". If your system is not configured for swapping this command does nothing.

If swap is configured it will be reenabled upon your next reboot. To disable swap permanently update your file system table configuration file, commenting out any volumes listed as "swap" (and there may be none):

```
ubuntu@labsys:~$ cat /etc/fstab

# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/sda1 during installation
UUID=ae4d6013-3015-4619-a301-77a55030c060 /               ext4    errors=remount-ro 0       1
# swap was on /dev/sda5 during installation
UUID=70f4d3ab-c8a1-48f9-bf47-2e35e4d4275f none            swap    sw              0       0

ubuntu@labsys:~$ sudo nano /etc/fstab && cat $_

# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/sda1 during installation
UUID=ae4d6013-3015-4619-a301-77a55030c060 /               ext4    errors=remount-ro 0       1
# swap was on /dev/sda5 during installation
# UUID=70f4d3ab-c8a1-48f9-bf47-2e35e4d4275f none            swap    sw              0       0

ubuntu@labsys:~$
```

2. Kubernetes Pre-Installation Steps

This lab will have you install Kubernetes using the standard `kubeadm` tool. Before running the Kubernetes installation, you need to install `kubeadm` and its dependencies as well as prepare Docker to interact with Kubernetes.

2a. Install the packages

To begin, update your package index:

```
ubuntu@labsys:~$ sudo apt-get update
```

```
Hit:1 http://us-east-1.ec2.archive.ubuntu.com/ubuntu focal InRelease
Hit:2 http://us-east-1.ec2.archive.ubuntu.com/ubuntu focal-updates InRelease
Hit:3 http://us-east-1.ec2.archive.ubuntu.com/ubuntu focal-backports InRelease
Hit:4 https://download.docker.com/linux/ubuntu focal InRelease
Hit:5 http://security.ubuntu.com/ubuntu focal-security InRelease
Reading package lists... Done

ubuntu@labsys:~$
```

Kubernetes packages are installed via https, so install `apt-https-transport` (it may already be installed):

```
ubuntu@labsys:~$ sudo apt-get install -y apt-transport-https

Reading package lists... Done
Building dependency tree
Reading state information... Done
apt-transport-https is already the newest version (2.0.8).
0 upgraded, 0 newly installed, 0 to remove and 61 not upgraded.

ubuntu@labsys:~$
```

Next add the Kubernetes package repository to the local system's package index. To ensure that apt can verify the Kubernetes packages add the repository key:

```
ubuntu@labsys:~$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

OK

ubuntu@labsys:~$
```

Now add the Kubernetes package URL to the apt sources list:

```
ubuntu@labsys:~$ echo "deb http://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee -a
/etc/apt/sources.list.d/kubernetes.list

deb http://apt.kubernetes.io/ kubernetes-xenial main

ubuntu@labsys:~$
```

Great. Now that we have added the Kubernetes package repository update the local package index to include the packages that can be found there:

```
ubuntu@labsys:~$ sudo apt-get update

...

Get:5 https://packages.cloud.google.com/apt kubernetes-xenial InRelease [9383 B]
Get:7 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 Packages [56.5 kB]
Fetched 65.9 kB in 1s (92.7 kB/s)
Reading package lists... Done

ubuntu@labsys:~$
```

You should see apt report updates from `kubernetes-xenial`.

Now we can install the core Kubernetes packages:

- kubelet - the node manager for Kubernetes
- kubeadm - the Kubernetes cluster control plane installer
- kubectl - the Kubernetes admin command line tool

Install the core packages:

> N.B. To ensure lab stability the lab instructions will use Kubernetes 1.24.0. You may omit the `=1.24.0-00` parts of the command or substitute it with a later or earlier version.

```
ubuntu@labsys:~$ sudo apt-get install -y kubelet=1.24.0-00 kubeadm=1.24.0-00 kubectl=1.24.0-00

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  conntrack cri-tools ebtables kubernetes-cni socat
Suggested packages:
  nftables
The following NEW packages will be installed:
  conntrack cri-tools ebtables kubeadm kubectl kubelet kubernetes-cni socat
0 upgraded, 8 newly installed, 0 to remove and 61 not upgraded.

...

Setting up conntrack (1:1.4.5-2) ...
Setting up kubectl (1.24.0-00) ...
Setting up ebtables (2.0.11-3build1) ...
Setting up socat (1.7.3.3-2) ...
Setting up cri-tools (1.24.0-00) ...
Setting up kubernetes-cni (0.8.7-00) ...
Setting up kubelet (1.24.0-00) ...
Created symlink /etc/systemd/system/multi-user.target.wants/kubelet.service →
/lib/systemd/system/kubelet.service.
Setting up kubeadm (1.24.0-00) ...
Processing triggers for man-db (2.9.1-1) ...

ubuntu@labsys:~$
```

2b. Prepare the container runtime

Next, you need to complete the following steps to enable Docker as the container runtime for Kubernetes:

- Configure Docker to use systemd as the cgroup driver
- Install `cri-dockerd`, the interface that allows Docker to communicate with the Kubelet.

Create a file at `/etc/docker/daemon.json` with the following contents, then restart Docker:

```
ubuntu@labsys:~$ sudo mkdir -p /etc/docker

ubuntu@labsys:~$ sudo nano /etc/docker/daemon.json && cat $_
```

```
{
  "exec-opts": ["native.cgroupdriver=systemd"]
}
```

```
ubuntu@labsys:~$ sudo systemctl restart docker

ubuntu@labsys:~$
```

Next, install the `cri-dockerd` interface, which consists of a binary, a systemd unit file, and a socket definition.

Download the latest `cri-dockerd` binary:

```
ubuntu@labsys:~$ wget https://github.com/Mirantis/cri-dockerd/releases/download/v0.2.2/cri-dockerd-
0.2.2.amd64.tgz

...

ubuntu@labsys:~$
```

Unpack the contents of the tarball and move them to your machine's `/usr/bin` directory:

```
ubuntu@labsys:~$ tar xvf cri-dockerd-0.2.2.amd64.tgz

cri-dockerd/
cri-dockerd/cri-dockerd

ubuntu@labsys:~$ sudo mv cri-dockerd /usr/bin/

ubuntu@labsys:~$
```

Next, set up the systemd unit which actually runs the `cri-dockerd` binary:

```
ubuntu@labsys:~$ sudo nano /etc/systemd/system/cri-docker.service && sudo cat $_
```

```
[Unit]
Description=CRI Interface for Docker Application Container Engine
Documentation=https://docs.mirantis.com
After=network-online.target firewalld.service docker.service
Wants=network-online.target
Requires=cri-docker.socket

[Service]
Type=notify
ExecStart=/usr/bin/cri-dockerd/cri-dockerd --container-runtime-endpoint fd:// --network-plugin=cni -
-cni-bin-dir=/opt/cni/bin --cni-cache-dir=/var/lib/cni/cache --cni-conf-dir=/etc/cni/net.d
ExecReload=/bin/kill -s HUP $MAINPID
TimeoutSec=0
RestartSec=2
Restart=always

StartLimitBurst=3
StartLimitInterval=60s

LimitNOFILE=infinity
LimitNPROC=infinity
LimitCORE=infinity

TasksMax=infinity
Delegate=yes
```

```
KillMode=process

[Install]
WantedBy=multi-user.target
```

```
ubuntu@labsys:~$
```

This systemd unit includes options that instruct `cri-dockerd` to use a container network interface (CNI) plugin to provide IP addresses to workloads orchestrated by Kubernetes.

After that, set up the socket which the Kubelet uses to communicate with the `cri-dockerd` service:

```
ubuntu@labsys:~$ sudo nano /etc/systemd/system/cri-docker.socket && sudo cat $_
```

```
[Unit]
Description=CRI Docker Socket for the API
PartOf=cri-docker.service

[Socket]
ListenStream=%t/cri-dockerd.sock
SocketMode=0660
SocketUser=root
SocketGroup=docker

[Install]
WantedBy=sockets.target
```

```
ubuntu@labsys:~$
```

Finally, enable both the `cri-dockerd` socket and service:

```
ubuntu@labsys:~$ sudo systemctl daemon-reload

ubuntu@labsys:~$ sudo systemctl enable cri-docker.service

Created symlink /etc/systemd/system/multi-user.target.wants/cri-docker.service →
/etc/systemd/system/cri-docker.service.

ubuntu@labsys:~$ sudo systemctl enable --now cri-docker.socket

Created symlink /etc/systemd/system/sockets.target.wants/cri-docker.socket →
/etc/systemd/system/cri-docker.socket.

ubuntu@labsys:~$
```

The Kubelet can now communicate with Docker through the `cri-dockerd` interface.

Great, we are ready to install a Kubernetes cluster.

3. Configure a Kubernetes cluster

The kubeadm command has an init subcommand we can use to setup a Kubernetes cluster on our lab system. Run `kubeadm init` with the `--kubernetes-version 1.24.0` option to declare which version of Kubernetes to use and `--cri-socket=unix:///run/cri-dockerd.sock` option to declare which container runtime to use:

```
ubuntu@labsys:~$ sudo kubeadm init --kubernetes-version 1.24.0 --cri-socket=unix:///run/cri-
dockerd.sock

[init] Using Kubernetes version: v1.24.0
[preflight] Running pre-flight checks
[preflight] Pulling images required for setting up a Kubernetes cluster
[preflight] This might take a minute or two, depending on the speed of your internet connection
[preflight] You can also perform this action in beforehand using 'kubeadm config images pull'
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [labsys kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 172.31.9.34]
[certs] Generating "apiserver-kubelet-client" certificate and key
[certs] Generating "front-proxy-ca" certificate and key
[certs] Generating "front-proxy-client" certificate and key
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [labsys localhost] and IPs [172.31.9.34
127.0.0.1 ::1]
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [labsys localhost] and IPs [172.31.9.34
127.0.0.1 ::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "sa" key and public key
[kubeconfig] Using kubeconfig folder "/etc/kubernetes"
[kubeconfig] Writing "admin.conf" kubeconfig file
[kubeconfig] Writing "kubelet.conf" kubeconfig file
[kubeconfig] Writing "controller-manager.conf" kubeconfig file
[kubeconfig] Writing "scheduler.conf" kubeconfig file
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-
flags.env"
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Starting the kubelet
[control-plane] Using manifest folder "/etc/kubernetes/manifests"
[control-plane] Creating static Pod manifest for "kube-apiserver"
[control-plane] Creating static Pod manifest for "kube-controller-manager"
[control-plane] Creating static Pod manifest for "kube-scheduler"
[etcd] Creating static Pod manifest for local etcd in "/etc/kubernetes/manifests"
[wait-control-plane] Waiting for the kubelet to boot up the control plane as static Pods from
directory "/etc/kubernetes/manifests". This can take up to 4m0s
[apiclient] All control plane components are healthy after 12.503704 seconds
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system"
Namespace
[kubelet] Creating a ConfigMap "kubelet-config" in namespace kube-system with the configuration for
the kubelets in the cluster
[upload-certs] Skipping phase. Please see --upload-certs
[mark-control-plane] Marking the node labsys as control-plane by adding the labels: [node-
role.kubernetes.io/control-plane node.kubernetes.io/exclude-from-external-load-balancers]
[mark-control-plane] Marking the node labsys as control-plane by adding the taints [node-
role.kubernetes.io/master:NoSchedule node-role.kubernetes.io/control-plane:NoSchedule]
[bootstrap-token] Using token: y70mwc.6rm1i1hccyt42uuq
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to get nodes
[bootstrap-token] Configured RBAC rules to allow Node Bootstrap tokens to post CSRs in order for
nodes to get long term certificate credentials
[bootstrap-token] Configured RBAC rules to allow the csrapprover controller automatically approve
CSRs from a Node Bootstrap Token
[bootstrap-token] Configured RBAC rules to allow certificate rotation for all node client
certificates in the cluster
```

```
[bootstrap-token] Creating the "cluster-info" ConfigMap in the "kube-public" namespace
[kubelet-finalize] Updating "/etc/kubernetes/kubelet.conf" to point to a rotatable kubelet client
certificate and key
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes control-plane has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

Alternatively, if you are the root user, you can run:

  export KUBECONFIG=/etc/kubernetes/admin.conf

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

Then you can join any number of worker nodes by running the following on each as root:

kubeadm join 172.31.9.34:6443 --token y70mwc.6rm1i1hccyt42uuq \
        --discovery-token-ca-cert-hash
sha256:6db8ab9bcf87aa456bee75d86fd3d7ba6203225b935198ef02fef5ca4452cf60

ubuntu@labsys:~$
```

The initialization process creates a Kubernetes control plane node, which runs all of the components that make up a minimal but functional Kubernetes deployment. We will follow the suggestions `kubeadm init` makes after the installation completes in the coming steps.

To communicate with the cluster we will need credentials. Kubeadm generates administrator credentials and stores them in the file `/etc/kubernetes/admin.conf`. Per the kubeadm comments, copy the admin credentials over to your user account so that you can administer the cluster without sudo:

```
ubuntu@labsys:~$ mkdir -p $HOME/.kube

ubuntu@labsys:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

ubuntu@labsys:~$ sudo chown $(id -u):$(id -g) $HOME/.kube/config

ubuntu@labsys:~$
```

To make kubectl suggest commands and resource names to us, enable bash completion on your machine:

```
ubuntu@labsys:~$ sudo apt install bash-completion

...

ubuntu@labsys:~$ source <(kubectl completion bash)

ubuntu@labsys:~$ echo "source <(kubectl completion bash)" >> ~/.bashrc

ubuntu@labsys:~$
```

Now use the kubectl tool to get the status of your cluster:

```
ubuntu@labsys:~$ kubectl cluster-info

Kubernetes control plane is running at https://labsys:6443
CoreDNS is running at https://labsys:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

ubuntu@labsys:~$
```

Great, Kubernetes is running. There are still a few things we need to do before using our cluster.

Kubernetes control plane nodes initialized by `kubeadm init` are not expected to run regular workloads. By default, kubeadm configures a control plane node with taints, or reservations, which prevent regular application containers from being able to run on it. Since we will use our one node for all kinds of workloads, remove the taints by removing the `node-role.kubernetes.io/master` and `node-role.kubernetes.io/control-plane` keys with `kubectl taint`:

```
ubuntu@labsys:~$ kubectl taint nodes --all node-role.kubernetes.io/master- node-
role.kubernetes.io/control-plane-

node/ubuntu untainted

ubuntu@labsys:~$
```

Be careful not to skip the `-` at the end of each key. The minus specified that the taint should be removed as opposed to added.

As a final step we need to add a software defined networking solution to support our container networking needs. Kubernetes supports a wide range of networking solutions through the popular CNI, Container Networking Interface. We'll use the Weave CNI solution on our cluster. Now that we have a Kubernetes cluster running, we can actually install Weave as a container workload. Run the following command:

```
ubuntu@labsys:~$ kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version |
base64 | tr -d '\n')"

serviceaccount/weave-net created
clusterrole.rbac.authorization.k8s.io/weave-net created
clusterrolebinding.rbac.authorization.k8s.io/weave-net created
role.rbac.authorization.k8s.io/weave-net created
rolebinding.rbac.authorization.k8s.io/weave-net created
daemonset.apps/weave-net created

ubuntu@labsys:~$
```

> This command my generate a deprecation warning about `kubectl version --short`. The manifest is still correctly retrieved and this warning can be ignored.

The above command installs weave-net as a DaemonSet after creating its required security roles and bindings. Weave Net deploys all of its components using a single declarative specification file in YAML format.

Kubernetes packages containers in "pods". List all of the pods running on your Kubernetes cluster:

```
ubuntu@labsys:~$ kubectl get pods --all-namespaces
```

```
NAMESPACE     NAME                                 READY   STATUS    RESTARTS      AGE
kube-system   coredns-6d4b75cb6d-l5hs6             1/1     Running   0             4m20s
kube-system   coredns-6d4b75cb6d-shfl7             1/1     Running   0             4m20s
kube-system   etcd-labsys                          1/1     Running   0             4m25s
kube-system   kube-apiserver-labsys                1/1     Running   0             4m25s
kube-system   kube-controller-manager-labsys       1/1     Running   0             4m25s
kube-system   kube-proxy-nnc9h                     1/1     Running   0             4m21s
kube-system   kube-scheduler-labsys                1/1     Running   0             4m25s
kube-system   weave-net-gflnt                      2/2     Running   1 (46s ago)   53s

ubuntu@labsys:~$
```

If you see something like the output above, you cluster is ready to use. If, for example, the `coredns` pods are not yet running, you may need to give the `weave-net` pods more time to start. DNS will not be enabled until the node is ready, which happens after a network plugin is configured.

4. Creating a Fluentd ConfigMap

There are many platform related tools administrators want or need to run on every node in their environment. For example, the Weave SDN agent has to be running on every node in our Kubernetes cluster for containers to be able to communicate with each other. Kubernetes has a special deployment type, known as a DaemonSet, which runs one copy of a given pod on potentially every system in the cluster.

DaemonSets are also a great way to deploy Fluentd. If you want to have Fluentd running as a log forwarding agent on many nodes in your cluster a Fluentd DaemonSet is the right choice. DaemonSets operate at the control plane level so when new nodes are added to the cluster, all of the DaemonSet pods are automatically launched on the node. This greatly simplifies cluster maintenance.

In this step we will deploy Fluentd using a DaemonSet. Before we begin consider the Fluentd configuration file. In prior labs we passed Fluentd a configuration file from the local host's filesystem. This is not going to work well in a Kubernetes cluster. Imagine copying a configuration file to hundreds of host and then trying to audit and maintain all of them. Not a great prospect.

Fortunately Kubernetes provides a resource known as ConfigMaps. A ConfigMap is a control plane managed resource that pods can access from anywhere in the cluster. ConfigMaps can contain any kind of typical configuration data including, data used for environment variables, passed on command lines, or stored in configuration files mapped into containers. If we save our Fluentd configuration file as a ConfigMap we can use it with every Fluentd pod in the cluster.

To initialize our ConfigMap we can start by simply creating a normal Fluentd configuration file:

```
ubuntu@labsys:~$ mkdir ~/fluent

ubuntu@labsys:~$ nano ~/fluent/fluentd-kube.conf && cat $_
```

```
<source>
  @type http
  port 24220
  bind 0.0.0.0
</source>

<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>

<match **>
  @type stdout
</match>
```

```
ubuntu@labsys:~$
```

Next, create a Kubernetes manifest that constructs a ConfigMap using the data in the config file. We can use the `kubectl create configmap` command to do this. Try it:

```
ubuntu@labsys:~$ kubectl create configmap fluentd-config --from-file ~/fluent/fluentd-kube.conf

configmap/fluentd-config created

ubuntu@labsys:~$
```

`kubectl` formulates an API request to create a ConfigMap containing the contents of the `~/fluent/fluentd-kube.conf` file, then sends it to the API server.

Use `kubectl get` to list your config map, then use `kubectl describe` to get a summary of its contents:

```
ubuntu@labsys:~$ kubectl get configmap

NAME               DATA   AGE
fluentd-config     1      6s
kube-root-ca.crt   1      6m22s

ubuntu@labsys:~$ kubectl describe configmap fluentd-config

Name:         fluentd-config
Namespace:    default
Labels:       <none>
Annotations:  <none>

Data
====
fluentd-kube.conf:
----
<source>
  @type http
  port 24220
  bind 0.0.0.0
</source>

<source>
  @type forward
  port 24224
  bind 0.0.0.0
</source>

<match **>
  @type stdout
</match>


BinaryData
====

Events:  <none>

ubuntu@labsys:~$
```

Looks good. The config file contents are stored under the ConfigMap's `fluentd-kube.conf` data key. Now we can run a Fluentd DaemonSet that uses the ConfigMap for its configuration.

5. Creating a Fluentd DaemonSet

With Kubernetes running and our ConfigMap created we're ready to craft a Fluentd DaemonSet manifest. Kubernetes manifests are coded as YAML files (https://yaml.org/). Create the following Kubernetes manifest:

```
ubuntu@labsys:~$ nano ~/fluent/fluentd-ds.yaml && cat $_
```

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-ds
  labels:
    k8s-app: fluentd-logging
    version: v1
spec:
  selector:
    matchLabels:
      k8s-app: fluentd-logging
  template:
    metadata:
      labels:
        k8s-app: fluentd-logging
        version: v1
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      - key: node-role.kubernetes.io/control-plane
        effect: NoSchedule
      terminationGracePeriodSeconds: 30
      containers:
      - name: fluentd-ds
        image: fluent/fluentd:v1.14.6-1.1
        resources:
          limits:
            memory: 200Mi
        env:
          - name:  FLUENTD_CONF
            value: "fluentd-kube.conf"
        volumeMounts:
        - name: fluentd-conf
          mountPath: /fluentd/etc
      volumes:
      - name: fluentd-conf
        configMap:
          name: fluentd-config
```

```
ubuntu@labsys:~$
```

While this is not a Kubernetes course, its worth examining the various components of the DaemonSet specification we have defined for our Fluentd Kubernetes pods. We'll examine the manifest block by block.

The first block includes general Kubernetes configuration used with all Kubernetes resource types:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-ds
  labels:
    k8s-app: fluentd-logging
    version: v1
```

The `apiVersion` and `kind` keys together tell Kubernetes what type of resource we are creating. The `metadata` key provides descriptive information for our resource. The `name` is required and `labels` allow us to add arbitrary identifiers to the resource that we can use when filtering large lists of resources in big clusters.

The next section is the DaemonSet specification:

```
spec:
  template:
    metadata:
      labels:
        k8s-app: fluentd-logging
        version: v1
```

This is the beginning of the specification for our DaemonSet. A DaemonSet maintains a template for a pod that the cluster will create on (potentially) every node. The pod template can have labels but not a name, since the template will be used to create many pods. DaemonSet pod names are generated automatically and given the DaemonSet's name as a prefix.

The next block defines tolerations:

```
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      - key: node-role.kubernetes.io/control-plane
        effect: NoSchedule
```

When we configured our Kubernetes cluster we removed the all taints from our one and only control plane node. In a normal cluster the control plane would not run application pods, however we probably do want to support log forwarding on our control plane nodes. We can allow operational work loads, like Fluentd, to run on all nodes in the cluster by "tolerating" any taints that are defined. In the code above we specify that our DaemonSet pods should be allowed to run on nodes with the taints having the `node-role.kubernetes.io/master` and `node-role.kubernetes.io/control-plane` keys with the effect `NoSchedule`. This is the standard taint assigned to control plane nodes with most installation tools.

Next we set a termination grace period for the pods in the DaemonSet:

```
      terminationGracePeriodSeconds: 30
```

Normally Kubernetes will kill a container that does not exit within 10 seconds of the initial SIGTERM. As we have seen, Fluentd can take 10-15 seconds to shutdown after the cancel (^C) signal is sent. This setting extends the window Kubernetes allows for container shutdown.

Next we specify the container that will run in the pod:

```
      containers:
      - name: fluentd-ds
        image: fluent/fluentd:v1.14.6-1.1
        resources:
          limits:
```

```
            memory: 200Mi
```

This block specifies the container image to run, `fluent/fluentd:v1.14.6-1.1`, which will automatically be pulled from DockerHub. It also tells Kubernetes that our container should be limited to 200MB of memory, keeping the logging system from eating up too much of the node's memory. Limits can not be exceeded.

> N.B. To ensure lab stability the image tag has been set to the latest available as of this publication. You may substitute the tag with the latest Fluentd version available, viewable on https://hub.docker.com/r/fluent/fluentd

The next block of YAML sets up the Fluentd configuration file:

```
        env:
          - name:  FLUENTD_CONF
            value: "fluentd-kube.conf"
        volumeMounts:
        - name: fluentd-conf
          mountPath: /fluentd/etc
```

Here we declare an environment variable `FLUENTD_CONF` set to `fluentd-kube.conf`. Then we mount a volume named `fluentd-conf` on the container path `/fluentd/etc`, the default location where Fluentd looks for its configuration file. This environment variable must be declared in full uppercase.

The last block defines the `fluentd-conf` volume mentioned above:

```
      volumes:
      - name: fluentd-conf
        configMap:
          name: fluentd-config
```

In the volume specification above we use the previously created ConfigMap to populate the `fluentd-conf` volume. Since the ConfigMap specifies the contents of a `fluentd-kube.conf` file, the container mount path `/fluentd/etc` will be populated with the ConfigMap's `fluentd-kube.conf` file. If you want to update the Fluentd configuration, you can create a new ConfigMap (with a name like `fluentd-config-v2`) with the updated file and change the name in this section. Doing so will automatically trigger an update to the Fluentd DaemonSet pods.

> N.B. You can also update the existing ConfigMap, but that may affect your ability to roll back to a known-good version of the previous configuration if required. You will need to restart the Fluentd processes in every pod manually by sending an appropriate signal to the processes or deleting the pods (which will be automatically recreated).

Now, create the DaemonSet:

```
ubuntu@labsys:~$ kubectl apply -f ~/fluent/fluentd-ds.yaml

daemonset.apps/fluentd-ds created

ubuntu@labsys:~$
```

List the DaemonSets on your system:

```
ubuntu@labsys:~$ kubectl get ds
```

```
NAME         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
fluentd-ds   1         1         1       1            1           <none>          4s

ubuntu@labsys:~$
```

Since we did not specify a namespace, Kubernetes runs the DaemonSet pods in the configured namespace `default`. In a production setting you might run Fluentd in the `kube-system` namespace or another namespace.

Within a few moments, a Fluentd pod prefixed `fluentd-ds` should be available. List the pods in the default namespace:

```
ubuntu@labsys:~$ kubectl get pods

NAME               READY   STATUS    RESTARTS   AGE
fluentd-ds-sj2xx   1/1     Running   0          8s

ubuntu@labsys:~$
```

To see the activity of the Fluentd pod, use the `kubectl logs` command to view its logs:

```
ubuntu@labsys:~$ kubectl logs -l k8s-app=fluentd-logging

2022-06-14 16:15:38 +0000 [info]: starting fluentd-1.3.2 pid=7 ruby="2.5.2"
2022-06-14 16:15:38 +0000 [info]: spawn command to main:  cmdline=["/usr/bin/ruby", "-Eascii-
8bit:ascii-8bit", "/usr/bin/fluentd", "-c", "/fluentd/etc/fluentd-kube.conf", "-p",
"/fluentd/plugins", "--under-supervisor"]
2022-06-14 16:15:38 +0000 [info]: gem 'fluentd' version '1.3.2'
2022-06-14 16:15:38 +0000 [info]: adding match pattern="**" type="stdout"
2022-06-14 16:15:38 +0000 [info]: adding source type="http"
2022-06-14 16:15:38 +0000 [info]: adding source type="forward"
2022-06-14 16:15:38 +0000 [info]: #0 starting fluentd worker pid=17 ppid=7 worker=0
2022-06-14 16:15:38 +0000 [info]: #0 listening port port=24224 bind="0.0.0.0"
2022-06-14 16:15:38 +0000 [info]: #0 fluentd worker is now running worker=0
2022-06-14 16:15:38.614288045 +0000 fluent.info: {"worker":0,"message":"fluentd worker is now
running worker=0"}

ubuntu@labsys:~$
```

Referring to the pod by the label `k8s-app=fluentd-logging` allows you to see the logs for the pod without knowing any specific pod names. If you have multiple DaemonSet pods, the logs from all pods will be displayed. Based on the output, Fluentd is running and so far everything looks good!

6. Testing the Fluentd DaemonSet

Try to curl the Fluentd HTTP port on localhost:

```
ubuntu@labsys:~$ curl -X POST -d 'json={"Hi":"Mom"}' http://localhost:24220/fluentd

curl: (7) Failed to connect to localhost port 24220: Connection refused

ubuntu@labsys:~$
```

This does not work because applications running within Kubernetes pods are separated from the host's network, instead running on the configured "pod network", in our case Weave. To interact with this Fluentd instance we either need to rerun it with its container ports mapped to host ports or access it from another pod within the cluster's pod network.

To communicate with a pod on the pod network we will need to get it's IP address. Get a listing of pods with `kubectl get pods` and

the `-o wide` option to find the Fluentd pod IP:

```
ubuntu@labsys:~$ kubectl get pod -o wide

NAME               READY   STATUS    RESTARTS   AGE    IP          NODE             NOMINATED NODE
READINESS GATES
fluentd-ds-dzml9   1/1     Running   0          100s   10.32.0.4   labsys           <none>
<none>

ubuntu@labsys:~$
```

Now use `kubectl run` to create a test client pod. Run an interactive pod to test our Fluentd DaemonSet with:

```
ubuntu@labsys:~$ kubectl run test-client --image centos:8 -it

If you don't see a command prompt, try pressing enter.

[root@test-client /]#
```

The command above runs a new pod named `test-client` with a container that uses the `centos:8` container image. The container launches with the `-i` and `-t` switches. The `t` creates a tty in the container and the `i` redirects the current shell's input to the container so that we can issue commands from within it.

From inside the container try to post some data to Fluentd, using the Fluentd pod's IP address retrieved by `kubectl get pods -o wide`:

```
[root@test-client /]# curl -X POST -d 'json={"Hi":"Mom"}' http://10.32.0.4:24220/test

[root@test-client /]#
```

Looks good. Exit the test container:

```
[root@test-client /]# exit

exit
Session ended, resume using 'kubectl attach test-client -c test-client -i -t' command when the pod
is running

ubuntu@labsys:~$
```

As the exit message reports, the pod is still running after we leave the shell and we can reattach if we like.

Now redisplay the log output of the Fluentd pod:

```
ubuntu@labsys:~$ kubectl logs -l k8s-app=fluentd-logging | tail -5

2022-06-14 16:15:38 +0000 [info]: #0 starting fluentd worker pid=17 ppid=7 worker=0
2022-06-14 16:15:38 +0000 [info]: #0 listening port port=24224 bind="0.0.0.0"
2022-06-14 16:15:38 +0000 [info]: #0 fluentd worker is now running worker=0
2022-06-14 16:15:38.614288045 +0000 fluent.info: {"worker":0,"message":"fluentd worker is now
running worker=0"}
2022-06-14 16:16:25.081212507 +0000 test: {"Hi":"Mom"}

ubuntu@labsys:~$
```

Great! Our Fluentd pod is operating perfectly.

To make it easier to communicate with the Fluentd pods (whose IP addresses can change, especially if they are reconfigured), it is ideal to create a corresponding Service for the Fluentd DaemonSet pods. The Service's job is to provide a single, stable network identity which routes traffic to a set of pods it represents.

Create the service definition:

```
ubuntu@labsys:~$ nano ~/fluent/fluentd-ds-svc.yaml && cat $_

apiVersion: v1
kind: Service
metadata:
  labels:
    app: fluentd
  name: fluentd
spec:
  ports:
  - name: 80-24220
    port: 80                    # The service will listen for requests on this port
    protocol: TCP
    targetPort: 24220           # The service sends traffic it receives to these container ports on
the target pods
  selector:
    k8s-app: fluentd-logging    # Establishes which pods the service submits traffic to
  type: ClusterIP

ubuntu@labsys:~$
```

When created, this service will have a virtual IP allocated to it. That virtual IP will listen on port 80 and then submit traffic to the Fluentd container at port 24220 (which is where we have a http endpoint listening for traffic).

Apply the service and check what IP address it gets:

```
ubuntu@labsys:~$ kubectl apply -f ~/fluent/fluentd-ds-svc.yaml

service/fluentd created

ubuntu@labsys:~$ kubectl get service

NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
fluentd      ClusterIP   10.99.73.8   <none>        80/TCP    11s
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   70m

ubuntu@labsys:~$
```

Now use `kubectl exec` to test the cluster virtual IP (a.k.a. Cluster IP) at port 80:

> N.B. Be sure to use the Cluster IP shown by your `kubectl get service` output.

```
ubuntu@labsys:~$ kubectl exec test-client -- curl -X POST -d 'json={"From":"Service"}'
http://10.99.73.8:80/test

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    17    0     0  100    17      0   5666 --:--:-- --:--:-- --:--:-- 17000
```

```
ubuntu@labsys:~$
```

In addition to the virtual IP, the service also receives a DNS name from `coredns` which is formatted as: `servicename.namespace.svc.clusterdomain`.

Try the test again, but this time with the DNS name. In this case, the DNS name is `fluentd.default.svc.cluster.local`:

```
ubuntu@labsys:~$ kubectl exec test-client -- curl -X POST -d 'json={"Via":"DNS"}'
http://fluentd.default.svc.cluster.local:80/test

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    17    0     0  100    17      0    680 --:--:-- --:--:-- --:--:--   708

ubuntu@labsys:~$
```

Finally, confirm that Fluentd has been receiving the requests you have been making:

```
ubuntu@labsys:~$ kubectl logs -l k8s-app=fluentd-logging | tail -5

2022-06-14 16:15:38 +0000 [info]: #0 fluentd worker is now running worker=0
2022-06-14 16:15:38.614288045 +0000 fluent.info: {"worker":0,"message":"fluentd worker is now
running worker=0"}
2022-06-14 16:16:25.081212507 +0000 test: {"Hi":"Mom"}
2022-06-14 16:25:54.975264042 +0000 test: {"From":"Service"}
2022-06-14 16:28:02.070436227 +0000 test: {"Via":"DNS"}

ubuntu@labsys:~$
```

Now your workloads can easily access Fluentd on Kubernetes! Fluentd pod configurations vary widely with the needs of each Kubernetes cluster but our example here demonstrates a number of key features and configuration approaches.

7. Cleanup

To return the cluster to its initial state, delete the `test-client` pod and Fluentd DaemonSet:

```
ubuntu@labsys:~$ kubectl delete pod test-client

pod "test-client" deleted

ubuntu@labsys:~$ kubectl get daemonset

NAME         DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
fluentd-ds   1         1         1       1            1           <none>          7m

ubuntu@labsys:~$  kubectl delete ds fluentd-ds

daemonset.apps "fluentd-ds" deleted

ubuntu@labsys:~$ kubectl get daemonset,pod

No resources found in default namespace.

ubuntu@labsys:~$
```

Optionally, you may uninstall Kubernetes using the following commands:

```
ubuntu@labsys:~$ sudo kubeadm reset

[reset] WARNING: Changes made to this host by 'kubeadm init' or 'kubeadm join' will be reverted.
[reset] Are you sure you want to proceed? [y/N]: y
[preflight] Running pre-flight checks
W0728 16:00:56.719433   47758 removeetcdmember.go:79] [reset] No kubeadm config, using etcd pod spec
to get data directory
[reset] No etcd config found. Assuming external etcd
[reset] Please, manually reset etcd to prevent further issues
[reset] Stopping the kubelet service
[reset] Unmounting mounted directories in "/var/lib/kubelet"
[reset] Deleting contents of config directories: [/etc/kubernetes/manifests /etc/kubernetes/pki]
[reset] Deleting files: [/etc/kubernetes/admin.conf /etc/kubernetes/kubelet.conf
/etc/kubernetes/bootstrap-kubelet.conf /etc/kubernetes/controller-manager.conf
/etc/kubernetes/scheduler.conf]
[reset] Deleting contents of stateful directories: [/var/lib/kubelet /var/lib/dockershim
/var/run/kubernetes /var/lib/cni]

The reset process does not clean CNI configuration. To do so, you must remove /etc/cni/net.d

The reset process does not reset or clean up iptables rules or IPVS tables.
If you wish to reset iptables, you must do so manually by using the "iptables" command.

If your cluster was setup to utilize IPVS, run ipvsadm --clear (or similar)
to reset your system's IPVS tables.

The reset process does not clean your kubeconfig files and you must remove them manually.
Please, check the contents of the $HOME/.kube/config file.

ubuntu@labsys:~$ sudo rm -r /etc/cni/net.d

ubuntu@labsys:~$
```

Congratulations, you have completed the Lab.