RX-M Cloud Native Consulting

Kubernetes

Lab 2 – Kubernetes Exploration

Kubernetes clusters track and manage objects of various “kinds”. Applications make use of four kinds of objects in particular:

- **Pods** – groups of containers deployed as a unit
- **Replica Sets** – sets of pods defined by a template which the Controller Manager replicates across the cluster
- **Deployments** – a rollout strategy for pods and replica sets
- **Services** – end points used to distribute requests to one of a pod’s replicas

Thus basic Kubernetes applications consist of pods, which implement the application functionality; replica sets, which ensure pods are always available; and Services which expose a dynamic set of pods to clients as a single endpoint. Deployments describe how to launch or upgrade a given application (set of pods).

The principal tool used to manage kubernetes clusters is kubectl. Using kubectl you can create deployments and services, monitor cluster components and pods, upgrade deployments and remove resources that are no longer required.

In this lab we will get familiar with kubectl and learn how to use it to manage many aspects of Kubernetes.

1. kubectl

The `kubectl` command provides a range of features we can use with Kubernetes. Run `kubectl` without arguments to get a list of the available commands.

```
user@ubuntu:~$ kubectl

kubectl controls the Kubernetes cluster manager.

 Find more information at:
https://kubernetes.io/docs/reference/kubectl/overview/

Basic Commands (Beginner):
  create         Create a resource from a file or from stdin.
  expose         Take a replication controller, service, deployment or pod and
expose it as a new Kubernetes Service
  run            Run a particular image on the cluster
  set            Set specific features on objects

Basic Commands (Intermediate):
  explain        Documentation of resources
  get            Display one or many resources
  edit           Edit a resource on the server
  delete         Delete resources by filenames, stdin, resources and names, or
by resources and label selector

Deploy Commands:
  rollout        Manage the rollout of a resource
  scale          Set a new size for a Deployment, ReplicaSet, Replication
Controller, or Job
  autoscale      Auto-scale a Deployment, ReplicaSet, or ReplicationController

Cluster Management Commands:
  certificate    Modify certificate resources.
  cluster-info   Display cluster info
  top            Display Resource (CPU/Memory/Storage) usage.
  cordon         Mark node as unschedulable
  uncordon       Mark node as schedulable
  drain          Drain node in preparation for maintenance
  taint          Update the taints on one or more nodes

Troubleshooting and Debugging Commands:
  describe      Show details of a specific resource or group of resources
  logs          Print the logs for a container in a pod
  attach        Attach to a running container
  exec          Execute a command in a container
  port-forward  Forward one or more local ports to a pod
  proxy         Run a proxy to the Kubernetes API server
  cp            Copy files and directories to and from containers.
  auth          Inspect authorization

Advanced Commands:
  diff          Diff live version against would-be applied version
  apply         Apply a configuration to a resource by filename or stdin
  patch         Update field(s) of a resource using strategic merge patch
  replace       Replace a resource by filename or stdin
  wait          Experimental: Wait for a specific condition on one or many
resources.
  convert       Convert config files between different API versions
  kustomize     Build a kustomization target from a directory or a remote url.

Settings Commands:
  label         Update the labels on a resource
  annotate      Update the annotations on a resource
  completion    Output shell completion code for the specified shell (bash or
zsh)

Other Commands:
  api-resources Print the supported API resources on the server
  api-versions  Print the supported API versions on the server, in the form of
"group/version"
  config        Modify kubeconfig files
  plugin        Provides utilities for interacting with plugins.
  version       Print the client and server version information

Usage:
  kubectl [flags] [options]

Use "kubectl <command> --help" for more information about a given command.
Use "kubectl options" for a list of global command-line options (applies to all
commands).

user@ubuntu:~$
```

Take a moment to review available options. One useful subcommand is the global options, take a moment to review the output of `kubectl options`.

To use the `kubectl` command to control a remote cluster we must specify the cluster endpoint to `kubectl`. The `kubectl` command can be used to control several clusters from a single workstation. Clusters are given a name and settings, including the IP address and port of the cluster API service.

To get configuration help issue the `kubectl help` subcommand.

```
user@ubuntu:~$ kubectl help config

Modify kubeconfig files using subcommands like "kubectl config set current-context my-context"

 The loading order follows these rules:

  1.  If the --kubeconfig flag is set, then only that file is loaded. The flag may only be set
once and no merging takes
place.
  2.  If $KUBECONFIG environment variable is set, then it is used as a list of paths (normal
path delimiting rules for
your system). These paths are merged. When a value is modified, it is modified in the file that
defines the stanza. When
a value is created, it is created in the first file that exists. If no files in the chain exist,
then it creates the
last file in the list.
  3.  Otherwise, ${HOME}/.kube/config is used and no merging takes place.
```

```
Available Commands:
  current-context Displays the current-context
  delete-cluster  Delete the specified cluster from the kubeconfig
  delete-context  Delete the specified context from the kubeconfig
  get-clusters    Display clusters defined in the kubeconfig
  get-contexts    Describe one or many contexts
  rename-context  Renames a context from the kubeconfig file.
  set             Sets an individual value in a kubeconfig file
  set-cluster     Sets a cluster entry in kubeconfig
  set-context     Sets a context entry in kubeconfig
  set-credentials Sets a user entry in kubeconfig
  unset           Unsets an individual value in a kubeconfig file
  use-context     Sets the current-context in a kubeconfig file
  view            Display merged kubeconfig settings or a specified kubeconfig file

Usage:
  kubectl config SUBCOMMAND [options]

Use "kubectl <command> --help" for more information about a given command.
Use "kubectl options" for a list of global command-line options (applies to all commands).

user@ubuntu:~$
```

Run the `kubectl config view` subcommand again to display the current client configuration.

```
user@ubuntu:~$ kubectl config view

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://192.168.228.157:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED

user@ubuntu:~$
```

When you run *kubectl* commands a context is required. The context tells *kubectl* which cluster to connect to and which user to authenticate as. As you can see the values kubeadm configured means the `kubectl` command tries to reach the API server on port 6443 via our host's IP with TLS.

To view the REDACTED elements, add `--flatten`.

We can configure `kubectl` explicitly so that we can adjust our cluster settings in the future if need be. Get help on the `config set-cluster` subcommand:

```
user@ubuntu:~$ kubectl help config set-cluster

Sets a cluster entry in kubeconfig.

 Specifying a name that already exists will merge new fields on top of existing values for those
fields.

Examples:
  # Set only the server field on the e2e cluster entry without touching other values.
  kubectl config set-cluster e2e --server=https://1.2.3.4
  # Embed certificate authority data for the e2e cluster entry
  kubectl config set-cluster e2e --certificate-authority=~/.kube/e2e/kubernetes.ca.crt

  # Disable cert checking for the dev cluster entry
  kubectl config set-cluster e2e --insecure-skip-tls-verify=true

Options:
      --embed-certs=false: embed-certs for the cluster entry in kubeconfig

Usage:
  kubectl config set-cluster NAME [--server=server] [--certificate-
authority=path/to/certificate/authority]
[--insecure-skip-tls-verify=true] [options]

Use "kubectl options" for a list of global command-line options (applies to all commands).

user@ubuntu:~$
```

`kubectl` configuration data is saved in a YAML file in your `$HOME/.kube` directory using thes commands. Display the configuration file we copied after the kubeadm install:

```
user@ubuntu:~$ ls -la ~/.kube/

total 24
drwxrwxr-x  4 user user 4096 Jan  8 12:26 .
drwxr-xr-x 15 user user 4096 Jan  8 12:25 ..
drwxr-x---  3 user user 4096 Jan  8 12:26 cache
-rw-------  1 user root 5451 Jan  8 12:25 config
drwxr-x---  3 user user 4096 Jan  8 12:31 http-cache

user@ubuntu:~$
```

Display the contents of the config file:

```
user@ubuntu:~$ cat ~/.kube/config

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data:
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUN5RENDQWJDZ0F3SUJBZ0lCQURBTkJna3Foa2lHOXcwQkFRc0ZBREFWTV
JNd0VRWURWUVFERXdwcmRXSmwKY201bGRHVnpNQjRYRFRFNU1EZ3dOakl3TWpNd04xb1hEVEk1TURnd016SXdNak13TjFvd0ZU
RVRNQkVHQTFVRQpBeE1LYTNWaVpYSnVaWFJsY3pDQ0FTSXdEUVlKS29aSWh2Y05BUUVCQlFBRGdnRVBBRENDQVFvQ2dnRUJBT3
dUCnNxbGJmS2VPc3Y4ZzByZG96aytKcm5TQ3YyQUxiMGdKbTFVTVRMbzRsNFFmZ3A0L09wbk9PaTZhbThWRlNNYTIKQTdEOFJV
RWN2Sm44SkNUVW1jMTBxWmhRdy9acU5KQ1AxZWdpTmdndGtFcXpjd2NwVGlkeUVKaG5CZ3ljRW1WSgoxVisrTlZ5WnliMmJ6L1
N2MGE2QjVib3hlSEhBdHZzclNEV2phTFBYbXZrRStuTWxXc2dwajkrZlZoSk5jNnZ4Cko2OGVwTm4rZlk2MnZjSHhkMDRtdTVu
alFNOWI0TGYyanEvWWV0V2xnTFNQZkN4THozdDVVb3g4NlVOYkRyZG0KaVMzczZ1R2lFeDVhYTZJNmIzS09weTRTSi95ak44VF
VVVnVnOFdwbTcvUXNPRFphcXhZT1R4NEdrNWJpT1pSVApzQ1JLU056bFpSbUI4R0lFWGlrQ0F3RUFBYU1qTUNFd0RnWURWUjBQ
QVFIL0JBUURBZ0trTUE4R0ExVWRFd0VCCi93UUZNQU1CQWY4d0RRWUpLb1pJaHZjTkFRRUxCUUFFZ2dFQkFIbUNSNm56VENhR1
NpSGtvSng3QWlsWnpQTW0KUHVTWU9wNzRFa1YzMFN0YU5EdzNYZmk3STNWQmF6ODFlSlZqSCtzQUsweWJIalpOYzVVYUmFzOTQ5
eTFsbSthRQpjUjZDQmVSNnBKUXJxRS90c094cVFEcG9QNlRHTlRyQ2NpSlJ1YzY1aFYzYytBMktWNmE5R2RuaG95RCtPd0pKCn
Z1cFRVSDFONktkT3pmRjhpN2R1UjBsUVhPbjlVcEtQMGZOOXVEQjB1SnJPSVNxMUZiaTNudFZnNUNjUFdJZUkKL2lZNXExbzJS
bEJTbTgyZEdybTc5Z2lMcEoydUsveUlUVjBSbGl0cFh3YW9weG4rYUdwV3E0NURQZlV0QkVhKwo4djdEenpTYXFaVlZhS0VEaG
tFLzVEMGNlK0Yyam9US1VtRmZXNGQwa0o1UWdkNTBnL3B6b2hwK1Z6az0KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
    server: https://192.168.228.157:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data:
```

```
LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUM4akNDQWRxZ0F3SUJBZ0lJWjNQenAweEZVaDR3RFFZSktvWklodmNOQV
FFTEJRQXdGVEVUTUJFR0ExVUUKQXhNS2EzVmlaWEp1WlhSbGN6QWVGdzB4T1RBNE1EWXlNREl6TURkYUZ3MHlNREE0TURVeU1E
SXpNRGxhTURReApGekFWQmdOVkJBb1REbk41YzNSbGJUcHRZWE4wWlhKek1Sa3dGd1lEVlFRREV4QnJkV0psY201bGRHVnpMV0
ZrCmJXbHVNSUlCSWpBTkJna3Foa2lHOXcwQkFRRUZBQU9DQVE4QU1JSUJDZ0tDQVFFQXJ3anFoQVBJdkFVTWhtNk8KdWFMWEZz
b0ZPTWJhbGhjTkZmZy9nb3FwTE93UWFGRUVjUVlUeXdhSUZvbERHekZ5eko4VFJQYjk5cld0K1IvTgoxeEVHODkwdnltc1Q5Tm
80U042REI4OTZtRWQ3djdBbTgxaWY5NWlQVG5SbkpFcWxwMUk5OXdSRytaR205VW1QClp3ZVRTL05tb3FTRFdvYjAzUnFrVXg2
bDdoWW91c1pzSkRIcVJHUWhrcWxtb0xlK1NVUFdGb0FwYjF0SDg0dS8KZjRxL25RQzFqS0U3bEJ6UzZRSXhIMkNHbUo1MXRydV
dldjZVYWx6TW1iUy8wVi9KT0xLSW90SVQrZG9oZHpqZQo5a0t5dDNJU3Yxakg2Ukh5cWo0REd6TmFQby91Z3VTL2RNT3BnNi9r
ZHdEOXpXeGJSb2RPUUxFZEVGZFRWRkxiCmY5ZWVLd0lEQVFBQm95Y3dKVEFPQmdOVkhROEJBZjhFQkFNQ0JhQXdFd1lEVlIwbE
JBd3dDZ1lJS3dZQkJRVUgKQXdJd0RRWUpLb1pJaHZjTkFRRUxCUUFEZ2dFQkFDVGxLVWI4SXdXSFl4T0RrVGtvWWluN1ZtNEdB
T3pHdk84RwpaTC9UVjU2YnNOcGtMUWRwMG44RGVrODJrRHBwNm9wN2dFejc2TVJHMjh4Q1lTODkxeFgyVS9PY0kwVThDVFRpCm
pSams0OVlYTWVMUzVSTHp4dW1Ub3RHMlVQbVhydHRvanhBR3ZoaHg5R3RTR1ZIclBKbkk3ZEZ2T1h4elhhOTUKNWZxV3YvY21T
UXZ0cUFJL2d0L1c0bmg4QU5iKzEvYjIrb3lQQ1pGcFVqSGhPb2lyV05LNjErL1JXVjU5K3pDaQpRZ1VaeGNLU1Q0SzJFNGxkaG
5SdzJNaFFsV3dJckR4dGNKbGtseDZMTUhZZ0w2WnFLUkVsUlIvaWJNb25WU0pzCk1JYk5tNjB4cTlpaWs5SUVBWnYxUERXcFB5
b0M5MVRTSys3OUY2TEF4T1JNejdLSjZTWT0KLS0tLS1FTkQgQ0VSVElGSUNBVEUtLS0tLQo=
    client-key-data:
LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFb2dJQkFBS0NBUUVBcndqcWhBUEl2QVVNaG02T3VhTFhGc29GT0
1iYWxoY05GZmcvZ29xcExPd1FhRkVFCmNRWVR5d2FJRm9sREd6Rnl6SjhUUlBiOTlyV3RrUi9OMXhFRzg5MHZ5bXNUOU5vNFNO
NkRCODk2bUVkN3Y3QW0KODFpZjk1aVBUblJuSkVxbHAxSTk5d1JHK1pHbTlVbVBad2VUUy9ObW9xU0RXb2IwM1Jxa1V4Nmw3aF
lvdXNacwpKREhxUkdRaGtxbG1vTGUrU1VQV0ZvQXBiMXRIODR1L2Y0cS9uUUMxaktFN2xCelM2UUl4SDJDR21KNTF0cnVXCmV2
NlVhbHpNbWJTLzBWL0pPTEtJb3RJVCtkb2hkemplOWtLeXQzSVN2MWpINlJIeXFqNERHek5hUG8vdWd1Uy8KZE1PcGc2L2tkd0
Q5eld4YlJvZE9RTEVkRUZkVFZGTGJmOWVlS3dJREFRQUJBb0lCQUFuNFV3dGovOG9kNjRHNAp5RzV3aFJucU5mUHU5OEoyMjZy
MXNnQm9qNHhjZ2U2L05xYU5keEVvZFJYN2laUUI5QTBOdnBZK0QyYy9JQUxCCnVnTFhHSE5KbFJCTm14eG1WYkJvNUVQTW9Lb0
40RkdoNGdZWWRUd3dOM0Y3bGVodVRCTkV6dnpFQkpyd25mYjQKMHE5R3NqQXdWQmROa2FIVmNCc3NDbElsdThQYzdTSTBuK0ty
ZzJ6akpYcU4rMHQxYVdYMWdXb2RMclJuYWkzTQpFbXF2SUNpazBHNnB0Zmh6K2R6dC9ReWJmVzlDWXRrdk9lU3dPWkpvM1NscT
lzMGlQYkZBMGo3dlB1Ti8zNW80CmZTUkRwQkh2VmxGTElObG9qMXkxTWYwTElpOU52NGJzNWF0Wmh0OFZxdElSanYyZm5BSWxK
MnhPb2E5dHJGMDgKeEROcnBhRUNnWUVBMnJqNWNCaVovQjRaYjlzeERXblNFRGt0RHNYYTA2cEpnRzRQOXFxTlYvbGk1Vmg5MW
NkcApQa1l6ZEFNcXRHUGRMbG5vWEVYaE9sd3pwOU8ydzhoUkhsditnZ0s5K1QzZzNWWjdXYkpxZTlWbTBFaDNVaFRwCm5pZnhX
bnlHZkRaR1JpMG52WHRjVEp6MVU5Q0Z3bVA4MVlBUFU4T1BLTkxZbCtuSUd6cmw2anNDZ1lFQXpOM1EKYUQyUEpWTzRMLzBMZF
NuMHh0ek1zRytycUJkYXRMZDFtUXpYNTFpcHZaWWdNeHdNcENNRV0lSKzlRa2RybmpZUwozS3h1ZGc5ZGYybmlRNSttbENFS01a
SlVUaDNJRnNxclVwcG5oTDlpWjRXR2xkRlJXdS9aSUZjQTIvSHZvSGlMCjd4azExYkZvTmJ3T3EzU0JDUHNnQmVqZFBXaUpnMU
5zSXFwUDdORUNnWUJXcWxTc0JoTjl3cTl4a3Zpc3gvRmsKWEVOdVJ4ZWVKeXRZcUVQTENXbFlJb2o5TjBNbElnNmhqazkwK0l4
RFRnNmwveG9DOWUxNG9uQVZYOTViVlZSUApJMFNGWDNESEFhM0lCaXg2TGlmalNYdWpyQk1iZ3czT2pTVWxKWkprUlYwekVWRm
Qwek9QWitJTmcrSjhWQUlxCjFzTTcvWkVNSy8rR3NpdUlIcDViWVFLQmdHSisvdFV2UzZSaHBQdUZpTDJ6QjFtWkROeURSQitU
NHV0aURTc2gKanFoTzY0VVlLYkVJK2xic2RxdEVURVVTZTM1Y2R5TWIwQnY0OVRYdUhYZnZ5VElNMUk3UzBLK2lCL0pWVEp0eA
pXdlhxNGcvdGxiQndLOUl4NE0xNHB4UVlOT0tOTXBJcEo4WHUvckJmRXhxQjhBdjJXUVllV0VoTysxWmxoR2NDClJWMFJBb0dB
YXlsTmpCL1VENzRoUzFVbCtzQ25wMUlaRGd1UkFpR0JyUVAyZ0plSUthREFUN3dkOG5uamdzMXIKc3hHWlFQZlFOY2tRTXFVeT
R5M0c2Y1ZaNUp0OUtQRmNPS01IcDZFalI2SENKbVBFV1dMLzNEek1hbkZrSHE1OAo0L08zV3ptT2dlVXI5MlJyc0MrMGFpMGw0
N1hSNnB1ZndEOEczNytMY3U5UnJxelpSSTA9Ci0tLS0tRU5EIFJTQSBQUklWQVRFIEtFWS0tLS0tCg==

user@ubuntu:~$
```

The `kubectl config view` command will display nearly the same data, obfuscating the key data. The config file is simple and can easily be pre-generated and distributed to any client systems that require connection to a given cluster.

2. Test the Cluster

Now with our cluster running and `kubectl` configured lets issue some commands to test the Kubernetes cluster. The `cluster-info` subcommand can be used to test the cluster API end point and the `get nodes` command can be used to see the nodes in the cluster.

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

Kubernetes master is running at https://192.168.228.157:6443
KubeDNS is running at https://192.168.228.157:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

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

user@ubuntu:~$
```

If you are really adventurous run the suggested command for a detailed cluster overview, careful though, its a lot of information!

```
user@ubuntu:~$ kubectl cluster-info dump |& wc -l

4972
```

```
user@ubuntu:~$
```

To get detailed node information use the `describe node` subcommand again on the desired node name:

```
user@ubuntu:~$ kubectl describe node ubuntu

Name:               ubuntu
Roles:              master
Labels:             beta.kubernetes.io/arch=amd64
                    beta.kubernetes.io/os=linux
                    kubernetes.io/arch=amd64
                    kubernetes.io/hostname=ubuntu
                    kubernetes.io/os=linux
                    node-role.kubernetes.io/master=
Annotations:        kubeadm.alpha.kubernetes.io/cri-socket: /var/run/dockershim.sock
                    node.alpha.kubernetes.io/ttl: 0
                    volumes.kubernetes.io/controller-managed-attach-detach: true
CreationTimestamp:  Wed, 08 Jan 2020 12:15:29 -0800
Taints:             <none>
Unschedulable:      false
Conditions:
  Type                 Status  LastHeartbeatTime                 LastTransitionTime
Reason                       Message
  ----                 ------  -----------------                 ------------------
------                       -------
  NetworkUnavailable   False   Wed, 08 Jan 2020 12:31:56 -0800   Wed, 08 Jan 2020 12:31:56 -0800
WeaveIsUp                    Weave pod has set this
  MemoryPressure       False   Wed, 08 Jan 2020 12:40:33 -0800   Wed, 08 Jan 2020 12:15:27 -0800
KubeletHasSufficientMemory   kubelet has sufficient memory available
  DiskPressure         False   Wed, 08 Jan 2020 12:40:33 -0800   Wed, 08 Jan 2020 12:15:27 -0800
KubeletHasNoDiskPressure     kubelet has no disk pressure
  PIDPressure          False   Wed, 08 Jan 2020 12:40:33 -0800   Wed, 08 Jan 2020 12:15:27 -0800
KubeletHasSufficientPID      kubelet has sufficient PID available
  Ready                True    Wed, 08 Jan 2020 12:40:33 -0800   Wed, 08 Jan 2020 12:32:02 -0800
KubeletReady                 kubelet is posting ready status. AppArmor enabled
Addresses:
  InternalIP:  192.168.228.157
  Hostname:    ubuntu
Capacity:
 cpu:                2
 ephemeral-storage:  18447100Ki
 hugepages-1Gi:      0
 hugepages-2Mi:      0
 memory:             2030628Ki
 pods:               110
Allocatable:
 cpu:                2
 ephemeral-storage:  17000847332
 hugepages-1Gi:      0
 hugepages-2Mi:      0
 memory:             1928228Ki
 pods:               110
System Info:
 Machine ID:                 6e883acc04fc7db3713776be57a3dac9
 System UUID:                5FBB4D56-33A0-3A9A-19B1-95D19AECC42F
 Boot ID:                    7c1dbf59-0da5-4010-8ea0-b7253b5446e4
 Kernel Version:             4.4.0-31-generic
 OS Image:                   Ubuntu 16.04.1 LTS
 Operating System:           linux
 Architecture:               amd64
 Container Runtime Version:  docker://19.3.5
 Kubelet Version:            v1.16.4
 Kube-Proxy Version:         v1.16.4
Non-terminated Pods:         (8 in total)
  Namespace                  Name                              CPU Requests  CPU Limits  Memory
Requests  Memory Limits  AGE
  ---------                  ----                              ------------  ----------  -------
--------  -------------  ---
  kube-system                coredns-5644d7b6d9-b4rnz          100m (5%)     0 (0%)      70Mi
(3%)        170Mi (9%)     25m
```

```
  kube-system                coredns-5644d7b6d9-lxdqv          100m (5%)     0 (0%)      70Mi
(3%)        170Mi (9%)     25m
  kube-system                etcd-ubuntu                       0 (0%)        0 (0%)      0 (0%)
0 (0%)         24m
  kube-system                kube-apiserver-ubuntu             250m (12%)    0 (0%)      0 (0%)
0 (0%)         24m
  kube-system                kube-controller-manager-ubuntu    200m (10%)    0 (0%)      0 (0%)
0 (0%)         24m
  kube-system                kube-proxy-npxks                  0 (0%)        0 (0%)      0 (0%)
0 (0%)         25m
  kube-system                kube-scheduler-ubuntu             100m (5%)     0 (0%)      0 (0%)
0 (0%)         24m
  kube-system                weave-net-rvhvk                   20m (1%)      0 (0%)      0 (0%)
0 (0%)         9m42s
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource           Requests    Limits
  --------           --------    ------
  cpu                770m (38%)  0 (0%)
  memory             140Mi (7%)  340Mi (18%)
  ephemeral-storage  0 (0%)      0 (0%)
Events:
  Type    Reason                   Age                From                Message
  ----    ------                   ----               ----                -------
  Normal  NodeHasSufficientMemory  26m (x8 over 26m)  kubelet, ubuntu     Node ubuntu status is
now: NodeHasSufficientMemory
  Normal  NodeHasNoDiskPressure    26m (x8 over 26m)  kubelet, ubuntu     Node ubuntu status is
now: NodeHasNoDiskPressure
  Normal  NodeHasSufficientPID     26m (x7 over 26m)  kubelet, ubuntu     Node ubuntu status is
now: NodeHasSufficientPID
  Normal  Starting                 25m                kube-proxy, ubuntu  Starting kube-proxy.
  Normal  NodeReady                9m27s              kubelet, ubuntu     Node ubuntu status is
now: NodeReady

user@ubuntu:~$
```

Describe provides a wealth of node information. Your report will be similar but different than the one above.

- How much memory does your node have?
- How many CPUs?
- How many pods can your node run?
- What container runtime is the `kubelet` using?
- What version of `kubelet` is your node running?

Previously we used the `version` subcommand to discover the version of the `kubectl` client but now that our config is in place we can also see the version of the cluster API Server.

```
user@ubuntu:~$ kubectl version

Client Version: version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.4",
GitCommit:"224be7bdce5a9dd0c2fd0d46b83865648e2fe0ba", GitTreeState:"clean", BuildDate:"2019-12-
11T12:47:40Z", GoVersion:"go1.12.12", Compiler:"gc", Platform:"linux/amd64"}
Server Version: version.Info{Major:"1", Minor:"16", GitVersion:"v1.16.4",
GitCommit:"224be7bdce5a9dd0c2fd0d46b83865648e2fe0ba", GitTreeState:"clean", BuildDate:"2019-12-
11T12:37:43Z", GoVersion:"go1.12.12", Compiler:"gc", Platform:"linux/amd64"}

user@ubuntu:~$
```

If you are familiar with Golang, notice the use of the `gc` tool chain (vs gccgo).

3. Creating Applications

With our cluster running and `kubectl` configured we can try to start a simple application on the cluster. The `kubectl` command provides a `get` subcommand which can be used to get information on any one of the key Kubernetes component types: deployments, pods, replica sets, and Services. While you can type `kubectl get replicasets`, that would be fairly inhumane so `kubectl` allows you to use the abbreviation `rs` for replica sets.

If you want to save yourself even more typing you can take advantage of kubectl's tab completion functionality. Try typing kubectl and then :

> That is, press the tab key twice...

```
user@ubuntu:~$ kubectl get

Desktop/                   .kube/                       Public/
...

user@ubuntu:~$ kubectl get
```

This output is the standard bash shell completion, which just lists the files in the working directory. Not very helpful. You can enable temporary kubectl bash completion with the following command, run it:

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

user@ubuntu:~$
```

Now try kubectl again:

```
user@ubuntu:~$ kubectl get

apiservices.apiregistration.k8s.io               daemonsets.apps
leases.coordination.k8s.io
...

user@ubuntu:~$ kubectl get
```

That is much better! You can now type "kubectl get ser" and it will autocomplete to "kubectl get service".

In a new shell, list the currently running services, deployments, replica sets, and pods on your cluster:

```
user@ubuntu:~$ kubectl get service,deployments,replicasets,pods

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   28m

user@ubuntu:~$
```

The only service running in our cluster is the *kubernetes* service itself. We have no deployments, replica sets, or pods yet (in our namespace). Do the same for the resources under the kube-system namespace, more on namespaces later.

```
user@ubuntu:~$ kubectl get service,deployments,replicaset,pods --namespace=kube-system

NAME               TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
service/kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   28m

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/coredns   2/2     2            2           28m

NAME                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/coredns-5644d7b6d9   2         2         2       28m

NAME                                 READY   STATUS    RESTARTS   AGE
pod/coredns-5644d7b6d9-b4rnz         1/1     Running   0          28m
pod/coredns-5644d7b6d9-lxdqv         1/1     Running   0          28m
pod/etcd-ubuntu                      1/1     Running   0          27m
pod/kube-apiserver-ubuntu            1/1     Running   0          27m
pod/kube-controller-manager-ubuntu   1/1     Running   0          27m
pod/kube-proxy-npxks                 1/1     Running   0          28m
```

```
pod/kube-scheduler-ubuntu            1/1     Running   0          27m
pod/weave-net-rvhvk                  2/2     Running   0          12m

user@ubuntu:~$
```

We can view all namespaces via `--all-namespaces` (if we have permission).

To test our cluster lets run a single container pod. When configured with the Docker Engine as the container manager, we can run any container image that Docker has preinstalled or knows how to download.

```
user@ubuntu:~$ kubectl run my-nginx --generator=run-pod/v1 --image=nginx:1.11 --port=80

pod/my-nginx created

user@ubuntu:~$
```

The pod name is "my-nginx" and the image we used is "nginx", an official image pulled from Docker Hub by the Docker Engine in the background. The port switch tells Kubernetes the service port for our pod which will allow us to share the service with its users over that port (the program must actually use that port for this to work).

List the pods running on the cluster:

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

NAME       READY   STATUS              RESTARTS   AGE
my-nginx   0/1     ContainerCreating   0          9s

user@ubuntu:~$ kubectl get pods

NAME       READY   STATUS    RESTARTS   AGE
my-nginx   1/1     Running   0          23s

user@ubuntu:~$
```

This shows that our pods are deployed and up to date. It may take a bit to pull the Docker images (Ready might be 0).

You can use the `docker container ls` subcommand to display the containers running under the Docker Engine:

```
user@ubuntu:~$ docker container ls --filter "name=nginx"

CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS
PORTS               NAMES
2e25387dfb91        nginx                  "nginx -g 'daemon of…"   28 seconds ago      Up 27
seconds                               k8s_my-nginx_my-nginx_default_2fbe6ccd-f305-4bb0-9d56-
64e5407ee30b_0
1097d5dc9ecf        k8s.gcr.io/pause:3.1   "/pause"                 41 seconds ago      Up 40
seconds                               k8s_POD_my-nginx_default_2fbe6ccd-f305-4bb0-9d56-
64e5407ee30b_0

user@ubuntu:~$
```

As you can see, while our *run* subcommand requested that Kubernetes run a container but 2 containers were launched at that time.

In Kubernetes, each Pod instance has an infrastructure container, which is the first container that the `kubelet` instantiates. The infrastructure container uses the image "k8s.gcr.io/pause:3.1" and acquires the pod's IP as well as a pod wide network and IPC namespace. All of the other containers in the pod then join the infrastructure container's network (--net) and IPC (--ipc) namespace allowing containers in the pod to easily communicate. The initial process ("/pause") that runs in the infrastructure container does nothing, its sole purpose is to act as the anchor for the pod and its shared namespaces.

You can learn more about the pause container by looking at the source and ultimately what is "pause()".

- https://github.com/kubernetes/kubernetes/tree/master/build/pause
- https://github.com/kubernetes/kubernetes/blob/master/build/pause/pause.c

- `man 2 pause` or http://man7.org/linux/man-pages/man2/pause.2.html

The Docker listing shows us 2 containers, the pod having an infrastructure container (pause) and the container we asked for (nginx).

Kubernetes gives each pod a name and reports on the pod status, the number of times the pod has been restarted and the pod's uptime. You can find the pod names embedded in the container names displayed by the `docker container ls` command:

```
user@ubuntu:~$ docker container ls --filter "name=nginx" --format "{{.Names}}"

k8s_my-nginx_my-nginx_default_2fbe6ccd-f305-4bb0-9d56-64e5407ee30b_0
k8s_POD_my-nginx_default_2fbe6ccd-f305-4bb0-9d56-64e5407ee30b_0

user@ubuntu:~$
```

Try killing the nginx container using the `docker container kill` subcommand and the ID of the underlying container based on the nginx image.

```
user@ubuntu:~$ docker container kill \
$(docker container ls --filter "ancestor=nginx:1.11" --format {{.ID}} | head -1)

2e25387dfb91

user@ubuntu:~$
```

```
user@ubuntu:~$ docker container ls --filter "name=nginx"

CONTAINER ID        IMAGE                  COMMAND                  CREATED              STATUS
PORTS               NAMES
a21bcfc9c4d7        5766334bdaa0           "nginx -g 'daemon of…"   10 seconds ago      Up 9
seconds                            k8s_my-nginx_my-nginx_default_2fbe6ccd-f305-4bb0-9d56-
64e5407ee30b_1
1097d5dc9ecf        k8s.gcr.io/pause:3.1   "/pause"                 2 minutes ago       Up 2
minutes                            k8s_POD_my-nginx_default_2fbe6ccd-f305-4bb0-9d56-
64e5407ee30b_0

user@ubuntu:~$
```

We can tell by the created time we have a new container. If you were fast enough, you may have seen the previous container exited. Docker terminates the container specified but Kubernetes has no knowledge of this action. When the Kubelet process, responsible for the pods assigned to this node, sees the missing container, it simply reruns the nginx image.

After some time, if you run the previous command with the `-a` flag, we can see the previous killed container and the newly created one.

```
user@ubuntu:~$ docker container ls -a --filter "name=nginx"

CONTAINER ID        IMAGE                  COMMAND                  CREATED              STATUS
PORTS               NAMES
a21bcfc9c4d7        5766334bdaa0           "nginx -g 'daemon of…"   52 seconds ago      Up 51
seconds                                     k8s_my-nginx_my-nginx_default_2fbe6ccd-f305-4bb0-
9d56-64e5407ee30b_1
2e25387dfb91        nginx                  "nginx -g 'daemon of…"   2 minutes ago       Exited
(137) 52 seconds ago                       k8s_my-nginx_my-nginx_default_2fbe6ccd-f305-4bb0-
9d56-64e5407ee30b_0
1097d5dc9ecf        k8s.gcr.io/pause:3.1   "/pause"                 2 minutes ago       Up 2
minutes                                      k8s_POD_my-nginx_default_2fbe6ccd-f305-4bb0-9d56-
64e5407ee30b_0

user@ubuntu:~$
```

Notice that we killed container *ae6e77e2b8c2* in the example but the new container *e6ecace9aba4* was created to replace it. Kubernetes does not "resurrect" containers that have failed. This is important because the container's state may be the reason it failed. Rather, Kubernetes runs a fresh copy of the original image, ensuring the container has a clean new internal state (cattle not pets!).

Having killed the container on the Docker level, check to see how Kubernetes handled the event:

```
user@ubuntu:~$ kubectl get pod

NAME       READY   STATUS    RESTARTS   AGE
my-nginx   1/1     Running   1          3m16s

user@ubuntu:~$
```

When Kubernetes saw that the pod's container had become unavailable, it restarted the pod (and not the container!), incrementing the amount of restarts. The pod itself is still the same, as seen by its AGE not rotating to a smaller number, but it restarted as it launched a new container.

4. Create a Service

In modern software engineering terms, a **service** is an encapsulated set of functionality made available to consumers through an API. The problem with our nginx application at present is that when containers die new ones are created. The fact that there are multiple containers and that containers come and go makes using the app difficult.

To simplify things Kubernetes makes it possible for us to expose our pods as a Service. The `kubectl` expose command does this.

Expose the my-nginx pod pod as a service:

```
user@ubuntu:~$ kubectl expose $(kubectl get pod -o=name) --port=80

service/my-nginx exposed

user@ubuntu:~$
```

This causes Kubernetes to create a conceptual Service for our pods, exposing the set of pods as a single endpoint for users. Use the *get services* subcommand to display your service.

```
user@ubuntu:~$ kubectl get services

NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   33m
my-nginx     ClusterIP   10.106.240.235   <none>        80/TCP    10s

user@ubuntu:~$
```

Kubernetes has given our service a virtual IP (VIP) address and it will now distribute client connections across any running my-nginx pods.

To test the Service try curling it:

```
user@ubuntu:~$ NX_CIP=$(kubectl get services -o=custom-
columns=NAME:.spec.clusterIP,NAME:.metadata.name \
| grep nginx | awk '{print $1}') && echo $NX_CIP

10.106.240.235

user@ubuntu:~$ curl -I $NX_CIP

HTTP/1.1 200 OK
Server: nginx/1.11.13
Date: Wed, 08 Jan 2020 20:49:02 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Tue, 04 Apr 2017 15:01:57 GMT
Connection: keep-alive
ETag: "58e3b565-264"
Accept-Ranges: bytes
```

```
user@ubuntu:~$
```

Success!

5. Pod exec

While Kubernetes delegates all of the direct container operations to the container manager (usually Docker) it does pass through some useful container features.

For example, imagine you need to discover the distro of one of your pods' containers. You can use the `kubectl exec` subcommand to run arbitrary commands within a pod.

Try listing the running pods and then executing the `cat /etc/os-release` command within one of your pods.

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

NAME       READY   STATUS    RESTARTS   AGE
my-nginx   1/1     Running   1          4m37s

user@ubuntu:~$ kubectl exec my-nginx -- cat /etc/os-release

PRETTY_NAME="Debian GNU/Linux 8 (jessie)"
NAME="Debian GNU/Linux"
VERSION_ID="8"
VERSION="8 (jessie)"
ID=debian
HOME_URL="http://www.debian.org/"
SUPPORT_URL="http://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"

user@ubuntu:~$
```

Running `cat /etc/os-release` via `kubectl exec` produces the information we needed. The `exec` subcommand chooses the first container within the pod to execute the command. The command to run on the pod was separated from the rest of the `kubectl` invocation with `--`.

If you would like to execute the command within a specific container in a multi-container pod, you can use the *-c* switch.

Try it first find a deployed pod with more than one container:

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

NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
default       my-nginx                          1/1     Running   1          5m58s
kube-system   coredns-5644d7b6d9-b4rnz          1/1     Running   0          34m
kube-system   coredns-5644d7b6d9-lxdqv          1/1     Running   0          34m
kube-system   etcd-ubuntu                       1/1     Running   0          33m
kube-system   kube-apiserver-ubuntu             1/1     Running   0          34m
kube-system   kube-controller-manager-ubuntu    1/1     Running   0          33m
kube-system   kube-proxy-npxks                  1/1     Running   0          34m
kube-system   kube-scheduler-ubuntu             1/1     Running   0          33m
kube-system   weave-net-rvhvk                   2/2     Running   0          18m

user@ubuntu:~$
```

The weave-net pod for our cluster's networking has two containers in it.

Try to check the os-release on that pod, making sure to use the `--namespace kube-system` so kubectl knows which namespace to look:

```
user@ubuntu:~$ kubectl --namespace kube-system exec weave-net-rvhvk -- cat /etc/os-release

Defaulting container name to weave.
Use 'kubectl describe pod/weave-net-rvhvk -n kube-system' to see all of the containers in this
```

```
pod.
NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.8.4
PRETTY_NAME="Alpine Linux v3.8"
HOME_URL="http://alpinelinux.org"
BUG_REPORT_URL="http://bugs.alpinelinux.org"

user@ubuntu:~$
```

Let's do as it suggest and see what containers are inside the weave-net pod. The *describe pod* command will give you a list of the containers within the pod, but we can also use `kubectl get` to retrieve a pod's JSON output.

Use `kubectl get pod` with the `-o json` option to retrieve information about the weave-net pod:

```
user@ubuntu:~$ kubectl --namespace kube-system get pod weave-net-rvhvk -o json
{
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
        "creationTimestamp": "2020-01-08T20:31:47Z",
        "generateName": "weave-net-",
        "labels": {
            "controller-revision-hash": "7f54576664",
            "name": "weave-net",
            "pod-template-generation": "1"
        },
        "name": "weave-net-rvhvk",
        "namespace": "kube-system",
        "ownerReferences": [
            {
                "apiVersion": "apps/v1",
                "blockOwnerDeletion": true,
                "controller": true,
                "kind": "DaemonSet",
                "name": "weave-net",
                "uid": "2fb69d44-878b-4b93-b740-e5252ba94cb5"
            }
        ],
        "resourceVersion": "1627",
        "selfLink": "/api/v1/namespaces/kube-system/pods/weave-net-rvhvk",
        "uid": "0028c4b6-26bb-4d4b-91dc-859d2a4272c9"
    },
    "spec": {

...

        "hostIP": "192.168.228.157",
        "phase": "Running",
        "podIP": "192.168.228.157",
        "podIPs": [
            {
                "ip": "192.168.228.157"
            }
        ],
        "qosClass": "Burstable",
        "startTime": "2020-01-08T20:31:47Z"
    }
}

user@ubuntu:~$
```

That's a lot of information to sift through, as the entire running pod's spec is presented to you in a single JSON document. You can use a JSON processor like `jq` to filter for specific information from the JSON output.

Use `apt` to install jq:

```
user@ubuntu:~$ sudo apt install jq -y
```

```
[sudo] password for user:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  libonig2
The following NEW packages will be installed:
  jq libonig2
0 upgraded, 2 newly installed, 0 to remove and 262 not upgraded.

...

Setting up jq (1.5+dfsg-1ubuntu0.1) ...
Processing triggers for libc-bin (2.23-0ubuntu3) ...

user@ubuntu:~$
```

And now try to retrieve just the container names from the weave-net pod by piping the JSON output of weave-net to jq:

```
user@ubuntu:~$ kubectl --namespace kube-system get pod weave-net-rvhvk -o json | jq
".spec.containers[].name" -r

weave
weave-npc

user@ubuntu:~$
```

Perfect. You will be using the `-o json` option with `jq` throughout the rest of the labs to gather information about your pods.

Use the –*c* switch to display the `os-release` file from the weave-npc container in the weave-net pod:

```
user@ubuntu:~$ kubectl --namespace kube-system exec -c weave-npc weave-net-rvhvk -- cat /etc/os-
release

NAME="Alpine Linux"
ID=alpine
VERSION_ID=3.8.4
PRETTY_NAME="Alpine Linux v3.8"
HOME_URL="http://alpinelinux.org"
BUG_REPORT_URL="http://bugs.alpinelinux.org"

user@ubuntu:~$
```

Being able to execute commands on pods ad-hoc can be very useful in debugging and normal operation scenarios.

6. System Logs

Each of the services composing our Kubernetes cluster emits a log file. In the current configuration, the `kubelet` log is controlled by systemd.

You can use the `journalctl` command to tail (`-n`) the output for the kubelet service unit (`-u`)

```
user@ubuntu:~$ journalctl -n 400 --no-pager -u kubelet.service | grep -v "no observation"

-- Logs begin at Wed 2020-01-08 11:49:44 PST, end at Wed 2020-01-08 12:56:36 PST. --
Jan 08 12:15:59 ubuntu kubelet[68925]: W0108 12:15:59.068301   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d
Jan 08 12:15:59 ubuntu kubelet[68925]: E0108 12:15:59.655595   68925 kubelet.go:2187] Container
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker:
network plugin is not ready: cni config uninitialized
Jan 08 12:16:04 ubuntu kubelet[68925]: W0108 12:16:04.068919   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d
Jan 08 12:16:04 ubuntu kubelet[68925]: E0108 12:16:04.663617   68925 kubelet.go:2187] Container
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker:
```

```
network plugin is not ready: cni config uninitialized
Jan 08 12:16:09 ubuntu kubelet[68925]: W0108 12:16:09.069770   68925 cni.go:237] Unable to
update cni config: no networks found in /etc/cni/net.d

...

user@ubuntu:~$
```

The rest of our services are running as containers.

We can use the `kubectl logs` command to display log output from our pods. Remember that Kubernetes system services run within the kube-system namespace by convention.

List the pods in the *kube-system* namespace:

```
user@ubuntu:~$ kubectl get pods --namespace=kube-system

NAME                              READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-b4rnz          1/1     Running   0          43m
coredns-5644d7b6d9-lxdqv          1/1     Running   0          43m
etcd-ubuntu                       1/1     Running   0          42m
kube-apiserver-ubuntu             1/1     Running   0          42m
kube-controller-manager-ubuntu    1/1     Running   0          41m
kube-proxy-npxks                  1/1     Running   0          43m
kube-scheduler-ubuntu             1/1     Running   0          42m
weave-net-rvhvk                   2/2     Running   0          27m

user@ubuntu:~$
```

Now display the last 10 lines from the API service:

```
user@ubuntu:~$ kubectl logs --namespace=kube-system --tail=10 kube-apiserver-ubuntu

I0108 20:15:30.509997       1 storage_scheduling.go:148] all system priority classes are created
successfully or already exist.
I0108 20:15:30.771391       1 controller.go:606] quota admission added evaluator for:
roles.rbac.authorization.k8s.io
I0108 20:15:30.801684       1 controller.go:606] quota admission added evaluator for:
rolebindings.rbac.authorization.k8s.io
W0108 20:15:30.937485       1 lease.go:222] Resetting endpoints for master service "kubernetes"
to [192.168.228.157]
I0108 20:15:30.940434       1 controller.go:606] quota admission added evaluator for: endpoints
I0108 20:15:31.955961       1 controller.go:606] quota admission added evaluator for:
serviceaccounts
I0108 20:15:31.968713       1 controller.go:606] quota admission added evaluator for:
deployments.apps
I0108 20:15:32.310219       1 controller.go:606] quota admission added evaluator for:
daemonsets.apps
I0108 20:15:39.903929       1 controller.go:606] quota admission added evaluator for:
replicasets.apps
I0108 20:15:39.919553       1 controller.go:606] quota admission added evaluator for:
controllerrevisions.apps

user@ubuntu:~$
```

Each Kubernetes service has its own log verbosity and each can be tuned. You can learn much by tracking the operations involved in starting a deployment.

Create a new single pod deployment with a descriptive name can be tracked for activity in the logs:

```
user@ubuntu:~$ kubectl run --generator=run-pod/v1 mylogtracker --image nginx:1.11

pod/mylogtracker created

user@ubuntu:~$
```

Again list the k8s system services, from here we can pick which logs to search for our new pod.

```
user@ubuntu:~$ kubectl get pod --namespace=kube-system

NAME                               READY   STATUS    RESTARTS   AGE
coredns-5644d7b6d9-b4rnz           1/1     Running   0          43m
coredns-5644d7b6d9-lxdqv           1/1     Running   0          43m
etcd-ubuntu                        1/1     Running   0          42m
kube-apiserver-ubuntu              1/1     Running   0          43m
kube-controller-manager-ubuntu     1/1     Running   0          42m
kube-proxy-npxks                   1/1     Running   0          43m
kube-scheduler-ubuntu              1/1     Running   0          42m
weave-net-rvhvk                    2/2     Running   0          27m

user@ubuntu:~$
```

Try the controller manager server first:

```
user@ubuntu:~$ kubectl logs --namespace=kube-system kube-controller-manager-ubuntu | grep
mylogtracker

user@ubuntu:~$
```

Controller manager only deals with replicated pods (ones using a controller); there won't be anything here for us.

Now take a look at the kubelet log:

```
user@ubuntu:~$ journalctl -u kubelet.service | grep mylogtracker

Jan 08 12:59:27 ubuntu kubelet[68925]: I0108 12:59:27.649965   68925 reconciler.go:207]
operationExecutor.VerifyControllerAttachedVolume started for volume "default-token-7bqf5"
(UniqueName: "kubernetes.io/secret/306f090b-46fc-4512-8f4a-a8b18dcd1a9e-default-token-7bqf5")
pod "mylogtracker" (UID: "306f090b-46fc-4512-8f4a-a8b18dcd1a9e")

user@ubuntu:~$
```

Kubelet only reports information about our pod's secret, which is mounted as volume.

You can also view the events taking place within the Kubernetes cluster itself using the events resource type.

Try getting events with `kubectl`:

```
user@ubuntu:~$ kubectl get events --sort-by='{.lastTimestamp}'

LAST SEEN   TYPE     REASON                    OBJECT             MESSAGE
44m         Normal   NodeHasSufficientPID      node/ubuntu        Node ubuntu status is now:
NodeHasSufficientPID
44m         Normal   NodeHasNoDiskPressure     node/ubuntu        Node ubuntu status is now:
NodeHasNoDiskPressure
44m         Normal   NodeHasSufficientMemory   node/ubuntu        Node ubuntu status is now:
NodeHasSufficientMemory
44m         Normal   RegisteredNode            node/ubuntu        Node ubuntu event: Registered
Node ubuntu in Controller
44m         Normal   Starting                  node/ubuntu        Starting kube-proxy.
28m         Normal   NodeReady                 node/ubuntu        Node ubuntu status is now:
NodeReady
15m         Normal   Scheduled                 pod/my-nginx       Successfully assigned
default/my-nginx to ubuntu
15m         Normal   Pulling                   pod/my-nginx       Pulling image "nginx:1.11"
15m         Normal   Pulled                    pod/my-nginx       Successfully pulled image
"nginx:1.11"
13m         Normal   Pulled                    pod/my-nginx       Container image "nginx:1.11"
already present on machine
13m         Normal   Started                   pod/my-nginx       Started container my-nginx
13m         Normal   Created                   pod/my-nginx       Created container my-nginx
```

```
50s         Normal   Scheduled                 pod/mylogtracker   Successfully assigned
default/mylogtracker to ubuntu
49s         Normal   Created                   pod/mylogtracker   Created container mylogtracker
49s         Normal   Pulled                    pod/mylogtracker   Container image "nginx:1.11"
already present on machine
48s         Normal   Started                   pod/mylogtracker   Started container mylogtracker

user@ubuntu:~$
```

While your events will be different you can see the value of the cluster event log. You can display event data associated with a given resource by supplying its name. You can also control the output format.

For example to make the data machine readable you could output it in JSON:

```
user@ubuntu:~$ kubectl get events -o json | grep mylogtracker -A5 | tail

                "creationTimestamp": "2020-01-08T20:59:29Z",
                "name": "mylogtracker.15e805007f603ffb",
                "namespace": "default",
                "resourceVersion": "3674",
                "selfLink": "/api/v1/namespaces/default/events/mylogtracker.15e805007f603ffb",
                "uid": "e6afaa59-9294-4f24-b9f2-6340dd647fd0"
            },
            "reason": "Started",
            "reportingComponent": "",
            "reportingInstance": "",

user@ubuntu:~$
```

7. Cleaning Up

Now that we have given our new cluster a good test we can clean up by deleting the service and deployments we have created. The `kubectl delete` subcommand allows you to delete objects you have created in the cluster.

To begin, delete the *my-nginx* Service:

```
user@ubuntu:~$ kubectl get services

NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP   45m
my-nginx     ClusterIP   10.106.240.235   <none>        80/TCP    12m

user@ubuntu:~$
```

```
user@ubuntu:~$ kubectl delete service my-nginx

service "my-nginx" deleted

user@ubuntu:~$
```

```
user@ubuntu:~$ kubectl get services

NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   46m

user@ubuntu:~$
```

Do not delete the kubernetes service.

Next we can delete the pods:

```
user@ubuntu:~$ kubectl get pod

NAME           READY   STATUS    RESTARTS   AGE
my-nginx       1/1     Running   1          17m
mylogtracker   1/1     Running   0          2m31s

user@ubuntu:~$
```

You can specify multiple resouces to by placing spaces between each resource:

```
user@ubuntu:~$ kubectl delete pod my-nginx mylogtracker

pod "my-nginx" deleted
pod "mylogtracker" deleted

user@ubuntu:~$
```

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

No resources found in default namespace.

user@ubuntu:~$
```

You Kubernetes cluster should now be cleaned up and ready for the next lab:

```
user@ubuntu:~$ kubectl get services,deployments,replicasets,pods

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   47m

user@ubuntu:~$
```

Be sure to leave the `service/kubernetes` service!

Congratulations, you have completed the lab!