Microservices

Lab 7 – Orchestration, Cloud Foundry Container Runtime

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts. Kubernetes seeks to foster an ecosystem of components and tools that relieve the burden of running applications in public and private clouds and can run on a range of platforms, from your laptop, to VMs on a cloud provider, to racks of bare metal servers.

In this lab we will setup and explore a minimal Kubernetes cluster. Installation has several prerequisites:

- **Linux** – Our lab system vm is preinstalled with Ubuntu 16.04 though most Linux distributions supporting modern container managers will work. Kubernetes is easiest to install on RHEL/Centos 7 and Ubuntu 16.04.
- **Docker** – Kubernetes will work with a variety of container managers but Docker is the most tested and widely deployed (various minimum versions of Docker are required depending on the installation approach you take). The latest Docker version is almost always recommended, though Kubernetes is often not tested with the absolute latest version.
- **etcd** – Kubernetes requires a distributed key/value store to manage discovery and cluster metadata; though Kubernetes was originally designed to make this function pluggable, etcd is the only practical option.
- **Kubernetes** – Kubernetes is a microservice-based system and is composed of several services. The Kubelet handles container operations for a given node, the API server supports the main cluster API, etc.

This lab will walk you through a basic Kubernetes installation. The model below illustrates the Kubernetes master and worker node roles, we will run both on a single system.

K8s

Once we have converted our lab system into a single node Kubernetes Cluster we will explore the orchestrator's functionality by deploying a containerized microservice.

1. Install Kubernetes Package Support

With Linux running and Docker installed we can set up Kubernetes. Kubernetes packages are distributed in DEB and RPM formats. We will use the DEB based APT repository here: apt.kubernetes.io.

First we need to address a few Kubernetes installation prerequisites. Some apt package repos use the aptitude protocol however the Kubernetes packages are served of https so we need to add the apt https transport:

```
ubuntu@ip-10-0-0-195:~$ sudo apt-get update && sudo apt-get install -y apt-transport-https

Hit:1 https://download.docker.com/linux/ubuntu xenial InRelease
Hit:2 http://security.ubuntu.com/ubuntu xenial-security InRelease
Hit:3 http://us-east-1.ec2.archive.ubuntu.com/ubuntu xenial InRelease
Hit:4 http://us-east-1.ec2.archive.ubuntu.com/ubuntu xenial-updates InRelease
Hit:5 http://us-east-1.ec2.archive.ubuntu.com/ubuntu xenial-backports InRelease
Reading package lists... Done
Reading package lists... Done
Building dependency tree
Reading state information... Done
apt-transport-https is already the newest version (1.2.32).
0 upgraded, 0 newly installed, 0 to remove and 15 not upgraded.

ubuntu@ip-10-0-0-195:~$
```

> `apt-transport-https` was installed as part of the Docker setup above, but here for completeness.

Next add the Google cloud packages repo key so that we can install packages hosted by Google:

```
ubuntu@ip-10-0-0-195:~$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

OK
```

```
ubuntu@ip-10-0-0-195:~$
```

Now add a repository list file with an entry for Ubuntu Xenial apt.kubernetes.io packages. The following command copies the repo url into the "kubernetes.list" file:

```
ubuntu@ip-10-0-0-195:~$ 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@ip-10-0-0-195:~$
```

Update the package indexes to add the Kubernetes packages from apt.kubernetes.io:

```
ubuntu@ip-10-0-0-195:~$ sudo apt-get update

...
Get:2 https://packages.cloud.google.com/apt kubernetes-xenial InRelease [8,993 B]
Get:7 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 Packages [26.9 kB]
Fetched 35.9 kB in 0s (47.0 kB/s)
Reading package lists... Done

ubuntu@ip-10-0-0-195:~$
```

Notice the new packages.cloud.google.com repository above. If you *do not* see it in your terminal output, you must fix the entry in `/etc/apt/sources.list.d/kubernetes.list` before moving on!

Now we can install standard Kubernetes packages.

2. Install kubeadm and other prereq packages

Kubernetes 1.4 added alpha support for the kubeadm tool, version 1.6 moved it to beta and as of Kubernetes 1.13 Kubeadm is GA. The `kubeadmn` tool simplifies the process of installing a Kubernetes cluster. To use `kubeadm` we'll also need the `kubectl` cluster CLI tool and the `kubelet` node manager. We'll also install Kubernetes CNI (Container Network Interface) support for multi-host networking.

> Note: Kubeadm offers no cloud provider (AWS/GCP/etc.) integrations (load balancers, etc.). Kops, Kubespray and other tools are often used for K8s installation on cloud systems, however, many of these systems use Kubeadm under the covers.

Use the aptitude package manager to install the needed packages:

```
ubuntu@ip-10-0-0-195:~$ sudo apt-get install -y kubelet kubeadm kubectl kubernetes-cni

Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  conntrack cri-tools ebtables socat
The following NEW packages will be installed:
  conntrack cri-tools ebtables kubeadm kubectl kubelet kubernetes-cni socat

...

Setting up kubernetes-cni (0.7.5-00) ...
Setting up socat (1.7.3.1-1) ...
Setting up kubelet (1.15.0-00) ...
Setting up kubectl (1.15.0-00) ...
Setting up kubeadm (1.15.0-00) ...
Processing triggers for systemd (229-4ubuntu21.21) ...
Processing triggers for ureadahead (0.100.0-19) ...

ubuntu@ip-10-0-0-195:~$
```

3. Install and start the Kubernetes Master Components

Before we use kubeadm take a look at the kubeadm help menu:

```
ubuntu@ip-10-0-0-195:~$ kubeadm help

 ┌──────────────────────────────────────────────────────────┐
 │ KUBEADM                                                  │
 │ Easily bootstrap a secure Kubernetes cluster             │
```

```
 Please give us feedback at:
 https://github.com/kubernetes/kubeadm/issues

Example usage:

Create a two-machine cluster with one control-plane node
(which controls the cluster), and one worker node
(where your workloads, like Pods and Deployments run).

 On the first machine:

 control-plane# kubeadm init

 On the second machine:

 worker# kubeadm join <arguments-returned-from-init>

You can then repeat the second step on as many other machines as you like.

Usage:
kubeadm [command]

Available Commands:
alpha       Kubeadm experimental sub-commands
completion  Output shell completion code for the specified shell (bash or zsh)
config      Manage configuration for a kubeadm cluster persisted in a ConfigMap in the cluster
help        Help about any command
init        Run this command in order to set up the Kubernetes control plane
join        Run this on any machine you wish to join an existing cluster
reset       Run this to revert any changes made to this host by 'kubeadm init' or 'kubeadm join'
token       Manage bootstrap tokens
upgrade     Upgrade your cluster smoothly to a newer version with this command
version     Print the version of kubeadm

Flags:
-h, --help                     help for kubeadm
  --log-file string          If non-empty, use this log file
  --log-file-max-size uint   Defines the maximum size a log file can grow to. Unit is megabytes. If the value is 0,
the maximum file size is unlimited. (default 1800)
  --rootfs string            [EXPERIMENTAL] The path to the 'real' host root filesystem.
  --skip-headers             If true, avoid header prefixes in the log messages
  --skip-log-headers         If true, avoid headers when opening log files
-v, --v Level                  number for the log level verbosity

Use "kubeadm [command] --help" for more information about a command.

ubuntu@ip-10-0-0-195:~$
```

Check the kubeadm version:

```
ubuntu@ip-10-0-0-195:~$ kubeadm version

kubeadm version: &version.Info{Major:"1", Minor:"15", GitVersion:"v1.15.0",
GitCommit:"e8462b5b5dc2584fdcd18e6bcfe9f1e4d970a529", GitTreeState:"clean", BuildDate:"2019-06-19T16:37:41Z",
GoVersion:"go1.12.5", Compiler:"gc", Platform:"linux/amd64"}

ubuntu@ip-10-0-0-195:~$
```

With all of the necessary prerequisites installed we can now use `kubeadm` to initialize a cluster.

NOTE in the output below, this line: `[apiclient] All control plane components are healthy after 17.986496 seconds` indicates the approximate time it took to get the cluster up and running; this includes time spent downloading Docker images for the control plane components, generating keys, manifests, etc. This example was captured with an uncontended wired connection--yours may take 5-10 minutes on slow or shared wifi, *be patient!*.

```
ubuntu@ip-10-0-0-195:~$ sudo kubeadm init

[init] Using Kubernetes version: v1.15.0
```

```
[preflight] Running pre-flight checks
	[WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver. The recommended driver is
"systemd". Please follow the guide at https://kubernetes.io/docs/setup/cri/
[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'
[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] Activating the kubelet service
[certs] Using certificateDir folder "/etc/kubernetes/pki"
[certs] Generating "etcd/ca" certificate and key
[certs] Generating "etcd/server" certificate and key
[certs] etcd/server serving cert is signed for DNS names [ip-10-0-0-195 localhost] and IPs [10.0.0.195 127.0.0.1
::1]
[certs] Generating "etcd/healthcheck-client" certificate and key
[certs] Generating "apiserver-etcd-client" certificate and key
[certs] Generating "etcd/peer" certificate and key
[certs] etcd/peer serving cert is signed for DNS names [ip-10-0-0-195 localhost] and IPs [10.0.0.195 127.0.0.1 ::1]
[certs] Generating "ca" certificate and key
[certs] Generating "apiserver" certificate and key
[certs] apiserver serving cert is signed for DNS names [ip-10-0-0-195 kubernetes kubernetes.default
kubernetes.default.svc kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 10.0.0.195]
[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 "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
[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 16.502201 seconds
[upload-config] Storing the configuration used in ConfigMap "kubeadm-config" in the "kube-system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config-1.15" 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 ip-10-0-0-195 as control-plane by adding the label "node-
role.kubernetes.io/master=''"
[mark-control-plane] Marking the node ip-10-0-0-195 as control-plane by adding the taints [node-
role.kubernetes.io/master:NoSchedule]
[bootstrap-token] Using token: 4s5lh3.7sie575dom4nhpu2
[bootstrap-token] Configuring bootstrap tokens, cluster-info ConfigMap, RBAC Roles
[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
[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

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 10.0.0.195:6443 --token 4s5lh3.7sie575dom4nhpu2 \
    --discovery-token-ca-cert-hash sha256:519f3ae3c5824232f225cfcb9be036422c952c0c454d4596f3779652b889b055
```

```
ubuntu@ip-10-0-0-195:~$
```

> N.B. If you receive a swap error from kubeadm simply disable memory swapping on your lab system.
>
> user@ubuntu:~$ sudo swapoff -a
>
> You should also edit the /etc/fstab and comment out any swap volumes so that swap is not reenabled after a reboot.

Examine the output from kubeadm above; you **do not** need to follow the steps just now, we will be discussing and performing them during the rest of this lab. Note that we get preflight check warnings:

```
        [WARNING IsDockerSystemdCheck]: detected "cgroupfs" as the Docker cgroup driver. The recommended driver is
"systemd". Please follow the guide at https://kubernetes.io/docs/setup/cri/
```

The complaint is that Docker is using cgroupfs as the cgroup driver; Docker can use both cgroupfs and systemd so this error can be safely ignored.

Control groups, or cgroups, are used to constrain resources that are allocated to processes. Using *cgroupfs* alongside *systemd* means that there will then be two different cgroup managers. A single cgroup manager will simplify the view of what resources are being allocated and will by default have a more consistent view of the available and in-use resources, but using *cgroupfs* will not impact the operations performed in this lab and subsequent labs.

The `kubeadm` tool generates an auth token which we can use to add additional nodes to the cluster, and then creates the keys and certificates necessary for TLS. The initial master configures itself as a CA and self signs its certificate. All of the PKI/TLS related files can be found in `/etc/kubernetes/pki` .

```
ubuntu@ip-10-0-0-195:~$ ls -l /etc/kubernetes/pki/

total 60
-rw-r--r-- 1 root root 1216 Jul  11 19:04 apiserver.crt
-rw-r--r-- 1 root root 1090 Jul  11 19:04 apiserver-etcd-client.crt
-rw------- 1 root root 1679 Jul  11 19:04 apiserver-etcd-client.key
-rw------- 1 root root 1679 Jul  11 19:04 apiserver.key
-rw-r--r-- 1 root root 1099 Jul  11 19:04 apiserver-kubelet-client.crt
-rw------- 1 root root 1679 Jul  11 19:04 apiserver-kubelet-client.key
-rw-r--r-- 1 root root 1025 Jul  11 19:04 ca.crt
-rw------- 1 root root 1679 Jul  11 19:04 ca.key
drwxr-xr-x 2 root root 4096 Jul  11 19:04 etcd
-rw-r--r-- 1 root root 1038 Jul  11 19:04 front-proxy-ca.crt
-rw------- 1 root root 1675 Jul  11 19:04 front-proxy-ca.key
-rw-r--r-- 1 root root 1058 Jul  11 19:04 front-proxy-client.crt
-rw------- 1 root root 1679 Jul  11 19:04 front-proxy-client.key
-rw------- 1 root root 1679 Jul  11 19:04 sa.key
-rw------- 1 root root  451 Jul  11 19:04 sa.pub

ubuntu@ip-10-0-0-195:~$
```

The `.crt` files are certificates with public keys embedded and the `.key` files are private keys. The `apiserver` files are used by the `kube-apiserver` the ca files are associated with the certificate authority that `kubeadm` created and the `sa` files are the Service Account keys used to gain root control of the cluster. Clearly all of the files here with a key suffix should be carefully protected.

4. Exploring the Cluster

The `kubeadm` tool launches the `kubelet` on the local system to bootstrap the cluster services. Using the `kubelet` , the kubeadm tool can run the remainder of the Kubernetes services in containers. This is, as they say, eating one's own dog food. Kubernetes is a system promoting the use of microservice architecture and container packaging. Once the `kubelet` is running, the balance of the Kubernetes microservices can be launched via container images.

Display information on the `kubelet` process:

```
ubuntu@ip-10-0-0-195:~$ ps -fwwp $(pidof kubelet) | sed -e 's/--/\n--/g'

UID        PID  PPID  C STIME TTY          TIME CMD
root      6720     1  1 19:04 ?        00:00:02 /usr/bin/kubelet
--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
--kubeconfig=/etc/kubernetes/kubelet.conf
--config=/var/lib/kubelet/config.yaml
--cgroup-driver=cgroupfs
--network-plugin=cni
--pod-infra-container-image=k8s.gcr.io/pause:3.1

ubuntu@ip-10-0-0-195:~$
```

The possible switches used to launch the `kubelet` include:

- `--bootstrap-kubeconfig` - kubeconfig file that will be used to get client certificate for kubelet
- `--kubeconfig` - `kubelet` config file, contains `kube-apiserver` address and keys to authenticate with
- `--config` - sets the location of the kubelet's config file, detailing various runtime parameters for the kubelet
- `--cgroup-driver` - sets the container runtime interface. Defaults to cgroupfs (the same as docker)
- `--network-plugin` - sets the network plugin interface to be used
- `--pod-infra-container-image` - the image whose network/ipc namespaces containers in each pod will use

The `kubeadm` utility has configured the `kubelet` as a systemd service and enabled it so it will restart automatically when we reboot. Examine the `kubelet` service configuration:

```
ubuntu@ip-10-0-0-195:~$ systemctl --all --full status kubelet

● kubelet.service - kubelet: The Kubernetes Node Agent
   Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset: enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
   Active: active (running) since Thu 2019-07-11 19:04:31 UTC; 2min 20s ago
     Docs: https://kubernetes.io/docs/home/
 Main PID: 6720 (kubelet)
    Tasks: 16
   Memory: 35.6M
      CPU: 2.491s
   CGroup: /system.slice/kubelet.service
           └─6720 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --
kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --cgroup-driver=cgroupfs --

Jul 11 19:06:31 ip-10-0-0-195 kubelet[6720]: W0711 19:06:31.175970    6720 cni.go:213] Unable to update cni config:
No networks found in /etc/cni/net.d
Jul 11 19:06:31 ip-10-0-0-195 kubelet[6720]: E0711 19:06:31.338337    6720 kubelet.go:2169] Container runtime
network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker: network plu
...

q

ubuntu@ip-10-0-0-195:~$
```

As you can see from the "Loaded" line the service is enabled, indicating it will start on system boot.

Take a moment to review the systemd service start up files. First the service file:

```
ubuntu@ip-10-0-0-195:~$ sudo cat /lib/systemd/system/kubelet.service

[Unit]
Description=kubelet: The Kubernetes Node Agent
Documentation=https://kubernetes.io/docs/home/

[Service]
ExecStart=/usr/bin/kubelet
Restart=always
StartLimitInterval=0
RestartSec=10

[Install]
WantedBy=multi-user.target

ubuntu@ip-10-0-0-195:~$
```

This just starts the service (/usr/bin/kubelet) and restarts in after 10 seconds if it crashes.

Now look over the configuration files in the service.d directory:

```
ubuntu@ip-10-0-0-195:~$ sudo ls /etc/systemd/system/kubelet.service.d

10-kubeadm.conf

ubuntu@ip-10-0-0-195:~$
```

Files in this directory are processed in lexical order. The numeric prefix ("10") makes it easy to order the files. Display the one config file:

```
ubuntu@ip-10-0-0-195:~$ sudo cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf

# Note: This dropin only works with kubeadm and kubelet v1.11+
```



```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --
kubeconfig=/etc/kubernetes/kubelet.conf"
Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
# This is a file that "kubeadm init" and "kubeadm join" generates at runtime, populating the KUBELET_KUBEADM_ARGS
variable dynamically
EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
# This is a file that the user can use for overrides of the kubelet args as a last resort. Preferably, the user
should use
# the .NodeRegistration.KubeletExtraArgs object in the configuration files instead. KUBELET_EXTRA_ARGS should be
sourced from this file.
EnvironmentFile=-/etc/default/kubelet
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS $KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS

ubuntu@ip-10-0-0-195:~$
```

Let's take a look at the kubelet's configuration, which in the above output is found as: `Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml`

```
ubuntu@ip-10-0-0-195:~$ sudo cat /var/lib/kubelet/config.yaml

address: 0.0.0.0
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  anonymous:
    enabled: false
  webhook:
    cacheTTL: 2m0s
    enabled: true
  x509:
    clientCAFile: /etc/kubernetes/pki/ca.crt

    ...

ubuntu@ip-10-0-0-195:~$
```

Within the configuration yaml file, let's search for any settings that relate to a storage path:

```
ubuntu@ip-10-0-0-195:~$ sudo cat /var/lib/kubelet/config.yaml | grep Path

staticPodPath: /etc/kubernetes/manifests

ubuntu@ip-10-0-0-195:~$
```

This directory is created during the *kubeadm init* process. Once created, kubelet will monitor that directory for any pods that the kubelet will need to run at startup. Let's list its contents:

```
ubuntu@ip-10-0-0-195:~$ ls -l /etc/kubernetes/manifests/

total 16
-rw------- 1 root root 1920 Jul  11 19:04 etcd.yaml
-rw------- 1 root root 3276 Jul  11 19:04 kube-apiserver.yaml
-rw------- 1 root root 2824 Jul  11 19:04 kube-controller-manager.yaml
-rw------- 1 root root  990 Jul  11 19:04 kube-scheduler.yaml

ubuntu@ip-10-0-0-195:~$
```

Each of these files specifies a pod description for each key component of our cluster's master node:

- The `etcd` component is the key/value store housing our cluster's state.
- The `kube-apiserver` is the service implementing the Kubernetes API endpoints.
- The `kube-scheduler` selects nodes for new pods to run on.
- The `kube-controller-manager` ensures that the correct number of pods are running.

These YAML files tell the `kubelet` to run the associated cluster components in their own pods with the necessary settings and container images. Display the images used on your system:

```
ubuntu@ip-10-0-0-195:~$ sudo grep image /etc/kubernetes/manifests/*.yaml

/etc/kubernetes/manifests/etcd.yaml:    image: k8s.gcr.io/etcd:3.3.10
/etc/kubernetes/manifests/etcd.yaml:    imagePullPolicy: IfNotPresent
```

```
/etc/kubernetes/manifests/kube-apiserver.yaml:    image: k8s.gcr.io/kube-apiserver:v1.15.0
/etc/kubernetes/manifests/kube-apiserver.yaml:    imagePullPolicy: IfNotPresent
/etc/kubernetes/manifests/kube-controller-manager.yaml:    image: k8s.gcr.io/kube-controller-manager:v1.15.0
/etc/kubernetes/manifests/kube-controller-manager.yaml:    imagePullPolicy: IfNotPresent
/etc/kubernetes/manifests/kube-scheduler.yaml:    image: k8s.gcr.io/kube-scheduler:v1.15.0
/etc/kubernetes/manifests/kube-scheduler.yaml:    imagePullPolicy: IfNotPresent

ubuntu@ip-10-0-0-195:~$
```

In the example above, etcd v3.3.10 and Kubernetes 1.15 are in use. All of the images are dynamically pulled by Docker from the k8s.gcr.io registry server using the "google_containers" public namespace.

List the containers running under Docker:

```
ubuntu@ip-10-0-0-195:~$ docker container ls --format "{{.Command}}" --no-trunc | awk -F"--" '{print $1}'

"/usr/local/bin/kube-proxy
"/pause"
"etcd
"kube-scheduler
"kube-controller-manager
"kube-apiserver
"/pause"
"/pause"
"/pause"
"/pause"

ubuntu@ip-10-0-0-195:~$
```

We will discuss the `pause` containers later.

Several Kubernetes services are running:

- kube-proxy - Modifies the system iptables to support the service routing mesh (runs on all nodes)
- etcd - The key/value store used to hold Kubernetes cluster state
- kube-scheduler - The Kubernetes pod scheduler
- kube-controller-manager - The Kubernetes replica manager
- kube-apiserver - The Kubernetes api server

The `kube-proxy` service addon is included by `kubeadm`.

Configure kubectl

The command line tool used to interact with our Kubernetes cluster is `kubectl`. While you can use `curl` and other programs to communicate with Kubernetes at the API level, the `kubectl` command makes interacting with the cluster from the command line easy, packaging up your requests and making the API calls for you.

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

```
ubuntu@ip-10-0-0-195:~$ kubectl config view

apiVersion: v1
clusters: []
contexts: []
current-context: ""
kind: Config
preferences: {}
users: []

ubuntu@ip-10-0-0-195:~$
```

As you can see the only value we have configured is the apiVersion which is set to v1, the current Kubernetes API version. The `kubectl` command tries to reach the API server on port 8080 via the localhost loopback without TLS by default.

Kubeadm establishes a config file during deployment of the control plane and places it in /etc/kubernetes as admin.conf. We will take a closer look at this config file in lab 3 but for now follow the steps kubeadm describes in its output, placing it in a new .kube directory under your home directory.

```
ubuntu@ip-10-0-0-195:~$ mkdir -p $HOME/.kube

ubuntu@ip-10-0-0-195:~$ sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

ubuntu@ip-10-0-0-195:~$ sudo chown ubuntu $HOME/.kube/config

ubuntu@ip-10-0-0-195:~$
```

Verify the kubeconfig we just copied is understood:

```
ubuntu@ip-10-0-0-195:~$ kubectl config view

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://10.0.0.195:6443
  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

ubuntu@ip-10-0-0-195:~$
```

The default context should be `kubernetes-admin@kubernetes` .

```
ubuntu@ip-10-0-0-195:~$ kubectl config current-context

kubernetes-admin@kubernetes

ubuntu@ip-10-0-0-195:~$
```

If is not already active, activate the kubernetes-admin@kubernetes context:

```
ubuntu@ip-10-0-0-195:~$ kubectl config use-context kubernetes-admin@kubernetes

Switched to context "kubernetes-admin@kubernetes".

ubuntu@ip-10-0-0-195:~$
```

Verify that the new context can access the cluster:

```
ubuntu@ip-10-0-0-195:~$ kubectl get nodes

NAME             STATUS     ROLES    AGE     VERSION
ip-10-0-0-195    NotReady   master   5m12s   v1.15.0

ubuntu@ip-10-0-0-195:~$
```

You can now use kubectl to gather information about the resources deployed with your Kubernetes cluster, but it looks like not the cluster is ready for operation.

Taints

During the default initialization of the cluster, kubeadm applies labels and taints to the master node so that no workloads will run there. Because we want to run a one node cluster for testing, this will not do.

In Kubernetes terms, the master node is tainted. A taint consists of a *key*, a *value*, and an *effect*. The effect must be *NoSchedule*, *PreferNoSchedule* or *NoExecute*. You can view the taints on your node with the `kubectl` command. Use the `kubectl describe` subcommand to see details for the master node having the host name "ubuntu":

```
ubuntu@ip-10-0-0-195:~$ kubectl describe $(kubectl get node -o name) | grep -i taints

Taints:                  node-role.kubernetes.io/master:NoSchedule

ubuntu@ip-10-0-0-195:~$
```

We will examine the full describe output later but as you can see the master has the "node-role.kubernetes.io/master" taint with the effect "NoSchedule".

This means the `kube-scheduler` can not place pods on this node. To remove this taint we can use the `kubectl taint` subcommand.

NOTE The command below removes ("-") the master taint from all (--all) nodes in the cluster. **Do not forget the trailing `-` following the taint key "master"!** The `-` is what tells Kubernetes to remove the taint!

> We know what you're thinking and we agree, "taint" is an awful name for this feature and a trailing dash with no space is an equally wacky way to remove something.

```
ubuntu@ip-10-0-0-195:~$ kubectl taint nodes --all node-role.kubernetes.io/master-

node/ip-10-0-0-195 untainted

ubuntu@ip-10-0-0-195:~$
```

Check again to see if the taint was removed:

```
ubuntu@ip-10-0-0-195:~$ kubectl describe $(kubectl get node -o name) | grep -i taints

Taints:             node.kubernetes.io/not-ready:NoSchedule

ubuntu@ip-10-0-0-195:~$
```

What happened?

Our first clue is that the taint mentions status `not ready`. Let's grep "ready" status from the node.

```
ubuntu@ip-10-0-0-195:~$ kubectl describe $(kubectl get node -o name) | grep -i ready

Taints:             node.kubernetes.io/not-ready:NoSchedule
  Ready            False   Thu, 11 Jul 2019 19:10:38 +0000   Thu, 11 Jul 2019 19:04:32 +0000   KubeletNotReady
runtime network not ready: NetworkReady=false reason:NetworkPluginNotReady message:docker: network plugin is not
ready: cni config uninitialized

ubuntu@ip-10-0-0-195:~$
```

Let's fix that!

5. Enable Networking and Related Features

In the previous step, we found out that our master node is hung in the `not ready` status, with docker reporting that the network plugin is not ready.

Try listing the pods running on the cluster.

```
ubuntu@ip-10-0-0-195:~$ kubectl get pods

No resources found.

ubuntu@ip-10-0-0-195:~$
```

Nothing is returned because we are configured to view the "default" cluster namespace. System pods run in the Kubernetes "kube-system" namespace. You can show all namespaces by using the *--all-namspaces* switch.

```
ubuntu@ip-10-0-0-195:~$ kubectl get pods --all-namespaces

NAMESPACE     NAME                                    READY   STATUS    RESTARTS   AGE
kube-system   coredns-5c98db65d4-jklpz                0/1     Pending   0          6m14s
kube-system   coredns-5c98db65d4-vkqkj                0/1     Pending   0          6m14s
kube-system   etcd-ip-10-0-0-195                      1/1     Running   0          5m26s
kube-system   kube-apiserver-ip-10-0-0-195            1/1     Running   0          5m24s
kube-system   kube-controller-manager-ip-10-0-0-195   1/1     Running   0          4m59s
kube-system   kube-proxy-z2wln                        1/1     Running   0          6m14s
kube-system   kube-scheduler-ip-10-0-0-195            1/1     Running   0          5m17s


ubuntu@ip-10-0-0-195:~$
```

Our DNS pods are in a "Pending" state.

We'll want to take a look at the running docker containers on our master node. For more readable output, lets install `jq` to more easily search and format the resulting JSON output.

```
ubuntu@ip-10-0-0-195:~$ sudo apt-get install jq -y
```

```
...

ubuntu@ip-10-0-0-195:~$
```

Next we'll call the Docker remote api for DNS related containers and parse the response via jq:

```
ubuntu@ip-10-0-0-195:~$ curl -s --unix-sock /var/run/docker.sock http:/containers/json \
| jq '.[].Image | match(".*dns.*")'

ubuntu@ip-10-0-0-195:~$
```

We've confirmed that no container(s) with DNS in the name are running, though we do see system pods responsible for dns are visible. Notice the STATUS is Pending and 0 of 1 containers are Running for each pod.

Why are they failing to start? Lets review the POD related events for readiness.

```
ubuntu@ip-10-0-0-195:~$ kubectl get events --namespace=kube-system

LAST SEEN   TYPE      REASON              OBJECT                                        MESSAGE
5s          Warning   FailedScheduling    pod/coredns-5c98db65d4-jklpz                  0/1 nodes are available: 1
node(s) had taints that the pod didn't tolerate.
5s          Warning   FailedScheduling    pod/coredns-5c98db65d4-vkqkj                  0/1 nodes are available: 1
node(s) had taints that the pod didn't tolerate.
6m49s       Normal    SuccessfulCreate    replicaset/coredns-5c98db65d4                 Created pod: coredns-
5c98db65d4-vkqkj
6m49s       Normal    SuccessfulCreate    replicaset/coredns-5c98db65d4                 Created pod: coredns-
5c98db65d4-jklpz
6m49s       Normal    ScalingReplicaSet   deployment/coredns                            Scaled up replica set coredns-
5c98db65d4 to 2
7m7s        Normal    Pulled              pod/etcd-ip-10-0-0-195                        Container image
"k8s.gcr.io/etcd:3.3.10" already present on machine
7m7s        Normal    Created             pod/etcd-ip-10-0-0-195                        Created container etcd
7m7s        Normal    Started             pod/etcd-ip-10-0-0-195                        Started container etcd
7m7s        Normal    Pulled              pod/kube-apiserver-ip-10-0-0-195              Container image
"k8s.gcr.io/kube-apiserver:v1.15.0" already present on machine
7m7s        Normal    Created             pod/kube-apiserver-ip-10-0-0-195              Created container kube-
apiserver
7m7s        Normal    Started             pod/kube-apiserver-ip-10-0-0-195              Started container kube-
apiserver
7m7s        Normal    Pulled              pod/kube-controller-manager-ip-10-0-0-195     Container image
"k8s.gcr.io/kube-controller-manager:v1.15.0" already present on machine
7m7s        Normal    Created             pod/kube-controller-manager-ip-10-0-0-195     Created container kube-
controller-manager
7m7s        Normal    Started             pod/kube-controller-manager-ip-10-0-0-195     Started container kube-
controller-manager
6m58s       Normal    LeaderElection      endpoints/kube-controller-manager             ip-10-0-0-195_13207cb1-2fc2-
4b02-84bf-056343ddbd20 became leader
6m49s       Normal    Scheduled           pod/kube-proxy-z2wln                          Successfully assigned kube-
system/kube-proxy-z2wln to ip-10-0-0-195
6m48s       Normal    Pulled              pod/kube-proxy-z2wln                          Container image
"k8s.gcr.io/kube-proxy:v1.15.0" already present on machine
6m48s       Normal    Created             pod/kube-proxy-z2wln                          Created container kube-proxy
6m48s       Normal    Started             pod/kube-proxy-z2wln                          Started container kube-proxy
6m49s       Normal    SuccessfulCreate    daemonset/kube-proxy                          Created pod: kube-proxy-z2wln
7m7s        Normal    Pulled              pod/kube-scheduler-ip-10-0-0-195              Container image
"k8s.gcr.io/kube-scheduler:v1.15.0" already present on machine
7m7s        Normal    Created             pod/kube-scheduler-ip-10-0-0-195              Created container kube-
scheduler
7m7s        Normal    Started             pod/kube-scheduler-ip-10-0-0-195              Started container kube-
scheduler
6m58s       Normal    LeaderElection      endpoints/kube-scheduler                      ip-10-0-0-195_d8309339-01e3-
4082-ad8b-01d0d9965c62 became leader


ubuntu@ip-10-0-0-195:~$
```

That gives us a hint; we have a node running but why isn't it ready? It turns out that we told Kubernetes we would use CNI for networking but we have not yet supplied a CNI plugin. We can easily add the Weave CNI VXLAN based container networking drivers using a POD spec from the Internet.

The weave-kube path below points to a Kubernetes spec for a DaemonSet, which is a resource that runs on every node in a cluster. You can review that spec via curl:

```
ubuntu@ip-10-0-0-195:~$ curl -L \
```

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

apiVersion: v1
kind: List
items:
  - apiVersion: v1
    kind: ServiceAccount
    metadata:
      name: weave-net

...

ubuntu@ip-10-0-0-195:~$
```

You can test the spec without running it using the `--dry-run=true` switch:

```
ubuntu@ip-10-0-0-195:~$ kubectl apply -f \
"https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version | base64 | tr -d '\n')" \
--dry-run=true

serviceaccount/weave-net created (dry run)
clusterrole.rbac.authorization.k8s.io/weave-net created (dry run)
clusterrolebinding.rbac.authorization.k8s.io/weave-net created (dry run)
role.rbac.authorization.k8s.io/weave-net created (dry run)
rolebinding.rbac.authorization.k8s.io/weave-net created (dry run)
daemonset.extensions/weave-net created (dry run)

ubuntu@ip-10-0-0-195:~$
```

The config file creates several resources:

- The ServiceAccount, ClusterRole, ClusterRoleBinding, Role and Rolebinding configure the RBAC permissions for Weave
- The DaemonSet ensures that the weaveworks SDN images are running in a pod on all hosts

Run it for real this time by omitting the `--dry-run` switch:

```
ubuntu@ip-10-0-0-195:~$ 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.extensions/weave-net created

ubuntu@ip-10-0-0-195:~$
```

Rerun your "get pods" subcommand to ensure that all containers in all pods are running (it may take a minute for everything to start):

```
ubuntu@ip-10-0-0-195:~$ kubectl get pods --all-namespaces

NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   coredns-5c98db65d4-jklpz                   0/1     Pending   0          7m18s
kube-system   coredns-5c98db65d4-vkqkj                   0/1     Pending   0          7m18s
kube-system   etcd-ip-10-0-0-195                         1/1     Running   0          6m30s
kube-system   kube-apiserver-ip-10-0-0-195               1/1     Running   0          6m28s
kube-system   kube-controller-manager-ip-10-0-0-195      1/1     Running   0          6m3s
kube-system   kube-proxy-z2wln                           1/1     Running   0          7m18s
kube-system   kube-scheduler-ip-10-0-0-195               1/1     Running   0          6m21s
kube-system   weave-net-xjfwm                            1/2     Running   0          5s

ubuntu@ip-10-0-0-195:~$ kubectl get pods --all-namespaces

NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   coredns-5c98db65d4-jklpz                   1/1     Running   0          7m38s
kube-system   coredns-5c98db65d4-vkqkj                   1/1     Running   0          7m38s
kube-system   etcd-ip-10-0-0-195                         1/1     Running   0          6m50s
kube-system   kube-apiserver-ip-10-0-0-195               1/1     Running   0          6m48s
kube-system   kube-controller-manager-ip-10-0-0-195      1/1     Running   0          6m23s
kube-system   kube-proxy-z2wln                           1/1     Running   0          7m38s
kube-system   kube-scheduler-ip-10-0-0-195               1/1     Running   0          6m41s
kube-system   weave-net-xjfwm                            2/2     Running   0          25s
```

```
ubuntu@ip-10-0-0-195:~$
```

If you are fast enough, you will see that the weave-net pod goes online first. Once it is fully ready, the coredns pods will initialize shortly after.

If we check related DNS pod events once more, we see progress!

```
ubuntu@ip-10-0-0-195:~$ kubectl get events --namespace=kube-system --sort-by='{.lastTimestamp}' | grep dns

8m1s        Normal    SuccessfulCreate    replicaset/coredns-5c98db65d4              Created pod: coredns-
5c98db65d4-jklpz
8m1s        Normal    SuccessfulCreate    replicaset/coredns-5c98db65d4              Created pod: coredns-
5c98db65d4-vkqkj
8m1s        Normal    ScalingReplicaSet   deployment/coredns                         Scaled up replica set coredns-
5c98db65d4 to 2
44s         Warning   FailedScheduling    pod/coredns-5c98db65d4-jklpz               0/1 nodes are available: 1
node(s) had taints that the pod didn't tolerate.
33s         Normal    Scheduled           pod/coredns-5c98db65d4-jklpz               Successfully assigned kube-
system/coredns-5c98db65d4-jklpz to ip-10-0-0-195
33s         Warning   FailedScheduling    pod/coredns-5c98db65d4-vkqkj               0/1 nodes are available: 1
node(s) had taints that the pod didn't tolerate.
32s         Normal    Created             pod/coredns-5c98db65d4-jklpz               Created container coredns
32s         Normal    Started             pod/coredns-5c98db65d4-jklpz               Started container coredns
32s         Normal    Pulled              pod/coredns-5c98db65d4-jklpz               Container image
"k8s.gcr.io/coredns:1.3.1" already present on machine
31s         Normal    Scheduled           pod/coredns-5c98db65d4-vkqkj               Successfully assigned kube-
system/coredns-5c98db65d4-vkqkj to ip-10-0-0-195
30s         Normal    Pulled              pod/coredns-5c98db65d4-vkqkj               Container image
"k8s.gcr.io/coredns:1.3.1" already present on machine
30s         Normal    Created             pod/coredns-5c98db65d4-vkqkj               Created container coredns
30s         Normal    Started             pod/coredns-5c98db65d4-vkqkj               Started container coredns

ubuntu@ip-10-0-0-195:~$
```

Another way to view logs is to use `lastTimestamp` in conjunction with `-w` which watches the logs (type control c to exit this mode).

```
ubuntu@ip-10-0-0-195:~$ kubectl get events --namespace=kube-system -w --sort-by=lastTimestamp

...

^c
ubuntu@ip-10-0-0-195:~$
```

Lets look at the logs of the DNS related containers. We'll retrieve the names of our dns pods, then grab the logs from one of them.

```
ubuntu@ip-10-0-0-195:~$ DNSPOD=$(kubectl get pods -o name --namespace=kube-system |grep dns |head -1) && echo
$DNSPOD

pod/coredns-5c98db65d4-jklpz

ubuntu@ip-10-0-0-195:~$ kubectl logs --namespace=kube-system $DNSPOD

.:53
2019-07-11T19:12:19.780Z [INFO] CoreDNS-1.3.1
2019-07-11T19:12:19.780Z [INFO] linux/amd64, go1.11.4, 6b56a9c
CoreDNS-1.3.1
linux/amd64, go1.11.4, 6b56a9c
2019-07-11T19:12:19.780Z [INFO] plugin/reload: Running configuration MD5 = 5d5369fbc12f985709b924e721217843

ubuntu@ip-10-0-0-195:~$
```

Perfect! Kubernetes is up and running.

6. Run a Container

Now that we have a cluster (of one) running we can run some containers. The atomic unit of deployment on Kubernetes is actually a Pod however. Pods define a collection of one or more containers and their settings which will be deployed as a unit. For example, to run our trash can inventory microservice we create a pod which contains just our trash can inventory container.

With the right permissions we can create a Pod on the fly using kubectl's run sub command. Try it:

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

pod/web created
```



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

the command above runs a pod using the "run-pod/v1" generator. The run command has changed a bit over the years in Kubernetes and may ultimately go away, but for now we can use it to easily start a pod in this way. the pod was named "web" and is based on the image "nginx".

Now examine your pod:

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

NAME   READY   STATUS    RESTARTS   AGE
web    1/1     Running   0          66s

user@ubuntu:~$
```

If your pod is not yet running the docker engine on the node is likely pulling the image for nginx down from docker hub. Keep checking on your pod until it is running.

We can verify that our nginx pod is up and running with curl but we need to get the pod's ip address. You can discover the ip address by displaying pod data with the wide output:

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

NAME   READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
web    1/1     Running   0          73s   10.32.0.4   ubuntu   <none>           <none>

user@ubuntu:~$
```

Now try to curl your pod:

```
user@ubuntu:~$ curl 10.32.0.4

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>

user@ubuntu:~$
```

This works because the Weave SDN we are using to manage Pod networking creates a route on the host to the "weave" network interface (a software based device connected to the weave pod network).

List the routes on your host:

```
user@ubuntu:~$ ip route

default via 192.168.83.2 dev ens33
10.32.0.0/12 dev weave  proto kernel  scope link  src 10.32.0.1
172.17.0.0/16 dev docker0  proto kernel  scope link  src 172.17.0.1
192.168.83.0/24 dev ens33  proto kernel  scope link  src 192.168.83.141

user@ubuntu:~$
```

As you can see, all traffic destined for 10.32/12 is delivered to the "weave" device. Not all SDNs support traffic from the host into the Pod network by default.

In some cases we would need to run a client pod to test the service pod. try it:

```
user@ubuntu:~$ kubectl run client --image busybox --generator=run-pod/v1 -it

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

/ # wget -O - 10.32.0.4

Connecting to 10.32.0.4 (10.32.0.4:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
-                    100%
|**************************************************************************************************************************
******|   612  0:00:00 ETA
written to stdout

/ # exit

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

user@ubuntu:~$
```

List your pods again:

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

NAME     READY   STATUS    RESTARTS   AGE
client   1/1     Running   1          78s
web      1/1     Running   0          14m

user@ubuntu:~$
```

Our client pod is still running and, per the comment from kubectl on exit, we can reattach to it whenever we like.

7. Run a Deployment

Next let's try running our inventory container. We'll use a more formal approach to run our inventory service in a pod. In general, devops teams do not run pods directly, they run deployments. Also devops teams create declarative manifests that specify the target state that they want to achieve. Unlike imperative commands, these declarative specs tell Kubernetes what you want, rather than how to get it. This way if something happens at 3AM that causes your desired state to be compromised, Kubernetes (not you) jumps into action to start a new pod, move a pod or whatever needs to be done to achieve your declared desired state.

Here's an example of a deployment that will run two copies of our inventory service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: can-inventory
```

```
  labels:
    app: inv
spec:
  replicas: 2
  selector:
    matchLabels:
      app: inv
  template:
    metadata:
      labels:
        app: inv
    spec:
      containers:
      - name: inv
        image: trash-can/inv:0.1
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
```

The `apiVersion` specifies the version of the resource we are using and the `kind` defines the resource type. A deployment in Kubernetes describes a pod template and a number of replicas of that pod to maintain. The deployment and the pods created have `metadata` which can include names and labels. Labels are simply arbitrary key value pairs you use to identify you resources. Typical labels might include the name of the team that owns the deployment ("team: green64"), the name of the application ("app: inv") or perhaps the criticality of the deployment ("criticality: low").

The `spec` section lists the operable parameters of hte resource. For example the spec of the deployment identifies the number of pods to create (`replicas`), the `selector` used to identify pods owned by the deployment and a `template` for the pods to create.

The pod template includes the metadata for the pods created and the pod spec. At a minimum a pod spec must identify the container image to run. In the example above, the container `imagePullPolicy` specifies when and if the kublet should pull the container image. In our case the container image does not exist in a network registry, rather it is in the local docker image cache. The "Never" pull policy ensures that docker will not be asked to pull the image from docker hub (or any other registry). The `ports` key is used to list and ports the container might listen on. In our case the inventory service listens on port 8080.

Create the deployment manifest and apply it to your kubernetes cluster to deploy the two pod replicas:

```
user@ubuntu:~$ cd trash-can/

user@ubuntu:~/trash-can$ cd inv/

user@ubuntu:~/trash-can/inv$ vim deploy.yaml

user@ubuntu:~/trash-can/inv$ cat deploy.yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: can-inventory
  labels:
    app: inv
spec:
  replicas: 2
  selector:
    matchLabels:
      app: inv
  template:
    metadata:
      labels:
        app: inv
    spec:
      containers:
      - name: inv
        image: trash-can/inv:0.1
        imagePullPolicy: Never
        ports:
        - containerPort: 8080
```

```
user@ubuntu:~/trash-can/inv$ kubectl apply -f deploy.yaml

deployment.apps/can-inventory created
```

```
user@ubuntu:~/trash-can/inv$
```

Great we're deployed! Well ... actually the "created" response means only that the manifest was successfully saved into the Kubernetes cluster state store (etcd). Many things can go wrong at this point, the container could crash on startup, the image might not be found, etc.

Let's verify that the deployment is up:

```
user@ubuntu:~/trash-can/inv$ kubectl get deploy

NAME            READY   UP-TO-DATE   AVAILABLE   AGE
can-inventory   2/2     2            2           2m47s

user@ubuntu:~/trash-can/inv$
```

The kubectl get command displays our deployment and show both pods up and running. Deployments create ReplicaSets under the covers to manage the set of pods with a given template. This way, when you upgrade the container images deployed the deployment can create a new ReplicaSet without losing the old one, allowing you to undo the rollout if needed.

List the ReplicaSets and Pods in your cluster:

```
user@ubuntu:~/trash-can/inv$ kubectl get rs

NAME                       DESIRED   CURRENT   READY   AGE
can-inventory-756c87dc9f   2         2         2       2m56s

user@ubuntu:~/trash-can/inv$ kubectl get pod

NAME                             READY   STATUS    RESTARTS   AGE
can-inventory-756c87dc9f-b45ns   1/1     Running   0          3m1s
can-inventory-756c87dc9f-lmxcp   1/1     Running   0          3m1s
client                           1/1     Running   1          14h
web                              1/1     Running   0          14h

user@ubuntu:~/trash-can/inv$
```

Looks good! As you can see the ReplicaSet and Pods have generated names that use the base name of the Deployment. Each Pod name is also prefixed with the id of its ReplicaSet.

Let's try accessing our pods. We can use a wide pod listing to get our pod IP addresses:

```
user@ubuntu:~/trash-can/inv$ kubectl get pod -o wide

NAME                             READY   STATUS    RESTARTS   AGE     IP          NODE     NOMINATED NODE
READINESS GATES
can-inventory-756c87dc9f-b45ns   1/1     Running   0          9m56s   10.32.0.7   ubuntu   <none>           <none>
can-inventory-756c87dc9f-lmxcp   1/1     Running   0          9m56s   10.32.0.6   ubuntu   <none>           <none>
client                           1/1     Running   1          14h     10.32.0.5   ubuntu   <none>           <none>
web                              1/1     Running   0          14h     10.32.0.4   ubuntu   <none>           <none>

user@ubuntu:~/trash-can/inv$
```

Now try curling the pods from the host system:

```
user@ubuntu:~/trash-can/inv$ curl 10.32.0.6:8080

{"version":"0.1"}

user@ubuntu:~/trash-can/inv$ curl 10.32.0.7:8080

{"version":"0.1"}

user@ubuntu:~/trash-can/inv$
```

Bingo! Our inventory service is running on Kubernetes!

Let's try the same test from our client pod to ensure that pods can reach our inventory service:

```
user@ubuntu:~/trash-can/inv$ kubectl attach client -it

Defaulting container name to client.
Use 'kubectl describe pod/client -n default' to see all of the containers in this pod.
If you don't see a command prompt, try pressing enter.
```

```
/ # wget -qO - 10.32.0.6:8080

{"version":"0.1"}

/ # wget -qO - 10.32.0.7:8080

{"version":"0.1"}

/ # exit

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

user@ubuntu:~/trash-can/inv$
```

Great the Pod network is working too!

8. Run a Service

Imagine your are coding a client pod for the Inventory service. Assuming the Inventory pods save their state in a shared data store, do you care which pod you talk to? Probably not. Do you want to have to look up the pod IPs and pick one? Probably not.

Kubernetes provides a simple resource type called a `service` for naming and load balancing connections to a deployment. Let's create a service and try using it to access our inventory service. Here's a sample service manifest:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: inv
spec:
  selector:
    app: inv
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

Kubernetes services acquire a "ClusterIP" address, also know as a virtual IP (VIP) address. When created, the KubeProxy agents running on all of the Kubernetes nodes will insert rules into the iptables of the hosts to forward connections targeting the service VIP on to one of the pods.

The service selector is used to identify which pods should be included in the service load balancing list. the ports section specifies that connection should be made to the service on port 80 but that they should be forwarded (port mapped) to port 8080 on the pod side.

Create the Service:

```
user@ubuntu:~/trash-can/inv$ vim service.yaml

user@ubuntu:~/trash-can/inv$ cat service.yaml

apiVersion: v1
kind: Service
metadata:
  name: inv
spec:
  selector:
    app: inv
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080

user@ubuntu:~/trash-can/inv$ kubectl apply -f service.yaml

service/inv created

user@ubuntu:~/trash-can/inv$ kubectl get service

NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
inv          ClusterIP   10.103.51.48   <none>        80/TCP    7s
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP   41h

user@ubuntu:~/trash-can/inv$ kubectl describe  service inv

Name:              inv
Namespace:         default
Labels:            <none>
Annotations:       kubectl.kubernetes.io/last-applied-configuration:
```

```
                      {"apiVersion":"v1","kind":"Service","metadata":{"annotations":
{},"name":"inv","namespace":"default"},"spec":{"ports":[{"port":80,"protocol...
Selector:          app=inv
Type:              ClusterIP
IP:                10.103.51.48
Port:              <unset>  80/TCP
TargetPort:        8080/TCP
Endpoints:         10.32.0.6:8080,10.32.0.7:8080
Session Affinity:  None
Events:            <none>

user@ubuntu:~/trash-can/inv$
```

Great, our service is created and, as you can see from the `kubectl describe service` output, the service has the correct pod endpoints. Let's try accessing the service from the host:

```
user@ubuntu:~/trash-can/inv$ curl inv

<wait a long time here>

user@ubuntu:~/trash-can/inv$
```

The command hangs. Why doesn't it work?! Because the DNS system that knows the service name is only configured in the /etc/resolv.conf of the pods on the cluster. Let's test the service again from within the client pod:

```
user@ubuntu:~/trash-can/inv$ kubectl attach client -it

Defaulting container name to client.
Use 'kubectl describe pod/client -n default' to see all of the containers in this pod.
If you don't see a command prompt, try pressing enter.

/ # wget -qO - inv

{"version":"0.1"}

/ # cat /etc/resolv.conf

nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local cluster.local localdomain
options ndots:5

/ # exit

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

user@ubuntu:~/trash-can/inv$
```

The wget on `inv` works from inside the pod because the pod uses the Kubernetes nameserver. The Kubernetes DNS server runs in the kube-system namespace. Try listing it:

```
user@ubuntu:~/trash-can/inv$ kubectl get service -n kube-system

NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   41h

user@ubuntu:~/trash-can/inv$
```

As you can see, the Cluster IP of the Kubernetes DNS service is the nameserver IP added to the pod /etc/resolve.conf. The kubelet configures this resolv.conf in every pod to ensure service resolution operates correctly.

Congratulations, you have completed the lab!