LFD259

Kubernetes for Developers

Version 2018-10-21

LFD259: Version 2018-10-21

Published by:

the **Linux Foundation**
http://www.linuxfoundation.org

Contents

Chapter 1

Course Introduction

1.1 Lab

Chapter 2

Kubernetes Architecture

2.1 Labs

Exercise 2.1: Deploy a New Cluster

Overview

We will create a two-node Ubuntu 16.04 cluster. Using two nodes allows an understanding of some issues and configurations found in a production environment. While 2 vCPU and 8G of memory allows for quick labs you could use much smaller VMs. Other Linux distributions should work in a very similar manner, but have not been tested.

Regardless of the platform used, VirtualBox, VMWare, AWS, GCE or even bare metal please remember that security software like SELinux, AppArmor, and firewalls can prevent the labs from working. While not something to do in production consider disabling the firewall and security software. GCE requires a new `VPC` to be created and a rule allowing all traffic to be included. The use of **wireshark** can be a helpful place to start with troubleshooting network and connectivity issues if youre unable to open all ports. The **kubeadm** utility currently requires that swap be turned off on every node. The **swapoff -a** command will do this until the next reboot, with various methods to disable swap persistently. Cloud providers typically deploy instances with swap disabled.

To assist with setting up your cluster please download the tarball of shell scripts and YAML files. The `k8sMaster.sh` and `k8sSecond.sh` scripts deploy a Kubernetes cluster using **kubeadm** and use `Project Calico` for networking. Should the file not be found you can always use a browser to investigate the parent directory.

```
student@ckad-1:~$ wget \
https://training.linuxfoundation.org/cm/LFD259/lfd259-example-file.1-12-1.tar \
--user=LFtraining --password=Penguin2014

student@ckad-1:~$ tar -xvf lfd259-example-file.1-12-1.tar
<output_omitted>
```

Deploy a Master Node using Kubeadm

1. Review the script to install and begin the configuration of the master kubernetes server.

```
student@ckad-1:~$ cat lfd259/k8sMaster.sh
#!/bin/bash -x
echo "This script is written to work with Ubuntu 16.04"
sleep 3
echo
echo "Disable swap until next reboot"
echo
sudo swapoff -a
echo "Update the local node"
sudo apt-get update && sudo apt-get upgrade -y
echo
echo "Install Docker"
sleep 3
sudo apt-get install -y docker.io
echo
echo "Install kubeadm and kubectl"
sleep 3
sudo sh -c "echo 'deb http://apt.kubernetes.io/ kubernetes-xenial main'
 >> /etc/apt/sources.list.d/kubernetes.list
<output_omitted>
```

2. Run the script as an argument to the bash shell. You will need the **kubeadm join** command shown near the end of the output when you add the worker/minion node in a future step.

```
student@ckad-1:~$ bash lfd259/k8sMaster.sh
<output_omitted>

Your Kubernetes master 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/

You can now join any number of machines by running the following on
each node as root:

  kubeadm join 10.128.0.3:6443 --token 69rdjq.2x2012j9ncexy37b
  --discovery-token-ca-cert-hash
sha256:72143e996ef78301191b9a42184124416aebcf0c7f363adf9208f9fa599079bd

<output_omitted>

+ kubectl get node
NAME              STATUS     ROLES     AGE       VERSION
ckad-1            NotReady    master      18s           v1.12.1
+ echo
+ echo 'Script finished. Move to the next step'
Script finished. Move to the next step
```

Deploy a Minion Node

3. Open a separate terminal into your second node. Having both terminal sessions allows you to monitor the status of the cluster while adding the second node. Copy the **k8sSecond.sh** file to the second node. View the file. You should see the same early steps as on the master system.

```
student@ckad-2:~$ cat lfd259/k8sSecond.sh
#!/bin/bash -x
sudo apt-get update && sudo apt-get upgrade -y
<output_omitted>
```

4. Run the script on the second node.

```
student@ckad-2:~$ bash lfd259/k8sSecond.sh
<output_omitted>
```

5. When the script is done the minion node is ready to join the cluster. The **kubeadm join** statement can be found near the end of the **kubeadm init** output on the master node. Your nodes will use a different IP address and hashes than the example below. Youll need to pre-pend **sudo** to run the script copied from the master node.

```
student@ckad-2:~$ sudo kubeadm join --token 118c3e.83b49999dc5dc034 \
 10.128.0.3:6443 --discovery-token-ca-cert-hash \
sha256:40aa946e3f53e38271bae24723866f56c86d77efb49aedeb8a70cc189bfe2e1d
<output_omitted>
```

Configure the Master Node

6. Return to the master node. We will configure command line completion and verify both nodes have been added to the cluster. The first command will configure completion in the current shell. The second command will ensure future shells have completion.

```
student@ckad-1:~$ source <(kubectl completion bash)

student@ckad-1:~$ echo "source <(kubectl completion bash)" >> ~/.bashrc
```

7. Verify that both nodes are part of the cluster. Until we remove taints the nodes may not reach `Ready` state.

```
student@ckad-1:~$ kubectl get node
NAME        STATUS     ROLES     AGE         VERSION
ckad-1      NotReady   master    4m11s       v1.12.1
ckad-2      NotReady   <none>    3m6s        v1.12.1
```

8. We will use the **kubectl** command for the majority of work with Kubernetes. Review the help output to become familiar with commands options and arguments.

```
student@ckad-1:~$ kubectl --help
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
  run-container  Run a particular image on the cluster. This
 command is deprecated, use "run" instead

Basic Commands (Intermediate):
<output_omitted>
```

9. With more than 40 arguments, you can explore each also using the **–help** option. Take a closer look at a few, starting with **taint** for example.

```
student@ckad-1:~$ kubectl taint --help
Update the taints on one or more nodes.

  * A taint consists of a key, value, and effect. As an argument
   here, it is expressed as key=value:effect.
  * The key must begin with a letter or number, and may contain
   letters, numbers, hyphens, dots, and underscores, up to
   253 characters.
  * Optionally, the key can begin with a DNS subdomain prefix
   and a single '/',
like example.com/my-app
<output_omitted>
```

10. By default the master node will not allow general containers to be deployed for security reasons. This is via a taint. Only containers which tolerate this taint will be scheduled on this node. As we only have two nodes in our cluster we will remove the taint, allowing containers to be deployed on both nodes. This is not typically done in a production environment for security and resource contention reasons. The following command will remove the taint from all nodes, so you should see one success and one not found error. The worker/minion node does not have the taint to begin with. Note the **minus sign** at the end of the command, which removes the preceding value.

```
student@ckad-1:~$  kubectl describe nodes |grep -i Taint
Taints:             node-role.kubernetes.io/master:NoSchedule
Taints:             node.kubernetes.io/not-ready:NoSchedule

student@ckad-1:~$ kubectl taint nodes --all \
      node-role.kubernetes.io/master-

node/ckad-1 untainted
taint "node-role.kubernetes.io/master:" not found
```

11. As of `v1.12.1` with the early October version of **Calico** new nodes now have another taint, `node.kubernetes.io/not-ready:NoSchedule`. We need to remove this taint so that the scheduler will begin to allocate **Calico** pods to both nodes. They need to be running prior to nodes showing `Ready`. As of this writing **the taint removal may take two or three times to work**. Wait a minute, view taints and run the command again until the taints are removed.

```
student@ckad-1:~$ kubectl describe nodes |grep -i taint
Taints:             node.kubernetes.io/not-ready:NoSchedule
Taints:             node.kubernetes.io/not-ready:NoSchedule

student@ckad-1:~$ kubectl taint nodes --all node.kubernetes.io/not-ready-
node/ckad-1 untainted
node/ckad-2 untainted

student@ckad-1:~$  kubectl describe nodes |grep -i Taint
Taints:             node.kubernetes.io/not-ready:NoSchedule
Taints:             <none>


student@ckad-1:~$ sleep 60 ; \
   kubectl taint nodes --all node.kubernetes.io/not-ready-
node/ckad-1 untainted
taint "node.kubernetes.io/not-ready:" not found
```

12. Check that both nodes are without a `Taint`. If they both are without taint the `nodes` should now show as `Ready`. It may take a minute or two for all pods to enter `Ready` state.

```
student@ckad-1:~$ kubectl describe nodes |grep -i taint
Taints:           <none>
Taints:           <none>

student@ckad-1:~$ kubectl get nodes
NAME                    STATUS   ROLES    AGE     VERSION
```

```
ckad-1               Ready    master   6m1s    v1.12.1
ckad-2               Ready    <none>   5m31s   v1.12.1
```

Create a Basic Pod

1. The smallest unit we directly control with Kubernetes is the pod. We will create a pod by creating a minimal YAML file. First we will get a list of current API objects and their `APIGROUP`. If value is not shown it may not exist, as with `SHORTNAMES`. Note that `pods` does not declare an `APIGROUP`. At the moment this indicates it is part of the stable `v1` group.

   ```
   student@ckad-1:~$ kubectl api-resources
   NAME                 SHORTNAMES   APIGROUP   NAMESPACED   KIND
   bindings                                     true         Binding
   componentstatuses    cs                      false        ComponentStatus
   configmaps           cm                      true         ConfigMap
   endpoints            ep                      true         Endpoints
   .....
   pods                 po                      true         Pod
   ....
   ```

2. Finding no declared `APIGROUP` we will use `v1` to denote a stable object. With that information we will add the other three required sections such as `metadata`, with a `name`, and `spec` which declares which **Docker** image to use and a `name` for the container. We will create a eight line YAML file. White space and indentation matters. Don't use **Tab**s.

   ```
   student@ckad-1:~$ vim basic.yaml
   apiVersion: v1
   kind: Pod
   metadata:
     name: basicpod
   spec:
     containers:
     - name: webcont
       image: nginx
   ```

3. Create the new pod using the recently created YAML file.

   ```
   student@ckad-1:~$ kubectl create -f basic.yaml
   pod/basicpod created
   ```

4. Make sure the pod has been created then use the **describe** sub-command to view the details. Among other values in the output you should be about to find the image and the container name.

   ```
   student@ckad-1:~$ kubectl get pod
   NAME       READY   STATUS    RESTARTS   AGE
   basicpod   1/1     Running   0          23s

   student@ckad-1:~$ kubectl describe pod basicpod
   Name:               basicpod
   Namespace:          default
   Priority:           0
   <output_omitted>
   ```

5. Shut down the pod and verify it is no longer running.

   ```
   student@ckad-1:~$ kubectl delete pod basicpod
   pod "basicpod" deleted

   student@ckad-1:~$ kubectl get pod
   No resources found.
   ```

6. We will now configure the pod to expose port 80. This configuration does not interact with the container to determine what port to open. We have to know what port the process inside the container is using, in this case port 80 as a web server. Add two lines to the end of the file. Line up the indentation with the `image` declaration.

```
student@ckad-1:~$ vim basic.yaml
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
spec:
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
```

7. Create the pod and verify it is running. Use the **-o wide** option to see the internal IP assigned to the pod. Using **curl** you should get the default web page.

```
student@ckad-1:~$ kubectl create -f basic.yaml
pod/basicpod created

student@ckad-1:~$ kubectl get pod -o wide
NAME     READY STATUS  RESTARTS AGE  IP              NODE                     NOMINATED NODE
basicpod 1/1   Running 0        104s 192.168.213.147   ckad-1
<none>


student@ckad-1:~$ curl http://192.168.213.147
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>

<output_omitted>

student@ckad-1:~$ kubectl delete pod basicpod
pod "basicpod" deleted
```

8. We will now create a simple service to expose the pod to other nodes and pods in the cluster. The `service` YAML will have the same four sections as a pod, but different `spec` configuration and the addition of a `selector`. We will also add a `label` to the pod and a `selector` to the service so it knows which object to communicate with.

```
student@ckad-1:~$ vim basicservice.yaml
apiVersion: v1
kind: Service
metadata:
    name: basicservice
spec:
  selector:
    type: webserver
  ports:
  - protocol: TCP
    port: 80

student@ckad-1:~$ vim basic.yaml
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
  labels:                   #<-- Add this line
    type: webserver         #<-- and this line which matches selector
spec:
....
```

9. Create the new pod and service. Verify both have been created.

```
student@ckad-1:~$ kubectl create -f basic.yaml
pod/basicpod created

student@ckad-1:~$ kubectl create -f basicservice.yaml
service/basicservice created

student@ckad-1:~$ kubectl get pod
NAME       READY   STATUS    RESTARTS   AGE
basicpod   1/1     Running   0          110s

student@ckad-1:~$ kubectl get svc
NAME          TYPE       CLUSTER-IP      EXTERNAL-IP  PORT(S)  AGE
basicservice  ClusterIP  10.100.223.139  <none>       80/TCP   104s
kubernetes    ClusterIP  10.96.0.1       <none>       443/TCP  47h
```

10. Test access to the web server using the `CLUSTER-IP` for the `basicservice`.

```
student@ckad-1:~$ curl http://10.100.223.139
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>

<output_omitted>
```

11. We will now expose the service to outside the cluster as well. Delete the service, edit the file and add a `type` declaration.

```
student@ckad-1:~$ kubectl delete svc basicservice
service "basicservice" deleted

student@ckad-1:~$  vim basicservice.yaml
apiVersion: v1
kind: Service
metadata:
    name: basicservice
spec:
  selector:
    type: webserver
  type: NodePort       #<--Add this line
  ports:
  - protocol: TCP
    port: 80
```

12. Create the service again. Note there is a different `TYPE` and `CLUSTER-IP` and also a high-numbered port.

```
student@ckad-1:~$  kubectl create -f basicservice.yaml
service/basicservice created

student@ckad-1:~$ kubectl get svc
NAME         TYPE      CLUSTER-IP     EXTERNAL-IP PORT(S)        AGE
basicservice NodePort  10.100.139.155 <none>      80:31514/TCP   3s
kubernetes   ClusterIP 10.96.0.1      <none>      443/TCP        47h
```

13. Using the public IP address of the node and the high port you should be able to test access to the webserver. In the example below the public IP is `35.238.3.83`, yours will be different. The high port will also probably be different.

```
local$ curl http://35.238.3.83:31514
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
<output_omitted>
```

14. Shut down the pod to prepare for the next section.

```
student@ckad-1:~$ kubectl delete pod basicpod
pod "basicpod" deleted
```

Multi-Container Pods

Using a single container per pod allows for the most granularity and decoupling. There are still some reasons to deploy multiple containers, sometimes called `composite containers`, in a single pod. The secondary containers can handle logging or enhance the primary, the `sidecar` concept, or acting as a proxy to the outside, the `ambassador` concept, or modifying data to meet an external format such as an `adapter`. All three concepts are secondary containers to perform a function the primary container does not.

1. We will add a second container to the pod to handle logging. Without going into details of how to use **fluentd** we will add a logging container to the exiting pod, which would act as a `sidecar`. At this state we will just add the second container and verify it is running. In the **Deployment Configuration** chapter we will continue to work on this pod by adding persistent storage and configure **fluentd** via a `configMap`.

   Edit the YAML file and add a **fluentd** container. The dash should line up with the previous container dash. At this point a name and image should be enough to start the second container.

```
student@ckad-1:~$ vim basic.yaml
....
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
  - name: fdlogger
    image: k8s.gcr.io/fluentd-gcp:1.30
```

2. Delete and create the pod again. The commands can be typed on a single line, separated by a semicolon. The backslash is only used to fit on the page. This time you should see `2/2` under the `READY` column. You should also find information on the **fluentd** container inside of the **kubectl describe** output.

```
student@ckad-1:~$ kubectl delete pod basicpod ; \
  kubectl create -f lfd259/basic.yaml

pod "basicpod" deleted
pod/basicpod created

student@ckad-1:~$ kubectl get pod
NAME       READY   STATUS    RESTARTS   AGE
basicpod   2/2     Running   0          2m8s

student@ckad-1:~$ kubectl describe pod basicpod
Name:               basicpod
Namespace:          default
Priority:           0
PriorityClassName:  <none>
<output_omitted>
```

3. For now shut down the pod. We will use it again in a future exercise.

```
student@ckad-1:~$ kubectl delete pod basicpod
pod "basicpod" deleted
```

Create a Simple Deployment

Creating a pod does not take advantage of orchestration abilities of Kubernetes. We will now create a `Deployment` which gives use scalability reliability and updates.

1. Now run a containerized webserver **nginx**. Use **kubectl create** to create a simple, single replica deployment running the nginx web server. It will create a single pod as we did previously but with new controllers to ensure it runs as well as other features.

```
student@ckad-1:~$  kubectl create deployment firstpod --image=nginx
deployment.apps/firstpod created
```

2. Verify the new deployment exists and the desired number of pods matches the current number. Using a comma, you can request two resource types at once. The **Tab** key can be helpful. Type enough of the word to be unique and press the **Tab** key, it should complete the word. The deployment should show a number `1` for each value, such that the desired number of pods matches the up-to-date and running number. The pod should show zero restarts.

```
student@ckad-1:~$ kubectl get deployment,pod
NAME                           DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
deployment.extensions/firstpod 1       1       1          1         2m42s

NAME                           READY STATUS  RESTARTS AGE
pod/firstpod-7d88d7b6cf-lrsbk  1/1   Running 0        2m42s
```

3. View the details of the deployment, then the pod. Work through the output slowly. Knowing what a healthy deployment and looks like can be helpful when troubleshooting issues. Again the **Tab** key can be helpful when using long auto-generated object names. You should be able to type firstpod**Tab** and the name will complete when viewing the pod.

```
student@ckad-1:~$ kubectl describe deployment firstpod
Name:                 firstpod
Namespace:            default
CreationTimestamp:    Fri, 25 Jul 2018 16:46:57 +0000
Labels:               run=firstpod
Annotations:          deployment.kubernetes.io/revision=1
Selector:             run=firstpod
Replicas:             1 desired | 1 updated | 1 total | 1 available....
StrategyType:         RollingUpdate
MinReadySeconds:      0
<output_omitted>

student@ckad-1:~$ kubectl describe pod firstpod-6bb4574d94-rqk76
Name:                firstpod-6bb4574d94-rqk76
Namespace:           default
Priority:            0
PriorityClassName:   <none>
Node:                ckad-1/10.128.0.2
Start Time:          Wed, 25 Jul 2018 06:13:18 +0000
Labels:              pod-template-hash=2660130850
                     run=firstpod
Annotations:         <none>
Status:              Running
IP:                  192.168.200.65
Controlled By:       ReplicaSet/firstpod-6bb4574d94

<output_omitted>
```

4. Note that the resources are in the default namespace. Get a list of available namespaces.

```
student@ckad-1:~$ kubectl get namespaces
NAME             STATUS    AGE
default          Active    20m
kube-public      Active     20m
kube-system      Active     20m
```

5. There are two other namespaces. Look at the pods in the kube-system namespace.

```
student@ckad-1:~$ kubectl get pod -n kube-system
NAME                                    READY   STATUS  RESTARTS AGE
calico-etcd-rvrpk                       1/1     Running 1        20m
calico-kube-controllers-d554689d5-lm687 1/1     Running 1        20m
calico-node-2ck9g                       2/2     Running 4        19m
calico-node-kkxvl                       2/2     Running 3        20m
etcd-ckad-1                             1/1     Running 1        20m
<output_omitted>
```

6. Now look at the pods in a namespace that does not exist. Note you do not receive an error.

```
student@ckad-1:~$ kubectl get pod -n fakenamespace
No resources found.
```

7. You can also view resources in all namespaces at once. Use the **–all-namespaces** options to select objects in all namespaces at once.

```
student@ckad-1:~$ kubectl get pod --all-namespaces
NAMESPACE    NAME                                    READY STATUS  RESTARTS AGE
default      firstpod-6bb4574d94-rqk76               1/1   Running 0        5m
kube-system  calico-etcd-z49kx                       1/1   Running 0        31m
kube-system  calico-kube-controllers-d554689d5-pfszw 1/1   Running 0        31m
<output_omitted>
```

8. View several resources at once. Note that most resources have a short name such as rs for ReplicaSet, po for Pod, svc for Service, and ep for endpoint.

```
student@ckad-1:~$ kubectl get deploy,rs,po,svc,ep
NAME                         DESIRED  CURRENT  UP-TO-DATE  AVAILABLE  AGE
deployment.extensions/firstpod 1        1        1           1          4m

NAME                                             DESIRED  CURRENT   READY....
replicaset.extensions/firstpod-6bb4574d94-rqk76   1         1         1 ....

NAME                           READY  STATUS    RESTARTS   AGE
pod/firstpod-6bb4574d94-rqk76 1/1    Running   0          4m

NAME                 TYPE       CLUSTER-IP    EXTERNAL-IP PORT(S)      AGE
service/basicservice NodePort   10.108.147.76 <none>      80:31601/TCP 21m
service/kubernetes   ClusterIP  10.96.0.1     <none>      443/TCP      21m

NAME                   ENDPOINTS         AGE
endpoints/basicservice <none>            21m
endpoints/kubernetes   10.128.0.3:6443   21m
```

9. Delete the ReplicaSet and view the resources again. Note that the age on the ReplicaSet and the pod it controls is now less than a minute. The deployment controller started a new ReplicaSet when we deleted the existing one, which started another pod when the desired configuration did not match the current status.

```
student@ckad-1:~$ kubectl delete rs firstpod-6bb4574d94-rqk76
replicaset.extensions "firstpod-6bb4574d94-rqk76" deleted

student@ckad-1:~$ kubectl get deployment,rs,po,svc,ep
NAME                          DESIRED  CURRENT  UP-TO-DATE AVAILABLE AGE
deployment.extensions/firstpod  1        1        1          1         7m

NAME                                              DESIRED   CURRENT....
replicaset.extensions/firstpod-6bb4574d94-rqk76   1          1     ....

NAME                          READY     STATUS    RESTARTS    AGE
pod/firstpod-7d99ffc75-p9hbw   1/1       Running   0           12s
```

```
NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   24m

NAME                    ENDPOINTS         AGE
endpoints/kubernetes    10.128.0.2:6443   80m
endpoints/basicservice    <none>            21m
```

10. This time delete the top-level controller. After about 30 seconds for everything to shut down you should only see the cluster service and endpoint remain for the cluster and the service we created.

```
student@ckad-1:~$ kubectl delete deployment firstpod
deployment.extensions "firstpod" deleted

student@ckad-1:~$ kubectl get deployment,rs,po,svc,ep
NAME                 TYPE      CLUSTER-IP    EXTERNAL-IP PORT(S)      AGE
service/basicservice NodePort  10.108.147.76 <none>      80:31601/TCP 35m
kubernetes           ClusterIP 10.96.0.1     <none>      443/TCP      24m

NAME         ENDPOINTS         AGE
kubernetes   10.128.0.3:6443   24m
```

11. As we won't need it for a while, delete the `basicservice` service as well.

```
student@ckad-1:~$ kubectl delete svc basicservice
service "basicservice" deleted
```

Chapter 3

Build

3.1 Labs

Exercise 3.1: Deploy a New Application

Overview

In this lab we will deploy a very simple python application, test it using Docker, ingest it into Kubernetes and configure probes to ensure it continues to run. This lab requires the completion of the previous lab, the installation and configuration of a Kubernetes cluster.

Working with Python

1. Install python on your master node. It may already be installed, as is shown in the output below.

```
student@ckad-1:~$ sudo apt-get -y install python
Reading package lists... Done
Building dependency tree
Reading state information... Done
python is already the newest version (2.7.12-1~16.04).
python set to manually installed.
0 upgraded, 0 newly installed, 0 to remove and 5 not upgraded.
student@ckad-1:~$
```

2. Locate the python binary on your system.

```
student@ckad-1:~$ which python
/usr/bin/python
```

3. Create and change into a new directory. The Docker build process pulls everything from the current directory into the image file by default. Make sure the chosen directory is empty.

```
student@ckad-1:~$ mkdir app1
```

```
student@ckad-1:~$ cd app1

student@ckad-1:~/app1$ ls -l
total 0
```

4. Create a simple python script which prints the time and hostname every 5 seconds. There are six commented parts to this script, which should explain what each part is meant to do. The script is included with others in the course tar file, though you are encouraged to create the file by hand if not already familiar with the process.

```
student@ckad-1:~/app1$ vim simple.py
#!/usr/bin/python
## Import the necessary modules
import time
import socket

## Use an ongoing while loop to generate output
while True :

## Set the hostname and the current date
  host = socket.gethostname()
  date = time.strftime("%Y-%m-%d %H:%M:%S")

## Convert the date output to a string
  now = str(date)

## Open the file named date in append mode
## Append the output of hostname and time
  f = open("date.out", "a" )
  f.write(now + "\n")
  f.write(host + "\n")
  f.close()

## Sleep for five seconds then continue the loop
  time.sleep(5)
```

5. Make the file executable and test that it works. Use **ctrl-c** to interrupt the while loop after 20 or 30 seconds. The output will be sent to a newly created file in your current directory called `date.out`.

```
student@ckad-1:~/app1$ chmod +x simple.py

student@ckad-1:~/app1$ ./simple.py
^CTraceback (most recent call last):
  File "./simple.py", line 42, in <module>
    time.sleep(5)
KeyboardInterrupt
```

6. View the `date.out` file. It should contain the hostname and timedate stamps.

```
student@ckad-1:~/app1$ cat date.out
2018-03-22 15:51:38
ckad-1
2018-03-22 15:51:43
ckad-1
2018-03-22 15:51:48
ckad-1
<output_omitted>
```

7. Create a `Dockerfile`. Note the name is important, it cannot have a suffix. We will use three statements, `FROM` to declare which version of Python to use, `ADD` to include our script and `CMD` to indicate the action of the container. Should you be including more complex tasks you may need to install extra libraries, shown commented out as `RUN pip install` in the following example.

```
student@ckad-1:~/app1$ vim Dockerfile
FROM python:2
ADD simple.py /
## RUN pip install pystrich
CMD [ "python", "./simple.py" ]
```

8. Build the container. The output below shows mid-build as necessary software is downloaded. You will need to use **sudo** in order to run this command. After the three step process completes the last line of output should indicate success.

```
student@ckad-1:~/app1$ sudo docker build -t simpleapp .
Sending build context to Docker daemon 3.072 kB
Step 1/3 : FROM python:2
2: Pulling from library/python
4176fe04cefe: Pull complete
851356ecf618: Pull complete
6115379c7b49: Pull complete
aaf7d781d601: Extracting [================>         ] 54.03 MB/135 MB
40cf661a3cc4: Download complete
c582f0b73e63: Download complete
6c1ea8f72a0d: Download complete
7051a41ae6b7: Download complete
<output_omitted>
Successfully built c4e0679b9c36
```

9. Verify you can see the new image among others downloaded during the build process, installed to support the cluster, or you may have already worked with. The newly created `simpleapp` image should be listed first.

```
student@ckad-1:~/app1$ sudo docker images
REPOSITORY          TAG      IMAGE ID         CREATED        SIZE
simpleapp           latest   c4e0679b9c36     2 minutes ago  681 MB
quay.io/calico/node v2.6.8   e96a297310fd     13 days ago    282 MB
python              2        d8690ef56706     2 weeks ago    681 MB
<output_omitted>
```

10. Use docker to run a container using the new image. While the script is running you wont see any output and the shell will be occupied running the image in the background. After 30 seconds use **ctrl-c** to interrupt. The local `date.out` file will not be updated with new times, instead that output will be a file of the container image.

```
student@ckad-1:~$ sudo docker run simpleapp
^CTraceback (most recent call last):
  File "./simple.py", line 24, in <module>
    time.sleep(5)
KeyboardInterrupt
```

11. Locate the newly created `date.out` file. The following command should show two files of this name, the one created when we ran simple.py and another under `/var/lib/docker` when run via a Docker container.

```
student@ckad-1:~/app1$ sudo find / -name date.out
/home/student/app1/date.out
/var/lib/docker/aufs/diff/ee814320c900bd24fad0c5db4a258d3c2b78a19cde
629d7de7d27270d6a0c1f5/date.out
```

12. View the contents of the `date.out` file created via Docker. Note the need for **sudo** as Docker created the file this time, and the owner is `root`. The long name is shown on several lines in the example, but would be a single line when typed or copied.

```
student@ckad-1:~/app1$ sudo tail \
/var/lib/docker/aufs/diff/ee814320c900bd24fa\
d0c5db4a258d3c2b78a19cde629d7de7d27270d6a0c1f5/date.out
2018-03-22 16:13:46
53e1093e5d39
2018-03-22 16:13:51
53e1093e5d39
2018-03-22 16:13:56
53e1093e5d39
```

Configure A Local Docker Repo

While we could create an account and upload our application to hub.docker.com, thus sharing it with the world, we will instead create a local repository and make it available to the nodes of our cluster.

1. Well need to complete a few steps with special permissions, for ease of use well become root using **sudo**.

```
student@ckad-1:~/app1$ cd
student@ckad-1:~$ sudo -i
```

2. Install the **docker-compose** software and utilities to work with the nginx server which will be deployed with the registry.

```
root@ckad-1:~# apt-get install -y docker-compose apache2-utils
<output_omitted>
```

3. Create a new directory for configuration information. Well be placing the repository in the root filesystem. A better location may be chosen in a production environment.

```
root@ckad-1:~# mkdir -p /localdocker/data

root@ckad-1:~# cd /localdocker/
```

4. Create a Docker compose file. Inside is an entry for the nginx web server to handle outside traffic and a registry entry listening to loopback port 5000 for running a local Docker registry.

```
root@ckad-1:/localdocker# vim docker-compose.yaml
nginx:
  image: "nginx:1.12"
  ports:
    - 443:443
  links:
    - registry:registry
  volumes:
    - /localdocker/nginx/:/etc/nginx/conf.d
registry:
  image: registry:2
  ports:
    - 127.0.0.1:5000:5000
  environment:
    REGISTRY_STORAGE_FILESYSTEM_ROOTDIRECTORY: /data
  volumes:
    - /localdocker/data:/data
```

5. Use the **docker-compose up** command to create the containers declared in the previous step YAML file. This will capture the terminal and run until you use **ctrl-c** to interrupt. There should be five registry_1 entries with info messages about memory and which port is being listened to. Once were sure the Docker file works well convert to a Kubernetes tool.

```
root@ckad-1:/localdocker# docker-compose up
Pulling nginx (nginx:1.12)...
1.12: Pulling from library/nginx
2a72cbf407d6: Pull complete
f37cbdc183b2: Pull complete
78b5ad0b466c: Pull complete
Digest: sha256:edad623fc7210111e8803b4359ba4854e101bcca1fe7f46bd1d35781f4034f0c
Status: Downloaded newer image for nginx:1.12
Creating localdocker_registry_1
Creating localdocker_nginx_1
Attaching to localdocker_registry_1, localdocker_nginx_1
registry_1  | time="2018-03-22T18:32:37Z" level=warning msg="No HTTP secret provided - generated ran
<output_omitted>
```

6. Test that you can access the repository. Open a second terminal to the master node. Use the **curl** command to test the repository. It should return {}, but does not have a carriage-return so will be on the same line as the following prompt. You should also see the GET request in the first, captured terminal, without error. Dont forget the trailing slash. Youll see a Moved Permanently message if the path doesnt match exactly.

```
student@ckad-1:~/localdocker$ curl http://127.0.0.1:5000/v2/
{}student@ckad-1:~/localdocker$
```

7. Now that we know **docker-compose** format is working, ingest the file into Kubernetes using **kompose**. Use **ctrl-c** to stop the previous **docker-compose** command.

```
^CGracefully stopping... (press Ctrl+C again to force)
Stopping localdocker_nginx_1 ... done
Stopping localdocker_registry_1 ... done
```

8. Download the kompose binary and make it executable. The command can run on a single line. Note that the option following the dash is the letter as in **o**utput.

```
root@ckad-1:/localdocker# curl -L \
https://github.com/kubernetes/kompose/releases/download/v1.1.0/kompose-linux-amd64 -o kompose
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   609    0   609    0     0   1963      0 --:--:-- --:--:-- --:--:--  1970
100 45.3M  100 45.3M    0     0  16.3M      0  0:00:02  0:00:02 --:--:-- 25.9M

root@ckad-1:/localdocker# chmod +x kompose
```

9. Move the binary to a directory in our $PATH. Then return to your non-root user.

```
root@ckad-1:/localdocker# mv ./kompose /usr/local/bin/kompose

root@ckad-1:/localdocker# exit
```

10. Create two physical volumes in order to deploy a local registry for Kubernetes. 200Mi for each should be enough for each of the volumes. Use the **hostPath** storageclass for the volumes.

    More details on how persistent volumes and persistent volume claims are covered in an upcoming chapter.

```
student@ckad-1:~$ vim vol1.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    type: local
  name: task-pv-volume
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 200Mi
  hostPath:
    path: /tmp/data
  persistentVolumeReclaimPolicy: Retain

student@ckad-1:~$ vim vol2.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  labels:
    type: local
  name: registryvm
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 200Mi
  hostPath:
    path: /tmp/nginx
  persistentVolumeReclaimPolicy: Retain
```

11. Create both volumes.

```
student@ckad-1:~$ kubectl create -f vol1.yaml
persistentvolume/task-pv-volume created

student@ckad-1:~$ kubectl create -f vol2.yaml
persistentvolume/registryvm created
```

12. Verify both volumes have been created. They should show an `Available` status.

```
student@ckad-1:~$ kubectl get pv
NAME            CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS
  CLAIM     STORAGECLASS   REASON    AGE
registryvm       200Mi      RWO            Retain           Available
                                     27s
task-pv-volume   200Mi      RWO            Retain           Available
                                     32s
```

13. Go to the configuration file directory for the local Docker registry.

```
student@ckad-1:~$ cd /localdocker/

student@ckad-1:~/localdocker$ ls
data  docker-compose.yaml  nginx
```

14. Convert the Docker file into a single YAML file for use with Kubernetes. Not all objects convert exactly from Docker to **kompose**, you may get errors about the mount syntax for the new volumes. They can be safely ignored.

```
student@ckad-1:~/localdocker$ sudo kompose convert -f docker-compose.yaml \
-o localregistry.yaml
WARN Volume mount on the host "/localdocker/nginx/" isn't supported - ignoring path on the host
WARN Volume mount on the host "/localdocker/data" isn't supported - ignoring path on the host
```

15. Review the file. Youll find that multiple Kubernetes objects will have been created such as `Services`, `Persistent Volume Claims` and `Deployments` using environmental parameters and volumes to configure the container within.

```
student@ckad-1:/localdocker$ less localregistry.yaml
apiVersion: v1
items:
- apiVersion: v1
  kind: Service
  metadata:
    annotations:
      kompose.cmd: kompose convert -f docker-compose.yaml -o localregistry.yaml
      kompose.version: 1.1.0 (36652f6)
    creationTimestamp: null
    labels:
<output_omitted>
```

16. View the cluster resources prior to deploying the registry. Only the cluster service and two available persistent volumes should exist in the default namespace.

```
student@ckad-1:~/localdocker$ kubectl get pods,svc,pvc,pv,deploy
NAME            TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes    ClusterIP   10.96.0.1        <none>              443/TCP   4h
```

```
NAME                             CAPACITY   ACCESS MODES   RECLAIM POLICY
STATUS      CLAIM      STORAGECLASS   REASON    AGE
persistentvolume/registryvm      200Mi      RWO            Retain
 Available                                  15s
persistentvolume/task-pv-volume 200Mi      RWO            Retain
 Available                                  17s
```

17. Use **kubectl** to create the local docker registry.

```
student@ckad-1:~/localdocker$ kubectl create -f localregistry.yaml
service/nginx created
service/registry created
deployment.extensions/nginx created
persistentvolumeclaim/nginx-claim0 created
deployment.extensions/registry created
persistentvolumeclaim/registry-claim0 created
```

18. View the newly deployed resources. The persistent volumes should now show as `Bound`. Find the `service IP` for the registry. It should be sharing port 5000. In the example below the IP address is 10.110.186.162, yours may be different.

```
student@ckad-1:~/localdocker$ kubectl get pods,svc,pvc,pv,deploy
NAME                            READY     STATUS    RESTARTS   AGE
pod/nginx-6b58d9cdfd-95zxq      1/1       Running   0          1m
pod/registry-795c6c8b8f-b8z4k   1/1       Running   0          1m

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    1h
service/nginx        ClusterIP   10.106.82.218    <none>        443/TCP    1m
service/registry     ClusterIP   10.110.186.162    <none>       5000/TCP   1m

NAME                                    STATUS    VOLUME
 CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/nginx-claim0      Bound     registryvm
 200Mi      RWO                           1m
persistentvolumeclaim/registry-claim0   Bound     task-pv-volume
 200Mi      RWO                           1m

NAME                            CAPACITY   ACCESS MODES   RECLAIM POLICY
   STATUS    CLAIM     STORAGECLASS   REASON    AGE
persistentvolume/registryvm     200Mi      RWO            Retain
   Bound
default/nginx-claim0                                        5m
persistentvolume/task-pv-volume 200Mi      RWO            Retain
  Bound
default/registry-claim0                                     6m

NAME                             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deployment.extensions/nginx      1         1         1            1           1m
deployment.extensions/registry   1         1         1            1           1m
```

19. Verify you get the same {} response using the Kubernetes deployed registry as we did when using **docker-compose**. Note you must use the trailing slash after `v2`. Please also note that if the connection hangs it may be due to a firewall issue. If running your nodes using GCE ensure your instances are using VPC setup and all ports are allowed. If using AWS also make sure all ports are being allowed.

```
student@ckad-1:~/localdocker$ curl http://10.110.186.162:5000/v2/
{}student@ckad-1:~/localdocker$
```

20. Edit the Docker configuration file to allow insecure access to the registry. In a production environment steps should be taken to create and use TLS authentication instead. Use the IP and port of the registry you verified in the previous step.

```
student@ckad-1:~$ sudo vim /etc/docker/daemon.json
{ "insecure-registries":["10.110.186.162:5000"] }
```

21. Restart docker on the local system. It can take up to a minute for the restart to take place.

```
student@ckad-1:~$ sudo systemctl restart docker.service
```

22. Download and tag a typical image from hub.docker.com. Tag the image using the IP and port of the registry. We will also use the latest tag.

```
student@ckad-1:~$ sudo docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
<output_omitted>
Digest: sha256:9ee3b83bcaa383e5e3b657f042f4034c92cdd50c03f73166c145c9ceaea9ba7c
Status: Downloaded newer image for ubuntu:latest


student@ckad-1:~$ sudo docker tag ubuntu:latest 10.110.186.162:5000/tagtest
```

23. Push the newly tagged image to your local registry. If you receive an error about an HTTP request to an HTTPS client check that you edited the /etc/docker/daemon.json file correctly and restarted the service.

```
student@ckad-1:~$ sudo docker push 10.110.186.162:5000/tagtest
The push refers to a repository [10.110.186.162:5000/tagtest]
db584c622b50: Pushed
52a7ea2bb533: Pushed
52f389ea437e: Pushed
88888b9b1b5b: Pushed
a94e0d5a7c40: Pushed
latest: digest: sha256:0847cc7fed1bfafac713b0aa4ddfb8b9199a99092ae1fc4e718cb28e8528f65f size: 1357
```

24. We will test to make sure we can also pull images from our local repository. Begin by removing the local cached images.

```
student@ckad-1:~$ sudo docker image remove ubuntu:latest
Untagged: ubuntu:latest
Untagged: ubuntu@sha256:e348fbbea0e0a0e73ab0370de151e7800684445c509d46195aef73e090a49bd6

student@ckad-1:~$ sudo docker image remove 10.110.186.162:5000/tagtest
Untagged: 10.110.186.162:5000/tagtest:latest
<output_omitted>
```

25. Pull the image from the local registry. It should report the download of a newer image.

```
student@ckad-1:~$ sudo docker pull 10.110.186.162:5000/tagtest
Using default tag: latest
latest: Pulling from tagtest
Digest: sha256:0847cc7fed1bfafac713b0aa4ddfb8b9199a99092ae1fc4e718cb28e8528f65f
Status: Downloaded newer image for 10.110.186.162:5000/tagtest:latest
```

26. Use docker tag to assign the simpleapp image and then push it to the local registry. The image and dependent images should be pushed to the local repository.

```
student@ckad-1:~$ sudo docker tag simpleapp 10.110.186.162:5000/simpleapp

student@ckad-1:~$ sudo docker push 10.110.186.162:5000/simpleapp
The push refers to a repository [10.110.186.162:5000/simpleapp]
321938b97e7e: Pushed
ca82a2274c57: Pushed
de2fbb43bd2a: Pushed
4e32c2de91a6: Pushed
6e1b48dc2ccc: Pushed
ff57bdb79ac8: Pushed
6e5e20cbf4a7: Pushed
86985c679800: Pushed
8fad67424c4e: Pushed
latest: digest: sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a size: 2218
```

27. Configure the worker (second) node to use the local registry running on the master server. Connect to the worker node. Edit the Docker daemon.json file with the same values as the master node and restart the service.

```
student@ckad-2:~$ sudo vim /etc/docker/daemon.json
{ "insecure-registries":["10.110.186.162:5000"] }

student@ckad-2:~$ sudo systemctl restart docker.service
```

28. Pull the recently pushed image from the registry running on the master node.

```
student@ckad-2:~$ sudo docker pull 10.110.186.162:5000/simpleapp
Using default tag: latest
latest: Pulling from simpleapp
f65523718fc5: Pull complete
1d2dd88bf649: Pull complete
c09558828658: Pull complete
0e1d7c9e6c06: Pull complete
c6b6fe164861: Pull complete
45097146116f: Pull complete
f21f8abae4c4: Pull complete
1c39556edcd0: Pull complete
85c79f0780fa: Pull complete
Digest: sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a
Status: Downloaded newer image for 10.110.186.162:5000/simpleapp:latest
```

29. Return to the master node and deploy the `simpleapp` in Kubernetes with several replicas. We will name the deployment `try1`. Scale to have six replicas. Multiple replicas the scheduler should run some containers on each node.

```
student@ckad-1:~$ kubectl create deployment try1 \
 --image=10.110.186.162:5000/simpleapp:latest
deployment.apps/try1 created

student@ckad-1:~$  kubectl scale deployment try1 --replicas=6
deployment.extensions/try1 scaled
```

30. View the running pods. You should see six replicas of `simpleapp` as well as two running the locally hosted image repository.

```
student@ckad-1:~$ kubectl get pods
NAME                        READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-j6jm6      1/1       Running   1          13m
registry-795c6c8b8f-5jnpn   1/1       Running   1          13m
try1-857bdcd888-6klrr       1/1       Running   0          25s
try1-857bdcd888-9pwnp       1/1       Running   0          25s
try1-857bdcd888-9xkth       1/1       Running   0          25s
try1-857bdcd888-tw58z       1/1       Running   0          25s
try1-857bdcd888-xj9lk       1/1       Running   0          25s
try1-857bdcd888-znpm8       1/1       Running   0          25s
```

31. On the `second node` use **sudo docker ps** to verify containers of `simpleapp` are running. The scheduler will try to deploy an equal number to both nodes by default.

```
student@ckad-2:~$ sudo docker ps | grep simple
3ae4668d71d8     10.110.186.162:5000/simpleapp@sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a \
            "python ./simple.py"     48 seconds ago      Up 48 seconds                                     \
             k8s_try1_try1-857bdcd888-9xkth_default_2e94b97e-322a-11e8-af56-42010a800004_0
ef6448764625     10.110.186.162:5000/simpleapp@sha256:67ea3e11570042e70cdcbad684a1e2986f59aaf53703e51725accdf5c70d475a \
            "python ./simple.py"     48 seconds ago      Up 48 seconds                                    \
             k8s_try1_try1-857bdcd888-znpm8_default_2e99f356-322a-11e8-af56-42010a800004_0
```

32. Return to the `master node`. Save the `try1` deployment as YAML. Use the **–export** option to remove unique identifying information. The command is shown on two lines for readability. You can remove the backslash when you run the command.

```
student@ckad-1:~/app1$ cd ~/app1/

student@ckad-1:~/app1$ kubectl get deployment try1 -o yaml \
  --export > simpleapp.yaml
```

33. Double check that use of **–export** removed `creationTimestamp`, `selfLink`, `uid`, `resourceVersion`, and all the `Status` information. In newer versions of Kubernetes it seems to no longer be necessary to remove these values in order to deploy again. Be aware older versions would error if these values were found in the YAML file. For backwards compatibility we will continue to remove these entries.

```
student@ckad-1:~/app1$ vim simpleapp.yaml
<output_omitted>
```

34. Delete and recreate the `try1` deployment using the YAML file. Verify the deployment is running with the expected six replicas.

```
student@ckad-1:~$ kubectl delete deployment try1
deployment.extensions "try1" deleted

student@ckad-1:~/app1$ kubectl create -f simpleapp.yaml
deployment.extensions/try1 created

student@ckad-1:~/app1$ kubectl get deployment
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx     1         1         1            1           17m
registry  1         1         1            1           17m
try1        6         6         6            6           7s
```

Configure Probes

When large datasets need to be loaded or a complex application launched prior to client access, a `readinessProbe` can be used. The pod will not become available to the cluster until a test is met and returns a successful exit code. Both `readinessProbes` and `livenessProbes` use the same syntax and are identical other than the name. Where the `readinessProbe` is checked prior to being ready, then not again, the `livenessProbe` continues to be checked. There are three types of liveness probes: a command returns a zero exit value, meaning success, an HTTP request returns a response code in the 200 to 500 range, and the third probe uses a TCP socket. In this example well use a command, **cat**, which will return a zero exit code when the file `/tmp/healthy` has been created and can be accessed.

1. Edit the YAML deployment file and add the stanza for a `readinessprobe`. Remember that when working with YAML whitespace matters. Indentation is used to parse where information should be associated within the stanza and the entire file. Do not use tabs. If you get an error about validating data, check the indentation. It can also be helpful to paste the file to this website to see how indentation affects the JSON value, which is actually what Kubernetes ingests: `https://www.json2yaml.com/`

```
student@ckad-1:~/app1$ vim simpleapp.yaml
....
    spec:
      containers:
      - image: 10.111.235.60:5000/simpleapp:latest
        imagePullPolicy: Always
        name: simpleapp
        readinessProbe:
          exec:
            command:
            - cat
            - /tmp/healthy
          periodSeconds: 5
        resources: {}
....
```

2. Delete and recreate the `try1` deployment.

```
student@ckad-1:~/app1$ kubectl delete deployment try1
deployment.extensions "try1" deleted
```

```
student@ckad-1:~/app1$ kubectl create -f simpleapp.yaml
deployment.extensions/try1 created
```

3. The new `try1` deployment should reference six pods, but show zero available. They are all missing the `/tmp/healthy` file.

```
student@ckad-1:~/app1$ kubectl get deployment
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx      1         1         1            1           39m
registry   1         1         1            1           39m
try1       6         6         6            0           5s
```

4. Take a closer look at the pods. Choose one of the `try1` pods as a test to create the health check file.

```
student@ckad-1:~/app1$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk        1/1       Running   1          40m
registry-795c6c8b8f-7vwdn     1/1       Running   1          40m
try1-9869bdb88-2wfnr          0/1       Running   0          26s
try1-9869bdb88-6bknl          0/1       Running   0          26s
try1-9869bdb88-786v8          0/1       Running   0          26s
try1-9869bdb88-gmvs4          0/1       Running   0          26s
try1-9869bdb88-lfvlx          0/1       Running   0          26s
try1-9869bdb88-rtchc          0/1       Running   0          26s
```

5. Run the bash shell interactively and touch the `/tmp/healthy` file.

```
student@ckad-1:~/app1$ kubectl exec  -it try1-9869bdb88-rtchc -- /bin/bash

root@try1-9869bdb88-rtchc:/# touch /tmp/healthy

root@try1-9869bdb88-rtchc:/# exit
exit
```

6. Wait at least five seconds, then check the pods again. Once the probe runs again the container should show available quickly. The pod with the existing `/tmp/healthy` file should be running and show `1/1` in a `READY` state. The rest will continue to show `0/1`.

```
student@ckad-1:~/app1$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk        1/1       Running   1          44m
registry-795c6c8b8f-7vwdn     1/1       Running   1          44m
try1-9869bdb88-2wfnr          0/1       Running   0          4m
try1-9869bdb88-6bknl          0/1       Running   0          4m
try1-9869bdb88-786v8          0/1       Running   0          4m
try1-9869bdb88-gmvs4          0/1       Running   0          4m
try1-9869bdb88-lfvlx          0/1       Running   0          4m
try1-9869bdb88-rtchc          1/1       Running   0          4m
```

7. Touch the file in the remaining pods. Consider using a **for** loop, as an easy method to update each pod. Note the >shown in the output represents the secondary prompt, you would not type in that character

```
student@ckad-1:~$ for name in try1-9869bdb88-2wfnr try1-9869bdb88-6bknl \
> try1-9869bdb88-786v8 try1-9869bdb88-gmvs4 try1-9869bdb88-lfvlx
> do
> kubectl exec $name touch /tmp/healthy
> done
```

8. It may take a short while for the probes to check for the file and the health checks to succeed.

```
student@ckad-1:~/app1$ kubectl get pods
NAME                                   READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk       1/1           Running     1                   1h
registry-795c6c8b8f-7vwdn  1/1           Running     1                   1h
try1-9869bdb88-2wfnr          1/1           Running      0                   22m
try1-9869bdb88-6bknl         1/1       Running   0          22m
try1-9869bdb88-786v8         1/1       Running   0          22m
try1-9869bdb88-gmvs4         1/1       Running   0          22m
try1-9869bdb88-lfvlx         1/1       Running   0          22m
try1-9869bdb88-rtchc         1/1       Running   0          22m
```

9. Now that we know when a pod is healthy, we may want to keep track that it stays healthy, using a `livenessProbe`. You could use one probe to determine when a pod becomes available and a second probe, to a different location, to ensure ongoing health.

   Edit the deployment again. Add in a `livenessProbe` section as seen below. This time we will add a `Sidecar` container to the pod running a simple application which will respond to port 8080. Note that the dash (**-**) in front of the name. Also `goproxy` is indented the same number of spaces as the **-** in front of the `image:` line for `simpleapp` earlier in the file. In this example that would be seven spaces

```
student@ckad-1:~/app1$ vim simpleapp.yaml
....
         terminationMessagePath: /dev/termination-log
         terminationMessagePolicy: File
      - name: goproxy
        image: k8s.gcr.io/goproxy:0.1
        ports:
        - containerPort: 8080
        readinessProbe:
          tcpSocket:
            port: 8080
          initialDelaySeconds: 5
          periodSeconds: 10
        livenessProbe:
          tcpSocket:
            port: 8080
          initialDelaySeconds: 15
          periodSeconds: 20
       dnsPolicy: ClusterFirst
       restartPolicy: Always
....
```

10. Delete and recreate the deployment.

```
student@ckad-1:~$ kubectl delete deployment try1
deployment.extensions "try1" deleted

student@ckad-1:~$ kubectl create -f simpleapp.yaml
deployment.extensions/try1 created
```

11. View the newly created pods. Youll note that there are two containers per pod, and only one is running. The new `simpleapp` containers will not have the `/tmp/healthy` file, so they will not become available until we touch the `/tmp/healthy` file again. We could include a command which creates the file into the container arguments. The output below shows it can take a bit for the old pods to terminate.

```
student@ckad-1:~$ kubectl get pods
NAME                         READY     STATUS              RESTARTS   AGE
nginx-6b58d9cdfd-g7lnk       1/1       Running             1          13h
registry-795c6c8b8f-7vwdn    1/1       Running             1          13h
try1-76cc5ffcc6-4rjvh        1/2       Running             0          3s
try1-76cc5ffcc6-bk5f5        1/2       Running             0          3s
try1-76cc5ffcc6-d8n5q        0/2       ContainerCreating   0          3s
try1-76cc5ffcc6-mm6tw        1/2       Running             0          3s
```

```
try1-76cc5ffcc6-r9q5n        1/2       Running              0          3s
try1-76cc5ffcc6-tx4dz        1/2       Running              0          3s
try1-9869bdb88-2wfnr         1/1       Terminating          0          12h
try1-9869bdb88-6bknl         1/1       Terminating          0          12h
try1-9869bdb88-786v8         1/1       Terminating          0          12h
try1-9869bdb88-gmvs4         1/1       Terminating          0          12h
try1-9869bdb88-lfvlx         1/1       Terminating          0          12h
try1-9869bdb88-rtchc         1/1       Terminating          0          12h
```

12. Create the health check file for the `readinessProbe`. You can use a **for** loop again for each action, with updated pod names. As there are now two containers in the pod, you should include the container name for which one will execute the command. If no name is given, it will default to the first container. Depending on how you edited the YAML file `try1` should be the first pod and `goproxy` the second. To ensure the correct container is updated, add **-c simpleapp** to the **kubectl** command. Your pod names will be different. Use the names of the newly started containers from the **kubectl get pods** command output. Note the >character represents the secondary prompt, you would not type in that character.

```
student@ckad-1:~$ for name in try1-76cc5ffcc6-4rjvh \
> try1-76cc5ffcc6-bk5f5 try1-76cc5ffcc6-d8n5q \
> try1-76cc5ffcc6-mm6tw try1-76cc5ffcc6-r9q5n \
> try1-76cc5ffcc6-tx4dz
> do
> kubectl exec $name -c simpleapp touch /tmp/healthy
> done
<output_omitted>
```

13. In the next minute or so the `Sidecar` container in each pod, which was not running, will change status to `Running`. Each should show `2/2` containers running.

```
student@ckad-1:~$ kubectl get pods
NAME                         READY     STATUS            RESTARTS    AGE
nginx-6b58d9cdfd-g7lnk       1/1       Running           1           13h
registry-795c6c8b8f-7vwdn    1/1       Running           1           13h
try1-76cc5ffcc6-4rjvh        2/2       Running           0           3s
try1-76cc5ffcc6-bk5f5        2/2       Running           0           3s
try1-76cc5ffcc6-d8n5q        2/2       Running           0           3s
try1-76cc5ffcc6-mm6tw        2/2       Running           0           3s
try1-76cc5ffcc6-r9q5n        2/2       Running           0           3s
try1-76cc5ffcc6-tx4dz        2/2       Running           0           3s
```

14. View the events for a particular pod. Even though both containers are currently running and the pod is in good shape, note the events section shows the issue.

```
student@ckad-1:~/app1$ kubectl describe pod try1-76cc5ffcc6-tx4dz | tail
  Normal   SuccessfulMountVolume  9m                kubelet, ckad-1-lab-x6dj
MountVolume.SetUp succeeded for volume "default-token-jf69w"
  Normal   Pulling                9m                kubelet, ckad-1-lab-x6dj
pulling image "10.108.143.90:5000/simpleapp"
  Normal   Pulled                 9m                kubelet, ckad-1-lab-x6dj
Successfully pulled image "10.108.143.90:5000/simpleapp"
  Normal   Created                9m                kubelet, ckad-1-lab-x6dj
Created container
  Normal   Started                9m                kubelet, ckad-1-lab-x6dj
Started container
  Normal   Pulling                9m                kubelet, ckad-1-lab-x6dj
pulling image "k8s.gcr.io/goproxy:0.1"
  Normal   Pulled                 9m                kubelet, ckad-1-lab-x6dj
Successfully pulled image "k8s.gcr.io/goproxy:0.1"
  Normal   Created                9m                kubelet, ckad-1-lab-x6dj
Created container
  Normal   Started                9m                kubelet, ckad-1-lab-x6dj
Started container
  Warning  Unhealthy              4m (x60 over 9m)  kubelet, ckad-1-lab-x6dj
Readiness probe failed: cat: /tmp/healthy: No such file or directory
```

15. If you look for the status of each container in the pod, they should show that both are `Running` and ready showing `True`.

```
student@ckad-1:~/app1$ kubectl describe pod try1-76cc5ffcc6-tx4dz | \
grep -E 'State|Ready'
    State:          Running
    Ready:          True
    State:          Running
    Ready:          True
    Ready           True
```

Chapter 4

Design

4.1 Labs

Exercise 4.1: Planning the Deployment

Overview

In this exercise we will investigate common network plugins. Each kubelet agent uses one plugin at a time. Due to complexity the entire cluster uses one plugin which is configured prior to application deployment. Some plugins dont honor security configurations such as network policies. Should you design a deployment which and use a network policy there wouldnt be an error, the policy would have no effect.

While still new, the community is moving towards **Container Network Interface (CNI)** specification `https://github.com/containernetworking/cni`. This provides the most flexibility and features in the fast changing space of container networking. A common alternative is **kubenet**, a basic plugin which relies on the cloud provider to handle routing and cross-node networking. In a previous lab exercise we configured **Project Calico**. Classic and external modes are also possible. Several software defined network projects intended for Kubernetes have been created recently, with new features added regularly.

Evaluate Network Plugins

1. Verify your nodes are using a CNI plugin. Look for options passed to kubelet. You may see other lines including the grep command itself and a shell script running in a container which configures Calico.

```
student@ckad-2-nzjr:~$ ps -ef | grep cni
student  13473 13442  0 22:55 pts/1    00:00:00 grep --color=auto cni
root     14118     1  2 Mar30 ?        02:57:27 /usr/bin/kubelet
--bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf
--kubeconfig=/etc/kubernetes/kubelet.conf
--pod-manifest-path=/etc/kubernetes/manifests --allow-privileged=true
--network-plugin=cni --cni-conf-dir=/etc/cni/net.d --cni-bin-dir=/opt/cni/bin
--cluster-dns=10.96.0.10 --cluster-domain=cluster.local
--authorization-mode=Webhook --client-ca-file=/etc/kubernetes/pki/ca.crt
```

```
--cadvisor-port=0 --rotate-certificates=true --cert-dir=/var/lib/kubelet/pki

root      30591 30570  0 Mar30 ?        00:00:10 /bin/sh /install-cni.sh
```

2. View the details of the `install-cni.sh` script. The script runs in a container, the path to which will be different than the example below. Read through the script to see what it does on our behalf.

```
student@ckad-2-nzjr:~$ sudo find / -name install-cni.sh
/var/lib/docker/aufs/mnt/e95a30499a76e79027502bbb8ee4eeb8464a657e276a4
93249f573f5d86e19b3/install-cni.sh
/var/lib/docker/aufs/diff/a7cd14de39089493b793f135ec965f0f3f79eeec8f9d
78e679d7be1a3bdf3345/install-cni.sh

student@ckad-2-nzjr:~$ sudo less \
   /var/lib/docker/aufs/diff/a7cd14de39089493b793f135ec965f0f3f79eeec8
f9d78e679d7be1a3bdf3345/install-cni.sh
```

3. There are many CNI providers possible. The following list represents some of the more common choices, but it is not exhaustive. With many new plugins being developed there may be another which better serves your needs. Use these websites to answer questions which follow. While we strive to keep the answers accurate, please be aware that this area has a lot of attention and development and changes often.

   Project Calico `https://docs.projectcalico.org/v3.0/introduction/`

   Calico with Canal `https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/canal`

   Weave Works `https://www.weave.works/docs/net/latest/kubernetes/kube-addon`

   Flannel `https://github.com/coreos/flannel`

   Romana `http://romana.io/how/romana_basics/`

   Kube Router `https://www.kube-router.io`

   Kopeio `https://github.com/kopeio/networking`

4. Which of the plugins allow vxlans?

5. Which are layer 2 plugins?

6. Which are layer 3?

7. Which allow network policies?

8. Which can encrypt all TCP and UDP traffic?

Solution 4.1

Plugin Answers

1. Which of the plugins allow vxlans? `Canal, Flannel, Kopeio-networking and Weave Net`

2. Which are layer 2 plugins? `Canal, Flannel, Kopeio-networking and Weave Net`

3. Which are layer 3? `Project Calico, Romana, and Kube Router`

4. Which allow network policies? `Project Calico, Canal, Kube Router, Romana and Weave Net`

5. Which can encrypt all TCP and UDP traffic? `Project Calico, Kopeio, and weave Net`

Multi-container Pod Considerations

Using the information learned from this chapter, consider the following questions:

1. Which deployment method would allow the most flexibility, multiple applications per pod or one per pod?
2. Which deployment method allows for the most granular scalability?
3. Which have the best performance?
4. How many IP addresses are assigned per pod?
5. What are some ways containers can communicate within the same pod?
6. What are some reasons you should have multiple containers per pod?

Solution 4.1

Multi Pod Answers

1. Which deployment method would allow the most flexibility, multiple applications per pod or one per Pod?

   `One per pod`

2. Which deployment method allows for the most granular scalability?

   `One per pod`

3. Which have the best inter-container performance?

   `Multiple per pod.`

4. How many IP addresses are assigned per pod?

   `One`

5. What are some ways containers can communicate within the same pod?

   `IPC, loopback or shared filesystem access.`

6. What are some reasons you should have multiple containers per pod?

   `Lean containers may not have functionality like logging. Able to maintain lean execution but add functionality as necessary, like Ambassadors and Sidecar containers.`

Exercise 4.2: Designing Applications With Duration

While most applications are deployed such that they continue to be available there are some which we may want to run a particular number of times called a `Job`, and others on a regular basis called a `CronJob`

Create A Job

1. Create a job which will run a container which sleeps for three seconds then stops.

```
student@ckad-1:~$ vim job.yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: sleepy
spec:
  template:
    spec:
      containers:
      - name: resting
        image: busybox
        command: ["/bin/sleep"]
        args: ["3"]
      restartPolicy: Never
```

2. Create the job, then verify and view the details. The example shows checking the job three seconds in and then again after it has completed. You may see different output depending on how fast you type.

```
student@ckad-1:~$ kubectl create -f job.yaml
job.batch/sleepy created

student@ckad-1:~$ kubectl get job
NAME     COMPLETIONS   DURATION   AGE
sleepy   0/1           3s         3s

student@ckad-1:~$ kubectl describe jobs.batch sleepy
Name:            sleepy
Namespace:       default
Selector:        controller-uid=24c91245-d0fb-11e8-947a-42010a800002
Labels:          controller-uid=24c91245-d0fb-11e8-947a-42010a800002
                 job-name=sleepy
Annotations:     <none>
Parallelism:     1
Completions:     1
Start Time:      Tue, 16 Oct 2018 04:22:50 +0000
Completed At:    Tue, 16 Oct 2018 04:22:55 +0000
Duration:        5s
Pods Statuses:   0 Running / 1 Succeeded / 0 Failed

student@ckad-1:~$ kubectl get job
NAME     COMPLETIONS   DURATION   AGE
sleepy   1/1           5s         17s
```

3. View the configuration information of the job. There are three parameters we can use to affect how the job runs. Use **-o yaml** to see these parameters. We can see that `backoffLimit`, `completions`, and the `parallelism`. We'll add these parameters next.

```
student@ckad-1:~$ kubectl get jobs.batch sleepy -o yaml

<output_omitted>
  uid: c2c3a80d-d0fc-11e8-947a-42010a800002
spec:
  backoffLimit: 6
  completions: 1
  parallelism: 1
  selector:
    matchLabels:
<output_omitted>
```

4. As the job continues to `AGE` in a completion state, delete the job.

```
student@ckad-1:~$ kubectl delete jobs.batch sleepy
job.batch "sleepy" deleted
```

5. Edit the YAML and add the `completions:` parameter and set it to **5**.

```
student@ckad-1:~$ vim job.yaml
<output_omitted>
metadata:
  name: sleepy
spec:
  completions: 5   #<--Add this line
  template:
    spec:
      containers:
<output_omitted>
```

6. Create the job again. As you view the job note that `COMPLETIONS` begins as zero of **5**.

```
student@ckad-1:~$ kubectl create -f job.yaml
job.batch/sleepy created

student@ckad-1:~$ kubectl get jobs.batch
NAME     COMPLETIONS   DURATION   AGE
sleepy   0/5           5s         5s
```

7. View the pods that running. Again the output may be different depending on the speed of typing.

```
student@ckad-1:~$ kubectl get pods
NAME                         READY   STATUS          RESTARTS    AGE
nginx-67f8fb575f-g4468       1/1     Running         2           2d
registry-56cffc98d6-xlhhf    1/1     Running         1           2d
sleepy-z5tnh                 0/1     Completed       0           8s
sleepy-zd692                 1/1     Running         0           3s
<output_omitted>
```

8. Eventually all the jobs will have completed. Verify then delete the job.

```
student@ckad-1:~$ kubectl get jobs
NAME     COMPLETIONS   DURATION   AGE
sleepy   5/5           26s        10m

student@ckad-1:~$ kubectl delete jobs.batch sleepy
job.batch "sleepy" deleted
```

9. Edit the YAML again. This time add in the `parallelism:` parameter. Set it to **2** such that two pods at a time will be deployed.

```
student@ckad-1:~$ job job.yaml
<output_omitted>
  name: sleepy
spec:
  completions: 5
  parallelism: 2   #<-- Add this line
  template:
    spec:
<output_omitted>
```

10. Create the `job` again. You should see the pods deployed two at a time until all five have completed.

```
student@ckad-1:~$ kubectl get pods
NAME                         READY   STATUS    RESTARTS   AGE
nginx-67f8fb575f-g4468       1/1     Running   2          2d
registry-56cffc98d6-xlhhf    1/1     Running   1          2d
sleepy-8xwpc                 1/1     Running   0          5s
sleepy-xjqnf                 1/1     Running   0          5s
try1-c9cb54f5d-b45gl         2/2     Running   0          8h
<output_omitted>

student@ckad-1:~$ kubectl get jobs
NAME     COMPLETIONS   DURATION   AGE
sleepy   3/5           11s        11s
```

11. Add a parameter which will stop the job after a certain number of seconds. Set the `activeDeadlineSeconds:` to `15`. The job and all pods will end once it runs for 15 seconds.

```
student@ckad-1:~$ vim job.yaml
<output_omitted>
  completions: 5
  parallelism: 2
  activeDeadlineSeconds: 15   #<-- Add this line
  template:
    spec:
      containers:
      - name: resting
        image: busybox
        command: ["/bin/sleep"]
        args: ["3"]
<output_omitted>
```

12. Delete and recreate the job again. It should run for four times then continue to age without further completions.

```
student@ckad-1:~$ kubectl delete jobs.batch sleepy
job.batch "sleepy" deleted

student@ckad-1:~$ kubectl create -f job.yaml
job.batch/sleepy created

student@ckad-1:~$ kubectl get jobs
NAME     COMPLETIONS   DURATION   AGE
sleepy   2/5           6s         6s

student@ckad-1:~$ kubectl get jobs
NAME     COMPLETIONS   DURATION   AGE
sleepy   4/5           16s        16s
```

13. View the `message:` entry in the `Status` section of the object YAML output.

```
student@ckad-1:~$ kubectl get job sleepy -o yaml
<output_omitted>
status:
  conditions:
  - lastProbeTime: 2018-10-16T05:45:14Z
    lastTransitionTime: 2018-10-16T05:45:14Z
    message: Job was active longer than specified deadline
    reason: DeadlineExceeded
    status: "True"
    type: Failed
  failed: 1
  startTime: 2018-10-16T05:44:59Z
  succeeded: 4
```

14. Delete the job.

```
student@ckad-1:~$ kubectl delete jobs.batch sleepy
job.batch "sleepy" deleted
```

Create a CronJob

A `CronJob` creates a watch loop which will create a batch job on your behalf when the time becomes true. We Will use our existing `Job` file to start.

1. Copy the `Job` file to a new file.

```
student@ckad-1:~$ cp job.yaml cronjob.yaml
```

2. Edit the file to look like the annotated file shown below.

```
student@ckad-1:~$ vim cronjob.yaml
apiVersion: batch/v1beta1    #<-- Add beta1 to be v1beta1
kind: CronJob                #<-- Update this line to CronJob
metadata:
  name: sleepy
spec:
  schedule: "*/2 * * * *"    #<-- Add Linux style cronjob syntax
  jobTemplate:               #<-- New jobTemplate and spec move
    spec:
      template:              #<-- This and following lines move
        spec:                #<-- four spaces to the right
          containers:
          - name: resting
            image: busybox
            command: ["/bin/sleep"]
            args: ["3"]
          restartPolicy: Never
```

3. Create the new `CronJob`. View the jobs. It will take two minutes for the `CronJob` to run and generate a new batch `Job`.

```
student@ckad-1:~$ kubectl create -f cronjob.yaml
cronjob.batch/sleepy created

student@ckad-1:~$ kubectl get cronjobs.batch
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        <none>          8s

student@ckad-1:~$ kubectl get job
No resources found.
```

4. After two minutes you should see jobs start to run.

```
student@ckad-1:~$ kubectl get cronjobs.batch
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        21s             2m1s

student@ckad-1:~$ kubectl get jobs.batch
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539722040   1/1           5s         18s

student@ckad-1:~$ kubectl get jobs.batch
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539722040   1/1           5s         5m17s
sleepy-1539722160   1/1           6s         3m17s
sleepy-1539722280   1/1           6s         77s
```

5. Ensure that if the job continues for more than 10 seconds it is terminated. We will first edit the **sleep** command to run for 30 seconds then add the `activeDeadlineSeconds:` entry to the container.

```
student@ckad-1:~$ vim cronjob.yaml
....
  jobTemplate:
    spec:
      template:
        spec:
          activeDeadlineSeconds: 10  #<-- Add this line
          containers:
          - name: resting
....
```

6. Delete and recreate the `CronJob`. It may take a couple of minutes for the batch `Job` to be created and terminate due to the timer.

```
student@ckad-1:~$ kubectl delete cronjobs.batch sleepy
cronjob.batch "sleepy" deleted

student@ckad-1:~$ kubectl create -f cronjob.yaml
cronjob.batch/sleepy created

student@ckad-1:~$ kubectl get jobs
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           61s        61s

student@ckad-1:~$ kubectl get cronjobs.batch
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     1        72s             94s

student@ckad-1:~$ kubectl get jobs
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           75s        75s

student@ckad-1:~$ kubectl get jobs
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           2m19s      2m19s
sleepy-1539723360   0/1           19s        19s

student@ckad-1:~$ kubectl get cronjobs.batch
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     2        31s             2m53s
```

7. Clean up by deleting the `CronJob`.

```
student@ckad-1:~$  kubectl delete cronjobs.batch sleepy
cronjob.batch "sleepy" deleted
```

Chapter 5

Deployment Configuration

5.1 Labs

Exercise 5.1: Configure the Deployment

Overview

In this lab we will add resources to our deployment with further configuration you may need for production. Well also work with updating deployed applications and automation of batch jobs and regular tasks.

Save a copy of your `~/app1/simpleapp.yaml` file, in case you would like to repeat portions of the labs, or you find your file difficult to use due to typos and whitespace issues.

```
student@ckad-1:~$ cp ~/app1/simpleapp.yaml ~/beforeLab5.yaml
```

Secrets and ConfigMap

There are three different ways a **ConfigMap** can ingest data, from a literal value, from a file, or from a directory of files.

1. Create a **ConfigMap** containing primary colors. We will create a series of files to ingest into the **ConfigMap**. First create a directory `primary` and populate it with four files. Then we create a file in our home directory with our favorite color.

   ```
   student@ckad-1:~/app1$ cd
   ```

   ```
   student@ckad-1:~$ mkdir primary
   ```

   ```
   student@ckad-1:~$ echo c > primary/cyan
   ```

   ```
   student@ckad-1:~$ echo m > primary/magenta
   ```

   ```
   student@ckad-1:~$ echo y > primary/yellow
   ```

```
student@ckad-1:~$ echo k > primary/black

student@ckad-1:~$ echo "known as key" >> primary/black

student@ckad-1:~$ echo blue > favorite
```

2. Generate a **configMap** using each of the three methods.

```
student@ckad-1:~$ kubectl create configmap colors \
 --from-literal=text=black \
 --from-file=./favorite \
 --from-file=./primary/
configmap/colors created
```

3. View the newly created **configMap**. Note the way the ingested data is presented.

```
student@ckad-1:~$ kubectl get configmap colors
NAME      DATA      AGE
colors    6         11s

student@ckad-1:~$ kubectl get configmap colors -o yaml
apiVersion: v1
data:
  black: |
    k
    known as key
  cyan: |
    c
  favorite: |
    blue
  magenta: |
    m
  text: black
  yellow: |
    y
kind: ConfigMap
metadata:
  creationTimestamp: 2018-04-05T19:49:59Z
  name: colors
  namespace: default
  resourceVersion: "13491"
  selfLink: /api/v1/namespaces/default/configmaps/colors
  uid: 86457ce3-390a-11e8-ba73-42010a800003
```

4. Update the YAML file of the application to make use of the **configMap** as an environmental parameter. Add the six lines from the `env: line` to `key:favorite`.

```
student@ckad-1:~$ vim app1/simpleapp.yaml
....
    spec:
      containers:
      - image: 10.105.119.236:5000/simpleapp:latest
        env:                                # Add from here
        - name: ilike
          valueFrom:
            configMapKeyRef:
              name: colors
              key: favorite                  # To here
        imagePullPolicy: Always
....
```

5. Delete and re-create the deployment with the new parameters.

```
student@ckad-1-lab-7xtx:~$ kubectl delete deployment try1
deployment.extensions "try1" deleted

student@ckad-1-lab-7xtx:~$ kubectl create -f app1/simpleapp.yaml
deployment.extensions/try1 created
```

6. Even though the `try1` container is not in a ready state, it is running and useful. Use **kubectl exec** to view a variables value. View the pod state then verify you can see the `ilike` value within.

```
student@ckad-1:~$ kubectl get po
<output_omitted>

student@ckad-1:~$ kubectl exec -c try1 -it try1-5db9bc6f85-whxbf -- \
      /bin/bash -c 'echo $ilike'
blue
```

7. Edit the YAML file again, this time adding the third method of using a **configMap**. Edit the file to add three lines. `envFrom` should be indented the same amount as `env` earlier in the file, and `configMapRef` should be indented the same as `configMapKeyRef`.

```
student@ckad-1:~$ vim app1/simpleapp.yaml
....
           configMapKeyRef:
              name: colors
              key: favorite
        envFrom:        #Add this and the following two lines
        - configMapRef:
            name: colors
        imagePullPolicy: Always
....
```

8. Again delete and recreate the deployment. Check the pods restart.

```
student@ckad-1:~$ kubectl delete deployment try1
deployment.extensions "try1" deleted

student@ckad-1:~$ kubectl create -f app1/simpleapp.yaml
deployment.extensions/try1 created

student@ckad-1:~$ kubectl get pods
NAME                        READY   STATUS        RESTARTS   AGE
nginx-6b58d9cdfd-9fnl4      1/1     Running       1          23h
registry-795c6c8b8f-hl5w    1/1     Running       2          23h
try1-d4fbf76fd-46pkb        1/2     Running       0          40s
try1-d4fbf76fd-9kw24        1/2     Running       0          39s
try1-d4fbf76fd-bx9j9        1/2     Running       0          39s
try1-d4fbf76fd-jw8g7        1/2     Running       0          40s
try1-d4fbf76fd-lppl5        1/2     Running       0          39s
try1-d4fbf76fd-xtfd4        1/2     Running       0          40s
```

9. View the settings inside the `try1` container of a pod. The following output is truncated in a few places. Omit the container name to observe the behavior. Also execute a command to see all environmental variables instead of logging into the container first.

```
student@ckad-1:~$ kubectl exec -it try1-d4fbf76fd-46pkb -- /bin/bash -c 'env'
Defaulting container name to try1.
Use 'kubectl describe pod/try1-d4fbf76fd-46pkb -n default' to see all of the containers in this pod.
REGISTRY_PORT_5000_TCP_ADDR=10.105.119.236
HOSTNAME=try1-d4fbf76fd-46pkb
TERM=xterm
yellow=y
<output_omitted>
REGISTRY_SERVICE_HOST=10.105.119.236
```

```
KUBERNETES_SERVICE_PORT=443
REGISTRY_PORT_5000_TCP=tcp://10.105.119.236:5000
KUBERNETES_SERVICE_HOST=10.96.0.1
text=black
REGISTRY_SERVICE_PORT_5000=5000
<output_omitted>
black=k
known as key

<output_omitted>
ilike=blue
<output_omitted>
magenta=m

cyan=c
<output_omitted>
```

10. For greater flexibility and scalability **ConfigMaps** can be created from a YAML file, then deployed and redeployed as necessary. Once ingested into the cluster the data can be retrieved in the same manner as any other object. Create another **configMap**, this time from a YAML file.

```
student@ckad-1:~$ vim car-map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fast-car
  namespace: default
data:
  car.make: Ford
  car.model: Mustang
  car.trim: Shelby

student@ckad-1:~$ kubectl create -f car-map.yaml
configmap/fast-car created
```

11. View the ingested data, note that the output is just as in file created.

```
student@ckad-1:~$ kubectl get configmap fast-car -o yaml
apiVersion: v1
data:
  car.make: Ford
  car.model: Mustang
  car.trim: Shelby
kind: ConfigMap
metadata:
  creationTimestamp: 2018-07-26T16:36:32Z
  name: fast-car
  namespace: default
  resourceVersion: "105700"
  selfLink: /api/v1/namespaces/default/configmaps/fast-car
  uid: aa19f8f3-39b8-11e8-ba73-42010a800003
```

12. Add the **configMap** settings to the `simpleapp.yaml` file as a volume. Both containers in the try1 deployment can access to the same volume, using `volumeMounts` statements. Remember that the volume stanza is of equal depth to the containers stanza, and should probably come after for readability.

```
student@ckad-1:~$ vim app1/simpleapp.yaml
.
    spec:
      containers:
      - image: 10.105.119.236:5000/simpleapp:latest
        volumeMounts:
        - mountPath: /etc/cars
```

```
          name: car-vol
        imagePullPolicy: Always
        name: try1
.
          initialDelaySeconds: 15
          periodSeconds: 20
      volumes:                                #Add this and following four lines
      - configMap:
          defaultMode: 420
          name: fast-car
        name: car-vol
      dnsPolicy: ClusterFirst
      restartPolicy: Always
.
```

13. Delete and recreate the deployment.

```
student@ckad-1:~$ kubectl delete deployment try1
deployment.extensions "try1" deleted

student@ckad-1:~$ kubectl create -f app1/simpleapp.yaml
deployment.extensions/try1 created
```

14. Verify the deployment is running. Note that we still have not automated the creation of the `/tmp/healthy` file inside the container, as a result the `AVAILABLE` count remains zero until we use the **for** loop to create the file. We will remedy this in the next step.

```
student@ckad-1:~$ kubectl get deployment
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx      1         1         1            1           1d
registry   1         1         1            1           1d
try1       6         6         6            0           39s
```

15. Our health check was the successful execution of a command. We will edit the command of the existing `readinessProbe` to check for the existence of the mounted configMap file and re-create the deployment. After a minute both containers should become available for each pod in the deployment.

```
student@ckad-1:~$ kubectl delete deployment try1
deployment.extensions "try1" deleted

student@ckad-1:~$ vim app1/simpleapp.yaml
....
        readinessProbe:
          exec:
            command:
            - ls                          #Add/Edit this and following line.
            - /etc/cars
          periodSeconds: 5
....

student@ckad-1:~$ kubectl create -f app1/simpleapp.yaml
deployment.extensions/try1 created
```

16. Wait about a minute and view the deployment and pods. All six replicas should be running and report that `2/2` containers are in a ready state within.

```
student@ckad-1:~$ kubectl get deployment
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx      1         1         1            1           1d
registry   1         1         1            1           1d
try1       6         6         6            6           1m

student@ckad-1:~$ kubectl get pods
```

```
NAME                          READY     STATUS    RESTARTS   AGE
nginx-6b58d9cdfd-9fnl4        1/1       Running   1          1d
registry-795c6c8b8f-hl5wf     1/1       Running   2          1d
try1-7865dcb948-2dzc8         2/2       Running   0          1m
try1-7865dcb948-7fkh7         2/2       Running   0          1m
try1-7865dcb948-d85bc         2/2       Running   0          1m
try1-7865dcb948-djrcj         2/2       Running   0          1m
try1-7865dcb948-kwlv8         2/2       Running   0          1m
try1-7865dcb948-stb2n         2/2       Running   0          1m
```

17. View a file within the new volume mounted in a container. It should match the data we created inside the configMap. Because the file did not have a carriage-return it will appear prior to the following prompt.

```
student@ckad-1:~$ kubectl exec -c try1  -it try1-7865dcb948-stb2n \
    -- /bin/bash -c 'cat /etc/cars/car.trim'
Shelbystudent@ckad-1:~$
```

Attaching Storage

There are several types of storage which can be accessed with Kubernetes, with flexibility of storage being essential to scalability. In this exercise we will configure an NFS server. With the NFS server we will create a new **persistent volume (pv)** and a **persistent volume claim (pvc)** to use it.

1. Use the `CreateNFS.sh` script from the tarball to set up NFS on your master node. This script will configure the server, export `/opt/sfw` and create a file `/opt/sfw/hello.txt`.

```
student@ckad-1:~$ bash lfd259/CreateNFS.sh
Hit:1 http://us-central1.gce.archive.ubuntu.com/ubuntu xenial InRelease
Get:2 http://us-central1.gce.archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]

<output_omitted>

Should be ready. Test here and second node

Export list for localhost:
/opt/sfw *
```

2. Test by mounting the resource from your **second node**. Begin by installing the client software.

```
student@ckad-2:~$ sudo apt-get -y install nfs-common nfs-kernel-server
<output_omitted>
```

3. Test you can see the exported directory using **showmount** from you second node.

```
student@ckad-2:~$ showmount -e ckad-1   ## First nodes name or IP
Export list for ckad-1:
/opt/sfw *
```

4. Mount the directory. Be aware that unless you edit `/etc/fstab` this is not a persistent mount. Change out the node name for that of your master node.

```
student@ckad-2:~$ sudo mount ckad-1:/opt/sfw /mnt
```

5. Verify the `hello.txt` file created by the script can be viewed.

```
student@ckad-2:~$ ls -l /mnt
total 4
-rw-r--r-- 1 root root 9 Sep 28 17:55 hello.txt
```

6. Return to the master node and create a YAML file for an object with kind **PersistentVolume**. The included example file needs an edit to the server parameter. Use the hostname of the master server and the directory you created in the previous step. Only syntax is checked, an incorrect name or directory will not generate an error, but a Pod using the incorrect resource will not start. Note that the `accessModes` do not currently affect actual access and are typically used as labels instead.

```
student@ckad-1:~$ cd lfd259; vim PVol.yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvvol-1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /opt/sfw
    server: ckad-1   #<-- Edit to match your master node name
    readOnly: false
```

7. Create and verify you have a new 1Gi volume named **pvvol-1**. Note the status shows as `Available`. Remember we made two persistent volumes for the image registry earlier.

```
student@ckad-1:~/lfd259$ kubectl create -f PVol.yaml
persistentvolume/pvvol-1 created

student@ckad-1:~/lfd259$ kubectl get pv
NAME             CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM    STORAGECLASS  REASON  AGE
pvvol-1          1Gi        RWX            Retain           Available                                  4s
registryvm       200Mi      RWO            Retain           Bound       default/nginx-claim0           4d
task-pv-volume   200Mi      RWO            Retain           Bound       default/registry-claim0        4d
```

8. Now that we have a new volume we will use a **persistent volume claim (pvc)** to use it in a Pod. We should have two existing claims from our local registry.

```
student@ckad-1:~/lfd259$ kubectl get pvc
NAME              STATUS   VOLUME           CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nginx-claim0      Bound    registryvm       200Mi      RWO                           4d
registry-claim0   Bound    task-pv-volume   200Mi      RWO                           4d
```

9. Create a yaml file with the kind **PersistentVolumeClaim**.

```
student@ckad-1:~/lfd259$ vim pvc.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-one
spec:
  accessModes:
  - ReadWriteMany
  resources:
     requests:
       storage: 200Mi
```

10. Create and verify the new pvc status is `bound`. Note the size is `1Gi`, even though `200Mi` was suggested. Only a volume of at least that size could be used, the first volume with found with at least that much space was chosen.

```
student@ckad-1:~/lfd259$ kubectl create -f pvc.yaml
persistentvolumeclaim/pvc-one created

student@ckad-1:~/lfd259$ kubectl get pvc
```

```
NAME           STATUS    VOLUME        CAPACITY   ACCESS MODES   STORAGECLASS   AGE
nginx-claim0       Bound     registryvm        200Mi      RWO                           4d
pvc-one            Bound     pvvol-1           1Gi        RWX                           4s
registry-claim0    Bound     task-pv-volume    200Mi      RWO                           4d
```

11. Now look at the status of the physical volume. It should also show as bound.

```
student@ckad-1:~/lfd259$ kubectl get pv ; cd
NAME           CAPACITY ACCESS MODES RECLAIM POLICY STATUS
 CLAIM      STORAGECLASS   REASON    AGE
pvvol-1        1Gi      RWX          Retain         Bound
 default/pvc-one                     14m
registryvm     200Mi    RWO          Retain         Bound
 default/nginx-claim0                4d
task-pv-volume 200Mi    RWO          Retain         Bound
 default/registry-claim0             4d
```

12. Edit the simpleapp.yaml file to include two new sections. One section for the container while will use the volume mount point, you should have an existing entry for car-vol. The other section adds a volume to the deployment in general, which you can put after the configMap volume section.

```
student@ckad-1:~$ vim app1/simpleapp.yaml
<output_omitted>
        volumeMounts:
        - mountPath: /etc/cars
          name: car-vol
        - name: nfs-vol          ## Add these two lines
          mountPath: /opt        ##

<output_omitted>
     volumes:
      - configMap:
          defaultMode: 420
          name: fast-car
        name: car-vol
      - name: nfs-vol            ## Add these three lines
        persistentVolumeClaim:   ##
          claimName: pvc-one     ##
<output_omitted>
```

13. Delete and re-create the deployment.

```
student@ckad-1:~/app1$ kubectl delete deployment try1 ; kubectl create -f \
 simpleapp.yaml
deployment.extensions "try1" deleted
deployment.extensions/try1 created
```

14. View the details any of the pods in the deployment, you should see nfs-vol mounted under /opt. The use to command line completion with the **tab** key can be helpful for using a pod name.

```
student@ckad-1:~/app1$ kubectl describe pod try1-594fbb5fc7-5k7sj
<output_omitted>
    Mounts:
      /etc/cars from car-vol (rw)
      /opt from nfs-vol (rw)
<output_omitted>
```

Using ConfigMaps Configure Ambassador Containers

In an earlier lab we added a second Ambassador container to handle logging. Now that we have learned about using ConfigMaps and attaching storage we will use configure our basic pod.

1. Review the YAML for our earlier simple pod. Recall that we added an Ambassador style logging container to the pod but had not fully configured the logging.

```
student@ckad-1:~$ cat basic.yaml
<output_omitted>
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
  - name: fdlogger
    image: k8s.gcr.io/fluentd-gcp:1.30
```

2. Let us begin by adding shared storage to each container. We will use the `hostPath` storage class to provide the `PV` and `PVC`. First we create the directory.

```
student@ckad-1:~$ sudo mkdir /tmp/weblog
```

3. Now we create a new `PV` to use that directory for the `hostPath` storage class. We will use the `storageClassName` of manual so that only `PVC`s which use that name will bind the resource.

```
student@ckad-1:~$ vim weblog-pv.yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: weblog-pv-volume
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 100Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/weblog"
```

4. Create and verify the new `PV` exists.

```
student@ckad-1:~$ kubectl create -f weblog-pv.yaml
persistentvolume/weblog-pv-volume created

student@ckad-1:~$ kubectl get pv weblog-pv-volume
NAME             CAPACITY ACCESS MODES  RECLAIM POLICY
  STATUS        CLAIM   STORAGECLASS   REASON   AGE

weblog-pv-volume 100Mi    RWO           Retain
  Available             manual                  21s
```

5. Next we will create a `PVC` to use the `PV` we just created.

```
student@ckad-1:~$ vim weblog-pvc.yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: weblog-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

6. Create the PVC and verify it shows as Bound to the the PV we previously created.

```
student@ckad-1:~$ kubectl create -f lfd259/weblog-pvc.yaml
persistentvolumeclaim/weblog-pv-claim created

student@ckad-1:~$ kubectl get pvc weblog-pv-claim
NAME             STATUS   VOLUME           CAPACITY ACCESS MODES
   STORAGECLASS    AGE
weblog-pv-claim  Bound    weblog-pv-volume 100Mi    RWO
   manual          79s
```

7. We are ready to add the storage to our pod. We will edit three sections. The first will declare the storage to the pod in general, then two more sections which tell each container where to make the volume available.

```
student@ckad-1:~$
apiVersion: v1
kind: Pod
metadata:
  name: basicpod
  labels:
    type: webserver
spec:
  volumes:                         #<--Same depth as containers
    - name: weblog-pv-storage      #
      persistentVolumeClaim:       #
        claimName: weblog-pv-claim # name from kubectl get output
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:                    #<--Same depth as ports
      - mountPath: "/var/log/nginx/" #
        name: weblog-pv-storage      # Must match volume name above
  - name: fdlogger
    image: k8s.gcr.io/fluentd-gcp:1.30
    volumeMounts:                    #<--Same depth as image:
      - mountPath: "/var/log"        # We will configure this soon
        name: weblog-pv-storage      # Must match volume name above
```

8. At this point we can create the pod again. When we create a shell we will find that the access.log for **nginx** is no longer a symbolic link pointing to stdout it is a writable, zero length file. Leave a **tailf** of the log file running. There won't be any output until you access the webserver in a following command.

```
student@ckad-1:~$ kubectl create -f lfd259/basic.yaml
pod/basicpod created

student@ckad-1:~$ kubectl exec -it basicpod -- /bin/bash
Defaulting container name to webcont.
Use 'kubectl describe pod/basicpod -n default' to see all
of the containers in this pod.

root@basicpod:/#  ls -l /var/log/nginx/access.log
-rw-r--r-- 1 root root 0 Oct 18 16:12 /var/log/nginx/access.log

root@basicpod:/# tailf /var/log/nginx/access.log
```

9. Open a second connection to your node. We will use the pod IP as we have not yet configured a service to expose the pod.

```
student@ckad-1:~$ kubectl get pods -o wide
NAME     READY STATUS   RESTARTS  AGE    IP               NODE
```

```
   NOMINATED NODE
basicpod 2/2   Running  0         3m26s  192.168.213.181  ckad-1
   <none>
```

10. Use **curl** to view the welcome page of the webserver. When the command completes you should see a new entry added to the log. Right after the `GET` we see a `200` response indicating success.

```
student@ckad-1:~$ curl http://192.168.213.181
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>


<New entry in tailf output:>
192.168.32.128 - - [18/Oct/2018:16:16:21 +0000] "GET / HTTP/1.1"
 200 612 "-" "curl/7.47.0" "-"
```

11. Now that we know the `webcont` container is writing to the `PV` we will configure the logger to use that directory as a source. For greater flexibility we will configure **fluentd** using a `configMap`. The details of the data settings can be found in **fluentd** documentation here: `https://docs.fluentd.org/v1.0/categories/config-file`

```
student@ckad-1:~$ vim weblog-configmap.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: fluentd-config
data:
  fluentd.conf: |
    <source>
      @type tail
      format none
      path /var/log/nginx/access.log
      tag count.format1
    </source>

    <match *.**>
      @type forward

      <server>
        name localhost
        host 127.0.0.1
      </server>
    </match>
```

12. Create the new configMap.

```
student@ckad-1:~$ kubectl create -f weblog-configmap.yaml
configmap/fluentd-config created
```

13. Now we will edit the pod yaml file so that the **fluentd** container will mount the configmap as a volume and reference the variables inside the config file. You will add three areas, the volume declaration to the pod, the `env` parameter and the mounting of the volume to the fluentd container

```
student@ckad-1:~$ vim basic.yaml
....
  volumes:
    - name: weblog-pv-storage
      persistentVolumeClaim:
        claimName: weblog-pv-claim
    - name: log-config              #<--Same depth as containers
      configMap:                    #
```

```
        name: fluentd-config        # Must match existing configMap
....
    image: k8s.gcr.io/fluentd-gcp:1.30
    env:                                        #Same depth as image:
    - name: FLUENTD_ARGS                        #
      value: -c /etc/fluentd-config/fluentd.conf #
....
    volumeMounts:
      - mountPath: "/var/log"
        name: weblog-pv-storage
      - name: log-config                #Must match volume name above
        mountPath: "/etc/fluentd-config" #Directory for env variable
```

14. At this point we can delete and re-create the pod. If we had a listening agent running on aggregator.example.com we would see new access message.

```
student@ckad-1:~$ kubectl delete pod basicpod
pod "basicpod" deleted

student@ckad-1:~$ kubectl create -f basic.yaml
pod/basicpod created

student@ckad-1:~$ kubectl get pod basicpod
```

15. Take a log at the logs for both containers. You should see some output for the fdlogger but not for webcont.

Rolling Updates and Rollbacks

When we started working with simpleapp we used a **Docker** tag called latest. While this is the default tag when pulling an image, and commonly used, it remains just a string, it may not be the actual latest version of the image.

1. Make a slight change to our source and create a new image. We will use updates and rollbacks with our application. Adding a comment to the last line should be enough for a new image to be generated.

```
student@ckad-1:~$ cd ~/app1
student@ckad-1:~/app1$ vim simple.py
<output_omitted>
## Sleep for five seconds then continue the loop

  time.sleep(5)

## Adding a new comment so image is different.
```

2. Build the image again. A new container and image will be created. Verify when successful. There should be a different image ID and a recent creation time.

```
student@ckad-1:~/app1$ sudo docker build -t simpleapp .
Sending build context to Docker daemon 7.168 kB
Step 1/3 : FROM python:2
 ---> 2863c80c418c
Step 2/3 : ADD simple.py /
 ---> cde8ecf8492b
Removing intermediate container 3e908b76b5b4
Step 3/3 : CMD python ./simple.py
 ---> Running in 354620c97bf5
 ---> cc6bba0ea213
Removing intermediate container 354620c97bf5
Successfully built cc6bba0ea213
```

```
student@ckad-1:~/app1$ sudo docker images
REPOSITORY                              TAG
 IMAGE ID            CREATED        SIZE
simpleapp                               latest
 cc6bba0ea213    8 seconds ago     679 MB
10.105.119.236:5000/simpleapp           latest
15b5ad19d313     4 days ago        679 MB
<output_omitted>

   \item
   Tag and push the updated image to your locally hosted
   registry. A reminder your IP address will be different
   than the example below. Use the tag \verb?v2? this time
   instead of \verb?latest?.
   \begin{raw}
student@ckad-1:~/app1$ sudo docker tag simpleapp \
   10.105.119.236:5000/simpleapp:v2

student@ckad-1:~/app1$ sudo docker push 10.105.119.236:5000/simpleapp:v2
The push refers to a repository [10.105.119.236:5000/simpleapp]
d6153c8cc7c3: Pushed
ca82a2274c57: Layer already exists
de2fbb43bd2a: Layer already exists
4e32c2de91a6: Layer already exists
6e1b48dc2ccc: Layer already exists
ff57bdb79ac8: Layer already exists
6e5e20cbf4a7: Layer already exists
86985c679800: Layer already exists
8fad67424c4e: Layer already exists
v2: digest: sha256:6cf74051d09463d89f1531fceb9c44cbf99006f8d9b407
dd91d8f07baeee7e9c size: 2218
```

3. Connect to a terminal running on your second node. Pull the `latest` image, then pull `v2`. Note the latest did not pull the new version of the image. Again, remember to use the IP for your locally hosted registry. Youll note the digest is different.

```
student@ckad-2:~$ sudo docker pull 10.105.119.236:5000/simpleapp
Using default tag: latest
latest: Pulling from simpleapp
Digest: sha256:cefa3305c36101d32399baf0919d3482ae8a53c926688be33
86f9bbc04e490a5
Status: Image is up to date for 10.105.119.236:5000/simpleapp:latest

student@ckad-2:~$ sudo docker pull 10.105.119.236:5000/simpleapp:v2
v2: Pulling from simpleapp
f65523718fc5: Already exists
1d2dd88bf649: Already exists
c09558828658: Already exists
0e1d7c9e6c06: Already exists
c6b6fe164861: Already exists
45097146116f: Already exists
f21f8abae4c4: Already exists
1c39556edcd0: Already exists
fa67749bf47d: Pull complete
Digest: sha256:6cf74051d09463d89f1531fceb9c44cbf99006f8d9b407dd91d8
f07baeee7e9c
Status: Downloaded newer image for 10.105.119.236:5000/simpleapp:v2
```

4. Use **kubectl edit** to update the image for the `try1` deployment to use `v2`. As we are only changing one parameter we could also use the **kubectl set** command. Note that the configuration file has not been updated, so a delete or a replace command would not include the new version. It can take the pods up to a minute to delete and to recreate each pod in sequence.

```
student@ckad-1:~/app1$ kubectl edit deployment try1
<output_omitted>
      containers:
      - image: 10.105.119.236:5000/simpleapp:v2
        imagePullPolicy: Always
<output_omitted>
```

5. Verify each of the pods has been recreated and is using the new version of the image. Note some messages will show the scaling down of the old **replicaset**, others should show the scaling up using the new image.

```
student@ckad-1:~/app1$ kubectl get events
LAST SEEN   FIRST SEEN   COUNT     NAME
KIND         SUBOBJECT      TYPE      REASON        SOURCE     MESSAGE
4s           4s           1         try1-594fbb5fc7-nxhfx.152422073b7084da
Pod          spec.containers{goproxy}   Normal    Killing
 kubelet, ckad-2-wdrq
   Killing container with id docker://goproxy:Need to kill Pod
<output_omitted>
2m           2m           1         try1.1524220c35a0d0fb
Deployment     Normal    ScalingReplicaSet
deployment-controller   Scaled up replica set try1-895fccfb to 5
2m           2m           3         try1.1524220e0d69a94a
   Deployment
         Normal    ScalingReplicaSet       deployment-controller
 (combined from similar events):
              Scaled down replica set try1-594fbb5fc7 to 0
```

6. View the images of a Pod in the deployment. Narrow the output to just view the images. The `goproxy` remains unchanged, but the `simpleapp` should now be `v2`.

```
student@ckad-1:~/app1$ kubectl describe pod try1-895fccfb-ttqdn |grep Image
    Image:          10.105.119.236:5000/simpleapp:v2
    Image ID:\
       docker-pullable://10.105.119.236:5000/simpleapp@sha256:6cf74051d09
463d89f1531fceb9c44cbf99006f8d9b407dd91d8f07baeee7e9c
    Image:          k8s.gcr.io/goproxy:0.1
    Image ID:\
       docker-pullable://k8s.gcr.io/goproxy@sha256:5334c7ad43048e3538775c
b09aaf184f5e8acf4b0ea60e3bc8f1d93c209865a5
```

7. View the update history of the deployment.

```
student@ckad-1:~/app1$ kubectl rollout history deployment try1
deployments "try1"
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

8. Compare the output of the **rollout history** for the two revisions. Images and labels should be different, with the image `v2` being the change we made.

```
student@ckad-1:~/app1$ kubectl rollout history deployment try1 \
  --revision=1 > one.out

student@ckad-1:~/app1$ kubectl rollout history deployment try1 \
  --revision=2 > two.out

student@ckad-/app11:~$ diff one.out two.out
1c1
< deployments "try1" with revision #1
---
> deployments "try1" with revision #2
3c3
<   Labels:        pod-template-hash=1509661973
---
```

```
>   Labels:        pod-template-hash=45197796
7c7
<     Image:        10.105.119.236:5000/simpleapp:latest
---
>     Image:        10.105.119.236:5000/simpleapp:v2
```

9. View what would be undone using the **–dry-run** option while undoing the rollout. This allows us to see the new template prior to using it.

```
student@ckad-1:~/app1$ kubectl rollout undo --dry-run=true deployment/try1
deployment.extensions/try1
Pod Template:
  Labels:        pod-template-hash=1509661973
        run=try1
  Containers:
   try1:
    Image:        10.105.119.236:5000/simpleapp:latest
    Port:        <none>
<output_omitted>
```

10. View the pods. Depending on how fast you type the `try1` pods should be about 2 minutes old.

```
student@ckad-1:~/app1$ kubectl get pods
NAME                         READY     STATUS     RESTARTS   AGE
nginx-6b58d9cdfd-9fnl4       1/1       Running    1          5d
registry-795c6c8b8f-hl5wf    1/1       Running    2          5d
try1-594fbb5fc7-7dl7c        2/2       Running    0          2m
try1-594fbb5fc7-8mxlb        2/2       Running    0          2m
try1-594fbb5fc7-jr7h7        2/2       Running    0          2m
try1-594fbb5fc7-s24wt        2/2       Running    0          2m
try1-594fbb5fc7-xfffg        2/2       Running    0          2m
try1-594fbb5fc7-zfmz8        2/2       Running    0          2m
```

11. In our case there are only two revisions, which is also the default number kept. Were there more we could choose a particular version. The following command would have the same effect as the previous, without the **–dry-run** option.

```
student@ckad-1:~/app1$ kubectl rollout undo deployment try1 --to-revision=1
deployment.extensions/try1
```

12. Again, it can take a bit for the pods to be terminated and re-created. Keep checking back until they are all running again.

```
student@ckad-1:~/app1$ kubectl get pods
NAME                         READY     STATUS         RESTARTS   AGE
nginx-6b58d9cdfd-9fnl4       1/1       Running        1          5d
registry-795c6c8b8f-hl5wf    1/1       Running        2          5d
try1-594fbb5fc7-7dl7c        2/2       Terminating    0          3m
try1-594fbb5fc7-8mxlb        0/2       Terminating    0          2m
try1-594fbb5fc7-jr7h7        2/2       Terminating    0          3m
try1-594fbb5fc7-s24wt        2/2       Terminating    0          2m
try1-594fbb5fc7-xfffg        2/2       Terminating    0          3m
try1-594fbb5fc7-zfmz8        1/2       Terminating    0          2m
try1-895fccfb-8dn4b          2/2       Running        0          22s
try1-895fccfb-kz72j          2/2       Running        0          10s
try1-895fccfb-rxxtw          2/2       Running        0          24s
try1-895fccfb-srwq4          1/2       Running        0          11s
try1-895fccfb-vkvmb          2/2       Running        0          31s
try1-895fccfb-z46qr          2/2       Running        0          31s
```

Chapter 6

Security

6.1 Labs

Exercise 6.1: Working with Security

Overview

In this lab we will implement security features for new applications, as the simpleapp YAML file is getting long and more difficult to read. Kubernetes architecture favors smaller, decoupled, and transient applications working together. Well continue to emulate that in our exercises.

In this exercise we will create two new applications. One will be limited in its access to the host node, but have access to encoded data. The second will use a `network security policy` to move from the default all-access Kubernetes policies to a mostly closed network. First we will set `security contexts` for pods and containers, then create and consume secrets, then finish with configuring a network security policy.

Set SecurityContext for a Pod and Container

1. Begin by making a new directory for our second application. Change into that directory.

```
student@ckad-1:~$ mkdir ~/app2

student@ckad-1:~$ cd ~/app2/
```

2. Create a YAML file for the second application. In the example below we are using a simple image, busybox, which allows access to a shell, but not much more. We will add a `runAsUser` to both the pod as well as the container.

```
student@ckad-1:~/app2$ vim second.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - image: busybox
    name: secondapp
    command:
      - sleep
      - "3600"
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
    name: busy
```

3. Create the secondapp pod and verify its running. Unlike the previous deployment this application is running as a pod. Look at the YAML output, to compare and contrast with what a deployment looks like. The status section probably has the largest contrast.

```
student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created

student@ckad-1:~/app2$ kubectl get  pod secondapp
NAME        READY     STATUS    RESTARTS   AGE
secondapp   1/1       Running   0          21s

student@ckad-1:~/app2$ kubectl get pod secondapp -o yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: 2018-04-18T18:58:53Z
  name: secondapp
<output_omitted>
```

4. Execute a Bourne shell within the Pod. Check the user ID of the shell and other processes. It should show the container setting, not the pod. This allows for multiple containers within a pod to customize their UID if desired. As there is only one container in the pod we do not need to use the **-c busy** option.

```
student@ckad-1:~/app2$ kubectl exec -it secondapp -- sh
/ $ ps aux
PID   USER     TIME  COMMAND
    1  2000       0:00  sleep 3600
    8  2000       0:00  sh
   12  2000      0:00  ps aux
```

5. While here check the capabilities of the kernel. In upcoming steps we will modify these values.

```
/ $ grep Cap /proc/1/status
CapInh:        00000000a80425fb
CapPrm:        0000000000000000
CapEff:        0000000000000000
CapBnd:        00000000a80425fb
CapAmb:        0000000000000000
/ $ exit
```

6. Use the capability shell wrapper tool, the **capsh** command, to decode the output. We will view and compare the output in a few steps. Note that there are 14 comma separated capabilities listed.

```
student@ckad-1:~/app2$ capsh --decode=00000000a80425fb
0x00000000a80425fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,
cap_kill,cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service,
cap_net_raw,cap_sys_chroot,cap_mknod,cap_audit_write,cap_setfcap
```

7. Edit the YAML file to include new capabilities for the container. A capability allows granting of specific, elevated privileges without granting full root access. We will be setting **NET_ADMIN** to allow interface, routing, and other network configuration. Well also set **SYS_TIME**, which allows system clock configuration. More on kernel capabilities can be read here: https://github.com/torvalds/linux/blob/master/include/uapi/linux/capability.h

   It can take up to a minute for the pod to fully terminate, allowing the future pod to be created.

```
student@ckad-1:~/app2$ kubectl delete pod secondapp
pod "secondapp" deleted

student@ckad-1:~/app2$ vim second.yaml
<output_omitted>
      - sleep
      - "3600"
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
      capabilities:               #Add this and following line
        add: ["NET_ADMIN", "SYS_TIME"]
    name: busy
```

8. Create the pod again. Execute a shell within the container and review the Cap settings under /proc/1/status. They should be different from the previous instance.

```
student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created

student@ckad-1:~/app2$ kubectl exec -it secondapp -- sh
/ $ grep Cap /proc/1/status
CapInh:        00000000aa0435fb
CapPrm:        0000000000000000
CapEff:        0000000000000000
CapBnd:        00000000aa0435fb
CapAmb:        0000000000000000
/ $ exit
```

9. Decode the output again. Note that the instance now has 16 comma delimited capabilities listed. **cap_net_admin** is listed as well as **cap_sys_time**.

```
student@ckad-1:~/app2$ capsh --decode=00000000aa0435fb
0x00000000aa0435fb=cap_chown,cap_dac_override,cap_fowner,
cap_fsetid,cap_kill,cap_setgid,cap_setuid,cap_setpcap,
cap_net_bind_service,cap_net_admin,cap_net_raw,cap_sys_chroot,
cap_sys_time,cap_mknod,cap_audit_write,cap_setfcap
```

Create and consume Secrets

Secrets are consumed in a manner similar to ConfigMaps, covered in an earlier lab. While at-rest encryption is just now enabled, historically a secret was just base64 encoded. There are three types of encryption which can be configured.

1. Begin by generating an encoded password.

```
student@ckad-1:~/app2$ echo LFTr@1n | base64
TEZUckAxbgo=
```

2. Create a YAML file for the object with an API object kind set to Secret. Use the encoded key as a password parameter.

```
student@ckad-1:~/app2$ vim secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: lfsecret
data:
  password: TEZUckAxbgo=
```

3. Ingest the new object into the cluster.

```
student@ckad-1:~/app2$ kubectl create -f secret.yaml
secret/lfsecret created
```

4. Edit secondapp YAML file to use the secret as a volume mounted under `/mysqlpassword`. Note as there is a command executed the pod will restart when the command finishes every 3600 seconds, or every hour.

```
student@ckad-1:~/app2$ vim second.yaml
<output_omitted>
      allowPrivilegeEscalation: false
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
    volumeMounts:
    - mountPath: /mysqlpassword
      name: mysql
    name: busy
  volumes:
  - name: mysql
    secret:
      secretName: lfsecret

student@ckad-1:~/app2$ kubectl delete pod secondapp
pod "secondapp" deleted

student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created
```

5. Verify the pod is running, then check if the password is mounted where expected. We will find that the password is available in its clear-text, decoded state.

```
student@ckad-1:~/app2$ kubectl get pod secondapp
NAME        READY     STATUS    RESTARTS   AGE
secondapp   1/1       Running   0          34s

student@ckad-1:~/app2$ kubectl exec -ti secondapp -- /bin/sh
/ $ cat /mysqlpassword/password
LFTr@1n
```

6. View the location of the directory. Note it is a symbolic link to `../data` which is also a symbolic link to another directory. After taking a look at the filesystem within the container, exit back to the node.

```
/ $ cd /mysqlpassword/
/mysqlpassword $ ls
password
/mysqlpassword $ ls -al
total 4
drwxrwxrwt    3 root     root           100 Apr 11 07:24 .
drwxr-xr-x   21 root     root          4096 Apr 11 22:30 ..
drwxr-xr-x    2 root     root            60 Apr 11 07:24 ..4984_11_04_07_24_47.831222818
lrwxrwxrwx    1 root     root            31 Apr 11 07:24 ..data -> ..4984_11_04_07_24_47.831222818
lrwxrwxrwx    1 root     root            15 Apr 11 07:24 password -> ..data/password

/mysqlpassword $ exit
```

Working with ServiceAccounts

We can use `ServiceAccounts` to assign cluster roles, or the ability to use particular HTTP verbs. In this section we will create a new `ServiceAccount` and grant it access to view secrets.

1. Begin by viewing secrets, both in the default namespace as well as all.

```
student@ckad-1:~/app2$ cd

student@ckad-1:~$ kubectl get secrets
NAME                  TYPE                                  DATA   AGE
default-token-c4rdg   kubernetes.io/service-account-token   3      4d16h
lfsecret              Opaque                                1      6m5s

student@ckad-1:~$ kubectl get secrets --all-namespaces
NAMESPACE     NAME
 TYPE                                DATA   AGE
default       default-token-c4rdg
 kubernetes.io/service-account-token   3      4d16h
kube-public   default-token-zqzbg
 kubernetes.io/service-account-token   3      4d16h
kube-system   attachdetach-controller-token-wxzvc
 kubernetes.io/service-account-token   3      4d16h
<output_omitted>
```

2. We can see that each agent uses a secret in order to interact with the API server. We will create a new `ServiceAccount` which will have access.

```
student@ckad-1:~$ vim serviceaccount.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
 name: secret-access-sa


student@ckad-1:~$ kubectl create -f serviceaccount.yaml
serviceaccount/secret-access-sa created

student@ckad-1:~$ kubectl get serviceaccounts
NAME               SECRETS   AGE
default            1         4d17h
secret-access-sa   1         34s
```

3. Now we will create a `ClusterRole` which will list the actual actions allowed cluster-wide. We will look at an existing role to see the syntax.

```
student@ckad-1:~$ kubectl get clusterroles
NAME                                                                   AGE
admin                                                                  4d17h
calico-cni-plugin                                                      4d17h
calico-kube-controllers                                                4d17h
cluster-admin                                                          4d17h
<output_omitted>
```

4. View the details for the `admin` and compare it to the `cluster-admin`. The `admin` has particular actions allowed, but `cluster-admin` has the meta-character '*' allowing all actions.

```
student@ckad-1:~$ kubectl get clusterroles admin -o yaml
<output_omitted>

student@ckad-1:~$ kubectl get clusterroles cluster-admin -o yaml
<output_omitted>
```

5. Using some of the output above, we will create our own file.

```
student@ckad-1:~$ vim clusterrole.yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
 name: secret-access-cr
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
```

6. Create and verify the new `ClusterRole`.

```
student@ckad-1:~$ kubectl create -f clusterrole.yaml
clusterrole.rbac.authorization.k8s.io/secret-access-cr created

student@ckad-1:~$ kubectl get clusterrole secret-access-cr -o yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: 2018-10-18T19:27:24Z
  name: secret-access-cr
<output_omitted>
```

7. Now we bind the role to the account. Create another YAML file which uses `roleRef:`:

```
student@ckad-1:~$ vim rolebinding.yaml
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: RoleBinding
metadata:
 name: secret-rb
subjects:
- kind: ServiceAccount
  name: secret-access-sa
roleRef:
 kind: ClusterRole
 name: secret-access-cr
 apiGroup: rbac.authorization.k8s.io
```

8. Create the new `RoleBinding` and verify.

```
student@ckad-1:~$ kubectl create -f rolebinding.yaml
rolebinding.rbac.authorization.k8s.io/secret-rb created

student@ckad-1:~$ kubectl get rolebindings
NAME        AGE
secret-rb   17s
```

9. View the `secondapp` pod and **grep** for secret settings. Note that it uses the default settings.

```
student@ckad-1:~$ kubectl describe pod secondapp |grep -i secret
   /var/run/secrets/kubernetes.io/serviceaccount from
     default-token-c4rdg (ro)
 Type:         Secret (a volume populated by a Secret)
 SecretName:   default-token-c4rdg
```

10. Edit the `second.yaml` file and add the use of the `serviceAccount`.

```
student@ckad-1:~$ vim second.yaml
....
  name: secondapp
spec:
  serviceAccountName: secret-access-sa  #<-- Add this line
  securityContext:
    runAsUser: 1000
....
```

11. We will delete the `secondapp` pod if still running, then create it again. View what the secret is by default.

```
student@ckad-1:~$ kubectl delete pod secondapp ; \
   kubectl create -f second.yaml
pod "secondapp" deleted
pod/secondapp created

student@ckad-1:~$ kubectl describe pod secondapp |grep -i secret
    /var/run/secrets/kubernetes.io/serviceaccount from
 secret-access-sa-token-wd7vm (ro)
  secret-access-sa-token-wd7vm:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  secret-access-sa-token-wd7vm
```

Implement a NetworkPolicy

An early architecture decision with Kubernetes was non-isolation, that all pods were able to connect to all other pods and nodes by design. In more recent releases the use of a `NetworkPolicy` allows for pod isolation. The policy only has effect when the network plugin, like **Project Calico**, are capable of honoring them. If used with a plugin like **flannel** they will have no effect. The use of `matchLabels` allows for more granular selection within the namespace which can be selected using a `namespaceSelector`. Using multiple labels can allow for complex application of rules. More information can be found here: https://kubernetes.io/docs/concepts/services-networking/network-policies

1. Begin by creating a default policy which denies all traffic. Once ingested into the cluster this will affect every pod not selected by another policy, creating a mostly-closed environment. If you want to only deny ingress or egress traffic you can remove the other `policyType`.

```
student@ckad-1:~/app2$ vim allclosed.yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-default
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

2. Before we can test the new network policy we need to make sure network access works without it applied. Update **secondapp** to include a new container running **nginx**, then test access. Begin by adding two lines for the **nginx** image and name `webserver`, as found below. It takes a bit for the pod to terminate, so well delete then edit the file.

```
student@ckad-1:~/app2$ kubectl delete pod secondapp
pod "secondapp" deleted

student@ckad-1:~/app2$ vim second.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - image: nginx
    name: webserver
    ports:
    - containerPort: 80
  - image: busybox
```

```
      name: secondapp
      command:
  <output_omitted>
```

3. Create the new pod. Be aware the pod will move from `ContainerCreating` to Error to `CrashLoopBackOff`, as only one of the containers will start. We will troubleshoot the error in following steps.

```
student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created

student@ckad-1:~/app2$ kubectl get pods
NAME                       READY   STATUS             RESTARTS   AGE
date-1523648520-z282m      0/1     Completed          0          3m
date-1523648580-zznjs      0/1     Completed          0          2m
date-1523648640-2h6qt      0/1     Completed          0          1m
date-1523648700-drfp5      1/1     Running            0          18s
nginx-6b58d9cdfd-9fnl4     1/1     Running            1          8d
Registry-795c6c8b8f-hl5wf  1/1     Running            2          8d
secondapp                  1/2     CrashLoopBackOff   1          13s
```

4. Take a closer look at the events leading up to the failure. The images were pulled and the container was started. It was the full execution of the container which failed.

```
student@ckad-1:~/app2$ kubectl get event
<output_omitted>
Normal     Created            kubelet, ckad-2-wdrq    Created container
5m          5m           secondapp.1525166daeeb0e43
Pod          spec.containers{busy}            Normal     Started
kubelet, ckad-2-wdrq    Started container
20s          5m            25        secondapp.1525166e5791a7fd
Pod          spec.containers{webserver}       Warning   BackOff
kubelet, ckad-2-wdrq    Back-off restarting failed container
```

5. View the logs of the **webserver** container mentioned in the previous output. Note there are errors about the user directive and not having permission to make directories.

```
student@ckad-1:~/app2$ kubectl logs secondapp webserver
2018/04/13 19:51:13 [warn] 1#1: the "user" directive makes sense
only if the master process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
nginx: [warn] the "user" directive makes sense only if the master
process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
2018/04/13 19:51:13 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp"
failed (13: Permission denied)
nginx: [emerg] mkdir() "/var/cache/nginx/client_temp" failed
(13: Permission denied)
```

6. Delete the pods. Edit the YAML file to comment out the setting of a UID for the entire pod.

```
student@ckad-1:~/app2$ kubectl delete -f second.yaml
pod "secondapp" deleted

student@ckad-1:~/app2$ vim second.yaml
<output_omitted>
  name: secondapp
spec:
#  securityContext:
#    runAsUser: 1000
  containers:
  - image: nginx
    name: webserver
    ports:
    - containerPort: 80
<output_omitted>
```

7. Create the pod again. This time both containers should run. You may have to wait for the previous pod to fully terminate, depending on how fast you type.

```
student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created

student@ckad-1:~/app2$ kubectl get pods
NAME                        READY     STATUS      RESTARTS    AGE
<output_omitted>
secondapp                   2/2       Running     0           5s
```

8. Expose the **webserver** using a `NodePort` service. Expect an error due to lack of labels.

```
student@ckad-1:~/app2$ kubectl expose pod secondapp --type=NodePort --port=80
error: couldn't retrieve selectors via --selector flag or introspection: the pod has no labels and cannot be exposed
See 'kubectl expose -h' for help and examples.
```

9. Edit the YAML file to add a label in the metadata, adding the `example: second` label right after the pod name. Note you can delete several resources at once by passing the YAML file to the delete command. Delete and recreate the pod. It may take up to a minute for the pod to shut down.

```
student@ckad-1:~/app2$ kubectl delete -f second.yaml
pod "secondapp" deleted

student@ckad-1:~/app2$ vim second.yaml
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
  labels:
    example: second
spec:
#  securityContext:
#    runAsUser: 1000
<output_omitted>

student@ckad-1:~/app2$ kubectl create -f second.yaml
pod/secondapp created

student@ckad-1:~/app2$ kubectl get pods
NAME                        READY     STATUS      RESTARTS    AGE
<output_omitted>
secondapp                   2/2       Running     0           15s
```

10. This time we will expose a `NodePort` again, and create the service separately, then add a label to illustrate how labels are essential for tying resources together inside of kubernetes.

```
student@ckad-1:~/app2$ kubectl create service nodeport secondapp --tcp=80
service/secondapp created
```

11. Look at the details of the service. Note the selector is set to `app: secondapp`. Also take note of the nodePort, which is `31655` in the example below, yours may be different.

```
student@ckad-1:~/app2$ kubectl get svc secondapp -o yaml
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: 2018-04-19T22:07:25Z
  labels:
    app: secondapp
  name: secondapp
  namespace: default
  resourceVersion: "216490"
  selfLink: /api/v1/namespaces/default/services/secondapp
```

```
    uid: 0aeaea82-441e-11e8-ac6e-42010a800007
  spec:
    clusterIP: 10.97.96.75
    externalTrafficPolicy: Cluster
    ports:
    - name: "80"
      nodePort: 31655
      port: 80
      protocol: TCP
      targetPort: 80
    selector:
      app: secondapp
    sessionAffinity: None
    type: NodePort
  status:
    loadBalancer: {}
```

12. Test access to the service using **curl** and the `ClusterIP` shown in the previous output. As the label does not match any other resources, the **curl** command should hang and eventually timeout.

```
student@ckad-1:~/app2$ curl http://10.97.96.75
```

13. Edit the service. We will change the label to match **secondapp**, and set the `nodePort` to a new port, one that may have been specifically opened by our firewall team, port `32000`.

```
student@ckad-1:~/app2$ kubectl edit svc secondapp
 <output_omitted>
  ports:
  - name: "80"
    nodePort: 32000       ## Edit this line
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    example: second       ## Edit this line, too
  sessionAffinity: None
<output_omitted>
```

14. Verify the updated port number is showing properly, and take note of the ClusterIP. The example below shows a ClusterIP of `10.97.96.75` and a port of `32000` as expected.

```
student@ckad-1:~/app2$ kubectl get svc
NAME         TYPE      CLUSTER-IP     EXTERNAL-IP  PORT(S)        AGE
<output_omitted>
secondapp    NodePort  10.97.96.75    <none>       80:32000/TCP   5m
```

15. Test access to the high port. You should get the default nginx welcome page both if you test from the node to the `ClusterIP:<low-port-number>` and from the exterior `hostIP:high-port-number>`. As the high port is randomly generated make sure its available. Both of your nodes should be exposing the web server on port 32000.

```
student@ckad-1:~/app2$ curl http://10.97.96.75
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>

[user@laptop ~]$ curl http://35.184.219.5:32000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

16. Now test egress from a container to the outside world. Well use the **netcat** command to verify access to a running web server on port 80. First test local access to nginx, then a remote server.

```
student@ckad-1:~/app2$ kubectl exec -it -c busy secondapp sh
/ $ nc -vz 127.0.0.1 80
127.0.0.1 (127.0.0.1:80) open

/ $ nc -vz www.linux.com 80
www.linux.com (151.101.185.5:80) open

/ $ exit
```

Testing the Policy

1. Now that we have tested both ingress and egress we can implement the network policy.

```
student@ckad-1:~/app2$ kubectl create -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default created
```

2. Use the ingress and egress tests again. Three of the four should eventually timeout. Start by testing from outside the cluster.

```
[user@laptop ~]$ curl http://35.184.219.5:32215
curl: (7) Failed to connect to 35.184.219.5 port
32000: Connection timed out
```

3. Then test from the host to the container.

```
student@ckad-1:~/app2$ curl http://10.97.96.75:80
curl: (7) Failed to connect to 10.97.96.75 port 80: Connection timed out
```

4. Now test egress. From container to container should work, as the filter is outside of the pod. Then test egress to an external web page. It should eventually timeout.

```
student@ckad-1:~/app2$ kubectl exec -it -c busy secondapp sh
/ $ nc -vz 127.0.0.1 80
127.0.0.1 (127.0.0.1:80) open

/ $ nc -vz www.linux.com 80
nc: bad address 'www.linux.com'

/ $ exit
```

5. Update the NetworkPolicy and comment out the Egress line. Then replace the policy.

```
student@ckad-1:~/app2$ vim allclosed.yaml
.
#  - Egress

student@ckad-1:~/app2$ kubectl replace -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default replaced
```

6. Test egress access to an outside site. Get the IP address of the **eth0** inside the container while logged in. The IP is `192.168.55.91` in the example below, yours may be different.

```
student@ckad-1:~/app2$ kubectl exec -it -c busy secondapp sh

/ $ nc -vz www.linux.com 80
www.linux.com (151.101.185.5:80) open

/ $ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1000
```

```
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tunl0@NONE: <NOARP> mtu 1480 qdisc noop qlen 1000
    link/ipip 0.0.0.0 brd 0.0.0.0
4: eth0@if59: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue
    link/ether 1e:c8:7d:6a:96:c3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.55.91/32 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::1cc8:7dff:fe6a:96c3/64 scope link
       valid_lft forever preferred_lft forever

/ $ exit
```

7. Now add a selector to allow ingress to only the nginx container. Use the IP from the **eth0** range.

```
student@ckad-1:~/app2$ vim allclosed.yaml
<output_omitted>
 policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 192.168.0.0/16
```

8. Recreate the policy, and verify its configuration.

```
student@ckad-1:~/app2$ kubectl replace -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default replaced

student@ckad-1:~/app2$ kubectl get networkpolicy
NAME           POD-SELECTOR     AGE
deny-default   <none>                    15s

student@ckad-1:~/app2$ kubectl get networkpolicy -o yaml
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: NetworkPolicy
  metadata:
<output_omitted>
```

9. Test access to the container both using **curl** as well as **ping**, the IP address to use was found from the IP inside the container.

```
student@ckad-1:~/app2$ curl http://192.168.55.91
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>

student@ckad-1:~/app2$ ping -c5 192.168.55.91
PING 192.168.55.91 (192.168.55.91) 56(84) bytes of data.
64 bytes from 192.168.55.91: icmp_seq=1 ttl=63 time=1.11 ms
64 bytes from 192.168.55.91: icmp_seq=2 ttl=63 time=0.352 ms
64 bytes from 192.168.55.91: icmp_seq=3 ttl=63 time=0.350 ms
64 bytes from 192.168.55.91: icmp_seq=4 ttl=63 time=0.359 ms
64 bytes from 192.168.55.91: icmp_seq=5 ttl=63 time=0.295 ms

--- 192.168.55.91 ping statistics ---
```

```
5 packets transmitted, 5 received, 0% packet loss, time 4054ms
rtt min/avg/max/mdev = 0.295/0.495/1.119/0.312 ms
```

10. Update the policy to only allow ingress for TCP traffic on port 80, then test with **curl**, which should work. The ports entry should line up with the from entry a few lines above.

```
student@ckad-1:~/app2$ vim allclosed.yaml
<output_omitted>
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 192.168.0.0/16
    ports:
    - port: 80
      protocol: TCP

student@ckad-1:~/app2$ kubectl replace -f allclosed.yaml
networkpolicy.networking.k8s.io/deny-default replaced

student@ckad-1:~/app2$ curl http://192.168.55.91
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

11. All five pings should fail, with zero received.

```
student@ckad-1:~/app2$ ping -c5 192.168.55.91
PING 192.168.55.91 (192.168.55.91) 56(84) bytes of data.

--- 192.168.55.91 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4098ms
```

Chapter 7

Exposing Applications

7.1 Labs

Exercise 7.1: Exposing Applications

Overview

In this lab we will explore various ways to expose an application to other pods and outside the cluster. We will add to the NodePort used in previous labs other service options.

Expose A Service

1. We will begin by using the default service type ClusterIP. This is a cluster internal IP, only reachable from within the cluster. Begin by viewing the existing services.

```
student@ckad-1:~$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP        8d
nginx        ClusterIP   10.108.95.67     <none>        443/TCP        8d
registry     ClusterIP   10.105.119.236   <none>        5000/TCP       8d
secondapp    NodePort    10.111.26.8      <none>        80:32000/TCP   7h
```

2. Delete the existing service for secondapp.

```
student@ckad-1:~/app2$ kubectl delete svc secondapp
service "secondapp" deleted
```

3. Create a YAML file for a replacement service, which would be persistent. Use the label to select the secondapp. Expose the same port and protocol of the previous service.

```
student@ckad-1:~/app2$ vim service.yaml
apiVersion: v1
kind: Service
metadata:
  name: secondapp
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    example: second
```

4. Create the service, find the new IP and port. Note there is no high number port as this is internal access only.

```
student@ckad-1:~/app2$ kubectl create -f service.yaml
service/secondapp created

student@ckad-1:~/app2$ kubectl get svc
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes   ClusterIP   10.96.0.1        <none>        443/TCP    8d
nginx        ClusterIP   10.108.95.67     <none>        443/TCP    8d
registry     ClusterIP   10.105.119.236   <none>        5000/TCP   8d
secondapp    ClusterIP   10.98.148.52     <none>        80/TCP     14s
```

5. Test access. You should see the default welcome page again.

```
student@ckad-1:~/app2$ curl http://10.98.148.52
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

6. To expose a port to outside the cluster we will create a NodePort. We had done this in a previous step from the command line. When we create a NodePort it will create a new ClusterIP automatically. Edit the YAML file again. Add type: NodePort. Also add the high-port to match an open port in the firewall as mentioned in the previous chapter. Youll have to delete and re-create as the existing IP is immutable, but not able to be reused. The NodePort will try to create a new ClusterIP instead.

```
student@ckad-1:~/app2$ vim service.yaml
apiVersion: v1
kind: Service
metadata:
  name: secondapp
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
    nodePort: 32000
  type: NodePort
  selector:
    example: second

student@ckad-1:~/app2$ kubectl delete svc secondapp ; kubectl create \
 -f service.yaml
service "secondapp" deleted
service/secondapp created
```

7. Find the new ClusterIP and ports for the service.

```
student@ckad-1:~/app2$ kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP   10.96.0.1      <none>        443/TCP        8d
```

```
nginx        ClusterIP    10.108.95.67   <none>        443/TCP       8d
registry     ClusterIP    10.105.119.236 <none>        5000/TCP      8d
secondapp    NodePort     10.109.134.221 <none>        80:32000/TCP  4s
```

8. Test the low port number using the new ClusterIP for the secondapp service.

```
student@ckad-1:~/app2$ curl 10.109.134.221
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

9. Test access from an external node to the host IP and the high container port. Your IP and port will be different. It should work, even with the network policy in place, as the traffic is arriving via a 192.168.0.0 port.

```
user@laptop:~/Desktop$ curl http://35.184.219.5:32000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

10. The use of a LoadBalancer makes an asynchronous request to an external provider for a load balancer if one is available. It then creates a NodePort and waits for a response including the external IP. The local NodePort will work even before the load balancer replies. Edit the YAML file and change the type to be LoadBalancer.

```
student@ckad-1:~/app2$ vim service.yaml
<output_omitted>
  - port: 80
    protocol: TCP
  type: LoadBalancer
  selector:
    example: second

student@ckad-1:~/app2$ kubectl delete svc secondapp
service "secondapp" deleted

student@ckad-1:~/app2$ kubectl create -f service.yaml
service/secondapp created
```

11. As mentioned the cloud provider is not configured to provide a load balancer; the External-IP will remain in pending state. Some issues have been found using this with VirtualBox.

```
student@ckad-1:~/app2$ kubectl get svc
NAME         TYPE          CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP     10.96.0.1        <none>        443/TCP        8d
nginx        ClusterIP     10.108.95.67     <none>        443/TCP        8d
registry     ClusterIP     10.105.119.236   <none>        5000/TCP       8d
secondapp    LoadBalancer  10.109.26.21     <pending>     80:32000/TCP   4s
```

12. Test again local and from a remote node. The IP addresses and ports will be different on your node.

```
serewic@laptop:~/Desktop$ curl http://35.184.219.5:32000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

Ingress Controller

If you have a large number of services to expose outside of the cluster, or to expose a low-number port on the host node you can deploy an ingress controller. While nginx and GCE have controllers officially supported by Kubernetes.io, the Traefik Ingress Controller is easier to install. At the moment.

1. As we have RBAC configured we need to make sure the controller will run and be able to work with all necessary ports, endpoints and resources. Create a YAML file to declare a clusterrole and a clusterrolebinding

```
student@ckad-1:~/app2$ vim ingress.rbac.yaml
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: traefik-ingress-controller
rules:
  - apiGroups:
      - ""
    resources:
      - services
      - endpoints
      - secrets
    verbs:
      - get
      - list
      - watch
  - apiGroups:
      - extensions
    resources:
      - ingresses
    verbs:
      - get
      - list
      - watch
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: traefik-ingress-controller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: traefik-ingress-controller
subjects:
- kind: ServiceAccount
  name: traefik-ingress-controller
  namespace: kube-system
```

2. Create the new role and binding.

```
student@ckad-1:~/app2$ kubectl create -f ingress.rbac.yaml
clusterrole.rbac.authorization.k8s.io/traefik-ingress-controller created
clusterrolebinding.rbac.authorization.k8s.io/traefik-ingress-controller created
```

3. Create the Traefik controller. We will use a script directly from their website. The shorter, easier URL for the following is: https://goo.gl/D2uEEF. We will need to download the script and make some edits before creating the objects.

```
student@ckad-1:~/app2$ wget \
https://raw.githubusercontent.com/containous/traefik/master/examples/k8s/traefik-ds.yaml
```

4. Edit the downloaded file. The output below represents the changes in a diff type output, from the downloaded to the edited file. One line should be added, six lines should be removed.

```
student@ckad-1:~/app2$ vim traefik-ds.yaml
23a24              ##  Add the following line 24
>       hostNetwork: true
34,39d34         ##  Remove these lines around line 34
<         securityContext:
<           capabilities:
<             drop:
<             - ALL
<             add:
<             - NET_BIND_SERVICE

The file should look like this:
.
      terminationGracePeriodSeconds: 60
      hostNetwork: True
      containers:
      - image: traefik
        name: traefik-ingress-lb
        ports:
        - name: http
          containerPort: 80
          hostPort: 80
        - name: admin
          containerPort: 8080
          hostPort: 8080
        args:
        - --api
.
```

5. Create the objects using the edited file.

```
student@ckad-1:~/app2$ kubectl apply -f traefik-ds.yaml
serviceaccount/traefik-ingress-controller created
daemonset.extensions/traefik-ingress-controller created
service/traefik-ingress-service created
```

6. Now that there is a new controller we need to pass some rules, so it knows how to handle requests. Note that the host mentioned is www.example.com, which is probably not your node name. We will pass a false header when testing. Also the service name needs to match the secondapp weve been working with.

```
student@ckad-1:~/app2$ vim ingress.rule.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-test
  annotations:
    kubernetes.io/ingress.class: traefik
spec:
  rules:
  - host: www.example.com
    http:
      paths:
      - backend:
          serviceName: secondapp
          servicePort: 80
        path: /
```

7. Now ingest the rule into the cluster.

```
student@ckad-1:~/app2$ kubectl create -f ingress.rule.yaml
ingress.extensions/ingress-test created
```

8. We should be able to test the internal and external IP addresses, and see the nginx welcome page. The loadbalancer would present the traffic, a curl request in this case, to the externally facing interface. Use ip a to find the IP address of the interface which would face the load balancer. In this example the interface would be ens4, and the IP would be 10.128.0.7.

```
student@ckad-1:~$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: ens4: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc mq state UP group default qlen 1000
    link/ether 42:01:0a:80:00:03 brd ff:ff:ff:ff:ff:ff
    inet 10.128.0.7/32 brd 10.128.0.3 scope global ens4
       valid_lft forever preferred_lft forever
<output_omitted>

student@ckad-1:~/app2$ curl -H "Host: www.example.com" http://10.128.0.7/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>

user@laptop:~$ curl -H "Host: www.example.com" http://35.193.3.179
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
<output_omitted>
```

9. At this point we would keep adding more and more web servers. Well configure one more, which would then be a process continued as many times as desired. Begin by deploying another nginx server. Give it a label and expose port 80.

```
student@ckad-1:~/app2$ kubectl run thirdpage --image=nginx \
  --port=80 -l example=third
deployment.apps "thirdpage" created
```

10. Expose the new server as a NodePort.

```
student@ckad-1:~/app2$ kubectl expose deployment thirdpage --type=NodePort
service "thirdpage" exposed
```

11. Now we will customize the installation. Run a bash shell inside the new pod. Your pod name will end differently. Install vim inside the container then edit the index.html file of nginx so that the title of the web page will be Third Page.

```
student@ckad-1:~/app2$ kubectl exec -it thirdpage-5cf8d67664-zcmfh -- /bin/bash

root@thirdpage-5cf8d67664-zcmfh:/# apt-get update

root@thirdpage-5cf8d67664-zcmfh:/#  apt-get install vim -y

root@thirdpage-5cf8d67664-zcmfh:/# vim /usr/share/nginx/html/index.html
<!DOCTYPE html>
<html>
<head>
<title>Third Page</title>
<style>
<output_omitted>
```

12. Edit the ingress rules to point the thirdpage service.

```
student@ckad-1:~/app2$ kubectl edit ingress ingress-test
<output_omitted>
  - host: www.example.com
    http:
      paths:
      - backend:
          serviceName: secondapp
          servicePort: 80
        path: /
  - host: thirdpage.org
    http:
      paths:
      - backend:
          serviceName: thirdpage
          servicePort: 80
        path: /
 status:
<output_omitted>
```

13. Test the second hostname using curl locally as well as from a remote system.

```
student@ckad-1:~/app2$ curl -H "Host: thirdpage.org" http://10.128.0.7/
<!DOCTYPE html>
<html>
<head>
<title>Third Page</title>
<style>
<output_omitted>
```

Chapter 8

Troubleshooting

8.1 Labs

Exercise 8.1: Troubleshooting

Overview

Troubleshooting can be difficult in a multi-node, decoupled and transient environment. Add in the rapid pace of change and it becomes more difficult. Instead of focusing and remembering a particular error and the fix it may be more useful to learn a flow of troubleshooting and revisit assumptions until the pace of change slows and various areas further mature.

Monitor Applications

1. View the `secondapp` pod, it should show as `Running`. This may not mean the application within is working properly, but that the pod is running. The restarts are due to the command we have written to run. The pod exists when done, and the controller restarts another container inside. The count depends on how long the labs have been running.

```
student@ckad-1/app1:~$ cd
student@ckad-1:~$ kubectl get pods secondapp
NAME                        READY     STATUS      RESTARTS    AGE
secondapp                   2/2       Running     49          2d
```

2. Look closer at the pod. Working slowly through the output check each line. If you have issues, are other pods having issues on the same node or volume? Check the state of each container. Both `busy` and `webserver` should report as `Running`. Note `webserver` has a restart count of zero while `busy` has a restart count of 49. We expect this as, in our case, the pod has been running for 49 hours.

```
student@ckad-1:~$ kubectl describe pod secondapp
Name:         secondapp
Namespace:    default
Node:         ckad-2-wdrq/10.128.0.2
Start Time:   Fri, 13 Apr 2018 20:34:56 +0000
Labels:       example=second
```

```
Annotations:  <none>
Status:       Running
IP:           192.168.55.91
Containers:
  webserver:
<output_omitted>
    State:          Running
      Started:      Fri, 13 Apr 2018 20:34:58 +0000
    Ready:          True
    Restart Count:  0
<output_omitted>

  busy:
<output_omitted>

    State:          Running
      Started:      Sun, 15 Apr 2018 21:36:20 +0000
    Last State:     Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Sun, 15 Apr 2018 20:36:18 +0000
      Finished:     Sun, 15 Apr 2018 21:36:18 +0000
    Ready:          True
    Restart Count:  49
    Environment:    <none>
```

3. There are three values for conditions. Check that the pod reports Initialized, Ready and scheduled.

```
<output_omitted>
Conditions:
  Type           Status
  Initialized    True
  Ready          True
  PodScheduled   True
<output_omitted>
```

4. Check if there are any events with errors or warnings which may indicate what is causing any problems.

```
Events:
  Type    Reason   Age                 From                Message
  ----    ------   ----                ----                -------
  Normal  Pulling  34m (x50 over 2d)  kubelet, ckad-2-wdrq  pulling
image "busybox"
  Normal  Pulled   34m (x50 over 2d)  kubelet, ckad-2-wdrq  Successfully
pulled image "busybox"
  Normal  Created  34m (x50 over 2d)  kubelet, ckad-2-wdrq  Created
container
  Normal  Started  34m (x50 over 2d)  kubelet, ckad-2-wdrq  Started
container
```

5. View each container log. You may have to sift errors from expected output. Some containers may have no output at all, as is found with `busy`.

```
student@ckad-1:~$ kubectl logs secondapp webserver
192.168.55.0 - - [13/Apr/2018:21:18:13 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.47.0" "-"
192.168.55.0 - - [13/Apr/2018:21:20:35 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.53.1" "-"
127.0.0.1 - - [13/Apr/2018:21:25:29 +0000] "GET" 400 174 "-" "-" "-"
127.0.0.1 - - [13/Apr/2018:21:26:19 +0000] "GET index.html" 400 174
"-" "-" "-"
<output_omitted>
```

```
student@ckad-1:~$ kubectl logs secondapp busy
student@ckad-1:~$
```

6. Check to make sure the container is able to use DNS and communicate with the outside world. Remember we still have limited the UID for `secondapp` to be UID **2000**, which may prevent some commands from running. It can also prevent an application from completing expected tasks.

```
student@ckad-1:~$ kubectl exec  -it  secondapp -c busy -- sh
/ $ nslookup www.linux.com
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      www.linux.com
Address 1: 151.101.45.5

/ $ cat /etc/resolv.conf
nameserver 10.96.0.10
search default.svc.cluster.local svc.cluster.local
cluster.local c.endless-station-188822.internal
google.internal
options ndots:5
```

7. Test access to a remote node using **nc (NetCat)**. There are several options to **nc** which can help troubleshoot if the problem is the local node, something between nodes or in the target. In the example below the connect never completes and a **control-c** was used to interrupt.

```
/ $ nc www.linux.com 25
^Cpunt!
```

8. Test using an IP address in order to narrow the issue to name resolution. In this case the IP in use is a well known IP for Googles DNS servers. The following example shows that Internet name resolution is working, but our UID issue prevents access to the `index.html` file.

```
/ $ wget http://www.linux.com/
Connecting to www.linux.com (151.101.45.5:80)
Connecting to www.linux.com (151.101.45.5:443)
wget: can't open 'index.html': Permission denied

/ $ exit
```

9. Make sure traffic is being sent to the correct Pod. Check the details of both the service and endpoint. Pay close attention to ports in use as a simple typo can prevent traffic from reaching the proper pod. Make sure labels and selectors dont have any typos as well.

```
student@ckad-1:~$ kubectl get svc
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)        AGE
kubernetes   ClusterIP      10.96.0.1        <none>        443/TCP        10d
nginx        ClusterIP      10.108.95.67     <none>        443/TCP        10d
registry     ClusterIP      10.105.119.236   <none>        5000/TCP       10d
secondapp    LoadBalancer   10.109.26.21     <pending>     80:32000/TCP   1d
thirdpage    NodePort       10.109.250.78    <none>        80:31230/TCP   1h

student@ckad-1:~$ kubectl get svc secondapp -o yaml
<output_omitted>
  clusterIP: 10.109.26.21
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 32000
    port: 80
    protocol: TCP
    targetPort: 80
  selector:
    example: second
<output_omitted>
```

10. Verify an endpoint for the service exists and has expected values, including namespaces, ports and protocols.

```
student@ckad-1:~$ kubectl get ep
NAME         ENDPOINTS             AGE
kubernetes   10.128.0.3:6443       10d
nginx        192.168.55.68:443     10d
registry     192.168.55.69:5000    10d
secondapp    192.168.55.91:80      1d
thirdpage    192.168.241.57:80     1h

student@ckad-1:~$ kubectl get ep secondapp -o yaml
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: 2018-04-14T05:37:32Z
<output_omitted>
```

11. If the containers, services and endpoints are working the issue may be with an infrastructure service like **kube-proxy**. Ensure its running, then look for errors in the logs. As we have two nodes we will have two proxies to look at. As we built our cluster with **kubeadm** the proxy runs as a container. On other systems you may need to use **journalctl** or look under `/var/log/kube-proxy.log`.

```
student@ckad-1:~$ ps -elf |grep kube-proxy
4 S root      2864  2847  0  80   0 - 14178 -      15:45 ?
 00:00:56 /usr/local/bin/kube-proxy --config=/var/lib/kube-proxy/config.conf
0 S student  23513 18282  0  80   0 -  3236 pipe_w 22:49 pts/0
 00:00:00 grep --color=auto kube-proxy

student@ckad-1:~$ journalctl -a | grep proxy
Apr 15 15:44:43 ckad-2-nzjr audit[742]: AVC apparmor="STATUS"
 operation="profile_load" profile="unconfined" \
  name="/usr/lib/lxd/lxd-bridge-proxy" pid=742 comm="apparmor_parser"
Apr 15 15:44:43 ckad-2-nzjr kernel: audit: type=1400
 audit(1523807083.011:11): apparmor="STATUS" \
  operation="profile_load" profile="unconfined" \
   name="/usr/lib/lxd/lxd-bridge-proxy" pid=742 comm="apparmor_parser"
Apr 15 15:45:17 ckad-2-nzjr kubelet[1248]: I0415 15:45:17.153670
 1248 reconciler.go:217] operationExecutor.VerifyControllerAttachedVolume\
  started for volume "xtables-lock" \
   (UniqueName: "kubernetes.io/host-path/e701fc01-38f3-11e8-a142-\
   42010a800003-xtables-lock") \
   pod "kube-proxy-t8k4w" (UID: "e701fc01-38f3-11e8-a142-42010a800003")
```

12. Look at both of the proxy logs. Lines which begin with the character **I** are `info`, **E** are `errors`. In this example the last message says access to listing an endpoint was denied by RBAC. It was because a default installation via Helm wasnt RBAC aware. If not using command line completion, view the possible pod names first.

```
student@ckad-1:~$ kubectl -n kube-system get pod

student@ckad-1:~$ kubectl -n kube-system logs kube-proxy-fsdfr
I0405 17:28:37.091224       1 feature_gate.go:190] feature gates: map[]
W0405 17:28:37.100565       1 server_others.go:289] Flag proxy-mode=""
unknown, assuming iptables proxy
I0405 17:28:37.101846       1 server_others.go:138] Using iptables Proxier.
I0405 17:28:37.121601       1 server_others.go:171] Tearing down
inactive rules.
<output_omitted>
E0415 15:45:17.086081       1 reflector.go:205] \
  k8s.io/kubernetes/pkg/client/informers/informers_generated/
  internalversion/factory.go:85: \
  Failed to list *core.Endpoints: endpoints is forbidden: \
    User "system:serviceaccount:kube-system:kube-proxy" cannot \
   list endpoints at the cluster scope:\
 [clusterrole.rbac.authorization.k8s.io "system:node-proxier" not found, \
   clusterrole.rbac.authorization.k8s.io "system:basic-user" not found,
 clusterrole.rbac.authorization.k8s.io \
 "system:discovery" not found]
```

13. Check that the proxy is creating the expected rules for the problem service. Find the destination port being used for the service, **30195** in this case.

```
student@ckad-1:~$ sudo iptables-save |grep secondapp
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/secondapp:" \
-m tcp --dport 30195 -j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/secondapp:" \
-m tcp --dport 30195 -j KUBE-SVC-DAASHM5XQZF5XI3E
-A KUBE-SEP-YDKKGXN54FN2TFPE -s 192.168.55.91/32 -m comment --comment \
"default/secondapp:" -j KUBE-MARK-MASQ
-A KUBE-SEP-YDKKGXN54FN2TFPE -p tcp -m comment --comment "default/secondapp:"\
 -m tcp -j DNAT --to-destination 192.168.55.91:80
-A KUBE-SERVICES ! -s 192.168.0.0/16 -d 10.109.26.21/32 -p tcp \
-m comment --comment "default/secondapp: \
      cluster IP" -m tcp --dport 80 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.109.26.21/32 -p tcp -m comment --comment \
"default/secondapp: cluster IP" -m tcp \
      --dport 80 -j KUBE-SVC-DAASHM5XQZF5XI3E
-A KUBE-SVC-DAASHM5XQZF5XI3E -m comment --comment "default/secondapp:" \
<output_omitted>
```

14. Ensure the proxy is working by checking the port targeted by **iptables**. If it fails open a second terminal and view the proxy logs when making a request as it happens.

```
student@ckad-1:~$ curl localhost:32000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

Conformance Testing

The **cncf.io** group is in the process of formalizing what is considered to be a conforming Kubernetes cluster. While that project matures there is an existing tool provided by **Heptio** which can be useful. We will need to make sure a newer version of **Golang** is installed for it to work. You can download the code from github and look around with git or with go, depending on which tool you are most familiar.

1. Create a new directory to hold the testing code.

```
student@ckad-1:~$ mkdir test

student@ckad-1:~$ cd test/
```

2. Use **git** to download the sonobuoy code. View the resource after it downloads.

```
student@ckad-1:~/test$ git clone https://github.com/heptio/sonobuoy
Cloning into 'sonobuoy'...
remote: Counting objects: 7847, done.
remote: Compressing objects: 100% (23/23), done.
remote: Total 7847 (delta 2), reused 0 (delta 0), pack-reused 7824
Receiving objects: 100% (7847/7847), 10.19 MiB | 0 bytes/s, done.
Resolving deltas: 100% (3818/3818), done.
Checking connectivity... done.

student@ckad-1:~/test$ ls
sonobuoy

student@ckad-1:~/test$ cd sonobuoy/

student@ckad-1:~/test/sonobuoy$ ls
cmd                 Gopkg.toml                           scripts
CODE_OF_CONDUCT.md  heptio-images-ee4b0474b93e.json.enc  SUPPORT.md
```

```
CONTRIBUTING.md     LICENSE                              test
Dockerfile          main.go                              travis-deploy.sh
docs                Makefile                             vendor
examples            pkg
Gopkg.lock          README.md

student@ckad-1:~/test/sonobuoy$ less README.md
```

3. The Heptio team suggests the use of an easy to use Golang tool **gimme**. We will follow their suggestion and use it to pull their code. Start by making sure you have a `bin` directory under your `home` directory.

```
student@ckad-1:~/test/sonobuoy$ cd ; mkdir ~/bin
```

4. Use **curl** to download the binary. Note the use of **-o** as in output to save the binary to the newly created directory.

```
student@ckad-1:~$ curl -sL -o ~/bin/gimme \
https://raw.githubusercontent.com/travis-ci/gimme/master/gimme
```

5. View the file. Note it is not yet executable. Make it so.

```
student@ckad-1:~$ ls -l ~/bin/gimme
-rw-rw-r-- 1 student student 27035 Apr 15 20:46 /home/student/bin/gimme

student@ckad-1:~$ chmod +x ~/bin/gimme
```

6. Use the **gimme** tool to download the stable version of Go.

```
student@ckad-1:~$ ~/bin/gimme stable

unset GOOS;
unset GOARCH;
export GOROOT='/home/student/.gimme/versions/go1.10.1.linux.amd64';
export PATH="/home/student/.gimme/versions/go1.10.1.linux.amd64/bin:${PATH}";
go version >&2;

export GIMME_ENV="/home/student/.gimme/envs/go1.10.1.env"
```

7. Ensure the expected path has been set and exported. You can copy the second from the previous output.

```
student@ckad-1:~$ export PATH=$GOROOT/bin:$GOPATH/bin:$PATH

student@ckad-1:~$ export \
 PATH="/home/student/.gimme/versions/go1.10.1.linux.amd64/bin:${PATH}"

student@ckad-1:~$ export GOPATH=$PATH
```

8. Use the **go** command to download the sonobuoy code. You may need to install the software package, `golang-go`, if it is not already installed.

```
student@ckad-1:~$ go get -u -v github.com/heptio/sonobuoy
github.com/heptio/sonobuoy (download)
created GOPATH=/home/student/go; see 'go help gopath'
github.com/heptio/sonobuoy/pkg/buildinfo
<output_omitted>
```

9. Execute the newly downloaded tool with the run option. Review the output. Take note of interesting tests in order to search for particular output in the logs. The binary has moved over time, so you may need to use the **find** command to locate it. The long path wraps, but you would type the command as one line.

```
student@ckad-1:~/test/sonobuoy$
         ~/.gimme/versions/go1.10.3.linux.amd64/bin/bin/sonobuoy run
Running plugins: e2e, systemd-logs
INFO[0000] created object    name=heptio-sonobuoy namespace= resource=na....
INFO[0000] created object    name=sonobuoy-serviceaccount namespace=hept....
```

```
INFO[0000] created object    name=sonobuoy-serviceaccount-heptio-sonobuo...
INFO[0000] created object    name=sonobuoy-serviceaccount namespace= res....
INFO[0000] created object    name=sonobuoy-config-cm namespace=heptio-....
INFO[0000] created object    name=sonobuoy-plugins-cm namespace=hepti....
```

10. Check the status of sonobuoy. It can take up to an hour to finish on large clusters. On our two-node cluster it will take about two minutes. The path is long so is presented on the line after the prompt.

```
student@ckad-1:~/test/sonobuoy$
 ~/.gimme/versions/go1.10.3.linux.amd64/bin/bin/sonobuoy status

PLUGIN                STATUS                COUNT
e2e                running                1
systemd_logs        complete        2

Sonobuoy is still running. Runs can take up to 60 minutes.
```

11. Look at the logs. If the tests are ongoing you will see incomplete logs.

```
student@ckad-1:~/test/sonobuoy$ ~/.gimme/versions/go1.10.3.linux.amd64/bin/bin/sonobuoy logs
namespace="heptio-sonobuoy" pod="sonobuoy-systemd-logs-daemon-set-\
e322ef32b0804cd2-d48np" container="sonobuoy-worker"
time="2018-04-15T20:50:48Z" level=info msg="Waiting for waitfile" \
  waitfile=/tmp/results/done
time="2018-04-15T20:50:49Z" level=info msg="Detected done file, \
  transmitting result file"
<output_omitted>
```

12. Change into the client directory and look at the tests and results generated.

```
student@ckad-1:~/test/sonobuoy$ cd /home/student/test/sonobuoy/pkg/client/

student@ckad-1:~/test/sonobuoy/pkg/client$ ls
defaults.go  doc.go  example_interfaces_test.go  gen_test.go    logs.go      mode.go       results      run.go
delete.go    e2e.go  gen.go                      interfaces.go  logs_test.go preflight.go  retrieve.go  status.go

student@ckad-1:~/test/sonobuoy/pkg/client$ cd results/

student@ckad-1:~/test/sonobuoy/pkg/client/results$ ls
doc.go  e2e  junit_utils.go  reader.go  reader_test.go
testdata  types.go

student@ckad-1:~/test/sonobuoy/pkg/client/results$ cd testdata/
student@ckad-1:~/test/sonobuoy/pkg/client/results/testdata$ ls -l
total 644
-rw-rw-r-- 1 student student 407010 Apr 15 20:43 results-0.10.tar.gz
-rw-rw-r-- 1 student student  32588 Apr 15 20:43 results-0.8.tar.gz
-rw-rw-r-- 1 student student 215876 Apr 15 20:43 results-0.9.tar.gz

student@ckad-1:~/test/sonobuoy/pkg/client/results/testdata$ tar -xf \
    results-0.8.tar.gz

student@ckad-1:~/test/sonobuoy/pkg/client/results/testdata$ ls
config.json  hosts  plugins  resources  results-0.10.tar.gz
results-0.8.tar.gz  results-0.9.tar.gz  serverversion

student@ckad-1:~/test/sonobuoy/pkg/client/results/testdata$ less \
plugins/e2e/results/e2e.log
Dec 13 17:06:53.480: INFO: Overriding default scale value of zero to 1
Dec 13 17:06:53.481: INFO: Overriding default milliseconds value of ....
Running Suite: Kubernetes e2e suite
===================================
Random Seed: 1513184813 - Will randomize all specs
Will run 1 of 698 specs

Dec 13 17:06:54.705: INFO: >>> kubeConfig:
Dec 13 17:06:54.707: INFO: Waiting up to 4h0m0s for all (but 0) nodes...
Dec 13 17:06:54.735: INFO: Waiting up to 10m0s for all pods (need at
namespace 'kube-system' to be running a
```

```
nd ready
Dec 13 17:06:54.895: INFO: 14 / 14 pods in namespace 'kube-system'
are running and ready (0 seconds elapsed)
Dec 13 17:06:54.895: INFO: expected 3 pod replicas in namespace
'kube-system', 3 are Running and Ready.
<output_omitted>
```

13. Find other files which have been generated, and their size.

```
student@ckad-1:~/t./testdata$ find .
.
./results-0.8.tar.gz
./results-0.10.tar.gz
./hosts
./hosts/ip-10-0-9-16.us-west-2.compute.internal
<output_omitted>
```

14. Continue to look through tests and results as time permits. There is also an online, graphical scanner. In testing inside GCE the results were blocked and never returned. You may have different outcome in other environments.