LFD259

Kubernetes for Developers

Version 2023-12-13

Version 2023-12-13

Published by:

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

Nondisclosure of Confidential Information

"Confidential Information" shall not include any of the following, even if marked confidential or proprietary: (a) information that relates to the code base of any open source or open standards project (collectively, "Open Project"), including any existing or future contribution thereto; (b) information generally relating or pertaining to the formation or operation of any Open Project; or (c) information relating to general business matters involving any Open Project.

This course does not include confidential information, nor should any confidential information be divulged in class.

Contents

Chapter 2

Kubernetes Architecture

Exercise 2.1: Overview and Preliminaries

We will create a two-node **Ubuntu 20.04** cluster. Using two nodes allows an understanding of some issues and configurations found in a production environment. Currently 2 vCPU and 8G of memory allows for quick labs. As we will be working with images it is suggested that you allocate at least **20G** of disk space for the control plane (cp) node, more is better. Other Linux distributions should work in a very similar manner, but have not been tested.

Summary:

- 2 CPU
- 8G memory
- 20G+ disk on control plane node
- Don't use 192.168 network for nodes
- No firewall
- Disable swap
- Disable SELinux and AppArmor

Very Important

Regardless of the platform used (**VirtualBox**, **VMWare**, **AWS**, **GCE** or even bare metal) please remember that security software like **SELinux**, **AppArmor**, and firewall configurations 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 you're 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.

Download shell scripts and YAML files

To assist with setting up your cluster please download the tarball of shell scripts and YAML files. The `k8scp.sh` and `k8sWorker.sh` scripts deploy a Kubernetes cluster using **kubeadm** and use `Cilium` for networking. Should the file not be found you can always use a browser to investigate the parent directory.

```
$ wget https://cm.lf.training/LFD259/LFD259_V2023-12-13_SOLUTIONS.tar.xz \
             --user=LFtraining --password=Penguin2014

$ tar -xvf LFD259_V2023-12-13_SOLUTIONS.tar.xz
```

(Note: depending on your software, if you are cutting and pasting the above instructions, the underscores may disappear and be replaced by spaces, so you may have to edit the command line by hand!)

Exercise 2.2: Deploy a New Cluster

Deploy a Control Plane Node using Kubeadm

1. Log into your nodes using **PuTTY** or using **SSH** from a terminal window. Unless the instructor tells you otherwise the user name to use will be **student**. You may need to change the permissions on the `pem` (or `ppk` on windows) file as shown in the following commands. Your file name and node IP address will probably be different.

   If using PuTTY, search for instructions on using a `ppk` key to access an instance. Use the **student** username when asked by PuTTY.

   ```
   localTerm:~$ chmod 400 LF-Class.pem
   localTerm:~$ ssh -i LF-Class.pem student@WW.XX.YY.ZZ
   ```

   ```
   student@cp:~$
   ```

2. Use the **wget** and **tar** commands shown above to download and extract the course tarball to your cp node. Install **wget** if the command is not found on your instance.

3. Be sure to type commands found in the exercises, instead of trying to copy and paste. Typing will cause you to learn the commands, whereas copy and paste tends not to be absorbed. Also, use the YAML files included in the course tarball, as white space issues can slow down the labs and be discouraging.

4. Review the script to install and begin the configuration of the cp kubernetes server. You may need to change the **find** command search directory for your home directory depending on how and where you downloaded the tarball.

   A **find** command is shown if you want to locate and copy to the current directory instead of creating the file. Mark the command for reference as it may not be shown for future commands.

   ```
   student@cp:~$ find $HOME -name <YAML File>
   student@cp:~$ cp LFD259/<Some Path>/<YAML File> .
   ```

   ```
   student@cp:~$ find $HOME -name k8scp.sh
   ```

   ```
   student@cp:~$ more LFD259/SOLUTIONS/s_02/k8scp.sh
   ```

   ```
   ....
   # Install the Kubernetes software, and lock the version
   sudo apt update
   sudo apt -y install kubelet=1.28.1-00 kubeadm=1.28.1-00 kubectl=1.28.1-00
   sudo apt-mark hold kubelet kubeadm kubectl

   # Ensure Kubelet is running
   ```

```
sudo systemctl enable --now kubelet

# Disable swap just in case
sudo swapoff -a
....
```

5. 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. Use the **tee** command to save the output of the script, in case you cannot scroll back to find the `kubeadm join` in the script output. Please note the following is one command and then its output.

   Using **Ubuntu 20.04** you may be asked questions during the installation. Allow restarts (yes) and use the local, installed software if asked during the update, usually (option 2).

   Copy files to your home directory first.

```
student@cp:~$ cp LFD259/SOLUTIONS/s_02/k8scp.sh .

student@cp:~$ bash k8scp.sh | tee $HOME/cp.out
```

```
<output_omitted>

Your Kubernetes control-plane has initialized successfully!

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

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

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

  export KUBECONFIG=/etc/kubernetes/admin.conf

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

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

kubeadm join 10.128.0.93:6443 --token zu7icq.2o0ouw7s044yi9w9 \
        --discovery-token-ca-cert-hash
        ↪  sha256:a26f2166ad39b6aba0fc0dd663ac2b3455af19526bd9fe80af5439255dd
<output_omitted>

NAME        STATUS     ROLES           AGE       VERSION
cp          NotReady   control-plane   20s       v1.28.1


Script finished. Move to the next step
```

Deploy a Worker Node

6. Open a separate terminal into your **second node**, which will be your worker. Having both terminal sessions allows you to monitor the status of the cluster while adding the second node. Change the color or other characteristic of the second terminal to make it visually distinct from the first. This will keep you from running commands on the incorrect instance, which probably won't work.

   Use the previous **wget** command download the tarball to the worker node. Extract the files with **tar** as before. Find and copy the `k8sWorker.sh` file to `student`'s home directory then view it. You should see the same early steps as found in the cp setup script.

```
student@worker:~$ more k8sWorker.sh


....
# Install the Kubernetes software, and lock the version
sudo apt update
sudo apt -y install kubelet=1.28.1-00 kubeadm=1.28.1-00 kubectl=1.28.1-00
sudo apt-mark hold kubelet kubeadm kubectl

# Ensure Kubelet is running
sudo systemctl enable --now kubelet

# Disable swap just in case
sudo swapoff -a
....
```

7. Run the script on the **second node**. Again please note you may have questions during the update. Allow daemons to restart, type `yes`, and use the local installed version, usually option 2. For troubleshooting you may want to write output to a a file using the **tee** command.

```
student@worker:~$ bash k8sWorker.sh | tee worker.out
```

```
<output_omitted>
```

8. When the script is done the worker node is ready to join the cluster. The `kubeadm join` statement can be found near the end of the `kubeadm init` output on the cp node. It should also be in the file `cp.out` as well. Your nodes will use a different IP address and hashes than the example below. You'll need to pre-pend **sudo** to run the script copied from the cp node. Also note that some non-Linux operating systems and tools insert extra characters when multi-line samples are copied and pasted. Copying one line at a time solves this issue. A helpful command to find a line in a file may be **grep -A2 join cp.out** on the control plane node.

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

```
<output_omitted>
```

Configure the Control Plane Node

9. Return to the cp node. Install a text editor. While the lab uses **vim**, any text editor such as **emacs** or **nano** will work. Be aware that Windows editors may have issues with special characters. Also install the **bash-completion** package, if not already installed. Use the locally installed version of a package if asked.

```
student@cp:~$ sudo apt-get install bash-completion vim -y
```

```
<output_omitted>
```

10. 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. You may need to exit the shell and log back in for command completion to work without error.

```
student@cp:~$ source <(kubectl completion bash)

student@cp:~$ echo "source <(kubectl completion bash)" >> $HOME/.bashrc
```

11. Verify that both nodes are part of the cluster. And show a `Ready` state.

```
student@cp:~$ kubectl get node
```

```
NAME     STATUS   ROLES           AGE     VERSION
cp       Ready    control-plane   4m11s   v1.28.1
worker   Ready    <none>          61s     v1.28.1
```

12. 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@cp:~$ 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

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

13. 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@cp:~$ 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>
```

14. By default the cp 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@cp:~$  kubectl describe nodes | grep -i taint
```

```
Taints:             node-role.kubernetes.io/control-plane:NoSchedule
Taints:             <none>
```

```
student@cp:~$ kubectl taint nodes --all node-role.kubernetes.io/control-plane-
```

```
node/cp untainted
error: taint "node-role.kubernetes.io/control-plane" not found
```

15. 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 infrastructure pods to enter `Ready` state, such that the `nodes` will show a `Ready` state.

```
student@cp:~$ kubectl describe nodes | grep -i taint
```

```
Taints:             <none>
Taints:             <none>
```

```
student@cp:~$ kubectl get nodes
```

```
NAME     STATUS   ROLES           AGE     VERSION
cp       Ready    control-plane   6m1s    v1.28.1
worker   Ready    <none>          5m31s   v1.28.1
```

Exercise 2.3: 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@cp:~$ kubectl api-resources
```

```
NAME                SHORTNAMES   APIVERSION   NAMESPACED   KIND
bindings                         v1           true         Binding
componentstatuses   cs           v1           false        ComponentStatus
configmaps          cm           v1           true         ConfigMap
endpoints           ep           v1           true         Endpoints
.....
pods                po           v1           true         Pod
....
```

2. From the output we see most are `v1` which is used to denote a stable object. With that information we will add the other three required sections for pods such as `metadata`, with a `name`, and `spec` which declares which container image to use and a `name` for the container. We will create an eight line YAML file. White space and indentation matters. Don't use **Tab**s. There is a `basic.yaml` file available in the tarball, as well as `basic-later.yaml` which shows what the file will become and can be helpful for figuring out indentation.

basic.yaml

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: basicpod
5 spec:
6   containers:
7   - name: webcont
8     image: nginx
```

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

```
student@cp:~$ 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@cp:~$ kubectl get pod
```

```
NAME      READY   STATUS    RESTARTS   AGE
basicpod  1/1     Running   0          23s
```

```
student@cp:~$ 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@cp:~$ kubectl delete pod basicpod
```

```
pod "basicpod" deleted
```

```
student@cp:~$ kubectl get pod
```

```
No resources found in default namespace.
```

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@cp:~$ vim basic.yaml
```

basic.yaml

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: basicpod
5  spec:
6    containers:
7    - name: webcont
8      image: nginx
9      ports:                        #<--Add this and following line
10     - 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, as well as `NOMINATED NODE`, which is used by the scheduler and `READINESS GATES` which show if experimental features are enabled. Using **curl** and the pods IP address you should get the default nginx welcome web page.

```
student@cp:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

```
student@cp:~$ kubectl get pod -o wide
```

```
NAME     READY STATUS  RESTARTS AGE  IP           NODE
  NOMINATED NODE  READINESS GATES
basicpod 1/1   Running 0        9s   192.168.1.3  cp
  <none>          <none>
```

```
student@cp:~$ curl http://192.168.1.3
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>

<output_omitted>
```

```
student@cp:~$ 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`. Again, copy of the example from the tarball instead of typing the file by hand.

```
student@cp:~$ vim basicservice.yaml
```

basicservice.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: basicservice
5  spec:
6    selector:
7      type: webserver
8    ports:
9    - protocol: TCP
10     port: 80
```

9. We will also add a `label` to the pod and a `selector` to the service so it knows which object to communicate with.

```
student@cp:~$ vim basic.yaml
```

basic.yaml

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: basicpod
5    labels:                    #<-- Add this line
6      type: webserver          #<-- and this line which matches selector
7  spec:
8  ....
```

10. Create the new pod and service. Verify both have been created. We will learn details of the output in later chapters.

```
student@cp:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

```
student@cp:~$ kubectl create -f basicservice.yaml
```

```
service/basicservice created
```

```
student@cp:~$ kubectl get pod
```

```
NAME       READY   STATUS    RESTARTS   AGE
basicpod   1/1     Running   0          110s
```

```
student@cp:~$ kubectl get svc
```

```
NAME           TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)     AGE
basicservice   ClusterIP   10.96.112.50    <none>        80/TCP      14s
kubernetes     ClusterIP   10.96.0.1       <none>        443/TCP     4h
```

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

```
student@cp:~$ curl http://10.96.112.50
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>

<output_omitted>
```

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

```
student@cp:~$ kubectl delete svc basicservice
```

```
service "basicservice" deleted
```

```
student@cp:~$  vim basicservice.yaml
```

basicservice.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4      name: basicservice
5  spec:
6    selector:
7      type: webserver
8    type: NodePort       #<--Add this line
9    ports:
10   - protocol: TCP
11     port: 80
```

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

```
student@cp:~$  kubectl create -f basicservice.yaml
```

```
service/basicservice created
```

```
student@cp:~$ 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
```

14. 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, as reported by a **curl** to ifconfig.io. Your IP will be different. The high port will also probably be different. Note that testing from within a GCE or AWS node will not work. Use a local to you terminal or web browser to test.

```
student@cp:~$ curl ifconfig.io
```

```
35.238.3.83
```

```
local$ curl http://35.238.3.83:31514
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
<output_omitted>
```

Exercise 2.4: 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 existing pod from its own repository. The second container 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@cp:~$ vim basic.yaml
```

basic.yaml

```
1 ....
2   containers:
3   - name: webcont
4     image: nginx
5     ports:
6     - containerPort: 80
7   - name: fdlogger
8     image: fluentd
```

2. Delete and create the pod again. The commands can be typed on a single line, separated by a semicolon. 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@cp:~$ kubectl delete pod basicpod ; kubectl create -f basic.yaml
```

```
pod "basicpod" deleted
pod/basicpod created
```

```
student@cp:~$ kubectl get pod
```

```
NAME       READY   STATUS    RESTARTS   AGE
basicpod   2/2     Running   0          2m8s
```

```
student@cp:~$ kubectl describe pod basicpod
```

```
Name:               basicpod
Namespace:          default
Priority:           0
Node:               cp/10.128.0.11

....

  fdlogger:
    Container ID:   docker://f0649457217f00175ce9aec35022d0b238b9b....
    Image:          fluent/fluentd

....
```

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

```
student@cp:~$ kubectl delete pod basicpod
```

```
pod "basicpod" deleted
```

Exercise 2.5: Create a Simple Deployment

Creating a pod does not take advantage of orchestration abilities of Kubernetes. We will now create a `Deployment` which gives us 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@cp:~$  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@cp:~$ kubectl get deployment,pod
```

```
NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/firstpod     1/1     1            1           10s

NAME                               READY   STATUS    RESTARTS   AGE
pod/firstpod-65c7f8b5bb-zmlp8      1/1     Running   0          10s
```

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@cp:~$ kubectl describe deployment firstpod
```

```
Name:                   firstpod
Namespace:              default
CreationTimestamp:      Mon, 21 Aug 2023 13:48:48 +0530
Labels:                 app=firstpod
Annotations:            deployment.kubernetes.io/revision=1
Selector:               app=firstpod
Replicas:               1 desired | 1 updated | 1 total | 1 available....
StrategyType:           RollingUpdate
MinReadySeconds:        0
<output_omitted>
```

```
student@cp:~$ kubectl describe pod firstpod-65c7f8b5bb-zmlp8
```

```
Name:             firstpod-65c7f8b5bb-zmlp8
Namespace:        default
Priority:         0
Service Account:  default
Node:             worker1/192.168.100.12
Start Time:       Mon, 21 Aug 2023 13:48:48 +0530
Labels:           app=firstpod
                  pod-template-hash=65c7f8b5bb
Annotations:      <none>
Status:           Running
IP:               10.0.1.3
IPs:
  IP:           10.0.1.3
Controlled By:  ReplicaSet/firstpod-65c7f8b5bb
<output_omitted>
```

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

```
student@cp:~$ kubectl get namespaces
```

```
NAME              STATUS     AGE
default           Active     20m
kube-node-lease   Active     20m
kube-public       Active     20m
kube-system       Active     20m
```

5. There are four default namespaces. Look at the pods in the `kube-system` namespace.

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

```
NAME                              READY   STATUS    RESTARTS   AGE
cilium-cddg2                      1/1     Running   0          66m
cilium-operator-b4dfbf784-f7qtf   1/1     Running   0          66m
coredns-5dd5756b68-dhsdp          1/1     Running   0          66m
coredns-5dd5756b68-fjlcb          1/1     Running   0          66m
etcd-cp                           1/1     Running   0          67m
<output_omitted>
```

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

```
student@cp:~$ kubectl get pod -n fakenamespace
```

```
No resources found in fakenamespaces namespace.
```

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@cp:~$ kubectl get pod --all-namespaces
```

```
NAMESPACE     NAME                                READY   STATUS    RESTARTS   AGE
default       firstpod-65c7f8b5bb-zmlp8           1/1     Running   0          4m5s
kube-system   cilium-cddg2                        1/1     Running   0          75m
kube-system   cilium-operator-b4dfbf784-f7qtf     1/1     Running   0          75m
kube-system   cilium-tc7j5                        1/1     Running   0          12m
<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. Note the endpoint still exists after we deleted the pod.

```
student@cp:~$ kubectl get deploy,rs,po,svc,ep
```

```
NAME                       READY  UP-TO-DATE  AVAILABLE  AGE
deployment.apps/firstpod   1/1    1           1          4m

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

NAME                            READY  STATUS    RESTARTS   AGE
pod/firstpod-65c7f8b5bb-zmlp8   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 of age. The deployment operator started a new `ReplicaSet` operator when we deleted the existing one. The new `ReplicaSet` started another pod when the desired spec did not match the current status.

```
student@cp:~$ kubectl delete rs firstpod-6bb4574d94-rqk76
```

```
replicaset.apps "firstpod-6bb4574d94-rqk76" deleted
```

```
student@cp:~$ kubectl get deployment,rs,po,svc,ep
```

```
NAME                            READY  UP-TO-DATE AVAILABLE AGE
deployment.apps/firstpod  1/1    1          1         7m

NAME                                         DESIRED   CURRENT....
replicaset.apps/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@cp:~$ kubectl delete deployment firstpod
```

```
deployment.apps "firstpod" deleted
```

```
student@cp:~$ 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
endpoints/basicservice <none>            21m
kubernetes              10.128.0.3:6443   24m
```

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

```
student@cp:~$ kubectl delete svc basicservice
```

```
service "basicservice" deleted
```

Exercise 2.6: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

1. Using a browser go to https://www.cncf.io/certification/ckad/ and read through the program description.
2. In the **Exam Resources** section open the Curriculum Overview and Candidate Handbook in new tabs. Both of these should be read and understood prior to sitting for the exam.
3. Navigate to the Curriculum Overview tab. You should see links for domain information for various versions of the exam. Select the latest version, such as **CKAD_Curriculum_V1.25.1.pdf**. The versions you see may be different. You should see a new page showing a PDF.
4. Read through the document. Be aware that the term `Understand`, such as `Understand Services`, is more than just knowing they exist. In this case expect it to also mean create, update, and troubleshoot.
5. Locate the **Application Design and Build** section. If you review the lab, you will see we have covered some of these steps such as multi-container Pod design. Again, please note this document will change, distinct from this book. **It remains your responsibility to check for changes in the online document**. They may change on an irregular basis.

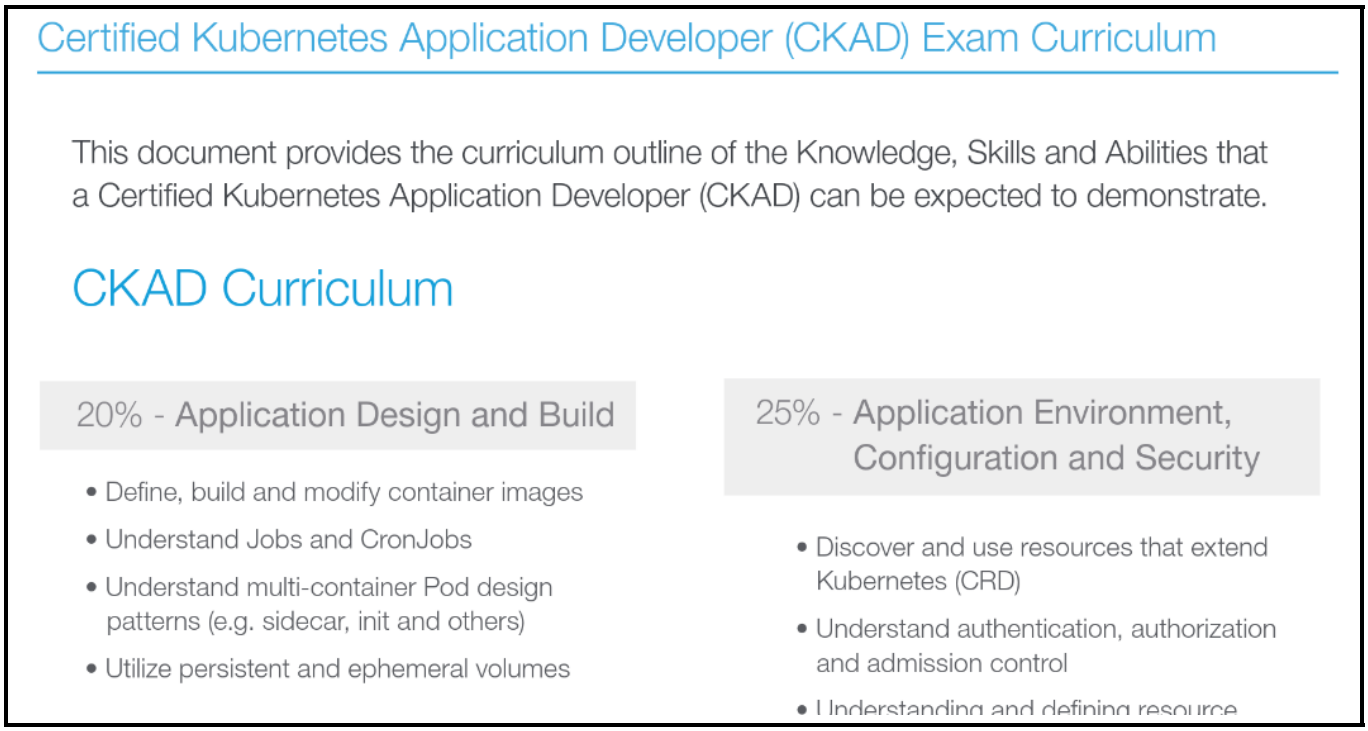

Figure 2.1: **Application Design and Build Domain**

6. Navigate to the Candidate Handbook tab. You are strongly encourage to read and understand this entire document prior to taking the exam. Again, please note this document will change, distinct from this book. **It remains your responsibility to check for changes in the online document**. They may change on an irregular and unannounced basis.
7. Find the **Important Instructions: CKA and CKAD** section in the document. Read through the section, locate current Kubernetes version for exams, such as v1.27, and find the **Resources allowed during exam** section.
8. Note the domains and subdomains you can use during the exam, with some stated conditions.

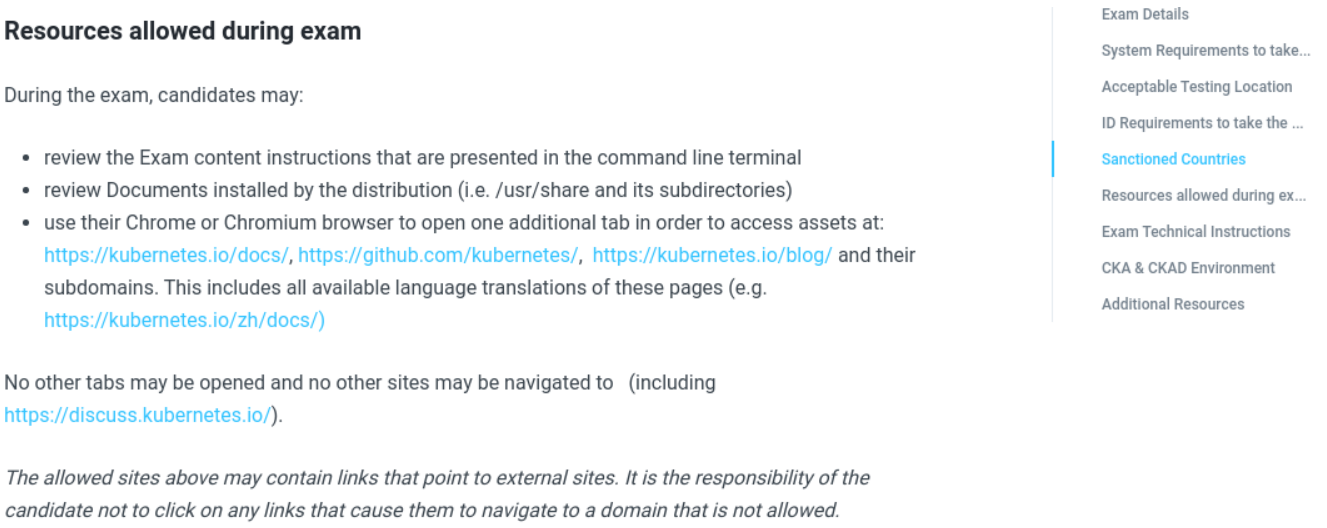

Figure 2.2: **Exam Handbook Resources Allowed**

9. Search for good YAML examples from the allowed resources. Get to know what the good page looks like. During the exam you will be using a secure browser. There should be existing bookmarks, but more than you may require. It will save time to quickly recognize the page again. Ensure it works for the version of the exam you are taking.

10. Using a timer, see how long it takes you to create and verify each of the bullet points. Try it again and see how much faster you can complete and test each step:

    - A new pod with the **nginx** image. Showing all containers running and a `Ready` status.
    - A new service exposing the pod as a **nodePort**, which presents a working webserver configured in the previous step.
    - Update the pod to run the **nginx:1.11-alpine** image and re-verify you can view the webserver via a nodePort.

11. Find and use the `architecture-review1.yaml` file included in the course tarball. Your path, such as course number, may be different than the one in the example below. Use the **find** output. Determine if the pod is running. Fix any errors you may encounter. The use of **kubectl describe** may be helpful.

```
student@cp:~$ find $HOME -name architecture-review1.yaml
```

```
<some_long_path>/architecture-review1.yaml
```

```
student@cp:~$ cp <copy-paste-from-above>  .

student@cp:~$ kubectl create -f architecture-review1.yaml
```

12. Remove any pods or services you may have created as part of the review before moving on to the next section. For example:

```
student@cp:~$ kubectl delete -f architecture-review1.yaml
```

Chapter 3

Build

Exercise 3.1: Deploy a New Application

Overview

In this lab we will deploy a very simple **Python** application, test it using **Podman**, 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.

Note that **Podman** and **crictl** use the same syntax as **docker**, so the labs `should` work the same way if you happen to be using Docker instead.

Working with A Simple Python Script

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

```
student@cp:~$ sudo apt-get -y install python3
```

```
Reading package lists... Done
Building dependency tree
Reading state information... Done
python3 is already the newest version (3.8.2-0ubuntu2).
The following package was automatically installed and is no longer required:
  libnuma1
Use 'sudo apt autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
```

2. Locate the python binary on your system.

```
student@cp:~$ which python3
```

```
/usr/bin/python3
```

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

```
student@cp:~$ mkdir app1
student@cp:~$ cd app1
student@cp:~/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, you may consider using the **find** command used before to find and copy over the file.

   While the command shows **vim** as an example other text editors such as **nano** work just as well.

```
student@cp:~/app1$ vim simple.py
```

simple.py

```
#!/usr/bin/python3
## 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@cp:~/app1$ chmod +x simple.py
student@cp:~/app1$ ./simple.py
```

```
^CTraceback (most recent call last):
  File "./simple.py", line 42, in <module>
    time.sleep(5)
KeyboardInterrupt
```

6. Verify the output has node name and timedate stamps.

```
student@cp:~/app1$ cat date.out
```

```
2022-03-22 15:51:38
cp
2022-03-22 15:51:43
cp
```

```
2022-03-22 15:51:48
cp
<output_omitted>
```

7. Create a text file named `Dockerfile`.

Very Important

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@cp:~/app1$ vim Dockerfile
```

Dockerfile

```
FROM docker.io/library/python:3
ADD simple.py /
## RUN pip install pystrich
CMD [ "python", "./simple.py" ]
```

8. As sometimes happens with open source projects the upstream version may have hiccups or not be available. When this happens we could make the tools from source code or install a binary someone else has created. For time reasons we will install an already created binary. Note the command is split on two lines. If you type it on one line there is no need for the backslash. Also notice the **cp** command and the trailing slashes, which may change the copy behaviour.

   Should the following binary not be available, you can search for an existing, and recent binary.

```
student@cp:~/app1$ curl -fsSL -o podman-linux-amd64.tar.gz \
https://github.com/mgoltzsche/podman-static/releases/latest/download/podman-linux-amd64.tar.gz

student@cp:~/app1$ tar -xf podman-linux-amd64.tar.gz

student@cp:~/app1$ sudo cp -r podman-linux-amd64/usr podman-linux-amd64/etc /
```

9. Build the container. The output below shows the end-build as necessary software was downloaded. You will need to use **sudo** in order to run this command. After the four step process completes the last lines of output should indicate success. Note the dot (.) at the end of the command indicates the current directory.

   The **podman** command has been built to replace all of the functionality of **docker**, and should accept the same syntax. As with any open source, fast changing project there `could` be slight differences. You may note the process takes almost a minute to finish, with a pause or two in output. Some may alias **docker** to **podman**.

   Choose to use the `docker.io` version of python.

```
student@cp:~/app1$ sudo podman build -t simpleapp .
```

```
STEP 1/3: FROM python:3
Resolved "python" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/python:3...
Getting image source signatures
Copying blob 0c6b8ff8c37e done
Copying blob 724cfd2dc19b done
Copying blob 808edda3c2e8 done
Copying blob 461bb1d8c517 done
Copying blob e6d3e61f7a50 done
Copying blob 412caad352a3 done
```

```
Copying blob 1bb6570cd7ac done
Copying blob aca06d6d45b1 done
Copying blob 678714351737 done
Copying config dfce7257b7 done
Writing manifest to image destination
Storing signatures
STEP 2/3: ADD simple.py /
--> 0e0bd8b3671
STEP 3/3: CMD [ "python", "./simple.py" ]
COMMIT simpleapp
--> 5b825f8d5f3
Successfully tagged localhost/simpleapp:latest
5b825f8d5f3a69660f2278a7119716e2a0fc1a5756d066800429f8030f83e978
```

10. 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@cp:~/app1$ sudo podman images
```

```
REPOSITORY                TAG         IMAGE ID      CREATED        SIZE
localhost/simpleapp       latest      11d4607c72e0  7 seconds ago  1.04 GB
docker.io/library/python  3           58a8f3dcd68a  3 days ago     1.04 GB
<output_omitted>
```

11. Use **sudo podman** to run a container using the new image. While the script is running you won't 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@cp:~/app1$ sudo podman run localhost/simpleapp
```

```
^CTraceback (most recent call last):
  File "./simple.py", line 24, in <module>
    time.sleep(5)
KeyboardInterrupt
```

12. 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/containers` when run via a podman or crio container.

```
student@cp:~/app1$ sudo find / -name date.out
```

```
/home/student/app1/date.out
/var/lib/containers/storage/overlay/
0dea104afa098f608dff06b17b2196d0ba12d09a775243781862abf016e3378a/diff/date.out
```

13. View the contents of the `date.out` file created via Podman. Note the need for **sudo** as Podman 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@cp:~/app1$ sudo tail \
 /var/lib/containers/storage/overlay/
0dea104afa098f608dff06b17b2196d0ba12d09a775243781862abf016e3378a/diff/date.out
```

```
2022-03-22 16:13:46
53e1093e5d39
2022-03-22 16:13:51
53e1093e5d39
2022-03-22 16:13:56
53e1093e5d39
```

Exercise 3.2: Configure A Local Repo

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

1. Create a simple registry using the `easyregistry.yaml` file included in the course tarball. Use the path returned by **find**, which may be different than the one found in the output below.

```
student@cp:~/app1$ find $HOME -name easyregistry.yaml
```

```
<some_long_path>/easyregistry.yaml
```

```
student@cp:~/app1$ kubectl create -f <path_from_output_above>
```

```
service/nginx created
service/registry created
deployment.apps/nginx created
persistentvolumeclaim/nginx-claim0 created
deployment.apps/registry created
persistentvolumeclaim/registry-claim0 created
persistentvolume/vol1 created
persistentvolume/vol2 created
```

2. Take note of the `ClusterIP` for the new registry service. In the example below it is `10.97.40.62`

```
student@cp:~/app1$ kubectl get svc | grep registry
```

```
registry      ClusterIP   10.97.40.62    <none>       5000/TCP,8080/TCP   5m35s
```

3. Verify the repo is working. Please 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.

   Edit the IP address to that of your `localrepo` service found in the previous command, and the listed port of `5000`

```
student@cp:~/app1$ curl 10.97.40.62:5000/v2/_catalog
```

```
{"repositories":[]}
```

4. Configure **podman** to work with non-TLS repos. The way to do this depends on the container engine in use. In our setup we will edit the `/etc/containerd/config.toml` file and add the registry we just created.

   We have included a script in the tar ball **local-repo-setup.sh** to setup local repository.

```
student@cp:~/app1$ find $HOME -name local-repo-setup.sh
student@cp:~/app1$ cp /home/student/LFD259/SOLUTIONS/s_03/local-repo-setup.sh $HOME
student@cp:~/app1$ chmod +x $HOME/local-repo-setup.sh
student@cp:~/app1$ . $HOME/local-repo-setup.sh       # dot <space> scriptname to set correct variables
```

```
. local-repo-setup.sh
Configuring local repo, Please standby
[[registry]]
location = "10.97.40.62:5000"
insecure = true
[plugins."io.containerd.grpc.v1.cri".registry.mirrors."*"]
endpoint = ["http://10.97.40.62:5000"]

Local Repo configured, follow the next steps
```

5. Download and tag a typical image from hub.docker.com. Tag the image using the IP and port of the registry, via the $repo variable.

```
student@cp:~app1/$ sudo podman pull docker.io/library/alpine
```

```
Resolved "alpine" as an alias (/etc/containers/registries.conf.d/000-shortnames.conf)
Trying to pull docker.io/library/alpine:latest...
Getting image source signatures
Copying blob 540db60ca938 done
Copying config 6dbb9cc540 done
Writing manifest to image destination
Storing signatures
6dbb9cc54074106d46d4ccb330f2a40a682d49dda5f4844962b7dce9fe44aaec
```

```
student@cp:~/app1$ sudo podman tag alpine $repo/tagtest
```

6. 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/containers/registry.conf file correctly and restarted the service.

```
student@cp:~/app1$ sudo podman push $repo/tagtest
```

```
Getting image source signatures
Copying blob b2d5eeeaba3a done
Copying config 6dbb9cc540 done
Writing manifest to image destination
Storing signatures
```

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

```
student@cp:~/app1$ sudo podman image rm alpine
```

```
Untagged: docker.io/library/alpine:latest
```

```
student@cp:~/app1$ sudo podman image rm $repo/tagtest
```

```
Untagged: 10.97.40.62:5000/tagtest:latest
Deleted: 6dbb9cc54074106d46d4ccb330f2a40a682d49dda5f4844962b7dce9fe44aaec
```

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

```
student@cp:~/app1$ sudo podman pull $repo/tagtest
```

```
Trying to pull 10.97.40.62:5000/tagtest:latest...
Getting image source signatures
Copying blob 74782b667c7d done
Copying config 6dbb9cc540 done
Writing manifest to image destination
Storing signatures
6dbb9cc54074106d46d4ccb330f2a40a682d49dda5f4844962b7dce9fe44aaec
```

9. Configure the worker (second) node to use the registry running on the cp server. Connect to the worker node.

```
student@worker:~$ find $HOME -name local-repo-setup.sh
student@worker:~$ cp /home/student/LFD259/SOLUTIONS/s_03/local-repo-setup.sh $HOME
student@worker:~$ chmod +x $HOME/local-repo-setup.sh
student@worker:~$ . $HOME/local-repo-setup.sh       # dot <space> scriptname to set correct variables
student@worker:~$ sudo podman pull $repo/tagtest # if needed export repo=10.97.40.62:5000
```

```
. local-repo-setup.sh
Configuring local repo, Please standby
[[registry]]
location = "10.97.40.62:5000"
insecure = true
[plugins."io.containerd.grpc.v1.cri".registry.mirrors."*"]
endpoint = ["http://10.97.40.62:5000"]

Local Repo configured, follow the next steps


Trying to pull 10.97.40.62:5000/tagtest:latest...
Getting image source signatures
Getting image source signatures
Copying blob 74782b667c7d done
Copying config 6dbb9cc540 done
Writing manifest to image destination
Storing signatures
a313e647ae05302fc54b57065c250ff58d2f580fab7b7625949651b141d9fca7
```

10. Now that we know **podman** on all nodes can use the repository we need to make sure Kubernetes knows about the new repository and settings as well. The simplest way is to reboot every node. Log back in after the connection closes.

```
student@cp:~$ sudo reboot

student@worker:~$ sudo reboot
```

11. Test that the repo works after the reboot, as good admins we want to know our configuration is persistent. Be aware it can take a minute or two after reboot for the kube-apiserver to fully start. If the `$repo` variable isn't set check the source statement in `.bashrc`.

```
student@cp:~$ curl $repo/v2/_catalog
```

```
{"repositories":[]}
```

12. Use **podman 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@cp:~/app1$ sudo podman tag simpleapp $repo/simpleapp

student@cp:~/app1$ sudo podman push $repo/simpleapp
```

```
Getting image source signatures
Copying blob 47458fb45d99 done
Copying blob a3c1026c6bcc done
Copying blob f1d420c2af1a done
Copying blob 461719022993 done
Copying blob d35c5bda4793 done
Copying blob 46829331b1e4 done
Copying blob ceee8816bb96 done
Copying blob da7b0a80a4f2 done
Copying blob e571d2d3c73c done
Copying blob 5c2db76bc949 done
Copying config a313e647ae done
Writing manifest to image destination
Storing signatures
```

13. Test that the image can be found in the repository, from both the cp and the worker

```
student@cp:~$ curl $repo/v2/_catalog
```

```
{"repositories":["simpleapp"]}
```

14. Return to the cp node and deploy the `simpleapp` in Kubernetes with several replicas. We will name the deployment `try1`. Scale to have six replicas. Increase the replica count until pods are deployed on both cp and worker.

```
student@cp:~$ kubectl create deployment try1 --image=$repo/simpleapp
```

```
deployment.apps/try1 created
```

```
student@cp:~$  kubectl scale deployment try1 --replicas=6
```

```
deployment.apps/try1 scaled
```

```
student@cp:~$  kubectl get pod -o wide
```

```
NAME                      READY   STATUS    RESTARTS   AGE     IP               NODE  ....
basicpod                  1/1     Running   0          20m     192.168.95.15    worker ....
registry-ddf6bcb7c-b8wcr  1/1     Running   0          16m     192.168.95.16    worker ....
try1-55f675ddd-28vgs      1/1     Running   0          17s     192.168.95.30    worker ....
try1-55f675ddd-2nrsj      1/1     Running   0          17s     192.168.95.26    worker ....
try1-55f675ddd-4vzt7      1/1     Running   0          17s     192.168.219.100  cp ....
<output_omitted>
```

15. On the `second node` use **sudo crictl ps** to verify containers of `simpleapp` are running. Even though **podman** has the options, cri-o is running the containers on our behalf. The scheduler will usually balance pod count across nodes. As the cp already has several pods running the new pods may be on the worker, so the number of pods returned will vary.

    You may need to configure **crictl** first to avoid error messages. You can omit the backslash and type the command on one line.

```
student@worker:~$ sudo crictl config \
  --set runtime-endpoint=unix:///run/containerd/containerd.sock \
  --set image-endpoint=unix:///run/containerd/containerd.sock
```

```
student@worker:~$ sudo crictl ps | grep simple
```

```
855750df82cd4       10.97.177.111:5000/
simpleapp@sha256:f7fc297b3c20c47c506896ee13f4d7f13b4c4c0ee0fafd2ab88c9809b2a1aed6   15 minutes ago
Running             simpleapp                 0                   6e31ba7491117
237af1260265a       10.97.177.111:5000/
simpleapp@sha256:f7fc297b3c20c47c506896ee13f4d7f13b4c4c0ee0fafd2ab88c9809b2a1aed6   15 minutes ago
Running             simpleapp                 0                   bc0d63564cf49
d0ae2910cf8b8       10.97.177.111:5000/
simpleapp@sha256:f7fc297b3c20c47c506896ee13f4d7f13b4c4c0ee0fafd2ab88c9809b2a1aed6   15 minutes ago
Running             simpleapp                 0                   b9bb90e04bec1
046a0c439fa43       10.97.177.111:5000/
simpleapp@sha256:f7fc297b3c20c47c506896ee13f4d7f13b4c4c0ee0fafd2ab88c9809b2a1aed6   15 minutes ago
Running             simpleapp                 0                   fe464425e41b1
```

16. Return to the `cp node`. Save the `try1` deployment as YAML.

```
student@cp:~/app1$ cd $HOME/app1/
student@cp:~/app1$ kubectl get deployment try1 -o yaml > simpleapp.yaml
```

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

```
student@cp:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@cp:~/app1$ kubectl create -f simpleapp.yaml
```

```
deployment.apps/try1 created
```

```
student@cp:~/app1$ kubectl get deployment
```

```
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
nginx      1/1     1            1           15m
registry   1/1     1            1           15m
try1       6/6     6            6           5s
```

Exercise 3.3: 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.

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 399 range, and the third probe uses a TCP socket. In this example we'll 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/ An edited file is also included in the tarball, but requires the image name to be edited to match your registry IP address.

```
student@cp:~/app1$ vim simpleapp.yaml
```

simpleapp.yaml

```
....
    spec:
      containers:
      - image: 10.111.235.60:5000/simpleapp
        imagePullPolicy: Always
        name: simpleapp
        readinessProbe:          #<--This line and next five
          periodSeconds: 5
          exec:
            command:
            - cat
            - /tmp/healthy
        resources: {}
....
```

2. Delete and recreate the `try1` deployment.

```
student@cp:~/app1$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@cp:~/app1$ kubectl create -f simpleapp.yaml
```

```
deployment.apps/try1 created
```

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

```
student@cp:~/app1$ kubectl get deployment
```

```
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
nginx      1/1     1            1           19m
registry   1/1     1            1           19m
try1       0/6     6            0           15s
```

4. Take a closer look at the pods. Use **describe pod** and **logs** to investigate issues, note there may be no logs. Choose one of the `try1` pods as a test to create the health check file.

```
student@cp:~/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@cp:~/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@cp:~/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@cp:~$ 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@cp:~/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@cp:~/app1$ vim simpleapp.yaml
```

YAML **simpleapp.yaml**

```
....
          terminationMessagePath: /dev/termination-log
          terminationMessagePolicy: File
       - name: goproxy                 #<-- Indented 7 spaces, add lines from here...
         image: registry.k8s.io/goproxy:0.1
         ports:
         - containerPort: 8080
         readinessProbe:
           tcpSocket:
             port: 8080
           initialDelaySeconds: 5
           periodSeconds: 10
         livenessProbe:                #<-- This line is 9 spaces indented, fyi
           tcpSocket:
             port: 8080
           initialDelaySeconds: 15
           periodSeconds: 20           #<-- ....to here
       dnsPolicy: ClusterFirst
       restartPolicy: Always
....
```

10. Delete and recreate the deployment.

```
student@cp:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@cp:~$ kubectl create -f simpleapp.yaml
```

```
deployment.apps/try1 created
```

11. View the newly created pods. You'll 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@cp:~$ 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, this setup will leverage labels so you don't have to look up the 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@cp:~$ for name in $(kubectl get pod -l app=try1 -o name)
> do
> kubectl exec $name -c simpleapp -- touch /tmp/healthy
> done
```

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@cp:~$ 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, but not a change in status or the probe success.

```
student@cp:~/app1$ kubectl describe pod try1-76cc5ffcc6-tx4dz | tail
```

```
Events:
  Type     Reason     Age                  From               Message
  ----     ------     ----                 ----               -------
  Normal   Scheduled  7m46s                default-scheduler  Successfully assigned
     default/try1-754bf9c75b-vx58x to cp
  Normal   Pulling    7m44s                kubelet            Pulling image
     "10.97.177.111:5000/simpleapp"
  Normal   Pulled     7m44s                kubelet            Successfully pulled image
     "10.97.177.111:5000/simpleapp" in 96.710062ms
  Normal   Created    7m43s                kubelet            Created container simpleapp
  Normal   Started    7m43s                kubelet            Started container simpleapp
  Warning  Unhealthy  7m6s (x9 over 7m42s)  kubelet            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@cp:~/app1$ kubectl describe pod try1-76cc5ffcc6-tx4dz | grep -E 'State|Ready'
```

```
    State:          Running
    Ready:          True
    State:          Running
    Ready:          True
  Ready             True
  ContainersReady   True
```

Exercise 3.4: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.

- Implement probes and health checks
- Define, build and modify container images
- Understand multi-container Pod design patterns
- Utilize container logs

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of YAML samples and can complete each step quickly.

The domain review items `may be vague or non-detailed`, just like what one would experience in a work or exam environment.

1. Using the three URL locations allowed by the exam, find and bookmark working YAML examples for LivenessProbes, ReadinessProbes, and multi-container pods.
2. Deploy a new nginx webserver. Add a LivenessProbe and a ReadinessProbe on port 80. Test that both probes and the webserver work.
3. Use the `build-review1.yaml` file to create a non-working deployment. Fix the deployment such that both containers are running and in a `READY` state. The web server listens on port 80, and the proxy listens on port 8080.

4. View the default page of the web server. When successful verify the `GET` activity logs in the container log. The message should look something like the following. Your time and IP may be different.

```
192.168.124.0 - - [30/Jan/2020:03:30:31 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```

5. Remove any resources created in this review.

Chapter 4

Design

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 don't honor security configurations such as network policies. Should you design a deployment which and use a network policy there wouldn't be an error; the policy would have no effect. While developers typically wouldn't care much about the mechanics of it can affect the availability of features and troubleshooting of newly decoupled microservices.

While still new, the community is moving towards the **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 Cilium**. Classic and external modes are also possible. Several software defined network projects intended for Kubernetes have been created recently, with new features added regularly. Be aware that **Cilium** is a dynamic project with ongoing and frequent changes.

Quick Understanding of Network Plugins

While developers don't need to configure cluster networking, they may need to understand the plugin in use, and it's particular features and quirks. This section is to ensure you have made a quick review of the most commonly used plugins, and where to find more information as necessary.

1. Verify your nodes are using a CNI plugin. Read through the startup process of CNI. Each message begins with a type of message and a time stamp, followed by the details of the message. Use **TAB** to complete for your node name. Examine both the controller and one of your proxy pods.

```
student@cp:~$ kubectl -n kube-system logs kube-controller-manager-<TAB>
```

```
I0216 17:06:11.729548       1 serving.go:348] Generated self-signed cert in-memory
I0216 17:06:12.318263       1 controllermanager.go:196] Version: v1.26.1
I0216 17:06:12.321279       1 dynamic_cafile_content.go:156] "Starting controller" name=....
I0216 17:06:12.321311       1 secure_serving.go:200] Serving securely on 127.0.0.1:10257
....
```

```
student@cp:~$ kubectl -n kube-system logs kube-proxy-<TAB>
```

```
I0216 17:06:19.299029       1 node.go:163] Successfully retrieved node IP: 10.128.0.67
I0216 17:06:19.299337       1 server_others.go:138] "Detected node IP" address="10.128.0.67"
I0216 17:06:19.299515       1 server_others.go:561] "Unknown proxy mode, assuming iptables proxy"
↪  proxyMode=""
```

2. 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
   - **Flannel**
     https://github.com/coreos/flannel
   - **Cilium**
     http://cilium.io/
   - **Kube Router**
     https://www.kube-router.io

3. Which of the plugins allow vxlans?
4. Which are layer 2 plugins?
5. Which are layer 3?
6. Which allow network policies?
7. Which can encrypt all TCP and UDP traffic?

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?

Do you really know?

When and why would you use a multi-container pod?

Have you found a YAML example online?

Go back and review multi-container pod types and content on decoupling if you can't easily answer these questions.

We touched on adding a second logging and a readiness container in a previous chapter and will work more with logging a future exercise.

Solution 4.1

Plugin Answers

1. Which of the plugins allow vxlans?
   Canal, Project Calico, Flannel, Weave Net, Cilium
2. Which are layer 2 plugins?
   Canal, Flannel, Weave Net
3. Which are layer 3?
   Project Calico, Romana, Kube Router
4. Which allow network policies?
   Project Calico, Canal, Kube Router, Weave Net, Cilium
5. Which can encrypt all TCP and UDP traffic?
   Project Calico, Weave Net, Cilium

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: Create a Job

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`

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

```
student@cp:~$ vim job.yaml
```

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@cp:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@cp:~$ kubectl get job
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   0/1           3s         3s
```

```
student@cp:~$ 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:     Sun, 03 Nov 2022  04:22:50 +0000
Completed At:   Sun, 03 Nov 2022  04:22:55 +0000
Duration:       5s
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
<output_omitted>
```

```
student@cp:~$ 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@cp:~$ 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@cp:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

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

```
student@cp:~$ vim job.yaml
```

job.yaml

```
1 <output_omitted>
2 metadata:
3   name: sleepy
4 spec:
5   completions: 5   #<--Add this line
6   template:
7     spec:
8       containers:
9 <output_omitted>
```

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

```
student@cp:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@cp:~$ 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@cp:~$ 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@cp:~$ kubectl get jobs
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   5/5           26s        10m
```

```
student@cp:~$ 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@cp:~$ vim job.yaml
```

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@cp:~$ kubectl create -f job.yaml
```

```
student@cp:~$ 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@cp:~$ 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@cp:~$ vim job.yaml
```

job.yaml

```
<output_omitted>
  completions: 5
  parallelism: 2
  activeDeadlineSeconds: 15   #<-- Add this line
  template:
    spec:
      containers:
      - name: resting
        image: busybox
        command: ["/bin/sleep"]
```

```
11          args: ["3"]
12  <output_omitted>
```

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

```
student@cp:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

```
student@cp:~$ kubectl create -f job.yaml
```

```
job.batch/sleepy created
```

```
student@cp:~$ kubectl get jobs
```

```
NAME     COMPLETIONS   DURATION   AGE
sleepy   2/5           6s         6s
```

```
student@cp:~$ 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. You may see less `status` if the job has yet to run. Wait and try again, if so.

```
student@cp:~$ kubectl get job sleepy -o yaml
```

```
<output_omitted>
status:
  conditions:
  - lastProbeTime: "2022-11-03T16:06:10Z"
    lastTransitionTime: "2022-11-03T16:06:10Z"
    message: Job was active longer than specified deadline
    reason: DeadlineExceeded
    status: "True"
    type: Failed
  failed: 1
  startTime: "2022-11-03T16:05:55Z"
  succeeded: 4
```

14. Delete the job.

```
student@cp:~$ kubectl delete jobs.batch sleepy
```

```
job.batch "sleepy" deleted
```

Exercise 4.3: Designing Applications With Duration: 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@cp:~$ cp job.yaml cronjob.yaml
```

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

```
student@cp:~$ vim cronjob.yaml
```

cronjob.yaml

```
1  apiVersion: batch/v1
2  kind: CronJob   #<-- Change this line
3  metadata:
4    name: sleepy
5  spec:                          #<-- Remove completions:, parallelism:, and activeDeadlineSeconds:
6    schedule: "*/2 * * * *"      #<-- Add Linux style cronjob syntax
7    jobTemplate:                 #<-- New jobTemplate and spec
8      spec:
9        template:   #<-- This and following lines space four to right
10         spec:
11           containers:
12           - name: resting
13             image: busybox
14             command: ["/bin/sleep"]
15             args: ["3"]
16           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@cp:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@cp:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        <none>          8s
```

```
student@cp:~$ kubectl get job
```

```
No resources found in default namespace.
```

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

```
student@cp:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     0        21s             2m1s
```

```
student@cp:~$ kubectl get jobs.batch
```

```
NAME                COMPLETIONS   DURATION   AGE
sleepy-1539722040   1/1           5s         18s
```

```
student@cp:~$ 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@cp:~$ vim cronjob.yaml
```

cronjob.yaml

```
....
  jobTemplate:
    spec:
      template:
        spec:
          activeDeadlineSeconds: 10  #<-- Add this line
          containers:
          - name: resting
....
            command: ["/bin/sleep"]
            args: ["30"]             #<-- Edit this line
          restartPolicy: Never
```

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@cp:~$ kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

```
student@cp:~$ kubectl create -f cronjob.yaml
```

```
cronjob.batch/sleepy created
```

```
student@cp:~$ sleep 120 ; kubectl get jobs
```

```
NAME               COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           61s        61s
```

```
student@cp:~$ kubectl get cronjobs.batch
```

```
NAME     SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
sleepy   */2 * * * *   False     1        72s             94s
```

```
student@cp:~$ kubectl get jobs
```

```
NAME               COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           75s        75s
```

```
student@cp:~$ kubectl get jobs
```

```
NAME               COMPLETIONS   DURATION   AGE
sleepy-1539723240   0/1           2m19s      2m19s
sleepy-1539723360   0/1           19s        19s
```

```
student@cp:~$ 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@cp:~$  kubectl delete cronjobs.batch sleepy
```

```
cronjob.batch "sleepy" deleted
```

Exercise 4.4: Using Labels

Create and work with labels. We will understand how the deployment, replicaSet, and pod labels interact.

1. Create a new deployment called `design2`

```
student@cp:~$ kubectl create deployment design2 --image=nginx
```

```
deployment.apps/design2 created
```

2. View the wide **kubectl get** output for the `design2` deployment and make note of the `SELECTOR`

```
student@cp:~$ kubectl get deployments.apps design2 -o wide
```

```
NAME      READY  UP-TO-DATE  AVAILABLE  AGE    CONTAINERS  IMAGES  SELECTOR
design2   1/1    1           1          2m13s  nginx       nginx   app=design2
```

3. Use the **-l** option to use the selector to list the pods running inside the deployment. There should be only one pod running.

```
student@cp:~$ kubectl get -l app=design2 pod
```

```
NAME                       READY   STATUS    RESTARTS   AGE
design2-766d48574f-5w274   1/1     Running   0          3m1s
```

4. View the pod details in YAML format using the deployment selector. This time use the **–selector** option. Find the pod label in the output. It should match that of the deployment.

```
student@cp:~$ kubectl get --selector app=design2 pod  -o yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2023-12-11T16:29:37Z"
  generateName: design2-766d48574f-
  labels:
    app: design2
    pod-template-hash: 766d48574f
....
```

5. Edit the pod label to be your favorite color.

```
student@cp:~$ kubectl edit pod design2-766d48574f-5w274
```

YAML

```
1 ....
2   labels:
3     app: orange                       #<<-- Edit this line
4     pod-template-hash: 766d48574f
5   name: design2-766d48574f-5w274
6 ....
```

6. Now view how many pods are in the deployment. Then how many have `design2` in their name. Note the `AGE` of the pods.

```
student@cp:~$ kubectl get deployments.apps design2 -o wide
```

```
NAME      READY  UP-TO-DATE  AVAILABLE  AGE  CONTAINERS  IMAGES  SELECTOR
design2   1/1    1           1          56s  nginx       nginx   app=design2
```

```
student@cp:~$ kubectl get pods | grep design2
```

```
design2-766d48574f-5w274   1/1     Running            0          82s
design2-766d48574f-xttgg   1/1     Running            0          2m12s
```

7. Delete the `design2` deployment.

```
student@cp:~$ kubectl delete deploy design2
```

```
deployment.apps "design2" deleted
```

8. Check again for pods with `design2` in their names. You should find one pod, with an `AGE` of when you first created the deployment. Once the label was edited the deployment created a new pod in order that the status matches the spec and there be a replica running with the intended label.

```
student@cp:~$ kubectl get pods | grep design2
```

```
design2-766d48574f-5w274  1/1    Running           0         38m
```

9. Delete the pod using the **-l** and the label you edited to be your favorite color in a previous step. The command details have been omitted. Use previous steps to figure out these commands.

Exercise 4.5: Setting Pod Resource Limits and Requirements

1. Create a new pod running the `vish/stress` image. A YAML `stress.yaml` file has been included in the course tarball.
2. Run the **top** command on the cp and worker nodes. You should find a `stress` command consuming the majority of the CPU on one node, the `worker`. Use **q** to exit from top. Delete the deployment.
3. Edit the `stress.yaml` file add in the following limits and requests.

```
student@cp:~$
```

```
....
        name: stressmeout
        resources:                  #<<-- Add this and following six lines
          limits:
            cpu: "1"
            memory: "1Gi"
          requests:
            cpu: "0.5"
            memory: "500Mi"
        args:
        - -cpus
 ....
```

4. Create the deployment again. Check the status of the pod. You should see that it shows an `OOMKilled` status and a growing number of restarts. You may see a status of `Running` if you catch the pod in early in a restart. If you wait long enough you may see `CrashLoopBackOff`.

```
student@cp:~$ kubectl get pod stressmeout-7fbbbcc887-v9kvb
```

```
NAME                           READY   STATUS      RESTARTS   AGE
stressmeout-7fbbbcc887-v9kvb   0/1     OOMKilled   2          32s
```

5. Delete then edit the deployment. Change the `limit:` parameters such that pod is able to run, but not too much extra resources. Try setting the memory limit to exactly what the stress command requests.

   As we allow the pod to run on the cp node, this could cause issues, such as the `kube-apiserver` restarting due to lack of resources. We will also add a `nodeSelector` to use the built in label of `kubernetes.io/hostname`.

```
student@cp:~$ kubectl delete -f stress.yaml

student@cp:~$ vim stress.yaml
```

```
....
    spec:
      nodeSelector:                     #<-- Uncomment and edit
        kubernetes.io/hostname: worker  #<-- to by YOUR worker hostname
      containers:

....
        resources:
          limits:
            cpu: "2"
            memory: "2Gi"
          requests:
....
```

6. Create the deployment and ensure the pod runs without error. Use **top** to verify the stress command is running on one of the nodes and view the pod details to ensure the CPU and memory limits are in use. Also use the **kubectl describe node** command to view the resources your cp and worker node are using. The command details have been omitted. Use previous steps to figure out the commands.

7. Change `limits` and `requests` to numbers higher than your node resources, and evaluate how the container and pod is handled. It may take a while for resources to be fully allocated.

8. Remove the `stressmeout` deployment when done.

```
student@cp:~$ kubectl delete deploy stressmeout
```

```
deployment.apps "stressmeout" deleted
```

Exercise 4.6: Simple initContainer

1. As we have already learned about creating pods, this exercise does not include the particular steps. You can use this to determine if you need to review previous content. Use the following YAML, which is also in the course tarball, to create a pod with an initContainer. It will fail, view what it looks like and investigate the errors. Change the command from /**bin/false** to a successful command such as /**bin/true** and create the pod again.

init-tester.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: init-tester
  labels:
    app: inittest
spec:
```

```yaml
  containers:
  - name: webservice
    image: nginx
  initContainers:
  - name: failed
    image: busybox
    command: [/bin/false]
```

Exercise 4.7: Exploring Custom Resource Definitions

1. View CRDs currently in the cluster

```
student@cp:~$ kubectl get crd
```

```
          NAME                                        CREATED AT
ciliumcidrgroups.cilium.io                    2023-08-21T07:08:59Z
ciliumclusterwidenetworkpolicies.cilium.io    2023-08-21T07:09:02Z
ciliumendpoints.cilium.io                     2023-08-21T07:08:59Z
ciliumexternalworkloads.cilium.io             2023-08-21T07:08:58Z
ciliumidentities.cilium.io                    2023-08-21T07:08:58Z
ciliuml2announcementpolicies.cilium.io        2023-08-21T07:08:59Z
ciliumloadbalancerippools.cilium.io           2023-08-21T07:08:59Z
ciliumnetworkpolicies.cilium.io               2023-08-21T07:09:02Z
ciliumnodeconfigs.cilium.io                   2023-08-21T07:08:58Z
ciliumnodes.cilium.io                         2023-08-21T07:09:00Z
ciliumpodippools.cilium.io                    2023-08-21T07:08:59Z
```

2. Take a detailed look at the IPPools CRD.

```
student@cp:~$ kubectl get crd ciliumpodippools.cilium.io -o yaml
```

```
....
spec:
  conversion:
    strategy: None
  group: cilium.io
  names:
    kind: CiliumPodIPPool
    listKind: CiliumPodIPPoolList
    plural: ciliumpodippools
    shortNames:
    - cpip
    singular: ciliumpodippool
  scope: Cluster
  versions:
  - name: v2alpha1
....
```

3. View if this newly declared resource exists.

```
student@cp:~$ kubectl get CiliumPodIPPool
```

```
No resources found
```

4. Take a detailed view of the other crds cluster is using.

Exercise 4.8: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter. They may be in multiple sections. The items below shows the topics covered in this chapter.

- Understand Jobs and CronJobs
- Understanding and defining resources requirements, limits and quotas
- Understand multi-container Pod design patterns (e.g. sidecar **init**, and others)
- Discover and use resources that extend Kubernetes (CRD).

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

1. Find and use the `design-review1.yaml` file to create a pod.
2. Determine the CPU and memory resource requirements of `design-pod1`.
3. Edit the pod resource requirements such that the CPU limit is exactly twice the amount requested by the container. (Hint: subtract .22)
4. Increase the memory resource limit of the pod until the pod shows a `Running` status. This may require multiple edits and attempts. Determine the minimum amount necessary for the `Running` status to persist at least a minute.
5. Use the `design-review2.yaml` file to create several pods with various labels.
6. Using **only** the - -selector value `tux` to delete only those pods. This should be half of the pods. Hint, you will need to view pod settings to determine the key value as well.
7. Create a new cronjob which runs `busybox` and the `sleep 30` command. Have the cronjob run every three minutes. View the job status to check your work. Change the settings so the pod runs 10 minutes from the current time, every week. For example, if the current time was 2:14PM, I would configure the job to run at 2:24PM, every Monday.
8. Delete any objects created during this review. You may want to delete all but the cronjob if you'd like to see if it runs in 10 minutes. Then delete that object as well.

Chapter 5

Deployment Configuration

Exercise 5.1: Configure the Deployment: ConfigMaps

Very Important

Save a copy of your `$HOME/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@cp:~$ cp $HOME/app1/simpleapp.yaml $HOME/beforeLab5.yaml
```

We will cover the use of **secrets** in the `Security` chapter lab.

Overview

In this lab we will add resources to our deployment with further configuration you may need for production.

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@cp:~/app1$ cd

student@cp:~$ mkdir primary
student@cp:~$ echo c > primary/cyan
student@cp:~$ echo m > primary/magenta
student@cp:~$ echo y > primary/yellow
student@cp:~$ echo k > primary/black
student@cp:~$ echo "known as key" >> primary/black
student@cp:~$ echo blue > favorite
```

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

```
student@cp:~$ 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@cp:~$ kubectl get configmap colors
```

```
NAME      DATA      AGE
colors    6         11s
```

```
student@cp:~$ 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:
<output_omitted>
```

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@cp:~$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```
1  ....
2      spec:
3        containers:
4        - image: 10.105.119.236:5000/simpleapp
5          env:                                  #<-- Add from here
6          - name: ilike
7            valueFrom:
8              configMapKeyRef:
9                name: colors
10               key: favorite                   #<-- to here
11         imagePullPolicy: Always
12 ....
```

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

```
student@cp-lab-7xtx:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@cp-lab-7xtx:~$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/try1 created
```

6. Even though the `try1` pod is not in a fully ready state, it is running and useful. Use **kubectl exec** to view a variable's value. View the pod state then verify you can see the `ilike` value within the `simpleapp` container. Note that the use of double dash (- -) tells the shell to pass the following as standard in.

```
student@cp:~$ kubectl get pod
```

```
<output_omitted>
```

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

```
blue
```

7. Edit the YAML file again, this time adding the another 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@cp:~$ vim $HOME/app1/simpleapp.yaml
```

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@cp:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@cp:~$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/try1 created
```

```
student@cp:~$ 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@cp:~$ kubectl exec -it try1-d4fbf76fd-46pkb -- /bin/bash -c 'env'
```

```
Defaulting container name to simpleapp.
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@cp:~$ vim car-map.yaml
```

car-map.yaml

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: fast-car
5   namespace: default
6 data:
7   car.make: Ford
8   car.model: Mustang
9   car.trim: Shelby
```

```
student@cp:~$ 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@cp:~$ kubectl get configmap fast-car -o yaml
```

```
apiVersion: v1
data:
  car.make: Ford
  car.model: Mustang
  car.trim: Shelby
kind: ConfigMap
metadata:
<output_omitted>
```

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 come after the containers have been declared, the example below has the volume added just before the `status`: output..

```
student@cp:~$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```
....
    spec:
      containers:
      - image: 10.105.119.236:5000/simpleapp
        volumeMounts:             #<-- Add this and following two lines
        - mountPath: /etc/cars
          name: car-vol
        env:
        - name: ilike
....
      securityContext: {}
      terminationGracePeriodSeconds: 30
      volumes:                    #<-- Add this and following four lines
      - name: car-vol
        configMap:
          defaultMode: 420
          name: fast-car
status:
....
```

13. Delete and recreate the deployment.

```
student@cp:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@cp:~$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/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@cp:~$ kubectl get deployment
```

```
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
nginx      1/1     1            1           1d
registry   1/1     1            1           1d
try1       0/6     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. Be sure you edit the `simpleapp` section, not the `goproxy` section.

```
student@cp:~$ kubectl delete deployment try1
```

```
deployment.apps "try1" deleted
```

```
student@cp:~$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```
....
        readinessProbe:
          exec:
            command:
            - ls                        #<-- Add/Edit this and following line.
            - /etc/cars
          periodSeconds: 5
....
```

```
student@cp:~$ kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps/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@cp:~$ kubectl get deployment
```

```
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
nginx      1/1     1            1           1d
registry   1/1     1            1           1d
try1       6/6     6            6           1m
```

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

```
Shelby student@cp:~$
```

Exercise 5.2: Configure the Deployment: 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. Search for pv and pvc YAML example files on http://kubernetes.io/docs and http://kubernetes.io/blog.
2. Use the CreateNFS.sh script from the tarball to set up NFS on your cp node. This script will configure the server, export /opt/sfw and create a file /opt/sfw/hello.txt. Use the **find** command to locate the file if you don't remember where you extracted the tar file. This example narrows the search to your $HOME directory. Change for your environment. directory. You may find the same file in more than one sub-directory of the tarfile.

```
student@cp:~$ find $HOME -name CreateNFS.sh
```

```
<some_path>/CreateNFS.sh
```

```
student@cp:~$ cp <path_from_output_above>/CreateNFS.sh $HOME
```

```
student@cp:~$ bash $HOME/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 *
```

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

```
student@worker:~$ sudo apt-get -y install nfs-common nfs-kernel-server
```

```
<output_omitted>
```

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

```
student@worker:~$ showmount -e cp #<-- Edit to be first node's name or IP
```

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

5. 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 cp node.

```
student@worker:~$ sudo mount cp:/opt/sfw /mnt
```

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

```
student@worker:~$ ls -l /mnt
```

```
total 4
-rw-r--r-- 1 root root 9 Sep 28 17:55 hello.txt
```

7. Return to the cp 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 cp 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@cp:~$ find $HOME -name PVol.yaml
```

```
<some_long_path>/PVol.yaml
```

```
student@cp:~$ cp <path_output_from_above>/PVol.yaml $HOME
```

```
student@cp:~$ vim PVol.yaml
```

PVol.yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pvvol-1
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    path: /opt/sfw
    server: cp                          #<-- Edit to match cp node name or IP
    readOnly: false
```

8. 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@cp:~$ kubectl create -f PVol.yaml
```

```
persistentvolume/pvvol-1 created
```

```
student@cp:~$ 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
```

9. 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@cp:~/$ 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
```

10. Create or copy a yaml file with the kind **PersistentVolumeClaim**.

```
student@cp:~$ vim pvc.yaml
```

pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-one
spec:
  accessModes:
  - ReadWriteMany
  resources:
```

```
      requests:
        storage: 200Mi
```

11. 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@cp:~$ kubectl create -f pvc.yaml
```

```
persistentvolumeclaim/pvc-one created
```

```
student@cp:~$ 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
```

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

```
student@cp:~$ kubectl get pv
```

```
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
```

13. 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@cp:~$ vim $HOME/app1/simpleapp.yaml
```

simpleapp.yaml

```
....
        volumeMounts:
        - name: car-vol
          mountPath: /etc/cars
        - name: nfs-vol                    #<-- Add this and following line
          mountPath: /opt
....
      volumes:
       - name: car-vol
         configMap:
           defaultMode: 420
           name: fast-car
       - name: nfs-vol                     #<-- Add this and following two lines
         persistentVolumeClaim:
           claimName: pvc-one
status:
....
```

14. Delete and re-create the deployment.

```
student@cp:~$ kubectl delete deployment try1 ; kubectl create -f $HOME/app1/simpleapp.yaml
```

```
deployment.apps "try1" deleted
deployment.apps/try1 created
```

15. 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@cp:~$ kubectl describe pod try1-594fbb5fc7-5k7sj
```

```
<output_omitted>
    Mounts:
      /etc/cars from car-vol (rw)
      /opt from nfs-vol (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-j7cqd (ro)
<output_omitted>
```

Exercise 5.3: Using ConfigMaps Configure Containers

In an earlier lab we added a second 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@cp:~$ cat basic.yaml
```

```
<output_omitted>
  containers:
  - name: webcont
    image: nginx
    ports:
    - containerPort: 80
  - name: fdlogger
    image: fluentd
```

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@cp:~$ 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@cp:~$ vim weblog-pv.yaml
```

weblog-pv.yaml

```
1 kind: PersistentVolume
2 apiVersion: v1
3 metadata:
4   name: weblog-pv-volume
5   labels:
6     type: local
7 spec:
```

```yaml
  storageClassName: manual
  capacity:
    storage: 100Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/tmp/weblog"
```

4. Create and verify the new PV exists and shows an Available status.

```
student@cp:~$ kubectl create -f weblog-pv.yaml
```

```
persistentvolume/weblog-pv-volume created
```

```
student@cp:~$ 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@cp:~$ vim weblog-pvc.yaml
```

weblog-pvc.yaml

```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@cp:~$ kubectl create -f weblog-pvc.yaml
```

```
persistentvolumeclaim/weblog-pv-claim created
```

```
student@cp:~$ 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@cp:~$ vim basic.yaml
```

basic.yaml

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: basicpod
5    labels:
6      type: webserver
7  spec:
8    volumes:                          #<-- Add three lines, same depth as containers
9      - name: weblog-pv-storage
10       persistentVolumeClaim:
11         claimName: weblog-pv-claim
12   containers:
13   - name: webcont
14     image: nginx
15     ports:
16     - containerPort: 80
17     volumeMounts:                   #<-- Add three lines, same depth as ports
18       - mountPath: "/var/log/nginx/"
19         name: weblog-pv-storage     # Must match volume name above
20   - name: fdlogger
21     image: fluentd
22     volumeMounts:                   #<-- Add three lines, same depth as image:
23       - mountPath: "/var/log"
24         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.

```
student@cp:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

```
student@cp:~$ kubectl exec -c webcont -it basicpod -- /bin/bash
```

On Container

```
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:/# tail -f /var/log/nginx/access.log
```

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

```
student@cp:~$ kubectl get pods -o wide
```

```
NAME      READY STATUS    RESTARTS  AGE    IP              NODE
   NOMINATED NODE
basicpod 2/2   Running  0         3m26s  192.168.213.181  cp
   <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. You can use **ctrl-c** and **exit** to return to the host shell prompt.

```
student@cp:~$ curl http://192.168.213.181
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

On Container

192.168.32.128 - - [18/Oct/2022:16:16:21 +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.

    Fluentd has many options for input and output of data. We will read from a file of the webcont container and write to standard out of the fdlogger container. The details of the data settings can be found in **fluentd** documentation here: https://docs.fluentd.org/v1.0/categories/config-file

```
student@cp:~$ vim weblog-configmap.yaml
```

weblog-configmap.yaml

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: fluentd-config
5    namespace: default
6  data:
7    fluent.conf: |
8      <source>
9        @type tail
10       format none
11       path /var/log/access.log
12       tag count.format1
13     </source>
14
15     <match *.**>
16     @type stdout
17     id stdout_output
18     </match>
```

12. Create the new configMap.

```
student@cp:~$ kubectl create -f weblog-configmap.yaml
```

```
configmap/fluentd-config created
```

13. View the logs for both containers in the basicpod. You should see some startup information, but not the HTTP traffic.

```
student@cp:~$ kubectl logs basicpod webcont
```

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
```

```
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
```

```
student@cp:~$  kubectl logs basicpod fdlogger
```

```
2020-09-02 19:32:59 +0000 [info]: reading config file path="/etc/fluentd-config/fluentd.conf"
2020-09-02 19:32:59 +0000 [info]: starting fluentd-0.12.29
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-config-placeholders' version '0.4.0'
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-plaintextformatter' version '0.2.6'

<output_omitted>

  <source>
    @type tail
    format none
    path /var/log/access.log
<output_omitted>
```

14. 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@cp:~$ vim basic.yaml
```

basic.yaml

```
....
  volumes:
    - name: weblog-pv-storage
      persistentVolumeClaim:
        claimName: weblog-pv-claim
    - name: log-config                    #<-- This and two lines following
      configMap:
        name: fluentd-config              # Must match existing configMap
....
    image: fluentd
    env:                                  #<-- This and two lines following
    - name: FLUENTD_OPT
      value: -c /etc/fluentd-config/fluent.conf
....
    volumeMounts:
      - mountPath: "/var/log"
        name: weblog-pv-storage
      - name: log-config                  #<-- This and next line
        mountPath: "/etc/fluentd-config"
```

15. At this point we can delete and re-create the pod, which would cause the configmap to be used by the new pod, among other changes.

```
student@cp:~$ kubectl delete pod basicpod
```

```
pod "basicpod" deleted
```

```
student@cp:~$ kubectl create -f basic.yaml
```

```
pod/basicpod created
```

```
student@cp:~$ kubectl get pod basicpod -o wide
```

```
NAME       READY   STATUS    RESTARTS   AGE   IP              NODE     NOMINATED....
basicpod   2/2     Running   0          8s    192.168.171.122 worker   <none>   ....
```

16. Use **curl** a few times to look at the default page served by `basicpod`

```
student@cp:~$ curl http://192.168.171.122
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
<output_omitted>
```

17. Look at the logs for both containers. In addition to the standard startup information, you should also see the HTTP requests from the curl commands you just used at the end of the `fdlogger` output.

```
student@cp:~$ kubectl logs basicpod webcont
```

```
/docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
/docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
/docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
10-listen-on-ipv6-by-default.sh: Getting the checksum of /etc/nginx/conf.d/default.conf
10-listen-on-ipv6-by-default.sh: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
/docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
/docker-entrypoint.sh: Configuration complete; ready for start up
```

```
student@cp:~$  kubectl logs basicpod fdlogger
```

```
2020-09-02 19:32:59 +0000 [info]: reading config file path="/etc/fluentd-config/fluentd.conf"
2020-09-02 19:32:59 +0000 [info]: starting fluentd-0.12.29
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-config-placeholders' version '0.4.0'
2020-09-02 19:32:59 +0000 [info]: gem 'fluent-mixin-plaintextformatter' version '0.2.6'

<output_omitted>

  <source>
    @type tail
    format none
    path /var/log/access.log

<output_omitted>

2020-09-02 19:47:38 +0000 count.format1: {"message":"192.168.219.64 - - [02/Sep/2020:19:47:38
↪  +0000] \"GET / HTTP/1.1\" 200 612 \"-\" \"curl/7.58.0\" \"-\""}
2020-09-02 19:47:41 +0000 count.format1: {"message":"192.168.219.64 - - [02/Sep/2020:19:47:41
↪  +0000] \"GET / HTTP/1.1\" 200 612 \"-\" \"curl/7.58.0\" \"-\""}
2020-09-02 19:47:47 +0000 count.format1: {"message":"192.168.219.64 - - [02/Sep/2020:19:47:47
↪  +0000] \"GET / HTTP/1.1\" 200 612 \"-\" \"curl/7.58.0\" \"-\""}
```

Exercise 5.4: Rolling Updates and Rollbacks

We created our `simpleapp` image using **podman**, and will create an update to the container. Remember that if you are using **docker** the command syntax should be the same.

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@cp:~$ cd $HOME/app1

student@cp:~/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. View the current images, tags and how long ago the images were created.

```
student@cp:~/app1$ sudo podman images |grep simple
```

```
10.97.177.111:5000/simpleapp        latest    fd6e141c3a2e  14 hours ago   925 MB
localhost/simpleapp                 latest    fd6e141c3a2e  14 hours ago   925 MB
```

3. 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 for the local image.

```
student@cp:~/app1$ sudo podman build -t simpleapp .
```

```
STEP 1: FROM python:2
STEP 2: ADD simple.py /
--> c1620c253cb
STEP 3: CMD [ "python", "./simple.py" ]
STEP 4: COMMIT simpleapp
--> fd46f4115eb
fd46f4115ebc87a064c15c57b08779b02deec97ab1abe92713689342ece496bd
```

```
student@cp:~/app1$ sudo podman images | grep simple
```

```
localhost/simpleapp                 latest    fd46f4115ebc  43 seconds ago  925 MB
10.97.177.111:5000/simpleapp        latest    fd6e141c3a2e  14 hours ago    925 MB
```

4. Tag and push the updated image to your locally hosted registry. Use the tag `v2` this time at the end of the image name.

```
student@cp:~/app1$ sudo podman tag simpleapp $repo/simpleapp:v2

student@cp:~/app1$ sudo podman push $repo/simpleapp:v2
```

```
Getting image source signatures
Copying blob 46829331b1e4 skipped: already exists
Copying blob 47458fb45d99 skipped: already exists
Copying blob 461719022993 skipped: already exists
Copying blob a3c1026c6bcc skipped: already exists
Copying blob d35c5bda4793 skipped: already exists
Copying blob f1d420c2af1a skipped: already exists
Copying blob bbcb11d3fa81 done
Copying blob ceee8816bb96 skipped: already exists
Copying blob da7b0a80a4f2 skipped: already exists
Copying blob e571d2d3c73c skipped: already exists
Copying config fd46f4115e done
Writing manifest to image destination
Storing signatures
```

5. Check your images again, there should be an additional entry with same image ID, but with the tag of `v2`.

```
student@cp:~/app1$ sudo podman images |grep simple
```

```
10.97.177.111:5000/simpleapp        v2        fd46f4115ebc  3 minutes ago  925 MB
localhost/simpleapp                 latest    fd46f4115ebc  3 minutes ago  925 MB
10.97.177.111:5000/simpleapp        latest    fd6e141c3a2e  14 hours ago   925 MB
```

6. Connect to a terminal running on your worker node. Pull the image without asking for a version, which one would expect to pull the `latest` image, then pull `v2`. Note the default did not pull the new version of the image. Literally the `latest` tag has no relationship to the latest image.

```
student@worker:~$ sudo podman pull $repo/simpleapp
```

```
Trying to pull 10.97.177.111:5000/simpleapp:latest...
Getting image source signatures
Copying blob ca488b1eb9fa skipped: already exists
Copying blob 1f22e54987ac skipped: already exists
Copying blob 33be93024b52 skipped: already exists
Copying blob 41b428188c3d skipped: already exists
Copying blob 6dae3f0239bb skipped: already exists
Copying blob 637977187da0 skipped: already exists
Copying blob 3263dd1bdf84 [--------------------------------------] 0.0b / 0.0b
Copying blob 7506b856c5df [--------------------------------------] 0.0b / 0.0b
Copying blob 19aa3a14ed80 [--------------------------------------] 0.0b / 0.0b
Copying blob c45f0d38c649 [--------------------------------------] 0.0b / 0.0b
Copying config fd6e141c3a done
Writing manifest to image destination
Storing signatures
fd6e141c3a2e2d663f56c0fa7a4d056f73f6d1f63da6e36b19d7ef006dd8fff6
```

```
student@worker:~$ sudo podman pull $repo/simpleapp:v2
```

```
Trying to pull 10.97.177.111:5000/simpleapp:v2...
Getting image source signatures
Copying blob ca488b1eb9fa skipped: already exists
Copying blob 1f22e54987ac skipped: already exists
Copying blob 6dae3f0239bb skipped: already exists
Copying blob 41b428188c3d skipped: already exists
Copying blob 637977187da0 skipped: already exists
Copying blob 33be93024b52 skipped: already exists
Copying blob 3263dd1bdf84 skipped: already exists
Copying blob 7506b856c5df skipped: already exists
Copying blob c45f0d38c649 skipped: already exists
Copying blob 4da1215fbbb8 done
Copying config fd46f4115e done
Writing manifest to image destination
Storing signatures
fd46f4115ebc87a064c15c57b08779b02deec97ab1abe92713689342ece496bd
```

7. Return to your cp node, 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@cp:~/app1$ kubectl edit deployment try1
```

```
....
      containers:
      - image: 10.105.119.236:5000/simpleapp:v2   #<-- Edit tag
        imagePullPolicy: Always
....
```

8. 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@cp:~/app1$ kubectl get events
```

```
42m         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fdbb5d557 to 6
32s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 2
32s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to
↪  5
32s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 3
23s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to
↪  4
23s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 4
22s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to
↪  3
22s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 5
18s         Normal    ScalingReplicaSet   Deployment   Scaled down replica set try1-7fdbb5d557 to
↪  2
18s         Normal    ScalingReplicaSet   Deployment   Scaled up replica set try1-7fd7459fc6 to 6
8s          Normal    ScalingReplicaSet   Deployment   (combined from similar events):
Scaled down replica set try1-7fdbb5d557 to 0
```

9. 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@cp:~/app1$ kubectl describe pod try1-895fccfb-ttqdn |grep Image
```

```
    Image:          10.105.119.236:5000/simpleapp:v2
    Image ID:       10.97.177.111:5000/simpleapp@sha256:c3848fd3b5cdf5f241ceb4aa7e96c5c2be9f09
f23eea1afe946f8507f3fcbc29
    Image:          registry.k8s.io/goproxy:0.1
    Image ID:       docker.io/library/goproxy@sha256:5334c7ad43048e3538775c
b09aaf184f5e8c
c8f1d93c209865c8f1d93c2098
```

10. View the update history of the deployment.

```
student@cp:~/app1$ kubectl rollout history deployment try1
```

```
deployment.apps/try1
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

11. 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@cp:~/app1$ kubectl rollout history deployment try1 --revision=1 > one.out

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

student@cp:~/app1:$ diff one.out two.out
```

```
1c1
< deployment.apps/try1 with revision #1
---
> deployment.apps/try1 with revision #2
4c4
<           pod-template-hash=1509661973
---
>           pod-template-hash=45197796
7c7
<     Image:        10.105.119.236:5000/simpleapp
---
>     Image:        10.105.119.236:5000/simpleapp:v2
```

12. 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@cp:~/app1$ kubectl rollout undo --dry-run=client deployment/try1
```

```
deployment.apps/try1 Pod Template:
  Labels:       app=try1
        pod-template-hash=754bf9c75b
  Containers:
   simpleapp:
    Image:        10.97.177.111:5000/simpleapp
    Port:         <none>
    Host Port:        <none>
<output_omitted>
```

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

```
student@cp:~/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
```

14. 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@cp:~/app1$ kubectl rollout undo deployment try1 --to-revision=1
```

```
deployment.apps/try1 rolled back
```

15. 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@cp:~/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
```

Exercise 5.5: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview. locate some of the topics we have covered in this chapter, which should match up with the following list of bullet points.

- Utilize persistent and ephemeral volumes
- Understand Deployments and how to perform rolling updates
- Understand ConfigMaps

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

Using only the allowed URLs and subdomains search for a YAML example to create and configure the resources called for in this review. Be familiar with what the page looks like so can quickly return to the working YAML.

1. Create a new secret called `specialofday` using the key `entree` and the value `meatloaf`.
2. Create a new deployment called `foodie` running the `nginx` image.
3. Add the `specialofday` secret to pod mounted as a volume under the `/food/` directory.
4. Execute a bash shell inside a `foodie` pod and verify the secret has been properly mounted.
5. Update the deployment to use the `nginx:1.12.1-alpine` image and verify the new image is in use.
6. Roll back the deployment and verify the typical, current stable version of nginx is in use again.
7. Create a new 200M NFS volume called `reviewvol` using the NFS server configured earlier in the lab.
8. Create a new PVC called `reviewpvc` which will uses the `reviewvol` volume.
9. Edit the deployment to use the PVC and mount the volume under `/newvol`
10. Execute a bash shell into the nginx container and verify the volume has been mounted.
11. Delete any resources created during this review.

Chapter 6

Understanding Security

Exercise 6.1: Set SecurityContext for a Pod and Container

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. We'll 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.

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

```
student@cp:~$ mkdir $HOME/app2

student@cp:~$ cd $HOME/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@cp:~/app2$ vim second.yaml
```

YAML **second.yaml**

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: secondapp
5  spec:
6    securityContext:
7      runAsUser: 1000
8    containers:
9    - name: busy
10     image: busybox
```

```
      command:
        - sleep
        - "3600"
      securityContext:
        runAsUser: 2000
        allowPrivilegeEscalation: false
```

3. Create the secondapp pod and verify it's 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@cp:~/app2$ kubectl create -f second.yaml
  pod/secondapp created

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

student@cp:~/app2$ kubectl get pod secondapp -o yaml
  apiVersion: v1
  kind: Pod
  metadata:
    creationTimestamp: "2023-12-11T21:23:12Z"
    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@cp:~/app2$ kubectl exec -it secondapp -- sh
```

On Container

```
/ $ 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.

On Container

```
/ $ grep Cap /proc/1/status
```

```
CapInh:         00000000000005fb
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         00000000000005fb
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 something like nine comma separated capabilities listed. The number may change in future versions.

```
student@cp:~/app2$ capsh --decode=00000000000005fb
```

```
0x00000000000005fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,
cap_setgid,cap_setuid,cap_setpcap,cap_net_bind_service
```

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. We'll 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@cp:~/app2$ kubectl delete pod secondapp
```

```
pod "secondapp" deleted
```

```
student@cp:~/app2$ vim second.yaml
```

second.yaml

```
<output_omitted>
      - sleep
      - "3600"
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
      capabilities:                        #<-- Add this and following line
        add: ["NET_ADMIN", "SYS_TIME"]
```

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@cp:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@cp:~/app2$ kubectl exec -it secondapp -- sh
```

On Container

```
/ $ grep Cap /proc/1/status
```

```
CapInh:         00000000020015fb
CapPrm:         0000000000000000
CapEff:         0000000000000000
CapBnd:         00000000020015fb
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@cp:~/app2$ capsh --decode=00000000020015fb
```

```
0x00000000020015fb=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_sys_time
```

Exercise 6.2: Create and consume Secrets

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

1. Begin by generating an encoded password.

```
student@cp:~/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@cp:~/app2$ vim secret.yaml
```

YAML **secret.yaml**

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: lfsecret
5 data:
6   password: TEZUckAxbgo=
```

3. Ingest the new object into the cluster.

```
student@cp:~/app2$ kubectl create -f secret.yaml
```

```
secret/lfsecret created
```

4. Edit secondapp YAML file to use the secret as a volume mounted under `/mysqlpassword`. `volumeMounts:` lines up with the container `name:` and `volumes:` lines up with `containers:` Note the pod will restart when the sleep command finishes every 3600 seconds, or every hour.

```
student@cp:~/app2$ vim second.yaml
```

second.yaml

```
....
      runAsUser: 2000
      allowPrivilegeEscalation: false
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
    volumeMounts:                          #<-- Add this and six following lines
    - name: mysql
      mountPath: /mysqlpassword
  volumes:
  - name: mysql
    secret:
      secretName: lfsecret
```

```
student@cp:~/app2$ kubectl delete pod secondapp
```

```
pod "secondapp" deleted
```

```
student@cp:~/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@cp:~/app2$ kubectl get pod secondapp
```

```
NAME        READY     STATUS    RESTARTS   AGE
secondapp   1/1       Running   0          34s
```

```
student@cp:~/app2$ kubectl exec -ti secondapp -- /bin/sh
```

On Container

```
/ $ 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.

On Container

```
/ $ cd /mysqlpassword/

/mysqlpassword $ ls
```

```
password
```

```
/mysqlpassword $ ls -al
```

```
total 4
drwxrwxrwt    3 root     root          100 May 19 16:06 .
dr-xr-xr-x    1 root     root         4096 May 19 16:06 ..
drwxr-xr-x    2 root     root           60 May 19 16:06 ..2021_05_19_16_06_41.694089911
lrwxrwxrwx    1 root     root           31 May 19 16:06 ..data ->
↪  ..2021_05_19_16_06_41.694089911
lrwxrwxrwx    1 root     root           15 May 19 16:06 password -> ..data/password
```

```
/mysqlpassword $ exit
```

Exercise 6.3: 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@cp:~/app2$ cd
```

```
student@cp:~$ kubectl get secrets
```

```
NAME                  TYPE                                  DATA   AGE
lfsecret              Opaque                                1      6m5s
```

```
student@cp:~$ kubectl get secrets --all-namespaces
```

```
NAMESPACE     NAME                     TYPE                            DATA   AGE
default       lfsecret                 Opaque                          1      69s
kube-system   bootstrap-token-j6r4vk   bootstrap.kubernetes.io/token   7      58m

<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@cp:~$ vim serviceaccount.yaml
```

serviceaccount.yaml

```
1 apiVersion: v1
2 kind: ServiceAccount
3 metadata:
4   name: secret-access-sa
```

```
student@cp:~$ kubectl create -f serviceaccount.yaml
```

```
serviceaccount/secret-access-sa created
```

```
student@cp:~$ kubectl get serviceaccounts
```

```
NAME              SECRETS   AGE
default           0         1d17h
secret-access-sa  0         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@cp:~$ kubectl get clusterroles
```

```
NAME                                                   AGE
admin                                                  1d17h
cilium                                                 1d17h
cilium-operator                                        1d17h
cluster-admin                                          1d17h
<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@cp:~$ kubectl get clusterroles admin -o yaml
```

```
<output_omitted>
```

```
student@cp:~$ kubectl get clusterroles cluster-admin -o yaml
```

```
<output_omitted>
```

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

```
student@cp:~$ vim clusterrole.yaml
```

clusterrole.yaml

```
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: ClusterRole
3  metadata:
4   name: secret-access-cr
5  rules:
6  - apiGroups:
7    - ""
8    resources:
9    - secrets
10   verbs:
11   - get
12   - list
```

6. Create and verify the new `ClusterRole`.

```
student@cp:~$ kubectl create -f clusterrole.yaml
```

```
clusterrole.rbac.authorization.k8s.io/secret-access-cr created
```

```
student@cp:~$ 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@cp:~$ vim rolebinding.yaml
```

rolebinding.yaml

```
1  apiVersion: rbac.authorization.k8s.io/v1
2  kind: RoleBinding
3  metadata:
4   name: secret-rb
5  subjects:
6  - kind: ServiceAccount
7    name: secret-access-sa
8  roleRef:
9   kind: ClusterRole
10  name: secret-access-cr
11  apiGroup: rbac.authorization.k8s.io
```

8. Create the new `RoleBinding` and verify.

```
student@cp:~$ kubectl create -f rolebinding.yaml
```

```
rolebinding.rbac.authorization.k8s.io/secret-rb created
```

```
student@cp:~$ kubectl get rolebindings
```

```
NAME        AGE
secret-rb   17s
```

9. View the `secondapp` pod and **grep** for the current `serviceAccount`. Note that it uses the default account.

```
student@cp:~$ kubectl get pod secondapp -o yaml |grep serviceAccount
```

```
serviceAccount: default
serviceAccountName: default
    - serviceAccountToken:
```

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

```
student@cp:~$ vim $HOME/app2/second.yaml
```

second.yaml

```
1  ....
2    name: secondapp
3  spec:
4    serviceAccountName: secret-access-sa  #<-- Add this line
5    securityContext:
6      runAsUser: 1000
7  ....
```

11. We will delete the `secondapp` pod if still running, then create it again. Note that the `serviceAccount` is no longer the `default`.

```
student@cp:~$ kubectl delete pod secondapp ; kubectl create -f $HOME/app2/second.yaml
```

```
pod "secondapp" deleted
pod/secondapp created
```

```
student@cp:~$ kubectl get pod secondapp -o yaml | grep serviceAccount
```

```
  serviceAccount: secret-access-sa
  serviceAccountName: secret-access-sa
      - serviceAccountToken:
```

Exercise 6.4: 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@cp:~$ cd $HOME/app2/

student@cp:~/app2$ vim allclosed.yaml
```

YAML **allclosed.yaml**

```
1  apiVersion: networking.k8s.io/v1
2  kind: NetworkPolicy
3  metadata:
4    name: deny-default
5  spec:
6    podSelector: {}
7    policyTypes:
8    - Ingress
9    - 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 we'll delete then edit the file.

```
student@cp:~/app2$ kubectl delete pod secondapp
```

```
  pod "secondapp" deleted
```

```
student@cp:~/app2$ vim second.yaml
```

YAML **second.yaml**

```
1  ....
2  spec:
3       serviceAccountName: secret-access-sa
4       securityContext:
5         runAsUser: 1000
6       containers:
7       - name: webserver                    #<-- Add this and following line
```

```
        image: nginx
      - name: busy
        image: busybox
        command:
....
```

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@cp:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@cp:~/app2$ kubectl get pods
```

```
NAME                        READY   STATUS             RESTARTS   AGE
nginx-6b58d9cdfd-9fnl4      1/1     Running            1          2d
Registry-795c6c8b8f-hl5wf   1/1     Running            2          2d
secondapp                   1/2     CrashLoopBackOff   1          13s
<output_omitted>
```

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@cp:~/app2$ kubectl get event
```

```
<output_omitted>
25s         Normal    Scheduled   Pod    Successfully assigned default/secondapp to cp
4s          Normal    Pulling     Pod    pulling image "nginx"
2s          Normal    Pulled      Pod    Successfully pulled image "nginx"
2s          Normal    Created     Pod    Created container
2s          Normal    Started     Pod    Started container
23s         Normal    Pulling     Pod    pulling image "busybox"
21s         Normal    Pulled      Pod    Successfully pulled image "busybox"
21s         Normal    Created     Pod    Created container
21s         Normal    Started     Pod    Started container
1s          Warning   BackOff     Pod    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@cp:~/app2$ kubectl logs secondapp webserver
```

```
2018/04/13 19:51:13 warn 1 the "user" directive makes sense
only if the cp process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
nginx: warn the "user" directive makes sense only if the cp
process runs with super-user privileges,
 ignored in /etc/nginx/nginx.conf:2
2018/04/13 19:51:13 emerg 11: 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@cp:~/app2$ kubectl delete -f second.yaml
```

```
pod "secondapp" deleted
```

```
student@cp:~/app2$ vim second.yaml
```

```
spec:
  serviceAccountName: secret-access-sa
#  securityContext:                        #<-- Comment this and following line
#    runAsUser: 1000
  containers:
  - name: webserver
```

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@cp:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@cp:~/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@cp:~/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@cp:~/app2$ kubectl delete -f second.yaml
```

```
pod "secondapp" deleted
```

```
student@cp:~/app2$ vim second.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secondapp
  labels:                       #<-- This and following line
    example: second
spec:
#  securityContext:
#    runAsUser: 1000
<output_omitted>
```

```
student@cp:~/app2$ kubectl create -f second.yaml
```

```
pod/secondapp created
```

```
student@cp:~/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@cp:~/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@cp:~/app2$ kubectl get svc secondapp -o yaml
```

```
apiVersion: v1
kind: Service
metadata:
  creationTimestamp: "2020-04-16T04:40:07Z"
  labels:
    example: second
  managedFields:

....

spec:
  clusterIP: 10.97.96.75
  externalTrafficPolicy: Cluster
  ports:
  - nodePort: 31665
    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 fail. If it hangs **control-c** to exit back to the shell.

```
student@cp:~/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@cp:~/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, note key and parameter change
  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@cp:~/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 it's available. Both of your nodes should be exposing the web server on port 32000. The example shows the use of the **curl** command, you could also use a web browser.

```
student@cp:~/app2$ curl http://10.97.96.75
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

```
student@cp:~/app2$ curl ifconfig.io
```

```
35.184.219.5
```

```
[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. We'll 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@cp:~/app2$ kubectl exec -it -c busy secondapp -- sh
```

On Container

```
/ $ 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
```

Exercise 6.5: Testing the Policy

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

```
student@cp:~/app2$ kubectl create -f $HOME/app2/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, and interrupt if you get tired of waiting.

```
[user@laptop ~]$ curl http://35.184.219.5:32000
```

```
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@cp:~/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@cp:~/app2$ kubectl exec -it -c busy secondapp -- sh
```

On Container

```
/ $ 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@cp:~/app2$ vim $HOME/app2/allclosed.yaml
```

allclosed.yaml

```
1 ....
2 spec:
3   podSelector: {}
4   policyTypes:
5   - Ingress
```

```
6  #  - Egress            #<-- Comment out this line
```

```
student@cp:~/app2$ kubectl replace -f $HOME/app2/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@cp:~/app2$ kubectl exec -it -c busy secondapp -- sh
```

On Container

```
/ $ 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 an ingress rule to allow ingress to only the nginx container. Use the IP from the **eth0** range.

```
student@cp:~/app2$ vim $HOME/app2/allclosed.yaml
```

allclosed.yaml

```
1  <output_omitted>
2   policyTypes:
3   - Ingress
4   ingress:                    #<-- Add this and following three lines
5   - from:
6     - podSelector: {}
7  # - Egress
```

8. Recreate the policy, and verify its configuration.

```
student@cp:~/app2$ kubectl replace -f $HOME/app2/allclosed.yaml
```

```
networkpolicy.networking.k8s.io/deny-default replaced
```

```
student@cp:~/app2$ kubectl get networkpolicy
```

```
NAME           POD-SELECTOR     AGE
deny-default   <none>           3m2s
```

```
student@cp:~/app2$ kubectl get networkpolicy -o yaml
```

```
apiVersion: v1
items:
- apiVersion: networking.k8s.io/v1
  kind: NetworkPolicy
  metadata:
<output_omitted>
```

9. Test access to the container using **ping**, use the IP address of the pod

```
student@cp:~/app2$ kubectl run -it test --rm=true --image alpine -- ping -c5 192.168.55.91
```

```
If you don't see a command prompt, try pressing enter.
64 bytes from 192.168.1.45: seq=1 ttl=63 time=0.094 ms
64 bytes from 192.168.1.45: seq=2 ttl=63 time=0.102 ms
64 bytes from 192.168.1.45: seq=3 ttl=63 time=0.103 ms
64 bytes from 192.168.1.45: seq=4 ttl=63 time=0.111 ms

--- 192.168.1.45 ping statistics ---
5 packets transmitted, 5 packets received, 0% packet loss
round-trip min/avg/max = 0.094/0.216/0.672 ms
Session ended, resume using 'kubectl attach test -c test -i -t' command when the pod is running
pod "test" deleted
```

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@cp:~/app2$ vim $HOME/app2/allclosed.yaml
```

allclosed.yaml

```
<output_omitted>
  - Ingress
  ingress:
  - from:
    - podSelector: {}
    ports:                      #<-- Add this and two following lines
    - port: 80
      protocol: TCP
#  - Egress
```

```
student@cp:~/app2$ kubectl replace -f $HOME/app2/allclosed.yaml
```

```
networkpolicy.networking.k8s.io/deny-default replaced
```

```
student@cp:~/app2$ kubectl run -it test --rm=true --image alpine -- ping -c5 192.168.55.91
```

All five pings should fail, with zero received.

```
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
```

11. You may want to remove the `default-deny` policy, in case you want to get to your registry or other pods.

```
student@cp:~/app2$ kubectl delete networkpolicies deny-default
```

```
networkpolicy.networking.k8s.io "deny-default" deleted
```

Exercise 6.6: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.

- Understand authentication, authorization, and admission control
- Create & consume Secrets
- Understand ServiceAccounts
- Understand SecurityContexts
- Demonstrate basic understanding of NetworkPolicies

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

1. Create a new deployment which uses the `nginx` image.
2. Create a new `LoadBalancer` service to expose the newly created deployment. Test that it works.
3. Create a new NetworkPolicy called netblock which blocks all traffic to pods in this deployment only. Test that all traffic is blocked to deployment.
4. Update the `netblock` policy to allow traffic to the pod on port 80 only. Test that you can access the default nginx web page.
5. Find and use the `security-review1.yaml` file to create a pod.

```
student@cp:~$ kubectl create -f security-review1.yaml
```

6. View the status of the pod.
7. Use the following commands to figure out why the pod has issues.

```
student@cp:~$ kubectl get pod securityreview

student@cp:~$ kubectl describe pod securityreview

student@cp:~$ kubectl logs securityreview
```

8. After finding the errors, log into the container and find the proper id of the nginx user.
9. Edit the yaml and re-create the pod such that the pod runs without error.

10. Create a new `serviceAccount` called securityaccount.
11. Create a ClusterRole named secrole which only allows create, delete, and list of pods in all apiGroups.
12. Bind the clusterRole to the serviceAccount.
13. Locate the token of the securityaccount. Create a file called `/tmp/securitytoken`. Put only the value of `token:` is equal to, a long string that may start with eyJh and be several lines long. Careful that only that string exists in the file.
14. Remove any resources you have added during this review

Chapter 7

Exposing Applications

Exercise 7.1: Exposing Applications: Expose a Service

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.

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@cp:~$ 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. Save then delete the existing service for secondapp. Ensure the same labels, ports, and protocols are used.

```
student@cp:~$ cd $HOME/app2

student@cp:~/app2$ kubectl get svc secondapp -o yaml > oldservice.yaml

student@cp:~/app2$ cat oldservice.yaml

student@cp:~/app2$ kubectl delete svc secondapp
```

```
service "secondapp" deleted
```

3. Recreate the service using a new YAML file. Use the same selector as the previous pod. Examine the new service after creation, note the `TYPE` and `PORT(S)`.

```
student@cp:~/app2$ vim newservice.yaml
```

newservice.yaml

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: secondapp
5  spec:
6    ports:
7    - port: 80
8      protocol: TCP
9    selector:
10     example: second
11   sessionAffinity: None
12 status:
13   loadBalancer: {}
```

```
student@cp:~/app2$ kubectl create -f newservice.yaml
```

```
service/secondapp created
```

```
student@cp:~/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
```

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

```
student@cp:~/app2$ curl http://10.98.148.52
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

5. Now create another simple web server. This time use the **httpd** image as the default page is different from **nginx**. Once we are sure it is running we will edit the service selector to point at the new server then back, which could be used as a deployment strategy. Commands have been omitted. You should be able to complete the steps. Refer to previous content otherwise.

```
student@cp:~/app2$ kubectl create deployment newserver --image=httpd
```

6. Locate the `newserver` labels.
7. Use **kubectl edit** to change the service to use `newserver`'s labels as the selector.
8. Test that the service now shows the new content and not the default **nginx** page.

```
student@cp:~/app2$ curl http://10.98.148.52
```

```
<html><body><h1>It works!</h1></body></html>
```

9. Edit the selector back to the nginx server and test. Then remove the `newserver` deployment.
10. 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. You'll have to delete and re-create as the existing IP is immutable. The NodePort will create a new ClusterIP.

```
student@cp:~/app2$ vim newservice.yaml
```

newservice.yaml

```
....
    protocol: TCP
    nodePort: 32000         #<-- Add this and following line
  type: NodePort
  selector:
    example: second
```

```
student@cp:~/app2$ kubectl delete svc secondapp ; kubectl create  -f newservice.yaml
```

```
service "secondapp" deleted
service/secondapp created
```

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

```
student@cp:~/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
```

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

```
student@cp:~/app2$ curl 10.109.134.221
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

13. 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. If you don't have a terminal on your local system use a browser.

```
student@cp:~/app2$ curl ifconfig.io
```

```
35.184.219.5
```

```
user@laptop:~/Desktop$ curl http://35.184.219.5:32000
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

14. 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@cp:~/app2$ vim newservice.yaml
```

newservice.yaml

```
....
  - port: 80
    protocol: TCP
    nodePort: 32000
  type: LoadBalancer    #<-- Edit this line
  selector:
    example: second
```

```
student@cp:~/app2$ kubectl delete svc secondapp ; kubectl create -f newservice.yaml
```

```
service "secondapp" deleted
service/secondapp created
```

15. 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@cp:~/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
```

16. 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>
```

17. You can also use DNS names provided by **CoreDNS** which dynamically are added when the service is created. Start by logging into the `busy` container of secondapp.

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

On Container

(a) Use the **nslookup** command to find the `secondapp` service. Then find the `registry` service we configured to provide container images. If you don't get the expected output try again. About one out of three requests works.

```
/ $ nslookup secondapp
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        secondapp.default.svc.cluster.local
Address: 10.96.214.133

*** Can't find secondapp.svc.cluster.local: No answer
*** Can't find secondapp.cluster.local: No answer
*** Can't find secondapp.c.endless-station-188822.internal: No answer
<output_omitted>
```

```
/ $ nslookup registry
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        registry.default.svc.cluster.local
Address: 10.110.95.21
<output_omitted>
```

(b) Lookup the FQDN associated with the DNS server IP displayed by the commands. Your IP may be different.

```
/ $ nslookup 10.96.0.10
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

10.0.96.10.in-addr.arpa        name = kube-dns.kube-system.svc.cluster.local
```

(c) Attempt to resolve the service name, which should not bring back any records. Then try with the FQDN. Read through the errors. You'll note that only the default namespaces is checked. You may have to check the FQDN a few times as it doesn't always reply with an answer.

```
/ $ nslookup kube-dns
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find kube-dns.default.svc.cluster.local: NXDOMAIN

*** Can't find kube-dns.svc.cluster.local: No answer
*** Can't find kube-dns.cluster.local: No answer
*** Can't find kube-dns.c.endless-station-188822.internal: No answer
```

```
/ $ nslookup kube-dns.kube-system.svc.cluster.local
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        kube-dns.kube-system.svc.cluster.local
Address: 10.96.0.10

*** Can't find kube-dns.kube-system.svc.cluster.local: No answer
```

(d) Exit out of the container

```
/ $ exit
```

18. Create a new namespace named `multitenant` and a new deployment named `mainapp`. Expose the deployment port 80 using the name `shopping`

```
student@cp:~/app2$ kubectl create ns multitenant
```

```
namespace/multitenant created
```

```
student@cp:~/app2$ kubectl -n multitenant create deployment mainapp --image=nginx
```

```
deployment.apps/mainapp created
```

```
student@cp:~/app2$ kubectl -n multitenant expose deployment mainapp --name=shopping \
                        --type=NodePort --port=80
```

```
service/shopping exposed
```

19. Log back into the secondapp busy container and test access to mainapp.

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

On Container

(a) Use **nslookup** to determine the address of the new service. Start with using just the service name. Then add the service name and the namespaces. There are a few hiccups, with how busybox and other applications interact with CoreDNS. Your responses may or may not work. Try each a few times.

```
/ $ nslookup shopping
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find shopping.default.svc.cluster.local: NXDOMAIN

*** Can't find shopping.svc.cluster.local: No answer
<output_omitted>
```

```
/ $ nslookup shopping.multitenant
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

** server can't find shopping.multitenant: NXDOMAIN

*** Can't find shopping.multitenant: No answer
```

```
/ $ nslookup shopping.multitenant.svc.cluster.local
```

```
Server:         10.96.0.10
Address:        10.96.0.10:53

Name:        shopping.multitenant.svc.cluster.local
Address: 10.101.4.142

*** Can't find shopping.multitenant.svc.cluster.local: No answer
```

(b) Now try to use the service name and then the name with namespace, to see if it works. The DNS using the namespace should work, even if you don't have access to the default page. RBAC could be used to grant access. Check the service ClusterIP returned and it will match the newly created service.

```
/ $ wget shopping
```

```
wget: bad address 'shopping'
```

```
/ $ wget shopping.multitenant
```

```
Connecting to shopping.multitenant (10.101.4.142:80)
wget: can't open 'index.html': Permission denied
```

(c) As we can see the error is about permissions we will try again, but not try to write locally, but instead to dash (-), which is standard out.

```
~ $ wget -O - shopping.multitenant
```

```
Connecting to shopping.multitenant (10.103.211.64:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>

....

</html>
-                    100% |*********************************************|   612
↪  0:00:00 ETA
written to stdout
```

Exercise 7.2: Service Mesh and 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 mentioned a lot in Kubernetes.io, there are many to chose from. Even more functionality and metrics come from the use of a service mesh, such as Istio, Linkerd, Contour, Aspen, or several others.

1. We will install linkerd using their own scripts. There is quite a bit of output. Instead of showing all of it the output has been omitted. Look through the output and ensure that everything gets a green check mark. Some steps may take a few minutes to complete. Each command is listed here to make install easier. As well these steps are in the `setupLinkerd.txt` file.

   The most recent versions of linkerd may have some issues. As a result we will use a stable version of Linkerd, such as 2.12.2, or 2.11.4. Feel free to experiment with other versions as time and interest allows. Investigate **linkerd uninstall** and **linkerd uninject** as necessary.

   Some commands will take a while to spin up various pods.

```
student@cp:~/app2$ curl -sL run.linkerd.io/install > setup.sh
```

```
student@cp:~/app2$ vim setup.sh
```

```
#!/bin/sh
```

```
set -eu

LINKERD2_VERSION=${LINKERD2_VERSION:-stable-2.12.2}  #<-- Edit to earlier stable version
INSTALLROOT=${INSTALLROOT:-"${HOME}/.linkerd2"}

happyexit() {
....
```

```
student@cp:~/app2$ sh setup.sh

student@cp:~/app2$ export PATH=$PATH:/home/student/.linkerd2/bin

student@cp:~/app2$ linkerd check --pre

student@cp:~/app2$ linkerd install --crds | kubectl apply -f -

student@cp:~/app2$ linkerd install | kubectl apply -f -

student@cp:~/app2$ linkerd check

student@cp:~/app2$ linkerd viz install | kubectl apply -f -

student@cp:~/app2$ linkerd viz check

student@cp:~/app2$ linkerd viz dashboard &
```

2. By default the GUI is only available on the localhost, setup in the final dashboard command. We will need to edit the service and the deployment to allow outside access, in case you are using a cloud provider for the nodes. Edit to remove all characters after equal sign for `-enforced-host`, which is around line 61.

```
student@cp:~/app2$ kubectl -n linkerd-viz edit deploy web
```

```
spec:
      containers:
      - args:
        - -linkerd-controller-api-addr=linkerd-controller-api.linkerd.svc.cluster.local:8085
        - -linkerd-metrics-api-addr=metrics-api.linkerd-viz.svc.cluster.local:8085
        - -cluster-domain=cluster.local
        - -grafana-addr=grafana.linkerd-viz.svc.cluster.local:3000
        - -controller-namespace=linkerd
        - -viz-namespace=linkerd-viz
        - -log-level=info
        - -enforced-host=                                    #<-- Remove everything after equal sign
        image: cr.l5d.io/linkerd/web:stable-2.12.2
        imagePullPolicy: IfNotPresent
```

3. Now edit the http nodePort and type to be a NodePort.

```
student@cp:~/app2$ kubectl edit svc web -n linkerd-viz
```

```
....
 ports:
   - name: http
     nodePort: 31500                                 #<-- Add line with an easy to remember port
     port: 8084
 ....
   sessionAffinity: None
   type: NodePort                                    #<-- Edit type to be NodePort
```

```
  status:
    loadBalancer: {}
  ....
```

4. Test access using a local browser to your public IP. Your IP will be different than the one shown below.

```
student@cp:~/app2$ curl ifconfig.io
```

```
104.197.159.20
```

5. From you local system open a browser and go to the public IP and the high-number nodePort, such as `31500`. It will not be the localhost port seen in output.

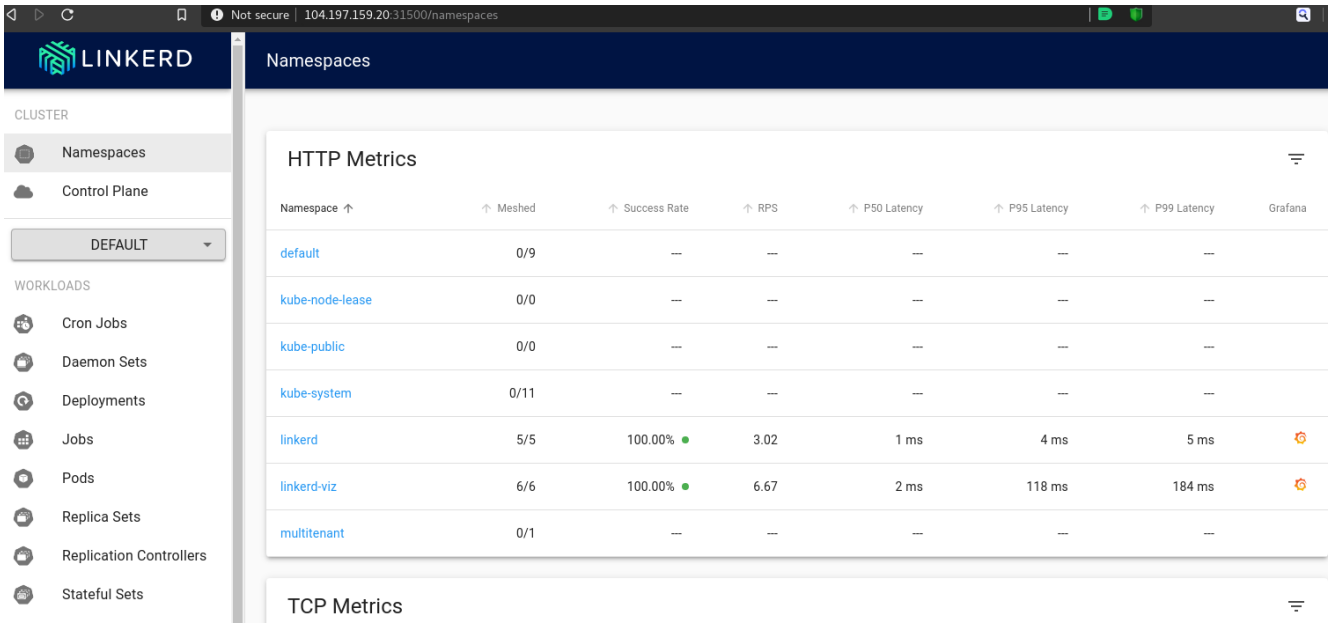

Figure 7.1: **Main Linkerd Page**

6. In order for linkerd to pay attention to an object we need to add an annotation. The **linkerd inject** command will do this for us. Generate YAML and pipe it to **linkerd** then pipe again to **kubectl**. Expect an error about how the object was created, but the process will work. The command can run on one line if you omit the back-slash.

```
student@cp:~/app2$  kubectl -n multitenant get deploy mainapp -o yaml | \
     linkerd inject - | kubectl apply -f -
```

```
<output_omitted>
```

7. Check the GUI, you should see that the `multitenant` namespaces and pods are now meshed, and the name is a link.
8. Generate some traffic to the pods, and watch the traffic via the GUI.

```
student@cp:~$ kubectl -n multitenant get svc
```

```
NAME       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
shopping   NodePort   10.102.8.205   <none>        80:32518/TCP   4h54m
```

```
student@cp:~$ curl 10.102.8.205
```

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Welcome to nginx!</title>
<output_omitted>
```

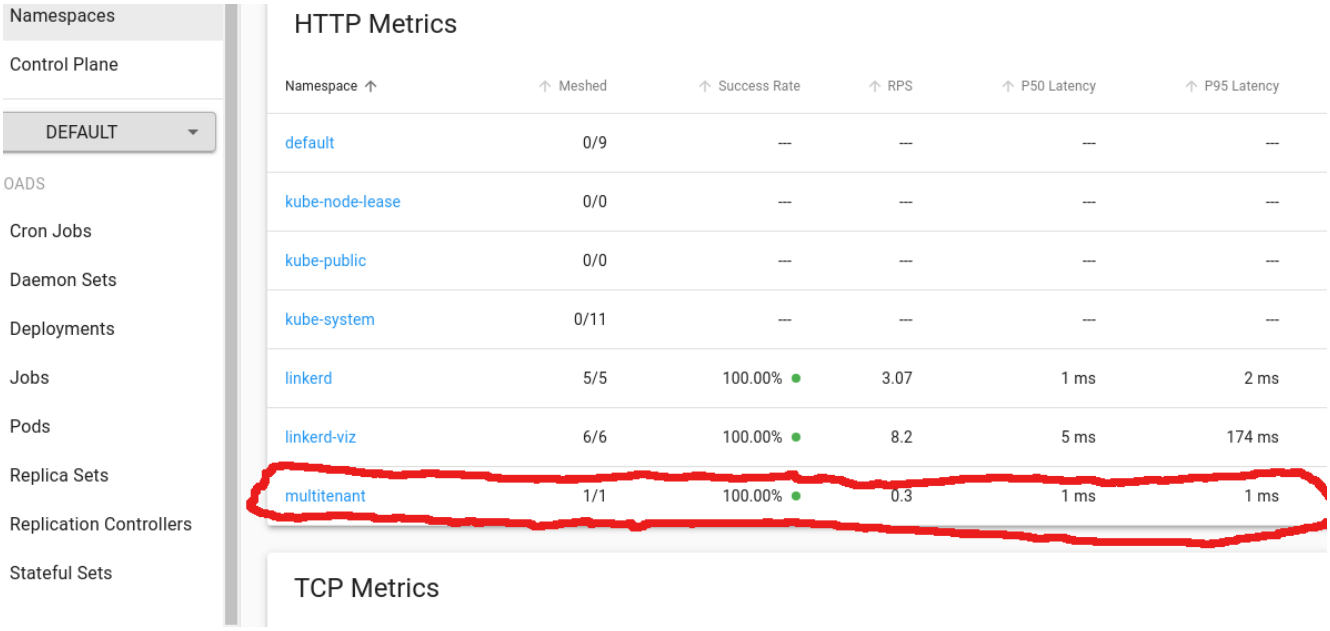

Figure 7.2: **Now shows meshed**

9. Scale up the `mainapp` deployment. Generate traffic to get metrics for all the pods.

```
student@cp:~$ kubectl -n multitenant scale deploy mainapp --replicas=5
```

```
deployment.apps/mainapp scaled
```

```
student@cp:~$ curl 10.102.8.205   #Several times
```

10. Explore some of the other information provided by the GUI.

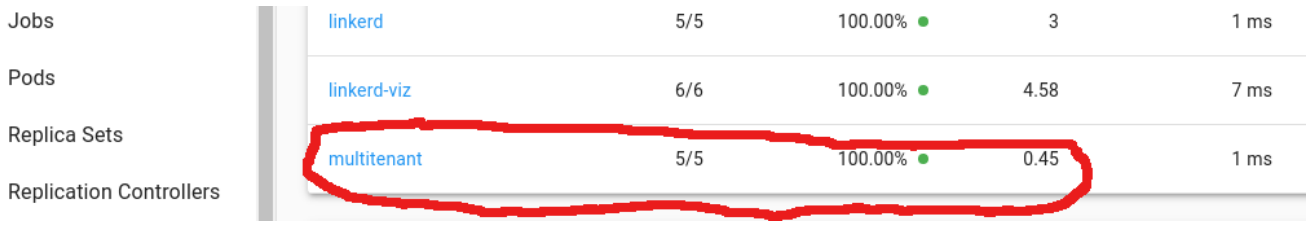

Figure 7.3: **Five meshed pods**

11. Linkerd does not come with an ingress controller, so we will add one to help manage traffic. We will leverage a **Helm** chart to install an ingress controller. Search the hub to find that there are many available.

```
student@cp:~$ helm search hub ingress
```

```
URL                                                 CHART VERSION
APP VERSION               DESCRIPTION
https://artifacthub.io/packages/helm/k8s-as-hel...  1.0.2
v1.0.0                    Helm Chart representing a single Ingress Kubern...
https://artifacthub.io/packages/helm/openstack-...  0.2.1
v0.32.0                   OpenStack-Helm Ingress Controller
<output_omitted>
```

```
https://artifacthub.io/packages/helm/api/ingres...        3.29.1
0.45.0                    Ingress controller for Kubernetes using NGINX a...
https://artifacthub.io/packages/helm/wener/ingr...        3.31.0
0.46.0                    Ingress controller for Kubernetes using NGINX a...
https://artifacthub.io/packages/helm/nginx/ngin...        0.9.2
1.11.2                    NGINX Ingress Controller
<output_omitted>
```

12. We will use a popular ingress controller provided by **NGINX**.

```
student@cp:~$ helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx
```

```
"ingress-nginx" has been added to your repositories
```

```
student@cp:~$ helm repo update
```

```
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "ingress-nginx" chart repository
Update Complete. -Happy Helming!-
```

13. Download and edit the `values.yaml` file and change it to use a `DaemonSet` instead of a `Deployment`. This way there will be a pod on every node to handle traffic if using an external load balancer.

```
student@cp:~$ helm fetch ingress-nginx/ingress-nginx --untar
```

```
student@cp:~$ cd ingress-nginx
```

```
student@cp:~/ingress-nginx$ ls
```

```
CHANGELOG.md  Chart.yaml  OWNERS  README.md  ci  templates  values.yaml
```

```
student@cp:~/ingress-nginx$ vim values.yaml
```

YAML **values.yaml**

```
1  ....
2    ## DaemonSet or Deployment
3    ##
4    kind: DaemonSet                      #<-- Change to DaemonSet, around line 150
5
6    ## Annotations to be added to the controller Deployment or DaemonSet
7  ....
```

14. Now install the controller using the chart. Note the use of the dot (.) to look in the current directory.

```
student@cp:~/ingress-nginx$ helm install myingress .
```

```
NAME: myingress
LAST DEPLOYED: Wed May 19 22:24:27 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The ingress-nginx controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running
'kubectl --namespace default get services -o wide -w myingress-ingress-nginx-controller'
```

```
An example Ingress that makes use of the controller:
<output_omitted>
```

15. We now have an ingress controller running, but no rules yet. View the resources that exist. Use the **-w** option to watch the ingress controller service show up. After it is available use **ctrl-c** to quit and move to the next command.

```
student@cp:~$ kubectl get ingress --all-namespaces
```

```
No resources found
```

```
student@cp:~$ kubectl --namespace default get services -o wide  myingress-ingress-nginx-controller
```

```
NAME                                 TYPE           CLUSTER-IP      EXTERNAL-IP
   PORT(S)                      AGE   SELECTOR
myingress-ingress-nginx-controller   LoadBalancer   10.104.227.79   <pending>
   80:32558/TCP,443:30219/TCP   47s   app.kubernetes.io/component=controller,
   app.kubernetes.io/instance=myingress,app.kubernetes.io/name=ingress-nginx
```

```
student@cp:~$ kubectl get pod --all-namespaces |grep nginx
```

```
default      myingress-ingress-nginx-controller-mrqt5    1/1     Running   0      20s
default      myingress-ingress-nginx-controller-pkdxm    1/1     Running   0      62s
default      nginx-b68dd9f75-h6ww7                       1/1     Running   0      21h
```

16. Now we can add rules which match HTTP headers to services. Remember to check the course files from the tarball.

```
student@cp:~$ vim ingress.yaml
```

ingress.yaml

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    name: ingress-test
5    annotations:
6      nginx.ingress.kubernetes.io/service-upstream: "true"
7    namespace: default
8  spec:
9    ingressClassName: nginx
10   rules:
11   - host: www.example.com
12     http:
13       paths:
14       - backend:
15           service:
16             name: secondapp
17             port:
18               number: 80
19         path: /
20         pathType: ImplementationSpecific
```

17. Create then verify the ingress is working. If you don't pass a matching header you should get a 404 error.

```
student@cp:~$ kubectl create -f ingress.yaml
```

```
ingress.networking.k8s.io/ingress-test created
```

```
student@cp:~$ kubectl get ingress
```

```
NAME           CLASS    HOSTS             ADDRESS   PORTS   AGE
ingress-test   nginx    www.example.com             80      5s
```

```
student@cp:~$ kubectl get pod -o wide |grep myingress
```

```
myingress-ingress-nginx-controller-mrqt5   1/1     Running   0          8m9s    192.168.219.118
  cp    <none>           <none>
myingress-ingress-nginx-controller-pkdxm   1/1     Running   0          8m9s    192.168.219.119
  worker    <none>           <none>
```

```
student@cp:~/ingress-nginx$ curl 192.168.219.118
```

```
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

18. Check the ingress service and expect another 404 error, don't use the admission controller.

```
student@cp:~/ingress-nginx$ kubectl get svc |grep ingress
```

```
myingress-ingress-nginx-controller             LoadBalancer   10.104.227.79    <pending>
  80:32558/TCP,443:30219/TCP   10m
myingress-ingress-nginx-controller-admission   ClusterIP      10.97.132.127    <none>
  443/TCP                      10m
```

```
student@cp:~/ingress-nginx$ curl 10.104.227.79
```

```
<html>
<head><title>404 Not Found</title></head>
<body>
<center><h1>404 Not Found</h1></center>
<hr><center>nginx</center>
</body>
</html>
```

19. Now pass a matching header and you should see the default web server page.

```
student@cp:~/ingress-nginx$ curl -H "Host: www.example.com" http://10.104.227.79
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
<output_omitted>
```

20. We can add an annotation to the ingress pods for Linkerd. You will get some warnings, but the command will work.

```
student@cp:~/ingress-nginx$ kubectl get ds myingress-ingress-nginx-controller -o yaml  | \
    linkerd inject --ingress - | kubectl apply -f -
```

```
daemonset "myingress-ingress-nginx-controller" injected

Warning: resource daemonsets/myingress-ingress-nginx-controller is missing the
kubectl.kubernetes.io/last-applied-configuration annotation which is required
by kubectl apply. kubectl apply should only be used on resources created
```

```
declaratively by either kubectl create --save-config or kubectl apply. The
missing annotation will be patched automatically.
daemonset.apps/myingress-ingress-nginx-controller configured
```

21. Go to the `Top` page, change the namespace to default and the resource to `daemonset/myingress-ingress-nginx-controller`. Press start then pass more traffic to the ingress controller and view traffic metrics via the GUI. Let top run so we can see another page added in an upcoming step.

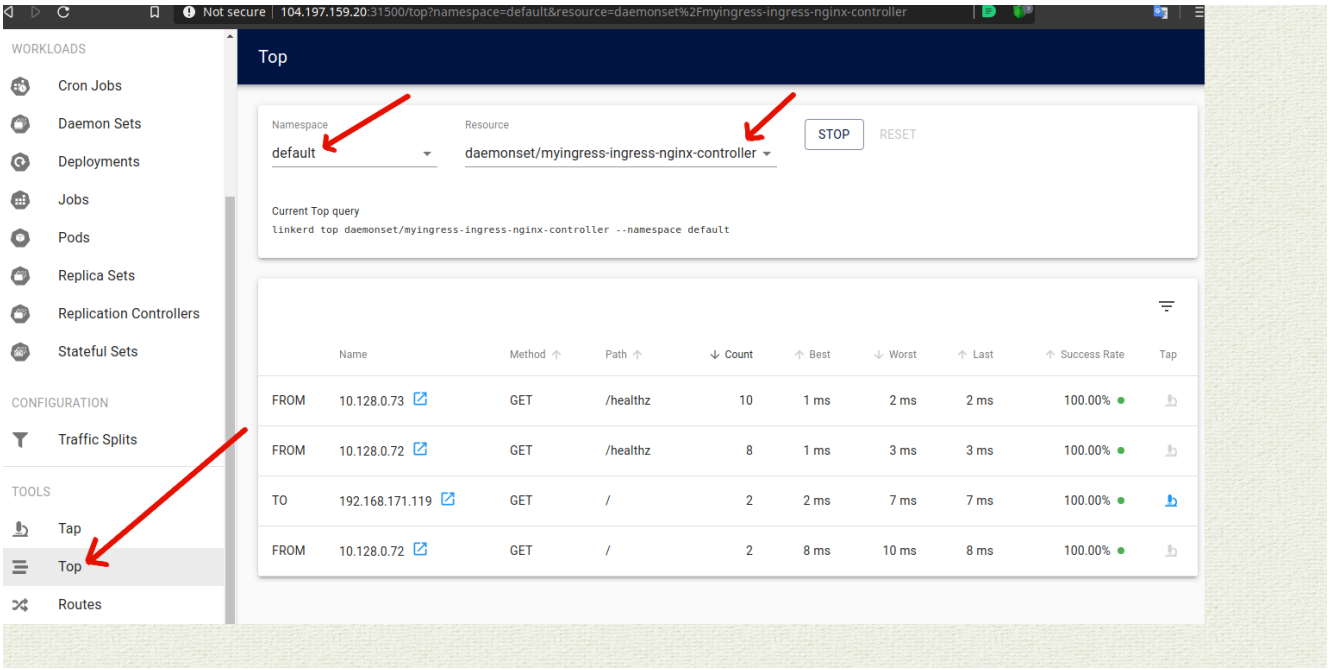

Figure 7.4: **Ingress Traffic**

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

```
student@cp:~/app2$ kubectl create deployment thirdpage --image=nginx
```

```
deployment.apps "thirdpage" created
```

23. Assign a label for the ingress controller to match against. Your pod name is unique, you can use the **Tab** key to complete the name.

```
student@cp:~/app2$ kubectl label pod thirdpage-<tab> example=third
```

24. Expose the new server as a NodePort.

```
student@cp:~/app2$ kubectl expose deployment thirdpage --port=80 --type=NodePort
```

```
service/thirdpage exposed
```

25. Now we will customize the installation. Run a bash shell inside the new pod. Your pod name will end differently. Install **vim** or an editor inside the container then edit the `index.html` file of nginx so that the title of the web page will be `Third Page`. Much of the command output is not shown below.

```
student@cp:~/app2$ kubectl exec -it thirdpage-<Tab> -- /bin/bash
```

On Container

```
root@thirdpage-:/# apt-get update

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

root@thirdpage-:/# vim /usr/share/nginx/html/index.html
```

```
<!DOCTYPE html>
<html>
<head>
<title>Third Page</title>     #<-- Edit this line
<style>
<output_omitted>
```

```
root@thirdpage-:/$ exit
```

Edit the ingress rules to point the thirdpage service. It may be easiest to copy the existing `host` stanza and edit the `host` and `name`.

26. `student@cp:~/app2$ kubectl edit ingress ingress-test`

ingress-test

```
....
spec:
  rules:
  - host: thirdpage.org
    http:
      paths:
      - backend:
          service:
            name: thirdpage
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
  - host: www.example.com
    http:
      paths:
      - backend:
          service:
            name: secondapp
            port:
              number: 80
        path: /
        pathType: ImplementationSpecific
status:
....
```

27. Test the second `Host:` setting using **curl** locally as well as from a remote system, be sure the `<title>` shows the non-default page. Use the main IP of either node. The Linkerd GUI should show a new `TO` line, if you select the small blue box with an arrow you will see the traffic is going to thirdpage.

`student@cp:~/app2$ curl -H "Host: thirdpage.org" http://10.128.0.7/`

```
<!DOCTYPE html>
<html>
<head>
```

```
<title>Third Page</title>
<style>
<output_omitted>
```

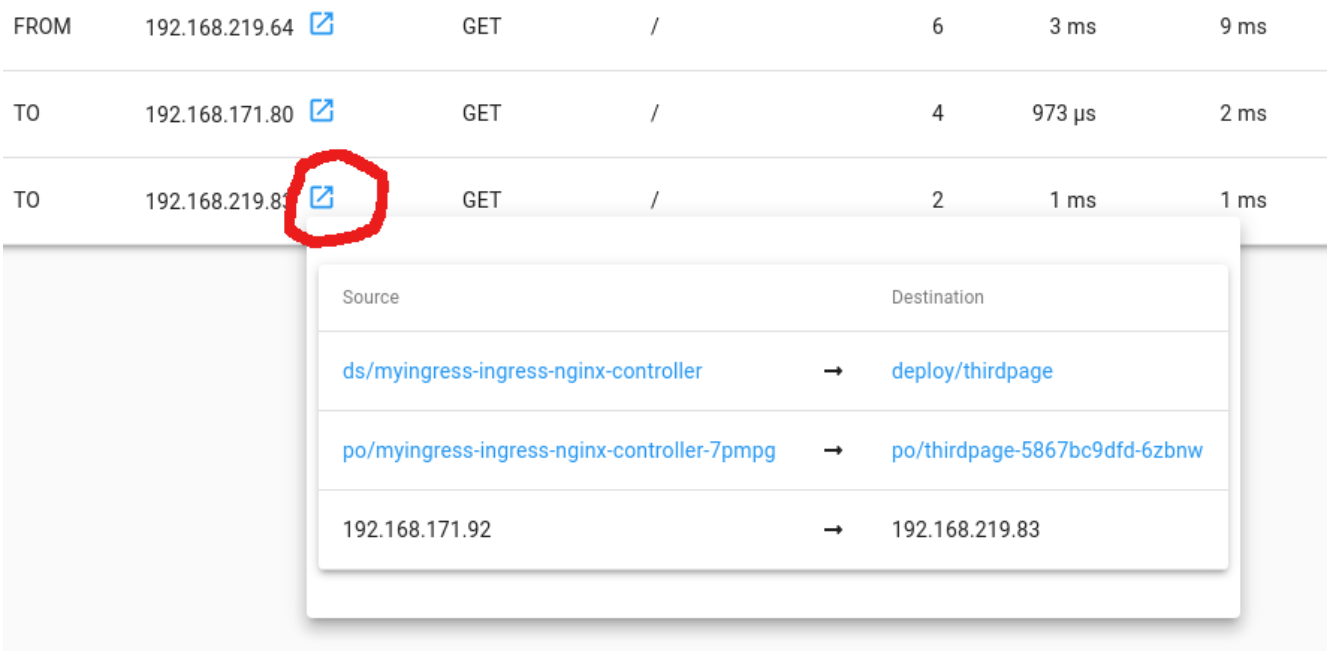

Figure 7.5: **Linkerd Top Metrics**

28. Consider how you would edit the ingress rules to point at a different server, as we did when editing the service selector. Deploy the **httpd** server again and test changing traffic from `thirdpage` over to a new `httpd` sever using only ingress rule edits.

Exercise 7.3: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.

- Use Kubernetes primitives to implement common deployment strategies (e.g. blue/gren or canary)
- Use the Helm package manager to deploy existing packages
- Provide and troubleshoot access to applications via services
- Use Ingress rules to expose applications

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you can complete each step quickly, and ensure each object is running properly.

Using the three URL locations allowed by the exam, find and be familiar with working YAML examples to do the following:

1. Create a new pod called `webone`, running the `nginx` service. Expose port 80.

2. Create a new service named `webone-svc`. The service should be accessible from outside the cluster.
3. Update both the pod and the service with selectors so that traffic for to the service IP shows the web server content.
4. Change the type of the service such that it is only accessible from within the cluster. Test that exterior access no longer works, but access from within the node works.
5. Deploy another pod, called `webtwo`, this time running the `wlniao/website` image. Create another service, called `webtwo-svc` such that only requests from within the cluster work. Note the default page for each server is distinct.
6. Install and configure an ingress controller such that requests for webone.com see the `nginx` default page, and requests for webtwo.org see the `wlniao/website` default page.
7. Remove any resources created in this review.

Chapter 8

Application Troubleshooting

Exercise 8.1: Troubleshooting: Monitor Applications

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.

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@cp/app2:~$ cd
student@cp:~$ 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@cp:~$ kubectl describe pod secondapp
```

```
Name:         secondapp
Namespace:    default
Node:         worker-wdrq/10.128.0.2
Start Time:   Fri, 13 Apr 2022 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 2022 20:34:58 +0000
    Ready:          True
    Restart Count:  0
<output_omitted>

  busy:
<output_omitted>

    State:          Running
      Started:      Sun, 15 Apr 2022 21:36:20 +0000
    Last State:     Terminated
      Reason:       Completed
      Exit Code:    0
      Started:      Sun, 15 Apr 2022 20:36:18 +0000
      Finished:     Sun, 15 Apr 2022 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, worker-wdrq  pulling
image "busybox"
  Normal  Pulled   34m (x50 over 2d)  kubelet, worker-wdrq  Successfully
pulled image "busybox"
  Normal  Created  34m (x50 over 2d)  kubelet, worker-wdrq  Created
container
  Normal  Started  34m (x50 over 2d)  kubelet, worker-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@cp:~$ kubectl logs secondapp webserver
```

```
192.168.55.0 - - [13/Apr/2022:21:18:13 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.47.0" "-"
192.168.55.0 - - [13/Apr/2022:21:20:35 +0000] "GET / HTTP/1.1" 200
612 "-" "curl/7.53.1" "-"
127.0.0.1 - - [13/Apr/2022:21:25:29 +0000] "GET" 400 174 "-" "-" "-"
127.0.0.1 - - [13/Apr/2022:21:26:19 +0000] "GET index.html" 400 174
"-" "-" "-"
<output_omitted>
```

```
student@cp:~$ kubectl logs secondapp busy
```

```
student@cp:~$
```

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, and other errors.

```
student@cp:~$ kubectl exec  -it  secondapp -c busy -- sh
```

On Container

```
/ $ nslookup www.linuxfoundation.org
```

```
/ $ nslookup www.linuxfoundation.org
Server:                10.96.0.10
Address:        10.96.0.10:53

Non-authoritative answer:
Name:        www.linuxfoundation.org
Address: 23.185.0.2

*** Can't find www.linuxfoundation.org: No answer
```

```
/ $ 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 Google's 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 don't have any typos as well.

```
student@cp:~$ 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@cp:~$ 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@cp:~$ 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@cp:~$ kubectl get ep secondapp -o yaml
```

```
apiVersion: v1
kind: Endpoints
metadata:
  creationTimestamp: 2022-04-14T05:37:32Z
<output_omitted>
```

11. If the containers, services and endpoints are working the issue may be with an infrastructure service like **kube-proxy**. Ensure it's 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@cp:~$ ps -elf |grep kube-proxy
```

```
<output_omitted>
4 S root      2652  2621  0  80   0 - 186908 -     15:23 ?        00:00:03
↪  /usr/local/bin/kube-proxy
--config=/var/lib/kube-proxy/config.conf --hostname-override=cp
0 S student  31656  9707  0  80   0 -  3715 pipe_w 19:02 pts/0    00:00:00 grep --color=auto
↪  kube-proxy
```

```
student@cp:~$ journalctl -a | grep proxy
```

```
Apr 15 15:44:43 worker-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 worker-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 worker-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 wasn't RBAC aware. This is only an example, you (hopefully) won't see any errors on your lab nodes.

    If not using command line completion, view the possible pod names first.

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

```
student@cp:~$ 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, **32000** in this case.

```
student@cp:~$ sudo iptables-save |grep secondapp
```

```
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/secondapp:" \
-m tcp --dport 32000 -j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/secondapp:" \
-m tcp --dport 32000 -j KUBE-SVC-DAASHM5XQZF5XI3E
-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
<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@cp:~$ curl localhost:32000
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<output_omitted>
```

Exercise 8.2: Update YAML File

The API continues to change, both as objects mature as well as new features and settings being added. As a result YAML that may have worked on a previous version of Kubernetes may need to be updated due to API deprecations.

1. Find the `brokendeploy.yaml` file from the course tarball. Try to create the deployment, which worked a few releases of Kubernetes prior. Create a new deployment from the command line for an understanding of what the current object requires. Edit the broken deployment YAML until you can deploy it and the pod runs.

```
student@cp:~$ find $HOME -name brokendeploy.yaml

student@cp:~$ cp <path_from_above> .

student@cp:~$ kubectl create -f brokendeploy.yaml
```

Exercise 8.3: OPTIONAL LAB: 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 **Sonobuoy** https://sonobuoy.io/ 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. **Things change quickly these steps may not work....today**

1. Download a compiled binary. A shorter URL is shown first, then the longer, just in case the link changes and you need to navigate. They should download the same file.

```
student@cp:~$ curl -sLO https://tinyurl.com/4cdeyz55

student@cp:~$ mv 4cdeyz55 sonobuoy.tar.gz

student@cp:~$ tar -xvf sonobuoy.tar.gz
```

```
LICENSE
sonobuoy
```

```
student@cp:~$ curl -sLO \
https://github.com/vmware-tanzu/sonobuoy/releases/download/v0.56.8/sonobuoy_0.56.8_linux_amd64.tar.gz
```

2. Run the test. Use the `--wait` option, which will capture the screen until the test finishes. This could take a while to finish, such as an hour. You should get some output indicating testing objects being created.

```
student@cp:~$ sudo mv sonobuoy /usr/local/bin/

student@cp:~$ sonobuoy run --wait
```

```
INFO[0000] create request issued                         name=sonobuoy namespace=
resource=namespaces
INFO[0000] create request issued                         name=sonobuoy-serviceaccount
namespace=sonobuoy resource=serviceaccounts
INFO[0000] create request issued                         name=sonobuoy-serviceaccount
-sonobuoy namespace= resource=clusterrolebindings
INFO[0000] create request issued                         name=sonobuoy-serviceaccount
-sonobuoy namespace= resource=clusterroles
INFO[0000] create request issued                         name=sonobuoy-config-cm
namespace=sonobuoy resource=configmaps
INFO[0000] create request issued                         name=sonobuoy-plugins-cm
namespace=sonobuoy resource=configmaps
INFO[0000] create request issued                         name=sonobuoy namespace=
sonobuoy resource=pods
INFO[0000] create request issued                         name=sonobuoy-aggregator
```

```
namespace=sonobuoy resource=services
14:45:09          PLUGIN       NODE     STATUS   RESULT                                PROGRESS
14:45:09             e2e     global    running           Passed:  0, Failed:  0, Remaining:354
14:45:09    systemd-logs       cp-1   complete
14:45:09    systemd-logs   worker-1   complete
14:45:09
14:45:09 Sonobuoy is still running. Runs can take 60 minutes or more depending
on cluster and plugin configuration.

<output_omitted>

16:38:29             e2e     global    running           Passed:345, Failed:  3, Remaining:  6
16:38:49             e2e     global   complete           Passed:351, Failed:  3, Remaining:  0
16:38:49 Sonobuoy plugins have completed. Preparing results for download.
16:39:09             e2e     global   complete   failed  Passed:351, Failed:  3, Remaining:  0
16:39:09    systemd-logs       cp-1   complete   passed
16:39:09    systemd-logs   worker-1   complete   passed
16:39:09 Sonobuoy has completed. Use `sonobuoy retrieve` to get results.
```

3. If you don't want to wait for the full results, open a second terminal session to the cp and use the **status** and **logs** sub commands.

```
student@cp:~$ sonobuoy status
```

```
      PLUGIN  STATUS      RESULT   COUNT
PROGRESS
       e2e    running     1   Passed: 31, Failed:  0, Remaining:323
systemd-logs  complete    2

Sonobuoy is still running. Runs can take 60 minutes or more depending on
cluster and plugin configuration.
```

```
student@cp:~$ sonobuoy logs
```

```
namespace="sonobuoy" pod="sonobuoy-systemd-logs-daemon-set-34bef90c2026439b-x7q7g"
container="sonobuoy-worker"
time="2022-07-23T14:44:51Z" level=trace msg="Invoked command single-node with args
[] and flags [level=trace logtostderr=true sleep=-1 v=6]"
time="2022-07-23T14:44:51Z" level=info msg="Waiting for waitfile"
waitfile=/tmp/sonobuoy/results/done
time="2022-07-23T14:44:51Z" level=info msg="Starting to listen on port 8099 for
progress updates and will relay them to
https://[192.168.193.162]:8080/api/v1/progress/by-node/cp-1/systemd-logs"
....
```

4. Wait until the results are ready, or log into the container and look for results.

```
student@cp:~$ sonobuoy retrieve
```

```
202207231444_sonobuoy_7d9c5bb5-e547-4827-bb2e-07e12f8fed25.tar.gz
```

5. View the results

```
student@cp:~$ sonobuoy results 202207231444_sonobuoy_7d9c5bb5-e547-4827-bb2e-07e12f8fed25.tar.gz
```

```
Plugin: e2e
Status: failed
Total: 6971
Passed: 351
Failed: 3
Skipped: 6617
```

```
Failed tests:
[sig-scheduling] SchedulerPredicates [Serial] validates that NodeSelector is respected if not
↪  matching  [Conformance]
<output_omitted>
```

6. Delete the pods and namespaces created by Sonobuoy.

```
student@cp:~$ sonobuoy delete --wait
```

```
```

Exercise 8.4: Domain Review

Very Important

The source pages and content in this review could change at any time. **IT IS YOUR RESPONSIBILITY TO CHECK THE CURRENT INFORMATION**.

Revisit the CKAD domain list on Curriculum Overview and locate some of the topics we have covered in this chapter.

- Understand API deprecations
- Use provided tools to monitor Kubernetes applications
- Debugging in Kubernetes

Focus on ensuing you have all the necessary files and processes understood first. Repeat the review until you are sure you have bookmarks of necessary YAML samples and can complete each step quickly, and ensure each object is running properly.

1. Find and use the `troubleshoot-review1.yaml` file to create a deployment. The **create** command will fail. Edit the file to fix issues such that a single pod runs for at least a minute without issue. There are several things to fix.

```
student@cp:~$ kubectl create -f troubleshoot-review1.yaml
```

```
<Fix any errors found here>
```

When fixed it should look like this:

```
student@cp:~$ kubectl get deploy igottrouble
```

```
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
igottrouble   1/1     1            1           5m13s
```

2. Remove any resources created during this review.