Although installing and configuring a Kubernetes cluster is not one of the objectives for the CKAD exam, it is important to get some hands-on experience with the concepts covered in this course. Therefore, it is useful to have a Kubernetes cluster with which you can experiment and try out the things that will be covered throughout the course. This lesson will guide you through the process of building a basic cluster that you can experiment with as you proceed.

Lesson Reference

If you want to follow along, there is a reference for the commands used in this lesson below.

On all 3 servers

First, set up the Docker and Kubernetes repositories:

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

sudo add-apt-repository    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) \
   stable"

curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

cat << EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list
deb https://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

Install Docker and Kubernetes packages:

Note that if you want to use a newer version of Kubernetes, change the version installed for kubelet, kubeadm, and kubectl. Make sure all three use the same version.

Note: There is currently a bug in Kubernetes 1.13.4 (and earlier) that can cause problems installaing the packages. Use 1.13.5-00 to avoid this issue.

```
sudo apt-get update

sudo apt-get install -y docker-ce=18.06.1~ce~3-0~ubuntu kubelet=1.14.5-00 kubeadm=1.14.5-00 kubectl=1.14.5-00

sudo apt-mark hold docker-ce kubelet kubeadm kubectl
```

Enable iptables bridge call:

```
echo "net.bridge.bridge-nf-call-iptables=1" | sudo tee -a /etc/sysctl.conf

sudo modprobe br_netfilter

sudo sysctl -p
```

On the Kube master server

Initialize the cluster:

```
sudo nano /proc/sys/net/ipv4/ip_forward
(Change from 0 to 1)

sudo kubeadm init --pod-network-cidr=10.244.0.0/16
```

Set up local kubeconfig:

```
mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Install Flannel networking:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/bc79dd1505b0c8681ece4de4c0d86c5cd2643275/Dc
```

Note: If you are using Kubernetes 1.16 or later, you will need to use a newer flannel installation yaml instead:

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/3f7d3e6c24f641e7ff557ebcea1136fdf4b1b6a1/Dc
```

On each Kube node server

Join the node to the cluster. Do this by copying the provided line from the output when initializing the master node. Keep in mind that when copying the command, the system will add a newline character if it stretches over multiple lines in the web terminal. To get around this, copy the command to a text editor and make sure it fits on one entire line. It should look something like the following:

```
sudo kubeadm join $controller_private_ip:6443 --token $token --discovery-token-ca-cert-hash $hash
```

On the Kube master server

Verify that all nodes are joined and ready:

```
kubectl get nodes
```

You should see all three servers with a status of Ready:

```
NAME                       STATUS   ROLES    AGE   VERSION
wboyd1c.mylabserver.com    Ready    master   54m   v1.13.4
wboyd2c.mylabserver.com    Ready    <none>   49m   v1.13.4
wboyd3c.mylabserver.com    Ready    <none>   49m   v1.13.4
```

Management of configuration data is one of the challenges involved in building and maintaining complex application infrastructures. Luckily, Kubernetes offers functionality that helps to maintain application configurations in the form of ConfigMaps. In this lesson, we will discuss what ConfigMaps are, how to create them, some of the ways that ConfigMap data can be passed in to containers running within Kubernetes Pods.

Relevant Documentation

- https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/

Lesson Reference

Here's an example of a yaml descriptor for a ConfigMap containing some data:

```
apiVersion: v1
kind: ConfigMap
metadata:
   name: my-config-map
data:
   myKey: myValue
   anotherKey: anotherValue
```

Passing ConfigMap data to a container as an environment variable looks like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-configmap-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo $(MY_VAR) && sleep 3600"]
    env:
    - name: MY_VAR
      valueFrom:
        configMapKeyRef:
          name: my-config-map
          key: myKey
```

It's also possible to pass ConfigMap data to containers, in the form of file using a mounted volume, like so:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-configmap-volume-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo $(cat /etc/config/myKey) && sleep 3600"]
    volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        name: my-config-map
```

In the lesson, we'll also use the following commands to explore how the ConfigMap data interacts with pods and containers:

```
kubectl logs my-configmap-pod

kubectl logs my-configmap-volume-pod

kubectl exec my-configmap-volume-pod -- ls /etc/config

kubectl exec my-configmap-volume-pod -- cat /etc/config/myKey
```

Occasionally, it's necessary to customize how containers interact with the underlying security mechanisms present on the operating systems of Kubernetes nodes. The `securityContext` attribute in a pod specification allows for making these customizations. In this lesson, we will briefly discuss what the securityContext is, and demonstrate how to use it to implement some common functionality.

Relevant Documentation

- https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

Lesson Reference

First, create some users, groups, and files on both worker nodes which we can use for testing.

```
sudo useradd -u 2000 container-user-0
sudo groupadd -g 3000 container-group-0
sudo useradd -u 2001 container-user-1
sudo groupadd -g 3001 container-group-1
sudo mkdir -p /etc/message/
echo "Hello, World!" | sudo tee -a /etc/message/message.txt
sudo chown 2000:3000 /etc/message/message.txt
sudo chmod 640 /etc/message/message.txt
```

On the controller, create a pod to read the message.txt file and print the message to the log.

```
vi my-securitycontext-pod.yml
```

Content of the YAML File

```
apiVersion: v1
kind: Pod
metadata:
  name: my-securitycontext-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "cat /message/message.txt && sleep 3600"]
    volumeMounts:
    - name: message-volume
      mountPath: /message
  volumes:
  - name: message-volume
    hostPath:
      path: /etc/message
```

Check the pod's log to see the message from the file:

```
kubectl logs my-securitycontext-pod
```

Delete the pod and re-create it, this time with a `securityContext` set to use a user and group that do not have access to the file.

```
kubectl delete pod my-securitycontext-pod --now
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-securitycontext-pod
spec:
  securityContext:
    runAsUser: 2001
    fsGroup: 3001
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "cat /message/message.txt && sleep 3600"]
    volumeMounts:
    - name: message-volume
      mountPath: /message
  volumes:
  - name: message-volume
    hostPath:
      path: /etc/message
```

Check the log again. You should see a "permission denied" message.

```
kubectl logs my-securitycontext-pod
```

Delete the pod and re-create it again, this time with a user and group that are able to access the file.

```
kubectl delete pod my-securitycontext-pod --now
```

```
apiVersion: v1
kind: Pod
metadata:
  name: my-securitycontext-pod
spec:
  securityContext:
    runAsUser: 2000
    fsGroup: 3000
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "cat /message/message.txt && sleep 3600"]
    volumeMounts:
    - name: message-volume
      mountPath: /message
  volumes:
  - name: message-volume
    hostPath:
      path: /etc/message
```

Check the log once more. You should see the message from the file.

```
kubectl logs my-securitycontext-pod
```

Kubernetes is a powerful tool for managing and utilizing available resources to run containers. Resource requests and limits provide a great deal of control over how resources will be allocated. In this lesson, we will talk about what resource requests and limits do, and also demonstrate how to set resource requests and limits for a container.

Relevant Documentation

- https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#resource-requests-and-limits-of-pod-and-container

Lesson Reference

Specify resource requests and resource limits in the container spec like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-resource-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

One of the challenges in managing a complex application infrastructure is ensuring that sensitive data remains secure. It is always important to store sensitive data, such as tokens, passwords, and keys, in a secure, encrypted form. In this lesson, we will talk about Kubernetes secrets, a way of securely storing data and providing it to containers. We will also walk through the process of creating a simple secret, and passing the sensitive data to a container as an environment variable.

Relevant Documentation

- https://kubernetes.io/docs/concepts/configuration/secret/

Lesson Reference

Create a secret using a yaml definition like this. It is a good idea to delete the yaml file containing the sensitive data after the secret object has been created in the cluster.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
stringData:
  myKey: myPassword
```

Once a secret is created, pass the sensitive data to containers as an environment variable:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-secret-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]
    env:
    - name: MY_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: myKey
```

Kubernetes allows containers running within the cluster to interact with the Kubernetes API. This opens the door to some powerful forms of automation. But in order to ensure that this gets done securely, it is a good idea to use specialized ServiceAccounts with restricted permissions to allow containers to access the API. In this lesson, we will discuss ServiceAccounts as they pertain to pod configuration, and we will walk through the process of specifying which ServiceAccount a pod will use to connect to the Kubernetes API.

Relevant Documentation

- https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin/
- https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

Lesson Reference

Creating a ServiceAccount looks like this:

```
kubectl create serviceaccount my-serviceaccount
```

Use the `serviceAccountName` attribute in the pod spec to specify which ServiceAccount the pod should use:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-serviceaccount-pod
spec:
  serviceAccountName: my-serviceaccount
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]
```

Multi-container pods provide an opportunity to enhance containers with helper containers that provide additional functionality. This lesson covers the basics of what multi-container pods are and how they are created. It also discusses the primary ways that containers can interact with each other within the same pod, as well as the three main multi-container pod design patterns: sidecar, ambassador, and adapter.

Be sure to check out the hands-on labs for this course (including the practice exam) to get some hands-on experience with implementing multi-container pods.

Relevant Documentation

- https://kubernetes.io/docs/concepts/cluster-administration/logging/#using-a-sidecar-container-with-the-logging-agent
- https://kubernetes.io/docs/tasks/access-application-cluster/communicate-containers-same-pod-shared-volume/
- https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/

Lesson Reference

Here is the YAML used to create a simple multi-container pod in the video:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.15.8
    ports:
    - containerPort: 80
  - name: busybox-sidecar
    image: busybox
    command: ['sh', '-c', 'while true; do sleep 30; done;']
```

Kubernetes is often able to detect problems with containers and respond appropriately without the need for specialized configuration. But sometimes we need additional control over how Kubernetes determines container status. Kubernetes probes provide the ability to customize how Kubernetes detects the status of containers, allowing us to build more sophisticated mechanisms for managing container health. In this lesson, we discuss liveness and readiness probes in Kubernetes, and demonstrate how to create and configure them.

Relevant Documentation

- https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes
- https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/

Lesson Reference

Here is a pod with a liveness probe that uses a command:

`my-liveness-pod.yml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-liveness-pod
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', "echo Hello, Kubernetes! && sleep 3600"]
    livenessProbe:
      exec:
        command:
        - echo
        - testing
      initialDelaySeconds: 5
      periodSeconds: 5
```

Here is a pod with a readiness probe that uses an http request:

`my-readiness-pod.yml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-readiness-pod
spec:
  containers:
  - name: myapp-container
    image: nginx
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 5
```

When managing containers, obtaining container logs is sometimes necessary in order to gain insight into what is going on inside a container. Kubernetes offers an easy way to view and interact with container logs using the `kubectl logs` command. In this lesson, we discuss container logs and demonstrate how to access them using `kubectl logs`.

Relevant Documentation

- https://kubernetes.io/docs/concepts/cluster-administration/logging/

Lesson Reference

A sample pod that generates log output every second:

```
apiVersion: v1
kind: Pod
metadata:
  name: counter
spec:
  containers:
  - name: count
    image: busybox
    args: [/bin/sh, -c, 'i=0; while true; do echo "$i: $(date)"; i=$((i+1)); sleep 1; done']
```

Get the container's logs:

```
kubectl logs counter
```

For a multi-container pod, specify which container to get logs for using the `-c` flag:

```
kubectl logs <pod name> -c <container name>
```

Save container logs to a file:

```
kubectl logs counter > counter.log
```

Monitoring is an important part of managing any application infrastructure. In this lesson, we will discuss how to view the resource usage of pods and nodes using the `kubectl top` command.

Relevant Documentation

- https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

Lesson Reference

Here are some sample pods that can be used to test `kubectl top`. They are designed to use approximately 300m and 100m CPU, respectively.

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-consumer-big
spec:
  containers:
  - name: resource-consumer
    image: gcr.io/kubernetes-e2e-test-images/resource-consumer:1.4
    resources:
      requests:
        cpu: 500m
        memory: 128Mi
  - name: busybox-sidecar
    image: radial/busyboxplus:curl
    command: [/bin/sh, -c, 'until curl localhost:8080/ConsumeCPU -d "millicores=300&durationSec=3600"; do sle
```

```
apiVersion: v1
kind: Pod
metadata:
  name: resource-consumer-small
spec:
  containers:
  - name: resource-consumer
    image: gcr.io/kubernetes-e2e-test-images/resource-consumer:1.4
    resources:
      requests:
        cpu: 500m
        memory: 128Mi
  - name: busybox-sidecar
    image: radial/busyboxplus:curl
    command: [/bin/sh, -c, 'until curl localhost:8080/ConsumeCPU -d "millicores=100&durationSec=3600"; do sle
```

Here are the commands used in the lesson to view resource usage data in the cluster:

```
kubectl top pods
kubectl top pod resource-consumer-big
kubectl top pods -n kube-system
kubectl top nodes
```

Problems will occur in any system, and Kubernetes provides some great tools to help locate and fix problems when they occur within a cluster. In this lesson, we will go through the process of debugging an issue in Kubernetes. We will use our knowledge of `kubectl get` and `kubectl describe` to locate a broken pod, and then explore various ways of editing Kubernetes objects to fix issues.

Relevant Documentation

- https://kubernetes.io/docs/tasks/debug-application-cluster/debug-application/
- https://kubernetes.io/docs/tasks/debug-application-cluster/debug-pod-replication-controller/
- https://kubernetes.io/docs/tasks/debug-application-cluster/debug-service/

Lesson Reference

I prepared my cluster before the video by creating a broken pod in the `nginx-ns` namespace:

```
kubectl create namespace nginx-ns
```

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: nginx-ns
spec:
  containers:
  - name: nginx
    image: nginx:1.158
```

Exploring the cluster to locate the problem

```
kubectl get pods

kubectl get namespace

kubectl get pods --all-namespaces

kubectl describe pod nginx -n nginx-ns
```

Fixing the broken image name

Edit the pod:

```
kubectl edit pod nginx -n nginx-ns
```

Change the container image to `nginx:1.15.8`.

Exporting a descriptor to edit and re-create the pod.

Export the pod descriptor and save it to a file:

```
kubectl get pod nginx -n nginx-ns -o yaml --export > nginx-pod.yml
```

Add this liveness probe to the container spec:

```
livenessProbe:
  httpGet:
    path: /
    port: 80
```

Delete the pod and recreate it using the descriptor file. Be sure to specify the namespace:

```
kubectl delete pod nginx -n nginx-ns

kubectl apply -f nginx-pod.yml -n nginx-ns
```

Kubernetes labels provide a way to attach custom, identifying information to your objects. Selectors can then be used to filter objects using label data as criteria. Annotations, on the other hand, offer a more freeform way to attach useful but non-identifying metadata. In this lesson, we will discuss labels, selectors, and annotations. We will also demonstrate how to use them in a cluster.

Relevant Documentation

- https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/
- https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/

Lesson Reference

Here is a pod with some labels.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-production-label-pod
  labels:
    app: my-app
    environment: production
spec:
  containers:
  - name: nginx
    image: nginx
```

You can view existing labels with `kubectl describe`.

```
kubectl describe pod my-production-label-pod
```

Here is another pod with different labels.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-development-label-pod
  labels:
    app: my-app
    environment: development
spec:
  containers:
  - name: nginx
    image: nginx
```

You can use various selectors to select different subsets of objects.

```
kubectl get pods -l app=my-app

kubectl get pods -l environment=production

kubectl get pods -l environment=development

kubectl get pods -l environment!=production

kubectl get pods -l 'environment in (development,production)'

kubectl get pods -l app=my-app,environment=production
```

Here is a simple pod with some annotations.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-annotation-pod
  annotations:
    owner: terry@linuxacademy.com
    git-commit: bdab0c6
spec:
  containers:
  - name: nginx
    image: nginx
```

Like labels, existing annotations can also be viewed using `kubectl describe`.

```
kubectl describe pod my-annotation-pod
```

Deployments provide a variety of features to help you automatically manage groups of replica pods. In this lesson, we will discuss what deployments are. We will also create a simple deployment and go through the process of scaling the deployment up and down by changing the number of desired replicas.

Relevant Documentation

- https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Lesson Reference

Here is a simple deployment for three replica pods running nginx.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

You can explore and manage deployments using the same kubectl commands you would use for other object types.

```
kubectl get deployments

kubectl get deployment <deployment name>

kubectl describe deployment <deployment name>

kubectl edit deployment <deployment name>

kubectl delete deployment <deployment name>
```

One powerful feature of Kubernetes deployments is the ability to perform rolling updates and rollbacks. These allow you to push out new versions without incurring downtime, and they allow you to quickly return to a previous state in order to recover from problems that may arise when deploying changes. In this lesson, we will discuss rolling updates and rollback, and we will demonstrate the process of performing them on a deployment in the cluster.

Relevant Documentation

- https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment
- https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#rolling-back-a-deployment

Lesson Reference

Here is a sample deployment you can use to practice rolling updates and rollbacks.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolling-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.1
        ports:
        - containerPort: 80
```

Perform a rolling update.

```
kubectl set image deployment/rolling-deployment nginx=nginx:1.7.9 --record
```

Explore the rollout history of the deployment.

```
kubectl rollout history deployment/rolling-deployment

kubectl rollout history deployment/rolling-deployment --revision=2
```

You can roll back to the previous revision like so.

```
kubectl rollout undo deployment/rolling-deployment
```

You can also roll back to a specific earlier revision by providing the revision number.

```
kubectl rollout undo deployment/rolling-deployment --to-revision=1
```

You can also control how rolling updates are performed by setting `maxSurge` and `maxUnavailable` in the deployment spec:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rolling-deployment
spec:
  strategy:
    rollingUpdate:
      maxSurge: 3
      maxUnavailable: 2
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.1
        ports:
        - containerPort: 80
```

Kubernetes provides the ability to easily run container workloads in a distributed cluster, but not all workloads need to run constantly. With jobs, we can run container workloads until they complete, then shut down the container. CronJobs allow us to do the same, but re-run the workload regularly according to a schedule. In this lesson, we will discuss Jobs and CronJobs and explore how to create and manage them.

Relevant Documentation

- https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/
- https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/
- https://kubernetes.io/docs/tasks/job/automated-tasks-with-cron-jobs/

Lesson Reference

This Job calculates the first 2000 digits of pi.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

You can use `kubectl get` to list and check the status of Jobs.

```
kubectl get jobs
```

Here is a CronJob that prints some text to the console every minute.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

You can use `kubectl get` to list and check the status of CronJobs.

```
kubectl get cronjobs
```

Deployments make it easy to create a set of replica pods that can be dynamically scaled, updated, and replaced. However, providing network access to those pods for other components is difficult. Services provide a layer of abstraction that solves this problem. Clients can simply access the service, which dynamically proxies traffic to the current set of replicas. In this lesson, we will discuss services and demonstrate how to create one that exposes a deployment's replica pods.

Relevant Documentation

- https://kubernetes.io/docs/concepts/services-networking/service/
- https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/

Lesson Reference

Create a deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Expose the deployment's replica pods with a service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
  - protocol: TCP
    port: 8080
    targetPort: 80
```

You can get more information about the service with these commands:

```
kubectl get svc
kubectl get endpoints my-service
```

From a security perspective, it is often a good idea to place network-level restrictions on any communication between different parts of your infrastructure. NetworkPolicies allow you to restrict and control the network traffic going to and from your pods. In this lesson, we will discuss NetworkPolicies and demonstrate how to create a simple policy to restrict access to a pod.

Relevant Documentation

- https://kubernetes.io/docs/concepts/services-networking/network-policies/

Lesson Reference

In order to use NetworkPolicies in the cluster, we need to have a network plugin that supports them. We can accomplish this alongside an existing flannel setup using canal:

```
wget -O canal.yaml https://docs.projectcalico.org/v3.5/getting-started/kubernetes/installation/hosted/canal/c

kubectl apply -f canal.yaml
```

Create a sample nginx pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: network-policy-secure-pod
  labels:
    app: secure-app
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Create a client pod which can be used to test network access to the Nginx pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: network-policy-client-pod
spec:
  containers:
  - name: busybox
    image: radial/busyboxplus:curl
    command: ["/bin/sh", "-c", "while true; do sleep 3600; done"]
```

Use this command to get the cluster IP address of the Nginx pod:

```
kubectl get pod network-policy-secure-pod -o wide
```

Use the secure pod's IP address to test network access from the client pod to the secure Nginx pod:

```
kubectl exec network-policy-client-pod -- curl <secure pod cluster ip address>
```

Create a network policy that restricts all access to the secure pod, except to and from pods which bear the `allow-access: "true"` label:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
spec:
  podSelector:
    matchLabels:
      app: secure-app
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          allow-access: "true"
    ports:
    - protocol: TCP
      port: 80
  egress:
  - to:
    - podSelector:
        matchLabels:
          allow-access: "true"
    ports:
    - protocol: TCP
      port: 80
```

Get information about NetworkPolicies in the cluster:

```
kubectl get networkpolicies
kubectl describe networkpolicy my-network-policy
```

Container storage is designed to be as temporary as the containers themselves. However, sometimes we need storage that is able to survive beyond the short life of a container. Kubernetes volumes allow us to mount storage to a container that isn't in the container itself. In this lesson, we will discuss volumes and demonstrate how to mount a simple volume to a container in a pod.

Relevant Documentation

- https://kubernetes.io/docs/concepts/storage/volumes/

Lesson Reference

This pod mounts a simple emptyDir volume, `my-volume`, to the container at the path `/tmp/storage`.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
  - image: busybox
    name: busybox
    command: ["/bin/sh", "-c", "while true; do sleep 3600; done"]
    volumeMounts:
    - mountPath: /tmp/storage
      name: my-volume
  volumes:
  - name: my-volume
    emptyDir: {}
```

PersistentVolumes (PVs) and PersistentVolumeClaims (PVCs) provide a way to easily consume storage resources, especially in the context of a complex production environment that uses multiple storage solutions. In this lesson, we will talk about implementing persistent storage using PersistentVolumes and PersistentVolumeClaims, and we will demonstrate how to set up a PV and PVC to consume storage resources in a pod.

Relevant Documentation

- https://kubernetes.io/docs/concepts/storage/persistent-volumes/
- https://kubernetes.io/docs/tasks/configure-pod-container/configure-persistent-volume-storage/

Lesson Reference

Create the PersistentVolume:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: my-pv
spec:
  storageClassName: local-storage
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data"
```

Create the PersistentVolumeClaim:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: local-storage
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 512Mi
```

We can use `kubectl` to check the status of existing PVs and PVCs:

```
kubectl get pv
kubectl get pvc
```

Create a pod to consume storage resources using a PVC:

```
kind: Pod
apiVersion: v1
metadata:
  name: my-pvc-pod
spec:
  containers:
  - name: busybox
    image: busybox
    command: ["/bin/sh", "-c", "while true; do sleep 3600; done"]
    volumeMounts:
```

```yaml
      - mountPath: "/mnt/storage"
        name: my-storage
    volumes:
    - name: my-storage
      persistentVolumeClaim:
        claimName: my-pvc
```