CKAD

2024 edition

Pearson

Intro

- This session is provided by Sander van Vugt
- It is a completely reworked class, based on my on-demand class "CKAD 4th edition (available august 2024)"
- Today is the first time I'm running this completely reworked class, it may contain bugs
- Participants are expected to know how to run an application in Kubernetes
- To follow along, use Minikube inside an Ubuntu Desktop environment
- Other environments may work, but are not supported

Poll Question 1

Rate your Kubernetes knowledge

- none
- poor
- average
- good
- more than good

Poll Question 2

- Where are you from?
- Middle East
- Africa
- India
- Asia (other)
- North/Central America
- South America
- Pacific region
- Europe

Agenda

Day 1

- Creating Custom Images
- Managing Pod Properties
- Running Applications
- Managing Application Access
- Managing Network Access

Day2

- Application observability and maintenance
- Custom Resources
- Storage
- ConfigMaps and Secrets
- DevOps Technologies
- Application Security

Using an environment in this course

- Recommended: create an Ubuntu-based minikube environment as described in the setup guide in https://github.com/sandervanvugt/microservices
- Alternatively: use O'Reilly sandbox, but functionality will be missing or different
- Note that the slide numbering corresponds to the numbering in CKAD 4th edition

Creating Custom Images

Lesson 2: Managing Container Images

2.4 Using Dockerfile to Build Custom Images

Using Dockerfile

- Dockerfile can be provided by application developers.
- In Podman environments, Dockerfile is referred to as Containerfile, there are no functional differences.
- It's also relatively easy to write your own.
- To build an image from a Dockerfile, use **docker built -t imagename .**
- In this command, **-t** (tag) specifies the name of the image you want to create.
- **.** refers to the current directory as the directory where the Dockerfile is found.

Demo: Build an Image from Dockerfile

- **cd ckad**
- **cat Dockerfile**
- **docker build -t myapp .**
- **docker images**
- **docker image inspect myapp**
- **docker run myapp**

Lesson 2: Managing Container Images

2.5 Creating Images from Running Containers

Demo: Creating Images with **docker commit**

- **docker run --name customweb -it nginx sh**
  - **touch /tmp/testfile**
  - **exit**
- **docker commit customweb nginx:custom**
- **docker images**
- **docker run -it localhost/nginx:custom /tmp/testfile**

Managing Pod Properties

Lesson 5: Pod Basic Features

5.6 Namespaces

Namespaces

- Kubernetes Namespace resources leverage Linux kernel namespaces to provide resource isolation.
- Different Namespaces can be used to strictly separate between customer resources and thus enable multi-tenancy.
- Namespaces are used to apply different security-related settings,
  - Role-Based Access Control (RBAC)
  - Quota
- By installing complex Kubernetes applications in their own Namespace, managing them is easier.

Managing Namespaces

- To show resources in all Namespaces, use **kubectl get ... -A**
- To run resources in a specific Namespace, use **kubectl run ... -n namespace**
- Use **kubectl create ns nsname** to create a Namespace.

Demo: Namespaces

- **kubectl get pods**
- **kubectl get pods -A**
- **kubectl create ns secret**
- **kubectl run pod secretpod --image=nginx -n secret**
- **kubectl get pods -n secret**

Lesson 6: Pod Advanced Features

6.1 init Containers

init Containers

- An init container is a special case of a multi-container Pod, where the init container runs to completion before the main container is started.
- Starting the main container depends on the success of the init container, if the init container fails the main container will never start.

Lesson 6: Pod Advanced Features

6.2 Sidecar Containers

Sidecar Containers

- A sidecar container is an initContainer that has the restartPolicy field set to Always.
- It doesn't occur as a specific attribute, to create a sidecar you need to create an initContainer with the restartPolicy set to Always.
- The sidecar container will be started before the main Pod is started and is typically used to repeatedly run a command.
- Like a regular initContainer, the sidecar container must complete once before the main Pod is started.

Lesson 6: Pod Advanced Features

6.4 restartPolicy

restartPolicy

- The Pod restartPolicy determines what happens if a container that is managed by a Pod crashes.
- If set to the default value restartPolicy=always, the container will be restarted after a crash.
- restartPolicy=always does not affect the state of the entire Pod.
- If the Pod is stopped or killed, restartPolicy=always won't restart it.

Demo: restartPolicy

- kubectl run nginx1 --image=nginx
- kubectl get pods nginx1 -o yaml | grep restartP
- kubectl delete pods nginx1
- kubectl get pods
- kubectl run nginx2 --image=nginx
- minikube ssh
- crictl ps | grep nginx
- crictl stop $(crictl ps | awk '/nginx1/ { print $1 }')
- exit
- kubectl get pods

Lesson 6: Pod Advanced Features

6.5 Jobs

Jobs

- A Job starts a Pod with the restartPolicy set to never.
- To create a Pod that runs to completion, use Jobs instead.
- Jobs are useful for one-shot tasks, like backup, calculation, batch processing, and more.
- Use spec.ttlSecondsAfterFinished to clean up completed Jobs automatically.

Job Types

3 different Job types can be started, which is specified by the completions and parallelism parameters:

- Non-parallel Jobs: one Pod is started, unless the Pod fails
  - completions=1
  - parallelism=1
- Parallel Jobs with a fixed completion count: the Job is complete after successfully running as many times as specified in jobs.spec.completions
  - completions=n
  - parallelism=m
- Parallel Jobs with a work queue: multiple Jobs are started, when one completes successfully, the Job is complete
  - completions=1
  - parallelism=n

Demo: Using Jobs

- **kubectl create job onejob --image=busybox -- date**
- **kubectl get jobs, pods**
- **kubectl get pods onejob-xxx -o yaml | grep restartPolicy**
- **kubectl delete job onejob**
- **kubectl create job mynewjob --image=busybox --dry-run=client -o yaml -- sleep 5 > mynewjob.yaml**
- Edit mynewjob.yaml and include the following in job.spec
  - completions: 3
  - ttlSecondsAfterFinished: 60
- **kubectl apply -f mynewjob.yaml**

Lesson 6: Pod Advanced Features

6.6 CronJobs

CronJobs

- Jobs are used to run a task a specific number of times.
- A CronJob adds a schedule to a Job.
- To add the schedule, Linux crontab syntax is used.
- When running a CronJob, a Job will be scheduled.
- This Job, on its turn, will start a Pod.
- To test a CronJob, use **kubectl create job myjob --from=cronjob/mycronjob**

Demo: Running CronJobs

- **kubectl create cronjob -h | less**
- **kubectl create cronjob runme --image=busybox --schedule="*/2 * * * *" -- echo greetings from your cluster**
- **kubectl create job runme --from=cronjob/runme**
- **kubectl get cronjobs,jobs,pods**
- **kubectl logs runme-xxx-yyy**
- **kubectl delete cronjob runme**

Lesson 15: Security

15.5 Resource Requirements, Limits, and Quota

Understanding Resources

- Resource requests can be set for containers in a Pod to ensure that the Pod is only scheduled on cluster nodes that meet the resource requests.
  - Use pod.spec.containers.resources.requests to set
- Resource limits can be set for Pods to maximize the use of system resources.
  - Use pod.spec.containers.resources.limits to define
- Quota are restrictions that can be set on a Namespace to maximize the availability of resources within that Namespace.
- To set resource requests and limits you don't have to use Quota.
- If a Namespace has Quota, all Pods running in that Namespace must have resources set.

Understanding Resource Limitations

- Memory as well as CPU limits can be used.
- CPU limits are expressed in millicore or millicpu, 1/1000 of a CPU core.
  - So, 500 millicore is 0.5 CPU
- When being scheduled, the kube-scheduler ensures that the node running the Pods has all requested resources available.
- If a Pod with resource limits cannot be scheduled, it will show a status of Pending.
- Use **kubectl set resources ...** to apply resource limits to running applications in deployments (covered later).

Understanding Quota

- Quota are restrictions that are applied to Namespaces.
- If Quota are set on a Namespace, applications started in that Namespace must have resource requests and limits set.
- Use **kubectl create quota ... -n mynamespace** to apply Quota

Demo: Using Resource Requests and Limits

- **kubectl create -f frontend-resources.yaml**
- **kubectl get pods**
- **kubectl describe pod frontend**
- **kubectl delete -f frontend-resources.yaml**

Demo: Using Quota

- **kubectl create ns restricted**
- **kubectl create quota myquota -n restricted --hard=cpu=2,--memory=1G,pods=3**
- **kubectl describe ns restricted**
- **kubectl run pod restrictedpod --image=nginx -n restricted** # will fail
- **kubectl create deploy restricteddeploy --image=nginx -n restricted**
- **kubectl set resources -n restricted deploy restricteddeploy --limits=cpu=200m,memory=2G**
- **kubectl describe -n restricted deploy restricteddeploy**
- **kubectl set resources -n restricted deploy restricteddeploy --limits=cpu=200m,memory=128M --requests=cpu=100m,memory=64M**

Running Applications

Lesson 8: Deployments

8.4 Deployment Updates

Understanding Application Updates

- Depoyments make updating applications easier.
- To manage how applications are updated, an update strategy is used:
  - strategy.type.rollingUpdate updates application instances in batches to ensure application functionality continues to be offered at any time
  - As a result of rollingUpdate, during the update different versions of the application will be running
  - For applications that don't support offering multiple versions simultaneously, set strategy.type.recreate
  - The recreate strategy brings down all application instances, after which the new application version is brought up.

Managing Rolling Updates

- To manage how rollingUpdate will happen, two parameters are used:
  - maxSurge specifies how many application instances can be running during the update above the regular number of application instances.
  - maxUnavailable defines how many application instances can be temporarily unavailable.
- Both parameters take an absolute number or a percentage as their argument.

Demo: Managing Updates

- **kubectl create deploy upapp --image=nginx:1.17 --replicas=5**
- **kubectl get deploy upapp -o yaml | grep -A5 strategy**
- **kubectl set image deploy/upapp nginx=nginx:1.18; kubectl get all --selector app=upapp**
- **kubectl edit deploy upapp**
  - change strategy.type to Recreate
- **kubectl set image deploy/upapp nginx=nginx:1.19; kubectl get all --selector app=upapp**

Lesson 8: Deployments

8.5 Deployment History

Understanding Deployment History

- During the Deployment update procedure, the Deployment creates a new ReplicaSet that uses the new properties.
- The old ReplicaSet is kept, but the number of Pods will be set to 0.
- This makes it easy to roll back to the previous state.
- **kubectl rollout history** will show the rollout history of a specific deployment, which can easily be reverted as well.
- Use **kubectl rollout history deployment mynginx --revision=1** to observe changes between versions.

Demo: Managing Rollout History

- **kubectl create –f rolling.yaml**
- **kubectl rollout history deployment**
- **kubectl edit deployment rolling-nginx** # change version to 1.15
- **kubectl rollout history deployment**
- **kubectl describe deployments rolling-nginx**
- **kubectl rollout history deployment rolling-nginx --revision=2**
- **kubectl rollout history deployment rolling-nginx --revision=1**
- **kubectl rollout undo deployment rolling-nginx --to-revision=1**

Managing Application Access

Lesson 10: Networking

10.2 Services

Services

- A Service is an API resource that is used to expose a set of Pods.
- Services are applying round-robin load balancing to forward traffic to specific Pods.
- The set of Pods that is targeted by a Service is determined by a selector (which is a label).
- The kube-controller-manager will continuously scan for Pods that match the selector and include these in the Service.
- If Pods are added or removed, they immediately show up in the Service.

Services and Decoupling

- Services exist independently from the applications they provide access to.
- The Service needs to be created independently of the application, and after removing an application, it also needs to be removed separately.
- The only thing they do is watch for Pods that have a specific label set matching the selector that is specified in the service.
- That means that one Service can provide access to Pods in multiple Deployments, and while doing so, Kubernetes will automatically load balance between these Pods.
- This strategy is used in canary Deployments (covered later).

Service Types

- ClusterIP: this default type exposes the service on an internal cluster IP address.
- NodePort: allocates a specific port on the node that forwards to the service IP address on the cluster network.
- LoadBalancer: provisions an external load balancer to handle incoming traffic to applications in public cloud.
- ExternalName: works on DNS names; redirection is happening at a DNS level, which is useful in migration.
- Headless: a Service used in cases where direct communication with Pods is required, which is used in StatefulSet.

For CKAD, focus on ClusterIP and NodePort.

Lesson 10: Networking

10.3 Creating Services

Creating Services

- **kubectl expose** can be used to create Services, providing access to Deployments, ReplicaSets, Pods or other services.
- In most cases **kubectl expose** exposes a Deployment, which allocates its Pods as the service endpoint.
- **kubectl create service** can be used as an alternative solution to create Services.
- While creating a Service, the **--port** argument must be specified to indicate the port on which the Service will be listening for incoming traffic.

Service Ports

- While working with Services, different ports are specified:
  - targetPort: the port on the application (container) that the service addresses.
  - port: the port on which the Service is accessible
  - nodePort: the port that is exposed externally while using the NodePort Service type.

Demo: Creating Services

```
kubectl create deployment nginxsvc --image=nginx
kubectl scale deployment nginxsvc --replicas=3
kubectl expose deployment nginxsvc --port=80
kubectl describe svc nginxsvc # look for endpoints
kubectl get svc nginxsvc -o=yaml
kubectl get svc
kubectl get endpoints
```

Demo: Creating Services

- **minikube ssh**
- **curl http://svc-ip-address**
- **exit**
- **kubectl edit svc nginxsvc**

```
  …
  protocol: TCP
  nodePort: 32000
type: NodePort
```

- **kubectl get svc**
- (from host): **curl http://$(minikube ip):32000**

Lesson 10: Networking

10.5 Services and DNS

Services and DNS

- Exposed Services automatically register with the Kubernetes internal coredns DNS server.
- The standard DNS name is composed as servicename.namespace.svc.clustername
- As a result, Pods within the same Namespace can access servicename by using its short name.
- To access servicenames in other Namespaces, the fully qualified domain name must be used.

Demo: Services and DNS

- **kubectl describe svc -n kube-system kubernetes**
- **kubectl create ns elsewhere**
- **kubectl run nginxpod -n elsewhere**
- **kubectl expose -n elsewhere nginxpod --port=80**
- **kubectl run testpod --image=busybox -- sleep infinity**
- **kubectl exec -it testpod -- cat /etc/resolv.conf**
- **kubectl exec -it testpod -- wget --spider --timeout=1 nginxpod** # fails
- **kubectl exec -it testpod -- wget --spider --timeout=1 nginxpod.elsewhere.svc.cluster.local**

Lesson 11: Ingress and Gateway API

11.1 Managing Incoming Traffic

Managing Incoming Traffic

- For a long time, Ingress has been the solution to manage incoming traffic.
- Recently, Ingress has gone into a "feature freeze" and will be replaced by Gateway API.
- Currently, Ingress is still in the exam objectives, this is expected to be replaced with Gateway API in the future.

Lesson 11: Ingress and Gateway API

11.2 Ingress Components

Understanding Ingress

- Ingress is used to provide external access to internal Kubernetes cluster resources.
- To do so, Ingress uses an external load balancer.
- This load balancer is implemented by the Ingress controller which is running as a Kubernetes application.
- As an API resource, Ingress uses Services to connect to Pods that are used as a service endpoint.
- To access resources in the cluster, the host name resolution (DNS or /etc/hosts) must be configured to resolve to the Ingress load balancer IP.

Understanding Ingress

- Ingress exposes HTTP and HTTPS routes from outside the cluster to Pods within the cluster.
- Traffic routing is controlled by rules defined on the Ingress resource.
- Ingress can be configured to do the following:
  - Give Services externally-reachable URLs
  - Load balance traffic
  - Terminate SSL/TLS
  - Offer name based virtual hosting

Lesson 11: Ingress and Gateway API

11.4 Using the Minikube Ingress Controller

Minikube Ingress

- Minikube is a Kubernetes distribution and comes with addons to integrate third-party solutions.
- Use **minikube addons list** to show available addons.
- Use **minikube addons enable** to enable a specific addon.

Demo: Using the Minikube Ingress Addon

- **minikube addons list**
- **minikube addons enable ingress**
- **kubectl get ns**
- **kubectl get all -n ingress-nginx**

Lesson 11: Ingress and Gateway API

11.5 Using Ingress

Demo: Configuring Ingress Rules

- **kubectl create deploy nginxsvc --image=nginx --port=80**
- **kubectl expose deploy nginxsvc**
- **kubectl create ingress nginxsvc-ingress --rule="/=nginxsvc:80" --rule="/hello=newdep:8080"**
- **echo "$(minikube ip) nginxsvc.info" >> /etc/hosts**
- **kubectl describe ing nginxsvc-ingress**
- **curl nginxsvc.info**
- **kubectl create deployment newdep --image=gcr.io/google-samples/hello-app:2.0**
- **kubectl expose deployment newdep --port=8080**
- **curl nginxsvc.info/hello**

Managing Network Access

Lesson 10: Networking

10.6 NetworkPolicy

NetworkPolicy

- By default, there are no restrictions to network traffic in K8s.
- Pods can always communicate, even if they're in other Namespaces.
- To limit this, NetworkPolicies can be used.
- NetworkPolicies need to be supported by the network plugin though,
  - The Weave plugin does NOT support network policies!
  - Calico is a common plugin that does support NetworkPolicy.
- If in a policy there is no match, traffic will be denied.
- If no NetworkPolicy is used, all traffic is allowed.

NetworkPolicy Identifiers

- In NetworkPolicy, three different identifiers can be used:
  - podSelector: specifies a label to match Pods.
  - namespaceSelector: used to grant access to specific namespaces.
  - ipBlock: marks a range of IP addresses that is allowed. notice that traffic to and from the node where a Pod is running is always allowed.
- When defining a Pod- or Namespace-based NetworkPolicy, a selector label is used to specify what traffic is allowed to and from the Pods that match the selector.
- NetworkPolicies do not conflict, they are additive.

Demo: Using NetworkPolicy

- **kubectl get pods -n kube-system | grep -i calico**
- **kubectl apply -f nwpolicy-complete-example.yaml**
- **kubectl expose pod nginx --port=80**
- **kubectl exec -it busybox -- wget --spider --timeout=1 nginx** will fail
- **kubectl label pod busybox access=true**
- **kubectl exec -it busybox -- wget --spider --timeout=1 nginx** will work

Application Observability

Lesson 17: Observability

17.3 Kubernetes API Health Endpoints

Health Probes

- To monitor if an application still is working as expected, health probes can be used.
- As a common practice, applications can be programmed to provide access to the /healthz endpoint to test application availability.
- The kube-apiserver itself exposes three endpoints to test that it is working:
  - /healthz: returns "ok" if the API server is healthy
  - /livez: indicates if the API server is alive
  - /readyz: indicates if the API server is ready to service requests
- Use **curl -k https://$(minikube ip):8443/healthz** to test, it should return "ok" as result.
- Similar endpoints may be provided by any web-based application.

Lesson 17: Observability

17.4 Using Probes to Monitor Applications

Understanding Probes

- The probe itself is a simple test that is defined as a container property, which is often a command.
- Probes are used to test if the application that uses it is still functional.
- If the probe doesn't respond, the application is restarted.
- The following probe test types are defined in pods.spec.container:
  - exec: a command is executed and returns a zero exit value.
  - httpGet: an HTTP request returns a response code between 200 and 399.
  - tcpSocket: connectivity to a TCP socket (available port) is successful.
- Probes can be configured with a failureTreshold to determine how long it can take the application to react.

Probe Types

- Kubernetes can use 3 different probe types:
  - livenessProbe: checks if the application is alive. Container will be restarted if the probe test fails.
  - readinessProbe: checks if the application is ready to service requests. Container will be removed from the list of available services if it fails.
  - startupProbe: used to verify initial startup of the application, useful if startup can be slow. No other probes are used before this probe finishes successfully.

Custom Resources

Lesson 14: Working with the API

14.3 Understanding API Deprecations

API Deprecations

- With new Kubernetes releases, old API versions may get deprecated.
- If an old version gets deprecated, it will be supported for a minimum of two more Kubernetes releases.
- When you see a deprecation message, make sure to take action and change your YAML manifest files!

Demo: Dealing with Deprecations

- **kubectl create -f redis-deploy.yaml**
- **kubectl api-versions**
- **kubectl explain --recursive deploy**

Lesson 14: Working with the API

14.4 Extending the API

Extending the API

- The Kubernetes API can be extended in different ways,
  - Using the CustomResourceDefinition API resource
  - Using Custom Controllers
  - Using API Aggregation

Lesson 14: Working with the API

14.5 CustomResourceDefinitions

Understanding CustomResourceDefinitions

- CustomResourceDefinitions allow users to add custom resources to clusters.
- Doing so allows anything to be integrated in a cloud-native environment.
- The crd allows users to add resources in a very easy way
    - The resources are added as extension to the original Kubernetes API server.
    - No programming skills required.

Creating Custom Resources

- Creating Custom Resources using crds is a two-step procedure.
  - First, you'll need to define the resource, using the CustomResourceDefinition API kind.
  - After defining the resource, it can be added through its own API resource.

Demo: Creating Custom Resources

- **cat crd-object.yaml**
- **kubectl create -f crd-object.yaml**
- **kubectl api-resources | grep backup**
- **cat crd-backup.yaml**
- **kubectl create -f crd-backup.yaml**
- **kubectl get backups**

Storage

Lesson 7: Kubernetes Storage

7.1 Ephemeral and Persistent Storage

Understanding Ephemeral Storage

- When a container is started, the container working environment is created as a directory on the host that runs the container.
- In this directory, a subdirectory is created to store changes inside the container.
- This subdirectory is ephemeral and disappears when the container disappears.
- The ephemeral storage is host-bound, and that doesn't work well in a cloud environment where multiple application instances are running.

Cloud Storage Needs

- To provide persistent storage, the store needs to be stored separately.
- Also, cloud storage should not be host-bound.
- When cloud storage is host-bound, it needs to be syncrhonized when replicated Pods run on different nodes.
- Pod volumes are a Pod property that allow containers to connect to any storage type that is defined within the Pod.
- PersisentVolumes are independent API resources and can be discovered dynamically while running Pods.

Lesson 7: Kubernetes Storage

7.2 Configuring Pod Volume Storage

Pod Volumes

- Pod volumes are defined as properties of Pods.
- Many types of storage can be addressed using volumes: see pod.spec.volumes for a list.
- Using Pod volumes works if Pods are used in an environment where a specific type of storage is used.
- For more flexibility, PersistentVolumes can be used.
- To use a Pod volume, the container needs to mount it, using pod.spec.containers.volumeMounts.
- There is no easy command to create a Pod with volumes, use the documentation to set it up.

Common Pod Volume Types

- emptyDir creates a temporary directory on the host that runs a Pod and is ephemeral.
- hostPath refers to a persistent directory on the host that runs the Pod.
- PersistentVolumeClaim connects to available PersistentVolumes (covered later).
- Other volume types such as fc and iscsi may make more sense in real life, but require additional setup (and for that reason are not on CKAD).

Demo: Creating a Pod with a Volume

- From a browser, got to https://kubernetes.io/docs and search for "configure a volume for a pod". This will show the redis.yaml file, which sets up redis with emptyDir storage (this file is also provided in the course Git repository).
- Run the redis Pod from the documentation, using **kubectl apply -f https://k8s.io/examples/pods/storage/redis.yaml**
- Use **kubectl describe pods redis** and check its configuration, which contains emptyDir storage, mounted on /data/redis.
- Use **kubectl exec -it redis -- touch /data/redis/helloworld**
- Use **minikube ssh** to access your Minikube host.
- Type **crictl stop $(crictl ps | awk '/redis/ { print $1 })'** to force a Pod restart.
- Use **exit** to exit the minikube shell.

Demo: Creating a Pod with a Volume

- Use **kubectl get pods** to see that the redis Pod is restarted.
- Type **kubectl exec -it redis -- ls -l /data/redis** to verify the helloworld file still exists.
- Use **minikube ssh**
- From there: **sudo find / -name "helloworld" 2>/dev/null**
- **exit**
- Use **kubectl delete --force pod redis**
- Open another **minikube ssh** session to verify that the **sudo find / -name "helloworld" 2>/dev/null** command doesn't give any results.

Lesson 7: Kubernetes Storage

7.3 Configuring PersistentVolumes

Understanding Persistent Storage

- A Pod volume can use a persistent storage type.
- A PersistentVolume is a specific API resource that defines the storage.
- Pods connect to PersistentVolumes using the PersistentVolumeClaim API Resource.
- The benefit of using PersistentVolumes is decoupling: the Pod doesn't connect to a specific storage type, but will pick up what is available.
- This is useful in DevOps environment, where different types of storage may be available for different environments.

Creating PersistentVolumes

- There is no easy way to create PersistentVolumes from the command line: search for "Create a persistentvolume" in the documentation.
- In many environments, PersistentVolumes are created automatically, using StorageClass resource and an automatic storage provisioner (covered later).
- When setting up PersistentVolumes manually, make sure they have the storageClassName property set.
- This property is used to connect to the PersistentVolume from a PersistentVolumeClaim.

Understanding storageClassName

- storageClassName can be used to group different types of storage:
  - Use storageClassName: preprod for preproduction storage.
  - Use storageClassName: prod for production storage.
- The storageClassName property is also used for storage that has automatically been created by a StorageClass.
- While requesting storage using PersistentVolumeClaim, storageClassName must be specified to bind to a specific type of storage.

Demo: Defining PersistentVolumes

- From the documentation, search for "Create a PersistentVolume" where you will find the pv-volume.yaml example file (also provided in this course Git repository).
- Use **kubectl apply -f https://k8s.io/examples/pods/storage/pv-volume.yaml** to create the PersistentVolume.
- Use **kubectl describe pv task-pv-volume** to learn about its properties.
- We'll later use a PersistentVolumeClaim to use this storage.

Lesson 7: Kubernetes Storage

7.4 StorageClass

StorageClass

- StorageClass works with a storage provisioner to create PersistentVolumes on-demand.
- Storage provisioners are not a part of vanilla Kubernetes, they are provided by the ecosystem and may be integrated in a Kubernetes distribution.
- The storage provisioner is an application that runs in Kubernetes to communicate with site-specific storage to create storage on-demand.
- Without storage provisioner, the StorageClass won't do anything.
- Configuring a StorageClass is not required in CKAD.

Demo: Exploring StorageClass

- **minikube addons list**
- **kubectl get storageclass**
- **kubectl describe storageclass**

Lesson 7: Kubernetes Storage

7.5 Configuring PersistentVolumeClaims

PersistentVolumeClaims

- The PersistentVolumeClaim (PVC) resource defines a request for storage.
- The purpose of using PVC is to bind to storage provided by a PersistentVolume at a specific site, without caring about its exact type.
- A PVC request for storage uses the following attributes:
  - storageClassName: used as a selector label
  - accessModes: ReadWriteOnce, ReadWriteMany or ReadOnly
  - resources: the required size of storage
- If a storageClassName is not defined, the PVC will only bind to PersistentVolumes created by a StorageClass.

Demo: Configuring PVCs

- From the documentation, search for "Create a PersistentVolumeClain" where you will find the pv-claim.yaml example file (also provided in this course Git repository).
- Use **kubectl apply -f https://k8s.io/examples/pods/storage/pv-claim.yaml**
- Type **kubectl get pvc,pv** and verify the claim is bound to a PersistentVolume.
- Notice that this PersistentVolume is selected based on the storageClassName attribute.

Lesson 7: Kubernetes Storage

7.6 Configuring Pod Storage with PV and PVC

Configuring Pod Storage

- Within the Pod, you'll configure a volume that uses the persistentVolumeClaim type.
- This volume is mounted using the volumeMounts container property.
- The PersistentVolumeClaim is defined separately - you might want to include it in the same YAML file.

Demo: Configuring Persistent Storage

- From the documentation, in the section "Configure a Pod to Use a PersistentVolume for Storage", you'll find the pv-pod.yaml file.
- Use **kubectl apply -f https://k8s.io/examples/pods/storage/pv-pod.yaml**
- Use **kubectl describe pod task-pv-pod** to see the Pod properties.
- Write a file to the persistent storage: **kubectl exec task-pv-pod -- touch /usr/share/nginx/html/testfile**
- Use **kubectl describe pv pv-volume** to find where the file has been written.

ConfigMaps and Secrets

Lesson 13: ConfigMaps and Secrets

13.2 Providing Variables to Kubernetes Applications

Providing Variables to Kubernetes Applications

- Providing variables while starting applications is not useful in fully automated environments,
  - Configuration as Code strategies require the variables to be included in configuration files
- Kubernetes does not offer a command line option to provide variables while running a Deployment with **kubectl create deploy**,
  - First, use **kubectl create deploy mydb --image=mariadb**
  - Next, use **kubectl set env deploy mydb MYSQL_ROOT_PASSWORD=password**
- While running a Pod, environment variables can be provided, but you shouldn't run naked Pods,
  - **kubectl run mydb --image=mysql --env="MYSQL_ROOT_PASSWORD=password"**

Demo: Generating a YAML File with Variables

- **kubectl create deploy mydb --image=mariadb**
- **kubectl describe pods mydb-xxx-yyy**
- **kubectl logs mydb-xxx-yyy**
- **kubectl set env deploy mydb MYSQL_ROOT_PASSWORD=password**
- **kubectl get deploy mydb -o yaml > mydb.yaml** # don't forget to clean it up!

Lesson 13: ConfigMaps and Secrets

13.3 Providing Variables with ConfigMaps

Understanding ConfigMaps

- The ConfigMap is an API resource to store site specific information.
- It has two different uses:
  - Variable storage
  - Configuration file(s) storage up to a size of 1MiB
- If bigger amounts of data are needed, they should be stored in a Pod Volume.

Using Variables in ConfigMaps

- Use **kubectl create cm** to create a ConfigMap,
  - **--from-literal key=value**
  - **--from-env-file=/path/to/file**
- An environment file is a file that has multiple variables defined on different lines.
- Add **--dry-run=client -o yaml** to generate YAML code instead of creating the resources.

Using Variables from ConfigMaps

- The easy way to use variables from ConfigMaps is using **kubectl set env**.
- **kubectl set env --from=configmap/mycm deploy/mydeploy**
- While using **kubectl set env**, the **--prefix** option can be used to put a prefix before the variable as defined in the ConfigMap.

Understanding ConfigMap Variable Use

- pod.spec.containers.env.name defines the variable name.
- pod.spec.containers.env.valueFrom.configMapKeyRef.name refers to the name of the ConfigMap.
- pod.spec.containers.env.valueFrom.configMapKeyRef.key defines the key in the ConfigMap from which the value must be set to the variable.

Demo: Working with ConfigMaps

- **kubectl create deploy mydb --image=mariadb --replicas=3**
- **kubectl create cm mydbvars --from-literal=ROOT_PASSWORD=password**
- **kubectl set env deploy/mydb --from configmap/mydbvars --prefix=MARIADB_**
- **kubectl get deploy mydb -o yaml | grep env -A 5**

Lesson 13: ConfigMaps and Secrets

13.4 Providing Configuration Files Using ConfigMaps

Using Configuration Files in ConfigMaps

- ConfigMap can contain one or more configuration files.
- In the data section of the ConfigMap, each file is referred to with its own key.
- To use configuration files from ConfigMaps, the ConfigMap needs to be used as a Pod volume and mounted on a directory.

Demo: Using ConfigMaps for Configuration Files

- **echo "hello world" > index.html**
- **kubectl create cm myindex --from-file=index.html**
- **kubectl describe cm myindex**
- **kubectl create deploy myweb --image=nginx**
- **kubectl edit deploy myweb**

  spec.template.spec

  ```
  volumes:
  - name: cmvol
    configMap:
      name: myindex
  ```

  spec.template.spec.containers

  ```
  volumeMounts:
  - mountPath: /usr/share/nginx/html
    name: cmvol
  ```

Lesson 13: ConfigMaps and Secrets

13.5 Secrets

Understanding Secrets

- A Secret is a base-64 encoded alternative for a ConfigMap.
- Secret types are used for typical scenarios:
  - generic: used for generic sensitive values like passwords
  - tls: stores TLS keys
  - docker-registry: used to store registry access credentials
- Using Secrets makes Kubernetes more secure, as the actual value itself doesn't have to be stored in the application manifest file.
- Using Secrets doesn't make the data completely secure, as the values are encoded and not encrypted.
- Kubernetes internally works with Secrets to deal with sensitive values.

Lesson 13: ConfigMaps and Secrets

13.6 Configuring Applications to Use Secrets

Using Secrets in Applications

- There are different use cases for using Secrets in applications:
  - To provide TLS keys to the application: **kubectl create secret tls my-tls-keys --cert=tls/my.crt --key=tls/my.key**
  - To provide security to passwords: **kubectl create secret generic my-secret-pw --from-literal=password=verysecret**
  - To provide access to an SSH private key: **kubectl create secret generic my-ssh-key --from-file=ssh-private-key=.ssh/id_rsa**
  - To provide access to sensitive files, which would be mounted in the application with root access only: **kubectl create secret generic my-secre-file --from-file=/my/secretfile**

Using Secrets in Applications

- As a Secret basically is an encoded ConfigMap, it is used in a similar way to using ConfigMaps in applications.
- If it contains variables, use **kubectl set env**.
- If it contains files, mount the Secret.
- While mounting the Secret in the Pod spec, consider using defaultMode to set the permissionmode: ...volumes.secret.defaultMode: 0400.
- Notice that mounted Secrets are automatically updated in the application when the Secret is updated.

Demo: Using a Secret to Provide Passwords

- **kubectl create secret generic dbpw --from-literal=ROOT_PASSWORD=password**
- **kubectl describe secret dbpw**
- **kubectl get secret dbpw -o yaml**
- **kubectl create deploy mynewdb --image=mariadb**
- **kubectl set env deploy mynewdb --from=secret/dbpw --prefix=MYSQL_**

DevOps Technologies

Lesson 12: Deploying Applications the DevOps Way

12.3 Canary Deployments

Understanding Canary Deployments

- A Canary Deployment upgrade strategy will expose a new version of the application to a limited number of users before completing the migration to the new version.
- This allows user exposure with a minimized risk.
- If things don't work out well, it's easy to revert to the previous situation by just removing the new application instance(s).

Service versus Ingress Canary

- Canary Deployments can be configured based on Services or Ingress.
- Using Ingress is preferred as the application picks up the change without reconnecting.
- Service-based Canary Deployments are configured to use a common selector label on the old as well as the new applications.
- Ingress-based Canary Deployments are using two Ingress resources pointing to the same Ingress virtual host.
- Canary Deployment solutions are also offered by alternative ecosystem solutions.

Demo: Service-based Canary Deployments

- **sed -i -e 's/new/old/' canary.yaml**
- **kubectl apply -f canary.yaml**
- **sed -i -e 's/old/new/' canary.yaml**
- **sed -i -e 's/replicas: 3/replicas: 1/' canary.yaml**
- **kubectl apply -f canary.yaml**
- **kubectl expose deploy old-version --name=theapp --port=80 --selector type=canary --type=NodePort**
- **kubectl get svc**
- **curl $(minikube ip):<nodeport>** # repeat at least 10 times

Application Security

Lesson 15: Security

15.1 Authentication and Authorization

Understanding Authentication

- Authentication is about where Kubernetes users come from.
- In vanilla Kubernetes and Minikube, a local Kubernetes admin account is used for authentication.
- In more advanced setups, you can create your own user accounts (covered in CKA).
- The **kubectl** config specifies to which cluster to authenticate.
  - Use **kubectl config view** to see current settings.
- The config is read from ~/.kube/config.

Understanding Authorization

- Authorization is what these users can do.
- Behind authorization, there is Role Based Access Control (RBAC) to take care of the different options.
- Use **kubectl auth can-i …** (like **kubectl auth can-i get pods**) to find out what you can do.

Lesson 15: Security

15.2 API Access and ServiceAccounts

Understanding ServiceAccounts

- All actions in a Kubernetes Cluster need to be authenticated and authorized.
- ServiceAccounts are used for basic authentication from within the Kubernetes cluster.
- RBAC is used to connect a ServiceAccount to a specific Role.
- Every Pod uses the default ServiceAccount to contact the API server.
- This default ServiceAccount allows a resource to get information from the API server, but not much else.
- Each ServiceAccount uses a Secret to automount API credentials.

Custom ServiceAccount Use Case

- Most Pods do fine with the default ServiceAccount.
- If a Pod needs access to resources in the cluster, a custom ServiceAccount that uses a RoleBinding to connect to a specific Role is needed.
- For instance, this is needed for network plugins, monitoring software and other additional components installed in Kubernetes.

Demo: Exploring ServiceAccounts

- **kubectl describe pod anypod** #look for the ServiceAccount
- **kubectl get sa -n default**
- **kubectl describe pod coredns -n kube-system** #look for ServiceAccount
- **kubectl get sa -n kube-system**

Lesson 15: Security

15.3 Role Based Access Control (RBAC)

Understanding RBAC

- RBAC uses 3 components to grant permissions to API objects.
  - A Role consists of Verbs which assign specific permissions like view, edit, and more.
  - A ServiceAccount is used by Pods that need access to API resources.
  - The RoleBinding connects a ServiceAccount to a Role.
- Roles and RoleBindings have a Namespace scope, ClusterRoles and ClusterRoleBindings have a cluster scope.
- In RBAC, users can be used for people that need access to specific resources (not covered here).

Demo: Configuring RBAC

- **kubectl create ns bellevue**
- **kubectl create role viewer --verb=get --verb=list --verb=watch --resource=pods -n bellevue**
- **kubectl create sa viewer -n bellevue**
- **kubectl create rolebinding --serviceaccount=bellevue:viewer --role=viewer -n bellevue**
- **kubectl create deploy viewginx --image=nginx --replicas=3 -n bellevue**
- **kubectl set serviceaccount deployment viewginx viewer -n bellevue**

Demo: Exploring RBAC Usage

- **kubectl describe serviceaccount coredns -n kube-system**
- **kubectl describe clusterrolebinding system:coredns**
- **kubectl describe clusterrole system:coredns**

Lesson 15: Security

15.4 SecurityContext

Understanding SecurityContext

A SecurityContext defines privilege and access control settings for a Pod or container, and includes the following:

- allowPrivilegeEscalation: whether or not a container can run with root privileges
- capabilities: POSIX capabilities used by the container
- runAsNotRoot: enforces a non-privileged UID
- readOnlyFilesystem: no writes to the container filesystems
- runAsUser: runs as a specific user
- seLinuxOptions: specifies SELinux context labels

Use **kubectl explain pod.spec.[containers.]securityContext** for a complete overview.

Using SecurityContext

- Notice that SecurityContext can be applied to Pods as well as containers.
- When SecurityContext prevents a Pod from running successfully, use **kubectl describe** to get additional information from the events.
- Expect Pods that fail because of SecurityContext restrictions to show a status of Pending.

Demo: Using SecurityContext

- **kubectl apply -f securitycontextdemo2.yaml**
- **kubectl exec -it security-context-demo -- sh**
  - **ps**
  - **cd /data; ls -l**
  - **cd demo; echo hello > testfile**
  - **ls -l**
  - **id**

Demo: Using SecurityContext

- **kubectl apply –f securitycontextdemo.yaml**
- **kubectl get pods shows pending**
- **kubectl get pods nginxsecure –o yaml** (wait 2 minutes)
- **kubectl describe pods nginxsecure**
  - note that the image wants to run as root, which is not allowed, which is why the Pod will never run.