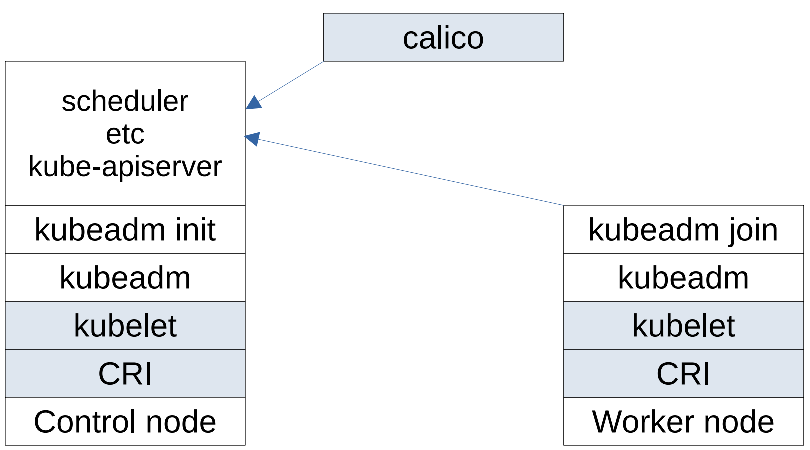
Common Kubernetes Distributions

- Canonical Kubernetes
- Google Anthos
- Rancher
- Red Hat OpenShift
- AKS
- GKE
- Minikube
- EKS
- Kind

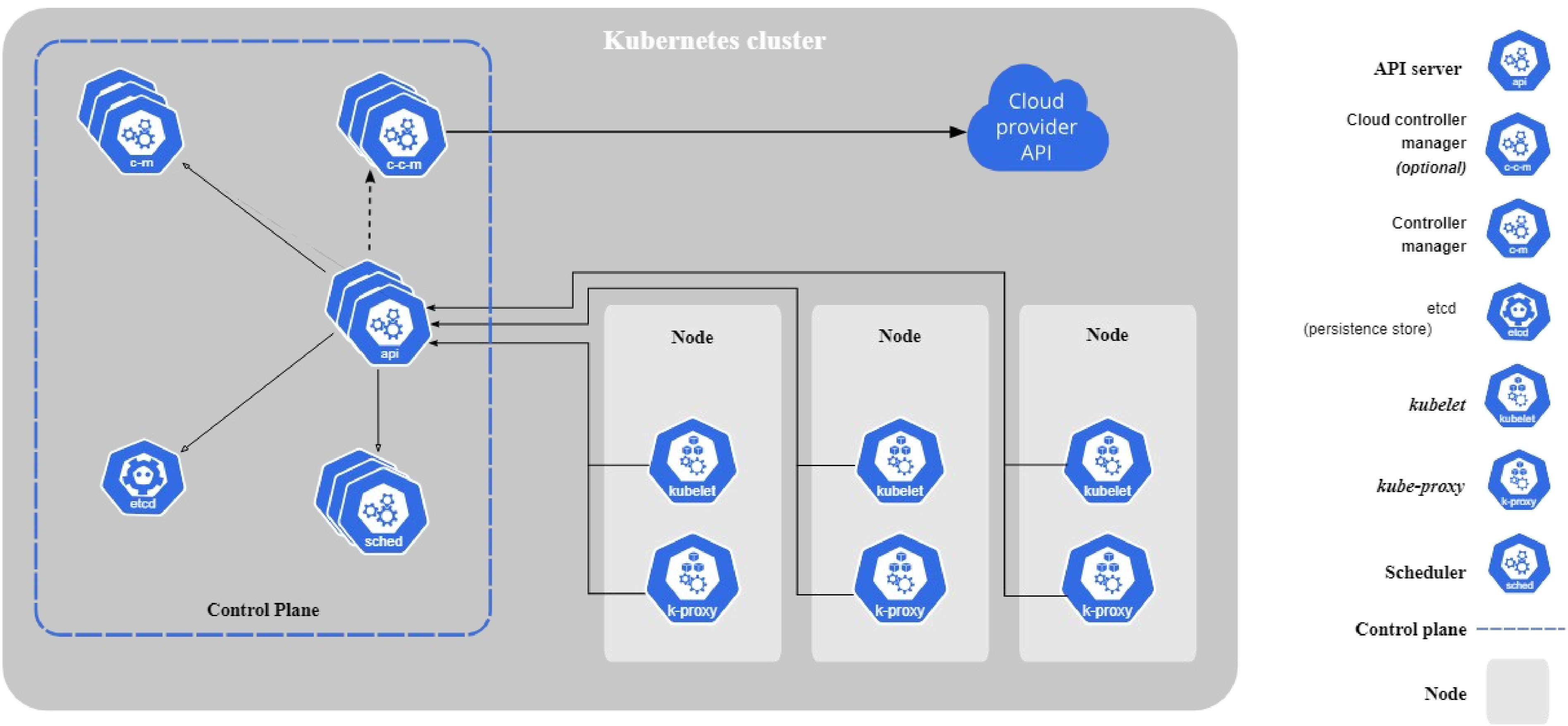
On-Premise Kubernetes installation sequence

- CRI - container runtime interface
- kubelet - to schedule pods on top of nodes
- kubectl (kubeadm)
- kubeadm init - to build cluster on
- kubeadm join - to connect to k8s services from worker node
- calico - plugin for networking



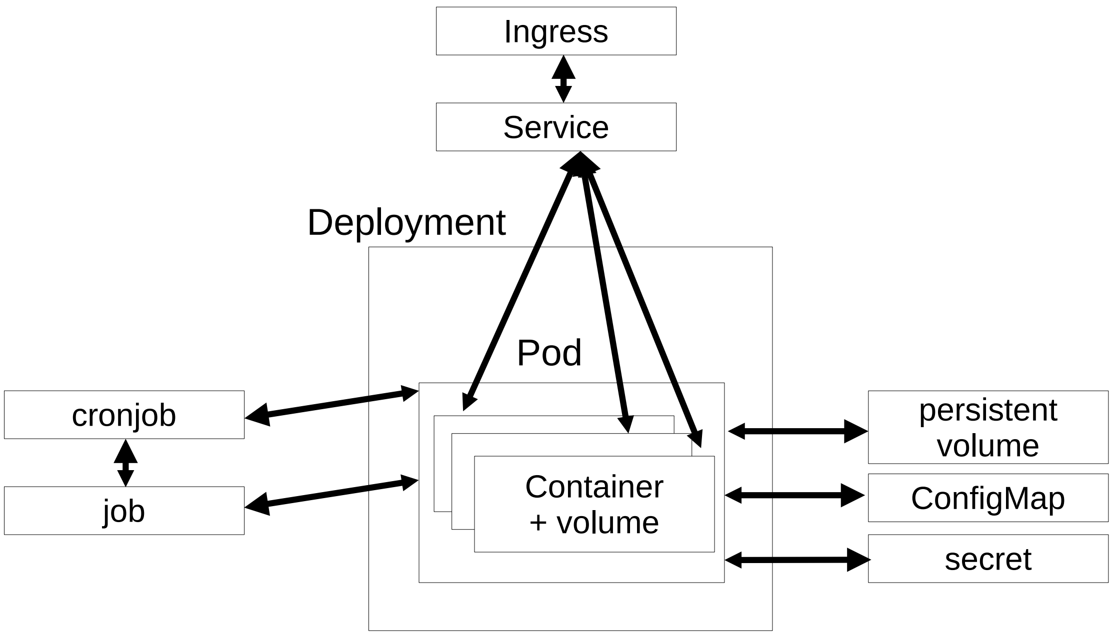
Kubernetes Components

- **API Server (kube-apiserver)** - front end for the Kubernetes, exposes Kubernetes API
- **Controller Manager (kube-controller-manager)** - managing tasks such as node operations, replication and endpoint management
- **etcd** - consistent and highly available key-value store to store cluster data
- **kubelet** - agent that runs on each node in the cluster, ensures that containers are running in a Pod and handles node-level operations
- **kube-proxy** - network proxy that runs on each node in the cluster, maintains network rules on nodes and allows network communication to your Pods from network sessions inside or outside of cluster



Kubernetes resources

- **Pod:** one or more containers that run the application
- **Deployment:** used to run a scalable application
- **Volume:** represents storage, either as a part of the Pod definition, or as its own resource
- **Service:** a policy that provides access to the application Pods
- **Jobs:** tasks that run as a Pod in the Kubernetes cluster
- **CronJobs:** used to run Jobs on a specific shedule



Kubernetes application creation

GitOps Code Requirements:
- Application updates should happen through the same code as application creation
- Code mus be idempotent

- To meet these requirements in a Kubernetes environment, changes should be applied the declarative way
- The **kubectl apply** command is what should be used thoughout

Running Applications the Declarative Way
- An easy way to run an application in Kubernetes is **kubectl create deploy testapp --image=testimage --replicas=3** and then to view **kubectl get all**
- Although useful, this way doesn’t work well in a GitOps environment
- In GitOps the application is defined in a YAML manifest file, and Kubernetes uses an operator to pick up and apply changes in the YAML file automatically
- To generate the YAML manifest file, use **kubectl create deploy testapp --image=testimage --replicas=3 --dry-run=client -o yaml > testapp.yaml**
- Next, push the YAML file to the Git repository and have the Kubernetes operator use **kubectl apply -f testapp.yaml** to apply the code to the cluster

- After defining YAML files, kubectl create -f myapp.yaml can be used to create the application. This command only works if the application doesn’t yet exist
- **kubectl apply -f testapp.yaml** is recommended in GitOps
  - if the application doesn’t yet exist, it will be updated
  - if the applicaiton already exists, modification will be applied
  - to preview modificaiton sthat wil be applied, use **kubectl diff -f testapp.yaml**
  - each time the application is updated, kubectl apply stores the configuration in the last-applied-configuration annotation, which allows **kubectl diff** to see any change in the manifest file

Kubernetes application creation

Example: Running Application in Declarative way
- **kubectl create deploy testserver --image=nginx**

  ya, it is not declarative, but it just to show, that if you run same command again...
- **kubectl create deploy testserver --image=nginx**

  ... appears message that this deployment already exists
- **kubectl delete deploy testserver**

Declarative way is
- **kubectl create deploy testserver --image=nginx --dry-run=client -o yaml > testserver.yaml**
- **kubectl apply -f testserver.yaml**
- **kubectl get deploy testserver -o yaml | less**

  Let’s run same command again ...
- **kubectl apply -f testserver.yaml**

  ... and now message tells us that deployment configured

Example: Running Application
- **source <(kubectl completion bash)**
- **kubectl create deploy webserver --image=nginx --replicas=3 --dry-run=client -o yaml > webserver.yaml**
- **cat webserver.yaml**
- **kubectl apply -f webserver.yaml**

  To see whats happen
- **kubectl get all**

  Let’s see pod
- **kubectl describe pod webserver[Tab]**
- edit webserver.yaml and add into **metadata: annotations: environment: qa**
- **kubectl diff -f webserver.yaml**
- **kubectl apply -f webserver.yaml**

Kubernetes accessing Applications

After running an application, it doesn’t automatically become accessible. To access it, different solutions can be used:

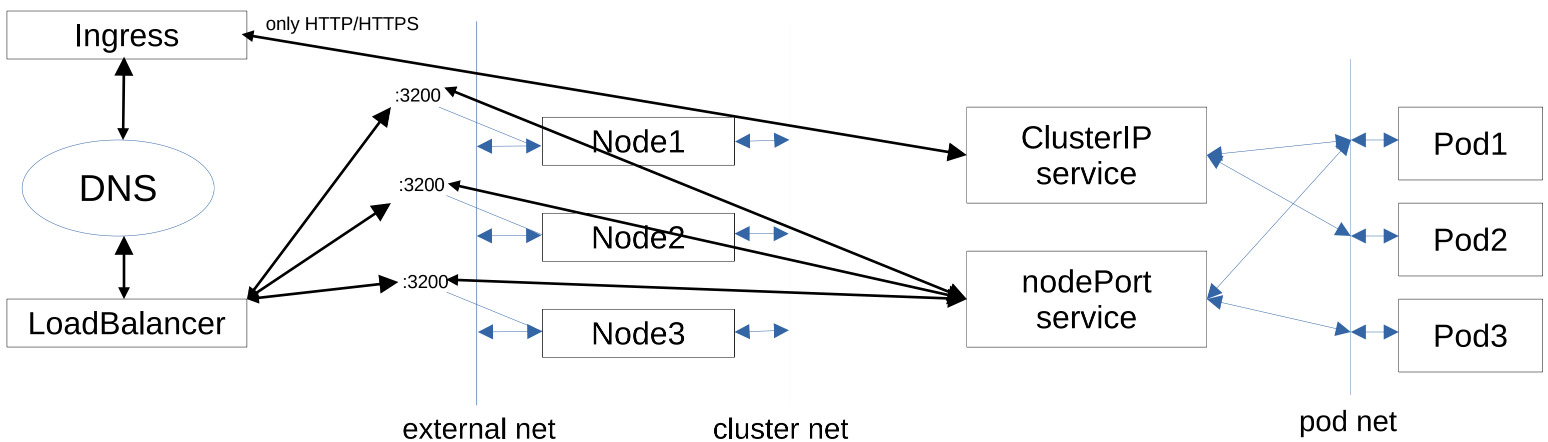
- **kubectl port-forward** is for testing puposes only, and exposes a port on the host that runs the **kubctl** client
- Services are Kubernetes objects that provide load balancing to the different Pod instances
- Ingress is an additional Kubernetes object that provides access to HTTP and HTTPS resources

Example: using kubectl port-forward

- Let’s see deployment running with usage of selector-label...
- **kubectl get all --selector app=webserver**
- Then try to check how port-forward works...
- **kubectl port-forward webserver-[Tab] 8080:80**
- **curl localhost:8080**

Kubernetes networking

- **Pod:** one or more containers that run the application
- **Deployment:** used to run a scalable application
- **Volume:** represents storage, either as a part of the Pod definition, or as its own resource
- **Service:** a policy that provides access to the application Pods
- **Jobs:** tasks that run as a Pod in the Kubernetes cluster
- **CronJobs:** used to run Jobs on a specific shedule



Service Types. Service Expose

- **ClusterIP**: the default type; provides internal access only
- **NodePort**: allocates a specific node port which needs to be opened on the firewall
- **LoadBalancer**: currently only implemented in public cloud
- **ExternalName**: a relatively new object that works on DNS names; redirection is happening at a DNS level
- **Service without selector**: use for direct connections based on IP/port, without an endpoint. Useful for connections to database, or between namespaces

  Example: Using Services. Service Expose
- **kubectl create deploy svcnginx --image=nginx --replicas=2**
- Let’s verify
  kubectl get all -o wide --selector app=svcnginx
- To expose deployment
  kubectl expose deploy svcnginx --port=80
- Let’s verify again
  kubectl get all -o wide --selector app=svcnginx
- or to observe services info
  kubectl get svc
- **curl <serviceIP>** #check will fail
- because our client workstation is external for k8s
- but if we will try from the minikube node
  minikube ssh
- **curl <serviceIP>; exit**
- **kubectl edit svc svcnginx** #change type: ClusterIP to type: NodePort
- to observe services info
  kubectl get svc
- **curl $(minikube ip):<nodeport>** #on windows you can try initially get ip by “minikube ip”

Service DNS registration

- Services automatically register with the Kubernetes interlan DNS server
- While obtaining networking information, all Pods use this internal DNS server
- As a result, all Pods can access all services (if no further network restrictions are applied)

  Example: Service Auto Registration
- **kubectl run -it busybox --image=busybox -- sh**
- To see DNS resolv
  cat /etc/resolv.conf
- **ping svcnginx**
- **exit**
- To see coredns service
  kubectl get pods -n kube-system -o wide
- To see kube-dns service
  kubectl get svc -n kube-system

Ingress

- Ingress exposes HTTP and HTTPS routes to services running inside cluster:
  - Services get externally reachable URLs
  - Ingress can load balance
  - Ingress can take care of TLS/SSL termination
- Ingress needs an Ingress controller to do the work
- Ingress only exposes HTTP/HTTPS, other service types are exposed using the NodePort or LoadBalancer Service type
- As an alternative to using Ingress, generic load balancers can be used to provide access to applications

Ingress controllers:

- Different Ingress controllers are provided by the Kubernetes ecosystem
- The Ingress controller consists of an API resource, as well as an Ingress Pod that is started in the cluster
- Minikube provides easy Ingress access using a Minikube addon: **minikube addon enable ingress**

Example: Ingress

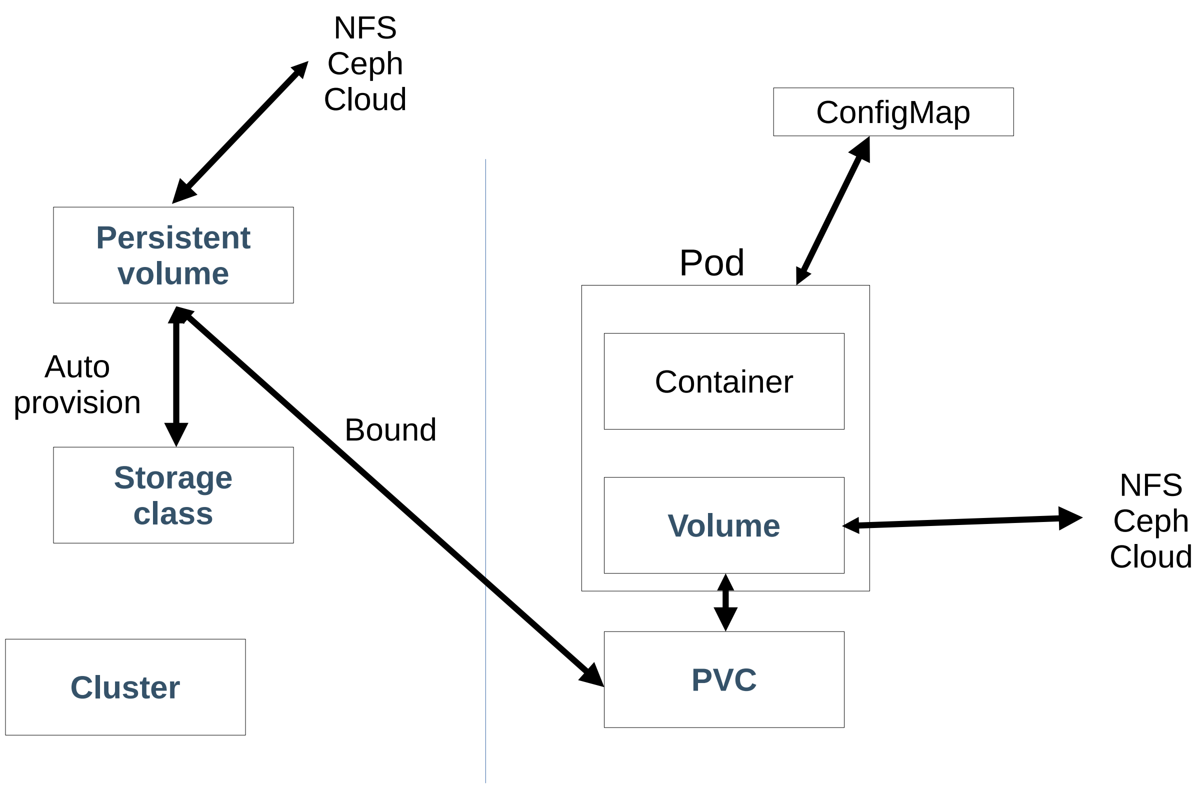
- **minikube addons list** # ingress addon is disabled by default
- **minikube addons enable ingress**
- **kubectl get ns** # to view new namespace ingress-nginx created
- **kubectl get pods -n ingress-nginx** #or **kubectl get all -n ingress-nginx**
- **kubectl describe pods -n ingress-nginx ingress-nginx-controller-xxx** #to observe controller pod
- **kubectl create ingress svcnginx-ingress --rule="/=svcnginx:80"** #specify to forward traffic to service svcnginx:80
- **sudo vim /etc/hosts** # or use hosts file in Windows
  - **$(minikube ip) svcnginx.info**
- **kubectl get ingress** #wait untill it shows an IP address
- **curl svcnginx.info**

Port-forwarding

- Services and Ingress enable access to applications
- Port-forwarding can be used to analyze or troubleshoot applications
- Port-forwarding by default is to a port at the loopback address of the **kubectl** client
- Use --address=... to expose on the NIC IP address
- **kubectl port-forward svc/svcnginx 8888:80** to forward to port 80 of svc/svcnginx.
  Use **curl localhost:8888** to check
- If you want solution accesible from outside - add ip-address:
  kubectl port-forward svc/svcnginx --address=192.168.29.110 8888:80

Decoupling Storage from Applications

- In Kubernetes, containers that are orchestrated and replicated should remain independent of specific hosts.
- To achieve this, it is essential to use network-based or cloud-based storage.
- This storage can be delivered in two main ways:
  - Pod volumes offer storage from within the Pods. To observe natively supported storage types use **kubectl explain pod.spec.volumes | less**
  - PersistentVolumes (PV) provide an external solution that entirely decouples storage from Pods.



Example: Pod Volume based on emptyDir

- It creates Pod with 2 containers based on centos image. Creates volumes of emptyDir type
- **vim myvolumes.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: myvol
spec:
  containers:
  - name: centos1
    image: centos:7
    command:
      - mountPath: /centos1
        name: test
  - name: centos2
    image: centos:7
    command:
      - sleep
      - “3600”
    volumeMounts:
      - mountPath: /centos2
        name: test
  volumes:
    - name: test
      emptyDir: {}
```

- **kubectl create -f myvolumes.yaml**
- **kubectl get pods**
- **kubectl describe pod myvol**
- **kubectl exec -it myvol -c centos1 -- touch /centos1/testfile**
- And what we will see if run command ...
  kubectl exec -it myvol -c centos2 -- ls /centos2
  we can see that file exists

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

Creating Persistent Volumes

PersistentVolume is a Kubernetes object that specifies the method to connect to external storage, utilizing various spec attributes:

- **capacity:** the amount of storage available
- **accessMode:** the access mode to be used
- **storageClassName:** (optional) the method to bind to a specific storage class
- **persistentVolumeReclaimPolicy:** the action to take when a corresponding PersistentVolumeClaim is deleted
- **type:** the specific storage type to use (e.g., NFS, azureDisk, gcePersistentDisk)

Example: persistent volume

- vim pv.yaml

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv-volume
  labels:
    type: local
spec:
  capacity:
    storage: 2Gi
  accessModes:
  - ReadWriteOnce
  hostPath:
     path: “/mydata”
```

- **kubectl create -f pv.yaml**
- **kubectl describe pv pv-volume**
- **minikube ssh**
- **ls /**
- **exit**
- it does not shows up because we should make something additional to start using this Persistent Volume

Creating PersistentVolumeClaim

- The PersistentVolumeClaim is a Kubernetes API object with a spec that outlines the required storage properties:
  - **accessModes**: the type of access needed
  - **resources**: the amount of storage required
  - **storageClassName**: the required storage class type
- Based on these properties, the PVC will bind to a specific PV for storage

  Example: PersistentVolumeClaim
  vim pvc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- **kubectl create -f pvc.yaml**
- **kubectl get pvc**
- we see that pvc is bound, but not to pv-volume, because it offers
- **kubectl get pvc,pv**
- **kubectl describe pv pvc-<numbers>**
- **kubectl get storageclass**
- if you don’t like default storage class you can disable it in minikube addons
- **minikube addons list**

Creating PersistentVolumeClaim

Example: setting up Pods to use Persistent Volume

```
vim pv-pod.yaml
kind: Pod
apiVersion: v1
metadata:
  name: pv-pod
spec:
  volumes:
    - name: pv-storage
      persistentVolumeClaim:
        claimName: pv-claim
  containers:
    - name: pv-container
      image: nginx
    ports:
      - containerPort: 80
        name: “http-server”
    volumeMounts:
       - mountPath: “/usr/share/nginx/html”
         name: pv-storage
```

- **kubectl create -f pv-pod.yaml**
- **kubectl describe pod pv-pod**
- **kubectl exec -it pv-pod -- touch /usr/share/nginx/html/hello.txt**
- # we are writing to persistent volume
- **minikube ssh**
- **ls -l /tmp/hostpath-provisioner/default/pv-claim/hello.txt**

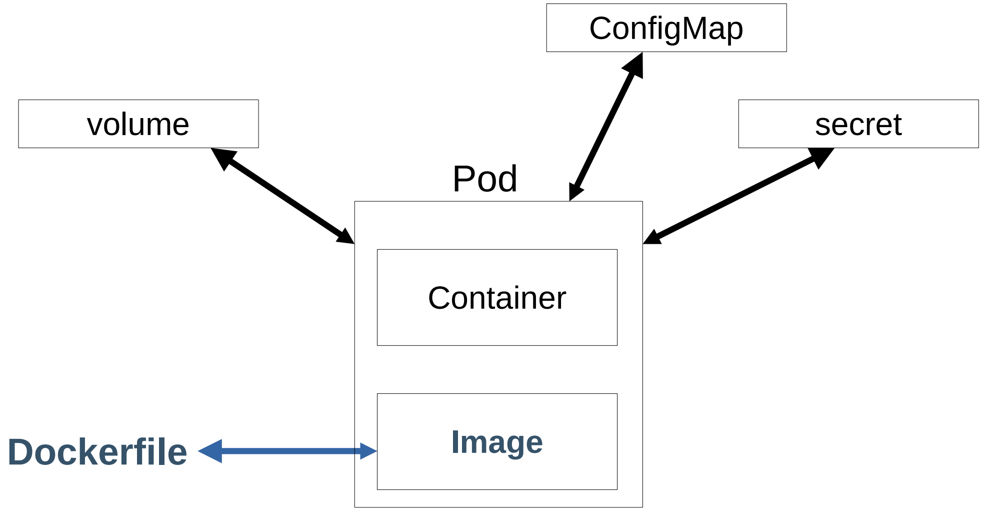
Example: configure Pod to use storage that is still exists after Pods lifetime

- **vim myvolumes.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: myvol
spec:
  containers:
  - name: centos1
    image: centos:7
    command:
      - sleep
      - "3600"
    volumeMounts:
      - mountPath: /centos1
        name: test
  - name: centos2
    image: centos:7
    command:
      - sleep
      - "3600"
    volumeMounts:
      - mountPath: /centos2
        name: test
  volumes:
    - name: test
      hostPath:
        path: /myfiles
```

- **kubectl create -f myvolumes.yaml**
- pod creation takes some time...
  kubectl describe pod myvol
- **kubectl exec -it myvol -c centos1 -- touch /centos1/hostPath.txt**
- **minikube ssh**
- **cd /myfiles**
- **ls**
  we can see hostPath.txt

Decoupling Configuration Files and Variables from Applications



Instruments to decouple infromation from the Application, running in Pod

ConfigMaps. Variables

- ConfigMaps can be used to store variables and configuration in the cloud
- The maximal size is 1MB, if configuration is bigger, it should be provided using volumes
- ConfigMaps can be used in two ways
  - To provide variables
  - To provide configuration files
- The ConfigMap will be addressed from a Pod according to how it is used
  - ConfigMaps containing variables are accessed usin envFrom
  - ConfigMap containing configuration files are mounted
- Secrets are base64 encoded ConfigMaps

Providing Variables with ConfigMaps

- While crating a ConfigMap with **kubectl create cm**, variables can be provided in two ways
  - Using **--from-env-file: kubectl create cm --from-env-file=dbvars**
  - Using **--from-literal: kubectl create cm --from-literal=MYSQL_USER=anna**
- Notice that it's possible to use multiple **--from-literal**, you cannot use multiple **--from-env-file**
- After creating the ConfigMap, use **kubectl set env --from=configmap/mycm deploy/myapp** to use the ConfigMap in your Deployment

Example: Providing Variables with ConfigMaps

- **vim varsfile**
  MYSQL_ROOT_PASSWORD=password
  MYSQL_USER=anna
- **kubectl create cm mydbvars --from-env-file=varsfile**
- **kubectl describe cm mydbvars**
- **kubectl create deploy mydb --image=mariadb --replicas=3**
- **kubectl get all --selector app=mydb**
- **kubectl describe pod mydb-6784929872**
- **kubectl logs mydb-6784929872**
- **kubectl set env deploy mydb --from=configmap/mydbvars**
- **kubectl get all --selector app=mydb**
- **kubectl get deploy mydb -o yaml**

ConfigMaps. Configuration Files

- Configuration files are typically used to provide site-specific information to applications
- To store configuration files in the cloud, ConfigMap can be used
- Use **kubectl create cm myconf --from=file=/my/file.conf**
- If a ConfigMap is created from a directory, all files in that directory are included in the ConfigMap
- To use the configuration file in an application, the ConfigMap must be mounted in the application
- There is no easy, imperative way to mount ConfigMaps in applications

Mounting a ConfigMap in an Application

- Note: Generate the base YAML code, and add the ConfigMap mount to it later
- In the application manifest, define a volume using the ConfigMap type
- Mount this volume on a specific directory
- The configuration file will appear inside that directory

Example: Using a ConfigMap with a Configuration File

- **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 deployments.apps myweb**

  spec.template.spec
  volumes:
  - name: cmvol
  configMap:
      name: myindex
      -----
  spec.template.spec.containers
  volumeMounts:
  - mountPath: /usr/share/nginx/html
  name: cmvol
- **kubectl describe pd myweb-239857**
- **kubectl exec -it myweb-239857 00 cat /usr/share/nginx/html/index.html**

Secrets

- Secrets facilitate the storage of sensitive data like passwords, authentication tokens, and SSH keys.
- Utilizing Secrets removes the necessity of placing this data directly in a Pod, thus minimizing the risk of unintentional exposure.
- While some Secrets are automatically generated by the system, users also have the capability to define their own Secrets.
- System-generated Secrets are crucial for enabling Kubernetes resources to connect with other resources within the cluster.
- It is important to note that Secrets are not encrypted; they are merely base64 encoded.

- Secrets are utilized to keep sensitive data separate from the Pods that require it.
- The base64 encoded information is stored within Etcd.
- Accessing Etcd requires Role Based Access Control (RBAC) permissions.
- For enhanced security, Etcd can be encrypted if necessary.
- When using Secrets for configuration files, it is advisable to use the defaultMode parameter during mounting: **defaultMode: 0400**.

  Three types of Secrets are available:
  - **docker-registry**: Used to store credentials needed to connect to a container registry.
  - **TLS**: Used to store TLS key material.
  - **generic**: Creates a secret from a local file, directory, or literal value.
- When defining the Secret, the type must be specified: **kubectl create secret generic ...**

- Before using it in an application, the Secret must be created the right way
  - 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=secret123**
  - 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**
  - 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 application when the Secret is updated

Example: Secret to Provide Passwords

- **kubectl create secret generic dbpw --from-literal=ROOT_PASSWORD=password**
- **kubectl describe secret dbpw**
- **kubectl get secret dbpw -o yaml**
  but if you run **echo <encoded secret> | base64 -d** you will see encoded secret
- **kubectl create deploy mynewdb --image=mariadb**
- if every application has it own prefix usage is more convenient
- **kubectl set env deploy mynewdb --from=secret/dbpw --prefix=MYSQL_**
- we can see running database by
- **kubectl get all --selector app=mynewdb**

Docker-registry Secrets

- Some container registries are only accesible for authenticated users
- To fetch images from such registries, the docker-registy Secret can be used
- After creating the docker-registry Secret type, use imagePullSecrets in the Pod specification to use it

  Example: using a Secret for Registry Credentials
- **kubectl create secret docker-registry dockercreds --docker-server=hub.docker.com --docker-username=myusername --docker-password=password --docker-email=myusername@user.com**
- **kubectl get secret dockercreds -o yaml**
- **kubectl run secretpod --image=nginx --dry-run=client -o yaml > secretpod.yaml**
- Add the following:
- vim secretpod.yaml

```
spec:
  containers:
  - ...
  imagePullSecrets:
  - name:dockercreds
```

- **kubectl create -f secretpod.yaml**
- **kubectl get pods**

Using ConfigMaps

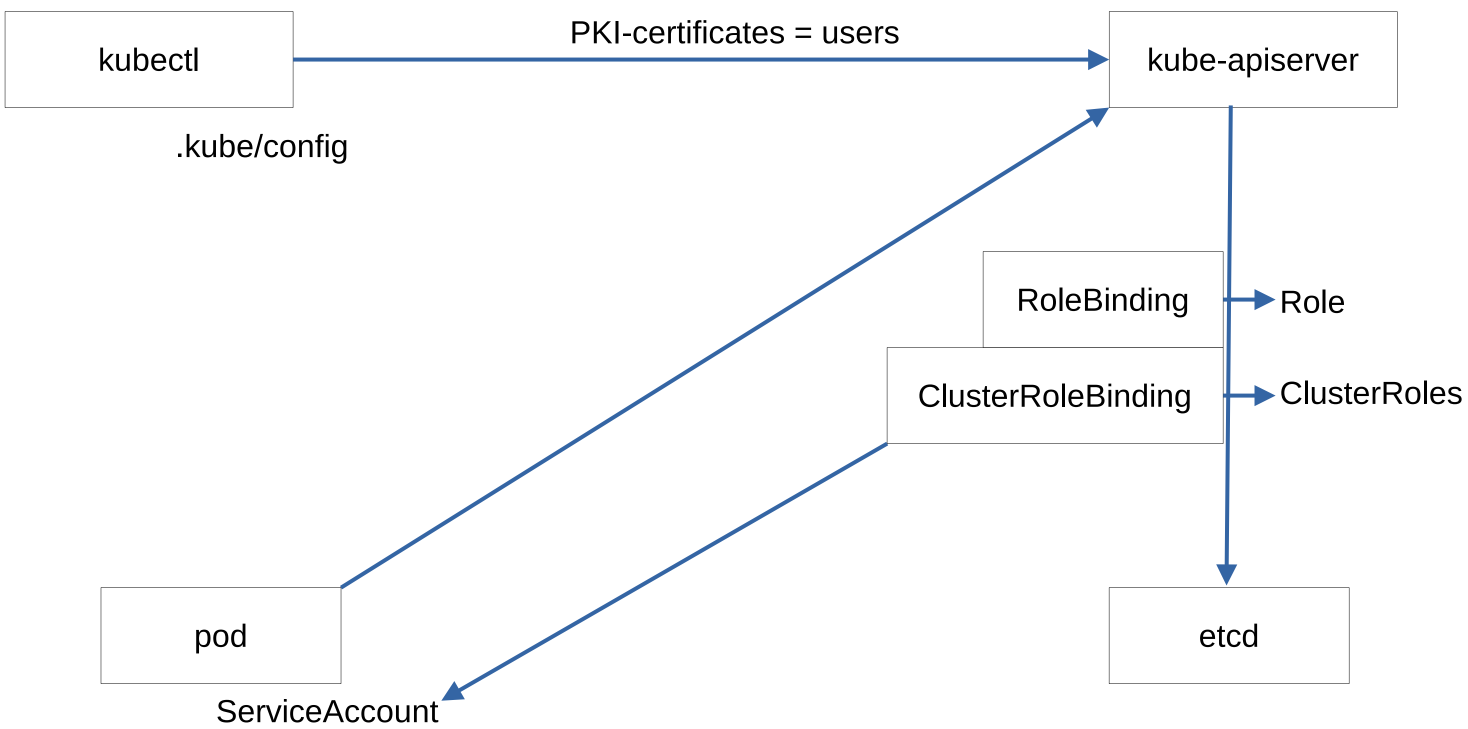
- Create a sample index.html file
- Store this file in a ConfigMap
- Run a deployment that uses nginx that is using the index.html file from the ConfigMap

- **vim index.html**
- hello, world!
- **kubectl create cm mynewindex --from-file=index.html**
- **kubectl get cm mynewindex -o yaml**
- Search in documentation “add configMap data to volume” https://raw.githubusercontent.com/kubernetes/website/main/content/en/examples/pods/pod-configmap-volume.yaml
- **vim lab.yaml**

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: registry.k8s.io/busybox
      command: [ "/bin/sh", "-c", "ls /etc/config/" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config
  volumes:
    - name: config-volume
      configMap:
        # Provide the name of the ConfigMap containing the files you want
        # to add to the container
        name: mynewindex
  restartPolicy: Never
```

- **kubectl create -f lab.yaml**
- **kubectl logs dapi-test-pod**

Kubernetes API server



Instruments to decouple infromation from the Application, running in Pod

Security Context

- A SecurityContext defines privilege and access control settings for Pods or containers and can include the following:
- UID- and GID-based Discretionary Access Control
- SELinux security labels
- Linux Capabilities
- Seccomp
- AppArmor
- The runAsNonRoot setting
- The AllowPrivilegeEscalation setting