CKA notes mumshad

ETCD

The time it waits for a pod to come back online is known as the pod-eviction-timeout and is set on the controller manager with a default value of five minutes. So whenever a node goes offline, the master node waits for up to five minutes before considering the node dead.

```
kube-controller-manager --pod-eviction-timeout=5m0s  ...
```

`kubectl drain node01 --ignore-daemonsets`

Why did the drain command fail on `node01`? It worked the first time!

Run: `kubectl get pods -o wide` and you will see that there is a single pod scheduled on node01 which is not part of a replicaset.

The drain command will not work in this case. To forcefully drain the node we now have to use the `--force` f

A forceful drain of the node will delete any pod that is not part of a replicaset.

---

usecase of cordon:

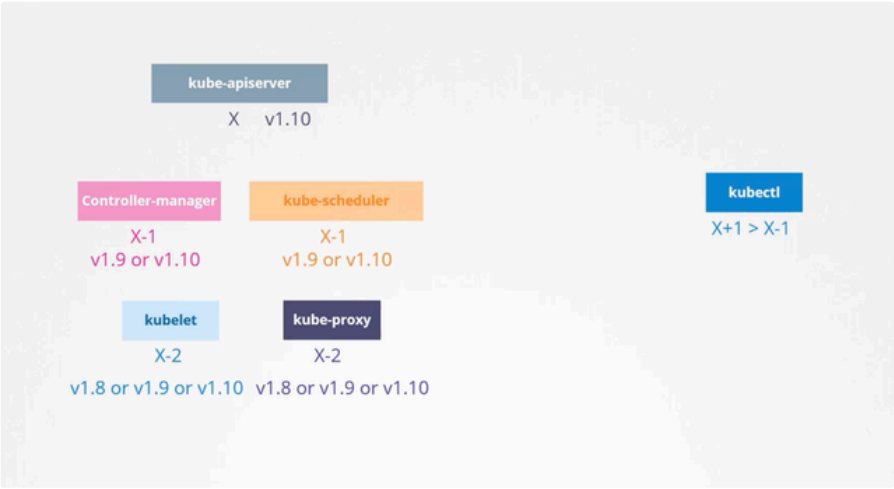
`hr-app` is a critical app and we do not want it to be removed and we do not want to schedule any more pods on `node01`.

Mark `node01` as `unschedulable` so that no new pods are scheduled on this node.

Make sure that `hr-app` is not affected.

Do not drain `node01`, instead use the `kubectl cordon node01` command. This will ensure that no new pods are scheduled on this node and the existing pods will not be affected by this operation.

You can find all the releases in the releases page of the Kubernetes GitHub repository. Download the Kubernetes.tar.gz file and extract it to find executables for all the Kubernetes components. The downloaded package, when extracted, has all the control plane components in it, all of them of the same version. Remember that there are other components `(The etcd cluster and CoreDNS servershave their own versions`, as they are separate projects.)within the control plane that do not have the same version numbers.



So if kube-apiserver was at 1.10, the controller-manager and scheduler could be at 1.10 or 1.9, and the kubelet and kube-proxy could be at 1.8.

None of them could be at a version higher than the kube-apiserver, like 1.11. Now, this is not the case with kubectl.

The kubectl utility could be at 1.11,a version higher than the API server, 1.10, the same version as the API server, or at 1.9, a version lower than the API server.

- In a `Kube admin setup`, the Kube controller manager is deployed as a POD in the `Kube-system` namespace on the master node.
- Options can be viewed within the POD definition file located at `/etc/kubernetes/manifests/` folder.
- In a `Non-Kube admin setup`, options can be inspected by viewing the `Kube Controller Manager's service` located at the `services` directory.

`ETCDCTL` is the CLI tool used to interact with ETCD.

- In etcd.service >> there is a line `advertise_client_url` that contains info:
  Default port is `2379` and `server ip` it will listen to. This is url need to configure in `kube_apiserver` to reach `etcd`.
- `--initial_cluster_controller` for HA env in etcd.service >> point to remember

- `ETCDCTL` can interact with `ETCD Server` using 2 API versions - `Version 2` and `Version 3`.
  By default its set to use `Version 2`.
  Each version has different sets of commands.
- For example ETCDCTL version 2 supports the following commands:
  - `etcdctl backupetcdctl cluster-healthetcdctl mketcdctl mkdiretcdctl set`
- Whereas the commands are different in version 3
  - `etcdctl snapshot save etcdctl endpoint healthetcdctl getetcdctl put`
- To set the right version of API set the environment variable ETCDCTL_API command
  - `export ETCDCTL_API=3`
- When `API version is not set`, it is assumed to be set to `version 2`. And `version 3` **commands** listed above **don't work**.
- When API version is set to `version 3`, `version 2` commands listed above `don't work`.
- Apart from that, you must also specify path to certificate files so that `ETCDCTL` can authenticate to the `ETCD API Server`. The certificate files are available in the `etcd-master` at the following path.
  - `--cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key`
- So for the commands I showed in the previous video to work you must specify the ETCDCTL API version and path to certificate files. Below is the final form:
  - `kubectl exec etcd-master -n kube-system -- sh -c "ETCDCTL_API=3 etcdctl get / --prefix --keys-only --limit=10 --cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key"`
- This is for kubeadm: inorder to query the key-values in etcd server `kubectl exec etcd -master -n kube-system etcdctl get / --prefix -key-only`
- By default it will run as pod in `kubesystem namspace` when you install `kubeadm`.
  `ETCDCTL_API=3`
  first we need check etcdctl is there or not , if not intsall it using
  `apt-get install etcdctl`
  Then create a secret, by default secret will be encoded, however if you check in etcd , you can still see the secret value using below cmd:

For example:

```
ETCDCTL_API=3 etcdctl \
   --cacert=/etc/kubernetes/pki/etcd/ca.crt   \
   --cert=/etc/kubernetes/pki/etcd/server.crt \
   --key=/etc/kubernetes/pki/etcd/server.key  \
   get /registry/secrets/default/secret1 (name of secret) | hexdump -C
```

so first you need identify if the secrest stored in etcd are encryped or not you can verify it by using
`ps -aux | grep -i kube-apiserver | grep --encryption-provider-config`

if no o/p then its not encrypted in etcd. that's how we determine whether encryption at rest is already enabled, hence steps to enable

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
      - configmaps
      - pandas.awesome.bears.example # a custom resource API
    providers:
      # This configuration does not provide data confidentiality. The first
      # configured provider is specifying the "identity" mechanism, which
      # stores resources as plain text.
      #
      - identity: {} # plain text, in other words NO encryption
      - aesgcm:
          keys:
            - name: key1
```

it references specifies the `identity` provider as the first encryption provider in the list, then you do not have at-rest encryption enabled (**the default** `identity` provider does not provide any confidentiality protection.)
note: it follows the order to encrypt or decrypt below providers

In order to encrypt data at rest, create a new encryption configuration file. The contents should be similar to:

```
---
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
  - resources:
      - secrets
      - configmaps
      - pandas.awesome.bears.example
    providers:
      - aescbc:
```

```
          keys:
            - name: key1
              # See the following text for more details about the secret value
              secret: <BASE 64 ENCODED SECRET>
      - identity: {} # this fallback allows reading unencrypted secrets;
                     # for example, during initial migration
```

Generate a 32-byte random key and base64 encode it. You can use this command:

```
head -c 32 /dev/urandom | base64
```

Use the new encryption configuration file

You will need to mount the new encryption config file to the `kube-apiserver` static pod. Here is an example on how to do that:

1. Save the new encryption config file to `/etc/kubernetes/enc/enc.yaml` on the control-plane node.
2. Edit the manifest for the `kube-apiserver` static pod: `/etc/kubernetes/manifests/kube-apiserver.yaml` so that it is similar to:

```
---
#
# This is a fragment of a manifest for a static Pod.
# Check whether this is correct for your cluster and for your API server.
#
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 10.20.30.40:443
  creationTimestamp: null
  labels:
    app.kubernetes.io/component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    ...
    - --encryption-provider-config=/etc/kubernetes/enc/enc.yaml  # add this line
    volumeMounts:
    ...
    - name: enc                           # add this line
      mountPath: /etc/kubernetes/enc      # add this line
      readOnly: true                      # add this line
    ...
  volumes:
  ...
  - name: enc                             # add this line
    hostPath:                             # add this line
      path: /etc/kubernetes/enc           # add this line
      type: DirectoryOrCreate             # add this line
  ...
```

ETCD Backup

If you check out the pods running in the `kube-system` namespace in `cluster1`, you will notice that `etcd` is running as a pod:

```
student-node ~ ➜  kubectl config use-context cluster1
Switched to context "cluster1".
student-node ~ ➜  kubectl get pods -n kube-system | grep etcd
etcd-cluster1-controlplane                   1/1     Running   0             9m26s
student-node ~ ➜
```

This means that ETCD is set up as a `Stacked ETCD Topology` where the distributed data storage cluster provided by `etcd` is stacked on top of the cluster formed by the nodes managed by kubeadm that run control plane components.

_____

To restore the cluster from this backup at a later point in time, first, stop the Kube API server service, as the restore process will require you to restart the etcd cluster, and the Kube API server depends on it. Then, run the etcd controls snapshot restore command, with the path set to the path of the backup file, which is the snapshot.db file.

command to take etch backup:

`ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key --cacert=/etc/kubernetes/pki/etcd/ca.crt snapshot save /opt/snapshot-pre-boot.db`

Snapshot saved at /opt/snapshot-pre-boot.db

to restore cluster :

First Restore the snapshot:

```
root@controlplane:~# ETCDCTL_API=3 etcdctl  --data-dir /var/lib/etcd-from-backup \
snapshot restore /opt/snapshot-pre-boot.db
2022-03-25 09:19:27.175043 I | mvcc: restore compact to 2552
2022-03-25 09:19:27.266709 I | etcdserver/membership: added member 8e9e05c52164694d [http://localhost:2380] to cluster cdf818194e3a8c32
root@controlplane:~#
```

Note: In this case, we are restoring the snapshot to a different directory but in the same server where we took the backup **(the controlplane node)** As a result, the only required option for the restore command is the `--data-dir`.

Next, update the `/etc/kubernetes/manifests/etcd.yaml`:

We have now restored the etcd snapshot to a new path on the controlplane - `/var/lib/etcd-from-backup`, so, the only change to be made in the YAML file, is to change the hostPath for the volume called `etcd-data` from old directory (`/var/lib/etcd`) to the new directory (`/var/lib/etcd-from-backup`).

```
  volumes:
  - hostPath:
      path: /var/lib/etcd-from-backup
      type: DirectoryOrCreate
    name: etcd-data
```

With this change, `/var/lib/etcd` on the container points to `/var/lib/etcd-from-backup` on the `controlplane` (which is what we want).

When this file is updated, the `ETCD` pod is automatically re-created as this is a static pod placed under the `/etc/kubernetes/manifests` directory.

> Note 1: As the ETCD pod has changed it will automatically restart, and also `kube-controller-manager` and `kube-scheduler`. Wait 1-2 to mins for this pods to restart. You can run the command: `watch "crictl ps | grep etcd"` to see when the ETCD pod is restarted.

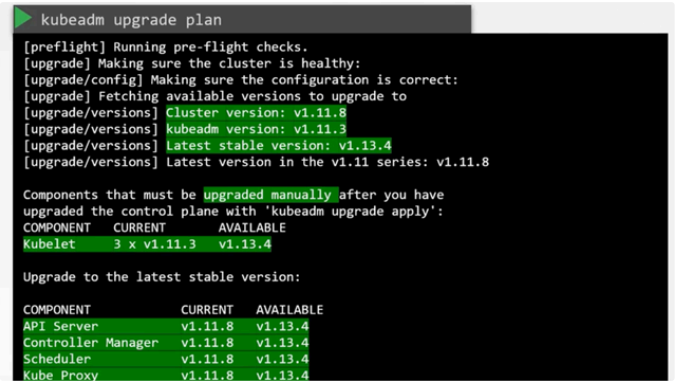
> Note 2: If the etcd pod is not getting `Ready 1/1`, then restart it by `kubectl delete pod -n kube-system etcd-controlplane` and wait 1 minute.

> Note 3: This is the simplest way to make sure that ETCD uses the restored data after the ETCD pod is recreated. You **don't** have to change anything else.

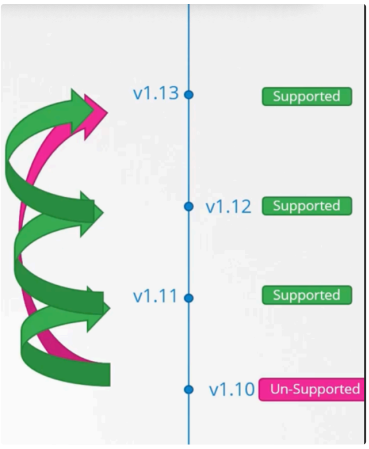
If you do change `--data-dir` to `/var/lib/etcd-from-backup` in the ETCD YAML file, make sure that the `volumeMounts` for `etcd-data` is updated as well, with the mountPath pointing to `/var/lib/etcd-from-backup` **(THIS COMPLETE STEP IS OPTIONAL AND NEED NOT BE DONE FOR COMPLETING THE RESTORE)**

Steps for master node upgrade:

1. `kubeadm upgrade plan`

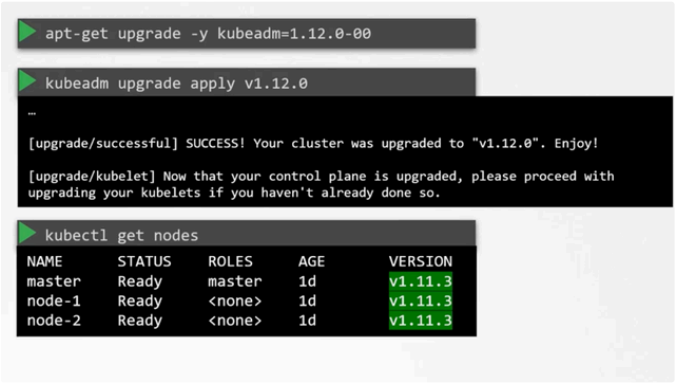


2.



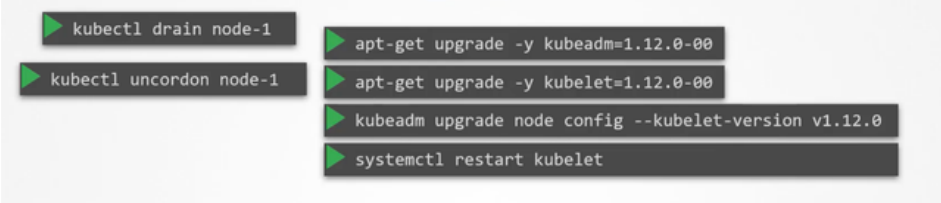
we can't directly go for 13 minor so we are upgrading to 12

3. upgrade the kubeadm tool: `apt-get upgrade -y kubeadm=1.12.0`
4. as per the plan: do apply `kubeadm upgrade apply v1.12.0`
5. if you run the `kubectl get nodes` command, you will still see the master node at 1.11. This is because in the output of this command,it is showing the versions of **kubelets** on each of these nodes registered with the API server and not the version of the API server itself.



6. now manually update the kubelet as per the plan: `apt-get upgrade -y kubelet=1.12.0`
6. restart the kubelet : `systemctl restart kubelet`
7. now it will show v1.12.0 in version section for master

steps to worker node upgrade:



`apt-get install kubelet=1.29.0-1.1`
`systemctl daemon-reload`
`systemctl restart kubelet`
`kubectl uncordon controlplane`

OS

`kubectl get pods -o wide` and you will see that there is a single pod scheduled on node01 which is not part of a replicaset.
The drain command will not work in this case. To forcefully drain the node we now have to use the `--force` flag.

Is it mandatory for all of these to have the same version? No. The components can be at different release versions. Since the **kube-apiserver is the primary component** in the control plane,and that is the component that all other components talk to,**none of the other components should ever be at a version higher than the kube-apiserver.**

The controller-manager and scheduler can be at one version lower. So if kube-apiserver was at x, **controller-managers and kube-schedulers** can be at **x-1**, and the **kubelet and kube-proxy** components can be at two versions lower, **x-2**

at any time, **Kubernetes supports only up to the recent three minor versions**. The recommended approach is to **upgrade one minor version at a time**, version 1.10 to 1.11, then 1.11 to 1.12, and then 1.12 to 1.13.

While the master is being upgraded, the control plane components, such as the API server, scheduler, and controller-managers, go down briefly. The master going down does not mean your worker nodes and applications on the cluster are impacted. All workloads hosted on the worker nodes continue to serve users as normal. **Since the master is down, all management functions are down**. You **cannot access the cluster using kubectl or other Kubernetes API.** You **cannot deploy new applications or delete or modify existing ones.**

kubeadm has an upgrade command that helps in upgrading clusters. With kubeadm, run the `kubeadm upgrade plan` command, and it will give you a lot of good informatio, the current cluster version, the kubeadm tool version,the latest stable version of Kubernetes. Then it lists all the control plane components and their versions and what version these can be upgraded to.

It also tells you that after **we upgrade the control plane components, you must manually upgrade the kubelet versions on each node.**

Remember, **kubeadm does not install or upgrade kubelets.** Finally, it gives you the command to upgrade the cluster. Also, note that **you must upgrade the kubeadm tool itself before you can upgrade the cluster.**

The kubeadm tool also follows the same software version as Kubernetes. So we're at 1.11, and we want to go to 1.13. But remember, we can only go one minor version at a time. So we first go to 1.12. First, upgrade the kubeadm tool itself to version 1.12. Then upgrade the cluster using the command from the upgrade plan output, `kubeadm upgrade apply`. It pulls the necessary images and upgrades the cluster components. Once complete, your control plane components are now at 1.12.

Upgrade cluster steps:

Make sure that the correct version of `kubeadm` is installed and then proceed to upgrade the `controlplane` node. Once this is done, upgrade the `kubelet` on the node.

1. first we need to drain the node-- `kubectl drain controlplane --ignore-daemonsets`
2. we need to update kubeadm tool itself cmd: `apt-get upgrade -y kubeadm=1.12.0.00`
3. then run the `kubeadm upgrade plan`, it will show versions to be updated
4. then apply- `kubeadm upgrade apply v1.12.0`
5. then if you run the command `kubectl get nodes`, still version is pointing to older version because it is the version of **kubelet** associated to master node not the version of apiserver.
6. then manually by running `apt-get upgrade -y kubelet=1.12.0-00` command we will upgarde the kubelet.
7. then restart the kubelet server(now if you run kubectl get nodes----it will update the version cloumn)
8. `uncordon the controlplane`

mumshad:
first we need to drain the node-- `kubectl drain controlplane --ignore-daemonsets`

To seamlessly transition from Kubernetes v1.28 to v1.29 and gain access to the packages specific to the desired Kubernetes minor version, follow these essential steps during the upgrade process. This ensures that your environment is appropriately configured and aligned with the features and improvements introduced in Kubernetes v1.29.

On the `controlplane` node:

Use any text editor you prefer to open the file that defines the Kubernetes apt repository.

```
vim /etc/apt/sources.list.d/kubernetes.list
```

Update the version in the URL to the next available minor release, i.e v1.29.

```
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /
```

After making changes, save the file and exit from your text editor. Proceed with the next instruction.

```
root@controlplane:~# apt update
root@controlplane:~# apt-cache madison kubeadm
```

Based on the version information displayed by `apt-cache madison`, it indicates that for Kubernetes version `1.29.0`, the available package version is `1.29.0-1.1`. Therefore, to install kubeadm for Kubernetes `v1.29.0`, use the following command:

```
root@controlplane:~# apt-get install kubeadm=1.29.0-1.1
```

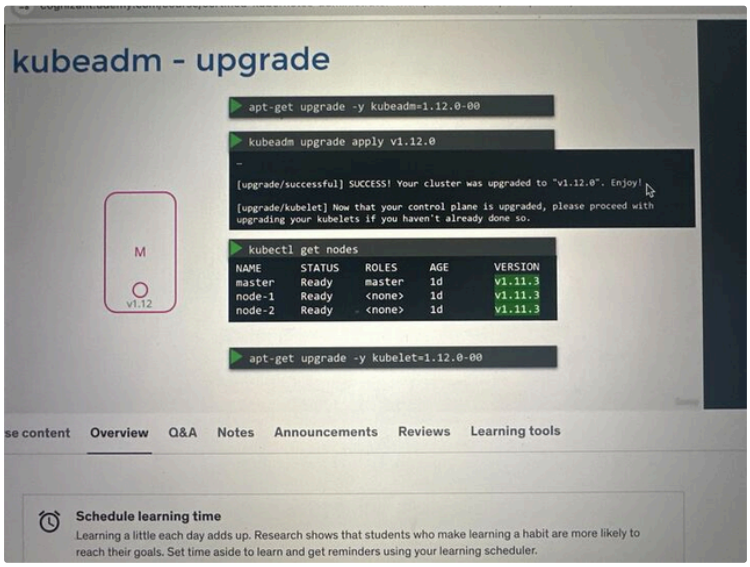
Run the following command to upgrade the Kubernetes cluster.

```
root@controlplane:~# kubeadm upgrade plan v1.29.0
root@controlplane:~# kubeadm upgrade apply v1.29.0
```

> Note that the above steps can take a few minutes to complete.

Now, upgrade the version and restart Kubelet. Also, mark the node (in this case, the "controlplane" node) as schedulable.

```
root@controlplane:~# apt-get install kubelet=1.29.0-1.1
root@controlplane:~# systemctl daemon-reload
root@controlplane:~# systemctl restart kubelet
root@controlplane:~# kubectl uncordon controlplane
```



Upgrade worker nodes:

1. first we need to drain the node-- `kubectl drain node1`
2. then upgrade the kubeadm `apt-get upgrade -y kubeadm=1.12.0.00`
3. then upgrade kubelet `apt-get upgrade -y kubelet=1.12.0-00`
4. then run the command `kubeadm upgrade node config --kubelet-version v1.12.0` --this will update the other node configuration to match kubelet version
5. restart the kubelet `systemctl restart kubelet`
6. `uncordon the node1`

> mumshad Node upgrade:
> in the `node01` node, run the following commands:

> If you are on the `controlplane` node, run `ssh node01` to log in to the `node01`.

Use any text editor you prefer to open the file that defines the Kubernetes apt repository.

```
vim /etc/apt/sources.list.d/kubernetes.list
```

Update the version in the URL to the next available minor release, i.e v1.29.

```
deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg] https://pkgs.k8s.io/core:/stable:/v1.29/deb/ /
```

After making changes, save the file and exit from your text editor. Proceed with the next instruction.

```
root@node01:~# apt update
root@node01:~# apt-cache madison kubeadm
```

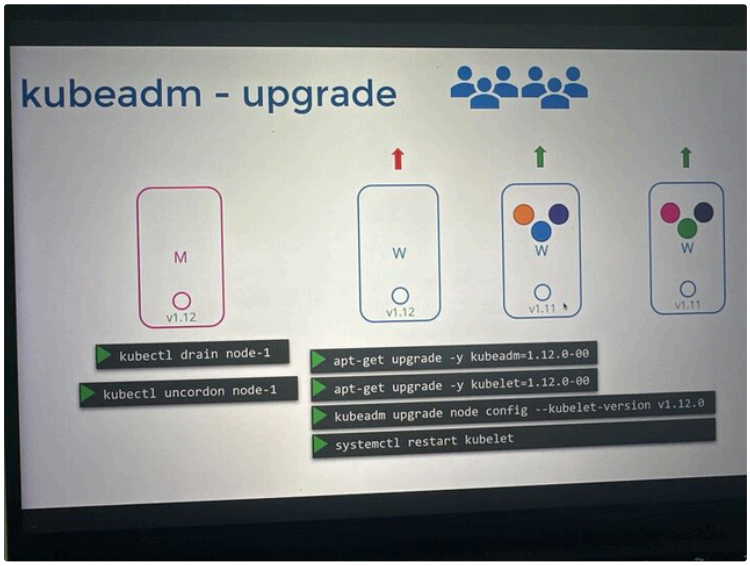
Based on the version information displayed by `apt-cache madison`, it indicates that for Kubernetes version `1.29.0`, the available package version is `1.29.0-1.1`. Therefore, to install kubeadm for Kubernetes `v1.29.0`, use the following command:

```
root@node01:~# apt-get install kubeadm=1.29.0-1.1
# Upgrade the node
root@node01:~# kubeadm upgrade node <<<<<<<<<<<<<<<command is different bcz while upgrading master there we have other master component so the cmd is
```

Now, upgrade the version and restart Kubelet.

```
root@node01:~# apt-get install kubelet=1.29.0-1.1
root@node01:~# systemctl daemon-reload
root@node01:~# systemctl restart kubelet
```

> Type `exit` or `logout` or enter `CTRL + d` to go back to the `controlplane` node.



How many nodes can host workloads in this cluster?
Inspect the applications and taints set on the nodes.
Check the taints on both controlplane and node01. If none exists, then both nodes can host workloads.
By running the `kubectl describe node` command, we can see that neither nodes have taints.

```
root@controlplane:~# kubectl describe nodes  controlplane | grep -i taint
Taints:             <none>
root@controlplane:~#
root@controlplane:~# kubectl describe nodes  node01 | grep -i taint
Taints:             <none>
root@controlplane:~#
```

Q: This means that both nodes have the ability to schedule workloads on them

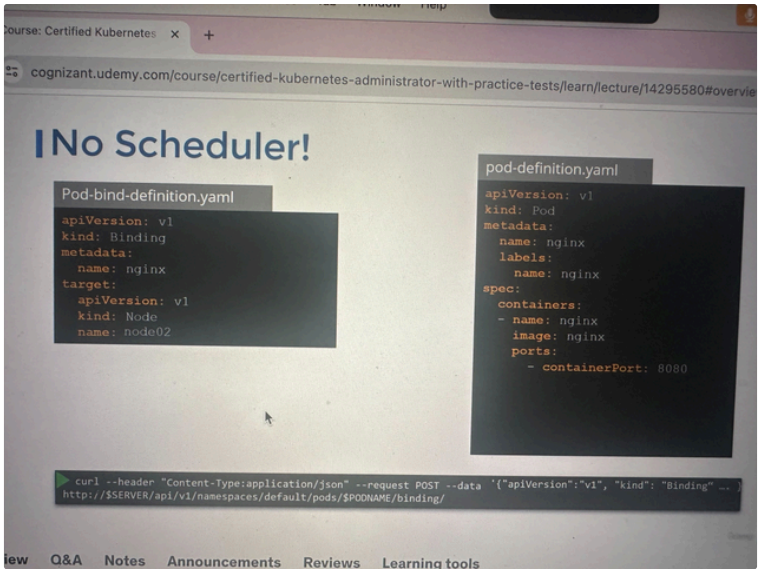
What is the latest version available for an upgrade with the current version of the kubeadm tool installed?
Use the kubeadm tool: v1.28.8

Upgrade the `controlplane` components to exact version `v1.29.0`

Scheduling

Manual scheduling: by configuring the `nodeName` in the spec section of the pod.

if pod is already deployed without nodeName section, by using **binding** kind we can manually deployed to specific node.



Run the command: `kubectl get pods --namespace kube-system` to see the status of `scheduler` pod. We have removed the scheduler from this Kubernetes cluster. As a result, as it stands, the pod will remain in a **pending state forever.**

How many objects are in the `prod` environment including PODs, ReplicaSets and any other objects?

- Run the command to get exact number of objects `kubectl get all --selector env=prod --no-headers | wc -l` **use --no-headers as** `get` `all` **has multiple headers and blank space**
- For **multiple selction** there should be **no space** Run the command `kubectl get all --selector env=prod,bu=finance,tier=frontend`
- **Taint and tolerations:** more like **node specific**
  toleration addition to pod: values will have inverted commas and tolerations is in array format

  Solution manifest file to create a pod called `bee` as follows:

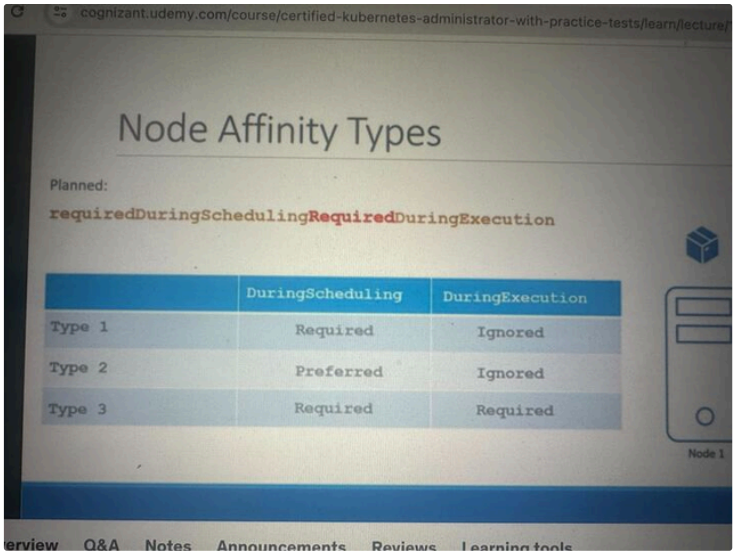
  ```
  ---
  apiVersion: v1
  kind: Pod
  metadata:
    name: bee
  spec:
    containers:
    - image: nginx
      name: bee
    tolerations:
    - key: spray
      value: mortein
      effect: NoSchedule
      operator: Equal
  ```

  then run the `kubectl create -f <FILE-NAME>.yaml` to create a pod.

- **To untaint the nod** Run the command: `kubectl taint nodes controlplane node-role.kubernetes.io/controlplane:NoSchedule-`

NodeSelector - In order use nodeselector , node should have a label

- At a time we can use only one lable
- we can specify nodeSelector under spec section:
  eg:
  nodeSelector:
  size: Large
  `kubectl label node node01 size=large`
  Node Affinity



-

note: If a key-value option is not available, the exist operator will be utilized under the affinity in the pod's spec section.
e.g. Create a new deployment named `red` with the `nginx` image and `2` replicas, and ensure it gets placed on the `controlplane` node only.

Use the label key - `node-role.kubernetes.io/control-plane` - which is already set on the `controlplane` node.

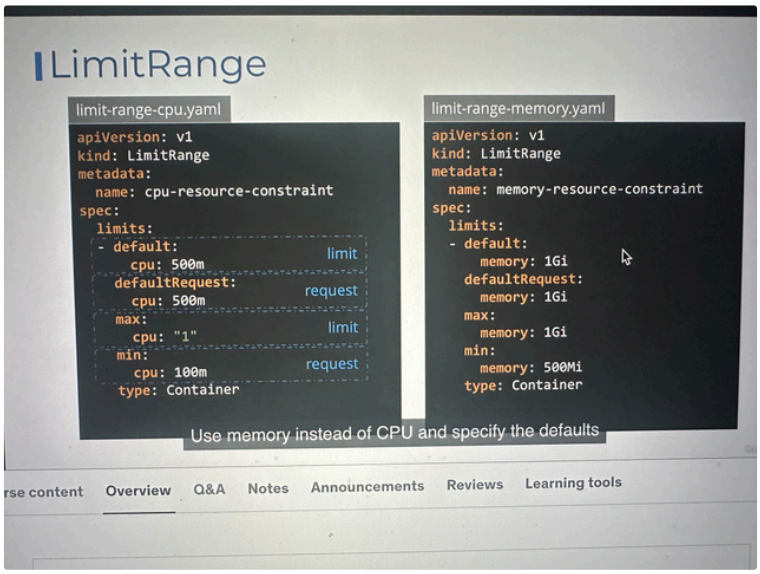
```
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: nginx
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: node-role.kubernetes.io/control-plane
            operator: Exists
```

Taints and toleration vs node affinity

- **Node Affinity**: It's like telling Kubernetes, "Hey, I prefer my pods to run on nodes that have certain characteristics." For example, you might want your pods to run on nodes with SSD storage or more memory. **however it doesn't mean other pods can't get schedule in that node**
- **Taints and Tolerations**: Imagine it as nodes in Kubernetes wearing certain "perfumes" or "fragrances" (taints). Pods can tolerate (like, not be bothered by) these smells or be repelled by them. So, if a node has a particular smell (taint), only pods that can handle that smell (toleration) will be scheduled there. **however it doesn't mean pods can't get schedule in other nodes**
- if we use node affinity on top of taints and tolerations, we can ensure a particular pod is schedules on particular node

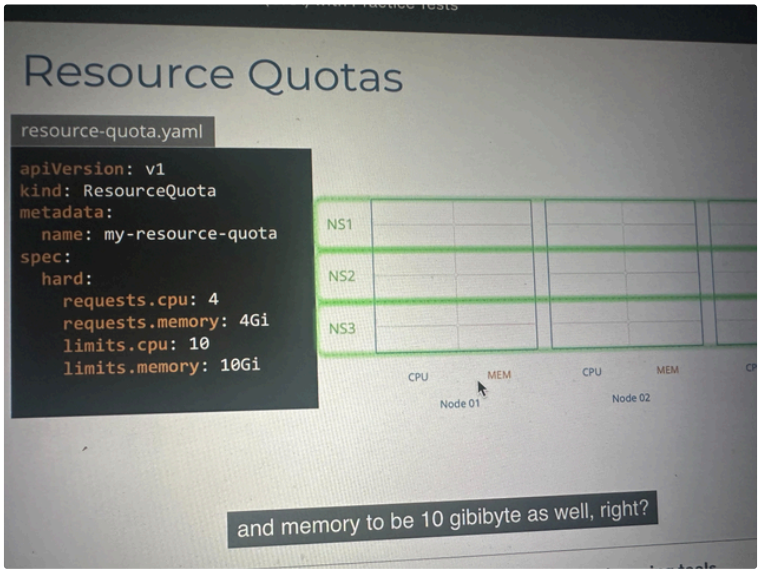
limits and request: It is a cluster-level object.

If you want by default all the new pods have



Resource quota is for all the pods combined at namespace level

So a resource quota is a namespace level object



Multiple-Scheduler:

So, if the pod is in a pending state, you can examine the logs using the 'kubectl describe' command. You may mostly observe that the scheduler isn't configured correctly. But how can you determine which scheduler picked it up when we have multiple schedulers?

This information can be viewed in the events by using the `'kubectl get events'` command with the `'-o wide'` option. This will display all events in the current namespace and allow you to identify scheduled events. As evident from this view, the source of these events is our custom scheduler.

For ensuring pod uses custom scheduler:

Insert `schedulerName` field under spec

Network-security

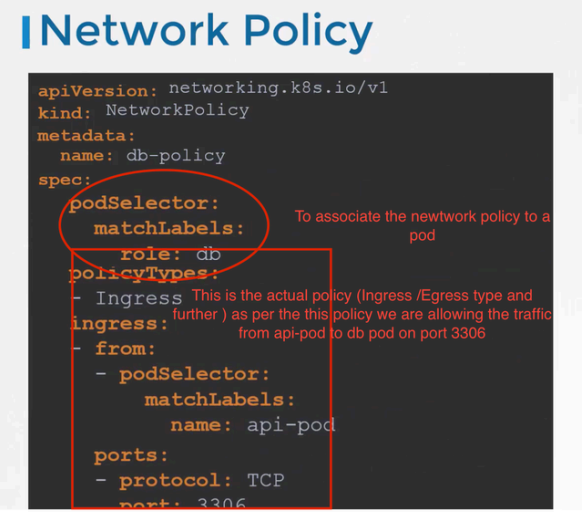
Kubernetes is configured by default with an all allow rule that allows traffic from any pod to any other pod or services within the cluster.

What if we do not want the front end web server to be able to communicate with the database server directly?

That is where you would implement a network policy to allow traffic to the DB server only from the API server. And network policy is another object in the Kubernetes namespace just like pods, replica setsor services, you link a network policy to one or more pods.

whatever there in meta data section of k8s object is only belongs to that k8s object(ex:labels)

if we want to call the same k8s object we will reference the those resource labels under a spec section of new k8s object(ex:selector)



Now note that ingress or egress isolation only comes into effect if you have ingress or egress in the `policy types`.

In this example, there's only ingress in the policy type which means only ingress traffic is isolated and all egress traffic is unaffected.
Meaning the pod is able to make any egress calls and they're not blocked. So for an egress or ingress isolation to take place,note that you have to add them under the policy types as seen here.

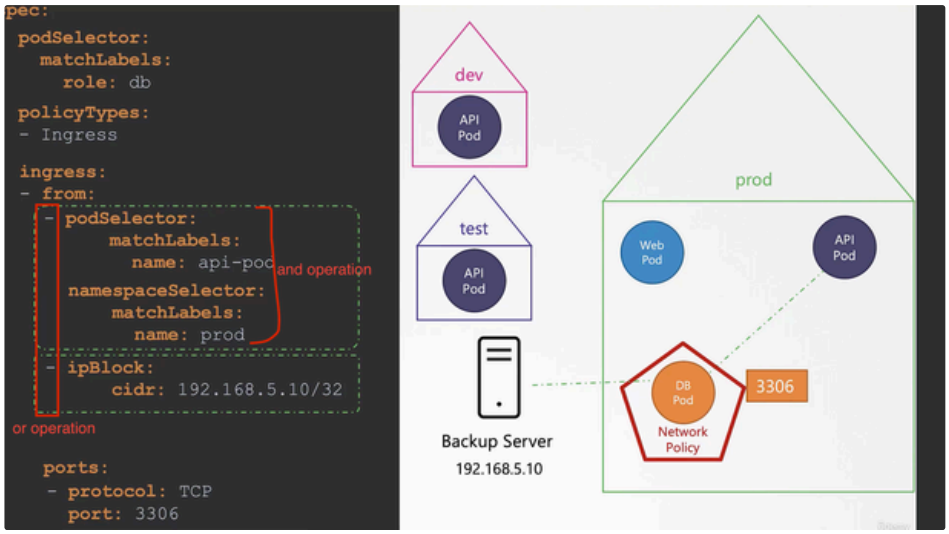
A few of them that are supported are Cube Router, Calico, Romana, and WaveNet. If you used Flannel as the networking solution, it does not support network policies as of this recording. Always referred to the Network Solutions Documentationto see support for network policies.Also, remember, even in a cluster configured with a solutionthat does not support network policies,you can still create the policies but they will just not be enforced.You will not get an error message saying the network solution does not support network policies.

If the network policy is applied to the database pod, any policies specified in that network policy will result in the pod being isolated, meaning it will not permit any traffic.

Ingress and Egress Policies: Ingress rules control incoming traffic to a pod, while egress rules control outgoing traffic from a pod.

- **From DB Pod's Perspective**:
  - Allowing API pod to connect: You only need an ingress rule to allow incoming traffic from the API pod to the database pod.
  - No separate rule needed for responses: Once incoming traffic is allowed, the response traffic (from DB to API) is automatically allowed.
- **No Reverse Connection**: If the DB pod needs to initiate a connection to the API pod, you would need a separate egress rule from the DB pod to the API pod.
- **Direction of Request**: Focus on where the request starts; the response is automatically allowed if the initial request is permitted.

So, to allow the API pod to query the DB pod, you only need an ingress rule on the DB pod.



Security-New

Kubeapiserver is crucial part so first we need to ensure that secure, 2 ways of thinking about security

- Who can access it and what they can access?
- Who: Password and username, Username and token, certificates, LDAP, Svc Acc.

- What : RBAC, ABAC, webhook, node authorization
- By default all pods can talk to all other pods within cluster, however we can control using RBAC
- In case you provisioning server using kubeadm we need to specify the path of .csv(username and password) in kubeapiserver.yaml, in case you are provisioning manually it should inside kube-apiserver.service

https://pluralsig…ght=540

let's assume we want to connect tot the server, so server has private and public cert, the public cert must signed from CA, then only we are going to share our private key to server, even browser will have same relation(the public cert must signed from CA,) b/w browser and CA.
Naming convention: Public crt >> .crt / .pem file and Private key>> .pem / .key

So, three types of certificates,

server certificates configured on the servers,**root certificate** configured on the CA servers, and then **client certificates** configured on the clients.

We learned that the CA has its own set of public and private keepers that it uses to sign server certificates. We will call them **root certificates.**

For this, as part of the initial trust building exercise, the server can request a certificate from the client. And so the client must generate a pair of keys and a signed certificate from a valid CA. The client then sends the certificate to the server for it to verify that the client is who they say they are.
Now, you must be thinking, you have never generated a client certificate to access a website.
Well, that's because TLS client certificates are not generally implemented on web servers. Even if they are, it's all implemented under the hosts.
So a normal user don't have to generate and manage certificates manually.So that was the final piece about client certificates

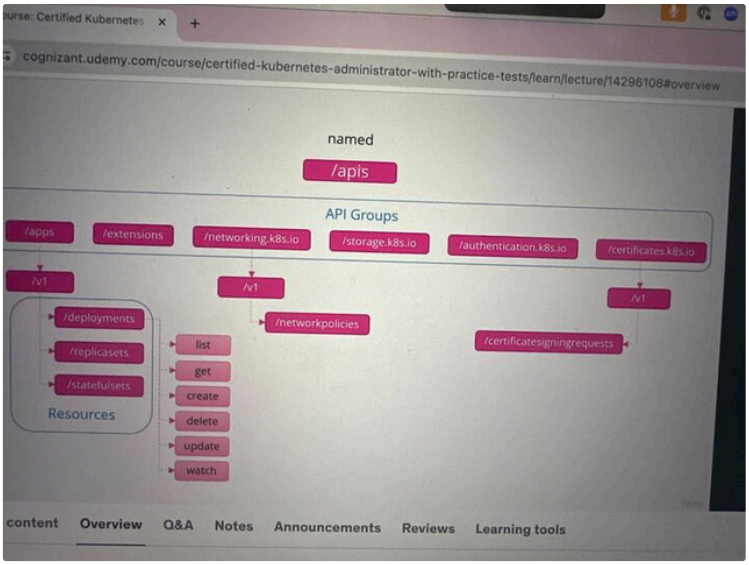
generating the Kuberentes component certs by using openssl

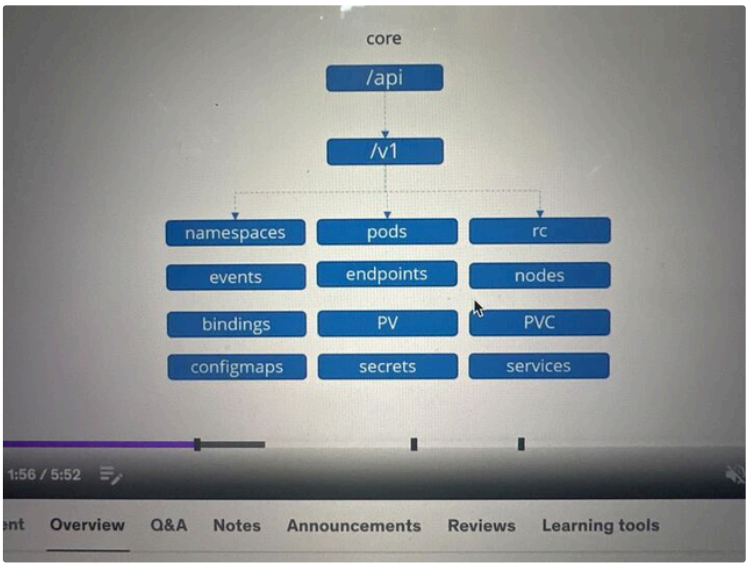

https://pluralsig…ght=370


https://pluralsig…ght=232

- we are ging to generate CA key(ca.key), then ca.csr(request to CA for signing), ca.crt(signed crt)
- we are going to generate all client side certificate(kube-admin, scheduler, control-manager, kubelet)
- inorder to distingiush b/w kube-admin(admin users)[if CN=Kube-admin or any name] and other master component certificate()[if CN=Kube-admin or any name/O=system:masters] similarly replace master with scheduler, control-manager, kubelet.

  ca.cert should be there in all the components yaml file to verify the client is connected is valid or not.
- In order to generate the kube-apiserver.crt, there are other common names such as Kubernetes, Kubernetes.default, Kubernetes.default.svc, Kubernetes.default.svc.cluster.local, we can store all this alternatives names as a dns key in openssl.cnf
- we can check all these alternative name ad details in server.crt file by using **OpenSSL Syntax:** `openssl x509 -in file-path.crt -text -noout`
- In order to generate the kubelet server.crt, crt will have nodes name associated with it. [node01, node02, node03]
  In order to generate the kubelet client.crt, crt will have system:node:node01 name associated with it.
- Kube-control manager is responsible for certificate related stuff
- new admin>> CSR>> approve /reject by existing admins>> control manager is responsible for getting is signing from CA (which is in Master node)
- for new CSR >> decode the request `cat myuser.csr | base64 | tr -d "\n"` using create a csr obj. and the decoded value in csr object along the with changing the name >> then approve or deny
- To use that context, run the command: `kubectl config --kubeconfig=/root/my-kube-config use-context research`
  To know the current context, run the command: `kubectl config --kubeconfig=/root/my-kube-config current-context`

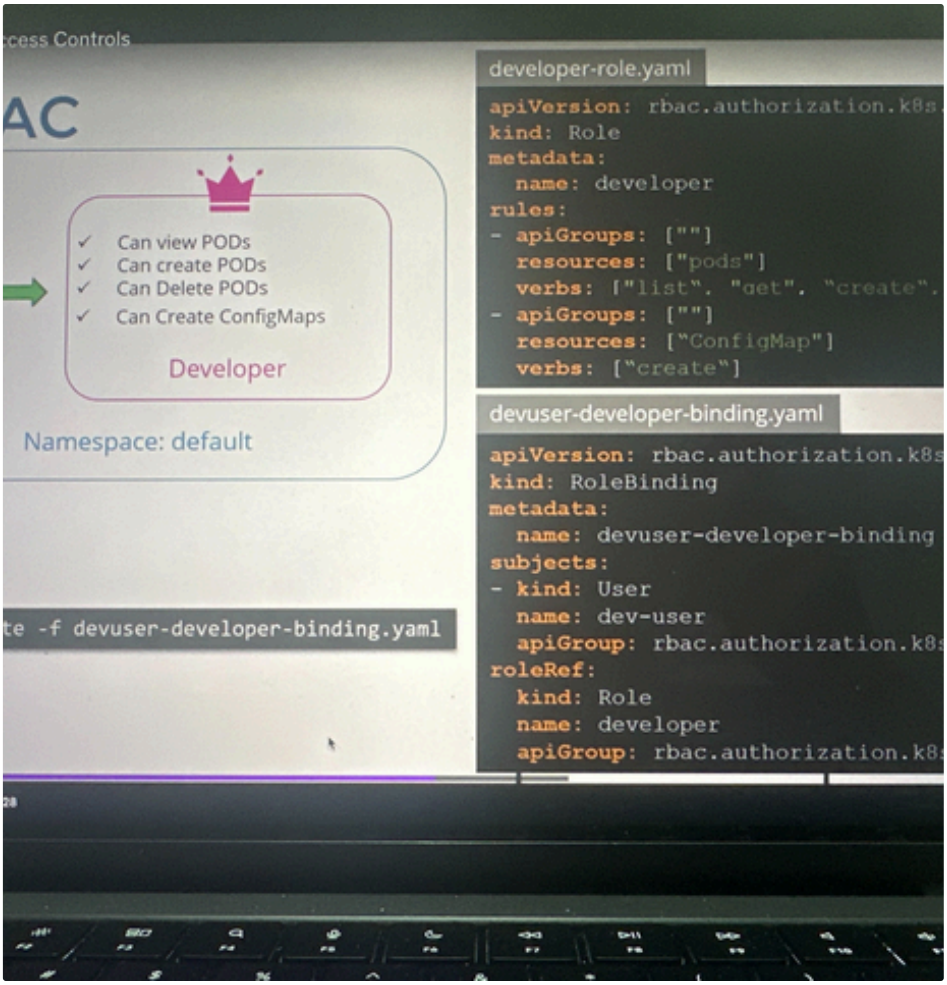




- So here are two terms that kind of sound the same; The `Kube proxy` and `Kube control proxy`. Well, they're not the same.
- Kube proxy It is used to enable connectivity between pods and services across different nodes in the cluster. Whereas `Kube control proxy` is an ACTP proxy service created by `Kube control utility` to access the `Kube API server`.
- So what to take away from this? All resources in Kubernetes are grouped into different API groups.
- At the top level, you have core API group and named API group. Under the named API group, you have one for each section.Under these API groups, you have the different resources and each resource has a set of associated actions known as verbs.

Authorization:

- Role and Role binding:
- role: roles defines the permission of user
- rolebinding: binding the user to role, if you are not going to mention any namespace in metada section rolebinding object, it will choose default namespace.



- **Service Accounts in Kubernetes**:
  - Used by machines (applications) to interact with the Kubernetes API.
  - Different from user accounts, which are used by humans.
- **Types of Accounts**:
  - **User Account**: For administrators or developers accessing the cluster.
  - **Service Account**: Used by applications like Prometheus or Jenkins to interact with the cluster.

Creating and Using Service Accounts

- **Creating a Service Account**:
  - Command: `kubectl create serviceaccount <account-name>`
  - Command: `kubectl get serviceaccount`
- **Service Account Tokens**:
  - Automatically created and stored as a secret object.
  - Command to view token: `kubectl describe secret <secret-name>`
- **Using the Token**:
  - Token used as a bearer token for authentication.
  - Can be manually copied to authenticate external applications.

Configuring Applications with Service Accounts

- **Applications on Kubernetes**:
  - If hosted on the Kubernetes cluster, service account tokens can be automatically mounted as volumes.
  - Example path: `/var/run/secrets/kubernetes.io/serviceaccount`
- **Default Service Account**:
  - Automatically created for each namespace.
  - Automatically mounted to any new pod unless specified otherwise.
- **Assigning a Specific Service Account to a Pod**:
  - Modify the pod definition file to include a `serviceAccount` field with the desired service account name.
  - Note: Existing pods cannot be modified; they must be recreated. However deployment you can directly edit.

Changes in Kubernetes Versions 1.22 and 1.24

- **Kubernetes 1.22**:
  - Introduction of the token request API for generating more secure and scalable tokens.
  - Tokens are audience-bound and time-bound.
  - Pods no longer rely on static service account tokens; use tokens from the token request API.
- **Kubernetes 1.24**:
  - Service accounts no longer automatically create secrets with tokens.
  - To generate a token: `kubectl create token <service-account-name>`
  - Tokens generated this way have a defined expiry time (default is 1 hour).
- **Creating Non-Expiring Tokens**:
  - Manually create a secret object with type `kubernetes.io/service-account-token` if needed.

Best Practices

- **Security Considerations**:
  - Prefer using the token request API for creating tokens with bounded lifetimes.
  - Avoid creating non-expiring tokens unless absolutely necessary due to security risks.
- **Disabling Automatic Service Account Mounting**:
  - Set `automountServiceAccountToken` to `false` in the pod spec to prevent automatic mounting.

To get the info of service account from pod, see the pod in yaml format using kubectl get po -o yaml

You shouldn't have to copy and paste the token each time. The Dashboard application is programmed to read token from the secret mount location. However currently, the `default` service account is mounted. Update the deployment to use the newly created ServiceAccount

Edit the deployment to change ServiceAccount from `default` to `dashboard-sa`.

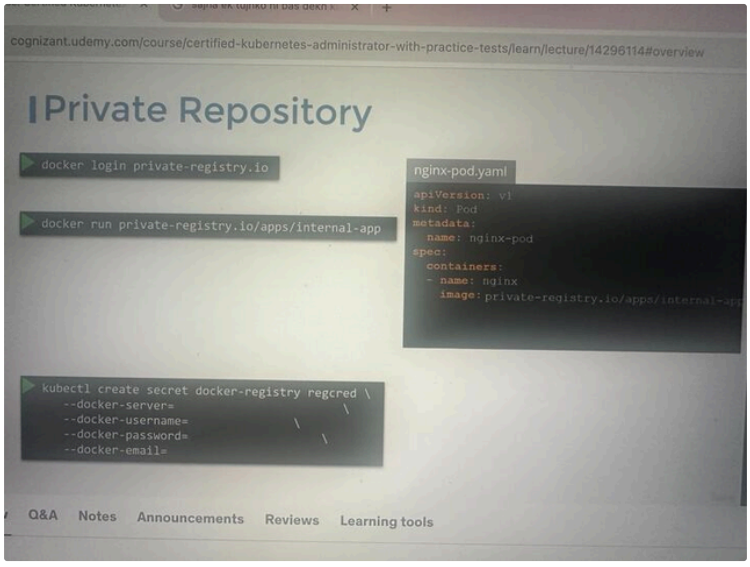
Use the `kubectl edit` command for the deployment and specify the `serviceAccountName` field inside the pod spec. OR

Make use of the `kubectl set` command. Run the following command to use the newly created service account: - `kubectl set serviceaccount deploy/web-dashboard dashboard-sa`

oR add the src acc name inside the spec section of (even while editing deployment)

Image Security:

if you want to pull the image from private registry to k8s cluster, we wre going to store the docker cred's in secret and in image section we will mention **"imagepullsecret"**



`kubectl create secret docker-registry private-reg-cred --docker-username=dock_user --docker-password=dock_password --docker-server=myprivateregistry.com:5000 --docker-email=dock_user@myprivateregistry.com`

q:
What is the user used to execute the sleep process within the `ubuntu-sleeper` pod?In the current(default) namespace.
Run the command: `kubectl exec ubuntu-sleeper -- whoami` and check the user that is running the container.

Network Policy:

Basic Networking and Security:

- **Traffic Flow Example:**

- User requests sent to web server at port 80.
- Web server sends requests to API server at port 5000.
- API server fetches data from the database server at port 3306.
- Types of traffic:
  - **Ingress:** Incoming traffic to a node.
  - **Egress:** Outgoing traffic from a node.

Kubernetes Networking:

- **Cluster Communication:**
  - Nodes, pods, and services each have unique IP addresses.
  - Pods communicate without additional routing configurations.
  - Default policy: All pods can communicate with each other.

Network Policies:

- Used to restrict pod communication within a cluster.
- Defined using Kubernetes network policy objects.
- **Policy Structure:**
  - **API version:** `networking.k8s.io/v1`
  - **Kind:** `NetworkPolicy`
  - **Metadata:** Name, namespace
  - **Spec:**
    - **Pod selector:** Specifies which pods the policy applies to.
    - **Policy types:** Ingress, Egress, or both.
    - **Rules:** Define specific traffic rules (e.g., allow ingress on port 3306 from API server).

Implementing Network Policies:

- **Define Labels and Selectors:** Link network policies to specific pods using labels.
- **Example Policy:**
  - **Ingress Rule:** Allows traffic from API pod on port 3306.
  - **Selectors:** Use labels to identify the API pod.

Network Solutions:

- Network policies are enforced by the network solution used in the Kubernetes cluster.
- Supported solutions: Cube Router, Calico, Romana, WaveNet.
- Unsupported solution: Flannel.

Also, remember, even in a cluster configured with a solution that does not support network policies,you can still create the policies but they will just not be enforced. You will not get an error message saying the network solution does not support network policies.

ques:
Create a network policy to allow traffic from the `Internal` application only to the `payroll-service` and `db-service`.

Use the spec given below. You might want to enable ingress traffic to the pod to test your rules in the UI.

Also, ensure that you allow egress traffic to DNS ports TCP and UDP (port 53) to enable DNS resolution from the internal pod.

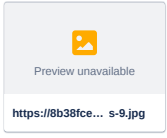
Policy Name: internal-policy

Policy Type: Egress

Egress Allow: payroll

Payroll Port: 8080

Egress Allow: mysql



Solution manifest file for a network policy `internal-policy` as follows:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: internal-policy
  namespace: default
spec:
```

```
  podSelector:
    matchLabels:
      name: internal
  policyTypes:
  - Egress
  - Ingress
  ingress:
    - {}
  egress:
  - to:
    - podSelector:
        matchLabels:
          name: mysql
    ports:
    - protocol: TCP
      port: 3306

  - to:
    - podSelector:
        matchLabels:
          name: payroll
    ports:
    - protocol: TCP
      port: 8080

  - ports:
    - port: 53
      protocol: UDP
    - port: 53
      protocol: TCP
```

Note: We have also allowed `Egress` traffic to `TCP` and `UDP` port. This has been added to ensure that the internal DNS resolution works from the `internal` pod.

Remember: The `kube-dns` service is exposed on port `53`:

```
root@controlplane:~> kubectl get svc -n kube-system
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
kube-dns   ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   18m

```

A quick note on editing Pods and Deployments

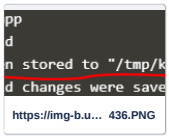
Edit a POD

Remember, you CANNOT edit specifications of an existing POD other than the below.

- spec.containers[*].image
- spec.initContainers[*].image
- spec.activeDeadlineSeconds
- spec.tolerations

For example you cannot edit the environment variables, service accounts, resource limits (all of which we will discuss later) of a running pod. But if you really want to, you have 2 options:

1. Run the `kubectl edit pod <pod name>` command. This will open the pod specification in an editor (vi editor). Then edit the required properties. When you try to save it, you will be denied. This is because you are attempting to edit a field on the pod that is not editable.



A copy of the file with your changes is saved in a temporary location as shown above.

You can then delete the existing pod by running the command:

`kubectl delete pod webapp`

Then create a new pod with your changes using the temporary file

`kubectl create -f /tmp/kubectl-edit-ccvrq.yaml`

2. The second option is to extract the pod definition in YAML format to a file using the command

`kubectl get pod webapp -o yaml > my-new-pod.yaml`

Then make the changes to the exported file using an editor (vi editor). Save the changes

`vi my-new-pod.yaml`

Then delete the existing pod

`kubectl delete pod webapp`

Then create a new pod with the edited file

`kubectl create -f my-new-pod.yaml`

Edit Deployments

With Deployments you can easily edit any field/property of the POD template. Since the pod template is a child of the deployment specification, with every change the deployment will **automatically delete and create a new pod** with the new changes. So if you are asked to edit a property of a POD part of a deployment you may do that simply by running the command

`kubectl edit deployment my-deployment`

For pod creation with cmd

Create a pod definition file in the manifests folder. To do this, run the command:
`kubectl run --restart=Never --image=busybox static-busybox --dry-run=client -o yaml --command -- sleep 1000 > /etc/kubernetes/manifests/static-busybox.yaml`

- [ ] 1000 rs for restart policy, paramesh lost bet

Service:

Run the command: `kubectl expose pod <pod-name> --port=<port-number> --name <service-name>`

`kubectl expose deployment <deployment-name> --type=NodePort --port=<port-number>`

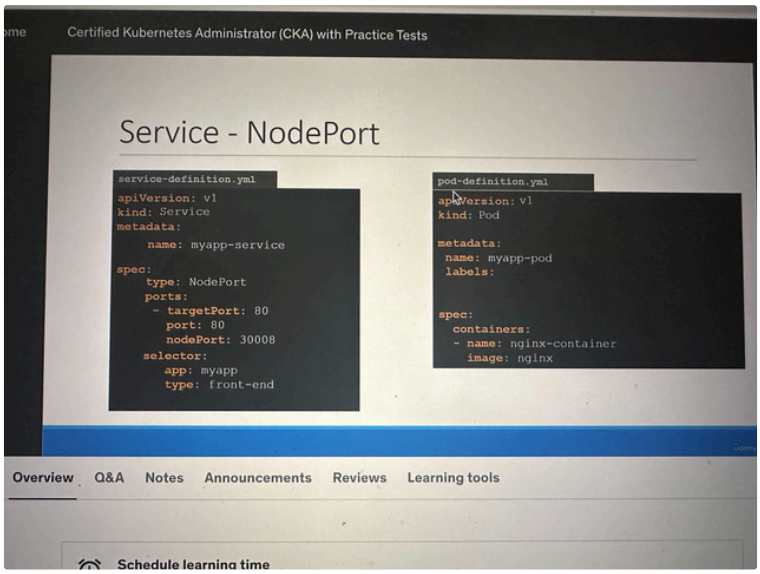
Create a new pod called `custom-nginx` using the `nginx` image and expose it on `container port 8080`.

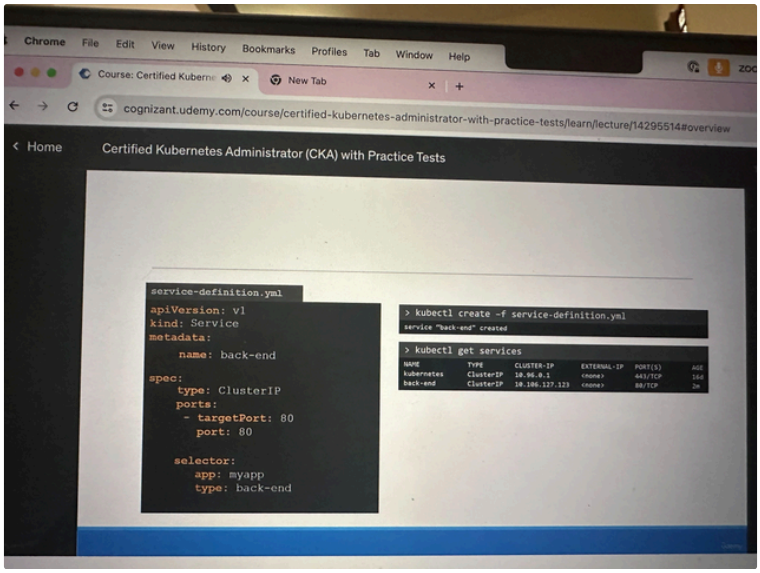
`kubectl run text1 --image=nginx --port=8080`

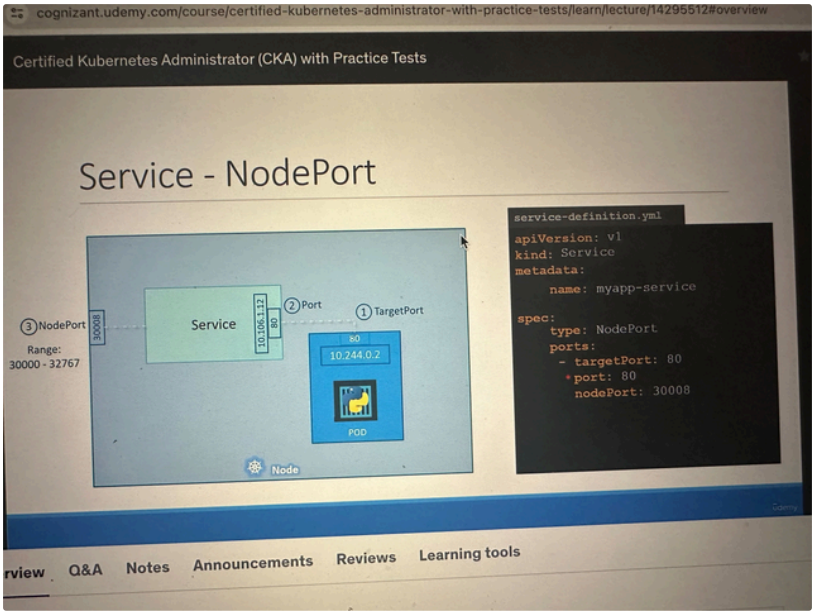
- It uses the NGINX Docker image from the Docker Hub repository.
- The deployment exposes port 8080 within the container.
- This command creates a deployment object, which manages a set of replica pods running the NGINX container.

Run the command: `kubectl run text --image=httpd:alpine --port=80 --expose`

- The deployment exposes port 80 within the container.
- Additionally, it exposes the created deployment as a Kubernetes service, making it accessible within the Kubernetes cluster.
- By default, the `--expose` flag creates a service of type "ClusterIP", which means the service is only accessible from within the Kubernetes cluster.







The deployment exposes port 8080 within the container. meaning?

Let me clarify what "exposes port 8080 within the container" means in the context of the command `kubectl run custom-nginx --image=nginx --port=8080`.

When we specify `--port=8080` in the `kubectl run` command, it indicates that the container within the pod, which is managed by the deployment created with this command, will listen for incoming network traffic on port 8080. This port is exposed within the container itself, meaning that processes running inside the container can accept incoming connections on port 8080.

In practical terms, if there's a web server running inside the NGINX container (which is the case with the default NGINX image), it would typically be configured to listen for HTTP requests on port 8080. So, any incoming HTTP requests to the NGINX service would be directed to port 8080 within the container.

To access the NGINX service running in this deployment from outside the Kubernetes cluster, you would typically create a Kubernetes service of type `NodePort` or `LoadBalancer` to expose this port externally. This would allow external clients to reach the NGINX service by connecting to the appropriate node IP address or load balancer IP and port combination.

kubectl apply: local file ---> last aplied file-----> live config

ex: let's assume we have deployment with 3 replicas.(meaning last applied 3 and live config 3) . so now we are changing the replicas from 3 to 2 , so it will consider the last applied file first as source of truth and compares with the local and make changes in live and later update the lat applied with the latest changes.

Replication Controller vs Replica set

ReplicationController uses **v1** and Replicaset uses **apps/v1**

ReplicationController yaml file consist api, kind, metadata: name, labels and spec: template (pod definition file without api and kind)and replicas

Replicaset yaml file consist api, kind, metadata: name, labels and spec: template (pod definition file without api and kind)and replicas and **selector: matchLabels:**
Run the command: `kubectl edit replicaset new-replica-set`, modify the `replicas` and then save the file OR run: `kubectl scale rs new-replica-set --replicas=5`

for creating new replicaset using kubectl create cmd, first create deploy then >> -o yaml>>

```
kubectl create deployment --image=nginx nginx --dry-run=client -o yaml > nginx-deployment.yaml
```

then vi `nginx-deployment.yaml` change kind to RS then create

Deployment

Create an NGINX Pod

```
kubectl run nginx --image=nginx
```

Generate POD Manifest YAML file (-o yaml). Don't create it(--dry-run)

```
kubectl run nginx --image=nginx --dry-run=client -o yaml
```

Create a deployment

```
kubectl create deployment --image=nginx nginx
```

Generate Deployment YAML file (-o yaml). Don't create it(--dry-run)

```
kubectl create deployment --image=nginx nginx --dry-run=client -o yaml
```

Generate Deployment YAML file (-o yaml). Don’t create it(–dry-run) and save it to a file.

```
kubectl create deployment --image=nginx nginx --dry-run=client -o yaml > nginx-deployment.yaml
```

Make necessary changes to the file (for example, adding more replicas) and then create the deployment.

```
kubectl create -f nginx-deployment.yaml
```

OR

In k8s version 1.19+, we can specify the --replicas option to create a deployment with 4 replicas.

```
kubectl create deployment --image=nginx nginx --replicas=4 --dry-run=client -o yaml > nginx-deployment.yaml
```

KubeAPIserver/kube control manager/scheduler/kubelet/kube-proxy

- authenticate-user
- validate request
- updating the etcd

Then, whenever you deploy a masternode component, such as kube-apiserver using kubeadm, it will be created in the kube-system namespace.

```
/etc/kubernetes/manifests/
```

controller-manager: nodecontroller | replication controller (similarly for everything like automation or intelligence we have a controller its like brain for k8s)

For every 5 seconds, it will check the health status, with a grace period of 40 seconds. If there's no response within the initial 40 seconds, it will wait for an additional 5 minutes; if there's still no response, all pods will be evicted from that node to a healthy one.

Replication controller: respnsible for maintainting actual state sould be equal to desired state

- in a Kube admin setup, the Kube controller manager is deployed as a POD in the Kube system namespace on the master node.
- Options can be viewed within the POD definition file located at Etsy Kubernetes Manifest folder.
- In a non-Kube admin setup, options can be inspected by viewing the Kube Controller Manager's service located at the services directory.

Scheduler: it is responsible for selecting the appropriate node based on certain critiria(cpu, memory,nodeselector, affinity, taints and toleration) **scheduling will be taken care by kubelet.**

kubelet: it is a captain component in worker node- it will establish communication b/w master node and worker node

- Kubelet is not automatically deployed if kubeadm is used to deploy the cluster.
- Manual installation on worker nodes is required.
- Installation involves downloading the installer, extracting it, and running it as a service.

kube-proxy: it will responsible for attaching the ip for the service based on ip-table rules(forwarding the request to other service)

- Kube-proxy configures services by creating iptables rules.
- Forwards traffic from service IP to pod IP.

CRI

In the early days, Docker dominated containerization. Kubernetes was tightly coupled with Docker for orchestration. As Kubernetes grew, it introduced the Container Runtime Interface (CRI) to support other runtimes like rkt. Docker, not built for CRI, relied on a hacky solution called dockershim. Containerd emerged, complying with CRI and becoming Kubernetes-

compatible. Eventually, Kubernetes phased out dockershim, removing direct Docker support but maintaining compatibility with Docker-built images. Docker CLI, API, and other tools remain, while Containerd serves as a standalone runtime for Kubernetes.

Docker itself big eco system i.e. Docker CLI, API, and other tools remain, k8s came to orchestrate docker, but there were many other container runtime tool, so K8s created CRI whoever falls under the OCI standards So OCI stands for `Open Container Initiative` ,and it consists of an `imagespec` (how an image should be built.)and a `runtimespec` (how any container runtime should be developed.)

ContainerD is part of docker eco system and matches CRI standards, therefore that's the one thing k8s supports, but containerd became independent tool, that means we can install containerd only without docker, but it have only limited fuctionality: you had the ctr and the nerd control (almost same cmds like docker just replace docker with `nerd control` in all cmds)that were built by the Containerd community, specifically for Containerd

crictl provides a CLI for CRI compatible container runtimes, Installed separately • Used to inspect and debug container runtimes • **Not to create containers ideally.**

crictl is created by k8s and runs and supports all the container runtime

Retrieve debugging information

docker cli	crictl	Description	Unsupported Features
attach	attach	Attach to a running container	--detach-keys , --sig-proxy
exec	exec	Run a command in a running container	--privileged , --user , --detach-keys
images	images	List images	
info	info	Display system-wide information	
inspect	inspect , inspecti	Return low-level information on a container, image or task	
logs	logs	Fetch the logs of a container	--details
ps	ps	List containers	
stats	stats	Display a live stream of container(s) resource usage statistics	Column: NET/BLOCK I/O, PIDs
version	version	Show the runtime (Docker, ContainerD, or others) version information	

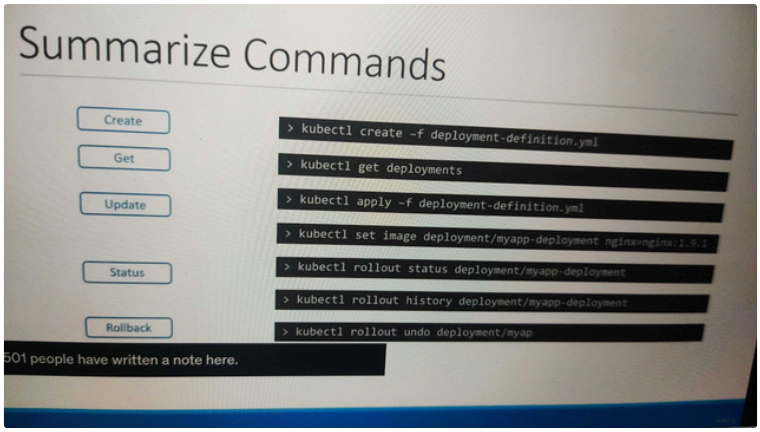
Perform Changes

docker cli	crictl	Description	Unsupported Features
create	create	Create a new container	
kill	stop (timeout = 0)	Kill one or more running container	--signal
pull	pull	Pull an image or a repository from a registry	--all-tags , --disable-content-trust
rm	rm	Remove one or more containers	
rmi	rmi	Remove one or more images	
run	run	Run a command in a new container	
start	start	Start one or more stopped containers	--detach-keys
stop	stop	Stop one or more running containers	
update	update	Update configuration of one or more containers	--restart , --blkio-weight and some other resource limit not supported by CRI.

```
$ ctr images pull docker.io/library/redis:alpine

$ ctr run docker.io/library/redis:alpine redis
```

Application Cycle lifecycle



In case of the CMD instruction, the command line parameters passed will get replaced entirely. Whereas in case of entry point, the command line parameters will get appended. CMD [“sleep 5”]

CMD: if only cmd command is there in docker file we can overide while running the image but we have to specify sleep 10 like this docker run image name sleep 10

In Docker, when we use `ENTRYPOINT`, it can't be overridden during runtime. Therefore, we're setting a key to "sleep". As for `CMD`, we'll define the value as ["10"]. So, when we run the Docker image, it'll execute the command for 15 seconds.

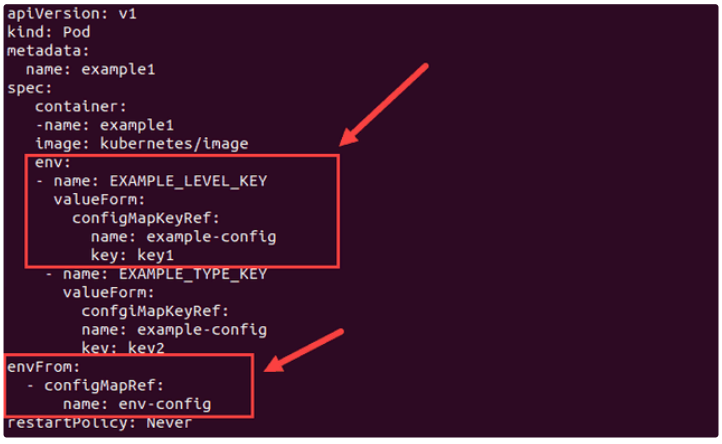
Command vs arguement:
So to summarize, there are two fields that correspond to two instructions in the docker file. The `command` field overrides the `entry point` instruction, and the `args` field overrides the `command` instruction in the docker file. Remember, it is not the command field that overrides the cmd instruction in the docker file.

CM vs secrets

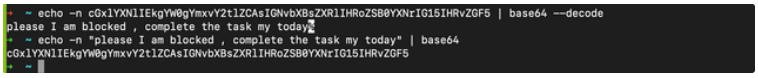
config Maps: used to store non-senstive data like dbhost.. etc

aptversion, kind, metadata, data

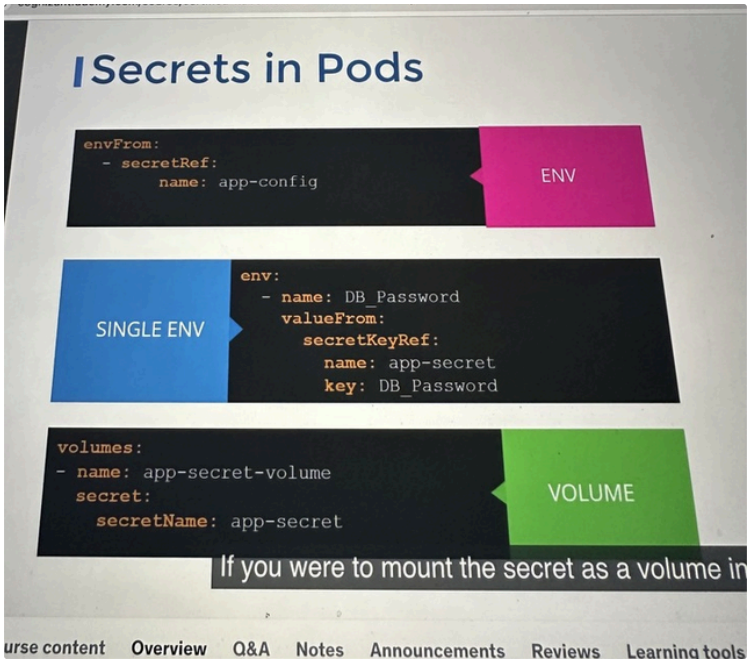
to attach to pod we have 2 ways



secret: same but we need to encode the values while writing the yaml file



Multiple ways to mount secret



Imperative cmd

Note: for creating secret using imperative cmd use **generic** word before the secret name

`kubectl create secret generic my-secret --from-literal=username=myuser --from-literal=password=mypass`

```
kubectl create configmap <configmap-name> --from-literal=<key1>=<value1> --from-literal=<key2>=<value2>
```

Or you can create a ConfigMap from a file:

`kubectl create configmap <configmap-name> --from-file=<path-to-file>`

How long after the creation of the POD will the application come up and be available to users?How long after th creation of the POD will the application come up and be available to users?

Check the commands used in the `initContainers`. The first one sleeps for 600 seconds (10 minutes) and the second one sleeps for 1200 seconds (20 minutes)