Contents

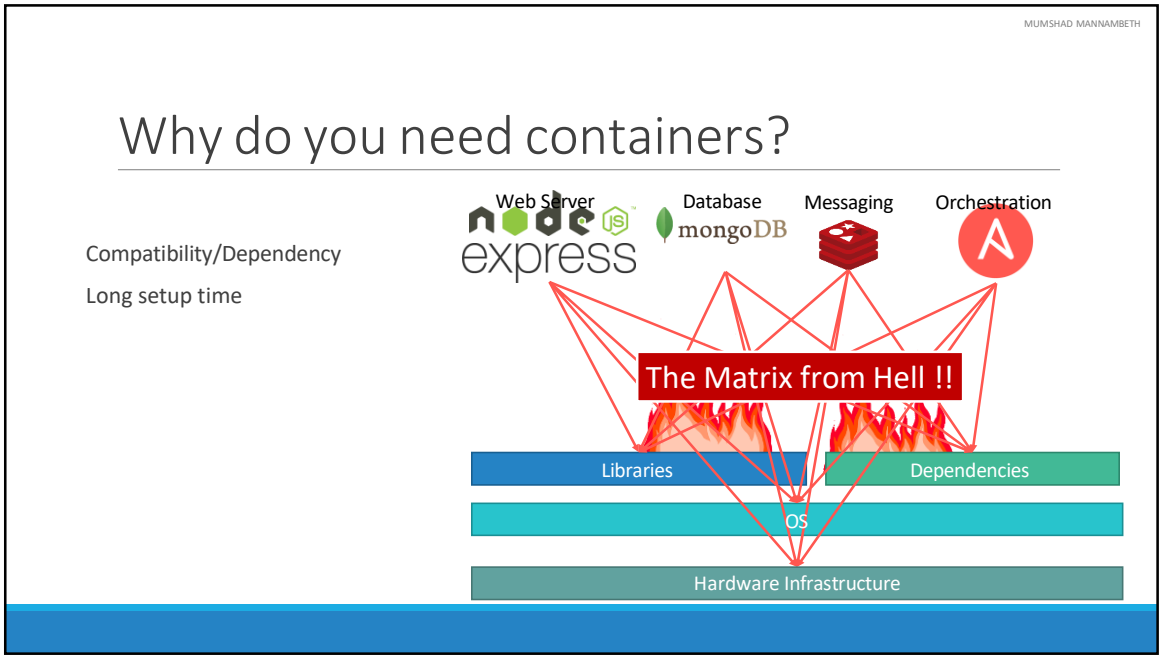
Kubernetes Overview

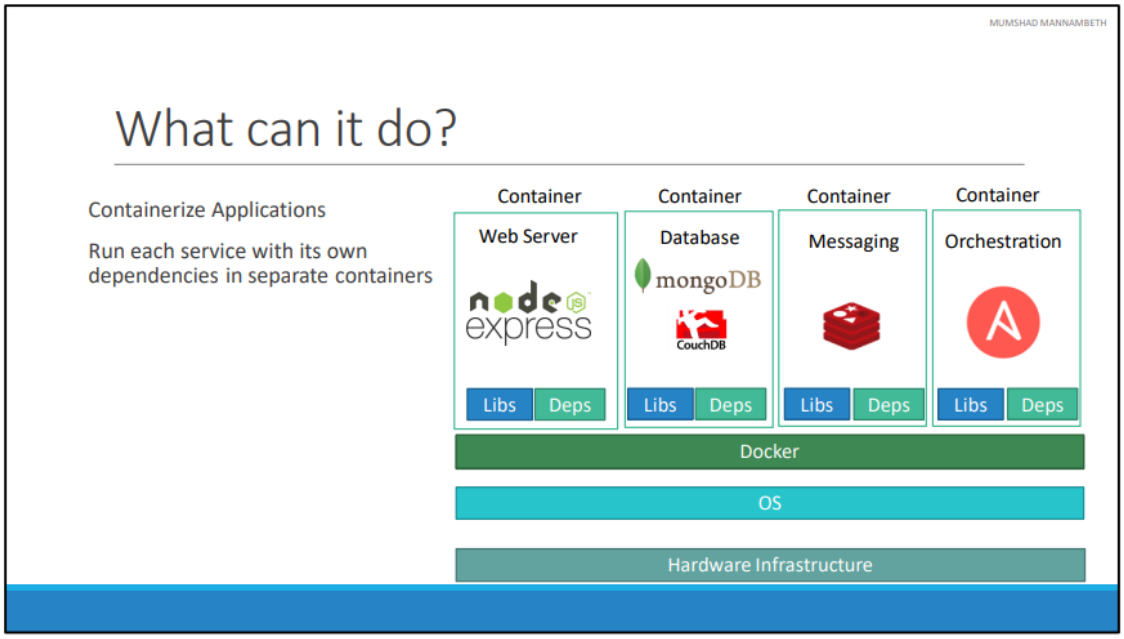
- It is developed by google and now an open-source project. It is also known as k8s

- Kubernetes is an open-source container orchestration platform to manage containerized workload. It provides both declarative (via kubectl) and automation way of managing them.
- Kubernetes is a container Orchestration technology used to orchestrate the deployment and management of 100s and 1000s of containers in a clustered environment.

Containers

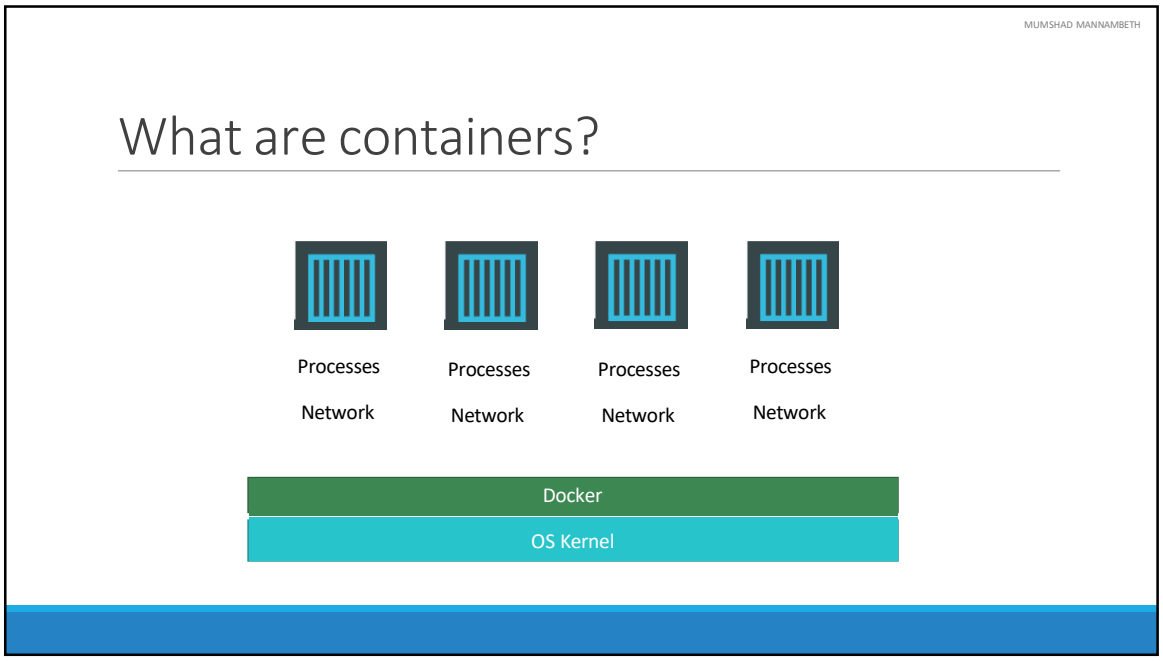
- Before docker things like -> setting up environment with different set of technology and tools was very difficult. Like tomcat, mongodb, redis etc all these must be compatible with OS first. Then what are compatibility among these services like what versions and dependency are compatible with each other. If architecture changes and we need to upgrade version of one component, then again everything need to be followed again.
- Whenever new developer comes setting up env requires lots of steps and instructions and takes time to do so. And for different environments like prod, test, dev and it was tough to make sure application working on 1 env will work in a exact same way on other env also.



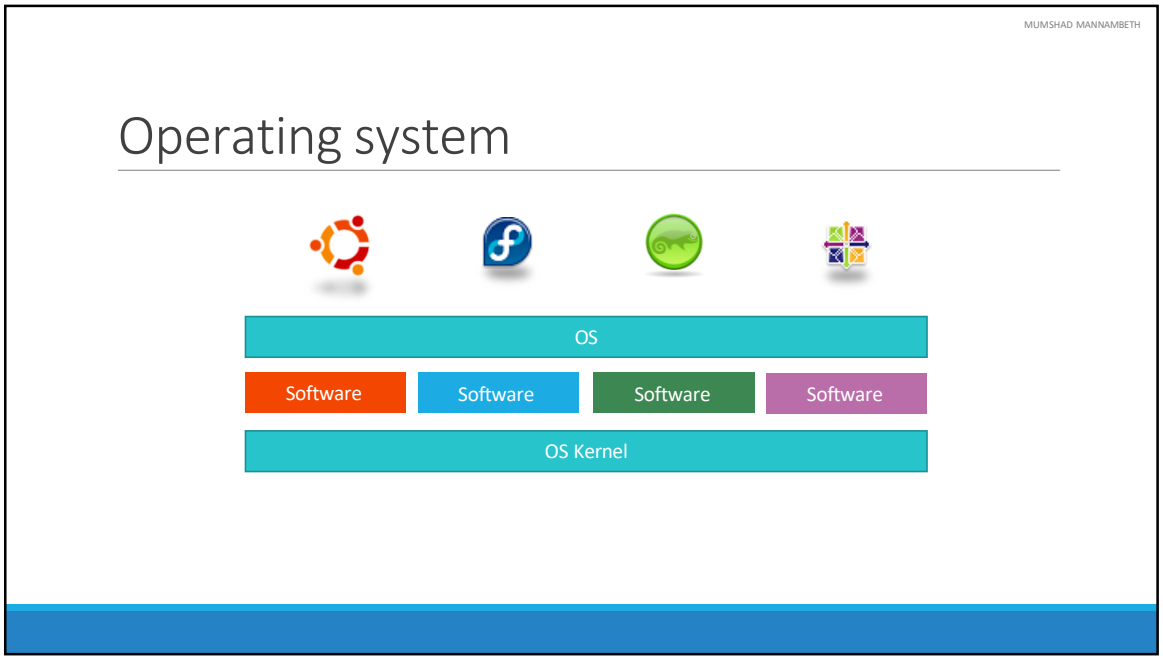


- We need something to make compatibility issue easier to deal and make each component independent of each other. So that change in 1 component does not impact other. Answer Is docker
- Docker is an open source containerization platform. Docker is an open platform for developing, shipping, and running applications. Application runs on a loosely isolated environment called a container.

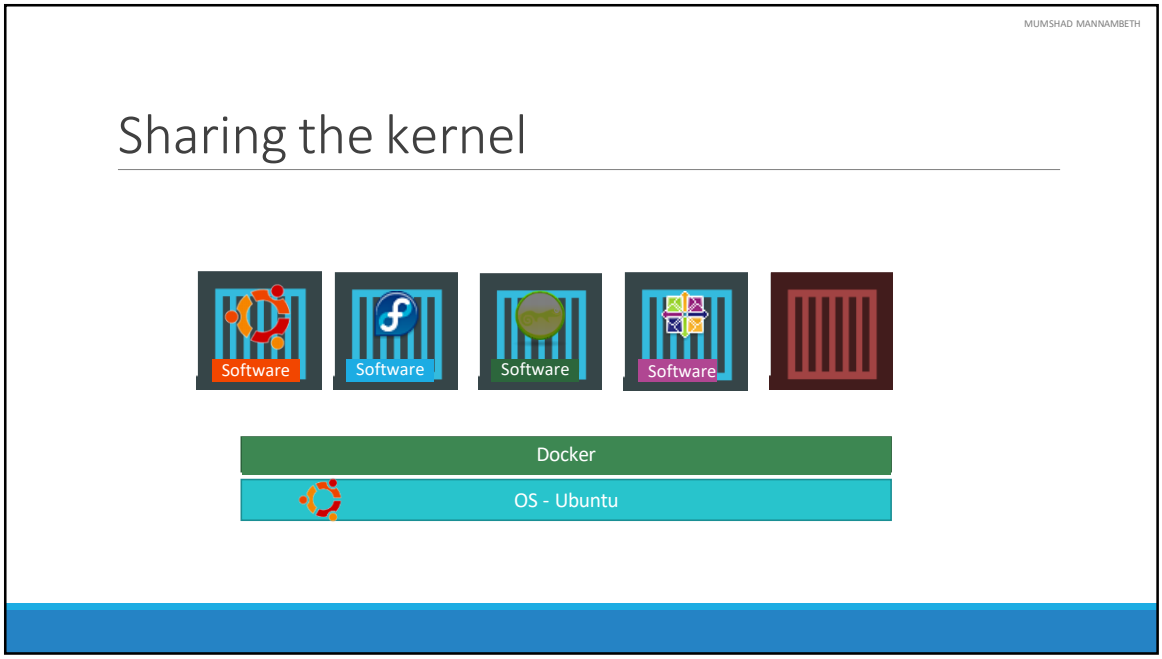
- With Docker I was able to run each component in a separate container – with its own libraries and its own dependencies. All on the same VM and the OS, but within separate environments or containers. We just had to build the docker configuration once, and all our developers could now get started with a simple “docker run” command.
- Container is a running instance of image.
- Containers are isolated environments which can have own set of process, services, network interfaces, own mounts just like VM except they all share same OS kernal. i.e. they don’t have their own OS.



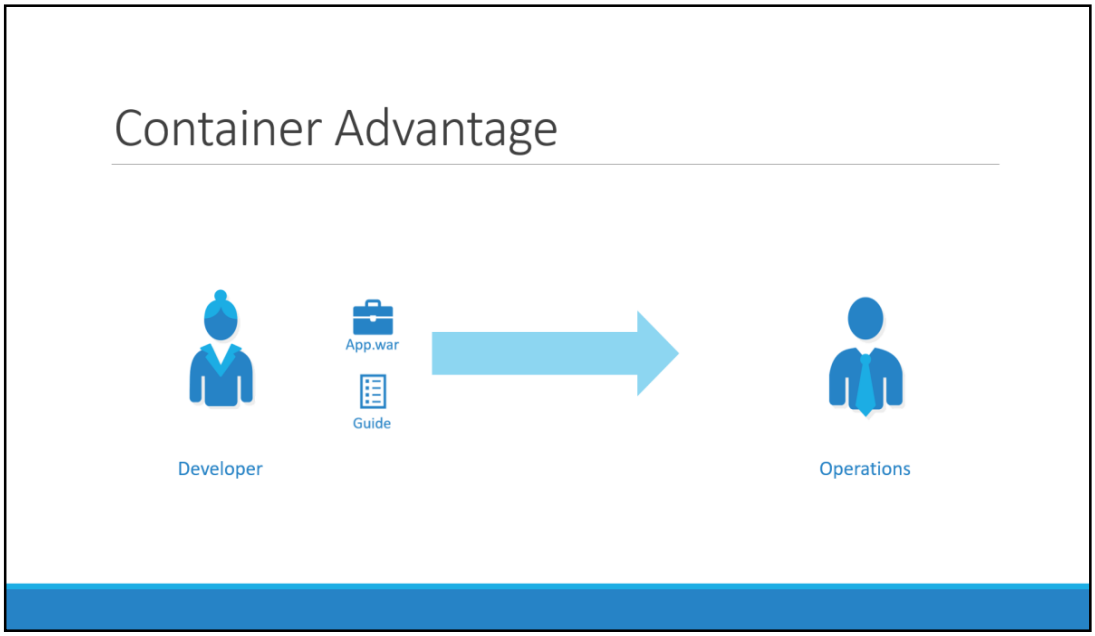
- Container existed from starting and docker uses LXC containers. Setting up these container environments is hard as they are very low level and that is were Docker offers a high-level tool making it really easy for end users like us.



- If you look at operating systems like Ubuntu, Fedora, Suse or Centos –they all consist of two things. An OS Kernel and a set of software. The OS Kernel is responsible for interacting with the underlying hardware. While the OS kernel remains the same– which is Linux in this case, it’s the software above it that make these Operating Systems different. This software may consist of a different User Interface, drivers, compilers, File managers, developer tools etc. So you have a common Linux Kernel shared across all Oses and some custom softwares that differentiate Operating systems from each other.



- We said earlier that Docker containers share the underlying kernel. What does that mean – sharing the kernel? Let's say we have a system with an Ubuntu OS with Docker installed on it. Docker can run any flavour of OS on top of it as long as they are all based on the same kernel – in this case Linux. If the underlying OS is Ubuntu, docker can run a container based on another distribution like debian, fedora, suse or centos. Each docker container only has the additional software, that we just talked about in the previous slide, that makes these operating systems different and docker utilizes the underlying kernel of the Docker host which works with all Oses above.
- You won't be able to run a windows-based container on a Docker host with Linux OS on it. As they don't share same kernel. So why docker install on windows machine able to run linux based containers on it. This is because behind the scene windows create linux based vm and run all the containers on top of it.

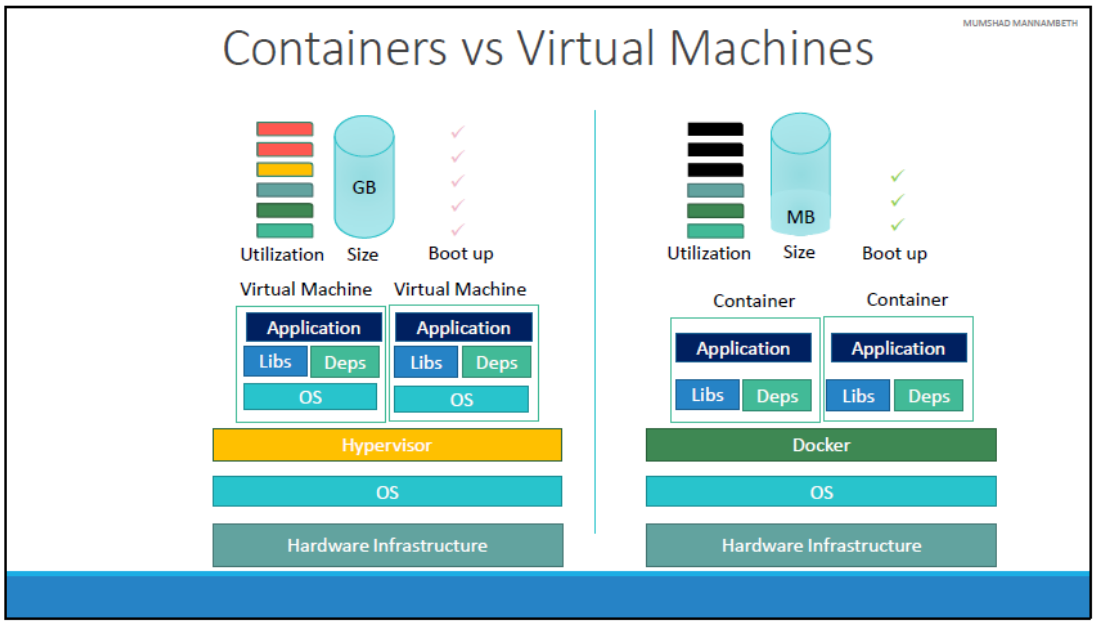


- Traditionally developers developed applications. Then they hand it over to Ops team to deploy and manage it in production environments. They do that by providing a set of instructions such as information about how the hosts must be setup, what pre-requisites are to be installed on the host and how the dependencies are to be configured etc. Since the Ops team did not develop the application on their own, they struggle with setting it up. When they hit an issue, they work with the developers to resolve it.
- The guide that the developers built previously to setup the infrastructure can now easily put together into a Dockerfile to create an image for their applications. This image is guaranteed to run the same way everywhere. So the Ops team now can simply use the image to deploy the application. Since the image was

already working when the developer built it and operations are not modifying it, it continues to work the same when deployed in production.

Containers VS Virtual Machine (VM)

- Docker is not meant to replace VM. The main purpose of Docker is to containerize applications and to ship them and run them.



- VM has own separate OS and hence we can install windows VM on linux OS. Docker don't has own O.S
- VM since has own OS . hence OS takes it's own memory and size. And has high start-up time as entire OS need to bootup.
- VM provide complete isolation with other VM as they have separate OS also. Docker containers share OS kernel. But each container can have own set of processes, services, network interface and mount which are independent of other containers.
- When you run docker run imageName. It will search if image is present in local if not it will search dockerHub for the image and download it and start container.
- An image is a package or a template. Using image you can run multiple containers. Containers are running instances off images that are isolated and have their own environments and set of processes

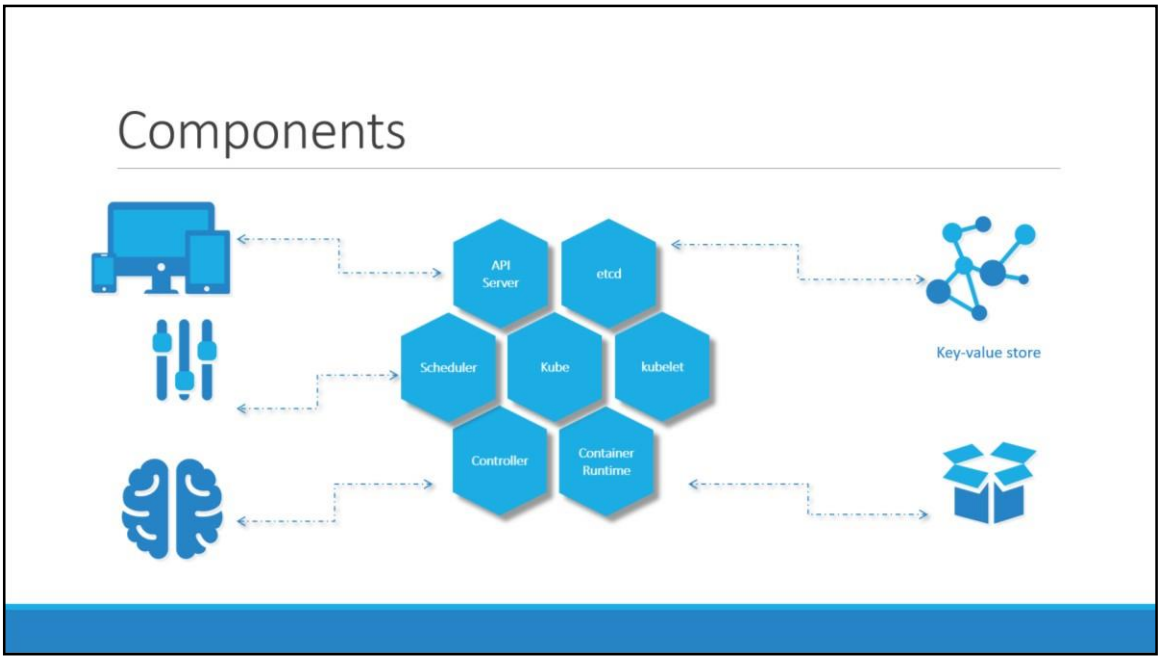
Containers Orchestration

- To manage containers like, When to run more and when to scale down and how to manage version upgrade. Make sure always minimum number of containers always running. We need a platform to manage it. This whole process of automatically deploying and managing containers is known as Container Orchestration.
- Kubernetes is a container Orchestration technology used to orchestrate the deployment and management of 100s and 1000s of containers in a clustered environment. **Apache Mesos and Docker Swarm** are also one of them.
- All public cloud providers like AWS, GCP, Azure has their own k8s service.

Advantages of Kubernetes

- Your application is now highly available as hardware failures do not bring your application down because you have multiple instances of your application running on different nodes.
- The user traffic is load balanced across the various containers. (yes in a way that load is distributed randomly across pods running in a single rs)
- When demand increases, deploy more instances of the application seamlessly and within a matter of second and we have the ability to do that at a service level.
- Scale the number of nodes up/down without having to take down the application.

Architecture

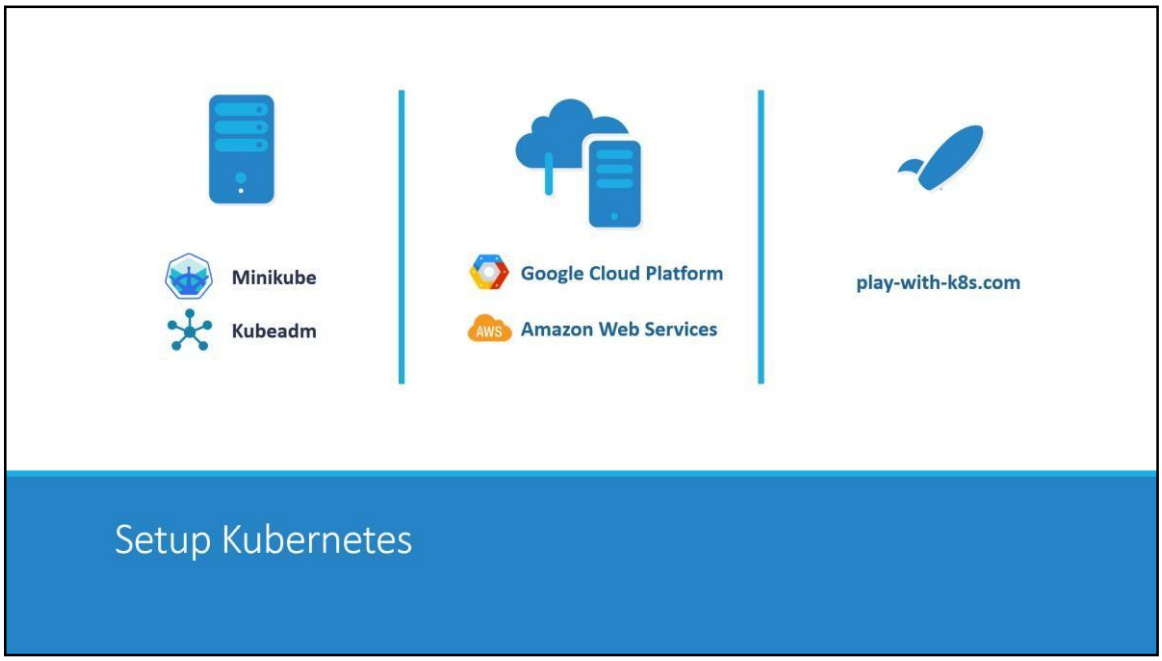


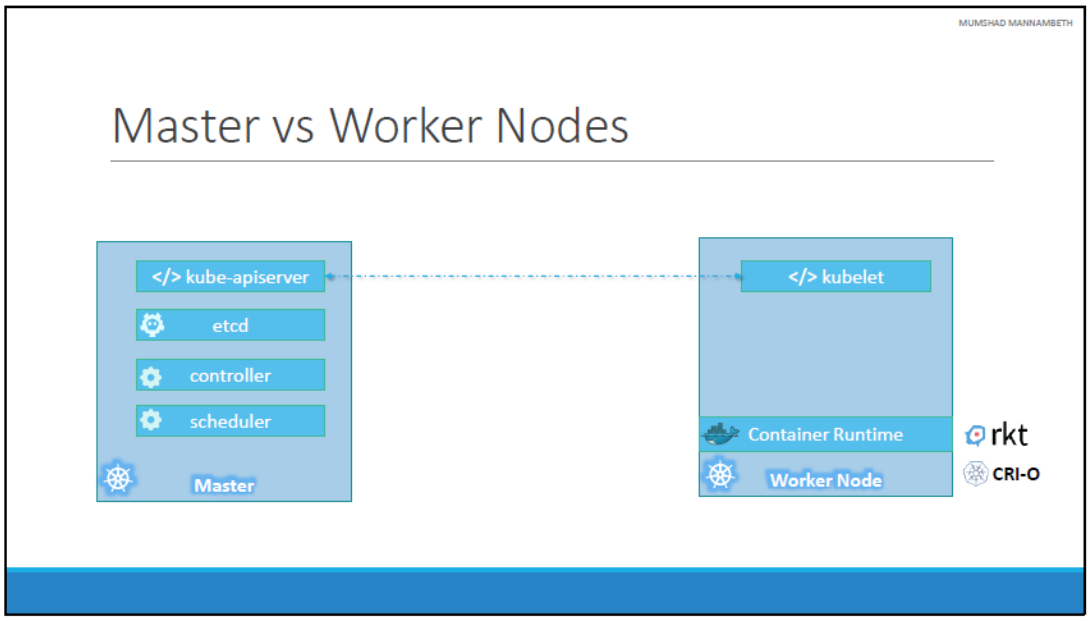
- These are major components of Kubernetes->
- **API server** acts as the front-end for kubernetes. Kubectl (i.e. via command line) or via any user interface of cloud all talk to the API server to interact with the Kubernetes cluster. It is present in Master Node.
- **etcd** is a distributed reliable key-value store used by kubernetes to store all data used to manage the cluster**.** It is present in master. if we have configured multiple master, etcd stores all that information on all the nodes in the cluster in a distributed manner. ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.
- **Scheduler** is responsible to schedule containers to the multiple nodes. It looks for newly created containers and assigns them to Nodes. It is present in Master.
- **Controller** is the brain of kubernetes**.** They are processes that monitor Kubernetes object and respond accordingly.
- They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers makes decisions to bring up new containers in such cases. And hence maintain the desired state of cluster. It is present in Master.
- **Container Runtime** is used to provide runtime to run containers. Example docker runtime will be required to run docker containers. Other container runtimes is rocket. It is present inside worker node(also known as minion node).
- **kubelet** is the agent that is responsible for interacting with the master to provide health information of the worker node and carry out actions requested by the master on the worker nodes. It is present in worker node.
- Two types of nodes are there-> Master and Worker node (also known as nodes or minion nodes).
- **Cluster** is a set of nodes grouped together. This way even if one node fails you have other serving.
- **Master Node** manages the worker node and is responsible to maintain desired state of cluster.
- **Worker node** are nodes where actual application is deployed.
- In public cloud like AWS, GCloud and Azure. They provide there own k8s service. There master node is managed by cloud provider, and we need to specify application only, which we want to run on worker node.

Setup

- We can setup it up ourselves locally on our laptops or virtual machines using solutions like Minikube and Kubeadmin. By using them we don't need to install each component of Kubernetes separately.
- **Minikube** can be used if aim is to setup kubernetes for single node.
- **Kubeadmin** is a tool used to configure kubernetes in a multi-node setup.
- Cloud provider also has k8s services available to use directly like -> EKS(elastic k service), AKS(azure k service), GKE (google k engine)

- We can create cluster by using kubeadm or any public cloud provided option.



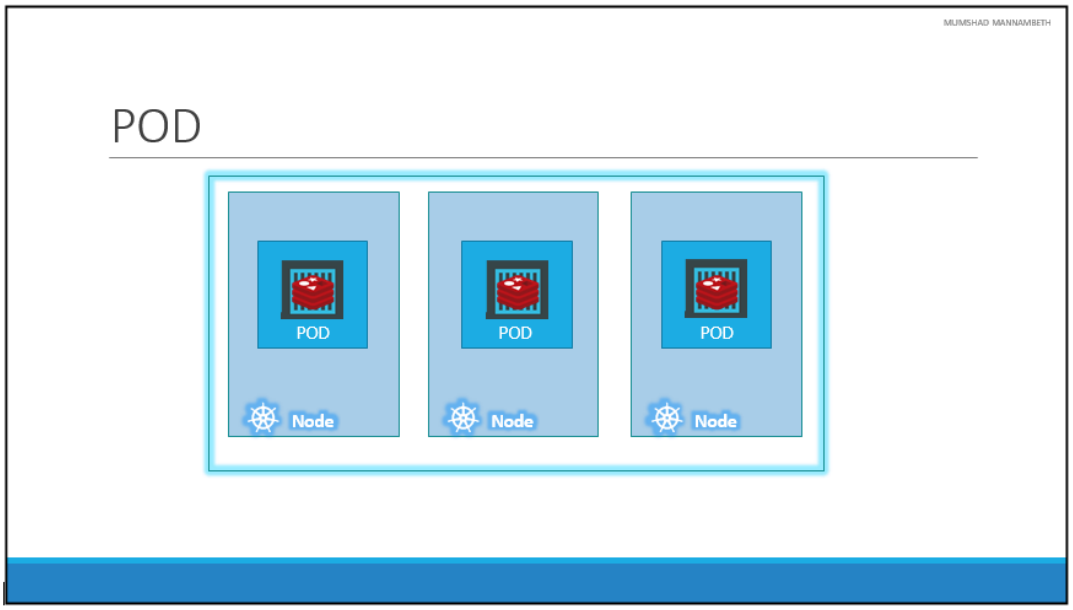


- There are other components as well, but we will stop there for now. And later discuss this

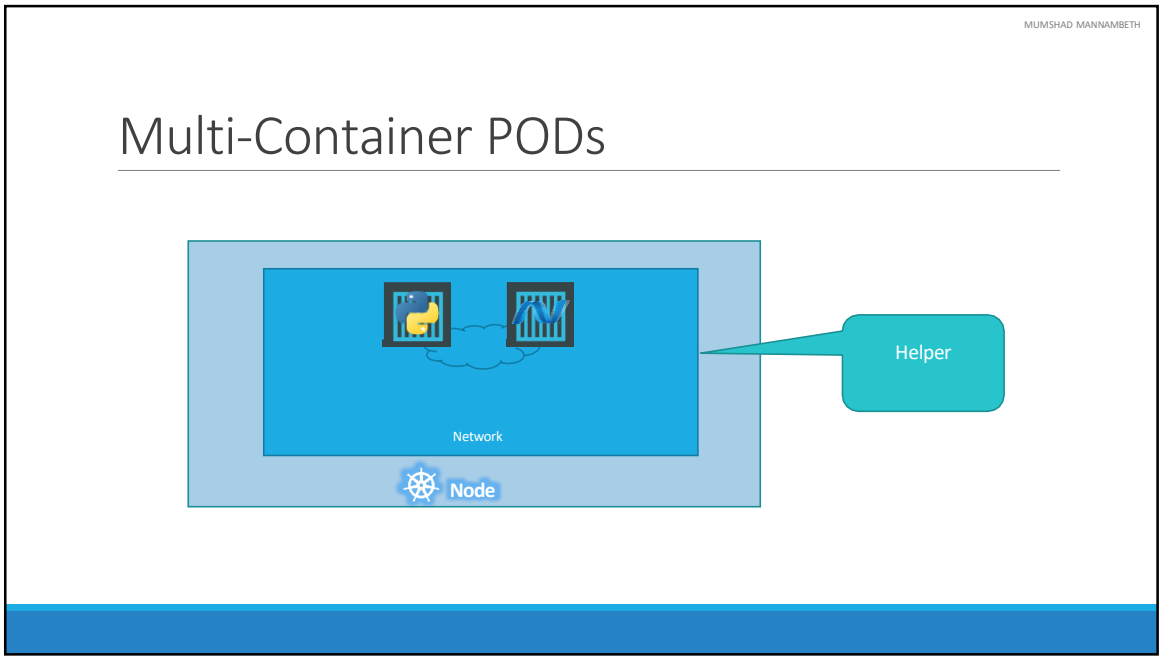
Kubectl concepts

PODS – po - imperative

- With kubernetes our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster. However, kubernetes does not deploy containers directly on the worker nodes. The containers are encapsulated into a Kubernetes object known as PODs.
- A POD is the smallest object, that you can create in kubernetes.



- A single pod usually contains single container of our application in it. A single POD CAN have multiple containers, except for the fact that they are usually not multiple containers of the same kind. 2nd container can be a helper container to provide help to our application container.
- 1 private container named pause container also runs in every pod. But if we want to create 5 more containers of our application we will create 5 more pods. And if all 5 require some helper container also then in such case each pod will have 1 helper and 1 app container. Example-> Helper container can be side car to take app log and put it into splunk.



PODS using yaml example

- We can also create definition file to create and run pod/deployment/services.
- Once yaml is created use below command to run pod.
- Kubectl apply -f samplePod.yaml
- Check if pod is successfully running via kubectl get pods
- Visual studio extension can be used to provide suggestions while creating yaml file.
- Example simplified yaml for PODS->

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx1
  labels:
```

```
    env: production
spec:
  containers:
    - name: nginx
      image: nginx
```

- apiVersion, kind, metadata and spec are root level properties and must be present in all yaml files.
- apiVersion for Pod/Service valid v1 is used and for Deployment and ReplicaSet apps/v1 is used.
- kind can be Pod, Service, Deployment, ReplicaSet etc.
- metadata provides name of kind and also labels. We can add as many key,value pair as possible. Label are user defined.
- Spec has containers array and here we specify all info required to run that container like name and image. For other kind like service, deployment spec content will change. So refer documentation for that.
- kubectl run nginx --image=nginx
- if you edit pod using, kubectl edit pod nginx and then change image, it will automatically update pod with newer image.

Replica controller and Replica set – rs - No

- Controller are brain behind Kubernetes. They are processes that monitor Kubernetes object and respond accordingly.
- Replica controller is one of them. It makes sure that desired number of pods are always up and running. Even if you have a single POD, the replication controller can help by automatically bringing up a new POD when the existing one fails.
- Another reason we need replication controller is to create multiple PODs to share the load across them. When the number of users increase, we can deploy additional POD to balance the load across the two pods by increasing replica count. It will load balance pods irrespective of if all of them are in 1 single node or distributed in multiple nodes.
- Replicas set and Replica controller are identical. There is only 1 major difference, In replica controller it is optional to specify selector tag but on rs it is mandatory. The selector section helps the replica set identify what pods fall under it. But why would you have to specify what PODs fall under it, if you have provided the contents of the pod-definition file itself in the template? It's BECAUSE, replica set can ALSO manage pods that were not created as part of the replica set creation. Say for example, there were pods created BEFORE the creation of the ReplicaSet that match the labels specified in the selector, the replica set will also take THOSE pods into consideration when creating the replicas. So suppose there are 2 pod already running with label 'front'. And in selector if we specifiy matchLabels as front and we want replicas as 3. Then in such case it will only create 1 more pod. And suppose later we try to explicitly create a new pod with same tag. RS will not allow it to create new pod as we have already desired no of pods running.

Replica set using yaml example

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
spec:
  selector:
    matchLabels:
      app: myapppod
  replicas: 2
  template:
    metadata:
      name: nginx1
      labels:
        app: myapppod
    spec:
      containers:
        - name: nginx
          image: nginx
          env:
            - name: password
              value: mysecretpassword
```

- Under the selector section we use the matchLabels filter and provide the same label that we used while creating the pods. This way the replicaset knows which pods to monitor.
- apiVersion for ReplicaSet is apps/v1 and for ReplicationController it's v1.
- Replicaset are used now a days as they provide better control.
- RS is a process that monitor pod health and maintain desired no of pod up every time.
- Template section is mandatory. Here we specify pod definition. In case there is a need of creating new pod to maintain replicas count we need to know what is the image and other detail of pods needed to make it running.
- If we change image inside rs by edit command then to make pods running with new image you need to delete existing all pods. but in deployment it will be handled automatically.

Deployment – deploy - imperative

- It comes higher in hierarchy then pod and rs. Container-> pod->rs->deployment.
- So far in this course we discussed about PODs, which deploy single instances of our application. Each container is encapsulated in POD. Multiple such PODs are deployed using Replication Controllers or Replica Sets. And then comes Deployment which is also a kubernetes object. The deployment provides us with capabilities to upgrade the underlying instances seamlessly using rolling updates, undo changes, and pause and resume changes to deployments. By deployment you can do declarative updates on pods and rs dynamically.
- When deployment is created it will create replica set also and pods also along with deployment object as per deployment definition file configuration.
- If we create 2 deployments say dep1 and dep2 and both uses exact same selector for pods and has same rs as 2. Then in such case how many pods will be created? Total 4.
  https://stackoverflow.com/questions/69755024/two-kubernetes-deployments-with-exactly-the-same-pod-labels

Deployment YAML

- It is same as rs just kind is changed to Deployment.
- So far there hasn't been much of a difference between replicaset and deployments, except for the fact that deployments created a new kubernetes object called deployments also along with rs. We will see how to take advantage of the deployment using the use cases we discussed in the previous point.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
spec:
  selector:
    matchLabels:
      app: myapppod
  replicas: 2
  template:
    metadata:
      name: nginx1
      labels:
        app: myapppod
    spec:
      containers:
        - name: nginx
          image: nginx
          env:
            - name: password
              value: mysecretpassword
```

After adding strategy type-> Recreate:

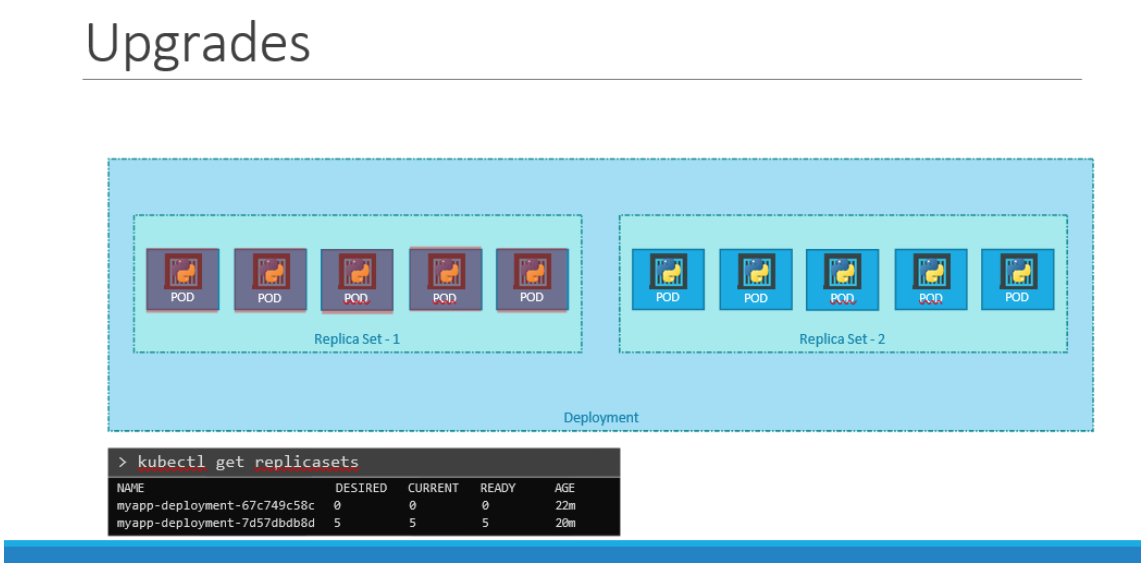
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
spec:
  selector:
    matchLabels:
      app: myapppod
  replicas: 2
  strategy:
    type: Recreate
  template:
    metadata:
      name: nginx1
      labels:
        app: myapppod
    spec:
      containers:
        - name: nginx
          image: nginx
          env:
            - name: password
              value: mysecretpassword
```

Deployment Rollout and version upgrade – no - imperative

kubectl rollout history deploy

- Before we look at how we upgrade our application, let's try to understand Rollouts and Versioning in a deployment. Whenever you create a new deployment or upgrade the images in an existing deployment it triggers a Rollout. **A rollout is the process of gradually deploying or upgrading your application containers**. When you first create a deployment, it triggers a rollout. A new rollout creates a new Deployment revision. Let's call it revision 1. In the future when the application is upgraded – meaning when the container image version is updated to a new one – a new rollout is triggered and a new deployment revision is created named Revision 2. This helps us keep track of the changes made to our deployment and enables us to rollback to a previous version of deployment if necessary.
- If we try to set image but that image is not available in such case since default upgrade strategy is RollingUpdate it will just kill 1 pod and try to run new pod with invalid image and hence system remains available as other pods are still running. Now we can use rollout undo command to remove last deployment upgrade and make it to switch to older stable version.

- You can see the status of your rollout by running the command: kubectl rollout status followed by the name of the deployment.
- To see the revisions and history of rollout run the command kubectl rollout history followed by the deployment name and this will show you the revisions. If used with --revision=4 flag, it will give history of 4 version only.
- How exactly DO you update your deployment? When I say update it could be different things such as updating your application version by updating the version of docker containers used, updating their labels etc. Since we already have a deployment definition file it is easy for us to modify this file. A new rollout is triggered and a new revision of the deployment is created.
- when the Recreate strategy is used the events indicate that the old replicaset was scaled down to 0 first and the new replica set scaled up to 5. However when the RollingUpdate strategy(which is default strategy) is used the old replica set was scaled down one at a time simultaneously scaling up the new replica set one at a time.
- **If we edit rs and update image, it will not automatically update pods. it will do nothing. But if we delete all pods associated with rs. In this case new pods will be created with updated image. In case of deployment it is handled automatically either using rollingupdate or recreate. Rs can scale dynamically but cannot do version upgrade.**
- Rollout and revision is used by deployment by which it can handle dynamic pod creation in case of image upgrade.



- Let's look at how a deployment performs an upgrade under the hoods. When a new deployment is created, say to deploy 5 replicas, it first creates a Replicaset automatically, which in turn creates the number of PODs required to meet the number of replicas. Now suppose we updated image to some latest version. Then It will start removing 1 pod at a time from old rs and start creating upgraded pod in new rs.
- Say for instance once you upgrade your application, you realize something isn't very right. Something's wrong with the new version of build you used to upgrade. So you would like to rollback your update. Kubernetes deployments allow you to rollback to a previous revision. To undo a change run the command kubectl rollout undo followed by the name of the deployment. The deployment will then destroy the PODs in the new replicaset and bring the older ones up in the old replicaset. And your application is back to its older format.
  Note – rollout will not be triggered if you just scale rs count. On image version changes it will create new rollout.:
  Example->

1. before undo command->

```
C:\kush\study\kubernetes\practicek8s>kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-68c5c84d7    0         0         0       5m47s
myapp-deployment-85c966c768   2         2         2       4m31s
```

2. after undo command->

```
C:\kush\study\kubernetes\practicek8s>kubectl rollout undo deployment/myapp-deployment
deployment.apps/myapp-deployment rolled back
```

```
C:\kush\study\kubernetes\practicek8s>kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-68c5c84d7    2         2         2       9m44s
myapp-deployment-85c966c768   0         0         0       8m28s
```

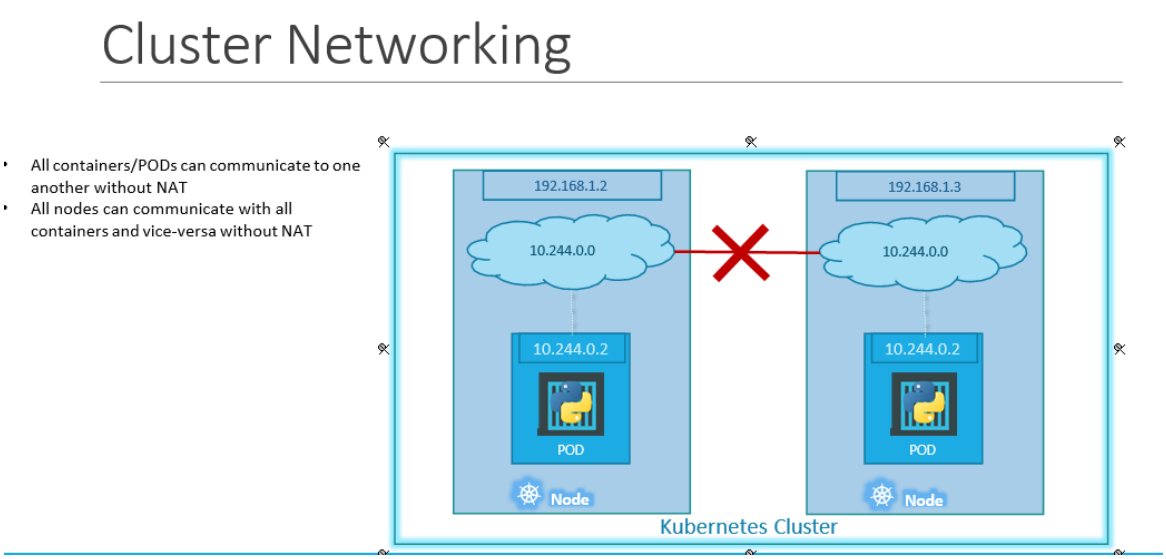
It can be seen old rs is now ready.

```
C:\kush\study\kubernetes\practicek8s>kubectl rollout history deployment/myapp-deployment
deployment.apps/myapp-deployment
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

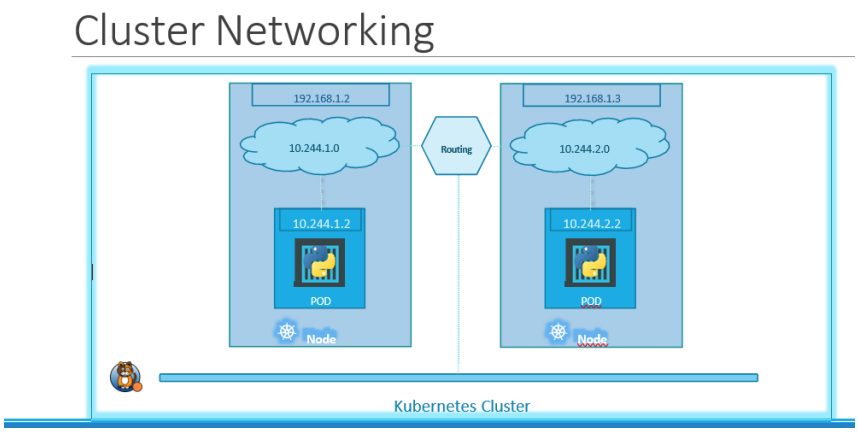
You can also see revision 1 is not present in history as it is same as rev 3 and hence Kubernetes deleted rev 1.

Networking

- Node is either a vm or physical machine and has a IP address by which we can ssh into node.
- If on the single node kubernetes cluster we have created a Single POD. As you know a POD hosts a container. Unlike in the docker world where an IP address is always assigned to a Docker CONTAINER,
  In Kubernetes the IP address is assigned to a POD. Each POD in kubernetes gets its own internal IP Address. In this case its in the range 10.244 series and the IP assigned to the POD is 10.244.0.2. So how is it getting this IP address? When Kubernetes is initially configured, it creates an internal private network for each node with the address 10.244.0.0 and all PODs are attached to it. When you deploy multiple PODs, they all get a separate IP assigned. The PODs can communicate to each other through this IP. But accessing other PODs using this internal IP address MAY not be a good idea as its subject to change when PODs are recreated. We will see BETTER ways to establish communication between PODs in a while.
- So it's all easy and simple to understand when it comes to networking on a single node. But how does it work when you have multiple nodes in a cluster? In this case we have two nodes running kubernetes and they have IP addresses 192.168.1.2 and 192.168.1.3 assigned to them. Note that they are not part of the same cluster yet. Each of them has a single POD deployed. As discussed in the previous slide these pods are attached to an internal network and they have their own IP addresses assigned. HOWEVER, if you look at the network addresses, you can see that they are the same. The two networks have an address 10.244.0.0 and the PODs deployed have the same address too.
- This is NOT going to work well when the nodes are part of the same cluster. The PODs have the same IP addresses assigned to them and that will lead to IP conflicts in the network. Now that's ONE problem. When a kubernetes cluster is SETUP, kubernetes does NOT automatically setup any kind of networking to handle these issues. As a matter of fact, kubernetes expects US to setup networking to meet certain fundamental requirements. Some of these are that all the containers or PODs in a kubernetes cluster MUST be able to communicate with one another without having to configure NAT. All nodes must be able to communicate with containers and all containers must be able to communicate with the nodes in the cluster. Kubernetes expects US to setup a networking solution that meets these criteria.



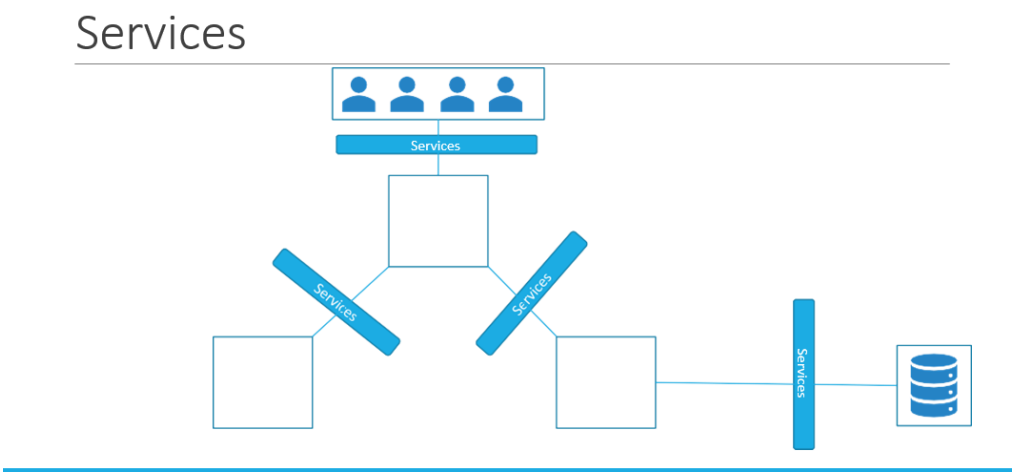
- Fortunately, we don't have to set it up ALL on our own as there are multiple pre-built solutions available. Some of them are the cisco ACI networks, Cilium, Big Cloud Fabric, Flannel, Vmware NSX-t and Calico. Depending on the platform you are deploying your Kubernetes cluster on you may use any of these solutions. For example, if you were setting up a kubernetes cluster from scratch on your own systems, you may use any of these solutions like Calico, Flannel etc. If you were deploying on a Vmware environment NSX-T may be a good option.
- Now suppose with calico network setup. it now manages the networks and Ips in my nodes and assigns a different network address for each network in the nodes. This creates a virtual network of all PODs and nodes where they are all assigned a unique IP Address. And by using simple routing techniques the cluster networking enables communication between the different PODs or Nodes to meet the networking requirements of kubernetes. Thus all PODs can now communicate to each other using the assigned IP addresses.
- We can do pod to pod communication via services.



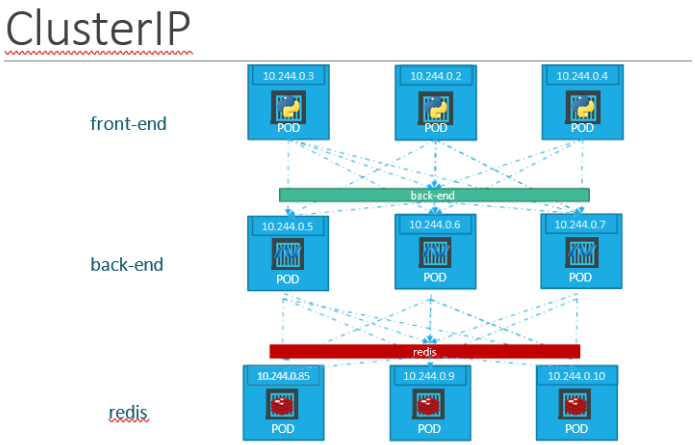
Services – svc - imperative

Kubernetes Services enable communication between various components within and outside of the application. Kubernetes Services helps us connect applications together with other applications or users. For example, our application has groups of PODs running various sections, such as a group for serving front-end load to users, another group running back-end processes, and a third group connecting to an external data source. It is Services that enable connectivity between these groups of PODs. Services enable the front-end application to be made available to users, it helps communication between back-end and front-end PODs,

and helps in establishing connectivity to an external data source. Thus services enable loose coupling between microservices in our application.



Services – ClusterIP



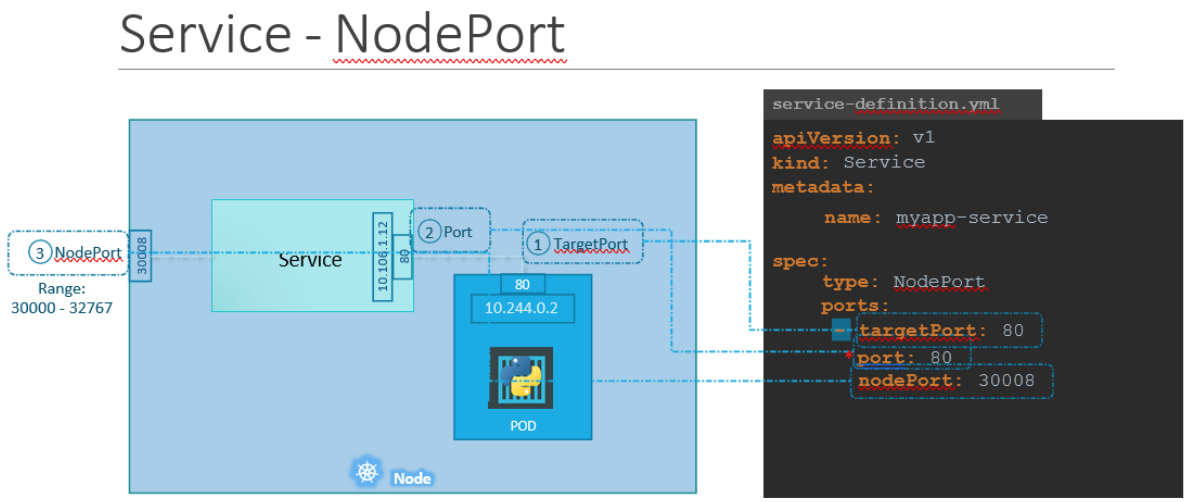
- In it a group of pods in a cluster can communicate with each other's.
- A kubernetes service can help us group these PODs together and provide a single interface to access the PODs in a group. For example a service created for the backend PODs will help group all the backend PODs together and provide a single interface for other PODs to access this service. The requests are forwarded to one of the PODs under the service randomly. Similarly, create additional services for Redis and allow the backend PODs to access the redis system through this service. This enables us to easily and effectively deploy a microservices based application on kubernetes cluster. Each layer can now scale or move as required without impacting communication between the various services. Each service gets an IP and name assigned to it inside the cluster and that is the name that should be used by other PODs to access the service. This type of service is known as ClusterIP.
- Under ports we have a targetPort and port. The target port is the port were the back-end is exposed, which in this case is 80. And the port is were the service is exposed. Which is 80 as well. To link the service to a set of PODs, we use selector.
- This is the default type of service created by Kubernetes. And if we want to access app outside cluster, we need to either use ingress or nodeport or load balancer.
- By default on creating svc object only svc is created

ClusterIP yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
  labels:
    tier: frontend
spec:
  type: ClusterIP
  selector:
    app: myapppod
  ports:
    - port: 80
      targetPort: 80
```

Services – NodePort

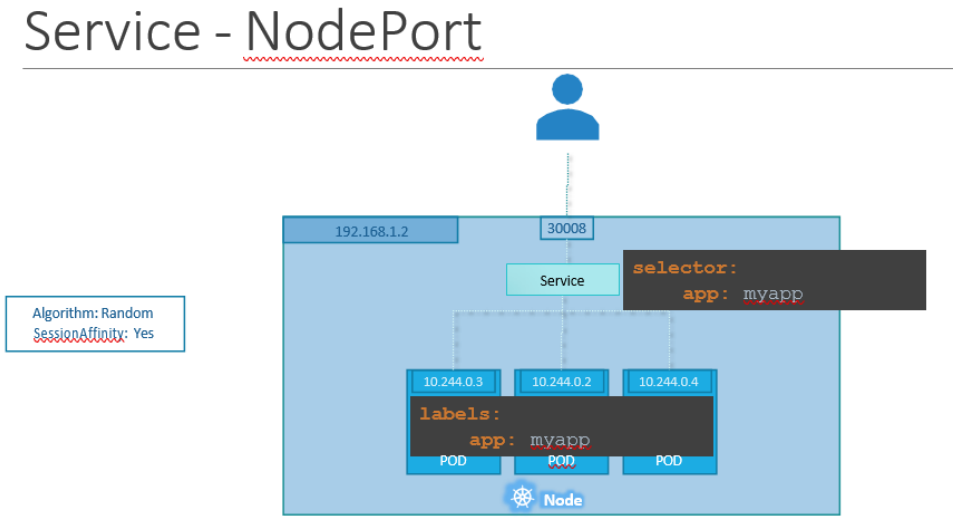
- Let's take a look at one use case of Services. So we deployed our POD having a web application running on it. How do WE as an external user access the web page? First of all, lets look at the existing setup. The Kubernetes Node has an IP address and that is 192.168.1.2. My laptop is on the same network as well, so it has an IP address 192.168.1.10. The internal POD network is in the range. 10.244.0.0 and the POD has an IP 10.244.0.2. Clearly, I cannot ping or access the POD at address 10.244.0.2 as its in a separate network. So what are the options to see the webpage?
- First, if we were to SSH into the kubernetes node at 192.168.1.2, from the node, we would be able to access the POD's webpage by doing a curl or if the node has a GUI, we could fire up a browser and see the webpage in a browser following the address http://10.244.0.2. But this is from inside the kubernetes Node and that's not what I really want. I want to be able to access the web server from my own laptop without having to SSH into the node and simply by accessing the IP of the kubernetes node. So we need something in the middle to help us map requests to the node from our laptop through the node to the POD running the web container.
- That is where the kubernetes service comes into play. The kubernetes service is an object just like PODs, Replicaset or Deployments that we worked with before. One of its use case is to listen to a port on the Node and forward requests on that port to a port on the POD running the web application. This type of service is known as a NodePort service because the service listens to a port on the Node and forwards requests to target PODs.
- When we create nodeport service object Kubernetes creates nodeport and cluster ip service both.



- Let's take a closer look at the Service. If you look at it, there are 3 ports involved. The port on the POD were the actual web server is running is port 80. And it is referred to as the targetPort, because that is where the service forwards the requests to. The second port is the port on the service itself. It is simply referred to as the port.
- Remember, these terms are from the viewpoint of the service. The service is in fact like a virtual server inside the node. Inside the cluster it has its own IP address. And that IP address is called the Cluster-IP of the

service. And finally we have the port on the Node itself which we use to access the web server externally. And that is known as the NodePort. As you can see it is 30008. That is because NodePorts can only be in a valid range which is from 30000 to 32767.

- If you don't provide a targetPort it is assumed to be the same as port and if you don't provide a nodePort a free port in the valid range between 30000 and 32767 is automatically allocated. Also note that ports is an array. You can have multiple such port mappings within a single service.
- Here selector part is also needed as it will use only those pod to forward request to whose label matches.
- In a production environment you have multiple instances of your web application running for high-availability and load balancing purposes. In this case we have multiple similar PODs running our web application. They all have the same labels with a key app set to value myapp. The same label is used as a selector during the creation of the service. So when the service is created, it looks for matching PODs with the labels and finds 3 of them. The service then automatically selects all the 3 PODs as endpoints to forward the external requests coming from the user. You don't have to do any additional configuration to make this happen. And if you are wondering what algorithm it uses to balance load, it uses a random algorithm. Thus the service acts as a built-in load balancer to distribute load across different PODs.



- And finally, lets look at what happens when the PODs are distributed across multiple nodes. In this case we have the web application on PODs on separate nodes in the cluster. When we create a service , without us having to do ANY kind of additional configuration, kubernetes creates a service that spans across all the nodes in the cluster and maps the target port to the SAME NodePort on all the nodes in the cluster. This way you can access your application using the IP of any node in the cluster and using the same port number which in this case is 30008.
- Nodeport service is not recommended for production as you are directly exposing port of nodes to outside world which is not safe. Instead use load balancer or ingress on that which will access exposed nodeport only. For POC or quick demo or development time we can use nodePort.

NodePort yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
  labels:
    tier: frontend
spec:
  type: NodePort
  selector:
    app: myapppod
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30090
```

Services – LoadBalancer

- The third type is a LoadBalancer, where it provisions a external load balancer provided by cloud providers for our service. In this case nodeport and clusterIP service is also created. Where clusterip is used for internal communication between group of pods. Nodeport exposes each node port but instead of exposing to external world. Only load balancer can access this nodeport. And via loadbalancer public ip external world can access the system.
- For example->
- We have a 3 node cluster with Ips 192.168.1.2,3 and 4. Our application is two tier, there is a database service and a front-end web service for users to access the application. The default service type –known as ClusterIP – makes a database service available internally within the kubernetes cluster for other applications to consume.
- python based web front-end application connects to the backend using Service created for the DB service. To expose the application to the end users, we create another service of type NodePort. Creating a service of type NodePort exposes the application on a high end port of the Node and the users can access the application at any IP of my nodes with the port 30008. Nodeport service is created for a group of frontend pods.
- Now, what IP do you give your end users to access your application? You cannot give them all three and let them choose one of their own. What end users really want is a single URL to access the application. For this, you will be required to setup a separate Load Balancer VM in your environment. In this case I deploy a new VM for load balancer purposes and configure it to forward requests that come to it to any of the Ips of the Kubernetes nodes. I will then configure my organizations DNS to point to this load balancer when a user hosts http://myapp.com. Now setting up that load balancer by myself is a tedious task, and I might have to do that in my local or on-prem environment. However, if I happen to be on a supported CloudPlatform, like Google Cloud Platform, I could leverage the native load balancing functionalities of the cloud platform to set this up. Again, you don't have to set that up manually, Kubernetes sets it up for you. Kubernetes has built-in integration with supported cloud platforms.

LoadBalancer yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-service
  labels:
    tier: frontend
spec:
  type: LoadBalancer
  selector:
    app: myapppod
  ports:
    - port: 80
      targetPort: 80
```

- kubectl create service nodeport nginx --tcp=80:80 --dry-run=client -o yaml > service.yaml

  here you need to edit and set nodePort explicitly , otherwise default nodeport will be assigned.

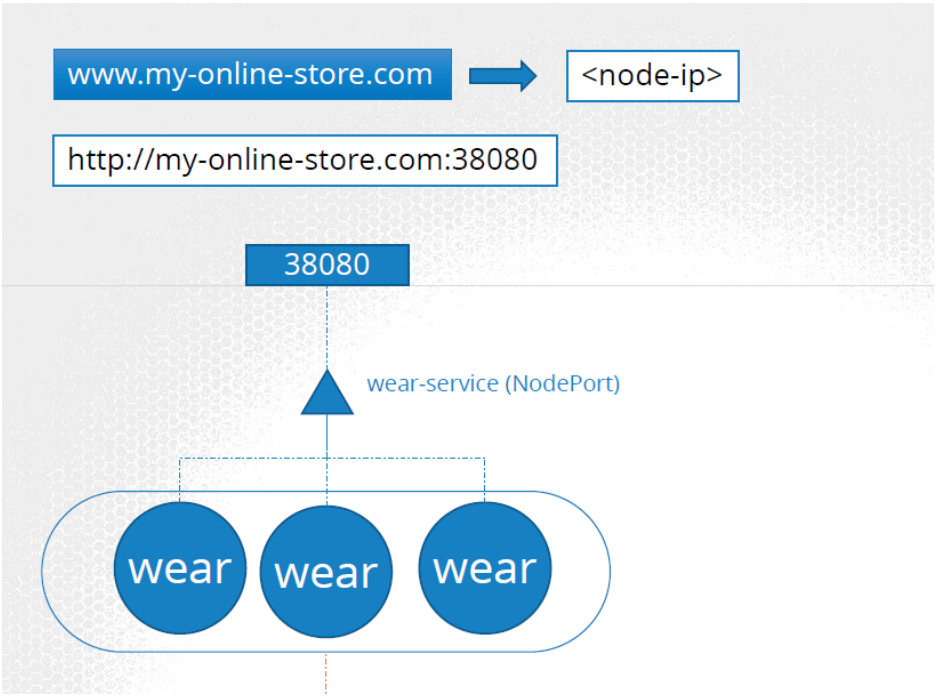
- kubectl run httpd --image=httpd:alpine --port=80 --expose

Ingress Networking – ing - imperative

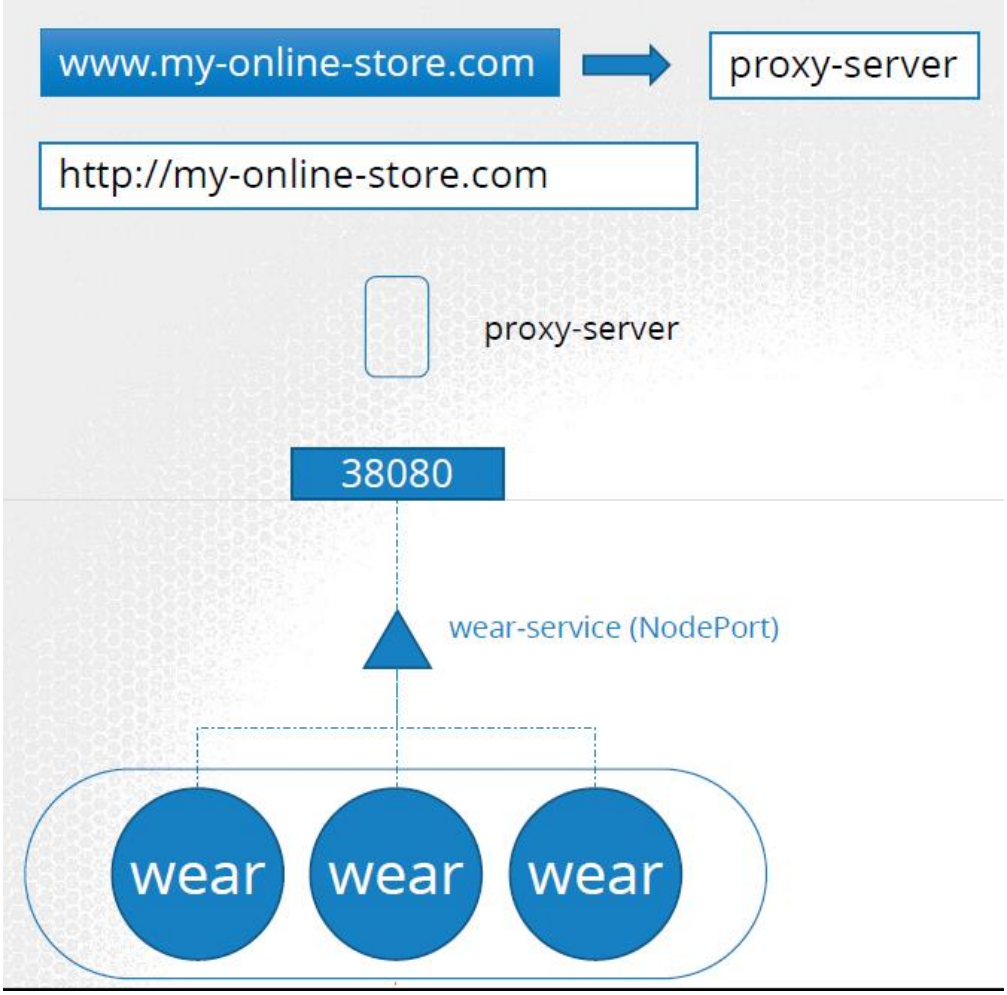
- Now, in k8s version 1.20+ we can create **an Ingress resource** from the imperative way like this:-
  Format - kubectl create ingress <ingress-resource-name> --rule="host/path=service:port"
  Example - kubectl create ingress ingress-test --rule="wear.my-online-store.com/wear*=wear-service:80"

- https://idemia.udemy.com/course/certified-kubernetes-application-developer/learn/lecture/13302764#questions
- Suppose You are deploying an application on Kubernetes for a company that has an online store selling products. Your application would be available at say my-online-store.com.

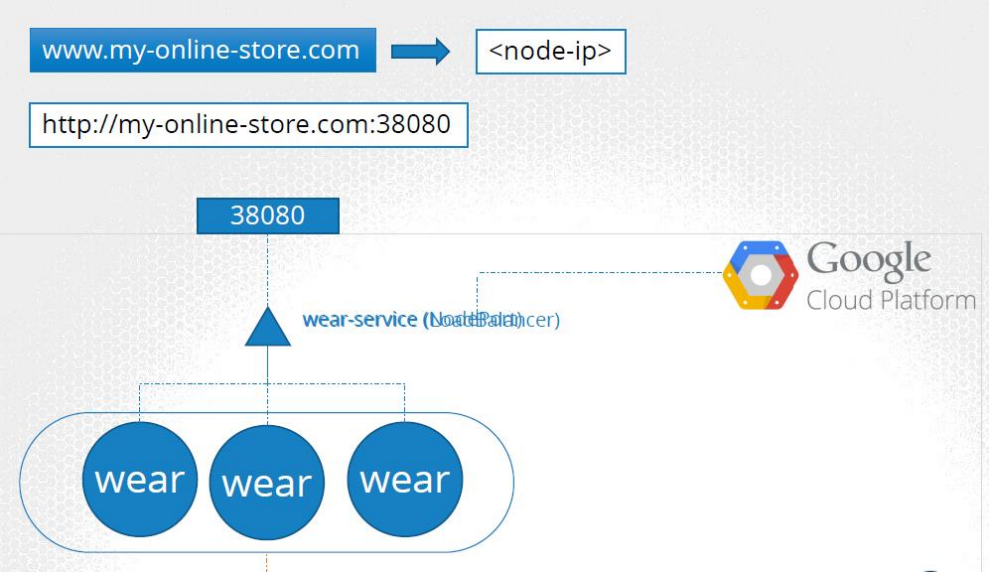
- You build the application into a Docker Image and deploy it on the kubernetes cluster as a POD in a Deployment. Your application needs a database so you deploy a MySQL database as a POD and create a service of type ClusterIP called mysql-service to make it accessible to your application. Your application is now working. To make the application accessible to the outside world, you create another service, this time of type NodePort and make your application available on a high-port on the nodes in the cluster. In this example a port 38080 is allocated for the service. The users can now access your application using the URL: http, colon, slash slashIP of any of your nodes followed by port 38080. That setup works and users are able to access the application.
- However, if you have deployed a production grade application before, you know that there are many more things involved in addition to simply splitting the traffic between the PODs.



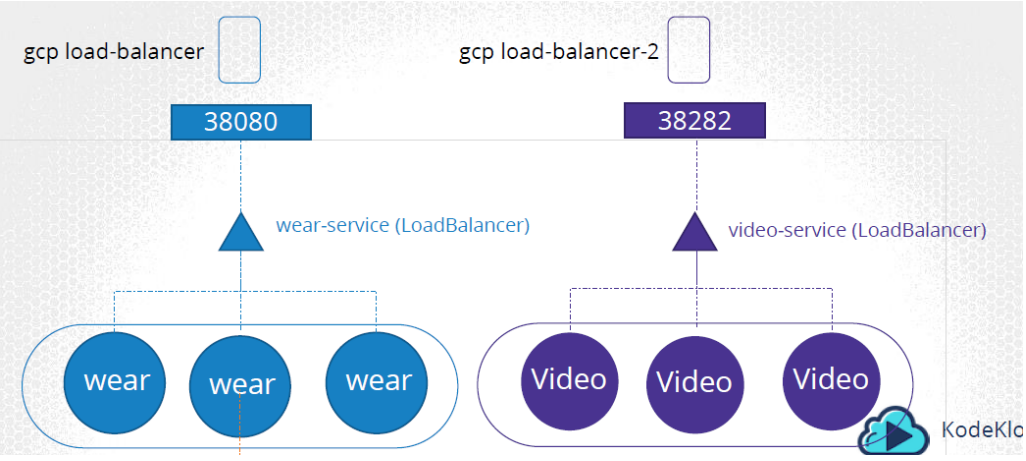
- For example, we do not want the users to have to type in IP address everytime. So you configure your DNS server to point to the IP of the nodes. Your users can now access your application using the URL http://my-online-store.com:38080.



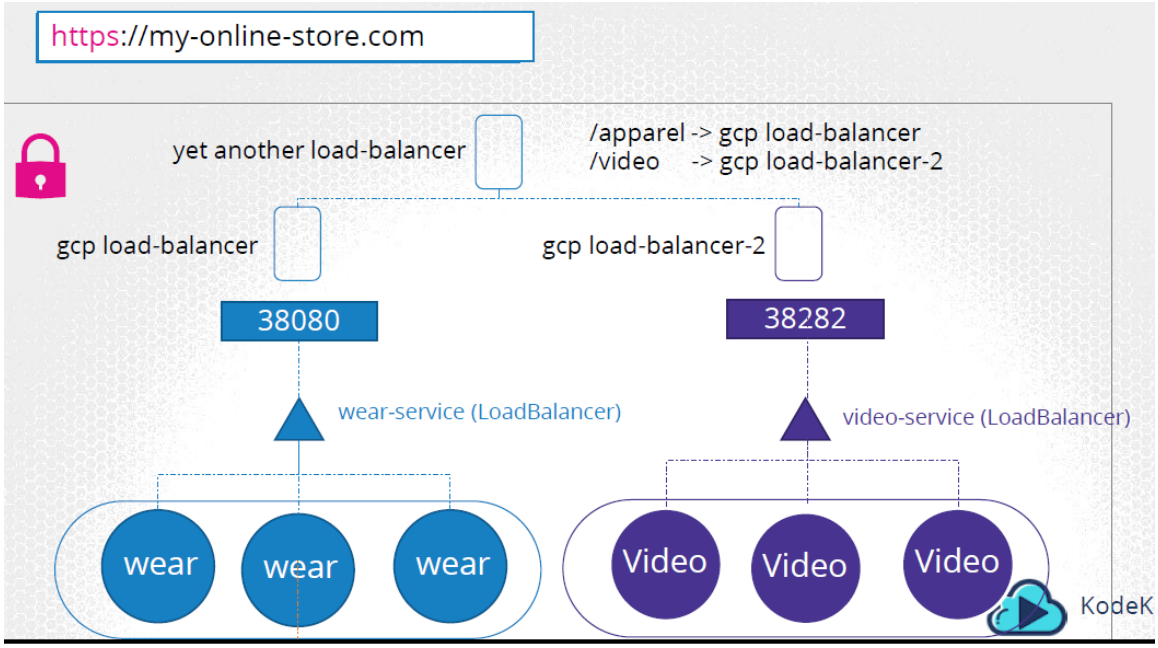
- Now, you don't want your users to have to remember port number either. However, Service NodePorts can only allocate high numbered ports which are greater than 30,000. So you then bring in an additional layer between the DNS server and your cluster, like a proxy server, that proxies requests on port 80 to port 38080 on your nodes. You then point your DNS to this server, and users can now access your application by simply visiting my-online-store.com. Now, this is if your application is hosted on-prem in your datacenter.
- In that case, instead of creating a service of type NodePort for your wear application, you could set it to type LoadBalancer. When you do that Kubernetes would still do everything that it has to do for a NodePort, which is to provision a high port for the service, but in addition to that kubernetes also sends a request to Google Cloud Platform to provision a native load balancer for this service. GCP would then automatically deploy a LoadBalancer configured to route traffic to the service ports on all the nodes and return its information to kubernetes. The LoadBalancer has an external IP that can be provided to users to access the application. In this case we set the DNS to point to this IP and users access the application using the URL.



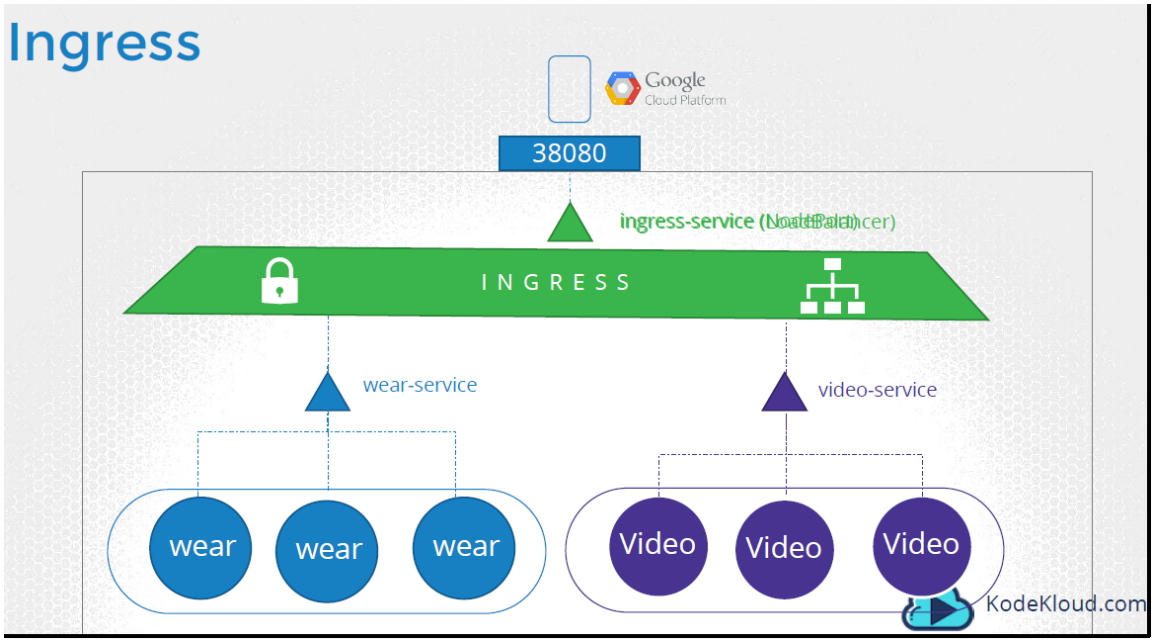
- On receiving the request, GCP would then automatically deploy a LoadBalancer configured to route traffic to the service ports on all the nodes and return its information to kubernetes. The LoadBalancer has an external IP that can be provided to users to access the application. In this case we set the DNS to point to this IP and users access the application using the URL my-online-store.com. Perfect!!
- Your companies business grows and you now have new services for your customers. For example, a video streaming service. You want your users to be able to access your new video streaming service by going to my-online-store.com/watch. You'd like to make your old application accessible at my-online.store.com/wear.
- Your developers developed the new video streaming application as a completely different application, as it has nothing to do with the existing one. However to share the cluster resources, you deploy the new application as a separate deployment within the same cluster. You create a service called video-service of type LoadBalancer. Kubernetes provisions port 38282 for this service and also provisions a cloud native LoadBalancer. The new load balancer has a new IP. Remember you must pay for each of these load balancers and having many such load balancers can inversely affect your cloud bill.



- So how do you direct traffic between each of these load balancers based on URL?
- You need yet another proxy or load balancer that can re-direct traffic based on URLs to the different services. Every time you introduce a new service you have to re-configure the load balancer.
- And finally you also need to enable SSL for your applications, so your users can access your application using https. Where do you configure that? Now that's a lot of different configuration and all of these becomes difficult to manage when your application scales. It requires involving different individuals in different teams. You need to configure your firewall rules for each new service. And its expensive as well, as for each service a new cloud native load balancer will be provisioned.



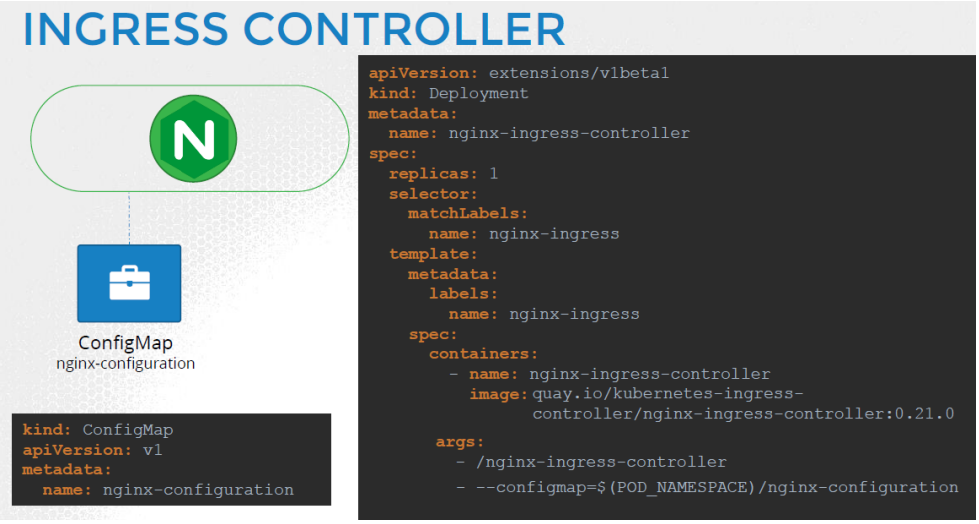
- Wouldn't it be nice if you could manage all of that within the Kubernetes cluster, and have all that configuration as just another kubernetes definition file, that lives along with the rest of your application deployment files?
- That's where Ingress comes into play. Ingress helps your users access your application using a single Externally accessible URL, that you can configure to route to different services within your cluster based on the URL path, at the same time terminate TLS.
- Simply put, think of ingress as a layer 7 load balancer built-in to the kubernetes cluster that can be configured using native kubernetes primitives just like any other object in kubernetes.
- Now remember, even with Ingress you still need to expose it to make it accessible outside the cluster. So you still have to either publish it as a NodePort or with a Cloud Native LoadBalancer. But that is just a one time thing. Going forward you are going to perform all your load balancing, Auth, SSL and URL based routing configurations on the Ingress controller. With addition of new service into system just configure ingress to manage routes. And external load balancer count will remain 1.



- So how does it work? What is it? Where is it? How can you see it? How can you configure it? So how does it load balance? How does it implement SSL?
- Without ingress, how would YOU do all of these? I would use a reverse-proxy or a load balancing solution like NGINX or HAProxyor Traefik. I would deploy them on my kubernetes cluster and configure them to route traffic to other services. The configuration involves defining URL Routes, SSL certificates etc.

- Ingress is implemented by Kubernetes in the same way. You first deploy a supported solution, which happens to be any of these nginx, isitio etc and then specify a set of rules to configure Ingress. The solution you deploy is called as an Ingress Controller. And the set of rules you configure is called as Ingress Resources. Ingress resources are created using definition files like the ones we used to create PODs, Deployments and services earlier in this course.
- Now remember a kubernetes cluster does NOT come with an Ingress Controller by default. So if you simply create ingress resources and expect them to work, they wont.

- Let's look at each of these in a bit more detail now. As I mentioned you do not have an Ingress Controller on Kubernetes by default. So you MUST deploy one. What do you deploy? There are a number of solutions available for Ingress, a few of them being GCE -which is Googles Layer 7 HTTP Load Balancer. NGINX, Contour, HAPROXY, TRAFIK and Istio. Out of this, GCE and NGINX are currently being supported and maintained by the Kubernetes project.



- And in this lecture, we will use NGINX as an example. An NGINX Controller is deployed as just another deployment in Kubernetes. So we start with a deployment file definition, named nginx-ingress-controller. With 1 replica and a simple pod definition template. We will label it nginx-ingress and the image used is nginx-ingress-controller with the right version. This is a special build of NGINX built specifically to be used as an ingress controller in kubernetes. So it has its own requirements. Within the image the nginx program is stored at location /nginx-ingress-controller. So you must pass that as the command to start the nginx-service. If you have worked with NGINX before, you know that it has a set of configuration options such as the path to store the logs, keep-alive threshold, ssl settings, session timeout etc. In order to decouple these configuration data from the nginx-controller image, you must create a ConfigMap object and pass that in. Now remember the ConfigMap object need not have any entries at this point. A blank object will do. But creating one makes it easy for you to modify a configuration setting in the future. You will just have to add it in to this ConfigMap.

- You must also pass in two environment variables that carry the POD's name and namespace it is deployed to. The nginxservice requires these to read the configuration data from within the POD. And finally specify the ports used by the ingress controller.
- We then need a service to expose the ingress controller to the external world. So we create a service of type NodePort with the nginx-ingress label selector to link the service to the deployment.

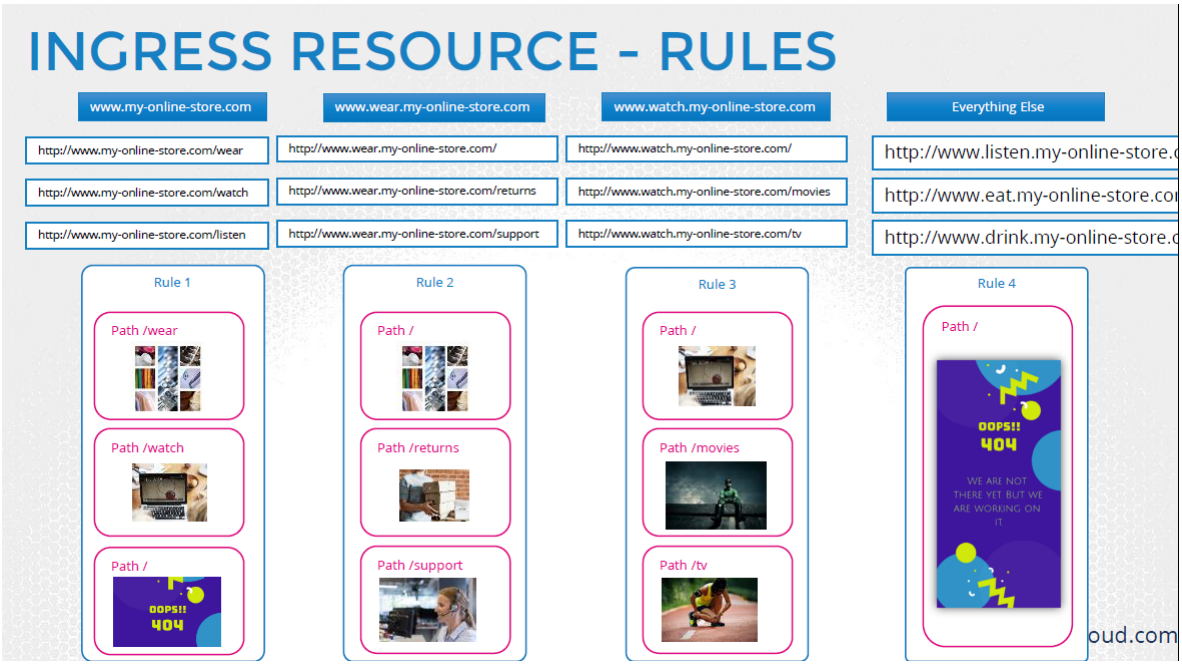
```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

- At last we need serviceaccount with right set of permissions to be able to work correctly and monitor and manages ingress resources. So we create service account with correct roles and roles bindings. And that service account should be there in ingress-controller deployment definition file.

```
root@controlplane:~# kubectl create sa ingress-serviceaccount -n ingress-space
serviceaccount/ingress-serviceaccount created
```

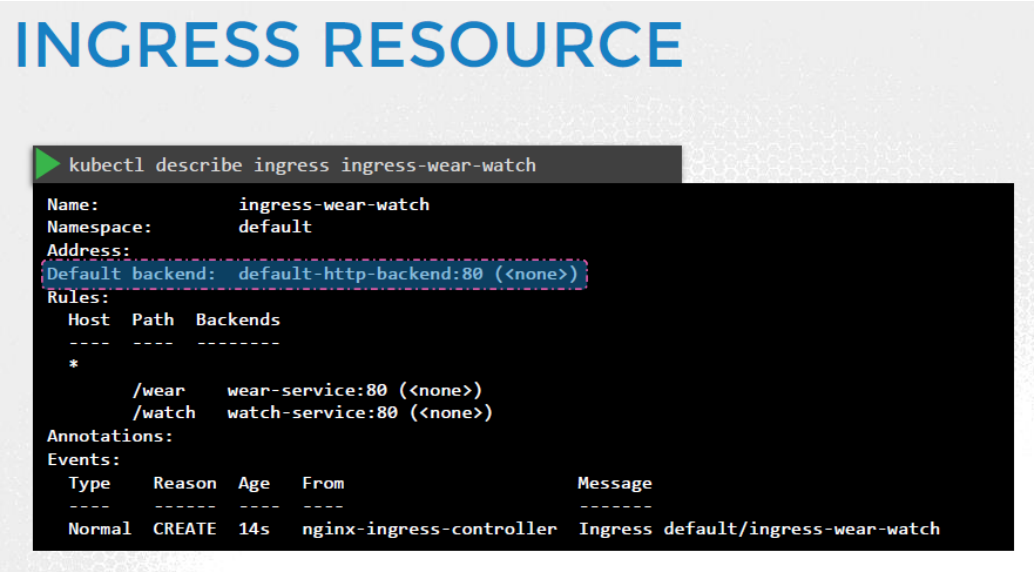
```yaml
kind: Deployment
metadata:
  name: ingress-controller
  namespace: ingress-space
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      serviceAccountName: ingress-serviceaccount
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.21.0
          args:
            - /nginx-ingress-controller
            - --configmap=$(POD_NAMESPACE)/nginx-configuration
            - --default-backend-service=app-space/default-http-backend
```

- You use rules, when you want to route traffic based on different conditions. For example, you create one rule for traffic originating from each domain or hostname. That means when users reach your cluster using the domain name, my-online-store.com, you can handle that traffic using rule1. When users reach your cluster using domain name wear.my-online-store.com, you can handle that traffic using a separate Rule2. Use Rule3 to handle traffic from watch.my-online-store.com. And say use a 4th rule to handle everything else. And you would achieve this, by adding multiple DNS entries, all of which would point to the same Ingress controller service on your kubernetes cluster.



```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        pathType: Prefix
        backend:
          service:
            name: wear-service
            port:
              number: 80
      - path: /watch
        pathType: Prefix
        backend:
          service:
            name: watch-service
            port:
              number: 80
```

- Now, let's look at how we configure ingress resources in Kubernetes. We will start where we left off. We start with a similar definition file. This time under spec, we start with a set of rules. Now our requirement here is to handle all traffic coming to my-online-store.com and route them based on the URL path. So we just need a single Rule for this, since we are only handling traffic to a single domain name, which is my-online-store.com in this case. Under rules we have one item, which is an http rule in which we specify different paths. So paths is an array of multiple items. One path for each url. Then we move the backend we used in the first example under the first path. The backend specification remains the same, it has a servicename and serviceport. Similarly, we create a similar backend entry to the second URL path, for the watch-service to route all traffic to the /watch url to the watch-service. Create the ingress resource using the kubectl create ingress command.



- describe ingress command. You now see two backend URLs under the rules, and the backend service they are pointing to. Just as we created it.
- Now, If you look closely in the output of this command, you see that there is something about a Default-backend. Hmmm. What might that be?
- If a user tries to access a URL that does not match any of these rules, then the user is directed to the service specified as the default backend. In this case it happens to be a service named default-http-backend. So you must remember to deploy such a service.
- Case where domain name is different. Now that we have two domain names, we create two rules. One for each domain. To split traffic by domain name, we use the host field. The host field in each rule matches the specified value with the domain name used in the request URL and routes traffic to the appropriate backend. In this case note that we only have a single backend path for each rule. Which is fine. All traffic from these

domain names will be routed to the appropriate backend irrespective of the URL Path used. You can still have multiple path specifications in each of these to handle different URL paths.

Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: watch-service
          servicePort: 80
```


- **It is not necessary that all resources related to ingress should be in same namespace**. Example-> configMap, serviceAccount, role, rolebindings, ingress-controller deployment, nodePort/loadBalancer service can be in same ingress-space namespace. But ingress resource should be in app-space where app services are there as ingress resource file should have direct access to svc.

```
root@controlplane:~# kubectl get deployments --all-namespaces
NAMESPACE     NAME              READY   UP-TO-DATE   AVAILABLE   AGE
app-space     default-backend   1/1     1            1           35s
app-space     webapp-video      1/1     1            1           35s
app-space     webapp-wear       1/1     1            1           35s
kube-system   coredns           2/2     2            2           28m
root@controlplane:~# kubectl create ns ingress-space
namespace/ingress-space created
root@controlplane:~# kubectl create configmap nginx-configuration -n ingress-space
configmap/nginx-configuration created
root@controlplane:~# kubectl create sa ingress-serviceaccount -n ingress-space
serviceaccount/ingress-serviceaccount created
```

```
root@controlplane:~# kubectl get roles -n ingress-space
NAME           CREATED AT
ingress-role   2022-05-08T06:04:46Z
root@controlplane:~# kubectl get rolebindings -n ingress-space
NAME                  ROLE                 AGE
ingress-role-binding  Role/ingress-role   103s
```

```
root@controlplane:~# kubectl create svc nodeport ingress --tcp=80:80 -n ingress-space --dry-run=
client -o yaml > svc1.yaml
root@controlplane:~# vi svc1.yaml
root@controlplane:~# kubectl create -f svc1.yaml
service/ingress created
```

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    app: ingress
  name: ingress
  namespace: ingress-space
spec:
  ports:
  - name: 80-80
    port: 80
    protocol: TCP
    targetPort: 80
    nodePort: 30080
  selector:
    name: nginx-ingress
  type: NodePort
```

```
root@controlplane:~# kubectl create ingress test-ingress -n app-space --rule="host/wear*=wear-service:8080" --dry-run=client -o yaml > ing.ya
ml
root@controlplane:~# vi ing.yaml
```

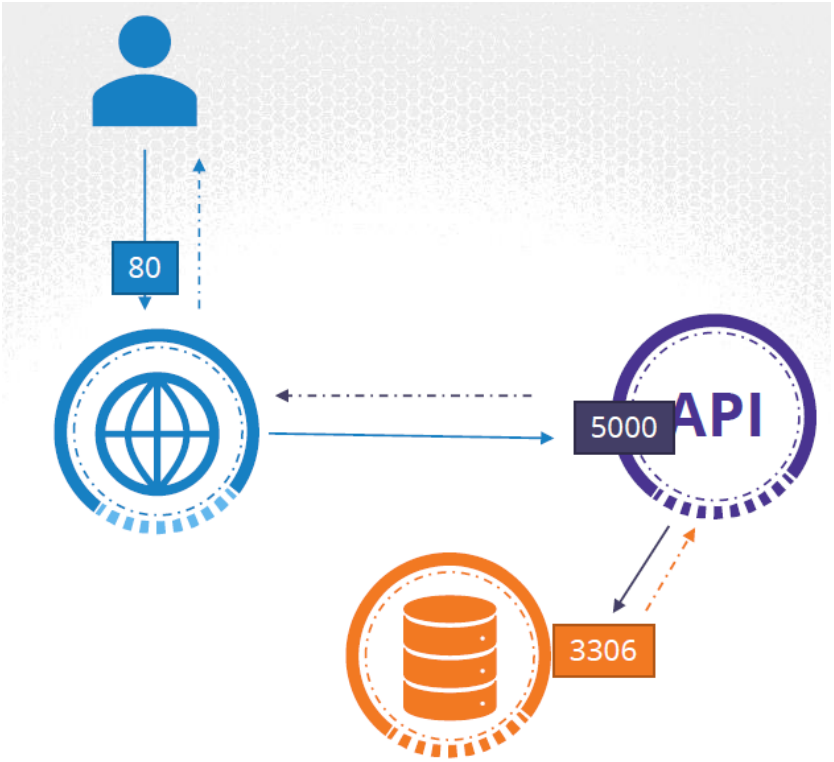
In vi remove host as we are not dealing with domain-name specification. And add one more entry of video-service for path /watch.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  namespace: app-space
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - http:
      paths:
      - backend:
          service:
            name: wear-service
            port:
              number: 8080
        path: /wear
        pathType: Prefix
      - backend:
          service:
            name: video-service
            port:
              number: 8080
        path: /watch
        pathType: Prefix
```

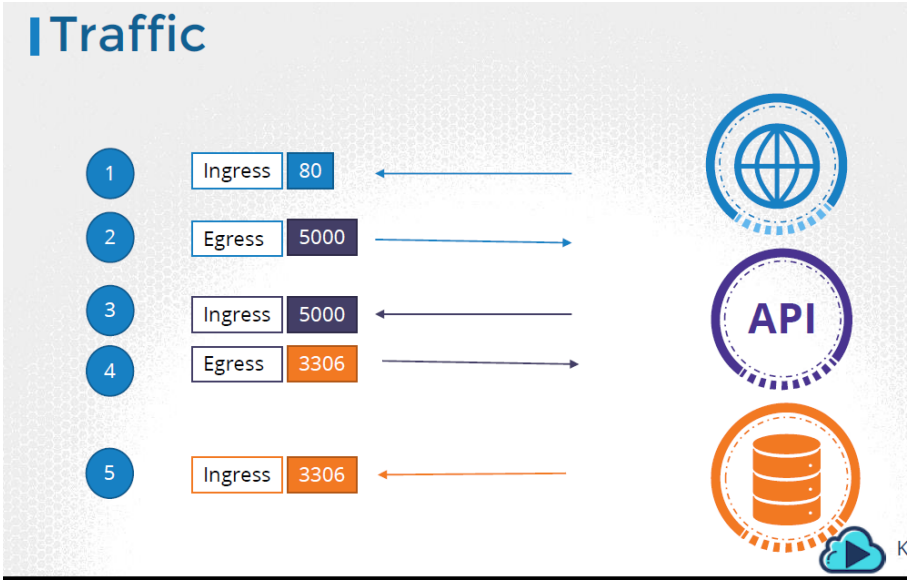
- Different ingress controllers have different options that can be used to customise the way it works. NGINX Ingress controller has many options that can be seen on documentation. rewrite-target is one of them.

Network policy -netpol - No

- Documentation -> https://kubernetes.io/docs/concepts/services-networking/network-policies/
- The user sends in a request to the web server at port 80. The web server then sends a request to the API server at port 5000 in the backend. The API server then fetches data from the database server at port 3306. And then sends the data back to the user.



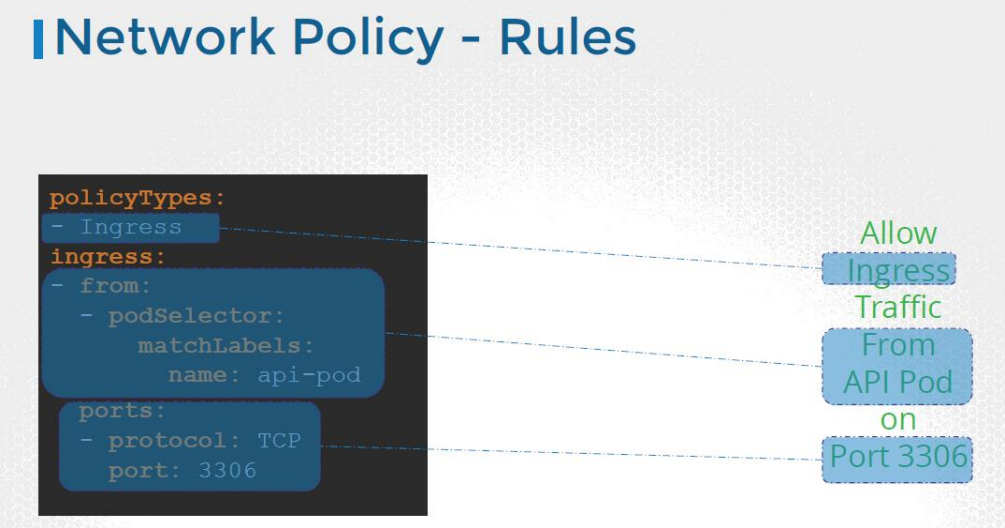
- So there are two types of traffic here. Ingress and Egress. For example, for a web server, the incoming traffic from the users is an Ingress Traffic. And the outgoing requests to the app server is Egress traffic. When you define ingress and egress, remember you are only looking at the direction in which the traffic originated. The response back to the user, denoted by the dotted lines do not really matter. Similarly, in case of the backend API server, it receives ingress traffic from the web server on port 80 and has egress traffic to port 3306 to the database server. And from the database servers perspective, it receives Ingress traffic on port 3306 from api server.
- By using network policy you can define such ingress and egress rules to pod. By default every pod is allowed to accept all ingress and egress traffic.



- all pods by default reach each other using the IPs or pod names or services configured for that purpose. Kubernetes is configured by default with an “All Allow” rule that allows traffic from any pod to any other pod or services. We create services to enable communication between the PODs as well as to the end user. For example if we have create a api service. So now using api service name any pod can connect to it. But what we want is restrictions like only incoming request from web server on port 5000 should be allowed on api service and similarly only db call on port 3306 can be made by api service. So such type of rules is possible via network policy.
- A Network policy is another object in the kubernetes namespace. Just like PODs, ReplicaSets or Services. You apply a network policy on selected pods. or Services to Pods. We label the Pod and use the same labels

on the pod selector field in the network policy. You can link a network policy to one or more pods. You can define rules within the network policy. Once policy is created, it blocks all other traffic to the Pod and only allows traffic that matches the specified rule. Again, this is only applicable to the Pod on which the network policy is applied.

- For example we want db service to allow ingress traffic from api service only on port 3306 only. In this case I would say, only allow Ingress Traffic from the API Pod on Port 3306.



- Remember that Network Policies are enforced by the Network Solution implemented on the Kubernetes Cluster. And not all network solutions support network policies. A few of them that are supported are kube-router, Calico, Romana and Weave-net. If you used Flannel as the networking solution, it does not support network policies as of this recording.
- If we did not specify namespaceSelector inside from.podSelector tag then it will allow ingress traffic to db on port 3306 from all api-pod from all namespaces if exists. We can restrict it to namespace by specify namespace selector which matches the label that we have assigned to desired namespace.
- We can also allow ingress and egress traffic to pod from outside Kubernetes cluster also. Like host machine which want to take backup of db data from our pod. In such case add ipBlock tag inside from tag parallel to podSelector tag and specify ip in cidr. no of tags mentioned parallel to podSelector specifies no of rules applied on that pod. They behave like OR operator.
- Below command can be used to view already created network policy->
  kubectl get netpol
  kubect get networkpolicy
- There is no imperative command shortcut of creating netpol
- Example ->
- Create a network policy to allow traffic from the “db” application only to the “payroll-service” and “db-service”.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: internal-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      name: db
  policyTypes:
    - Egress
    - Ingress
  ingress:
    - {}
  egress:
    - to:
        - podSelector:
            matchLabels:
              name: payroll
      ports:
        - protocol: TCP
          port: 8080
    - to:
        - podSelector:
            matchLabels:
              name: mysql
      ports:
        - protocol: TCP
          port: 3306
```

Above spec -> name=mysql and name=payroll are labels of pods on which db pod want to connect.

```
root@controlplane:~# kubectl create  -f net2.yaml
networkpolicy.networking.k8s.io/internal-policy created
root@controlplane:~# kubectl get networkpolicy
NAME              POD-SELECTOR   AGE
internal-policy   name=db        17s
payroll-policy    name=payroll   36m
root@controlplane:~# kubectl describe networkpolicy internal-policy
Name:         internal-policy
Namespace:    default
Created on:   2022-05-09 07:02:04 +0000 UTC
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     name=db
  Not affecting ingress traffic
  Allowing egress traffic:
    To Port: 8080/TCP
    To:
      PodSelector: name=payroll
    ----------
    To Port: 3306/TCP
    To:
      PodSelector: name=mysql
  Policy Types: Egress
```

- Example 2-> Create a network policy to allow traffic to “payroll” application, only from internal application.

  Here payroll pod can accept request port 8080 from pod named internal traffic only.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: payroll-policy
spec:
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: internal
    ports:
    - port: 8080
      protocol: TCP
  podSelector:
    matchLabels:
      name: payroll
  policyTypes:
  - Ingress
```

Namespaces

- We can arrange our k8s resources in a group named namespaces. By this with-in single cluster we can create as many namespace as possible. And each namespace resources are isolated from each other. By default if we did not specify namespace in our command, k8s will assign that resource to default namespace.
- kube-system: it is already created by k8s. and it contains deployments, pods, services, rs, daemonset etc that are used by k8s only for internal purpose. Like they are used by network services, dns server etc.
- kube-public : it is also created by k8s when cluster is created just like kube-system. Here we add resources that should be available to all users.
- 1 benefit of namespace is that by isolation somebody cannot accidently deletes resources outside there namespace.
- To list down pods under all namespace use. kubectl get pod --all-namespaces OR kubectl get pod -A
- In each namespace you can define custom policy that are applicable to resources with-in that namespace.
- In each namespace you can also specify quota like max cpu,ram to limit the usage.
- With-in a namespace you can directly access service in a pod by using service-name. But if a pod in abc namespace want to access service under xyz namespace. Then below dns name should be used.
  Example 1-> within same namespace.
  mysql.connect(“db-service”)
  Example 2-> access service present in dev namespace.
  mysql.connect(“db-service.dev.svc.cluster.local”)
- It is working because when service is created its dns entry is added in above format.
- Dns name contains 4 sections -> first is service name, second namespace, third service and fourth is domain which is cluster.local.

Create Namespace – ns - imperative

- Two ways to create namespace is ->

Create Namespace

namespace-definition.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: team1

spec:
```

```
kubectl create –f namespace-definition.yml
namespace “team1” created
```

Second ->
kubectl create ns name

```
C:\Users\kush.gupta\Documents>kubectl create ns dummytest
namespace/dummytest created

C:\Users\kush.gupta\Documents>kubectl run nginx --image=nginx -n dummytest
pod/nginx created

C:\Users\kush.gupta\Documents>kubectl get pod -n dummytest
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          15s
```

- When we create pod it will be created in default namespace. To change that either specify namespace inside pod.yaml under metadata tag. Or specify it in kubectl command itself.

```
C:\Users\kush.gupta\Documents>kubectl run nginx --image=nginx --namespace=kube-public
pod/nginx created

C:\Users\kush.gupta\Documents>kubectl get pod -n kube-public
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          24s
```

- Or in pod specification file also we can write namespace-

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: kube-public
  labels:
    run: nginx
```

Then run kubectl apply -f pod.yaml.

- By specifying namespace in yaml itself we will make sure that always nginx pod in this case will be created in kube-public namespace.

Set default Namespace

- If we want to set default namespace so we don’t need to mention it every time. Use below command-> **kubectl config set-context --current --namespace=dev**
- Now we can skip -n dev options from commands.
- And if we wanted to get pods of other namespace which is not present in dev(default namespace after above command). Then we can type->
  kubectl get pod -n prod

- We can also create kind: ResourceQuota definition file to specify limits under spec section like cpu, no of pods allowed, memory etc. in it under metadata we can specify namespace then this quota will be applicable to that namespace. Otherwise it will get applied to default namespace.
- To count total no of namespaces in system. Use->
  kubectl get ns | wc -l
  It will give total count +1 as headers are also counted by above command.
- To identify what is the current namespace use->
  kubectl config view | grep -i namespace

Microservice example

Git url of microservices project along with instructions in readMe file.

https://github.com/kushguptacse/kubernetes

Imperative Commands

- Imperative commands can help in getting one time tasks done quickly, as well as generate a definition template easily.
- use the --dry-run=client option. This will not create the resource, instead, tell you whether the resource can be created and if your command is right.
- -o yaml: This will output the resource definition in YAML format on screen.

POD

Create an NGINX Pod

kubectl run nginx --image=nginx

kubectl run redis --image=redis:alpine -l tier=db

above command will create pod with name redis using image redis:alpine with label tier=db

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

kubectl run nginx --image=nginx --dry-run=client -o yaml > pod.yaml

- If you try to edit some specifications of pod dynamically using kubectl edit pod podname command it might be possible that it give error. For example - edit the environment variables is not possible dynamically. Updating security context, changing resources, tolerance etc are not possible at pod level.
  Approach 1->

```
master $ kubectl edit pod webapp
error: pods "webapp" is invalid
A copy of your changes has been stored to "/tmp/kubectl-edit-ccvrq.yaml"
error: Edit cancelled, no valid changes were saved.
master $
```

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
```

Approach 2 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
```

Deployment

Create a deployment

kubectl create deployment --image=nginx nginx

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

kubectl create deployment httpd-frontend --replicas=3 --image=httpd:2.4-alpine --dry-run=client -o yaml > deploy.yaml

Generate Deployment with 4 Replicas

kubectl create deployment nginx --image=nginx --replicas=4

You can also scale a deployment using the kubectl scale command.

kubectl scale deployment nginx --replicas=4

Another way to do this is to save the YAML definition to a file and modify

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

You can then update the YAML file with the replicas or any other field before creating the deployment. Then run

Kubectl create -f nginx-deployment.yaml

Edit Deployment:

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
```

Service

Create a Service yaml file without running service

kubectl create service nodeport nginx --tcp=80:80 --dry-run=client -o yaml > service.yaml

here you need to edit and set nodePort explicitly , otherwise default nodeport will be assigned.

kubectl create svc clusterip redis-service --tcp=6379:6379 --dry-run=client -o yaml > svc.yaml

now after verifying just run kubectl apply -f service.yaml.

Command to create and run pod on port 80 and expose it as service

kubectl run httpd --image=httpd:alpine --port=80 --expose

Configuration

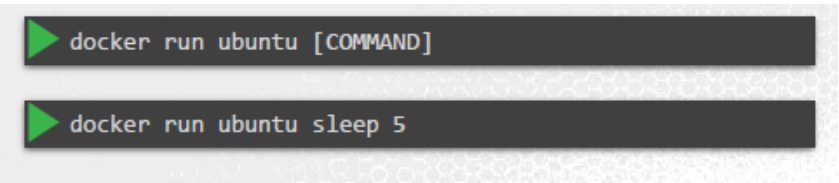
Commands and Arguments in Docker

- When you run the “docker run ubuntu” command, it runs an instance of Ubuntu image and exits immediately. Docker container ls will list nothing. And docker container ls -a will show one stop container of ubuntu.
- Unlike Virtual Machines, containers are not meant to host an Operating System. Containers are meant to run a specific task or process. Such as to host an instance of a Web Server, or Application Server or a database or simply to carry out some kind of computation or analysis. Once the task is complete the container exits. A container only lives as long as the process inside it is alive. If the web service inside the container is stopped or crashes the container exits.

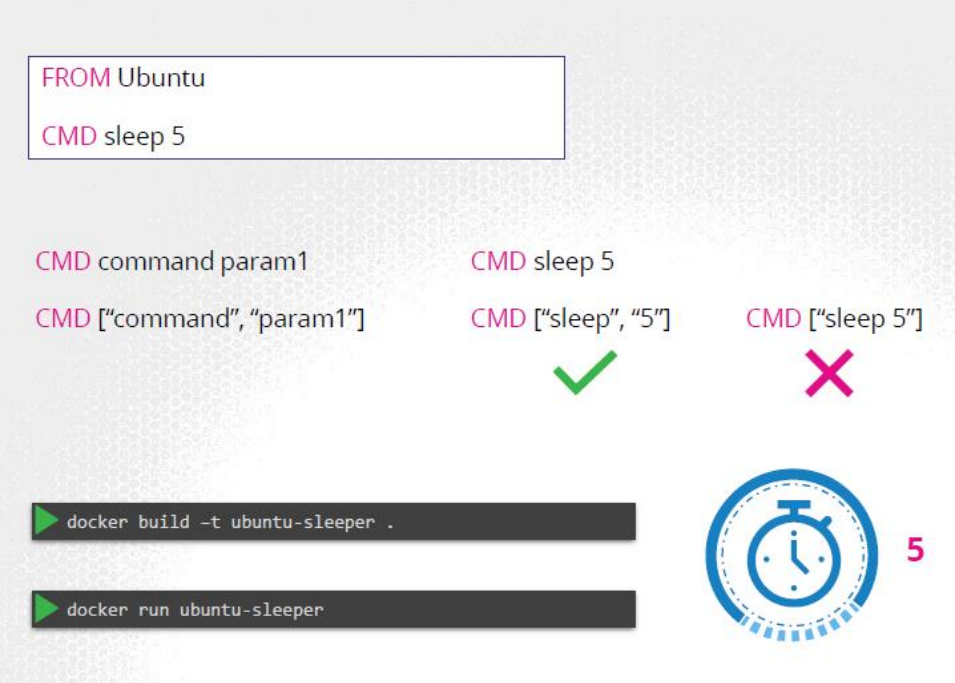
- So who defines what process is run within the container? If you look at the Dockerfile for the NGINX image, you will see an Instruction called CMD which stands for command that defines the program that will be run within the container when it starts. For the NGINX image it is the nginx command, for the mysql image it is the mysqld command.

```
# Pull base image.
FROM ubuntu:14.04

# Install.
RUN \
  sed -i 's/# \(.*multiverse$\)/\1/g' /etc/apt/sources.list && \
  apt-get update && \
  apt-get -y upgrade && \
  apt-get install -y build-essential && \
  apt-get install -y software-properties-common && \
  apt-get install -y byobu curl git htop man unzip vim wget && \
  rm -rf /var/lib/apt/lists/*

# Add files.
ADD root/.bashrc /root/.bashrc
ADD root/.gitconfig /root/.gitconfig
ADD root/.scripts /root/.scripts

# Set environment variables.
ENV HOME /root

# Define working directory.
WORKDIR /root

# Define default command.
CMD ["bash"]
```

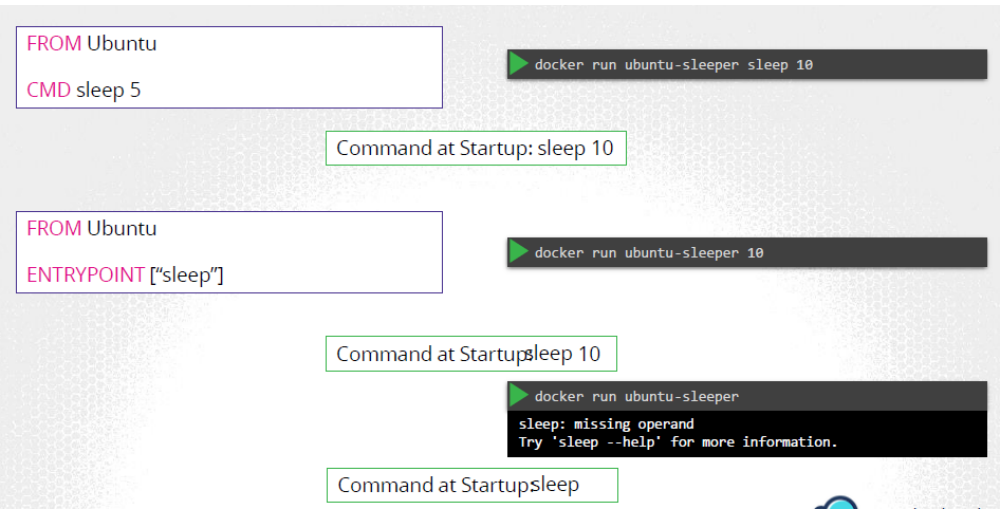
- Let us look at the Dockerfile for this image. You will see that it uses “bash” as the default command. Now, bash is not really a process like a web server or database. It is a shell that listens for inputs from a terminal. If it cannot find a terminal it exits. When we ran the Ubuntu container earlier, Docker created a container from the Ubuntu image, and launched the bash program. By default Docker does not attach a terminal to a container when it is run. And so the bash program does not find a terminal and so it exits. Since the process, that was started when the container was created, finished, the container exits as well.
- So how do you specify a different command to start the container? One option is to append a command to the docker run command and that way it overrides the default command specified within the image which was CMD [“bash”] in ubuntu case. If I run the docker run ubuntu command with the “sleep 5” command as the added option. This way when the container starts it runs the sleep program, waits for 5 seconds and then exits.



- We can also create new image say ubuntu-sleeper from ubuntu and then add -> CMD [“sleep”,”5”] or CMD sleep 5 at last. So I now build by new image using the docker build command, and name it as ubuntu-sleeper. I could now simply run the docker ubuntu sleeper command and get the same results. It always sleeps for 5 seconds and exits.



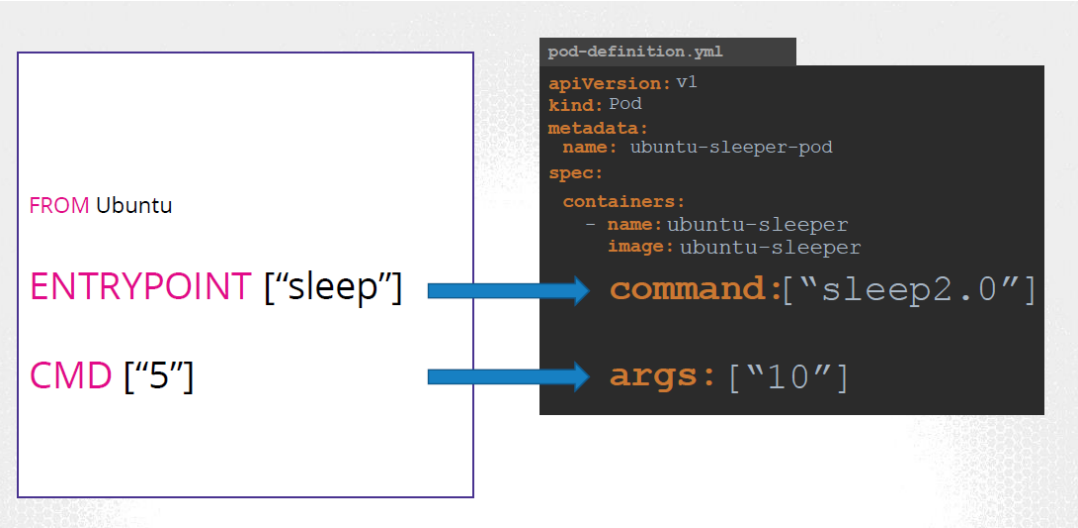
- But what if we want to change sleep time to some dynamic value. For that use-> docker run ubuntu-sleeper sleep 10. But it does not seems right as name itself is ubuntu-sleeper then why to specify sleep 10. Why cant we just run it like -> docker run ubuntu-sleeper 10. But if we try this then existing entry CMD["sleep","5"] will be replaced with CMD["10"] which is nothing and hence gives error on container start.
- And that is where the entrypoint instruction comes into play. The entrypoint instruction is like the command instruction, as in you can specify the program that will be run when the container starts. And whatever you specify on the command line, in below case 10, will get appended to the entrypoint. So the command that will be run when the container starts is sleep 10.
- So that's the difference between the two. In case of the CMD instruction the command line parameters passed will get replaced entirely, whereas in case of entrypoint the command line parameters will get appended. Now, in the third case below what if I run the ubuntu-sleeper without appending the number of seconds? Then the command at startup will be just sleep and you get the error that the operand is missing.



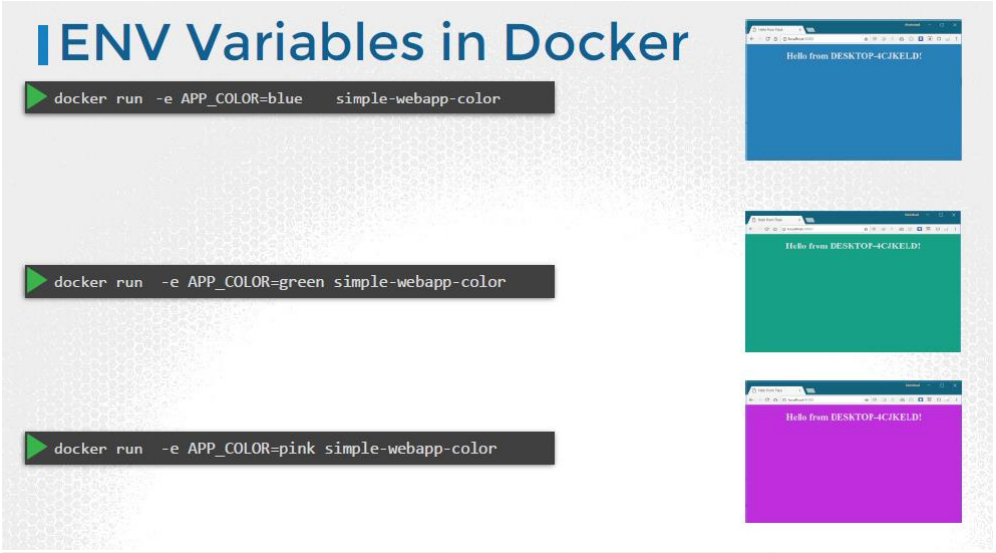
- What if you want default value as well. For that we can use combo of entrypoint and cmd both. In this case the command instruction will be appended to the entrypoint instruction. So at startup, the command would be sleep 5, if you didn't specify any parameters in the command line. If you did, then that will override the command instruction. And remember, for this to happen, **you should always specify the entrypoint and command instructions in a JSON format.**
- Finally, what if you really want to modify the entrypoint during run time? Say from sleep to a hypothetical sleep2.0 command? Well in that case you can override it by using the –entrypoint option in the docker run command.
  docker run --entrypoint sleep2.0 ubuntu-sleeper 10

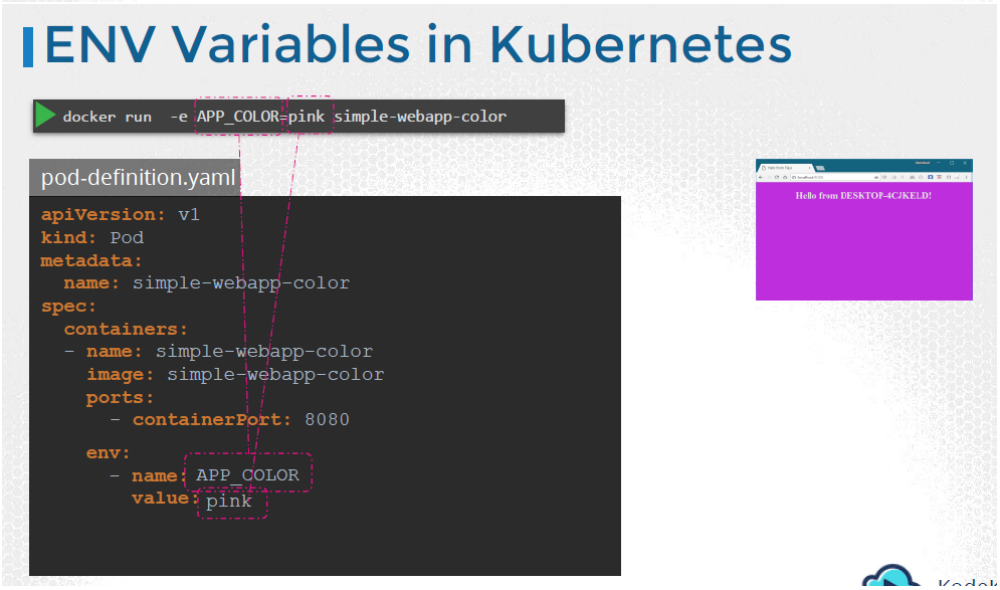
Commands and Arguments in Kubernetes

- If you want to achieve same thing we did earlier for ubuntu docker image with Kubernetes we can use **command** and **args** options in pod specification file. In this way when container run inside pod it will be running exactly like we wanted, .i.e. entry point will be sleep2.0 and cmd will be 10. if we did not specify anything here in pod spec.containers section then docker container will run with default values. The command and arguments that you define in the configuration file override the default command and arguments provided by the container image. If you define args, but do not define a command, the default command is used with your new arguments.
- So to summarize, there are two fields that correspond to two instructions in the Dockerfile. The command overrides the entrypoint instruction and the args field overrides the command instruction in the Dockerfile. Remember the command field does not override the CMD instruction in the Dockerfile.
- Note- values of command and args always a string array.



Environment variables in Kubernetes

- 

- 

- As we can see in k8s we provide env inside spec.containers in the form of key value pair. environment variable, use the ENV property. ENV is an array. So every item under the env property starts with a dash, indicating an item in the array. Each item has a name and a value property. The name is the name of the environment variable made available within the container and the value is its value.

Environment value types in Kubernetes

- What we just saw was a direct way of specifying the environment variables using a plain key value pair format. However there are other ways of setting the environment variables –such as using ConfigMaps and Secrets. The difference in this case is that instead of specifying value, we say valueFrom. And then a specification of configMap or secret.

ConfigMaps – cm - imperative

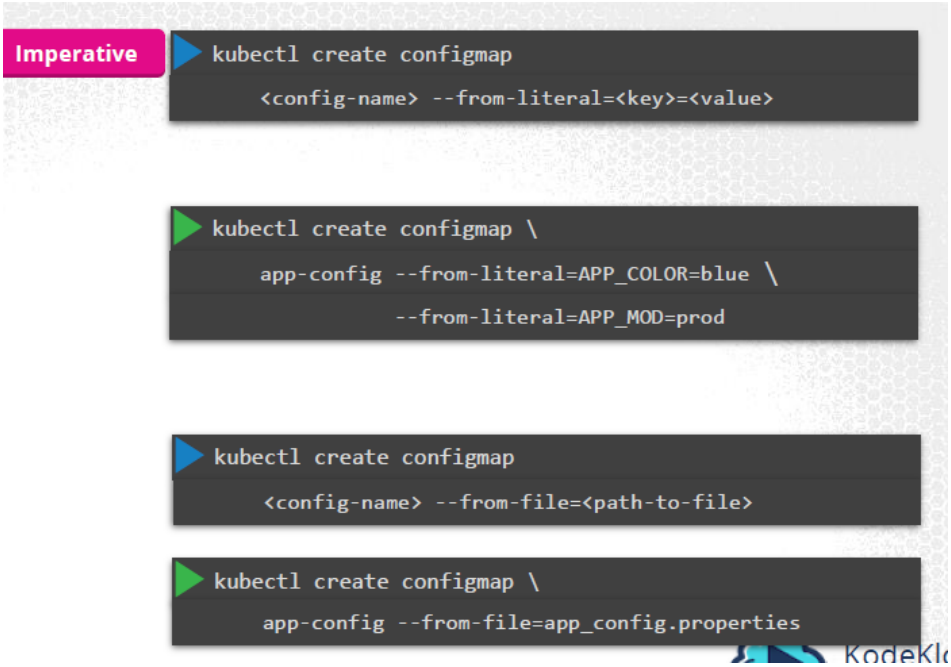
- By simple env we can pass key value pairs. But sometimes with lot of entries it is tough to mention each of them pair by pair. Better approach could be to create separate property file and then use it directly. For that another Kubernetes object ConfigMaps can be used.
- So it is a 2 step process. Step1 -> create config map with desired data loaded from property file. Step 2-> When a POD is created, inject the ConfigMap into the POD, so the key value pairs are available as environment variables for the application hosted inside the container in the POD.

Create ConfigMaps Imperative

Kubectl create cm name1 --from-literal=APP_COLOR=blue --from-literal=type=frontend

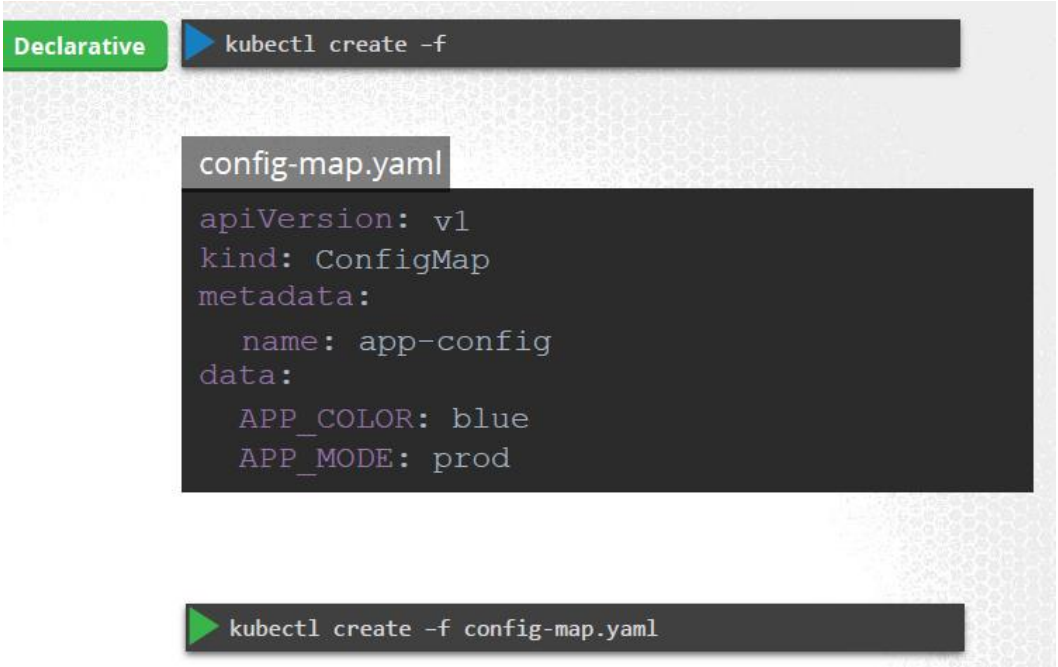
Kubectl create configmap name2 --from-file=sample.properties

Where sample.properties contains key: value entries.



Create ConfigMaps declartive

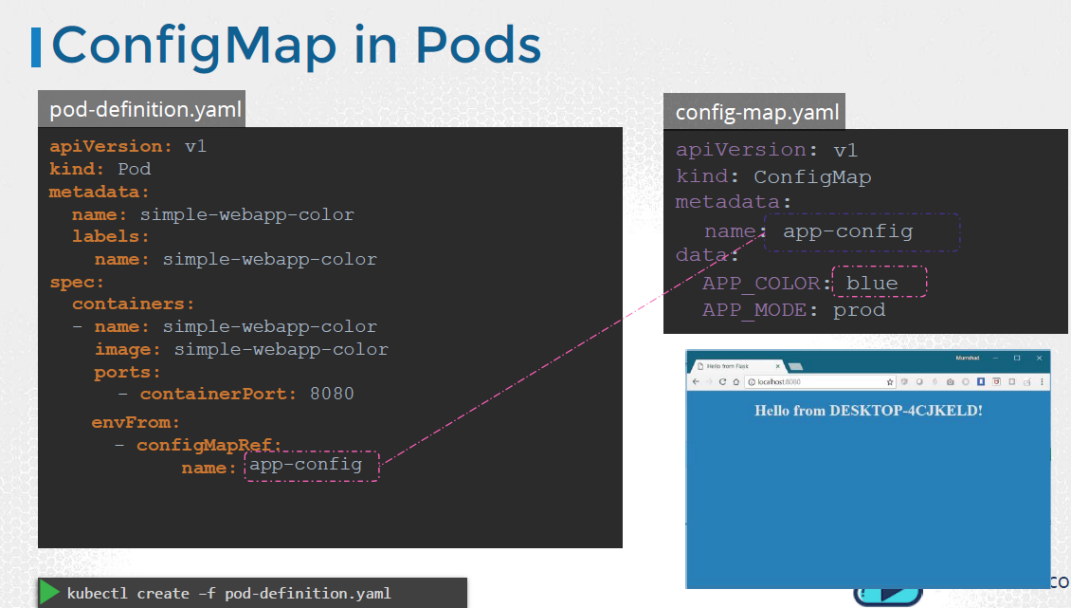
kubectl create configmap name1 --from-literal=APP_COLOR=blue --from-literal=type=frontend



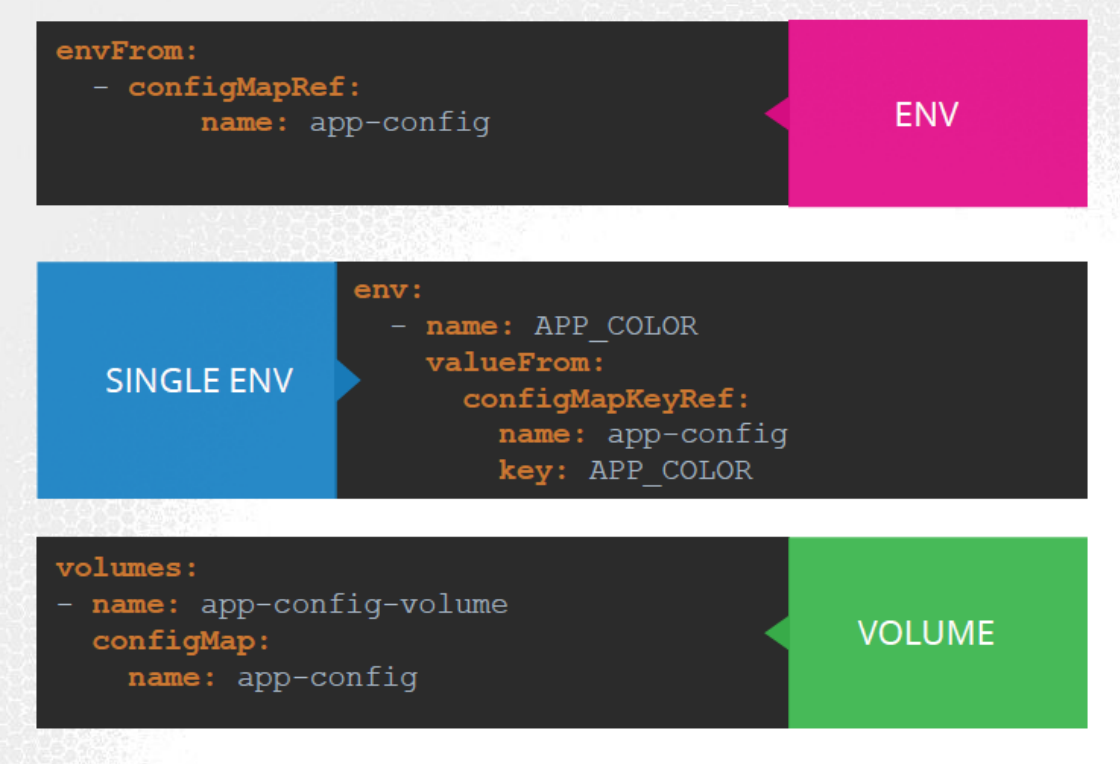
- So you can create different configmap for different environments like mysql, redis etc. then associate them with pod accordingly.

Inject configmap to pod

- Just replace env with envFrom and then add tag configMapRef and inside it add name: nameofConfigMap. By this way entire data in configMap is provided as environment variables to desired pod.



- What we just saw was using configMaps to inject environment variables. There are other ways to inject configuration data into PODs. You can inject a single environment variable or inject the whole configuration data as files in a volume.



Secret - imperative

- ConfigMap store key value pairs in plain text which is not good to store password. Secret are similar to configMap except they are store in a encoded and hashed format.
- They are not the best security practice as the value can be decoded easily by using any base64 decoder. But it will still better then plain text as it is not human readable.
- Kubelet stores secret into tmpfs so that they are not stored on disk. Secret is only sent to node if pod inside that node needed it. If pod that depend on secret is deleted, kubelet will delete local copy of secret data as well.

Create Secret Imperative

- kubectl create secret generic secretName --from-literal=pwd1=myPassword1 --from-literal=pwd2=myPassword2

```
01/05/2022 11:25.48 /home/mobaxterm kubectl create secret generic secret1 --from-literal=pwd1=password1 --from
-literal=pwd2=password2
secret/secret1 created

01/05/2022 11:26.43 /home/mobaxterm kubectl get secret
NAME                  TYPE                                  DATA   AGE
default-token-nr9tp   kubernetes.io/service-account-token   3      241d
secret1               Opaque                                2      11s

01/05/2022 11:26.54 /home/mobaxterm kubectl describe secret secret1
Name:         secret1
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
pwd1:  9 bytes
pwd2:  9 bytes
```

- if you edit secret by kubectl edit secret secretName. You can see secrets are not shown as plain text.

```
apiVersion: v1
data:
  pwd1: cGFzc3dvcmQx
  pwd2: cGFzc3dvcmQy
kind: Secret
metadata:
  creationTimestamp: "2022-05-01T05:56:43Z"
  name: secret1
  namespace: default
  resourceVersion: "224806817"
  uid: 5bfd2c1e-d5f4-40e2-ab45-5f2f36da233e
type: Opaque
```

- if you want to see decoded base64 value then copy text and use below command to check content.

```
01/05/2022 11:30.56 /home/mobaxterm echo -n cGFzc3dvcmQx | base64 -d
password1
```

- kubectl create secret generic secretName --from-file=secret.properties

```
01/05/2022 11:37.19 /home/mobaxterm cat sec.properties
pwd1: password1
pwd2: password2
```

```
01/05/2022 11:36.06 /home/mobaxterm kubectl create secret generic secret2 --from-file=sec.properties
secret/secret2 created
```

```
01/05/2022 11:36.44 /home/mobaxterm kubectl describe secret secret2
Name:         secret2
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
sec.properties:  32 bytes
```

- on edit also we can verify property file name is also encoded

```yaml
apiVersion: v1
data:
  sec.properties: cHdkMTogcGFzc3dvcmQxCnB3ZDI6IHBhc3N3b3JkMgo=
kind: Secret
metadata:
  creationTimestamp: "2022-05-01T06:06:32Z"
  name: secret2
  namespace: default
  resourceVersion: "224812334"
  uid: b1bc82f7-074a-43f9-bd82-58238c9f033c
type: Opaque
```

- Create TLS secret webhook-server-tls for secure webhook communication in webhook-demo namespace using
  Certificate : /root/keys/webhook-server-tls.crt
  Key: /root/keys/webhook-server-tls.key
- kubectl create secret tls webhook-server-tls --cert=/root/keys/webhook-server-tls.crt --key=/root/keys/webhook-server-tls.key -n webhook-demo

```
root@controlplane ~ ➜ kubectl describe  secret webhook-server-tls -n webhook-demo
Name:         webhook-server-tls
Namespace:    webhook-demo
Labels:       <none>
Annotations:  <none>

Type:  kubernetes.io/tls

Data
====
tls.crt:  1204 bytes
tls.key:  1679 bytes
```

Create Secret Declarative

- Declarative way is similar to that of configMap way. only difference is that you cannot store value as plain text. If you try to create secret with plain text value it will give error.

```
 01/05/2022  11:46.28  /home/mobaxterm  echo -n password1 | base64
cGFzc3dvcmQx

 01/05/2022  11:46.36  /home/mobaxterm  echo -n password2 | base64
cGFzc3dvcmQy
```

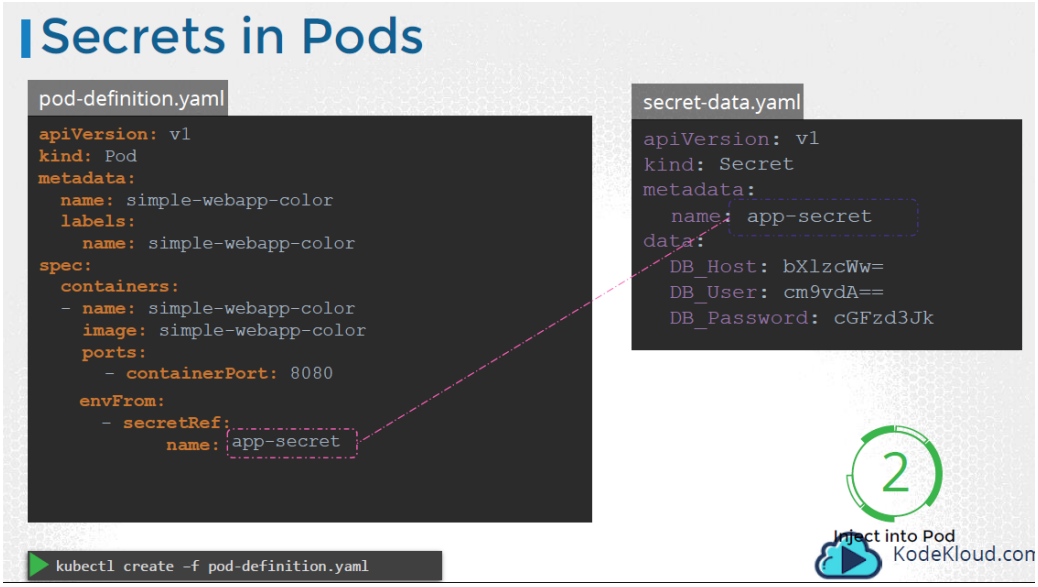
- Example-> below is sec.yaml definition file.

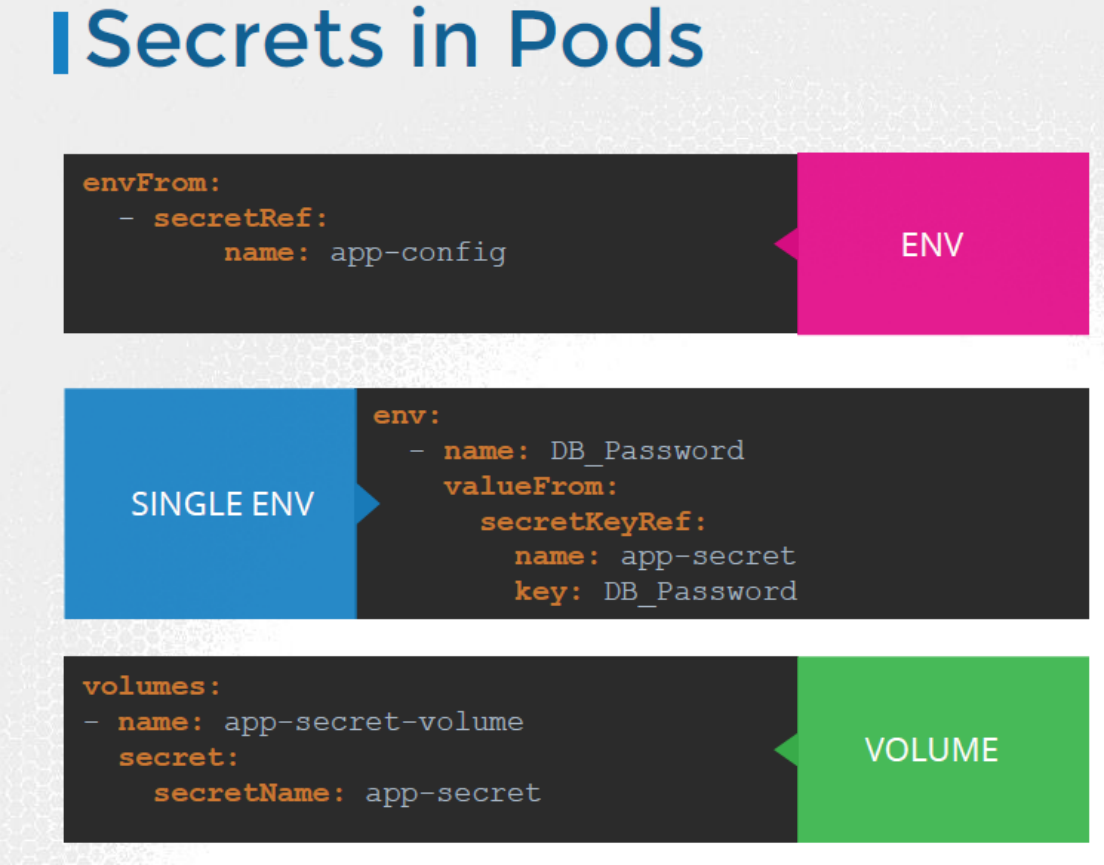
```yaml
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: null
  name: secret3
data:
  pwd1: cGFzc3dvcmQx
  pwd2: cGFzc3dvcmQy
```

```
C:\Users\kush.gupta\Documents\Statutory Benefits>kubectl create -f sec.yaml
secret/secret3 created

C:\Users\kush.gupta\Documents\Statutory Benefits>kubectl describe secret secret3
Name:         secret3
Namespace:    default
Labels:       <none>
Annotations:  <none>

Type:  Opaque

Data
====
pwd1:  9 bytes
pwd2:  9 bytes
```

Inject secret to pod

- It is similar to configMap here just use secretRef instead of configMapRef

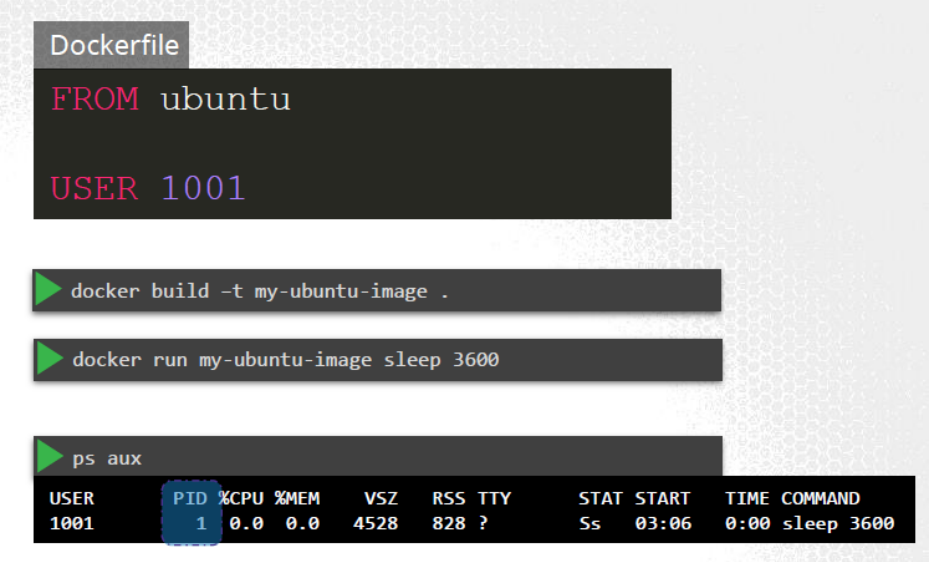




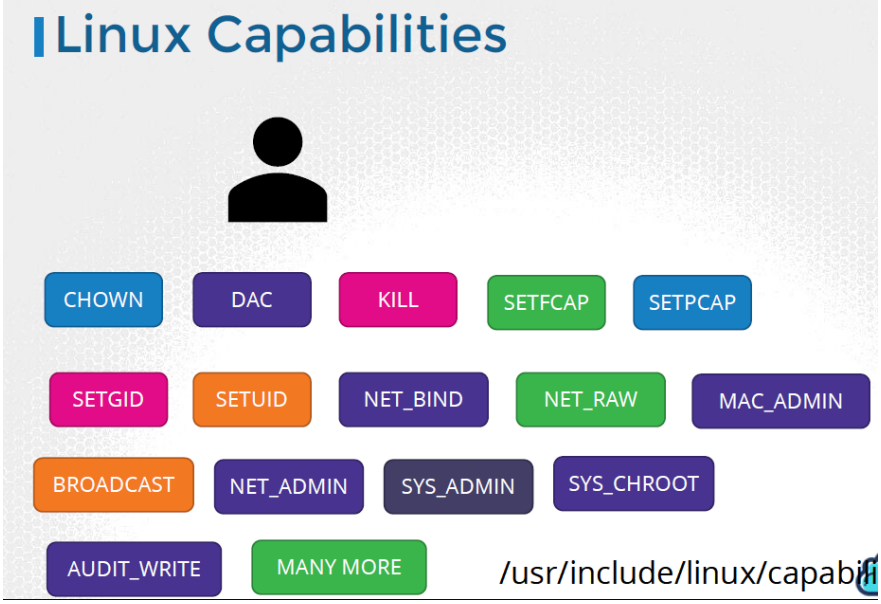
- If you mount secret file as shown above to volume. Then it will create separate file for each secret. Where secret key is file name and it's value as content.

Docker Security

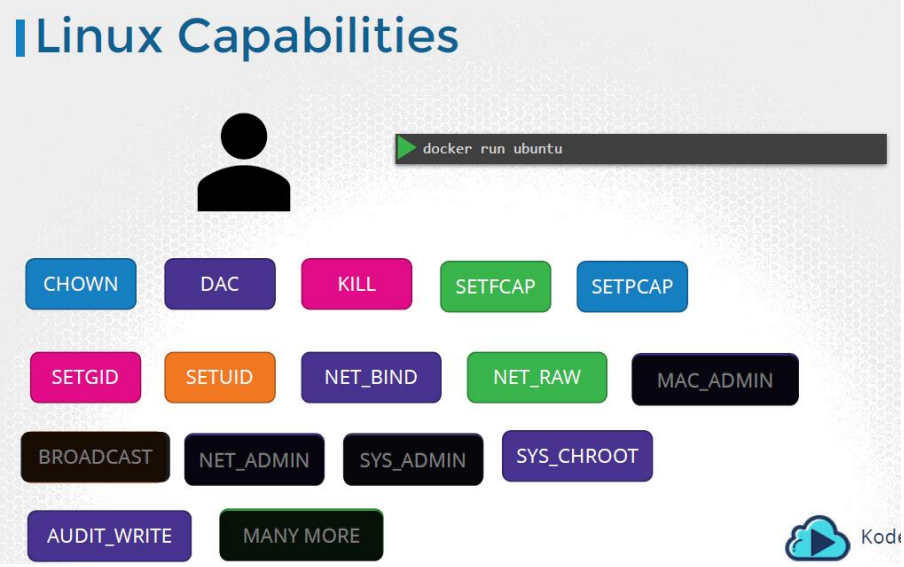
- Host on which docker is installed has it's own set of processes running including docker daemon itself.
- We have learned that unlike virtual machines containers are not completely isolated from their host. Containers and the hosts share the same kernel. Containers are isolated using namespaces in Linux. The host has a namespace and the containers have their own namespace. All the processes run by the containers are in fact run on the host itself, but in their own namespaces.
- So if you type px aux inside container to check no of processes running it will list down only 1. But on other hand host machine will list down all the processes including one that docker has created but with different process ID. This is because the processes can have different process IDs in different namespaces and that's how Docker isolates containers within a system. So that's process isolation.
- By default all processes inside container run as root user. Both within the container and outside the container on the host, the process is run as the root user. Now if you do not want the process within the container to run as the root user, you may set the user using the user option with the docker run command and specify the new user ID. You can also mention user in docker image also.



- Let us take a step back. What happens when you run containers as the root user? Is the root user within the container the same as the root user on the host? Can the process inside the container do anything that the root user can do on the system? If so isn't that dangerous? Well, docker implements a set of security features that limits the abilities of the root user within the container. So the root user within the container isn't really like the root user on the host. Docker uses Linux Capabilities to implement this.
- When process Is run by root user on host machine below set of capabilities (basically all) is given to that process.



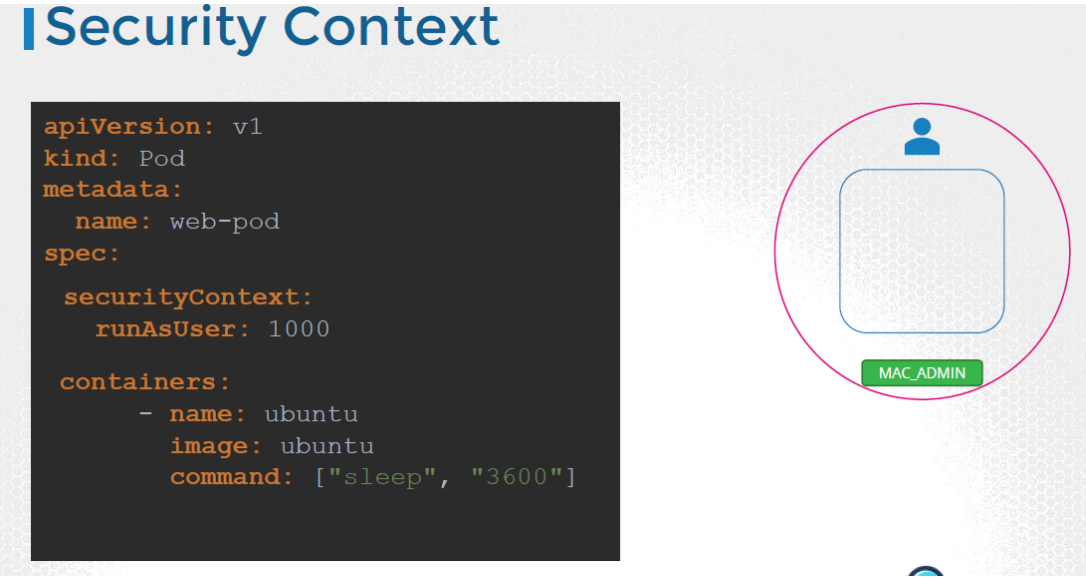
- Now, if docker container is run by root user. Then below set of capabilities by-default is provided. processes running within the container do not have the privileges to say, reboot the host or perform operations that can disrupt the host or other containers running on the same host.



- You can change this default capabilities by adding or dropping some of them when you are running container via docker run command. Similarly you can also provide all the privileges also which will be similar to processes run by host root user.

kubernetes securityContext

- As we saw in the previous lecture, when you run a Docker Container you have the option to define a set of security standards, such as the ID of the user used to run the container, the Linux capabilities that can be added or removed from the container etc. These can be configured in Kubernetes as well.
- In k8s you can specify such info under securityContext. Either at pod level or in container level. If specified at pod level it will be applicable to all the containers running inside that pod. If specified at both level. container level config will overwrite pod level configuration.
- In you want to do changes in securityContext of already running pod. Then you have to delete it first. Then create new pod with changes.
- In pod level. Under spec section you can specify securityContext and mention runAsUser to run containers inside pod with that user.



- At container level just move securityContext from spec section to containers section. Here you can also add/drop capabilities. Note- capabilities are only supported at container level not at pod level.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
      - name: ubuntu
        image: ubuntu
        command: ["sleep", "3600"]
        securityContext:
          runAsUser: 1000
          capabilities:
              add: ["MAC_ADMIN"]
```

- To check user who has run a podName. Type below command. ps aux | grep -i podName

```
controlplane ~ ➜ ps aux | grep -i ubuntu-sleeper
 6137 root      0:00 grep -i ubuntu-sleeper
```

kubernetes serviceAccount – sa - imperative

- 2 types of accounts exists in k8s. user account and service account. User account is used by users like admin or developer and will be discussed later. Example dev can access cluster to deploy application using user account. Service account is used by application to access cluster. For example a monitoring application like Prometheus uses a service account to poll the kubernetes API for performance metrics. An automated build tool like Jenkins uses service accounts to deploy applications on the kubernetes cluster.
- kubectl create serviceaccount name can be used to create service account.
- kubectl create sa name

```
kubectl create serviceaccount dashboard-sa
serviceaccount "dashboard-sa" created
```

```
kubectl get serviceaccount
NAME                 SECRETS   AGE
default              1         218d
dashboard-sa         1         4d
```

```
kubectl describe serviceaccount dashboard-sa
Name:                dashboard-sa
Namespace:           default
Labels:              <none>
Annotations:         <none>
Image pull secrets:  <none>
Mountable secrets:   dashboard-sa-token-kbbdm
Tokens:              dashboard-sa-token-kbbdm
Events:              <none>
```

- So when a service account is created, it first creates the service account object and then generates a token for the service account. It then creates a secret object and stores that token inside the secret object. The secret object is then linked to the service account. To view the token, view the secret object by running the command kubectl describe secret. This token can then be used as an authentication bearer token while making a rest call to the kubernetes API.
- So, that's how you create a new service account and use it. You can create a service account, assign the right permissions using Role based access control mechanisms and export your service account tokens and use it to configure your third party application to authenticate to the kubernetes API. But what if your third

party application is hosted on the kubernetes cluster itself. For example, we can have our custom-kubernetes-dashboard or the Prometheus application used to monitor kubernetes, deployed on the kubernetes cluster itself. In that case, this whole process of exporting the service account token and configuring the third party application to use it can be made simple by automatically mounting the service token secret as a volume inside the POD hosting the third party application. That way the token to access the kubernetes API is already placed inside the POD and can be easily read by the application.

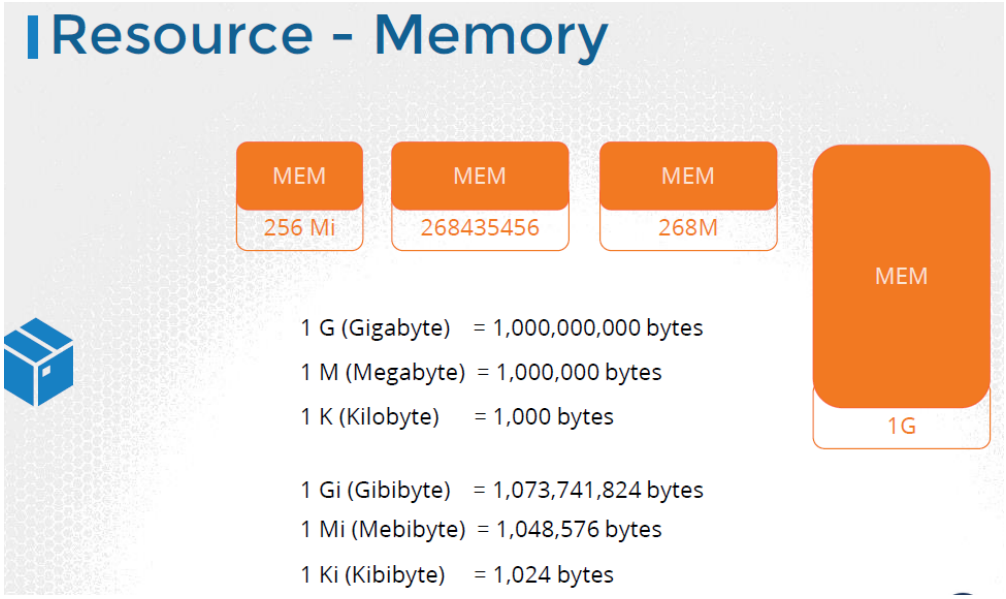
- For every namespace in kubernetes a service account named default is automatically created. Each namespace has its own default service account.
- Every pod will by default attached to this default service account. This default service account provides limited access to kube-api. In case you want to customized it, then create own service account and modify the pod definition file to include a serviceAccountName and specify the name of the new service account. Remember, you cannot edit the service account of an existing pod, so you must delete and re-create the pod. However in case of a deployment, you will be able to edit the serviceAccount, as any changes to the pod definition will automatically trigger a new roll-out for the deployment. So the deployment will take care of deleting and re-creating new pods with the right service account.
- Example yaml->

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: postgres-pod
  labels:
    name: postgres-pod
    app: demo-voting-app
spec:
  automountServiceAccountToken: false
  serviceAccountName: default
  securityContext:
    runAsUser: 1000
  containers:
    - name: postgres
      image: postgres
```

- automountServiceAccountToken indicates whether a service account token should be automatically mounted. Which is by default true.

kubernetes Resource Requirement

- Every node has a set of CPU, Memory and Disk resources available. Every POD consumes a set of resources. Whenever a POD is placed on a Node, it consumes resources available to that node. So scheduler takes into consideration, the amount of resources required by a POD and those available on the Nodes before scheduling pod to a node.
- If there is no sufficient resources available on any of the nodes, Kubernetes holds back scheduling the POD, and you will see the POD in a pending state. If you look at the events, you will see the reason for example– insufficient cpu.
- 1 count of CPU is equivalent to 1 vCPU. That's 1 vCPU in AWS, or 1 Core in GCP or Azure.



- By default docker container will take as much cpu as possible when needed and memory also. Which might lead other processes starve on that node. So to prevent that you can set resource/limits under containers section of pod definition. When the pod is created, kubernetes sets new limits for the container. Remember that the limits and requests are set for each container.

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    resources:
      requests:
        memory: "1Gi"
        cpu: 1
      limits:
        memory: "2Gi"
        cpu: 2
```

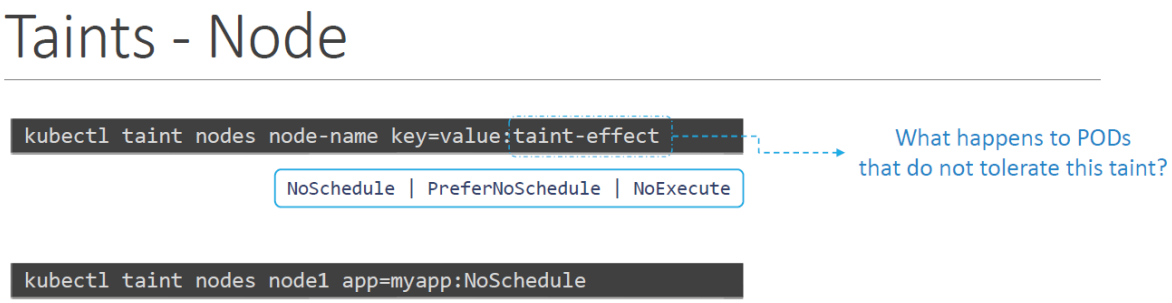
- So what happens when a pod tries to exceed resources beyond its specified limit. In case of the CPU, kubernetes throttles the CPU so that it does not go beyond the specified limit. A container cannot use more CPU resources than its limit. However, this is not the case with memory. A container CAN use more memory resources that its limit. So if a pod tries to consume more memory than its limit constantly, the POD will be terminated.
- If you want to set default value of these limit and request. So that each pod will have such values. Then create LimitRange k8s object.
  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace/
  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/
  https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource

Taint and Tolerance - imperative

- You can Taint node in k8s. by which only those pod who has same tolerance as taint value will be able to run on that node. All pods which does not have any/matching tolerance set cannot run on that node. It might be possible that if a pod has some tolerance set can run on node which does not have any/matching taint set.

This is because taint just guarantee only one rule -> if any pod need to run on node it must have tolerance matching with taint.

- kubectl taint nodes nodeName app=myapp:NoSchedule.




```
root@controlplane:~# kubectl taint node node01 spray=mortein:NoSchedule
node/node01 tainted
```

- once taint is created you can apply them on pod. You need to restart pod.

pod-definition.yml

```
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: nginx-container
    image: nginx

  tolerations:
  - key:"    "
    operator:"Equal"
    value:"    "
    effect:"          "
```

- here key is spray, value is mortein and effect can be NoSchedule, NoExecute, PreferNoSchedule.
- All values are string above.
- NoSchedule will not allow any new pod to schedule on node if it does not have matching node.
- NoExecute will not allow any new pod to schedule and also will delete existing pod running on that node if taint does not matched with pod tolerations.
- PreferNoSchedule will try not to schedule unmatching pod to node. But does not strictly follow.
- If you want pod to be scheduled only to desired node then in such case NodeAffinity, Node selector can be used.
- Scheduler does not schedule any pod on master. To implement that on master node a new taint is attached with effect as NoSchedule and since no other app container has that tolerance set , no app pod can run on master node.
- To untaint a node. Use same command just at last add minus.

```
root@controlplane:~# kubectl taint node controlplane node-role.kubernetes.io/master:NoSchedule
node/controlplane tainted
root@controlplane:~# kubectl taint node controlplane node-role.kubernetes.io/master:NoSchedule-
node/controlplane untainted
```

Node selectors

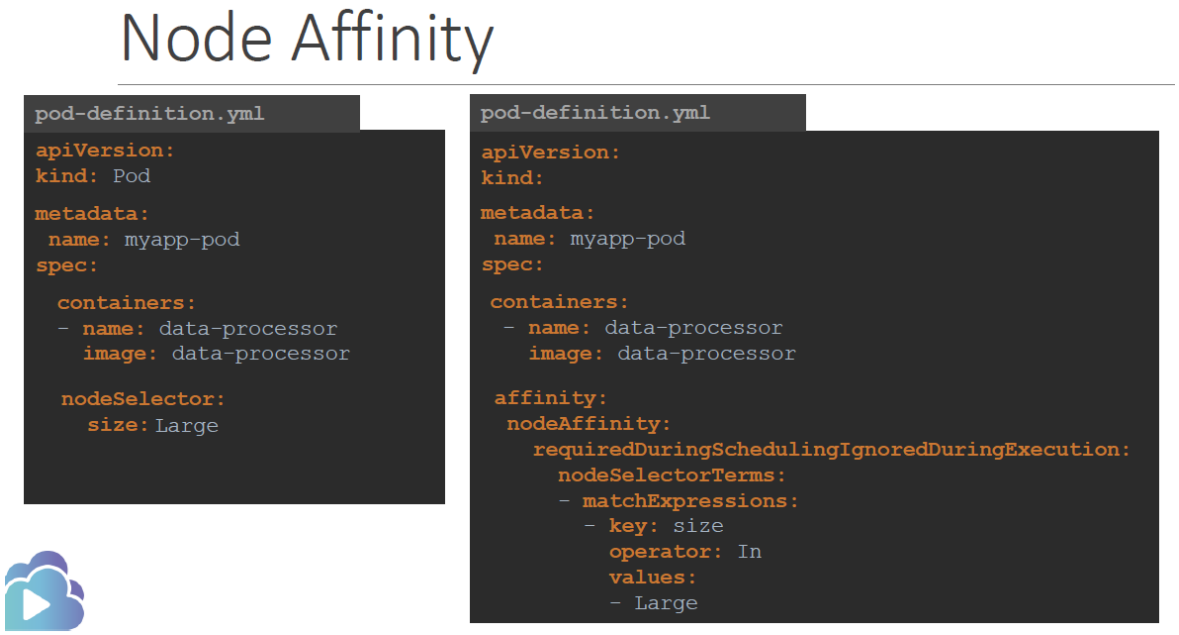
- To do one to one mapping of pod with node. Node selectors can be used. By this pod will run on exact node which we want. How to do it. Create label and attach it to node first. Then in pod specs section add nodeSelectors. and add label key value pair.
- kubectl label node nodeName key1=value1
- above command will create label and attach it to nodeName
- then in pod.yaml ->

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx1-dummy
  labels:
    app: myapppod
spec:
  nodeSelector:
    key1: value1
  automountServiceAccountToken: false
  serviceAccountName: default
  securityContext:
    runAsUser: 1001
  tolerations:
    - key: "color"
      value: "blue"
      operator: "Equal"
      effect: "NoSchedule"
  containers:
    - name: nginx
      image: nginx
```

- limitations is that it is one to one mapping only as it check only one label selector. We cannot write key1=value1 or key2=value2
- similarly we cannot right ! operator also.
- For these NodeAffinity can be used.

Node Affinity

- It provides all the possible expressions by which a pod can be configured to run on a node. It's syntax is little complicated but provide all options.
- To compare below image contain nodeSelector equivalent nodeAffinity. Here In operator is saying that any values mentioned in values array can match.



- nodeAffinity configuration can also include NotIn operator. We can also check Exists operator which just check if label matching key exists on node and ignore value. If not that pod will not be placed. There are some other operators also.
- What will happen if someone changed label of node in future, will pod continues to run? What if pod did not find any matching node.

Node Affinity Types

Available:

requiredDuringSchedulingIgnoredDuringExecution

preferredDuringSchedulingIgnoredDuringExecution

	DuringScheduling	DuringExecution
Type 1	Required	Ignored
Type 2	Preferred	Ignored

Node Affinity Types

Planned:

requiredDuringSchedulingRequiredDuringExecution

	DuringScheduling	DuringExecution
Type 1	Required	Ignored
Type 2	Preferred	Ignored
Type 3	Required	Required

- Type1- It means when new pod is created and at time scheduling, scheduler will follow nodeAffinity rule to check pod can be run on that node or not. But if pod is already running on node and somebody updated label on that node in such case existing pod will still be kept on running.
- Type2- It means when scheduler will try to prefer matching node as per nodeAffinity configuration. It is not mandatory though. This config can be used to ensure that pod will get hosted at least to any node even in case of no match found.
- Type 3- it is not available at present and will be available in future. In such case existing pod will also be terminated if label is changed in node.

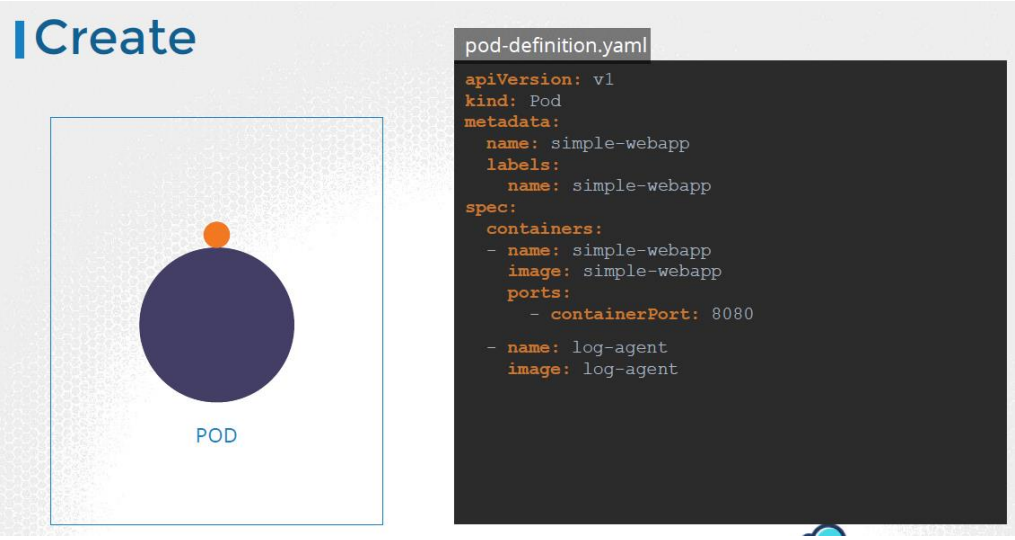
Node Affinity vs taint and tolerance.

- Taint and toleration makes sure that tainted node can run only those pod which has matching tolerance. But it might be possible that pod with matching tolerance can run on other nodes.
- NodeAffinity makes sure pod1 will got desired node1 with matching label. But if in a same cluster we have some other pods which does not have any node affinity set in such case. That pod2 can run on our node1 also which we thought should run only pods with matching label. But since pod2 does not have any node affinity set. It can run on any node including node1.
- To avoid that we can use combination of taint and tolerance and nodeaffinity. Taint and tolerance will ensure that no unwanted pod can run on node1. And node affinity make sure our intended pod1 will be run on node1.

Multi-Container Pods

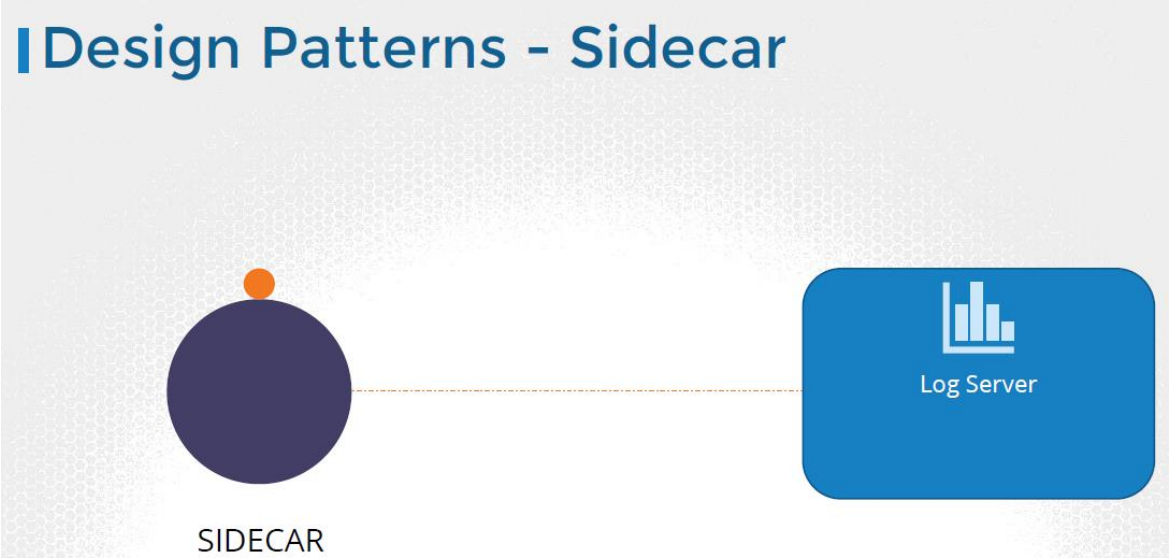
- There are different patterns of multi-container pods such as the Ambassador, Adapter and Sidecar. We will look at each of these in this section.
- Sometimes we need two container to work together in a single pod. Like logging container which collect log of application container and forward it to logging server. In such case logging container can interact with app container using local host as they share same pod. they share the same lifecycle –which means they are created together and destroyed together. They share the same network space, which means they can refer

to each other as localhost. And they have access to the same storage volumes. This way, you do not have to establish, volume sharing or services between the PODs to enable communication between them.

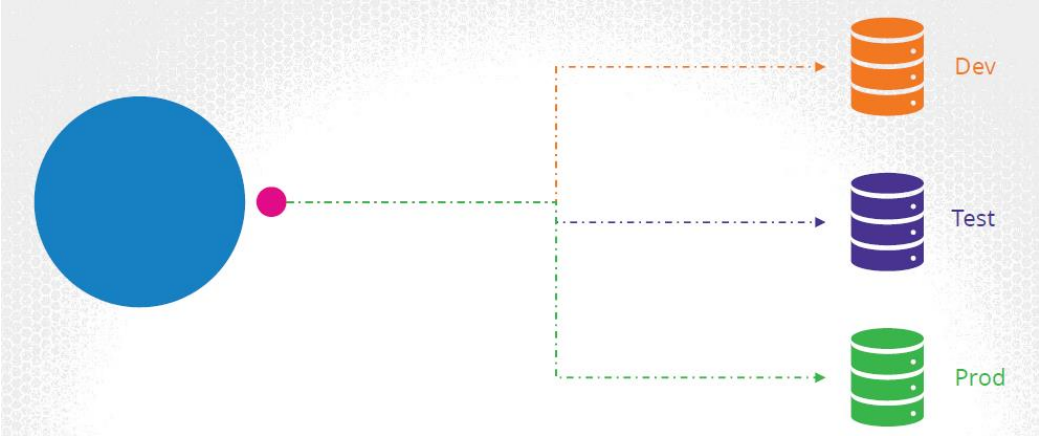


Design patterns in Multi-container pod.

- Configuration wise all design pattern will have same pod-definition file syntax as they are just guidelines which one follow while designing helper application.
- **SideCar**: A good example of a side car pattern is deploying a logging agent along side a web app to collect logs and forward them to a central log server.



- **Adaptor ->** Building on that example, say we have multiple applications generating logs in different formats. It would be hard to process the various formats on the central logging server. So, before sending the logs to the central server, we would like to convert the logs to a common format. For this we deploy an adapter container. The adapter container processes the logs, before sending it to the central server.
- **Ambassador ->** So your application communicates to different database instances at different stages of development. A local database for development, one for testing and another for production. You must ensure to modify this connectivity depending on the environment you are deploying your application to. You may choose to outsource such logic to a separate container within your POD, so that your application can always refer to a database at localhost, and the new container, will proxy that request to the right database. This is known as an ambassador container.



Observability

Pod lifecycle

- Pod has different stages in the lifecycle of a POD. A POD has a pod status and some conditions.
- When a POD is first created, it is in a **Pending state**. This is when the Scheduler tries to figure out where to place the POD. If the scheduler cannot find a node to place the POD, it remains in a Pending state. To find out why it's stuck in a pending state, run the kubectl describe pod command.
- Once the POD is scheduled, it goes into a **ContainerCreatingstatus**, were the images required for the application are pulled and the container starts. Once all the containers in a POD starts, it goes into a **running state**, where it continues to be until the program completes successfully or is terminated.
- You can see the pod status in the output of the **kubectl get pods** command. So remember, at any point in time the POD status can only be one of these values and only gives us a high level summary of a POD.

```
osboxes@kubemaster:~$ kubectl get pods
NAME                       READY     STATUS    RESTARTS   AGE
jenkins-566f687bf-c7nzf    1/1       Running   0          12m
nginx-65899c769f-9lwzh     1/1       Running   0          6h
redis-b48685f8b-fbnmx      1/1       Running   0          6h
```

- Sometimes we want additional information apart from these 3 pod stages values. For that pod conditions can be used to get that info.
- Conditions compliment POD status. **It is an array of true or false values that tell us the state of a POD**. When a POD is scheduled on a Node, the **PodScheduled** condition is set to True. When the POD is **initialized**, it's value is set to True. We know that a POD has multiple containers. When all the containers in the POD are ready, the **Containers Ready** condition is set to True and finally the POD itself is considered to be **Ready**.
- To see the state of POD conditions run the **kubectl describe POD** command and look for the conditions section.

```
04/05/2022  11:26.38  /home/mobaxterm  kubectl describe pod keng03-dev01-ath87-oam87-0 -n keng03-dev01-ath87-1648
463927 | grep -i conditions -A 7
Conditions:
  Type                                Status
  cloud.google.com/load-balancer-neg-ready   True
  Initialized                         True
  Ready                               True
  ContainersReady                     True
  PodScheduled                        True
Volumes:
```

Readiness Probe

- The ready conditions indicate that the application inside the POD is running and is ready to accept user traffic. What does that really mean? The containers could be running different kinds of applications in them. It could be a simple script that performs a job. It could be a database service. Or a large web server, serving front end users. The script may take a few milliseconds to get ready. The database service may take a few seconds to power up. Some web servers could take several minutes to warm up. If you try to run an instance of a Jenkins server, you will notice that it takes about 10-15 seconds for the server to initialize before a user can access the web UI. Even after the Web UI is initialized, it takes a few seconds for the server to warm up

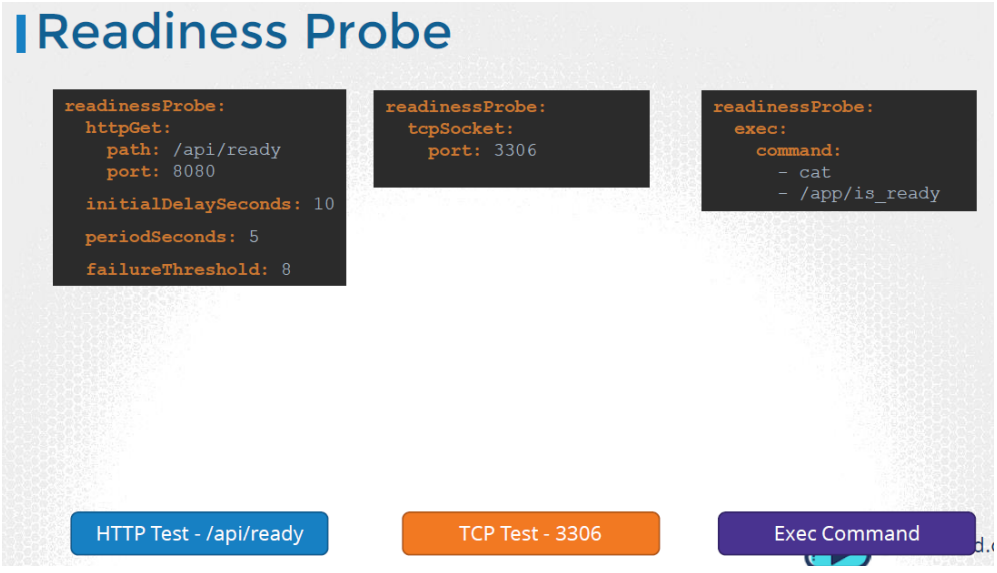
and be ready to serve users. During this wait period if you look at the state of the pod, it continues to indicate that the POD is ready, which is not very true.

- By default, Kubernetes assumes that as soon as the container is created, it is ready to serve user traffic. So it sets the value of the "Ready Condition" for each container to True. But if the application within the container took longer to get ready, the service is unaware of it and sends traffic through as the container is already in a ready state, causing users to hit a POD that isn't yet running a live application.
- What we need here is a way to tie the ready condition to the actual state of the application inside the container. As a Developer of the application, YOU know better what it means for the application to be ready.
- There are different ways that you can define if an application inside a container is actually ready. You can setup different kinds of tests or Probes, which is the appropriate term. In case of a web application it could be when the API server is up and running. So you could run a HTTP test to see if the API server responds. In case of database, you may test to see if a particular TCP socket is listening. Or You may simply execute a command within the container to run a custom script that would exit successfully if the application is ready.

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
      - containerPort: 8080

    readinessProbe:
      httpGet:
        path: /api/ready
        port: 8080
```

- In the pod definition file, add a new field called readinessProbe and use the httpGet option. Specify the port and the ready api. Now when the container is created, kubernetes does not immediately set the ready condition on the container to true, instead, it performs a test to see if the api responds positively. Until then the service does not forward any traffic to the pod, as it sees that the POD is not ready.



- There are some additional options as well. If you know that your application will always take a minimum of, say, 10 seconds to warm up, you can add an initial delay to the probe. If you'd like to specify how often to probe, you can do that using the periodSeconds option. By default if the application is not ready after 3 attempts, the probe will stop and failed. If you'd like to make more attempts, use the failureThreshold option.
- If the liveness/Readiness probe fails, **the kubelet kills the container, and the container is subjected to its restart policy**. If a container does not provide a liveness probe, the default state is Success.
- Say you have a replica set or deployment with multiple pods. And a service serving traffic to all the pods. There are two PODs already serving users. Say you were to add an additional pod. And let's say the Pod takes a minute to warm up. Without the readinessProbe configured correctly, the service would immediately start routing traffic to the new pod. That will result in service disruption to at least some of the users. Instead if the pods were configured with the correct readinessProbe, the service will continue to serve traffic only to the older pods and wait until the new pod is ready. Once ready, traffic will be routed to the new pod as well, ensuring no users are affected.

Liveness Probe

- It is similar to readinessProbe in a way that here also we define in exact same way with same options to tell what will be considered to verify if application inside container is healthy Except here you use livenessProbe instead of readinessProbe.
- The liveness probe is configured in the pod definition file as you did with the readinessProbe.
- In general, Every time the application crashes, kubernetes makes an attempt to restart the container to restore service to users. You can see the count of restarts increase in the output of kubectl get pods command. Now this works just fine.
- However, what if the application is not really working but the container continues to stay alive? Say for example, due to a bug in the code, the application is stuck in an infinite loop. As far as kubernetes is concerned, the container is up, so the application is assumed to be up. But the users hitting the container are not served. In that case, the container needs to be restarted, or destroyed and a new container is to be brought up. That is where the liveness probe can help us. A liveness probe can be configured on the container to periodically test whether the application within the container is actually healthy. If the test fails, the container is considered unhealthy and is destroyed and recreated.
- But again, as a developer, you get to define what it means for an application to be healthy.

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
  - name: simple-webapp
    image: simple-webapp
    ports:
      - containerPort: 8080

    livenessProbe:
      httpGet:
        path: /api/healthy
        port: 8080
```

Container Logging

- kubectl logs podName

- Above command will work and give logs of the container running in pod. But if pod contains multiple containers then along with podname container name also needed to be provided.
- kubectl logs podname containername
- you can add -f option to check live logging of container in a pod. Example -> kubectl logs -f pod1

Monitor and debug application

- we would like to know Node level metrics such as the number of nodes in the cluster, how many of them are healthy as well as performance metrics such as CPU. Memory, Network and disk utilization.
- As well as POD level metrics such as the number of PODs, and performance metrics of each POD such the CPU and Memory consumption. So we need a solution that will monitor these metrics, store them and provide analytics around this data.
- As of this recording , Kubernetes does not come with a full featured built-in monitoring solution. However, there are a number of open-source solutions available today, such as the Metrics-Server, Prometheus, the Elastic Stack, and proprietary solutions like Datadog and Dynatrace.
- **Metrics-Server:** You can have one metrics server per kubernetes cluster.
- So how are the metrics generated for the PODs on these nodes? Kubernetes runs an agent on each node known as the kubelet, which is responsible for receiving instructions from the kubernetes API master server and running PODs on the nodes. The kubelet also contains a subcomponent known as **cAdvisor** or Container Advisor. cAdvisor is responsible for retrieving performance metrics from pods, and exposing them through the kubeletAPI to make the metrics available for the Metrics Server. Then metric-server uses this api to collect data and show them.
- Clone git repo of metric-server and then create Kubernetes resources using yaml definition files.

```
git clone https://github.com/kubernetes-incubator/metrics-server.git
```

```
kubectl create -f deploy/1.8+/
clusterrolebinding "metrics-server:system:auth-delegator" created
rolebinding "metrics-server-auth-reader" created
apiservice "v1beta1.metrics.k8s.io" created
serviceaccount "metrics-server" created
deployment "metrics-server" created
service "metrics-server" created
clusterrole "system:metrics-server" created
clusterrolebinding "system:metrics-server" created
```

- This will deploys a set of pods, services and roles to enable metrics server to poll for performance metrics from the nodes in the cluster.
- Once deployed, give the metrics-server some time to collect and process data. Once processed, cluster performance can be viewed by running the command kubectl top node. This provides the CPU and Memory consumption of each of the nodes.
- Use the kubectl top pod command to view performance metrics of pods in kubernetes.

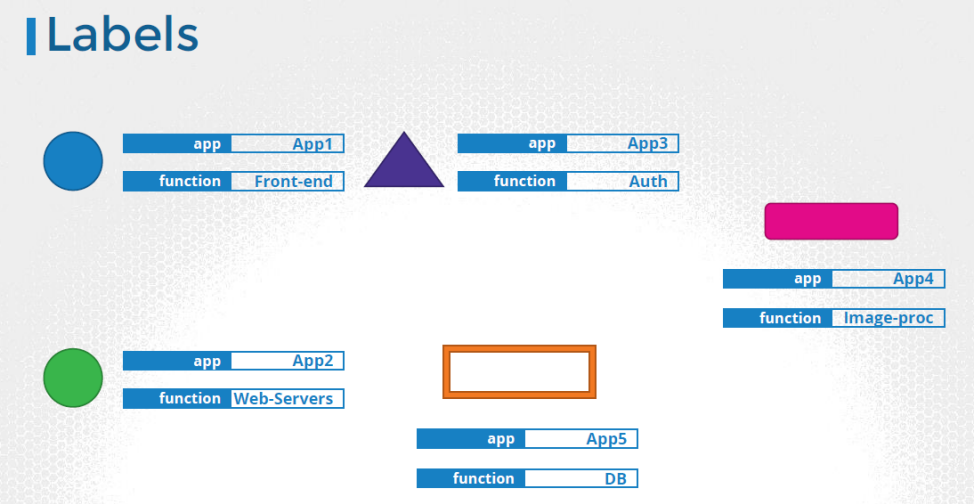
```
kubectl top node
NAME         CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
kubemaster   166m         8%        1337Mi          70%
kubenode1    36m          1%        1046Mi          55%
kubenode2    39m          1%        1048Mi          55%
```

```
kubectl top pod
NAME         CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
nginx        166m         8%        1337Mi          70%
redis        36m          1%        1046Mi          55%
```

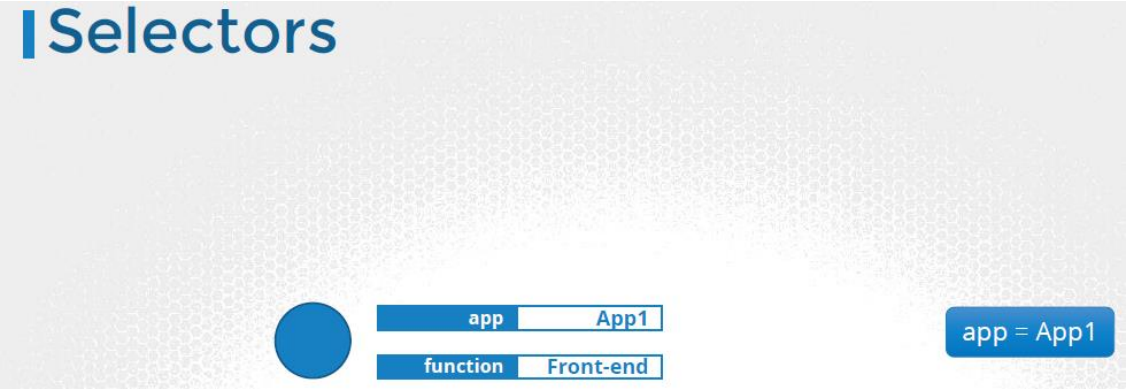
POD Design

Label, Selectors and Annotations

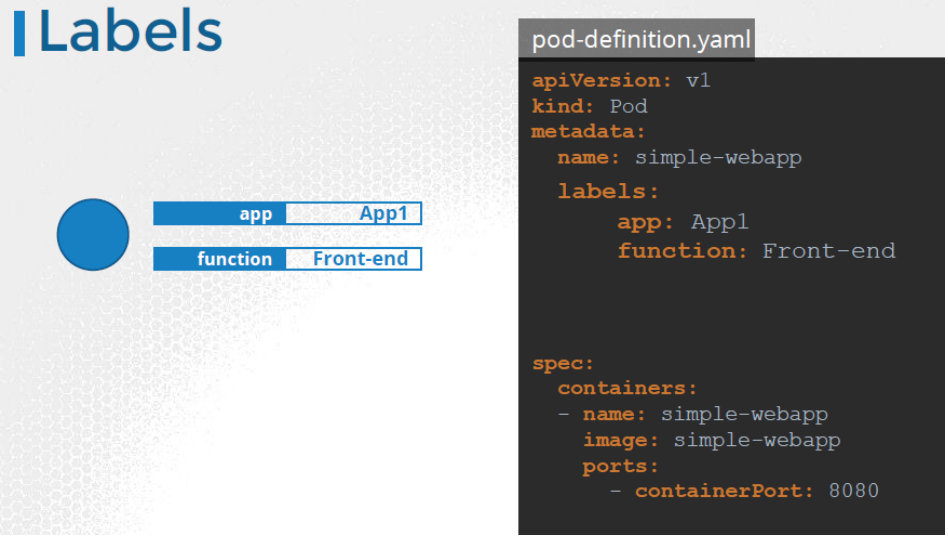
- To group Kubernetes objects we can assign labels to them. For each object attach labels as per your needs, like app, function etc.



- In this way we can also filter out objects using labels. We do this filtering by selectors.
  kubectl get pods --selector app=APP1
  kubectl get pods -l app=APP1

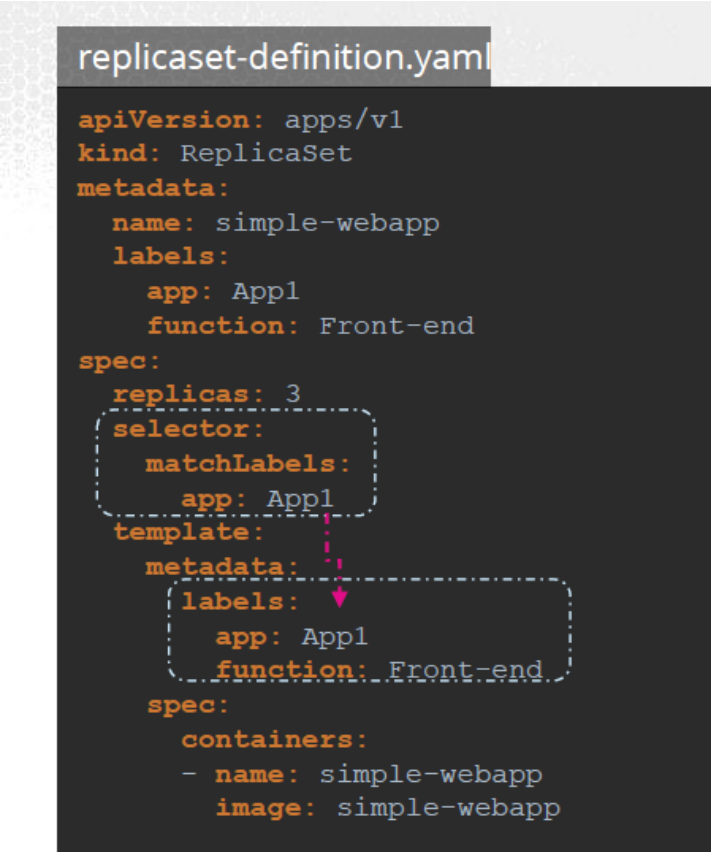


- Label example in pod-definition.yaml

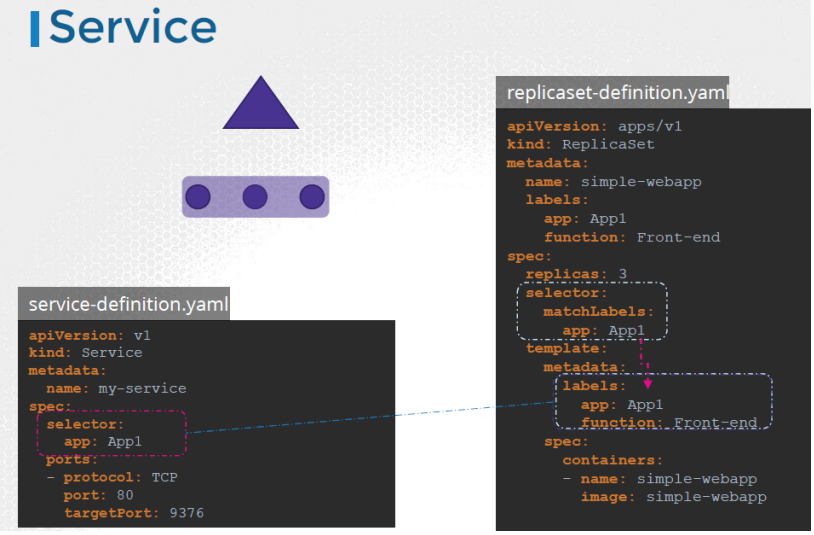


- Now this is one use case of labels and selectors. Kubernetes objects use labels and selectors internally to connect different objects together. For example to create a replicaset consisting of 3 different pods, we first label the pod definition and use selector in a replicaset to group the pods . In the replica-set definition file, you will see labels defined in two places. Note that this is an area where beginners tend to make a mistake. The labels defined under the template section are the labels configured on the pods. The labels you see at the top are the labels of the replica set. We are really concerned about label in template section as of now,

because we are trying to get the replicaset to discover the pods. On creation, if the labels match, the replicaset is created successfully.



- It works the same for other objects like a service. When a service is created, it uses the selector defined in the service definition file to match the labels set on the pods in the replicaset-definition/deployment-definition file.



- Finally let's look at **annotations**. While labels and selectors are used to group and select objects, annotations are used to record other details for informatorily purpose. For example tool details like name, version build information etc or contact details, phone numbers, email ids etc, that may be used for some kind of integration purpose.

replicaset-definition.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
  annotations:
      buildversion: 1.34
spec:
  replicas: 3
  selector:
    matchLabels:
      app: App1
  template:
    metadata:
      labels:
        app: App1
        function: Front-end
    spec:
      containers:
      - name: simple-webapp
        image: simple-webapp
```

Jobs - imperative

- By default k8s will make sure pod/container is always up and running. Like db server , web app. But there are some tasks which are short live like copy images from one folder to other. Or do some image processing. So what we want is whenever that particular task completes container should exit. But k8s container does not work like that, it will restart it again and same computation will be done again. And this continuous to happen until a threshold is reached. This is due to restartPolicy whose default value is 'Always'. To solve that we can use other values like Never or OnFailure. It kind of resolve our issue.

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: math-pod
spec:
    containers:
      - name: math-add
        image: ubuntu
        command: ['expr', '3', '+', '2']

    restartPolicy: Never
```

- But for complex cases like We have large data sets that requires multiple pods to process the data in parallel or in sequence. and we want to make sure that all PODs perform the task assigned to them successfully and then exit. So we need a manager that can create as many pods as we want to get a work done and ensure that the work get done successfully.
- That is what **JOBs** in Kubernetes do. While a ReplicaSet is used to make sure a specified number of PODs are running at all times, a Job is used to run a set of PODs to perform a given task to completion. Let us now see how we can create a job.

```
C:\Users\kush.gupta>kubectl create job addjob --image=ubuntu --dry-run=client -o yaml>job.yaml

C:\Users\kush.gupta>vi job.yaml
```

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: addjob
  labels:
    operation: add
spec:
  template:
    metadata:
        labels:
           app: myapp
    spec:
      containers:
      - image: ubuntu
        name: addjob
        command : ['expr','4','+','5']
      restartPolicy: Never
```

```
C:\Users\kush.gupta>kubectl get all
NAME                READY   STATUS      RESTARTS   AGE
pod/addjob-qh7hv    0/1     Completed   0          7s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.44.0.1    <none>        443/TCP   247d

NAME                COMPLETIONS   DURATION   AGE
job.batch/addjob    1/1           5s         9s

C:\Users\kush.gupta>kubectl logs addjob-qh7hv
9
```

```
C:\Users\kush.gupta>kubectl delete job addjob
job.batch "addjob" deleted
```

- By kubectl get pods command. We see that it is in a completed state with 0 Restarts, indicating that kubernetes did not try to restart the pod.
- So we just ran one instance of the pod in the previous example. To run multiple pods, we set a value for completions under the job specification. And we set it to 3 to run 3 PODs. Now, by default, the PODs are created one after the other. The second pod is created only after the first is finished.

```
C:\Users\kush.gupta>kubectl get job
NAME     COMPLETIONS   DURATION   AGE
addjob   1/3           5s         5s

C:\Users\kush.gupta>kubectl get job
NAME     COMPLETIONS   DURATION   AGE
addjob   3/3           10s        10s
```

- what if the pods fail? For example. When I create this job, first pod completes successfully, the second one fails, so a third one is created and that completes successfully and the fourth one fails, and so does the fifth one and so to have 3 completions, the job creates a new pod which happen to complete successfully. And that completes the job.
- There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so, set .spec.backoffLimit to specify the number of retries before considering a Job as failed. The back-off limit is set by default to 6. Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s ...) capped at six minutes. The back-off count is reset when a Job's Pod is deleted or successful without any other Pods for the Job failing around that time.
- Instead of getting the pods created sequentially we can get them created in parallel. For this add a property called parallelism to the job specification. We set it to 3 to create 3 pods in parallel. So the job first creates 3 pods at once. And if Two of which completes successfully. So we only need one more, so it's intelligent enough to create one pod now, until we get a total of 3 completed pods.

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: addjob
  labels:
    operation: add
spec:
  backoffLimit: 15
  completions: 3
  parallelism: 3
  template:
    metadata:
        labels:
           app: myapp
    spec:
      containers:
      - image: ubuntu
        name: addjob
        command : ['expr','4','+','5']
      restartPolicy: Never
```

- If you want to change something in already created job then in such case you need to delete existing joba dn then create new job. It is not dynamic like deployment. Same for cronjob.

CronJobs (cj)-imperative

- If just like linux cronjob you want to create a job and schedule it say to be executed daily at 10pm. In such case we can create cronjob.

```
C:\Users\kush.gupta>kubectl create cronjob cronjob1 --image=ubuntu --dry-run=client --schedule "*/1 * * * *" -o yaml > cron.yaml
```

- The apiVersion as of today is batch/v1beta1. The kind is CronJob with a capital C and J. Under spec you specify a schedule. The schedule option takes a cron like format string where you can specify the time when the job is to be run. Then you have the Job Template, which is the actual job that should be run. Move all of the content from the spec section of the job definition under this. Notice that the cronjob definition now gets a little complex. So you must be extra careful. There are now 3 spec sections, one for the cron-job, one for the job and one for the pod.

```yaml
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cronjob1
  labels:
    operation: add
spec:
  jobTemplate:
    metadata:
      labels:
        operation: add
      name: cronjob1
    spec:
      template:
        metadata:
          labels:
            app: myapp
        spec:
          containers:
          - image: ubuntu
            name: cronjob1
          restartPolicy: OnFailure
  schedule: '*/1 * * * *'
```

```
C:\Users\kush.gupta>kubectl create -f cron.yaml
cronjob.batch/cronjob1 created

C:\Users\kush.gupta>kubectl get all
NAME                              READY   STATUS      RESTARTS   AGE
pod/cronjob1-1651863900-drjsn     0/1     Completed   0          3s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.44.0.1    <none>        443/TCP   247d

NAME                              COMPLETIONS   DURATION   AGE
job.batch/cronjob1-1651863900     1/1           3s         6s

NAME                      SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob.batch/cronjob1    */1 * * * *   False     1        6s              18s
```

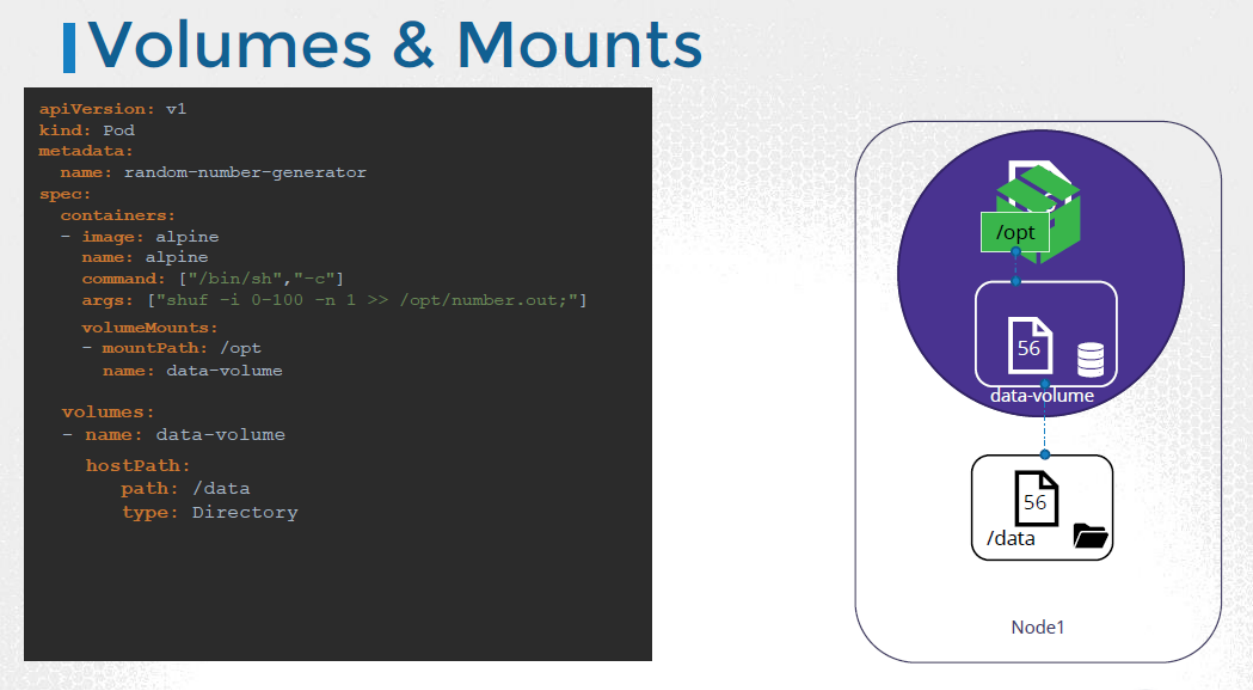
```
C:\Users\kush.gupta>kubectl describe cronjob cronjob1
Name:                          cronjob1
Namespace:                     default
Labels:                        operation=add
Annotations:                   <none>
Schedule:                      */1 * * * *
Concurrency Policy:            Allow
Suspend:                       False
Successful Job History Limit:  3
Failed Job History Limit:      1
Starting Deadline Seconds:     <unset>
Selector:                      <unset>
Parallelism:                   <unset>
Completions:                   <unset>
Pod Template:
  Labels:  app=myapp
  Containers:
   cronjob1:
    Image:          ubuntu
    Port:           <none>
    Host Port:      <none>
    Environment:    <none>
    Mounts:         <none>
  Volumes:          <none>
Last Schedule Time:  Sat, 07 May 2022 01:17:00 +0530
Active Jobs:         <none>
Events:
  Type    Reason            Age   From                Message
  ----    ------            ----  ----                -------
  Normal  SuccessfulCreate  18s   cronjob-controller  Created job cronjob1-1651866420
  Normal  SawCompletedJob   8s    cronjob-controller  Saw completed job: cronjob1-1651866420, status: Complete
```

```
C:\Users\kush.gupta>kubectl get cronjob
NAME       SCHEDULE      SUSPEND   ACTIVE   LAST SCHEDULE   AGE
cronjob1   */1 * * * *   False     0        32s             2m44s

C:\Users\kush.gupta>kubectl delete cronjob cronjob1
cronjob.batch "cronjob1" deleted
```

State Persistence

Volumes

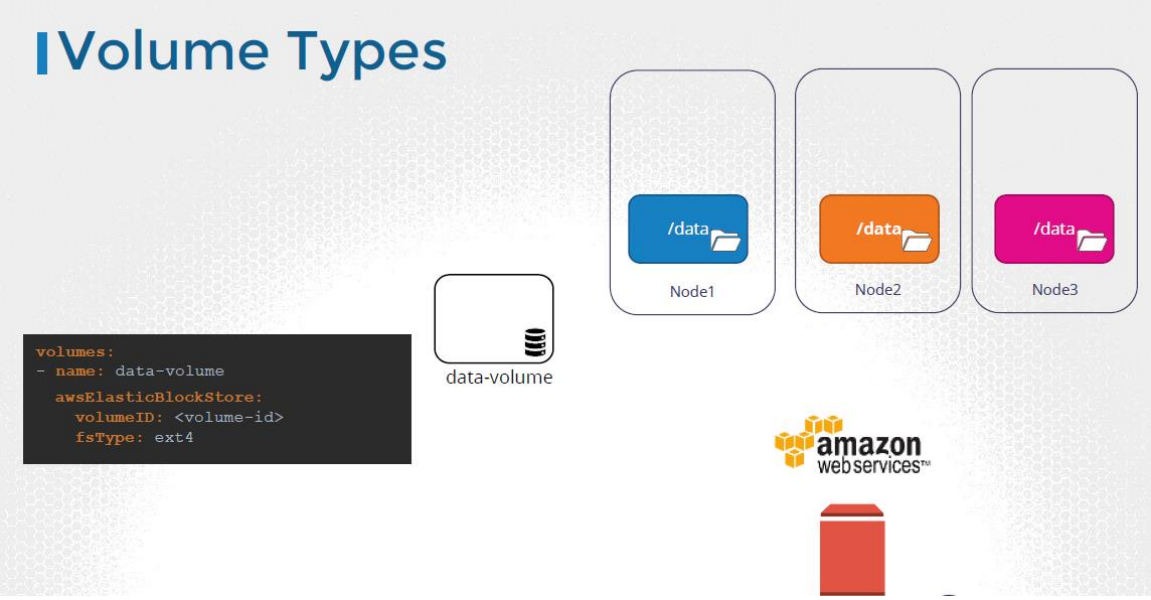
- Docker container are transient in nature. It means the data is destroyed when the container destroyed.
- To persist data processed by the containers, we attach a volume to the containers when they are created. The data processed by the container is now placed in this volume, thereby retaining it permanently. Since data is outside container. So, even if the container is deleted, the data generated or processed by it remains.
- Just as in Docker, the PODs created in Kubernetes are transient in nature. When a POD is created to process data and then deleted, the data processed by it gets deleted as well. For this we attach a volume to the POD. The data generated by the POD is now stored in the volume, and even after the POD is delete, the data remains.
- When you create a volume you can chose to configure it's storage in different ways. We will look at the various options in a bit, but for now we will simply configure it to use a directory on the host. In this case I specify a path /data on the host. This way any files created in the volume would be stored in the directory data on my node.
- Once the volume is created, to access it from a container we mount the volume to a directory inside the container. We use the volumeMounts field in each container to mount the data-volume to the directory /opt within the container. The data will now be written to /opt mount inside the container, which happens to be on the data-volume which is in fact /data directory on the host. When the pod gets deleted, the file still lives on the host.



- Let's take a step back and look at the Volume Storage option. We just used the hostPath option to configure a directory on the host as storage space for the volume. Now that works on a single node.
- However it is not recommended for use in a multi-node cluster. This is because the PODs would use the /data directory on all the nodes, and expect all of them to be the same and have the same data. Since they are on different servers, they are in fact not the same, unless you configure some kind of external replicated clustered storage solution.

Volumes Types

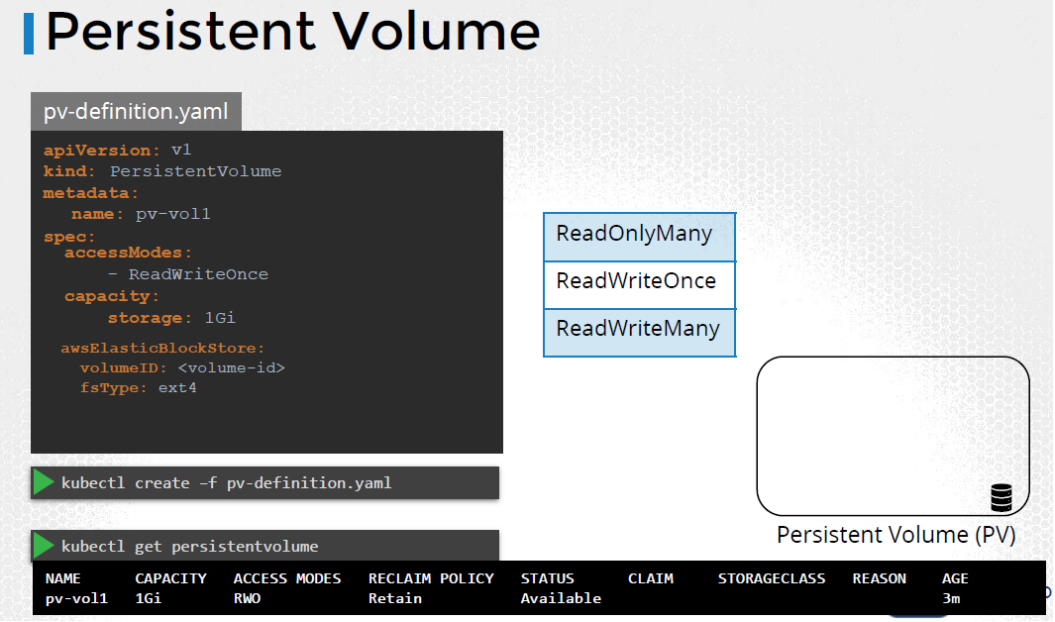
- Kubernetes supports several types of standard storage solutions such as NFS, glusterFS, Flocker, FibreChannel, CephFS, ScaleIO or public cloud solutions like AWS EBS, Azure Disk or File or Google's Persistent Disk.



- For example, to configure an AWS Elastic Block Store volume as the storage or the volume, we replace hostPath field of the volume with awsElasticBlockStore field along with the volumeID and filesystem type. The Volume storage will now be on AWS EBS.

Persistent Volumes -pv -no

- When we created volumes in the previous section we configured volumes within the POD definition file. So every configuration information required to configure storage.
- Now, when you have a large environment with a lot of users deploying a lot of PODs, the users would have to configure storage every time for each POD. Whatever storage solution is used, the user who deploys the PODs would have to configure that on all POD definition files in his environment. Every time a change is to be made, the user would have to make them on all of his PODs.
- You would like it to be configured in a way that an administrator can create a large pool of storage, and then have users carve out pieces from it as required. That is where **Persistent Volumes** can help us. **A Persistent Volume is a Cluster wide pool of storage volumes configured by an Administrator, to be used by users deploying applications on the cluster**. **The users can now select storage from this pool using Persistent Volume Claims.**
- Let us now create a Persistent Volume. We start with the base template and update the apiVersion, set the Kind to PersistentVolume, and name it pv-vol1. Under the spec section specify the accessModes.
- Access Mode defines how the Volume should be mounted on the hosts. Weather in a ReadOnlymode, or ReadWritemode. The supported values are ReadOnlyMany, ReadWriteOnce or ReadWriteManymode.
- Next, is the capacity. Specify the amount of storage to be reserved for this Persistent Volume. Which is set to 1GB here.
- Next comes the volume type.

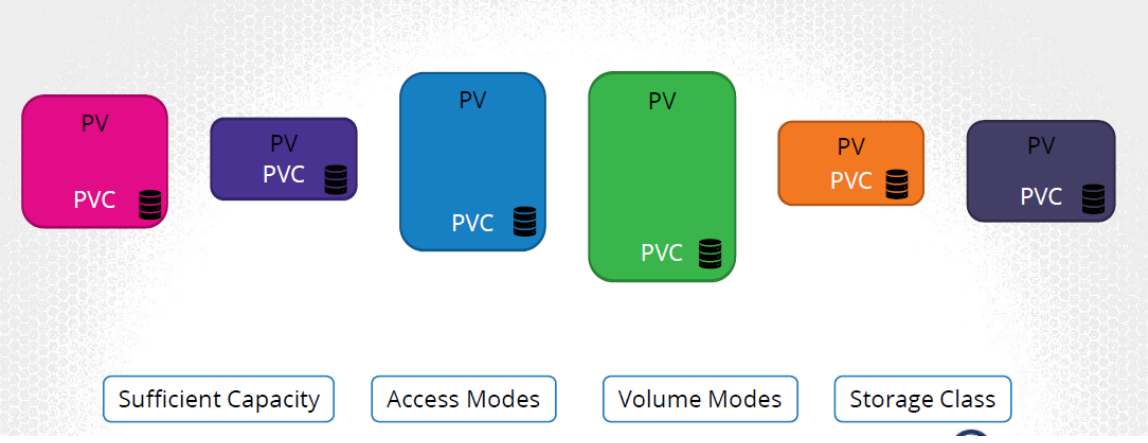


Another example

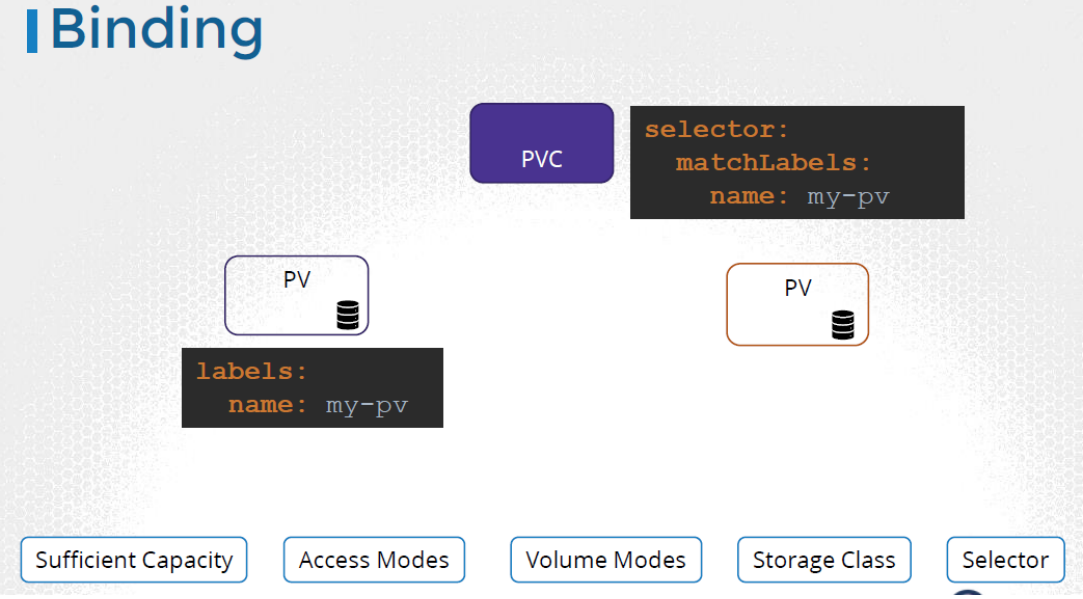
```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-log
spec:
  capacity:
    storage: 100Mi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  hostPath:
      path: /pv/log
```

Persistent Volume Claims - pvc -no

- In the previous lecture we created a Persistent Volume. Now we will create a Persistent Volume Claim to make the storage available to a node.
- Persistent Volumes and Persistent Volume Claims are two separate objects in the Kubernetes namespace. An Administrator creates a set of Persistent Volumes and a user creates Persistent Volume Claims to use the storage. Once the Persistent Volume Claims are created, Kubernetes binds the Persistent Volumes to Claims based on the request and properties set on the volume.
- Every Persistent Volume Claim is bound to a single Persistent volume. During the binding process, kubernetes tries to find a Persistent Volume that has sufficient Capacity as requested by the Claim, and any other requested properties such as Access Modes, Volume Modes, Storage Class etc.

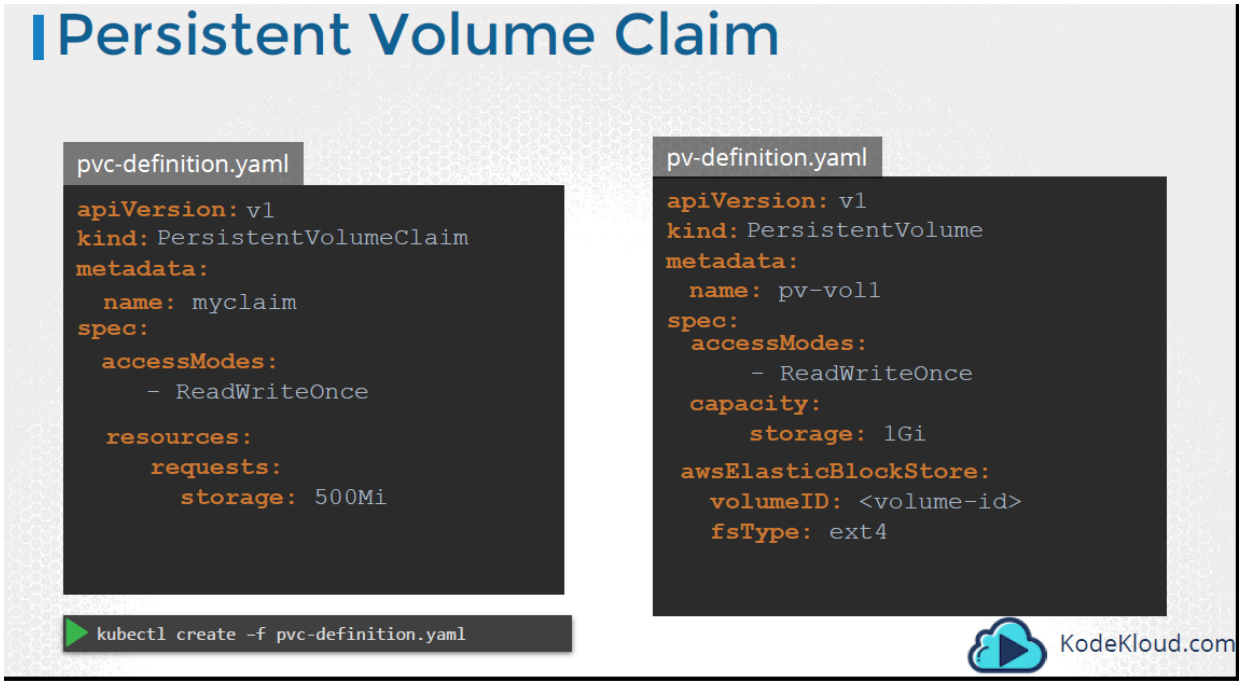


- However, if there are multiple possible matches for a single claim, and you would like to specifically use a particular Volume, you could still use labels and selectors to bind to the right volumes.



- Finally, note that a smaller Claim may get bound to a larger volume if all the other criteria matches and there are no better options. There is a one-to-one relationship between Claims and Volumes, so no other claim can utilize the remaining capacity in the volume. If there are no volumes available the Persistent Volume Claim will remain in a pending state, until newer volumes are made available to the cluster. Once newer volumes are available the claim would automatically be bound to the newly available volume.
- Let us now create a Persistent Volume Claim. We start with a blank template. Set the apiVersion to v1 and kind to PersistentVolumeClaim. We will name it myclaim. Under specification set the accessModes to ReadWriteOnce. And set resources to request a storage of 500 mega bytes. Create the claim using kubectlcreate command.
- To view the created claim run the kubectl get persistentvolumeclaim command. We see the claim in a pending state.

- When the claim is created, kubernetes looks at the volume created previously. The access Modes match. The capacity requested is 500 Megabytes but the volume is configured with 1 GB of storage. Since there are no other volumes available, the PVC is bound to the PV.



- When we run the get volumes command again, we see the claim is bound to the persistent volume we created. Perfect!

- kubectl delete persistentvolumeclaim pvcName
  But what happens to the Underlying Persistent Volume when the claim is deleted? You can choose what is to happen to the volume. By default, It is set to Retain. Meaning the Persistent Volume will remain until it is manually deleted by the administrator. It is not available for re-use by any other claims. Or it can be Deleted automatically. This way as soon as the claim is deleted, the volume will be deleted as well. Or a third option is to recycle. In this case the data in the volume will be scrubbed before making it available to other claims.
- What will happen if we delete pvc and it is already in use by running pod? In such case pvc will remain in terminating state and once the pod is deleted then pvc will be deleted (no need to run delete pvc command again as it is already run and in terminating state). After that pv will be moved to state as mentioned in above point.
- Once you create a PVC use it in a POD definition file by specifying the PVC Claim name under persistentVolumeClaim section in the volumes section like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

- The same is true for ReplicaSets or Deployments. Add this to the pod template section of a Deployment on ReplicaSet.
- https://kubernetes.io/docs/concepts/storage/persistent-volumes/#claims-as-volumes

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim-log-1
spec:
  accessModes:
    - ReadWriteMany
  volumeMode: Filesystem
  resources:
    requests:
      storage: 50Mi
```

Storage Class (NA) – sc - No

- Earlier we created pv which maps the actual volume available on external nodes. It is usually created by admin. Then out of all volume available, users create pvc to claim volumes available as per pv. Then finally claim is attached to pod. For example suppose we have 1gb of compute disk storage available in gcp which we have purchased and then pv is created which contains same details as disk created on gcp. Then via pvc user claimed 200mb out of it. This whole process is **static provisioning**. As here we have statically purchased disc first of 1gb and then created pv for that disc.
- By using storage class k8s object we can dynamically ask gcp to give us storage as per need when claim is made. In such case we don't need to create pv object. Storage class will be used and then using pvc volume is request by user which will be verified from storageclass not from pv. And then pod will be bind to pvc.
- Internally storage class will create pv for us. But we don't need to create pv object.
- In pvc definition file provide storageClassName in spec. this matches with the storageclass object created.
- All public cloud vendors provide dynamic support for pvc mapping. They all have options specific to vendor which they take via parameters. Like gcp can take param as type of volume(standard or ssd) and replication-type (regional-pd or none).
- kubectl get sc

```
controlplane ~ ➜ kubectl get sc
NAME                   PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUM
EEXPANSION   AGE
local-path (default)   rancher.io/local-path   Delete          WaitForFirstConsumer   false
             4m35s
```

- Local volumes do not currently support dynamic provisioning, however a StorageClass should still be created to delay volume binding until Pod scheduling. This is specified by the WaitForFirstConsumer volume binding mode. Delaying volume binding allows the scheduler to consider all of a Pod's scheduling constraints when choosing an appropriate PersistentVolume for a PersistentVolumeClaim.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer
```

- We can still create pv and provide storageclass name there in spec. and then while creating pvc specify same storage class name. although pv is not needed as sc can directly be used in pvc and then in pod specifiy pvc. But it is possible to create pv also and link it to sc->

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 500Mi
  local:
    path: /opt/vol1
  persistentVolumeReclaimPolicy: Retain
  storageClassName: local-storage
  volumeMode: Filesystem
```

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: local-pvc
spec:
  storageClassName: local-storage
  resources:
    requests:
      storage: 500Mi
  accessModes:
    - ReadWriteOnce
```

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

Stateful Sets (NA)

- It is similar to deployment. But by it we can provide feature like below which is not possible by deployment.
- Can order pod creation. Like pod2 will be only created after pod 1 created.
- Make static name of pods which is unique. By it we can implement master slave model where write will be done on master pod only and read can be done from slaves. And pod2 can sync from pod1 also. And suppose if first pod goes down then new pod will be created with same name.
- If no of pods in stateful set scaled up then newly created pod will be in order and will take sync from last node. And on scale down, pod will be deleted from last so that index will be maintained in order from 0.
- Watch video of *Mumshad* for complete detail of config

Define, Build and Modify container images

- You can define docker images in DockerFile and Build a docker image.
  docker build -t webapp-color .
  docker build -t webapp-color:lite .

- Docker images are build layer wise. So suppose if any of the layer failed while building image. Process will not start from scratch.
- Docker container ps -a list all stopped and running containers.
- Docker image list all images.
- Find base Operating System used by the python:3.6 image
  docker run python:3.6 cat /etc/*release*
- To list down image by name.
  docker images webapp-color
- Run an instance of the image webapp-color and publish port 8080 on the container to 8282 on the host.
  docker run -p8282:8080 webapp-color

Authentication, Authorization and Admission Control

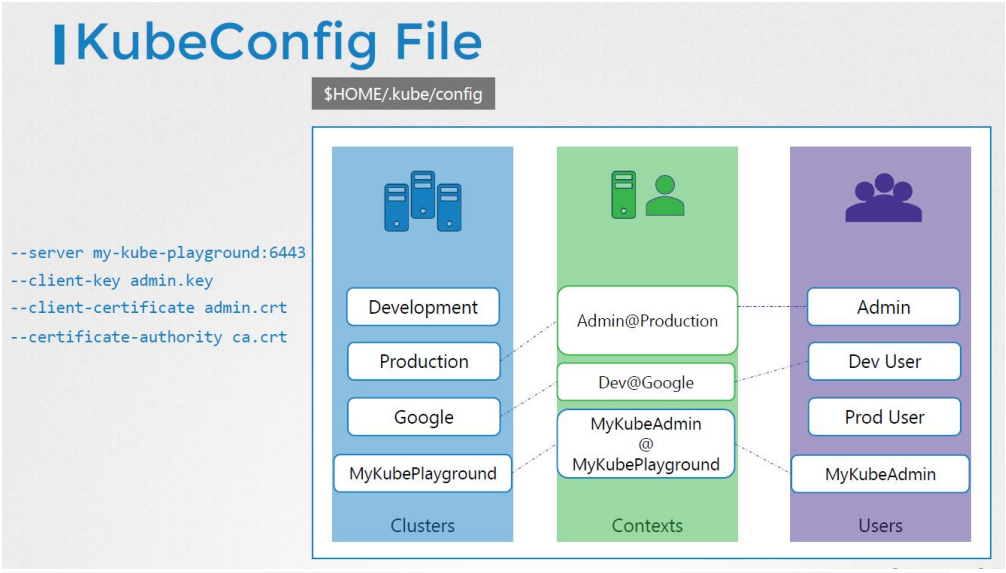
Authentication

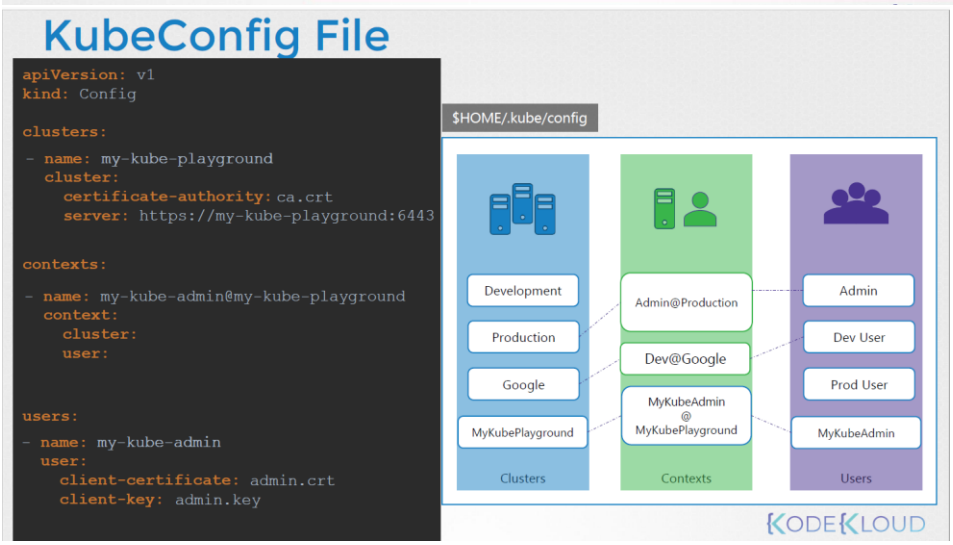
- Make node secure. Example use ssh instead of username and password is one way.
- kube-api server is the api used to access cluster and perform operation on cluster. So it is very imp to protect it.
- Who can access the kube-api server is defined by authentication. We can use different authentication mechanism to secure it. Like Files – Username and Passwords, Username and tokens, certificates, or external authentication providers - ldap solution.
- What operations/commands can user run via kubeapi-server is controlled by Authorization. And by default pods can communicate with each other. To restrict it network policy can be used. For authorization we can do node authorization, RBAC, ABAC and webhook Mode.
- All communication between various architecture component should be secured via tls certificate.
- We have 4 types of users-> admin who perform administrative task on the cluster, Developers who test and deploy applications, end users who uses application deployed on cluster and 3rd party application who uses kube-api server to access cluster for integration purposes.
- For end user authentication security is maintained by application itself internally and we don't need to handle that. For 3rd party integration like Prometheus, we create **serviceAccount**. And for users like admin and developer **user accounts** are used.
- Service account is already discussed in earlier section and it provide access control to 3rd party app.
- Kube-api server is the way by which any user can interact with cluster either via kubectl or via ui of public cloud or any other way. For authentication of users you can create static password file or static token file, certificates or external identity services.
- For first options. Create csv file with password, username and userid and pass it to kube-apiserver.service file. For 2nd options create csv with token, password, username and userid. Then user when access it can provide username/password for 1st option and auth bearer token in request.
- The first 2 options are not recommended to use as they are stored in plain text file and are static. These basic authentication mechanism is deprecated in 1.19 Kubernetes release and no longer available.

Kube-config

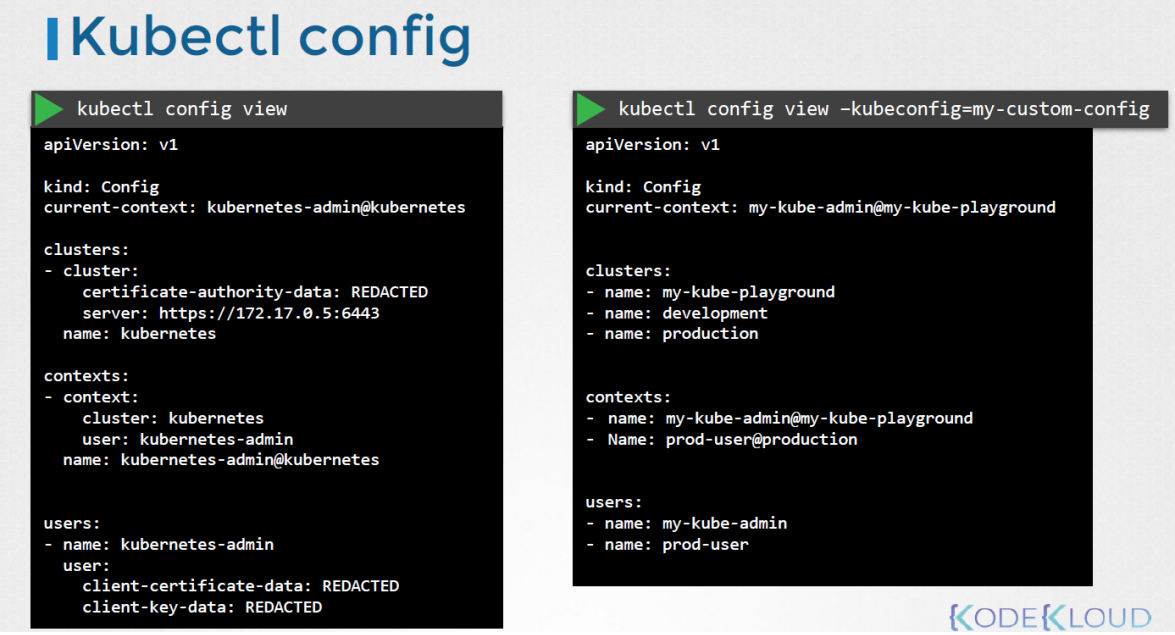
- For security to pass config like server, client-key, client-certificate and certificate-authority in every kubectl command is not easy. For that we can provide these info in kube config file and place it in /root/.kube/config. Now we don't need to write anything in kubectl command. like kubectl get pods.
- Why it works? Because by-default Kubernetes looks for kube-config into default path for entries. And hence not needed. But if we want to use kube-config file from some other place. Then provide file name in every command.
  kubectl get pods --kubeconfig config
- By using above command we can securely connect to cluster via kube-api.
- Kube config file has 3 sections – cluster, users and contexts.
- In cluster we can specify all the cluster available and in users all the users created. Context defines which user account will be used to access cluster. i.e. bind users with cluster. Like admin@development binding tell admin user can access development cluster.

- Server info goes into cluster and client-key, client-certificate and certificate authority goes into users.
- Kubeconfig file has structure as apiVersion: v1 , kind: Config and 3 sections clusters, contexts, users. All these 3 takes array.





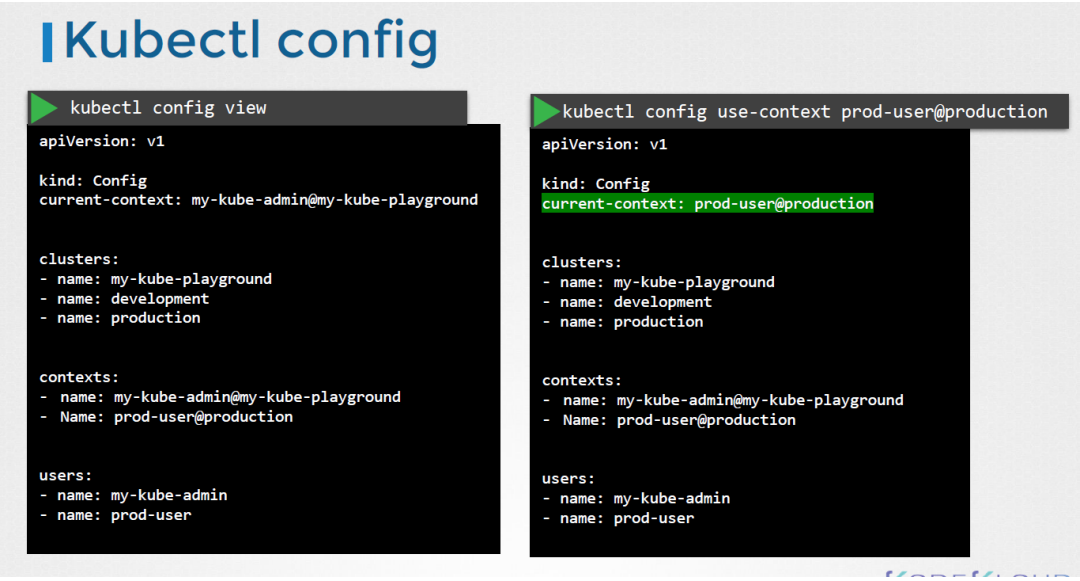
- Unlike other Kubernetes objects we does not create config object explicitly.it is read by kubectl command and referred on start up.
- How does kubectl knows which context to use as default. We can add field current-context parallel to kind in config file where we specify default context.
- kubectl config view . it will list down config file content at default location
- kubectl config view --kubeconfig my-kube-config-file. it will list down config file content at specified location.



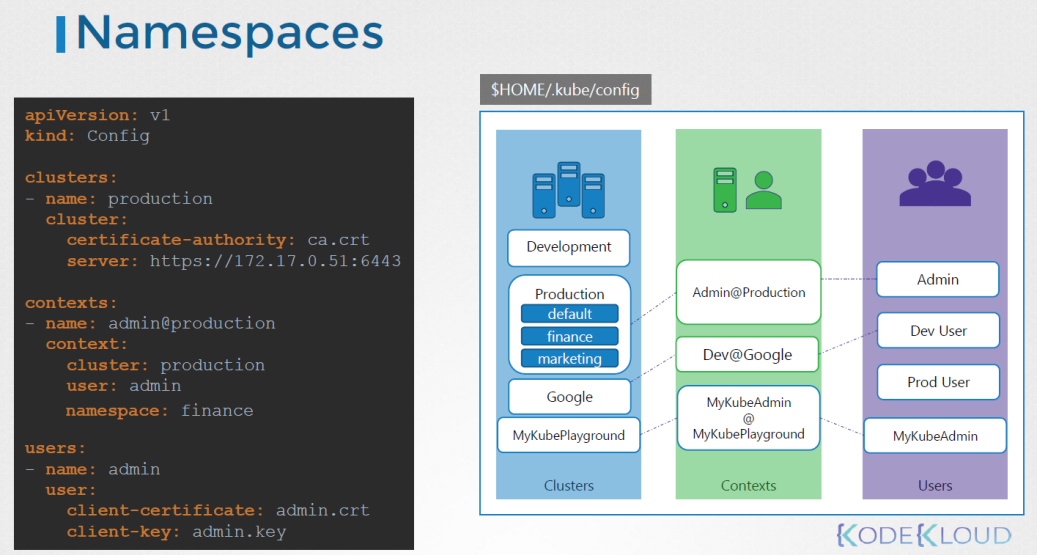
- To change current context, use command -> kubectl config use-context prod-user@production and to check current context->

```
root@controlplane ~ ➜ kubectl config --kubeconfig /root/my-kube-config use-context research
Switched to context "research".

root@controlplane ~ ➜ kubectl config --kubeconfig /root/my-kube-config current-context
research
```

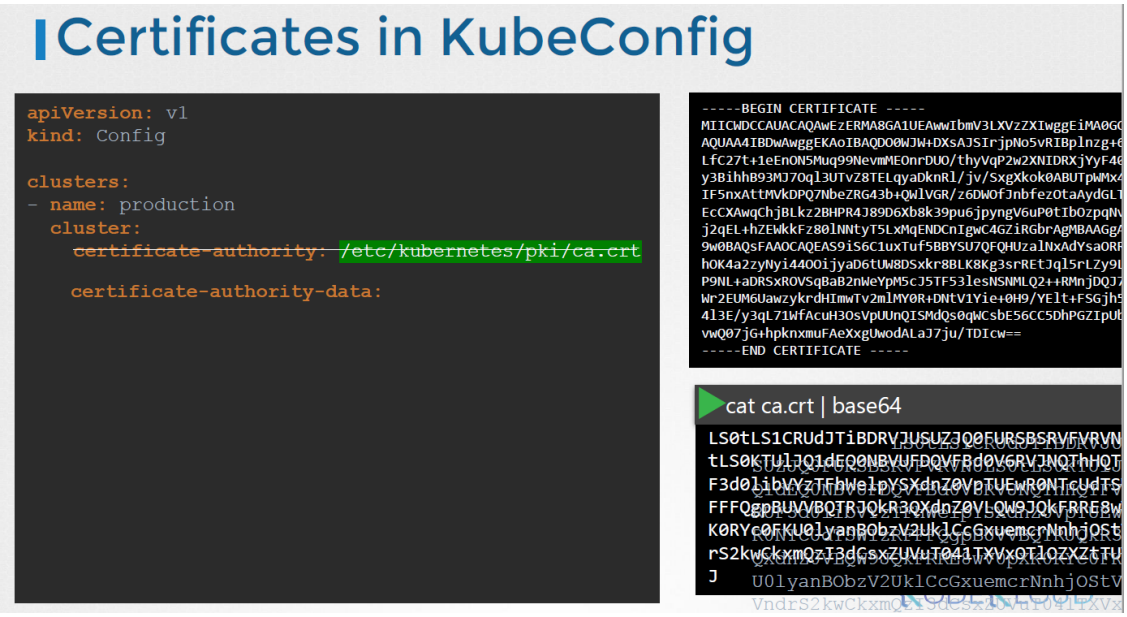
Since above context research is not defind in default kubeconfig file located at .kube folder. We need to provide option of kubeconfig file in each kubectl command. like ->
kubectl get pods --kubeconfig config
To avoid it replace content of /root/.kube/config file with our custom config file.



- To know options use kubectl config -h
- You can also specify namespace in each context in config file. By this when we change content they will point to specific namespace in cluster.

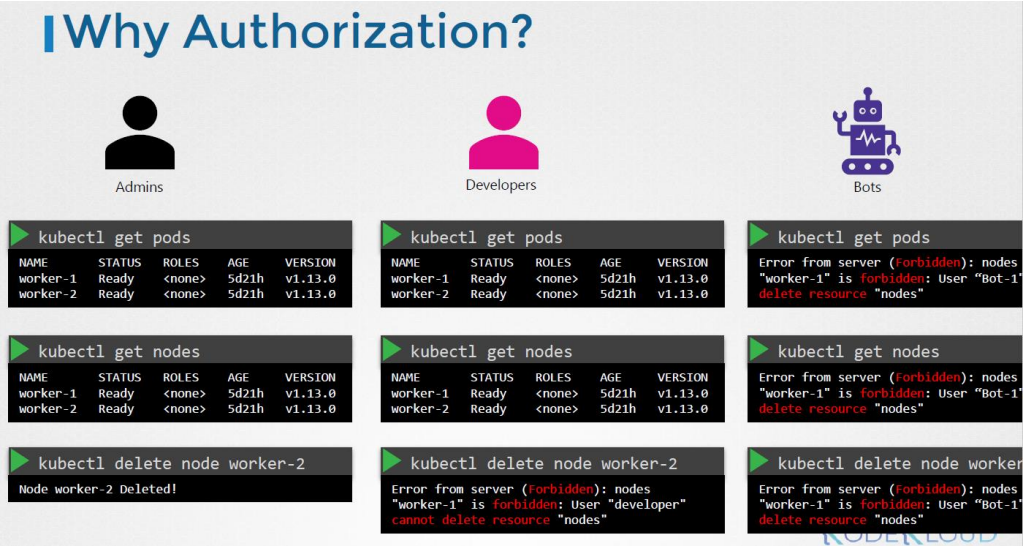


- certificate-authority path entered in kubeconfig file can also be replaced with certificate-authority-data where we can provide base64 encoded content of data. cat ca.crt | base64 will generate base64 encoded value of certificate data.

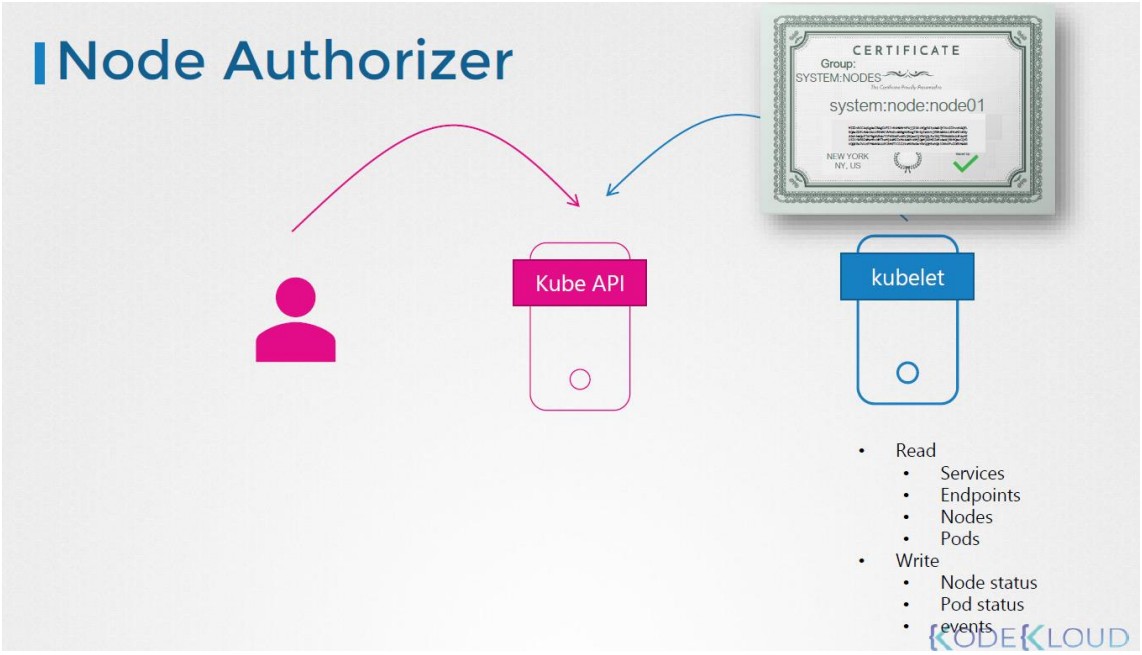


Authorization

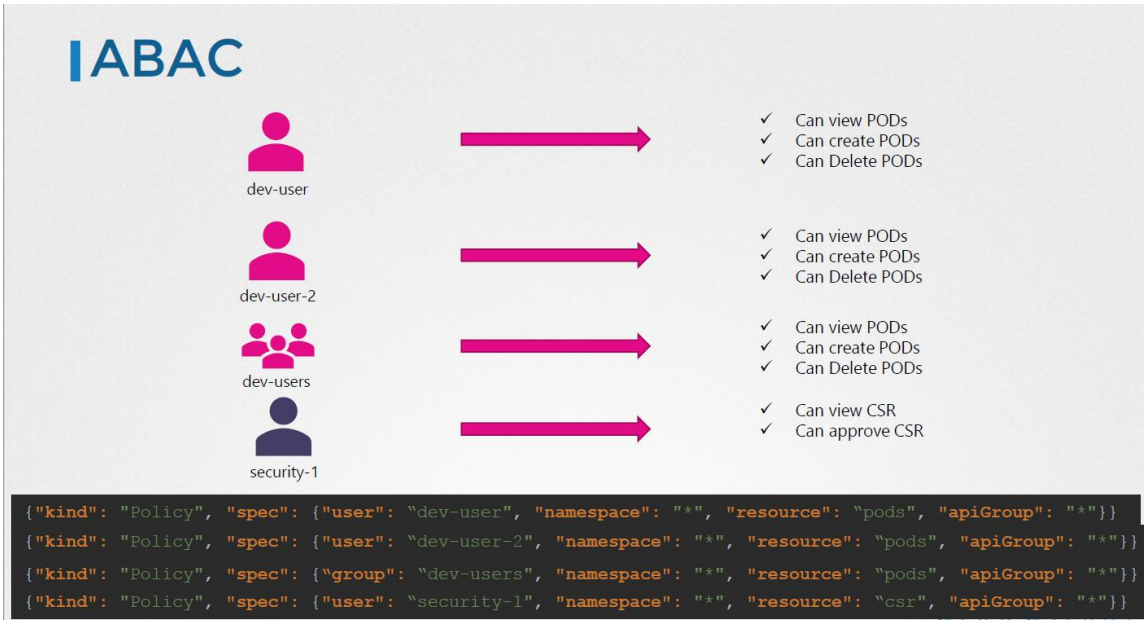
- by default user has admin right by which we can do anything like delete node, deployment, cluster. But we want different level of authorization for different types of users. Like developer, tester, admin etc.



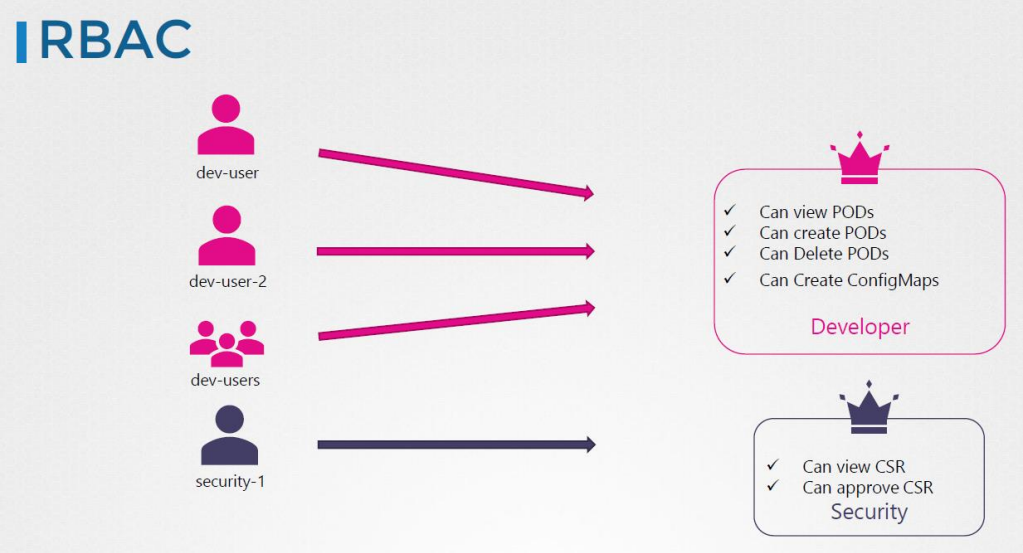
- There are 4 types of authorizations -> Node, Attribute based authorization (ABAC), Role based authorization RBAC and webhook.
- **Node**-> It is used with-in nodes. kubeapi server is invoked by user or by kubelet with-in nodes with-in cluster for management purpose. Kubelet gets info of services, endpoints, nodes, pods from kube-api server and kubelet provide info to kube-api server about node status, pod status and events. These requests are handled by special authorizer known as node authorizer. And usually managed by certificates.



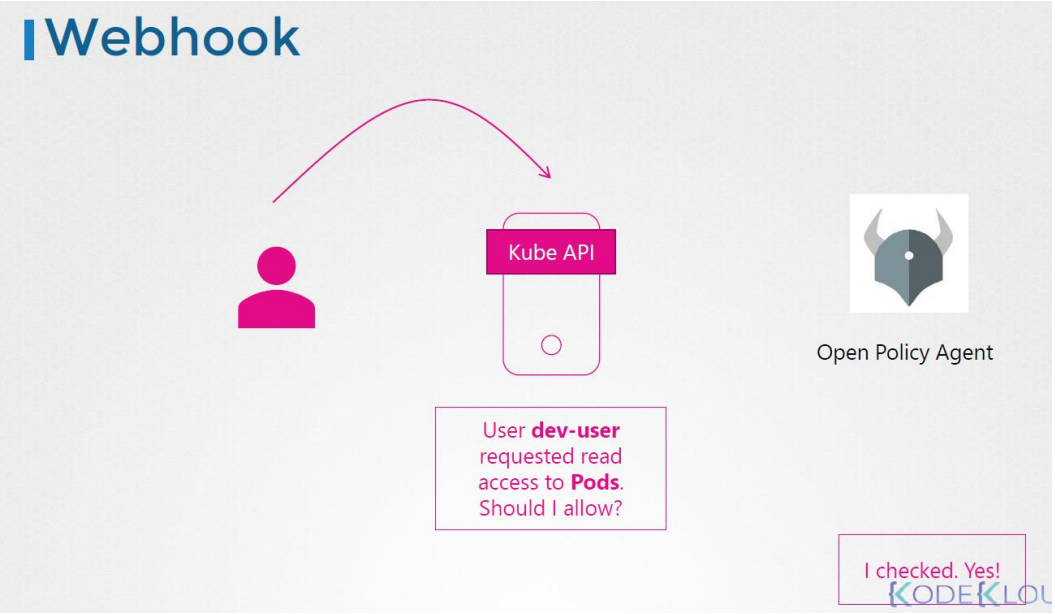
- **ABAC** -> In above points we discuss about kubelet node with-in cluster. For external users like us Attribute based authorization is used to grant access to set of user or groups, like they can view, create pods and delete pod. For that create policy file. And pass this to kube api server and restart it. Now every time you need to add or make changes in security you need to update this file manually and restart kube-api server. Hence abac is tough to maintain.



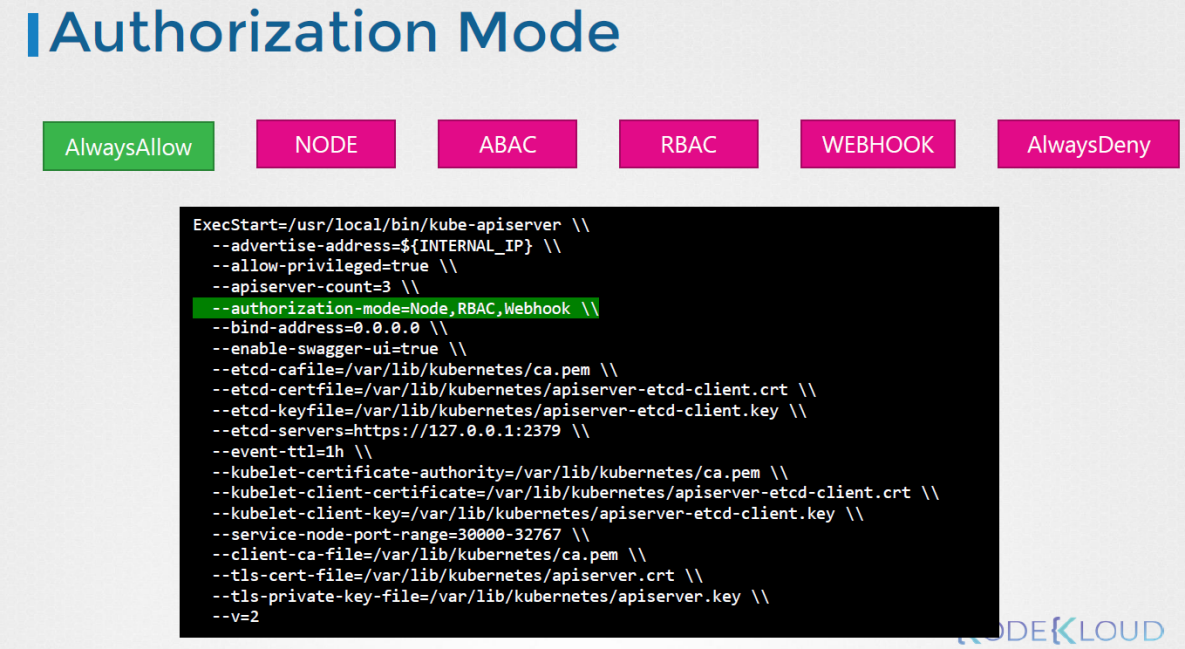
- **RBAC->** instead of directly associating users or user group with set of permissions. We create role and associate permissions to each role. Like developer role can create, delete, view pod. Security role will have manage cluster permissions. And now we can assign role to users or groups. By this in future if we want to give some extra permission to say developers. we can edit role and add 1 more permission and it will be applied to all the associated users or groups.



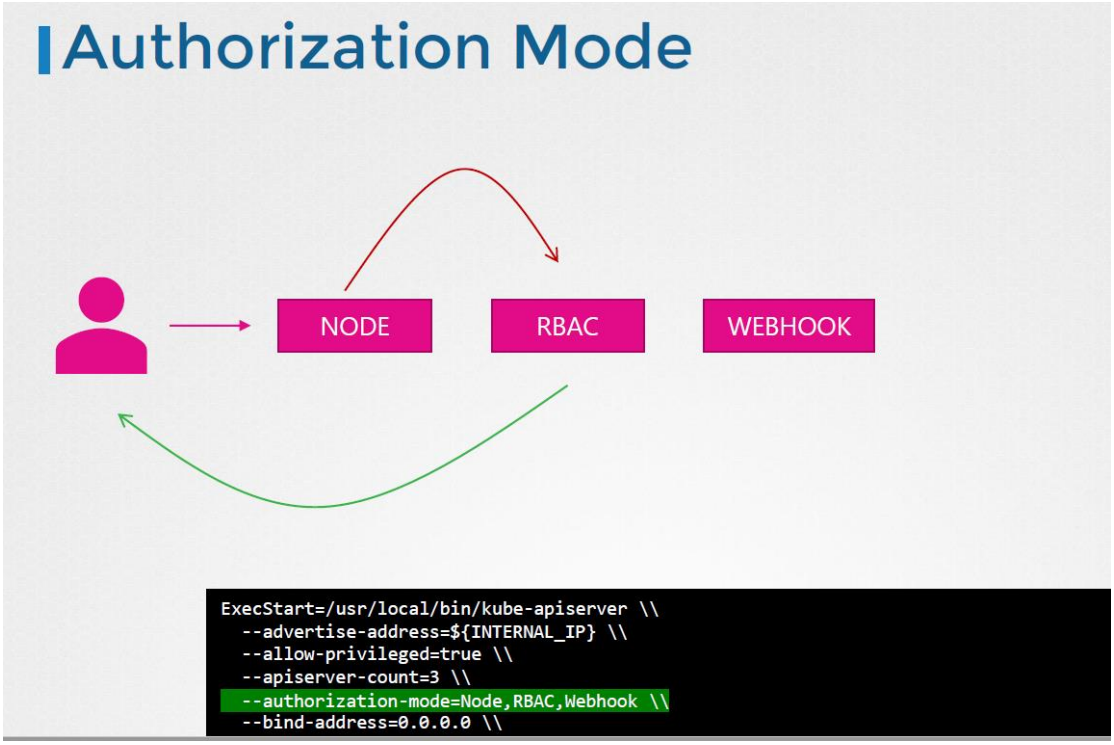
- **Webhook ->** if you want to manage authorization from external solution, Webhook can be used. Like openPolicyAgent is a tool that can provide authorization. So in such case when call goes to kube-api server it will ask access permission from openPolicyAgent with username and action requested to check if user should be allowed. Based on that response user is granted access.



- There are 2 more modes -> **AlwaysAllow** and **AlwaysDeny**. As name suggest they allow or deny all requests to access resource via kube-api server. In kube-api service config file by default AlwaysAllow is mentioned inside --authorization-mode field.
- You can change it value from AlwaysAllow to comma separated value like --authorization-mode=Node,RBAC,Webhook



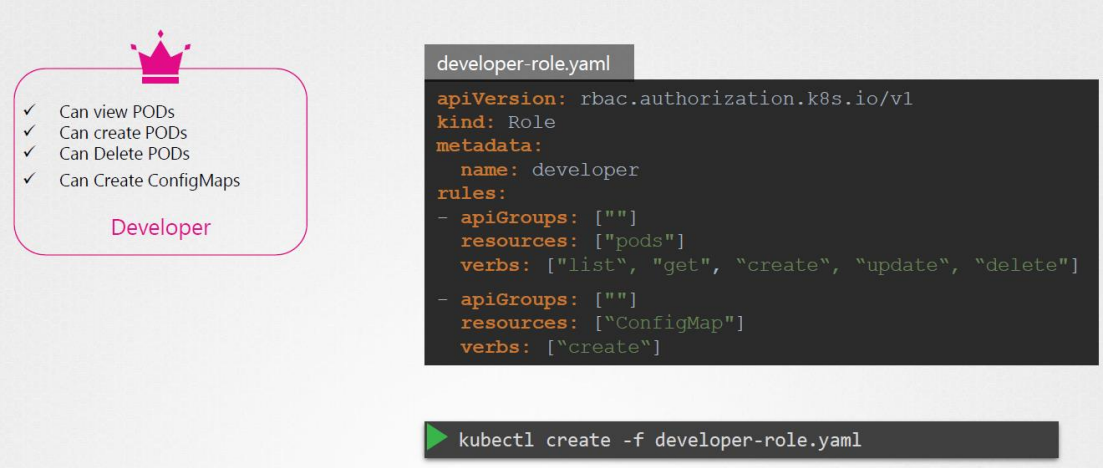
- In such case first request goes to first mechanism if it succeeds it is returned if not it will be forward to next mechanism in chain till last mechanism. If all fail, permission is denied.



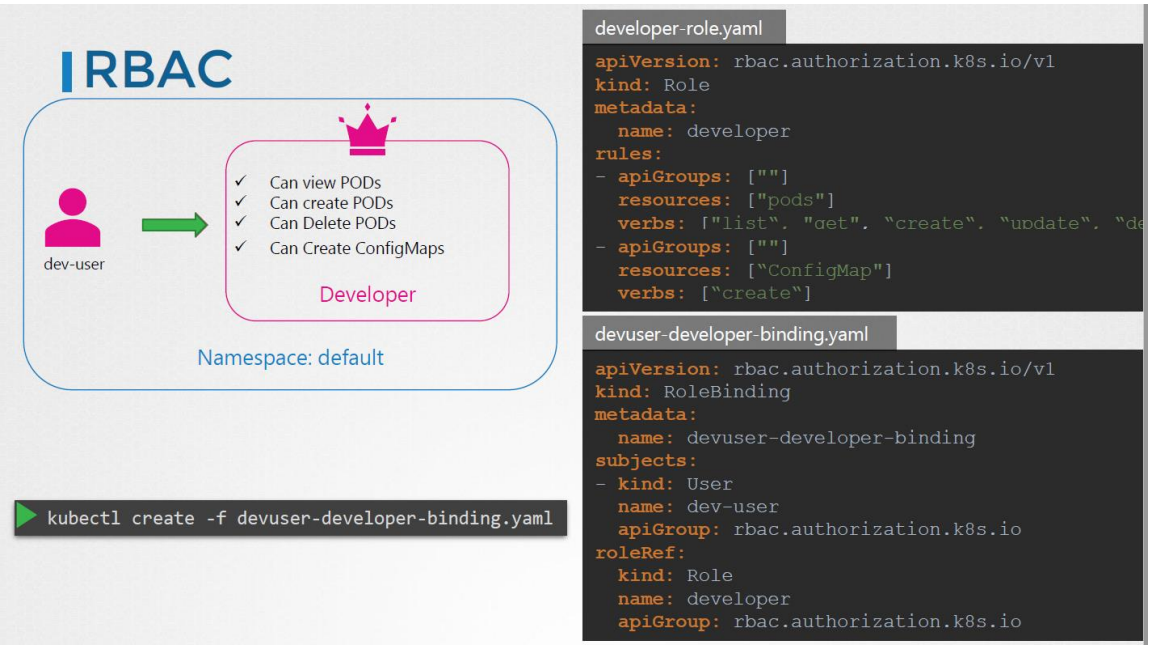
- Example -> user called kubectl get pods command. it go to Node, since Node handles only with-in cluster nodes authorization, it denies then it will go to RBAC if it allow chain is stopped and permission will be granted.
- To check the entries present inside kube-api service file like authorization-mode, type below command -> **kubectl describe pod kube-apiserver-controlplane -n kube-system**

RBAC

- Now we create role definition file and then use create command to create role object.
- Below example we create role developer and provides rules to it. apiGroups for core api is blank. resources contains resources which we want to provide access like pods. And verbs as action which can be allowed. These 3 fields will be explained in api group section.



- Now role is created. We need to assign user to this role. For that we create RoleBinding object. It provides mapping of user and role.



- In rolebinding we map 1 role to multiple users. But we cannot create 1 rolebinding with multiple roles.
- Since namespace is not mentioned in metadata it will be applied to default namespace.
- Kubectl get roles and kubectl get rolebindings to get list

View RBAC

```
kubectl get roles
NAME        AGE
developer   4s
```

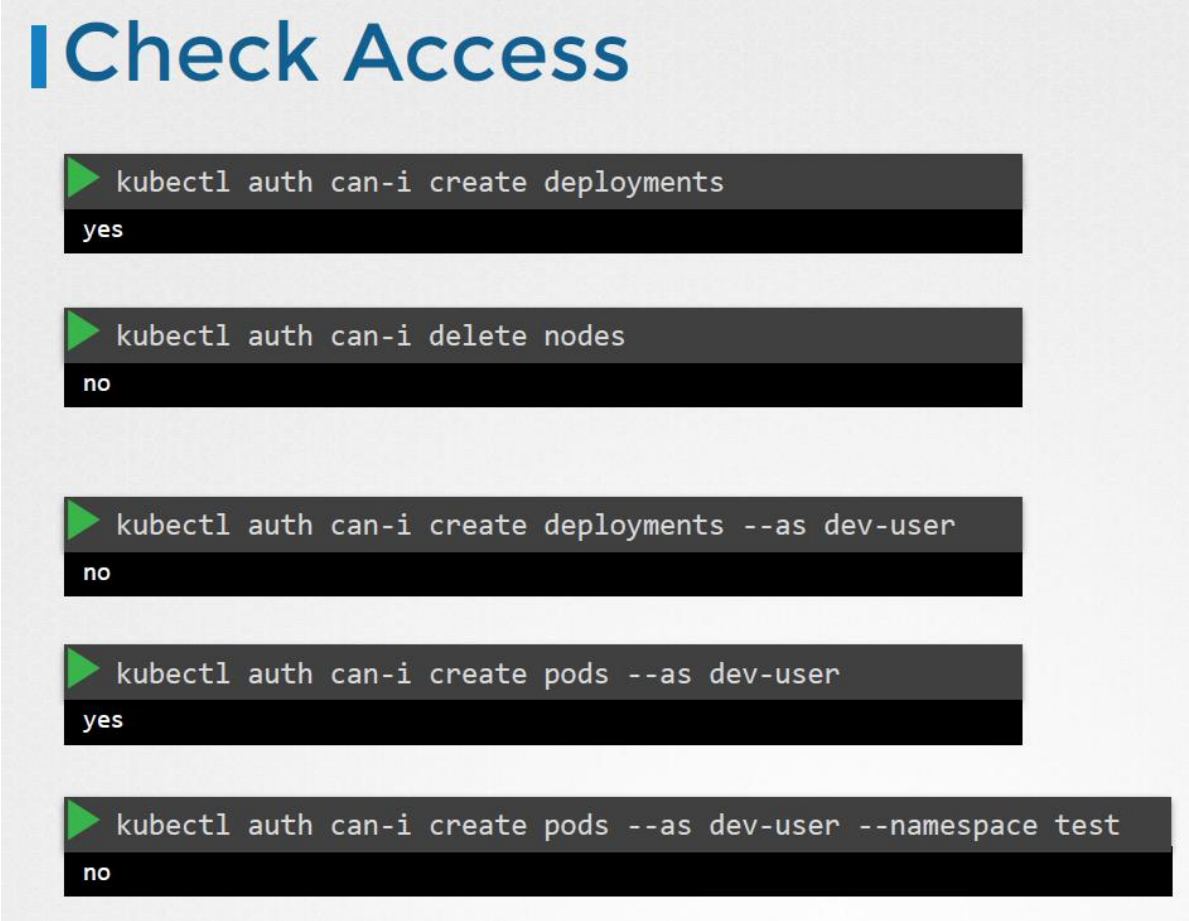
```
kubectl get rolebindings
NAME                        AGE
devuser-developer-binding   24s
```

```
kubectl describe role developer
Name:         developer
Labels:       <none>
Annotations:  <none>
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  ConfigMap  []                 []              [create]
  pods       []                 []              [get watch list create delete]
```

View RBAC

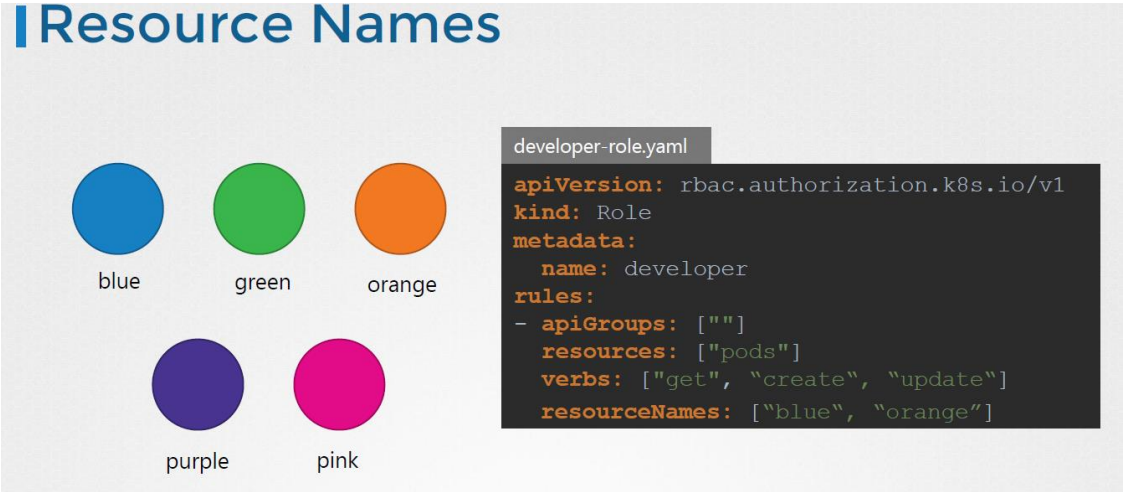
```
kubectl describe rolebinding devuser-developer-binding
Name:         devuser-developer-binding
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  Role
  Name:  developer
Subjects:
  Kind  Name      Namespace
  ----  ----      ---------
  User  dev-user
```

- As a user you want to check what access you have you can type
  kubectl auth can-i create deployments
- You can also check for some other user access values.
  kubectl auth can-i create deployments --as dev-user



- Example – check if dev-user can list pods in default namespace

  kubectl auth can-i get pods --as dev-user
  no
- You can also restrict access to specific pod only. Like out of 5 only blue and orange named pod should be allowed. For that specify resourceNames under rules section parallel to verbs.



Create Role and RoleBinding – role and rolebinding – no

- kubectl create role developer --namespace=default --verb=list,create,delete --resource=pods
- kubectl create rolebinding dev-user-binding --namespace=default --role=developer --user=dev-user

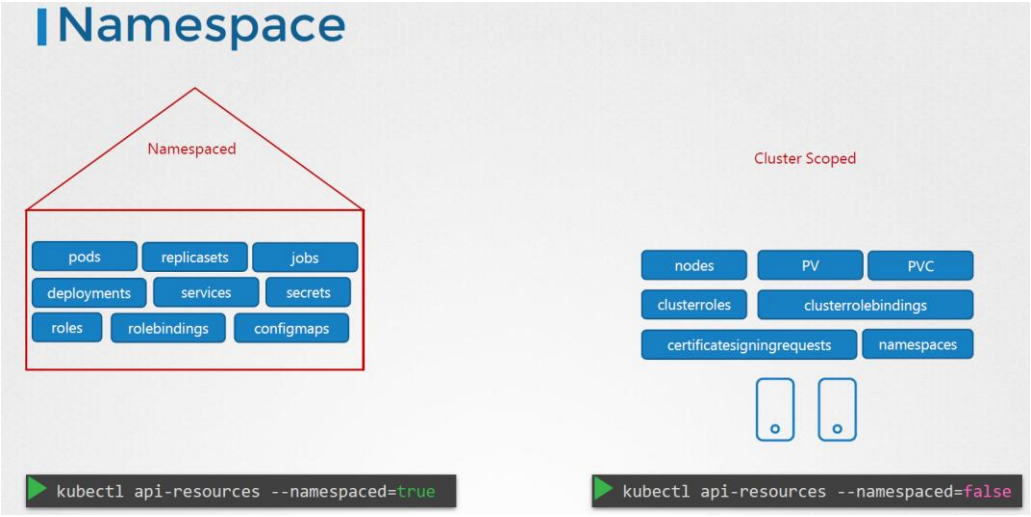
```
root@controlplane ~ ➜ kubectl create role developer --verb=get --resource=pods --dry-run=client
 -o yaml > role.yaml

root@controlplane ~ ➜ vi role.yaml

root@controlplane ~ ➜ kubectl create -f role.yaml
role.rbac.authorization.k8s.io/developer created

root@controlplane ~ ➜ kubectl create rolebinding dev-user-binding --role=developer --user=dev-u
ser --dry-run=client -o yaml > roleb.yaml

root@controlplane ~ ➜ vi roleb.yaml

root@controlplane ~ ➜ kubectl create -f roleb.yaml
rolebinding.rbac.authorization.k8s.io/dev-user-binding created
```

```
root@controlplane ~ ➜ cat role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  creationTimestamp: null
  name: developer
rules:
- apiGroups:
  - ""
  resources:
  - pods
  verbs:
  - list
  - create
  - delete
```

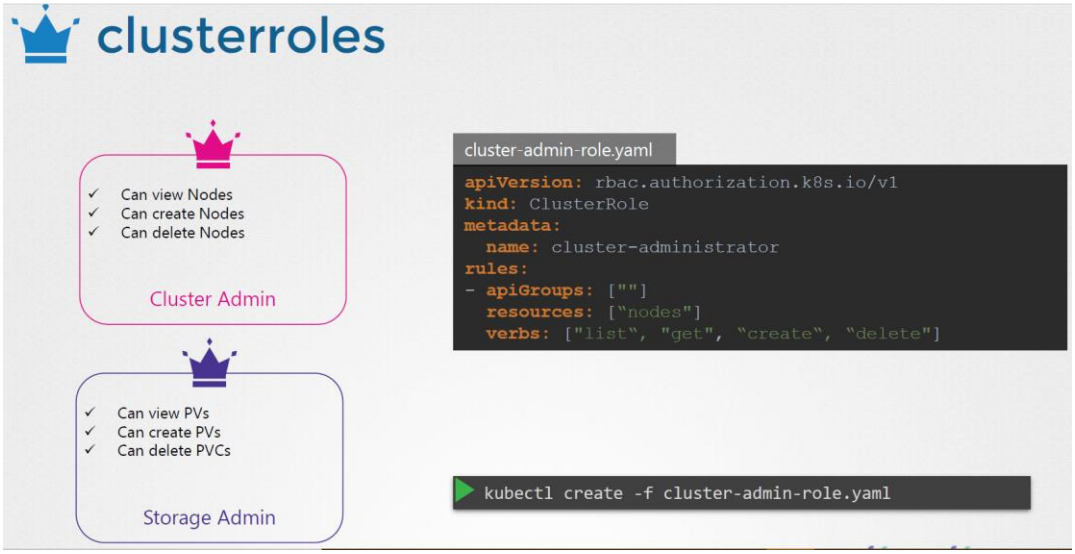
Both role and rolebinding changes can be done dymanically.

Cluster Roles - no

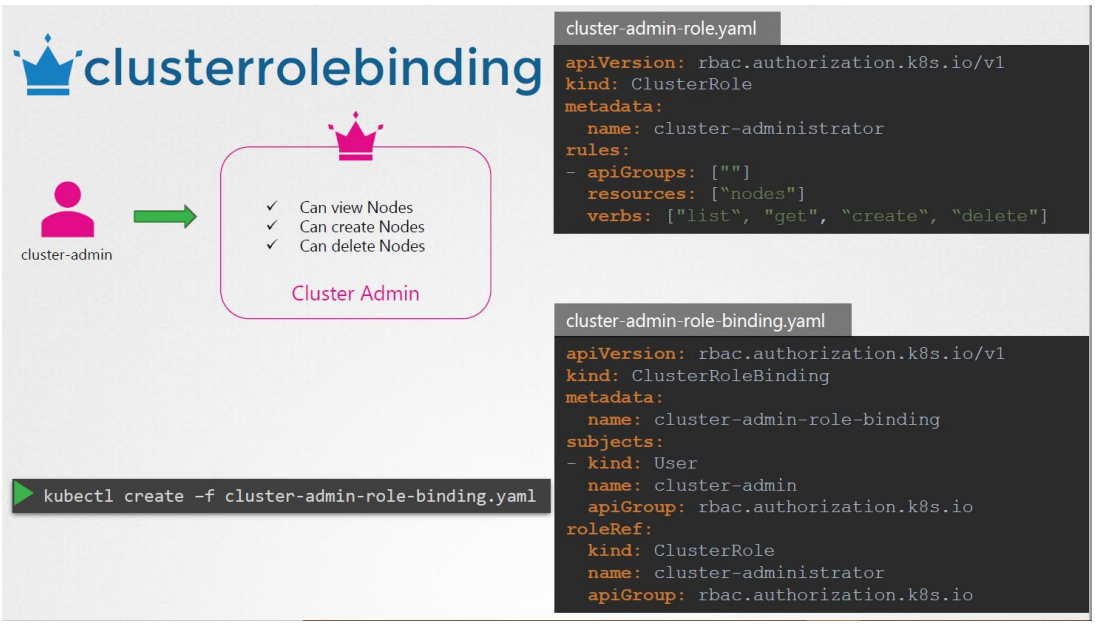
- As we know roles and rolesbinding is specific to namespace. It means rules specified in them is applicable to that namespace only (or default namespace if not specified). So it means by them, we can only control access to kubernetes resources that can be scoped to namespace. But what about resources which are not limited to namespace. Like nodes, you cannot bind nodes to namespace as it might be possible that in a single node we might have two pods running with different namespaces.
- Nodes are cluster scoped resources. So we can categorize resources into 2 groups -> **Namespaced** and **Cluster scoped.**



- Run command kubectl api-resources --namespaced=true to list down all namespace resources and by false it will list down all cluster based resources.
- Persistence volume, pvc, nodes, namespaces itself all these are objects are non namespace bind.
- To apply Authorization (i.e. Role and RoleBindings) to such cluster scoped resources clusterole and clusterrolebindings are used.



- Here we have created cluster role for resource nodes and allowed create view and delete verbs (permission)



- We can also create cluster role and cluseterrolebinding objects for namespace scoped resources also. Example -> If we specify suppose “pods” as resource in cluster role definition file then in such case this user which will be bind via clusterRole binding can perform operations on pods in all namespaces. In earlier role and rolebinding example rules are limited to current namespace only. earlier In other namespaces user is allowed to do everything.
- kubectl get clusterrole and kubectl get clusterrolebinding

```
controlplane ~ ➜ kubectl describe clusterrole cluster-admin
Name:         cluster-admin
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  *.*        []                 []              [*]
             [*]                []              [*]
```

```
controlplane ~ ✖ kubectl describe clusterrolebinding cluster-admin
Name:         cluster-admin
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
Role:
  Kind:  ClusterRole
  Name:  cluster-admin
Subjects:
  Kind   Name            Namespace
  ----   ----            ---------
  Group  system:masters
```

- imperative way to create cluster role and binding. Example -> allow access of nodes to michelle to perform any operation on any nodes.
  Kubectl create clusterrole noderole –verb=get,list,delete,create –resource=node –dry-run=client -o yaml

```
controlplane ~ ➜ kubectl create clusterrole noderole --verb=list,watch,get,create,delete --reso
urce=node --dry-run=client -o yaml > role.yaml

controlplane ~ ➜ cat role.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  creationTimestamp: null
  name: noderole
rules:
- apiGroups:
  - ""
  resources:
  - nodes
  verbs:
  - list
  - watch
  - get
  - create
  - delete

controlplane ~ ➜ kubectl create -f role.yaml
clusterrole.rbac.authorization.k8s.io/noderole created
```

```
controlplane ~ ✖ kubectl create clusterrolebinding nodecrb --user=michelle --clusterrole=noderol
e --dry-run=client -o yaml > crb.yaml

controlplane ~ ➜ cat crb.yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  creationTimestamp: null
  name: nodecrb
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: noderole
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: michelle

controlplane ~ ➜ kubectl create -f crb.yaml
clusterrolebinding.rbac.authorization.k8s.io/nodecrb created
```

```
controlplane ~ ➜ kubectl auth can-i get nodes --as michelle
Warning: resource 'nodes' is not namespace scoped
yes
```

```
controlplane ~ ✖ kubectl auth can-i get nodes --as michelle --all-namespaces
yes
```

- don’t use clusterrolebindings in place of clusterrolebinding. It won’t work.

Admission Controllers

- In general when we run kubectl command to create resources like pod. It goes to kube-api server and then it will create pod and information is stored to etcd db.
- At kube-api server we implemented authentication which verify user is allowed to run kubectl commands. For that certificates are used and cert info is placed in ~/.kube/config file.
- For authorization we then uses Role based access control achieved via role and rolebinding. Via rbac you can implement authorization like if user is allowed to create pod via kubeapi.
- But we are still limited to security and control over kubectl only. What if we want -> when pod create request comes we want to reject it’s creation if docker image it was referring to is from public docker repo. Or we want to make sure if container is running via root user don’t allow to create it. Or we want to check capability first Or pod definition should always contain labels. These types of things cannot be achieved with Authorization and Authentication. For such cases **Admission Controllers** comes-in to picture. It comes after Authorization.
- kubectl-> Authentication -> Authorization -> Admission Controllers -> Create Pods.
- Some of the pre in-built storage admission controllers are AlwaysPullImages, DefaultStorageClass, EventRateLimit, NamespaceLifecycle etc.
- AlwaysPullImages will make sure images are always pulled. DefaultStorageClass is enabled by default and check if any pvc is created default storage class should be applied, EventRateLimit is used to make sure that kube-api server will have limited desired request coming only. NamespaceLifecycle can provide options like if we run kubectl command and namespace provided is not already exists it will create one. By default NamespaceLifecycle make sure that it give error if namespace used in create command is not already created. It also make sure that namespaces like kube-system, kube-public cannot not be deleted.
- To add new admission controllers update enable-admission-plugins flag of kube-apiserver service. i.e. edit and save /etc/kubernetes/manifests/kube-apiserver.yaml.
- For disabling admission controllers update disable-admission-plugin flag with controller you want to disable.
- Now when we create new pod it will go to authentication, rbac and then controller. If all well then only it will create pod.
- /etc/kubernetes/manifests/kube-apiserver.yaml, inside it you can verify what controllers enabled and disabled.

```
root@controlplane /etc/kubernetes/manifests ✗ cat kube-apiserver.yaml
apiVersion: v1
kind: Pod
metadata:
  annotations:
    kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 10.26.121.8:6443
  creationTimestamp: null
  labels:
    component: kube-apiserver
    tier: control-plane
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
    - --advertise-address=10.26.121.8
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision
    - --enable-bootstrap-token-auth=true
    - --disable-admission-plugins=DefaultStorageClass
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
```

- Since the kube-apiserver is running as pod you can check the process to see enabled and disabled plugins. ps -ef | grep kube-apiserver | grep admission-plugins

```
root@controlplane /etc/kubernetes/manifests ➜ ps -ef | grep kube-apiserver | grep admission-plugins
root     21278 21261  0 11:46 ?        00:00:16 kube-apiserver --advertise-address=10.26.121.8 --allow-privileged=true --authorization-mo
e=Node,RBAC --client-ca-file=/etc/kubernetes/pki/ca.crt --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision --enable-bootstr
ap-token-auth=true --disable-admission-plugins=DefaultStorageClass --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt --etcd-certfile=/etc/kube
rnetes/pki/apiserver-etcd-client.crt --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key --etcd-servers=https://127.0.0.1:2379 --
insecure-port=0 --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt --kubelet-client-key=/etc/kubernetes/pki/api
server-kubelet-client.key --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname --proxy-client-cert-file=/etc/kubernetes/pki/fr
ont-proxy-client.crt --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key --requestheader-allowed-names=front-proxy-client -
-requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt --requestheader-extra-headers-prefix=X-Remote-Extra- --requestheader-
group-headers=X-Remote-Group --requestheader-username-headers=X-Remote-User --secure-port=6443 --service-account-issuer=https://kubernetes
.default.svc.cluster.local --service-account-key-file=/etc/kubernetes/pki/sa.pub --service-account-signing-key-file=/etc/kubernetes/pki/sa
.key --service-cluster-ip-range=10.96.0.0/12 --tls-cert-file=/etc/kubernetes/pki/apiserver.crt --tls-private-key-file=/etc/kubernetes/pki/
apiserver.key
```

Validating and Mutating Admission Controllers

- All admission controllers are classified into 2 groups. Validating and mutating. Mutating generally runs before validating and are the one's that modifies resources like pods before it is created. like we can attach DefaultStorageClass when new pvc is created. Validating controllers applies validations like you cannot delete kube-system namespace. If validation fails command is rejected.
- Apart from these built-in admission controllers. We can create our own such admission controllers to provide custom implementations. For that 2 built-in admission controllers can be used-> MutatingAdmissionWebhook and ValidatingAdmissionWebhook.
- In such case first request goes to all built-in controllers and after that it will got to custom admission controllers. If at any controller request validation fails, chain breaks and error message is shown to user.
- For that we can configure our admissionwebhooks to point to admission webhook service. It can be running in same k8s cluster or outside that cluster also. So when flow reaches let say mutatingadmissionwebhook it will make call to admission webhook service by passing admission review object in json format. This object contains all details like user , operation user want to perform, on what object and object details etc. after processing request admission webhook service return admission review object with result whether the request is allowed or not. In the json field named "allowed": true indicate this request is allowed.

- Create and deploy admission webhook server which has own logic of request allowance on somewhere either on k8s cluster or on external machine and then create web configuration object which will connect to above server and ask for decision.
- Server can be written in any language. Only rule is that it must accept mutate and validate apis and respond with admission review object in json format which our k8s webhooks understands.
- We don't need to write code of admission webhook server for ckad exam.
- If above server deployed as pod in k8s cluster. We need to expose it as service also so that they can be used with webhook.
- Now we have webhook deployed and exposed as service. Now we need to create and configure webhook admission controller object.
- Create mutatingwebhook configuration file with sample fields like below and then create object->

```yaml
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: demo-webhook
webhooks:
  - name: webhook-server.webhook-demo.svc
    clientConfig:
      service:
        name: webhook-server
        namespace: webhook-demo
        path: "/mutate"
      caBundle: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS....
    rules:
      - operations: [ "CREATE" ]
        apiGroups: [""]
        apiVersions: ["v1"]
        resources: ["pods"]
```

- Above sample file apiVersion is now deprecated and not preferred.

```
root@controlplane ~ ➜ kubectl create -f webhook-configuration.yaml
Warning: admissionregistration.k8s.io/v1beta1 MutatingWebhookConfiguration is deprecated in v1.1
6+, unavailable in v1.22+; use admissionregistration.k8s.io/v1 MutatingWebhookConfiguration
mutatingwebhookconfiguration.admissionregistration.k8s.io/demo-webhook created

root@controlplane ~ ➜ kubectl get mutatingwebhookconfiguration
NAME           WEBHOOKS   AGE
demo-webhook   1          63s

root@controlplane ~ ➜ kubectl delete mutatingwebhookconfiguration demo-webhook
mutatingwebhookconfiguration.admissionregistration.k8s.io "demo-webhook" deleted
```

Api Groups

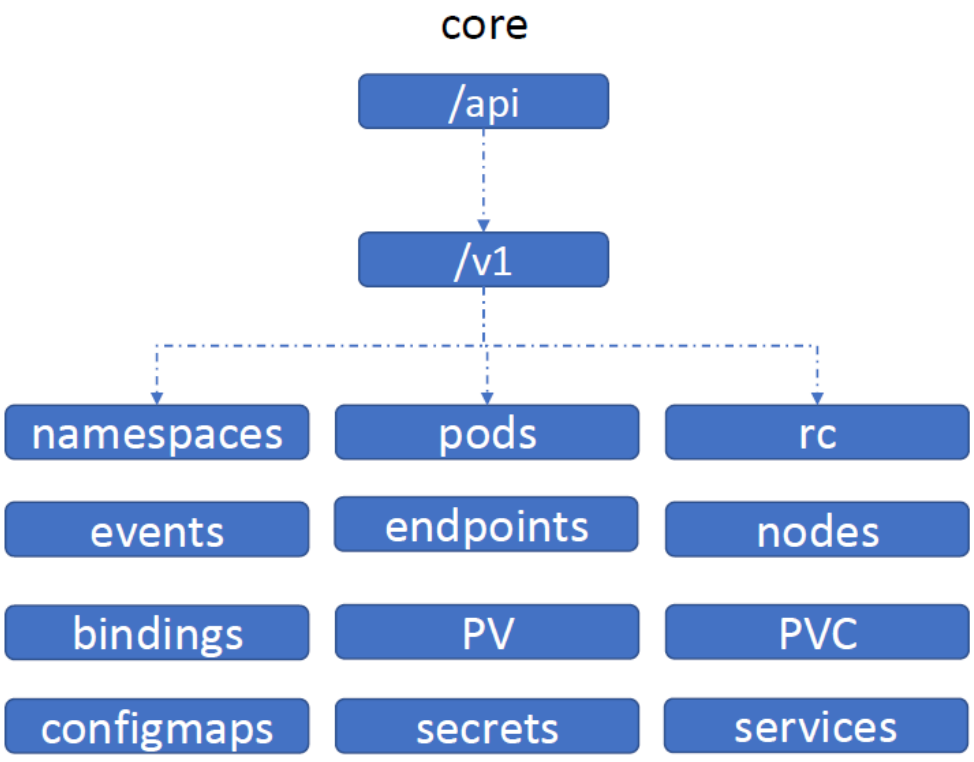
- kube-api server is the one by which we can interact with Kubernetes clusters. We can use either kubectl command to invoke kubeapi server or we can use rest end-points.
- Some examples ->

```
curl https://kube-master:6443/version
{
  "major": "1",
  "minor": "13",
  "gitVersion": "v1.13.0",
  "gitCommit": "ddf47ac13c1a9483ea035a79cd7c10005ff21a6d",
  "gitTreeState": "clean",
  "buildDate": "2018-12-03T20:56:12Z",
  "goVersion": "go1.11.2",
  "compiler": "gc",
  "platform": "linux/amd64"
}
```

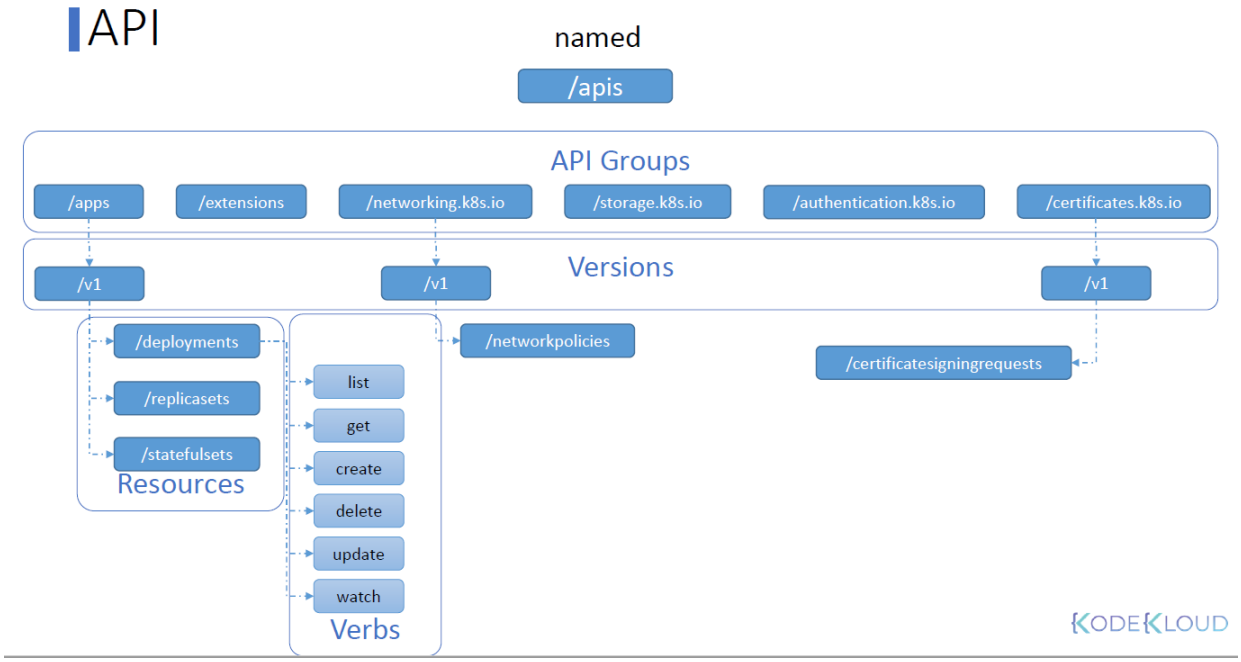
```
curl https://kube-master:6443/api/v1/pods
{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "selfLink": "/api/v1/pods",
    "resourceVersion": "153068"
  },
  "items": [
    {
      "metadata": {
        "name": "nginx-5c7588df-ghsbd",
        "generateName": "nginx-5c7588df-",
        "namespace": "default",
        "creationTimestamp": "2019-03-20T10:57:48Z",
        "labels": {
          "app": "nginx",
          "pod-template-hash": "5c7588df"
        },
        "ownerReferences": [
          {
            "apiVersion": "apps/v1",
            "kind": "ReplicaSet",
            "name": "nginx-5c7588df",
            "uid": "398ce179-4af9-11e9-beb6-020d3114c7a7",
            "controller": true,
            "blockOwnerDeletion": true
          }
        ]
      },
```

- Kubernetes api are groups into multiple such groups based on purpose.

- version api is used to view version of cluster, metrics and health api is used to monitor health of cluster. Logs are used for integrating to third party logging app. Here we will focus on apis responsible for cluster functionality.
- Api are divided into 2 category-> core api and named api



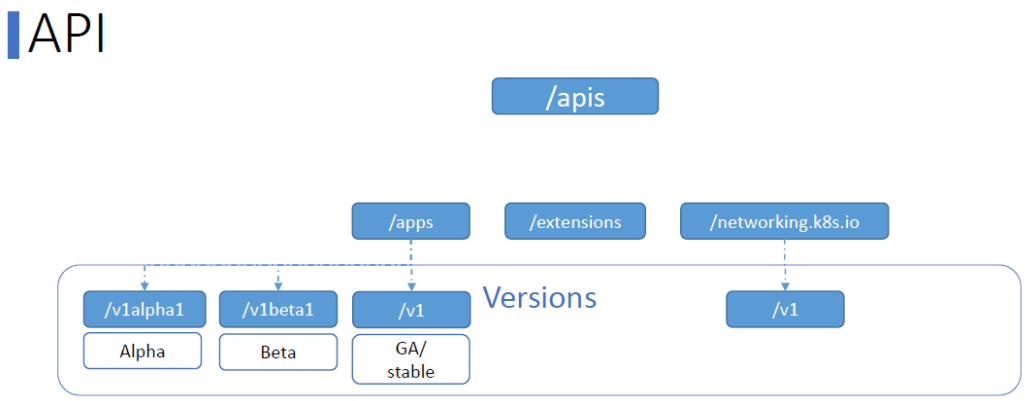
- Named group apis are more organized and going forward new features will be introduced to this group only.



- Run kubectl proxy command. it will create proxy service locally on http port 8001 to access the api server. By this you don't need to pass certificates in every rest end point call. As this proxy will use kubeconfig file present at default location to forward request along with certificates to kube-api server.

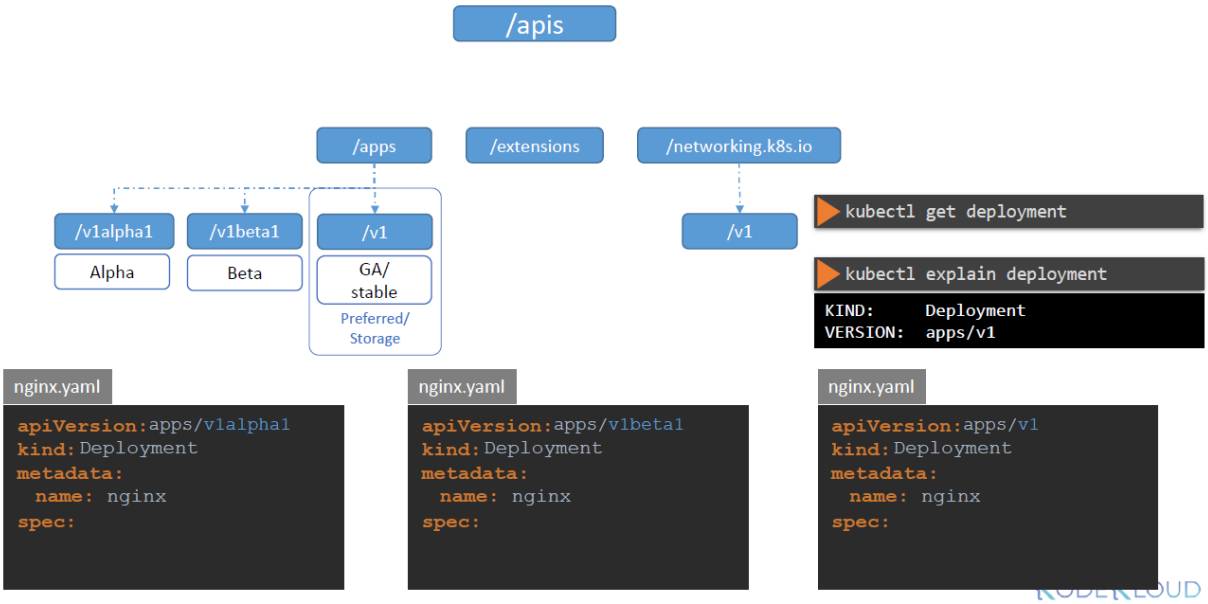
Api Versions

- Api version comes after api group. And support different versions of any particular api.
- Different type of apiVersion supported are alphav1, betav1 and GA which is version v1



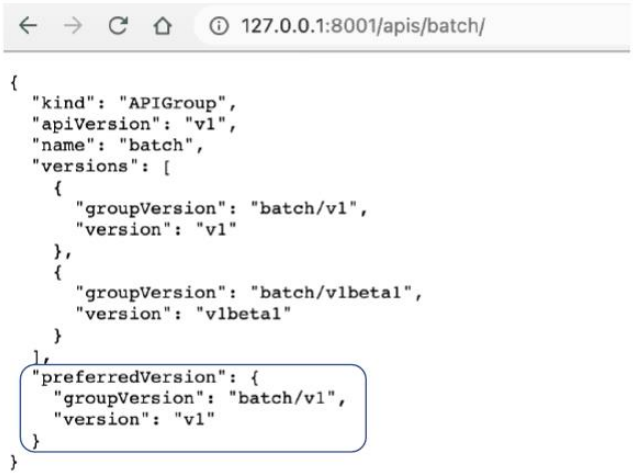
- Alpha is the first time when an api is developed and merged into k8s codebase and become part of k8s release first time. It is by-default made disabled which we can enable explicitly via kube-apiserver. Such api's are added to just to get initial feedback of those who wanted to try it out. If not needed such alpha api can be dropped later by k8s. they may have bugs also.
- Once it is little stabilized it moves to beta stage, here it is enabled by default and may have some minor bugs. But it is assured that they will be moved to stable final release later which is GA.
- Once everything worked fine it moves to GA stage where it available to all users and highly reliable.
- Some of the api like batch or autoscaling has multiple version available. So it is not like that once a specific api moves to GA it's beta and alpha will no longer in use. batch has v1 and v1beta1 available.
- It means you can create same object with any of the version which is allowed. But we still have single preferred and storage version.
- When we have multiple version available and if we run kubectl get deployments command which version kubectl will query? It is defined by preferred version.

API



- Similarly when multiple version are present only one version can be storage version. It defines in which version object is stored in etcd. So suppose default storage version is v1 and we have created pod with version v1alpha1 then it will be first converted into v1 and then it will be stored into etcd db.
- Mostly preferred and storage version are same but they can be different.
- Get the default value of preferredVersion by hitting that specific api.

Preferred Version



- By default you cannot see default storage version. For that you need to query etcd db.

Storage Version

- To enable alpha api we can provide that api version in kube-apiserver service runtime-config parameter and restart kube-apiservice. Check commands section point 39 for exact path and steps.

Enabling/Disabling API groups

```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --etcd-keyfile=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --etcd-servers=https://127.0.0.1:2379 \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-certificate=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --kubelet-client-key=/var/lib/kubernetes/apiserver-etcd-client.key \\
  --kubelet-https=true \\
  --runtime-config=batch/v2alpha1\\
  --service-account-key-file=/var/lib/kubernetes/service-account.pem \\
  --service-cluster-ip-range=10.32.0.0/24 \\
  --service-node-port-range=30000-32767 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --tls-cert-file=/var/lib/kubernetes/apiserver.crt \\
  --tls-private-key-file=/var/lib/kubernetes/apiserver.key \\
  --v=2
```

- If wanted to know default apigroup and api-version of deployment.
  kubectl explain deploy.

```
root@controlplane ~ ➜ k explain deployment
KIND:     Deployment
VERSION:  apps/v1

DESCRIPTION:
     Deployment enables declarative updates for Pods and ReplicaSets.

FIELDS:
   apiVersion   <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest interna
```

  Here group is apps and version is v1

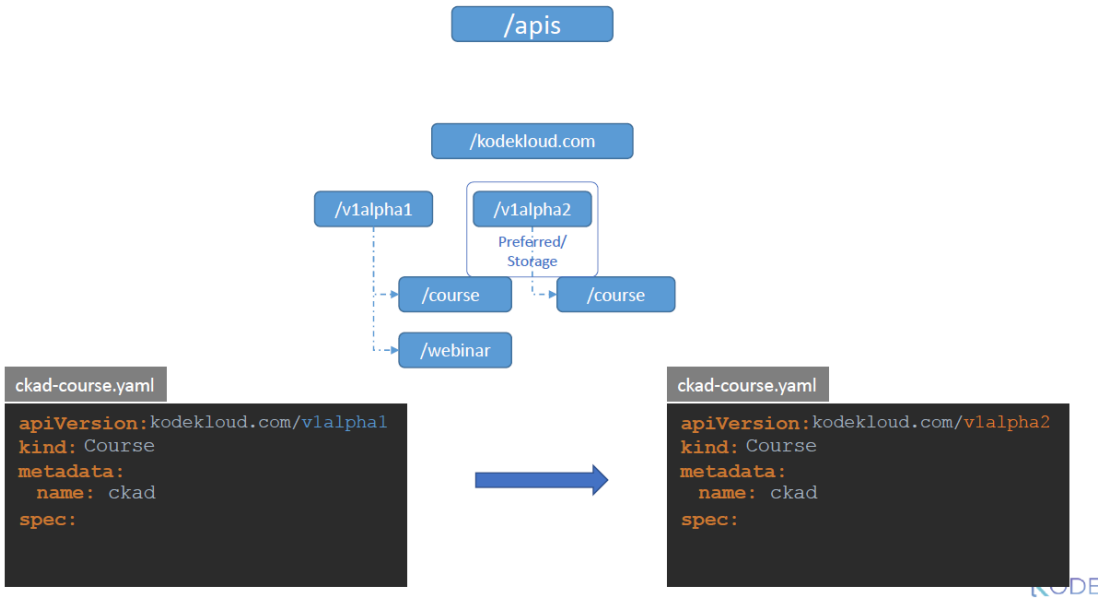
Api Deprecations

- We know that a single api can support multiple versions at a time. But how many we should support or why to support older or how long should we support older versions? This is govern by some standard rules and deprecation policy created by k8s.
- Suppose we have created 2 new api and introduced in release x as v1alpha1. and suppose 2nd api we want to remove. So by rule 1 we cannot just simply delete it from v1aplha1. With newer release v1alpha2 we can delete 2nd api but it will still remain visible in v1aplha1 version.

API Deprecation Policy Rule #1

API elements may only be removed by incrementing the version of the API group.

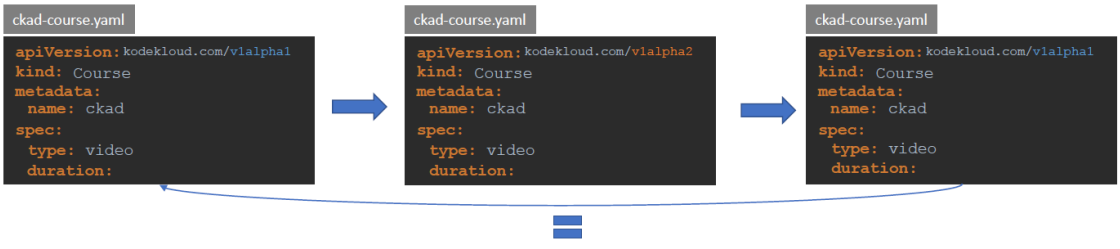
API



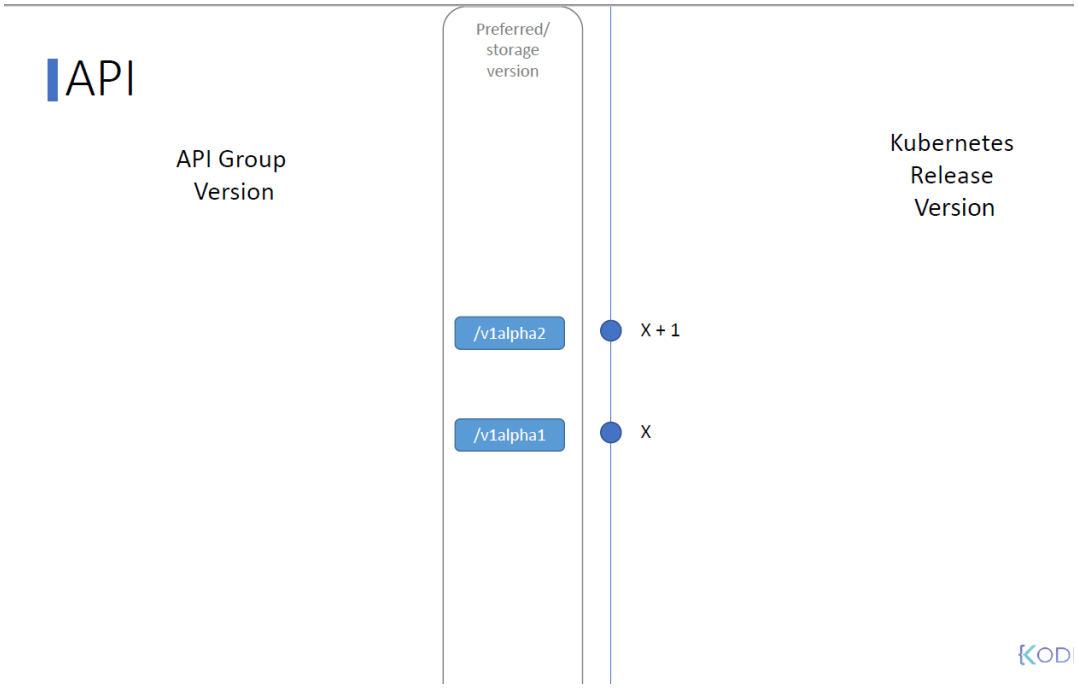
- In above example suppose we want to remove webinar. Then we removed it in v1alpha2 and it is still present at v1alpha1. We have set preferred and storage version as v1alpha2 course. By this user can still create resource course as v1alpha1 but it will be stored as v1alpha2 now.
- Rule 2 states that if suppose we added new field 'duration' in v1alpha2 release then it should be backward compatible. In such case similar field should be added to earlier release to support conversion.

API Deprecation Policy Rule #2

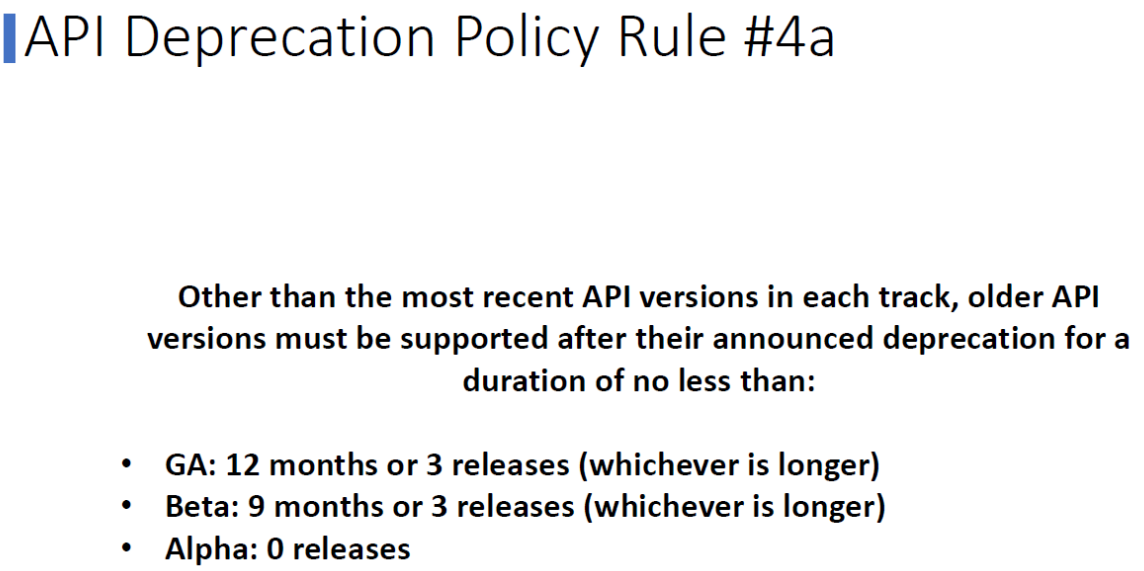
API objects must be able to round-trip between API versions in a given release without information loss, with the exception of whole REST resources that do not exist in some versions.



- For example, an object can be written as v1 and then read back as v2 and converted to v1, and the resulting v1 resource will be identical to the original. The representation in v2 might be different from v1, but the system knows how to convert between them in both directions. Additionally, any new field added in v2 must be able to round-trip to v1 and back, which means v1 might have to add an equivalent field or represent it as an annotation.
- Now suppose we introduce new api in x release of k8s, hence version will be v1alpha1. Since it is the only version it is preferred/storage version. Now after some testing we want to release v1alpha2 version. Since we are in alpha phase we don't need to keep v1alpha1 version in X+1 k8s release.

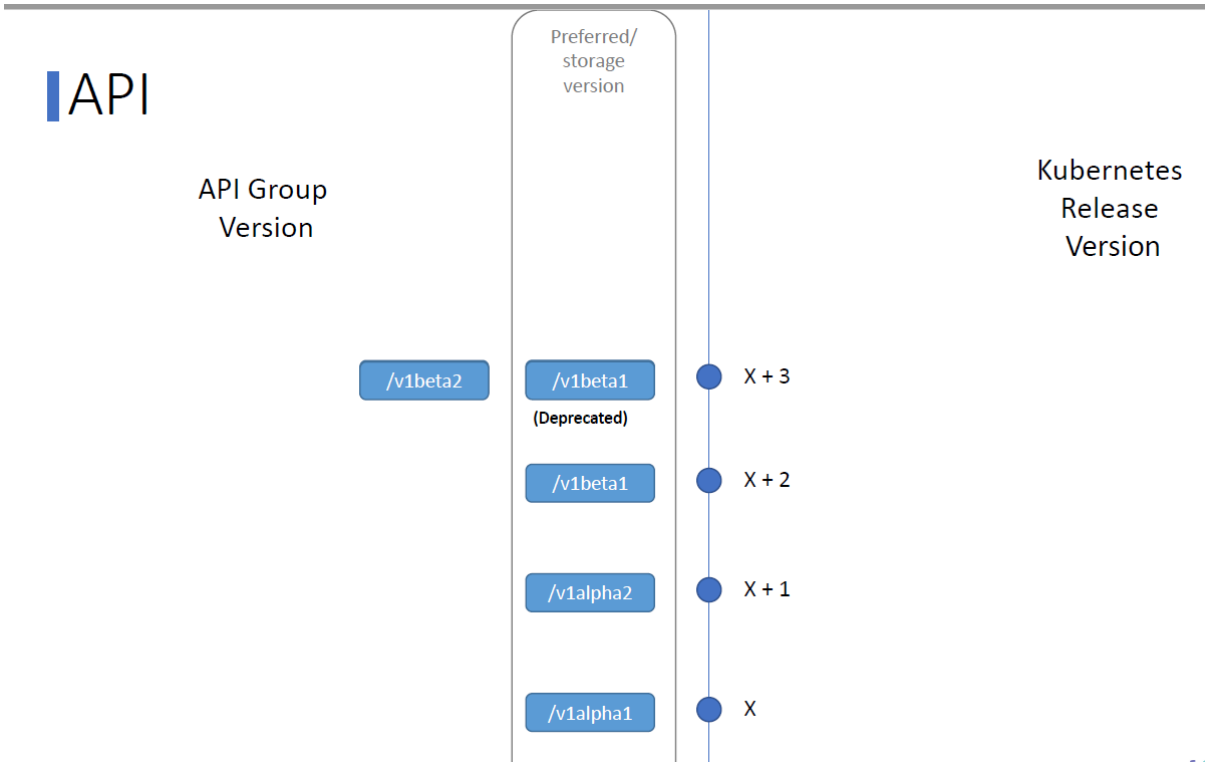


- Above point is govern by rule 4a. that is why alpha1 release need not to be maintained when alpha2 is released. In such case in release note just mention that v1alpha1 is no longer available and one must update there code if they are already using it.



- Now suppose we are sure that api is stable enough to proceed to beta phase. Since alpha does not need to maintained in x+2 release, we remove it and just mention that in release note that v1alpha2 is removed.

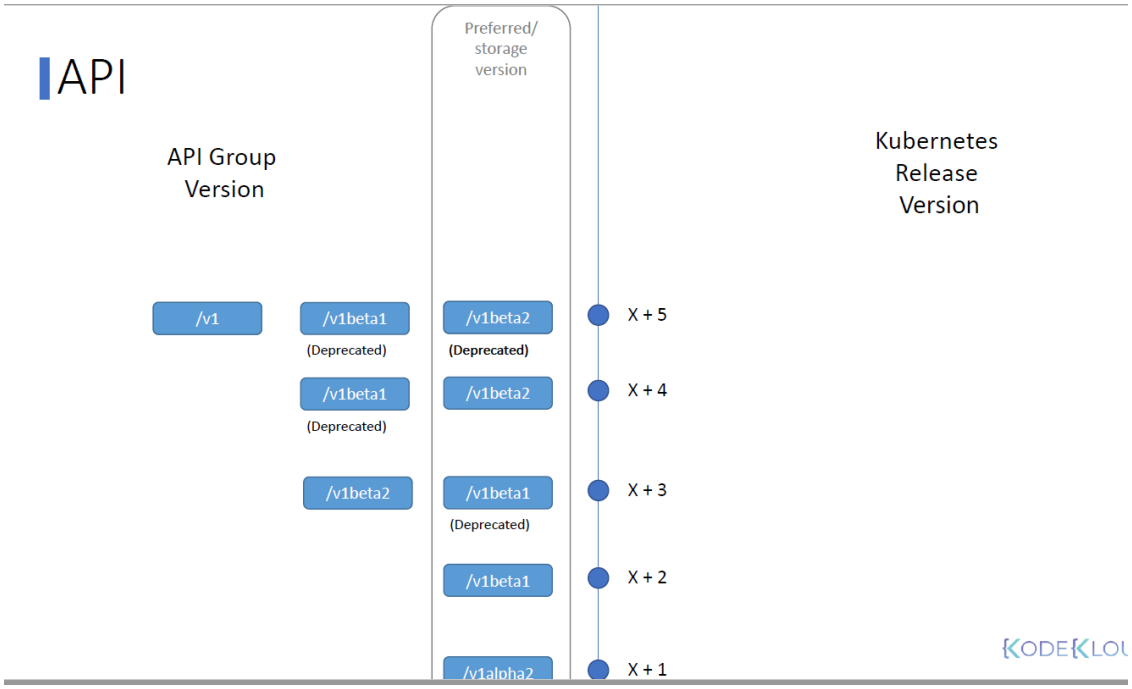
- Now suppose we want to introduce v1beta2 release in x+3 k8s release. Since for beta release as per rule 4a you cannot delete v1beta1 release till next 3 release. Hence it will be marked as deprecated but not deleted. And also note that preferred version will never be the first occurrence of new release. Hence v1beta1 will even though deprecated will be default version. (rule 4b)



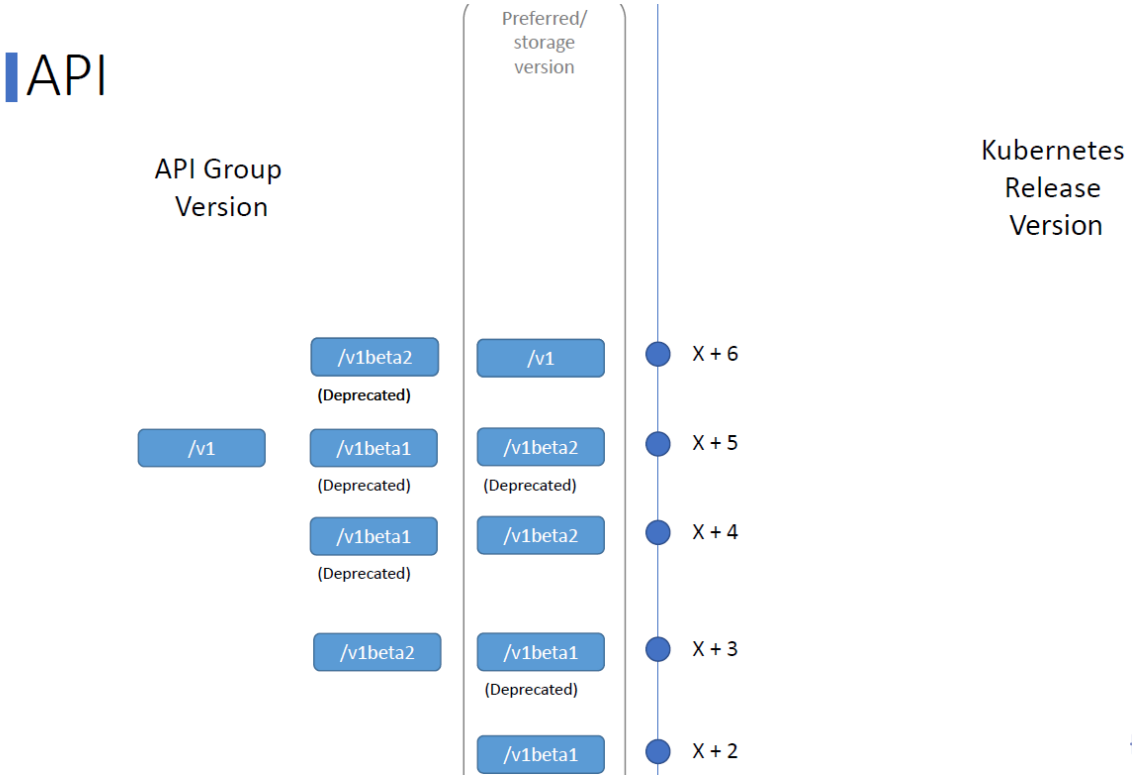
API Deprecation Policy Rule #4b

The "preferred" API version and the "storage version" for a given group may not advance until after a release has been made that supports both the new version and the previous version

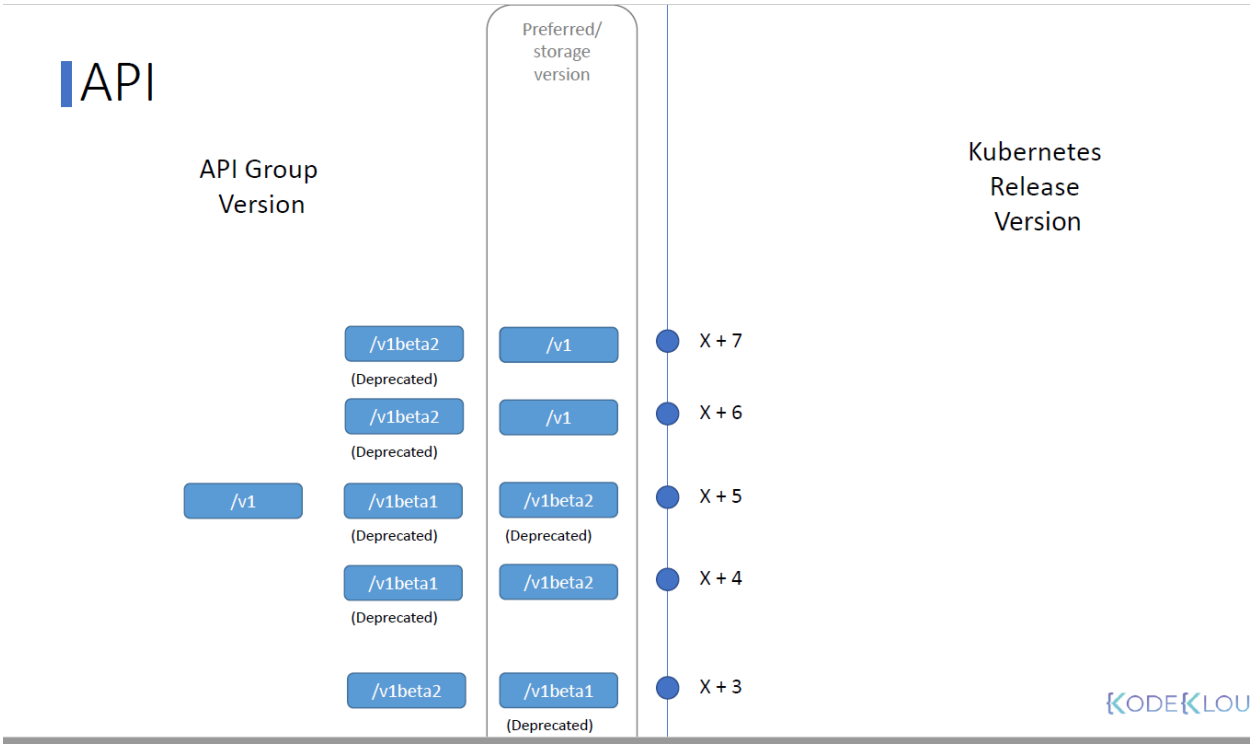
- Now in next release of k8s x+4 preferred version can be changed to v1beta2 as per above rule 4b.
- In next release of k8s we have v1 version of api ready. Now v1beta2 will be marked deprecated and will be default preferred version. Still we cannot remove v1beta1 as still 3 releases not yet covered.



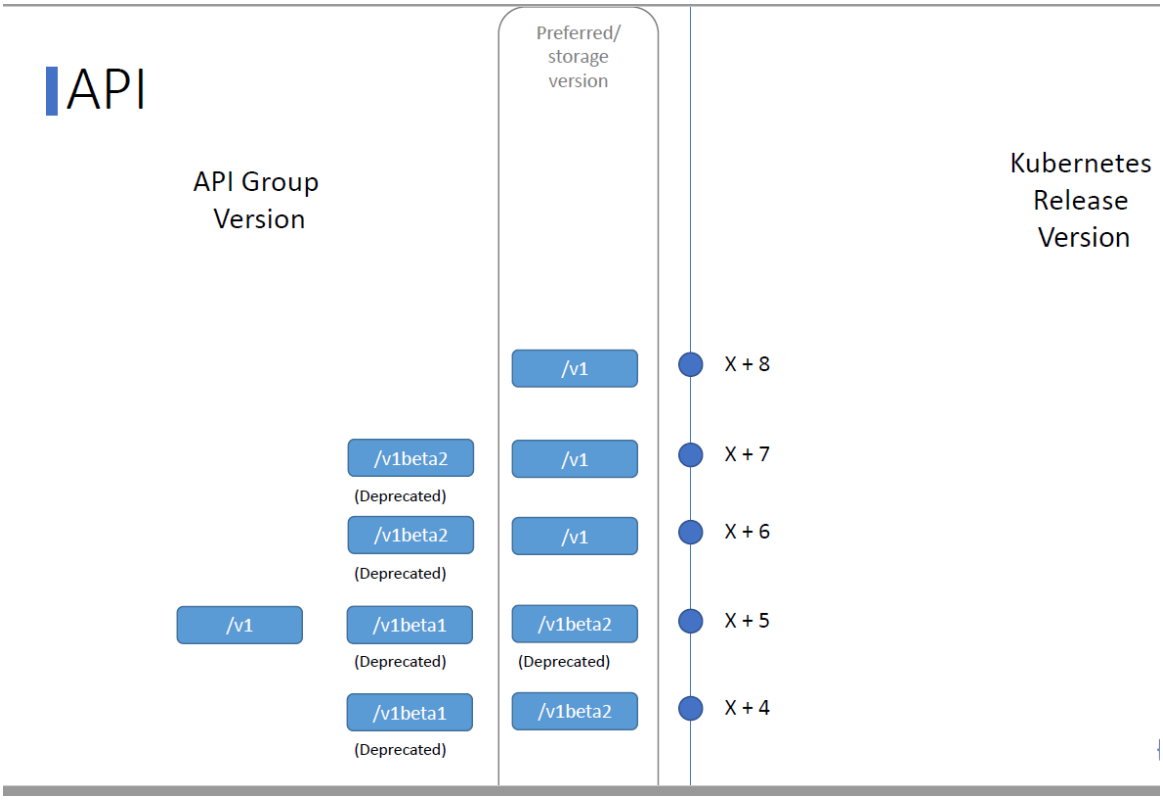
- Now In next release x+6, v1beta1 can be deleted as per rule 4a and v1 will be made preferred version.



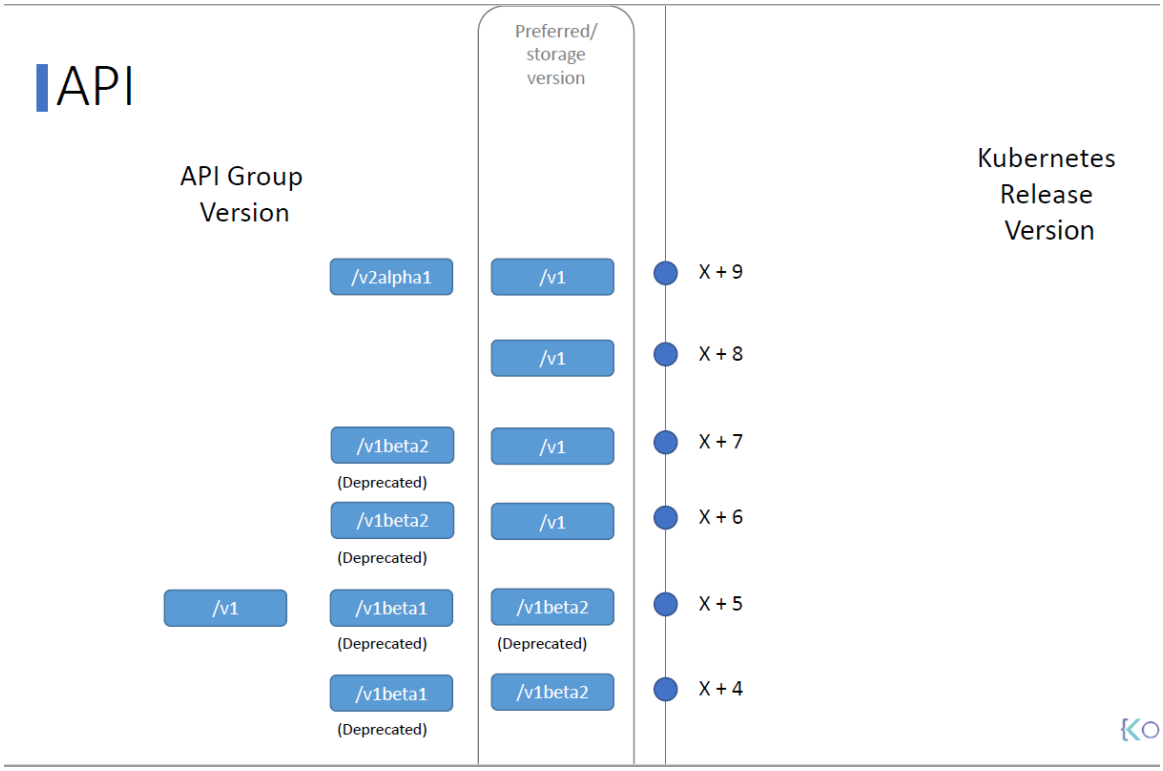
- In X+7 nothing changed as v1beta2 cannot be deleted.



- In x+8 we can delete v1beta2



- Suppose we want to release v2alpha1 version of our api. So should we make v1 now deprecated? No. as per rule 3 only stable version can make other deprecated. Example ga version can only be replaced with ga version. Beta version can make alpha version deprecated. Alpha version can make only alpha version deprecated. So in this case till v2alpha1 goes to beta and then finally to GA v2 then only it can make v1 deprecated.

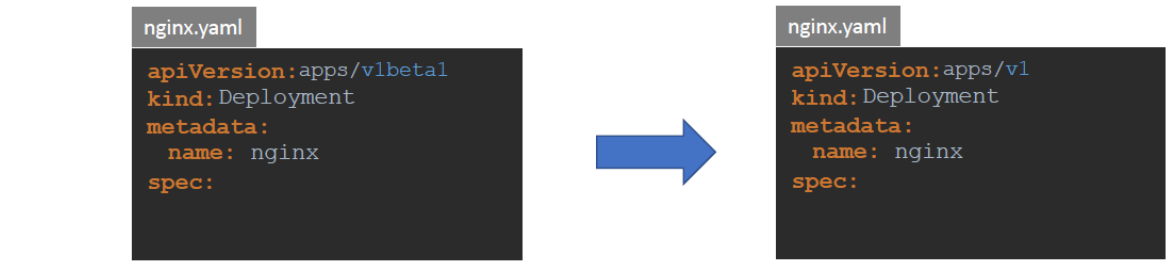


API Deprecation Policy Rule #3

An API version in a given track may not be deprecated until a new API version at least as stable is released.

- **kubectl convert plugin ->** now when old deprecated api removed from release following above rules we need to convert our existing manifest definition yaml file to the updated version to work correctly. For example-> deployment v1beta1 need to be converted v1 now. For that we can use kubectl convert plugin.
- kubectl convert -f nginx.yaml --output-version apps/v1

Kubectl Convert




```
kubectl convert -f <old-file> --output-version <new-api>
```

```
kubectl convert -f nginx.yaml --output-version apps/v1
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: nginx
  name: nginx
```

Example 2->

```
root@controlplane ~ ➜ kubectl convert -f ingress-old.yaml --output-version networking.k8s.io/v1
> in.yaml

root@controlplane ~ ➜ kubectl apply -f in.yaml
ingress.networking.k8s.io/ingress-space created
```

Content of in.yaml file after convert operation->

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
  creationTimestamp: null
  name: ingress-space
spec:
  rules:
  - http:
      paths:
      - backend:
          service:
            name: ingress-svc
            port:
              number: 80
        path: /video-service
        pathType: Prefix
status:
  loadBalancer: {}
```

- This plugin is not by default available. So, You might need to install it.

- Example -> Check what is the preferred version for authorization.k8s.io api group?

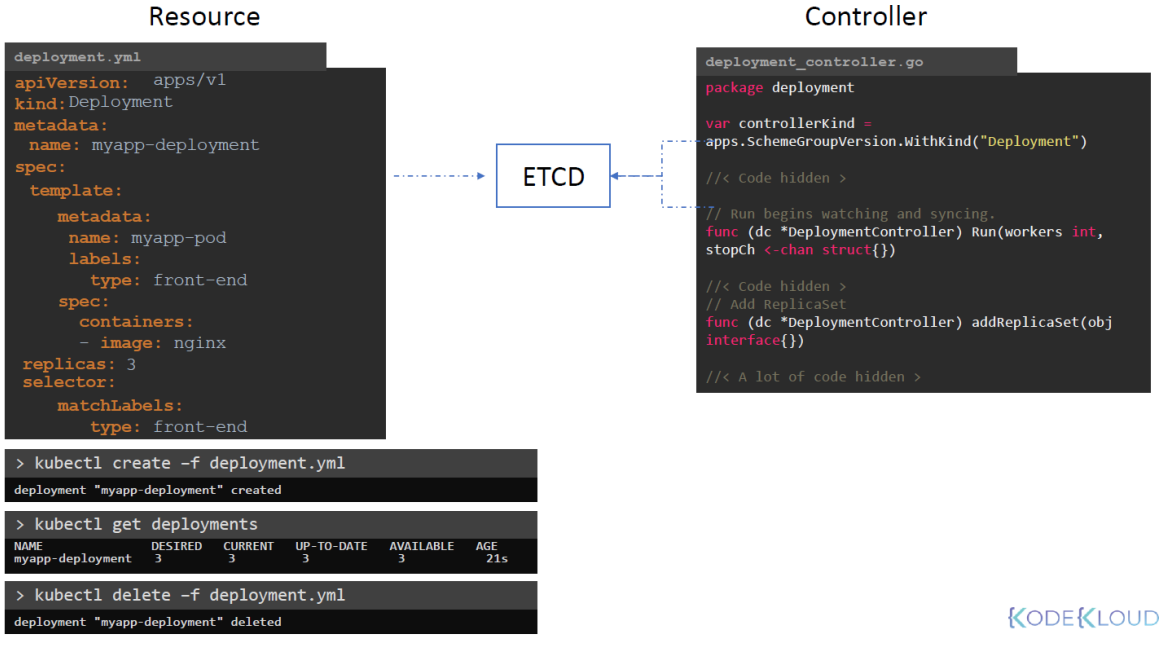
```
root@controlplane ~ ➜ kubectl proxy 8001&
[1] 20335

root@controlplane ~ ✦ ➜ Starting to serve on 127.0.0.1:8001
curl localhost:8001/apis/authorization.k8s.io
{
  "kind": "APIGroup",
  "apiVersion": "v1",
  "name": "authorization.k8s.io",
  "versions": [
    {
      "groupVersion": "authorization.k8s.io/v1",
      "version": "v1"
    },
    {
      "groupVersion": "authorization.k8s.io/v1beta1",
      "version": "v1beta1"
    }
  ],
  "preferredVersion": {
    "groupVersion": "authorization.k8s.io/v1",
    "version": "v1"
  }
```

Connection Closed

Where & runs the command in the background and kubectl proxy command starts the proxy to the kubernetes API server

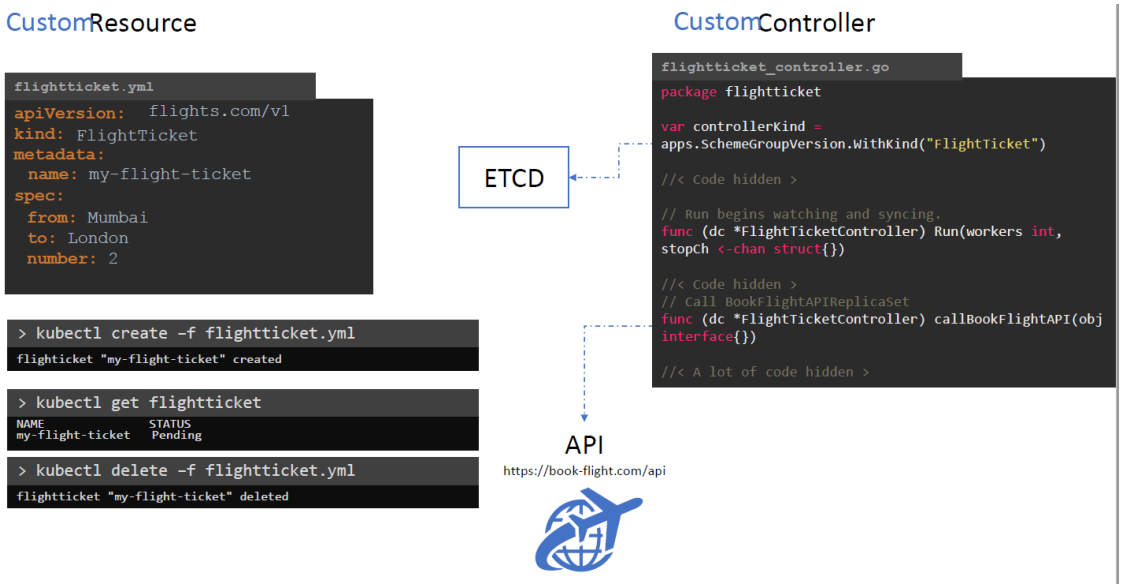
Custom Resource Definition (CRD)

- When we create k8s object say deployment. k8s create pods, rs and deployment object and store its info to etcd db. So who is responsible for reading info from etcd and create all these required no of resource. It's controller job. It is a process that run in background and it monitor status of resources it suppose to managed to perform actions. In this case deployment controller will monitor deployment related resources. And when we create, delete and update deployment it make required changes in the cluster.



- Below are some resources and there controller which manage such resources ->

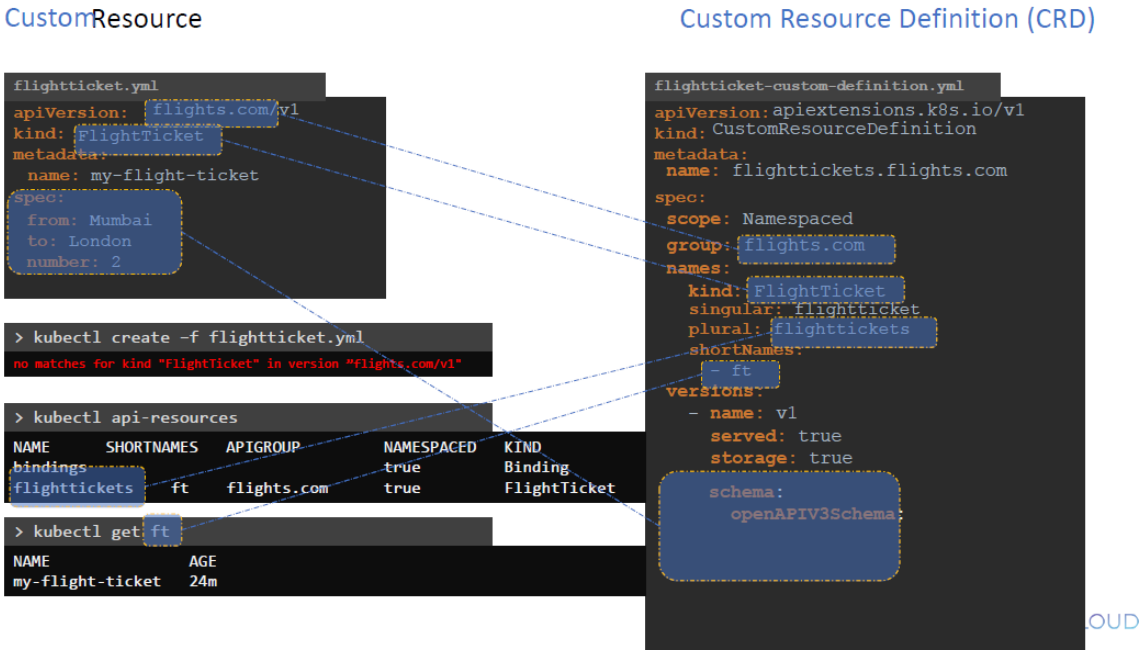
- Now what if we want to create our own custom resources to create our own custom resource object to perform certain action. Let's say, we want flightTicket resource object which will book flight for us.



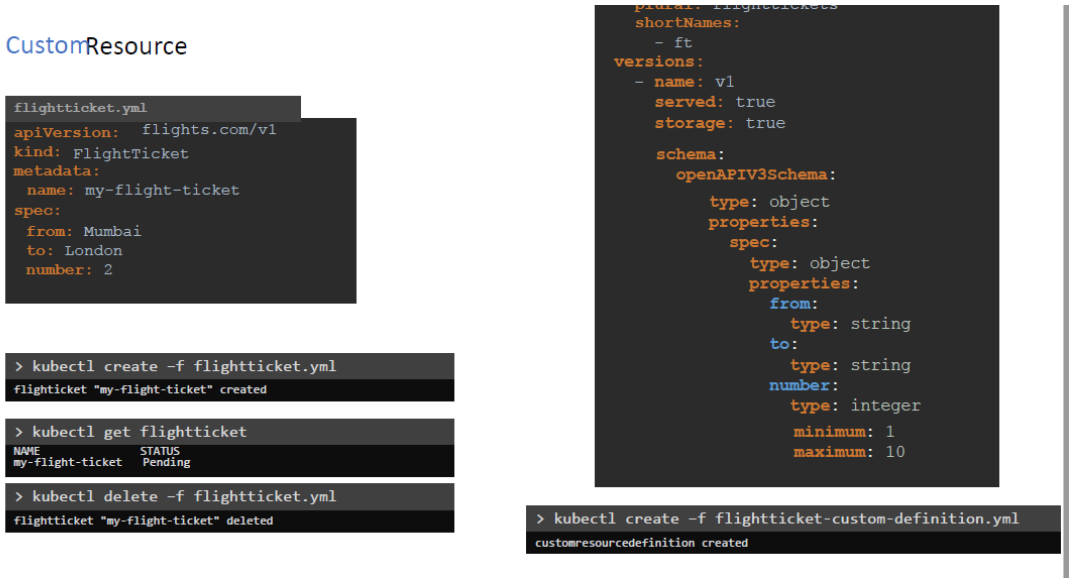
- Here we want to create flight ticket object with details like from and to station and no of tickets we want to book. Now when we create object flight ticket should be created and we want to list them and might want to delete flight ticket booking. By using above definition file and create command flightTicket object will be created and saved into etcd db. But what we also want is some actions like when we create flightTicket, apart from saving and creating our custom resource object in etcd and actual flight ticket should be booked.. and for that we want to invoke an external api of flight booking website. for that we need to also create CustomController named FlightController which will do actions like book flight. This controller will monitor flightTicket resource and perform required actions by invoking actual api of book flight or delete flight tickets.
- Suppose now we run command kubectl create -f flightticket.yaml. will it create resource in etcd? NO? why? Because k8s does not know kind: FlightTicket. We need to provide this info to k8s first along with other schema details like to ,from and number fields and there data type also.

```
> kubectl create -f flightticket.yml
no matches for kind "FlightTicket" in version "flights.com/v1"
```

- For that we will create CRD file for flightTicket and by reading this file k8s will now understand every info it needs related to our custom resource flightTicket.



- We will also mention schema here we provide datatype and fields that we wanted to support in our custom resource.



- Now when we create flightticket crd object and now when we create flightTicket object via definition files it will not give error. It will create required object in etcd.
- Now We need customController to be able to perform actual booking actions.

Custom Controllers

- For exam you do need to create own controller. Just basics would work. So let's suppose we have created our customController for flight ticket booking and make docker image of it and run it as pod in k8s cluster itself. It will listen for events related to flightTicket resource and perform desired action.
- For reference just attached required steps to create controller in go language ->

Custom Controller

```
> go
Go is a tool for managing Go source code.

go <command> [arguments]
```

```
> git clone https://github.com/kubernetes/sample-controller.git
Cloning into 'sample-controller'...

Resolving deltas: 100% (15787/15787), done.
```

```
> cd sample-controller
```

Customize controller.go with our custom logic

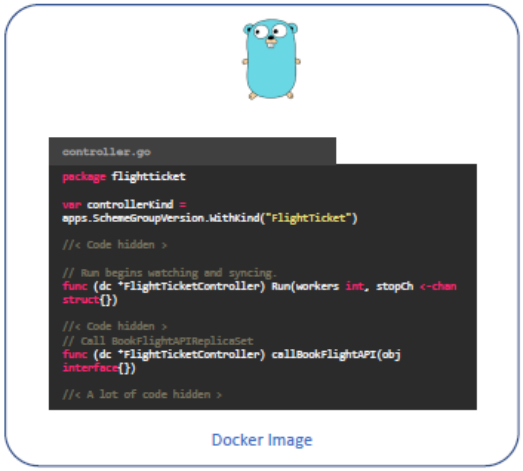
```
> go build -o sample-controller .
go: downloading k8s.io/client-go v0.0.0-20211001003700-dbfa30b9d908

go: downloading golang.org/x/text v0.3.6
```

```
> ./sample-controller -kubeconfig=$HOME/.kube/config
I1013 02:11:07.489479   40117 controller.go:115] Setting up event handlers

I1013 02:11:07.489701   40117 controller.go:156] Starting FlightTicket controller
```

controller.go

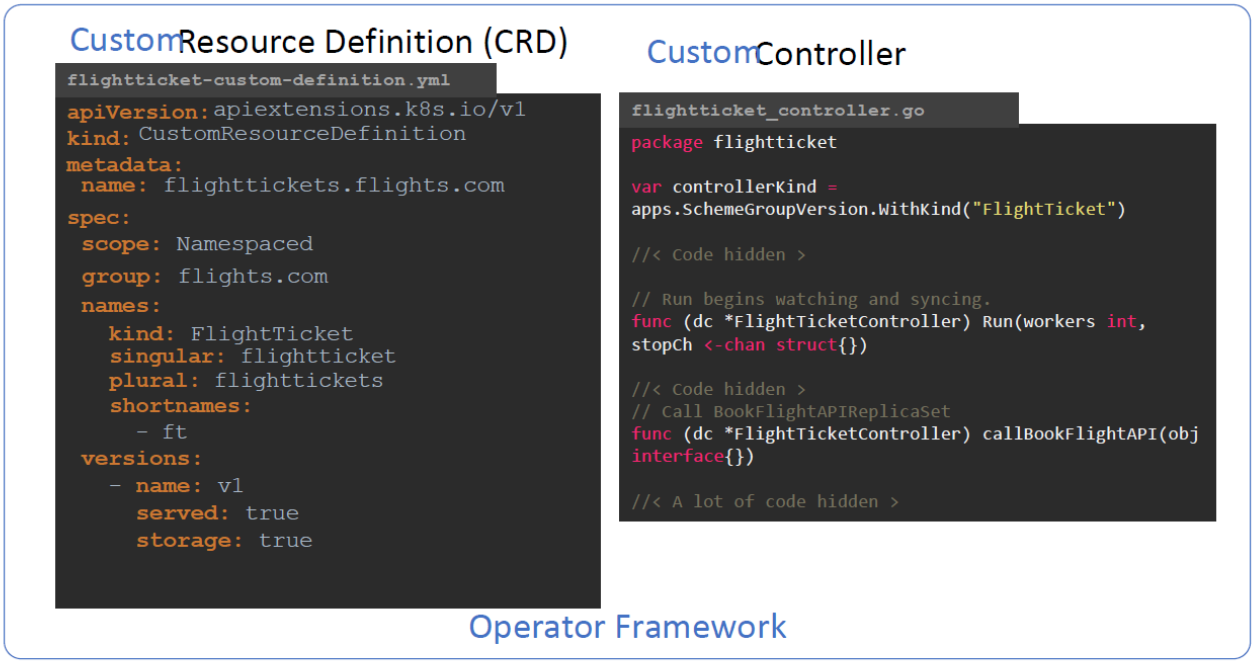
```go
package flightticket

var controllerKind =
apps.SchemeGroupVersion.WithKind("FlightTicket")

//< Code hidden >

// Run begins watching and syncing.
func (dc *FlightTicketController) Run(workers int,
stopCh <-chan struct{})

//< Code hidden >
// Call BookFlightAPIReplicaSet
func (dc *FlightTicketController) callBookFlightAPI(ob
interface{})

//< A lot of code hidden >
```

Custom Controller



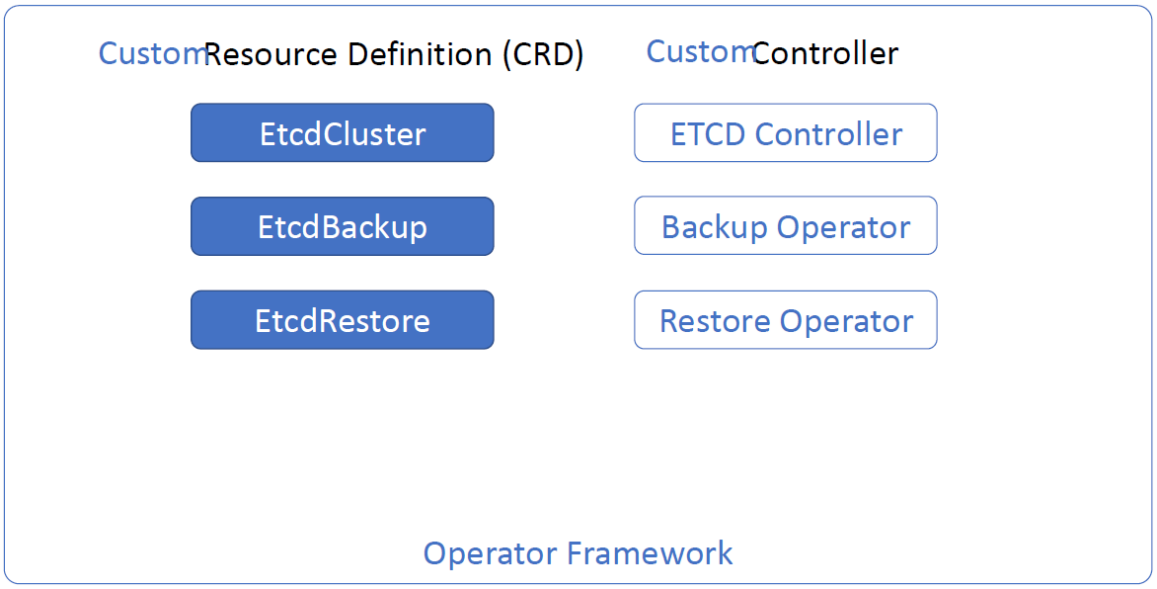
Operator Framework(NA)

- Now we know crd and custom controller work together to make our custom resource working. We need to manually create crd first and then custom controller. Then we create actual resources object. We can combine all of them together to deployed as a single entity known as operator framework. It will internally create resources , crd and controller.




```
> kubectl create -f flight-operator.yaml
```

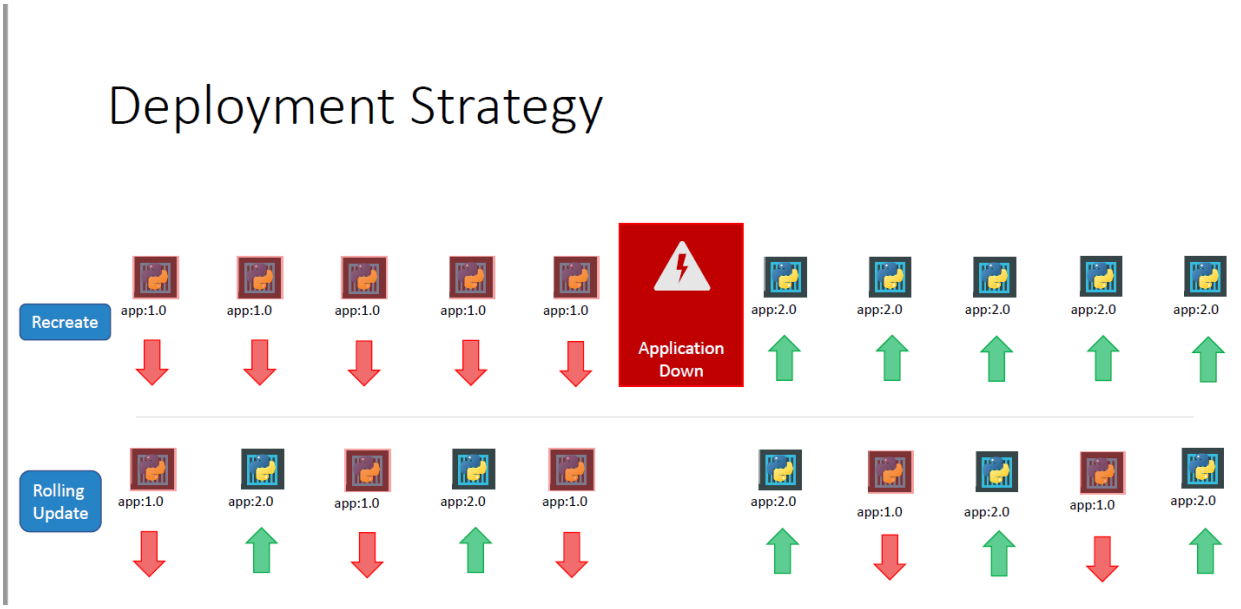
- One of the most popular existing operator framework is etcd operator.



- Operator are not part of syllabus of ckad.

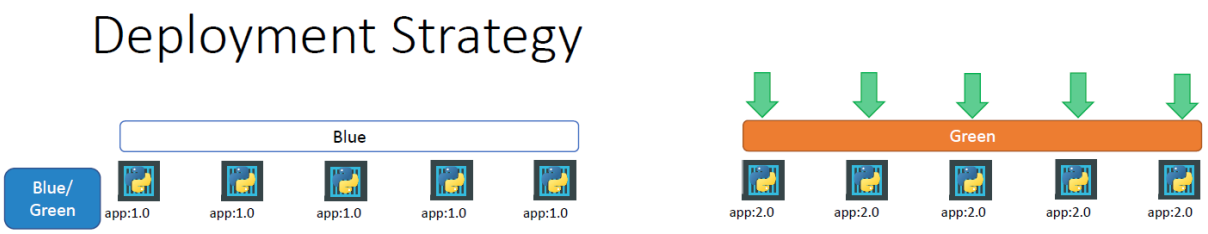
Deployment Strategy

- Till now we know 2 deployment strategy are there to perform upgrade. 1. rollingUpdate and 2. Recreate
- Rolling Update is default deployment strategy used where 1 at a time old pods are deleted and new pod are created. In this way application will remain up. After some time only new pods will be running to serve users.
- Recreate-> in this strategy all the pods of older version made down and when all newer pods are running as per rs, then it is used to serve users. In this approach application will be unavailable till newer pods are up and running.

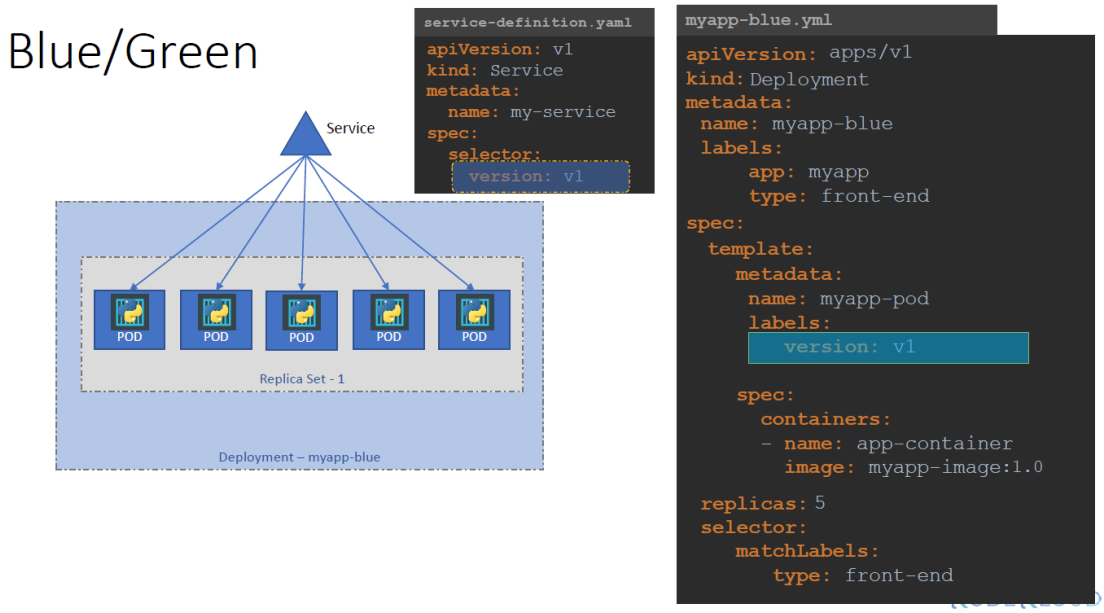


Blue Green Deployment

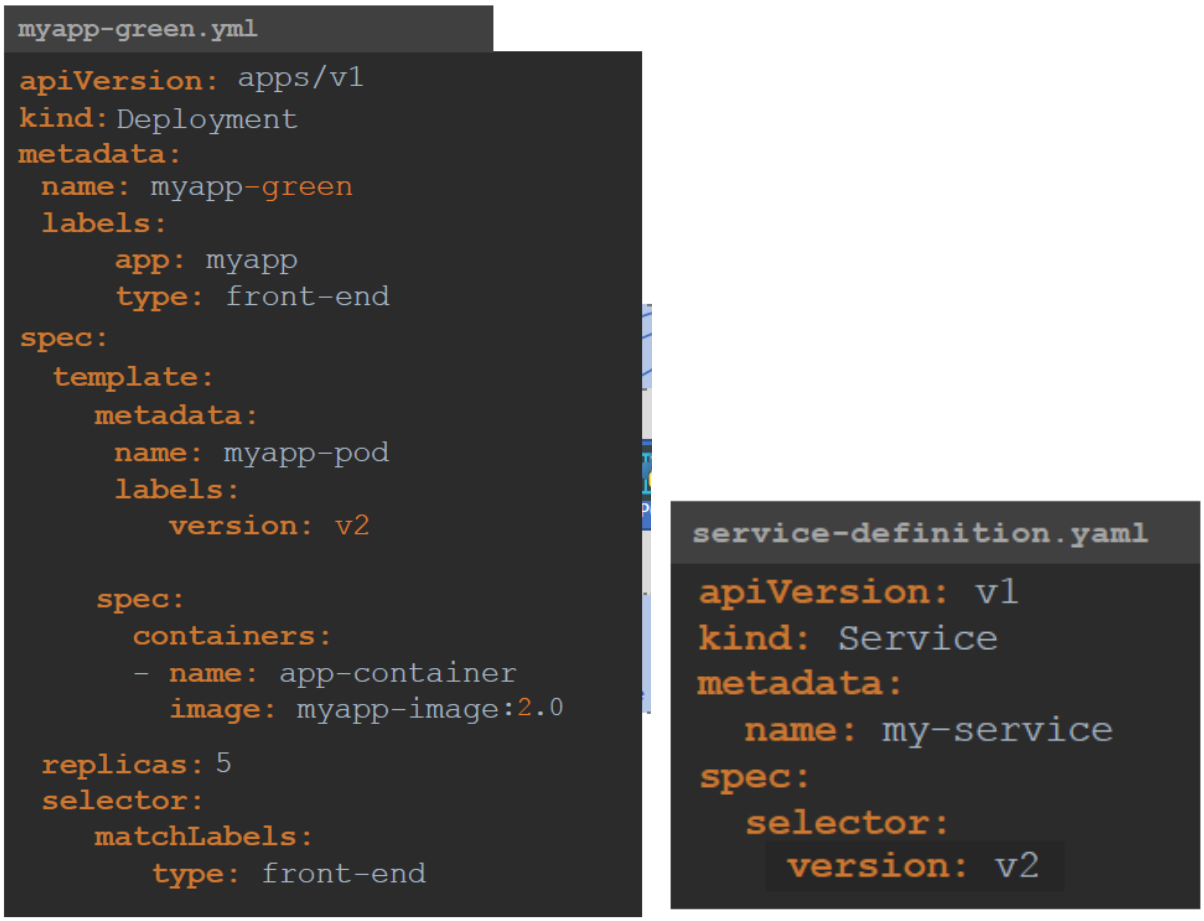
- It is a sort of design pattern or practice that we can follow. Still deployment can be created by two options only either rolling update or recreate. But we can follow this pattern to do deployment during upgrade.
- According to it, In system there will be two replica sets blue and green. Blue is the one which is having older version. Now green rs will be created which will run pods with newer application version. Traffic will be served to older pod only till the green deployment is up and ready. Once it is ready, Application traffic is switched to green deployment.



- How to implement such design using native k8s? for that we will use label and selector. Let's say we have a deployment with rs 5. And there is a service expose to access pods in that rs. So we will make label "v1" to all pods in deployment definition file and service label selector as "v1". Traffic will be served to 1 deployment only. Now we create another deployment now named "dep-green" with newer docker image and apply label as "v2" to pods of that deployment in rs. Once all pods in rs running. We update the service definition file to match label selector as v2 instead of v1. So now it will start serving second deployment only.
- Before deployment 2



- After Deployment 2 up and running we will update service definition.



- Service mesh like istio provide better control to above native approach. We will cover that later.

Canary Deployment

- It is again design practice/pattern which we can follow. Here with existing rs we will make only 1 pod of newer version up and running. Suppose old version rs has 5 pods and now 1 newer pod is also up and running. In such case load is distributed among all these 6 pods equally. And hence only 17% percent traffic will be served to newer version pod. Once we find everything up and running and working as per desired behaviour, then we replace older with newer ones. For replacing older pods with newer later we can use rolling update or recreate strategy whichever is suitable.

- By using service mesh like istio we can also limit percent of traffic to that 1 newer pod. For example if we want only 1 percent traffic to go to newer pod. So to achieve it using native k8s we need to run 99 older pods. as k8s can only distribute load among pods equally.
- How to achieve it using native k8s ? for that let's say we have deployment 1 (deployment primary) with rs as 5 and label of pod as "front-end" and we will expose service with label selector as "front-end". Now we will create new deployment named deployment2 ()with rs as 1 only and label of pod as "front-end". So now, same service will start serving request to 6 pods (5 old and 1 new).
- Later when we feel that everything is fine with newer version pod. Then in such case we will make version upgrade of pods in primary deployment with rolling update strategy and delete deployment2 which has single pod.

myapp-primary.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-primary
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        version: v1
        app: front-end
    spec:
      containers:
      - name: app-container
        image: myapp-image:1.0
  replicas: 5
  selector:
    matchLabels:
       type: front-end
```

service-definition.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: front-end
```

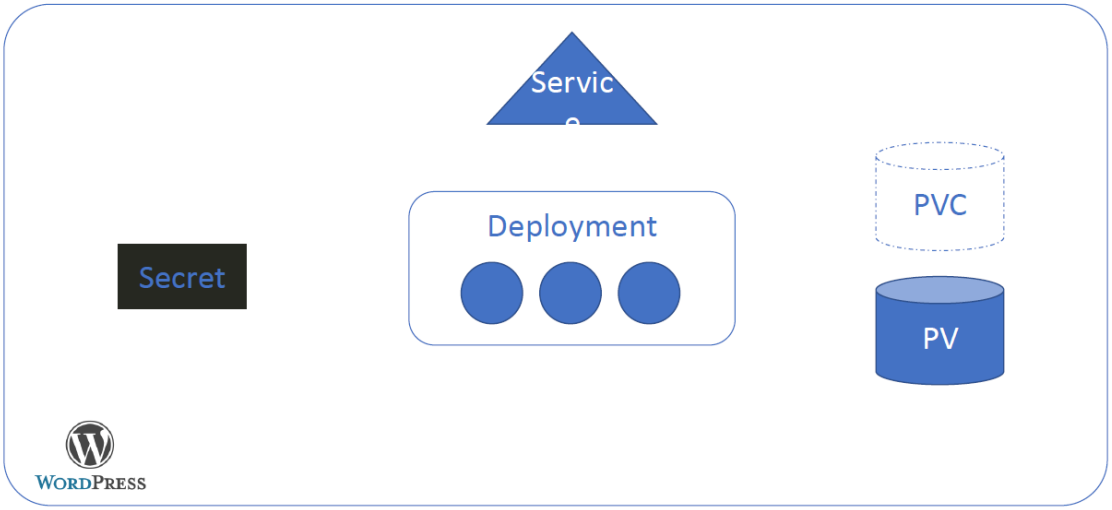
myapp-canary.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-canary
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        version: v2
        app: front-end
    spec:
      containers:
      - name: app-container
        image: myapp-image:2.0
  replicas: 1
  selector:
    matchLabels:
       type: front-end
```

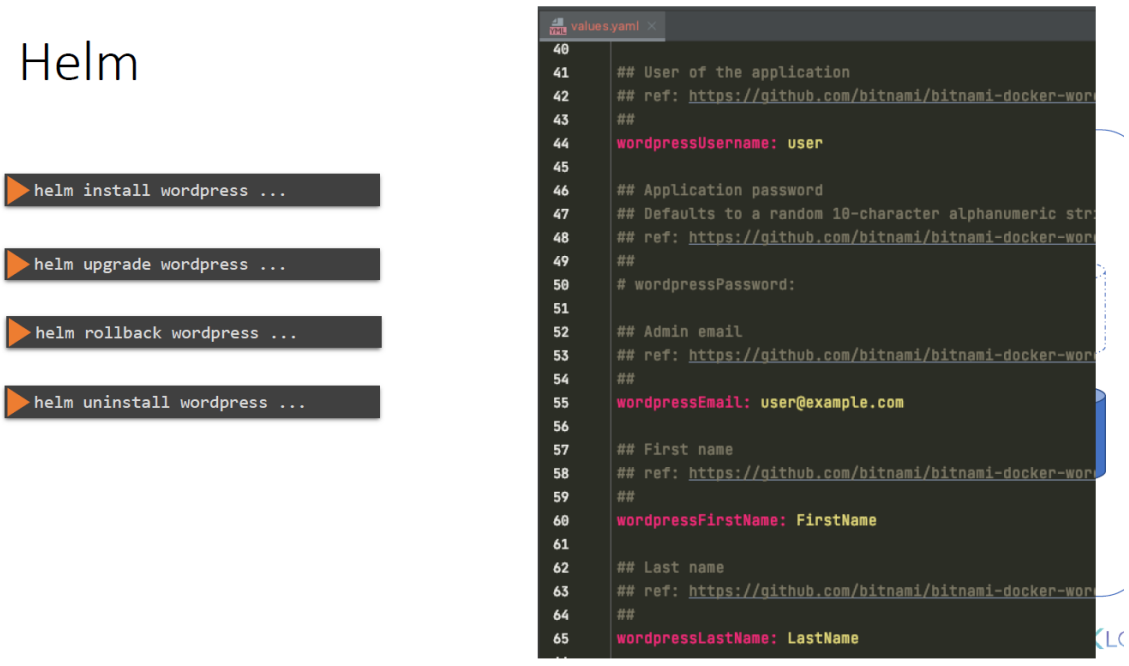
Helm

- Suppose we want to create very simple wordpress app with minimum features and want to deploy and manage it on k8s. we need deployment, service, pv, secret, pvc atleast. And for each object we need to create separate yaml file. And then run kubectl apply to each of them. And with application evolves we might need to for example increase pv. Or might need to increase rs. So we need to go and edit to deployment and do required changes carefully. Or later we want to update image version we are using. Then again go to yaml and edit it. Similarly in future if we want to delete all apps we need to delete all of them one by one. Even if instead of making separate yaml file of each object we create single file with all required conf. in such case it will become much tougher to edit and do the changes.

WordPress



- Here comes helm. It is also known as package manager for k8s. It will help me to maintain and group these k8s objects together as a single package wordpress here. As k8s does not think them as a group, it just makes sure required no of objects should be maintained in cluster. But it does not know that they all are part of wordpress and exists as whole.
- Lets Take example of game. It comes with installer and it's responsibility of a installer to put all desired config and files to desired location so that application can use it and run perfectly. We just need to run that installer and that's it. Similar work is done by helm, it packages k8s resources which make up our application into single unit. Example by using single command helm install wordpress we can install whole app even if it needs 100 objects. Similarly upgrade and rollback is also easy as again via single command we can do all. For that values.yaml file can be used where we specify all details required by helm.

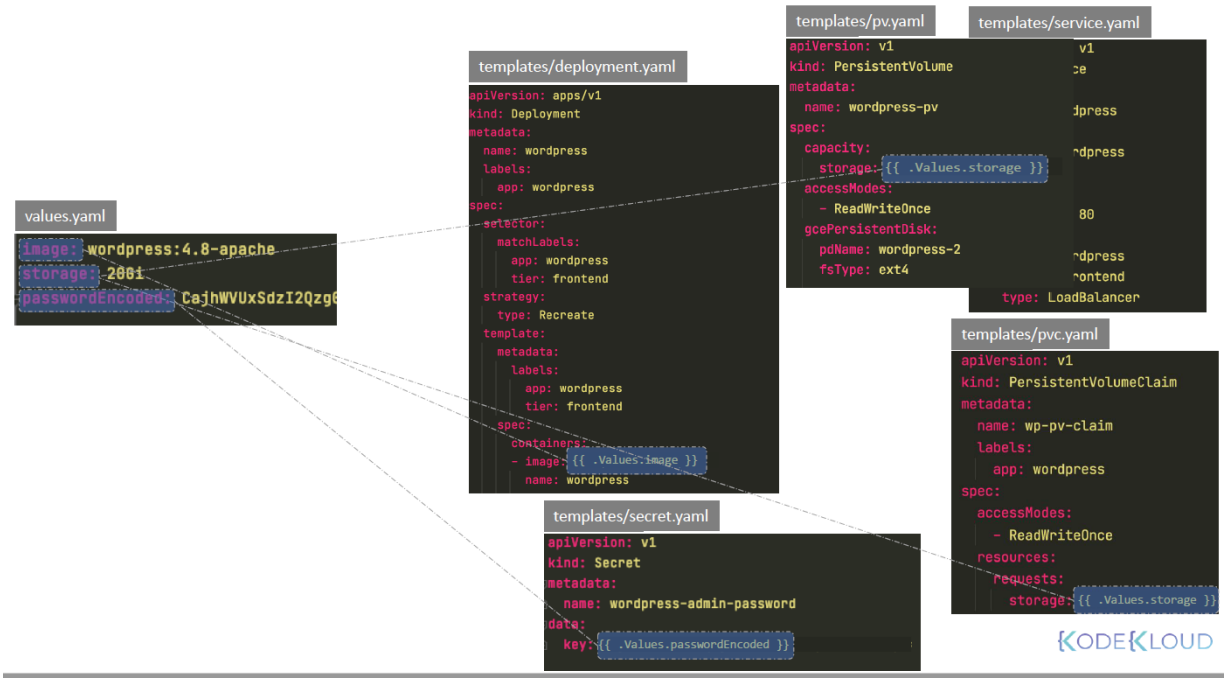


  - helm -h ->will list all options
  - helm env -> list down all env values.
  - Helm version

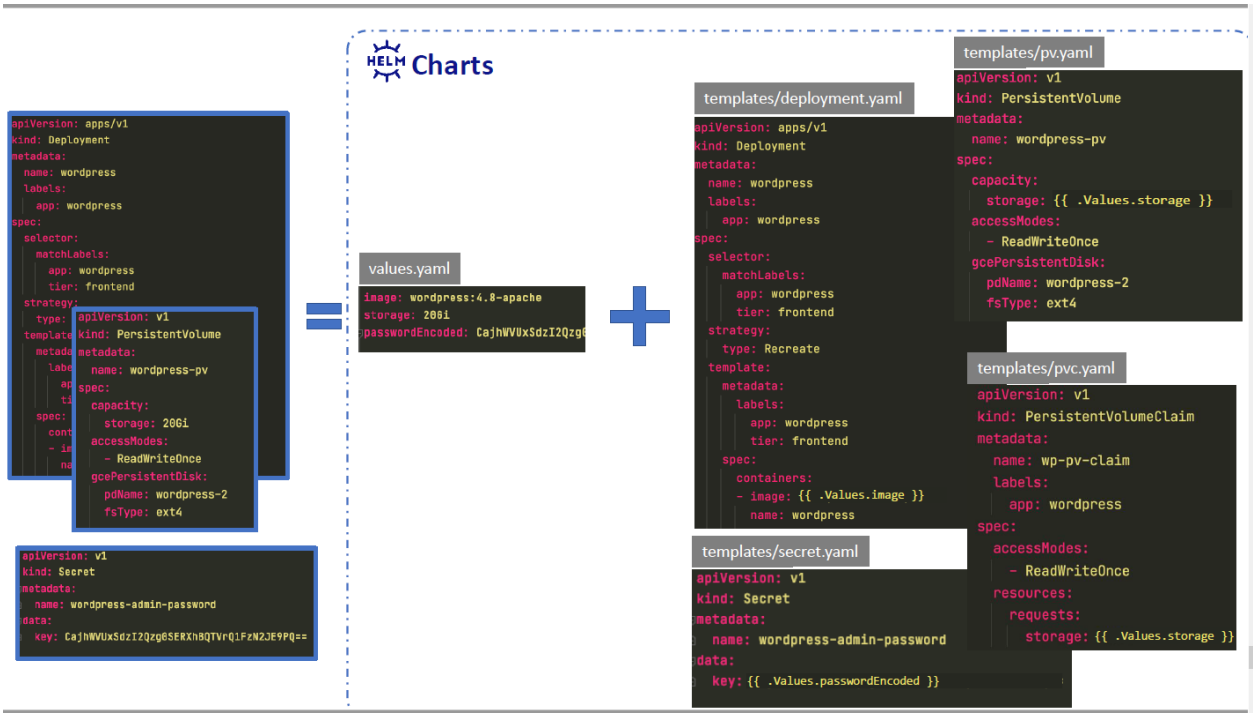
Helm Concepts

- Helm solves above challenges using charts. In above examples where we have we wordpress application, we have different yaml files which has most of content static and some of it might change in future like image, pv quota etc. these values might also change for different env. Like prod might have different pv and

develop env have different. So for that we can make use of values.yaml where we specify such variables (key,value pair) and can then use these variables in deployment.yaml , secret.yaml etc.



- So this combination of values.yaml and corresponding template files like deployment, secret etc forms a helm chart. Here we will have single chart for wordpress entire application which contains values.yaml, template files and chart.yaml.

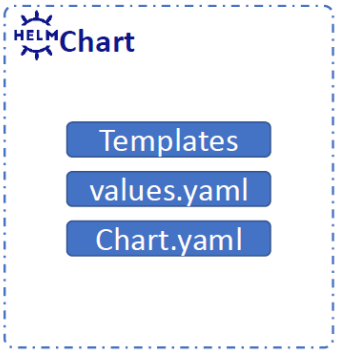


- Chart.yaml contains all the information of chart like name of chart, version, description, keywords, homepage, maintainers etc.

Helm Chart

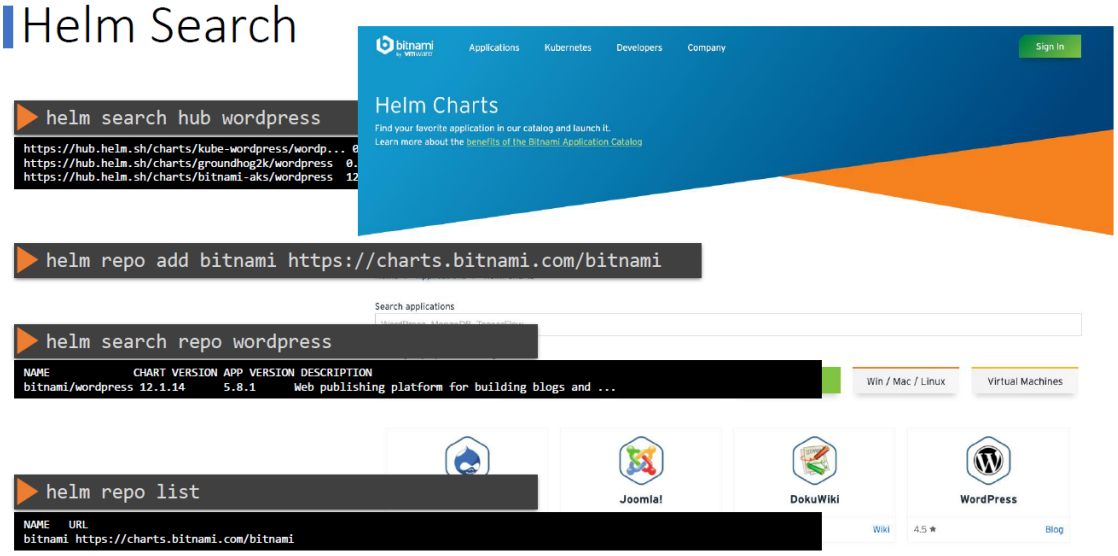
Chart.yaml

```yaml
apiVersion: v2
name: Wordpress
version: 9.0.3
description: Web publishing platform for building blogs and websites.
keywords:
  - wordpress
  - cms
  - blog
  - http
  - web
  - application
  - php
home: http://www.wordpress.com/
sources:
  - https://github.com/bitnami/bitnami-docker-wordpress
maintainers:
  - email: containers@bitnami.com
    name: Bitnami
```

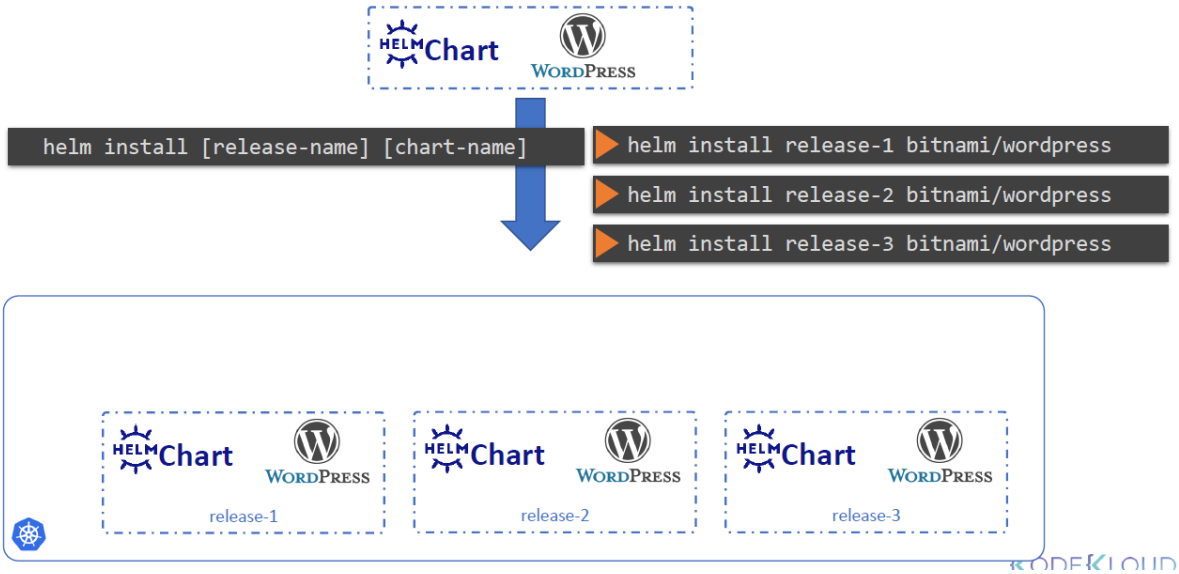


- You can create own chart for your own application or can take already created chart present in artifacthub. This hub is repo which stores various helm charts. You can use command helm search hub wordpress. To search for wordpress chart in hub.
- There are other repo also which we can use if we have chart available there which we wanted to use. like bitnami repo is one of them. For that you need to add bitnami repo and then use helm search repo wordpress.

Helm Search



- Now you have desired repo configured. You need to install helm chart for that use –
  Helm install release-name chartname
  When command is run. helm chart package of wordpress will be downloaded from repo and extracted and installed locally. Each installation of chart is called release and each release has release name.

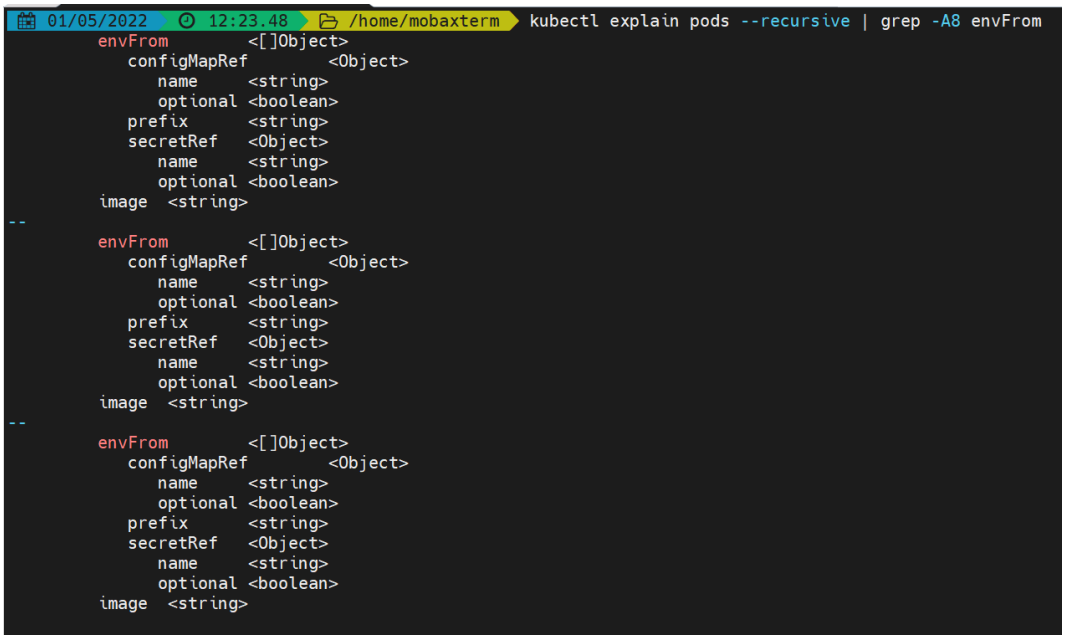


- Each release is isolated from other release. And hence we can install same application multiple times with different release name.
- To list installed packages run command helm list.
- To uninstall packages from helm run command helm uninstall releasename.
- Helm install, download and install package. To only download it and unzip it after downloading, Use helm pull --untar chartName
- Then we can verify downloaded content and once ready we can use helm install releaseName folder, It will install the chart from directory.

Exam Tips

1. Shortcut to find syntax of some fields that needed to write in definition file.

- Instead of searching documentation to find syntax of some attributes of definition file.you can use explain and grep combo ->
- kubectl explain pods --recursive | grep -A8 envFrom
- it will print 8 lines after keyword “envFrom”



- kubectl explain pods --recursive | grep -A8 -B5 envFrom
  print 5 lines before and 8 lines after envFrom

2. delete pod quickly to save time with force flag.

   kubectl delete pod nginx1 -n namespace --force.

   It will delete pod quickly. Don't do that in prod

3. delete all resources with matching label.

   kubectl delete all -l app=dep

   kubectl delete svc svc1 svc2 svc3

4. Get explanation of nested specific field value

   kubectl explain deploy.spec.strategy.rollingUpdate --recursive

   kubectl explain deploy.spec.strategy.rollingUpdate

```
root@IN-BYCK533:~# k explain deploy.spec.strategy.rollingUpdate
W0606 14:15:46.008713   11395 gcp.go:120] WARNING: the gcp auth plugin is deprecat
To learn more, consult https://cloud.google.com/blog/products/containers-kubernete
KIND:     Deployment
VERSION:  apps/v1

RESOURCE: rollingUpdate <Object>

DESCRIPTION:
     Rolling update config params. Present only if DeploymentStrategyType =
     RollingUpdate.

     Spec to control the desired behavior of rolling update.

FIELDS:
   maxSurge     <string>
     The maximum number of pods that can be scheduled above the desired number
     of pods. Value can be an absolute number (ex: 5) or a percentage of desired
     pods (ex: 10%). This can not be 0 if MaxUnavailable is 0. Absolute number
     is calculated from percentage by rounding up. Defaults to 25%. Example:
     when this is set to 30%, the new ReplicaSet can be scaled up immediately
     when the rolling update starts, such that the total number of old and new
     pods do not exceed 130% of desired pods. Once old pods have been killed,
     new ReplicaSet can be scaled up further, ensuring that total number of pods
     running at any time during the update is at most 130% of desired pods.

   maxUnavailable       <string>
     The maximum number of pods that can be unavailable during the update. Value
     can be an absolute number (ex: 5) or a percentage of desired pods (ex:
     10%). Absolute number is calculated from percentage by rounding down. This
     can not be 0 if MaxSurge is 0. Defaults to 25%. Example: when this is set
     to 30%, the old ReplicaSet can be scaled down to 70% of desired pods
     immediately when the rolling update starts. Once new pods are ready, old
     ReplicaSet can be scaled down further, followed by scaling up the new
     ReplicaSet, ensuring that the total number of pods available at all times
     during the update is at least 70% of desired pods.
```

5. kubectl api-resources command to list down short names of kubectl commands.

kubectl api-resources

```
root@controlplane ~ ➜ kubectl api-resources
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED
   KIND
bindings                                       v1                                     true
   Binding
componentstatuses                 cs           v1                                     false
   ComponentStatus
configmaps                        cm           v1                                     true
   ConfigMap
endpoints                         ep           v1                                     true
   Endpoints
events                            ev           v1                                     true
   Event
limitranges                       limits       v1                                     true
   LimitRange
namespaces                        ns           v1                                     false
   Namespace
nodes                             no           v1                                     false
   Node
persistentvolumeclaims            pvc          v1                                     true
   PersistentVolumeClaim
persistentvolumes                 pv           v1                                     false
```

it will list down all commands and there short names.

```
root@controlplane ~ ➜ kubectl api-resources | grep -i deploy
deployments                     deploy       apps/v1                                true
   Deployment

root@controlplane ~ ➜ kubectl api-resources | grep -i deploy | grep -i cron

root@controlplane ~ ✖ kubectl api-resources | grep -i cron
cronjobs                        cj           batch/v1beta1                          true
   CronJob

root@controlplane ~ ➜ kubectl api-resources | grep -i customres
customresourcedefinitions       crd,crds     apiextensions.k8s.io/v1                false
   CustomResourceDefinition
```

6. kubectl api-versions command.

Similarly kubectl api-versions command will list down all api with there versions available in current k8s release

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl api-versions
admissionregistration.k8s.io/v1
admissionregistration.k8s.io/v1beta1
apiextensions.k8s.io/v1
apiextensions.k8s.io/v1beta1
apiregistration.k8s.io/v1
apiregistration.k8s.io/v1beta1
apps/v1
authentication.k8s.io/v1
authentication.k8s.io/v1beta1
authorization.k8s.io/v1
authorization.k8s.io/v1beta1
autoscaling.gke.io/v1beta1
autoscaling.k8s.io/v1
autoscaling.k8s.io/v1beta2
autoscaling/v1
autoscaling/v2beta1
autoscaling/v2beta2
batch/v1
batch/v1beta1
certificates.k8s.io/v1
certificates.k8s.io/v1beta1
cilium.io/v2
cilium.io/v2alpha1
cloud.google.com/v1
cloud.google.com/v1beta1
coordination.k8s.io/v1
coordination.k8s.io/v1beta1
discovery.k8s.io/v1
discovery.k8s.io/v1beta1
dynatrace.com/v1alpha1
events.k8s.io/v1
extensions/v1beta1
flowcontrol.apiserver.k8s.io/v1beta1
internal.autoscaling.gke.io/v1alpha1
internal.autoscaling.k8s.io/v1alpha1
k8s.nginx.org/v1
k8s.nginx.org/v1alpha1
kubernetes-client.io/v1
metrics.k8s.io/v1beta1
networking.gke.io/v1
```

7. copy file from 1 path to other.

   Copy file my-kube-config present in current directory to folder. kube with name config present in current directory

   ```
   cp my-kube-config .kube/config
   ```

8. count no of pods with matching label shortcut

   ```
   controlplane ~ ➜ kubectl get pods -l env=dev --no-headers | wc -l
   7
   ```

9. Sample pod file with almost all content.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx1-dummy
  labels:
    app: myapppod
spec:
  nodeSelector:
    key1: value1
  automountServiceAccountToken: false
  serviceAccountName: default
  securityContext:
    runAsUser: 1001
  tolerations:
    - key: "color"
      value: "blue"
      operator: "Equal"
      effect: "NoSchedule"
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      resources:
        requests:
          memory: "1Gi"
          cpu: "0.5"
        limits:
          memory: "2Gi"
          cpu: "2"
      env:
        - name: password
          value: mysecretpassword
      envFrom:
        - configMapRef:
            name: configMap1
        - secretRef:
            name: secret1
      securityContext:
        runAsUser: 1002
        capabilities:
          add: ["All"]
          drop:
            - "CHOWN"
```

10. Search word in output of kubectl describe/explain command and display n lines after word.
    - kubectl describe pod podName | grep -i word -A7

```
04/05/2022 11:26.38 /home/mobaxterm kubectl describe pod keng03-dev01-ath87-oam87-0 -n keng03-dev01-ath87-1648
463927 | grep -i conditions -A 7
Conditions:
  Type                                Status
  cloud.google.com/load-balancer-neg-ready   True
  Initialized                         True
  Ready                               True
  ContainersReady                     True
  PodScheduled                        True
Volumes:
```

```
04/05/2022 11:26.47 /home/mobaxterm kubectl describe pod keng03-dev01-ath87-oam87-0 -n keng03-dev01-ath87-1648
463927 | grep -i conditions -A 7 -B 4
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-fhg7b (ro)
Readiness Gates:
  Type                                Status
  cloud.google.com/load-balancer-neg-ready   True
Conditions:
  Type                                Status
  cloud.google.com/load-balancer-neg-ready   True
  Initialized                         True
  Ready                               True
  ContainersReady                     True
  PodScheduled                        True
Volumes:
```

- kubectl explain pod --recursive | grep -i tolerations -A7 -B4

```
04/05/2022 11:31.31 /home/mobaxterm kubectl explain pod --recursive | grep -i tolerations -A7 -B4
     setHostnameAsFQDN <boolean>
     shareProcessNamespace     <boolean>
     subdomain <string>
     terminationGracePeriodSeconds     <integer>
     tolerations       <[]Object>
        effect <string>
        key    <string>
        operator       <string>
        tolerationSeconds      <integer>
        value  <string>
     topologySpreadConstraints <[]Object>
        labelSelector  <Object>
           matchExpressions    <[]Object>
              key      <string>
              operator <string>
              values   <[]string>
```

- kubectl get pod podName -o yaml | grep -i run

```
root@controlplane ~ ➜ kubectl get pod pod-with-defaults -o yaml | grep -i run
          f:runAsNonRoot: {}
          f:runAsUser: {}
    - echo I am running as user $(id -u)
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
    runAsNonRoot: true
    runAsUser: 1234
```

11. find shortnames of all available resources for imperative creation.

kubectl create

it will list down all details including available commands

12. Copy definition file of already running resource into new yaml

kubectl get pod webapp -o yaml > webapp.yaml

13. Check os version

cat /etc/*release*

14. Create own alias to save time

Like we can create alias for kubectl as ‘k’

alias k=kubectl

k get pods

15. Set default namespace

kubectl config set-context --current --namespace=dev

Kubectl commands

1. Check kubectl version

kubectl version

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl version
Client Version: version.Info{Major:"1", Minor:"21+", GitVersion:"v1.21.9-dispatcher", GitCommit:"2a8027f41d28b788b001389f3091c245cd0a9a60", GitTreeState:"clean", BuildDate:
"2022-01-21T20:34:16Z", GoVersion:"go1.16.12", Compiler:"gc", Platform:"windows/amd64"}
Server Version: version.Info{Major:"1", Minor:"20+", GitVersion:"v1.20.15-gke.2500", GitCommit:"750002971a60d8a06e0a403c52724257f0f68481", GitTreeState:"clean", BuildDate:"
2022-02-23T09:27:51Z", GoVersion:"go1.15.15b5", Compiler:"gc", Platform:"linux/amd64"}
```

2. Run pods on cluster

kubectl run nginx1 --image nginx -n namespace

it runs pod with name nginx1 from image nginx

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl run nginx1 --image nginx
pod/nginx1 created
```

3. describe pod

kubectl describe pod nginx1

From here we get all detail of pod like ip, namespace, containerId, ImageID, start time of pods, current status, events happened.

Some details are ->

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl describe pod nginx1
Name:         nginx1
Namespace:    default
Priority:     0
Node:         gke-keng01-use1-r100-use1-dev01-cl01--439809b4-fbgl/10.3.177.139
Start Time:   Fri, 22 Apr 2022 14:13:53 +0530
Labels:       run=nginx1
Annotations:  <none>
Status:       Running
IP:           10.72.0.184
IPs:
  IP:  10.72.0.184
Containers:
  nginx1:
    Container ID:   containerd://bbb8d3be672b046f246584bf295460d9ce828a0f43ce155da2b3b181fa839a99
    Image:          nginx
    Image ID:       docker.io/library/nginx@sha256:859ab6768a6f26a79bc42b231664111317d095a4f04e4b6fe79c
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Fri, 22 Apr 2022 14:13:56 +0530
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-nr9tp (ro)
Conditions:
```

4. delete pod

kubectl delete pod nginx1 -n namespace

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl delete pod nginx1
pod "nginx1" deleted
```

kubectl delete pod nginx1 -n namespace --force.

It will delete pod quickly. Don’t do that in prod

5. kubectl cluster-info

gives ip of master node. Control plane is same master node

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl cluster-info
Kubernetes control plane is running at https://10.251.214.50
GLBCDefaultBackend is running at https://10.251.214.50/api/v1/namespaces/kube-system/services/default-http-backend:http/proxy
KubeDNS is running at https://10.251.214.50/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
KubeDNSUpstream is running at https://10.251.214.50/api/v1/namespaces/kube-system/services/kube-dns-upstream:dns/proxy
Metrics-server is running at https://10.251.214.50/api/v1/namespaces/kube-system/services/https:metrics-server:/proxy
```

6. kubectl get nodes

it list all the nodes part of the cluster under a namespace.

kubectl get nodes -n namespaceName

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl get nodes  -n keng03-dev01-ath98-1649862354
NAME                                                STATUS   ROLES    AGE    VERSION
gke-keng01-use1-r100-use1-dev01-cl01--439809b4-cft7   Ready    <none>   6d2h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--439809b4-fbgl   Ready    <none>   6d3h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--439809b4-w5cc   Ready    <none>   6d2h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--439809b4-zimj   Ready    <none>   6d3h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--52498f70-1k1h   Ready    <none>   6d2h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--52498f70-duf2   Ready    <none>   6d1h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--52498f70-lq3j   Ready    <none>   6d1h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--52498f70-t7ux   Ready    <none>   6d1h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--52498f70-zbbi   Ready    <none>   6d2h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--fc45fa2d-hud2   Ready    <none>   6d3h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--fc45fa2d-rgak   Ready    <none>   6d3h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-cl01--fc45fa2d-tqty   Ready    <none>   6d3h   v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-np01--5ab89b9c-3935   Ready    <none>   6d     v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-np01--5ab89b9c-774k   Ready    <none>   6d     v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-np01--6eb76921-oes4   Ready    <none>   6d     v1.20.15-gke.2500
gke-keng01-use1-r100-use1-dev01-np01--8ea8e1f9-2yxl   Ready    <none>   6d     v1.20.15-gke.2500
```

7. kubectl get namespace

it print all namespace under cluster.

Kubectl get ns

Kubectl get namespace

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl get ns
NAME                                STATUS   AGE
default                             Active   232d
dynatrace                           Active   192d
forgerock-vs                        Active   63d
forgerock-vs-coe-ingress            Active   61d
gepoc                               Active   174d
harsh-test-authn                    Active   9d
kamran                              Active   37d
keng03-dev01-ath87-1648463927       Active   24d
keng03-dev01-ath98-1649862354       Active   27h
keng03-dev01-ath99-1649651815       Active   11d
keng03-dev01-ath99-1649779921       Active   8d
keng03-dev01-ecs07-1649828063       Active   9d
keng03-dev01-ecs66-1649084657       Active   17d
keng03-dev01-ecs75-1650611627       Active   31m
keng03-dev01-ecs76-1650519210       Active   26h
keng03-dev01-hca93-1642569638       Active   93d
keng03-dev01-hub83                  Active   16d
kube-node-lease                     Active   232d
kube-public                         Active   232d
kube-system                         Active   232d
monitoring                          Active   190d
nginx-ingress                       Active   158d
ops-sup                             Active   136d
sm                                  Active   219d
splunk                              Active   223d
test                                Active   192d
testdt                              Active   192d
```

8. kubectl get pods

display down list of pods under a namespace.

kubectl get pods -n namespace

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl get pods -n keng03-dev01-ath98-1649862354
NAME                                      READY   STATUS    RESTARTS   AGE
keng03-dev01-ath98-oam98-0                1/1     Running   0          20h
keng03-dev01-ath98-oam98-1                1/1     Running   0          13h
keng03-dev01-ath98-oam98-ic-9cd757bd5-46hvv   1/1     Running   0          26h
keng03-dev01-ath98-oam98-ic-9cd757bd5-h4gpw   1/1     Running   0          26h
keng03-dev01-ath98-oam98-ic-9cd757bd5-wzljn   1/1     Running   0          26h
keng03-dev01-ath98-odj-cfg98-0            1/1     Running   0          69m
keng03-dev01-ath98-odj-cfg98-1            1/1     Running   0          75m
keng03-dev01-ath98-odj-cts98-0            1/1     Running   0          68m
keng03-dev01-ath98-odj-cts98-1            1/1     Running   0          75m
keng03-dev01-ath98-odj-rcs98-0            1/1     Running   0          69m
keng03-dev01-ath98-odj-rcs98-1            1/1     Running   0          73m
keng03-dev01-ath98-odj-rut98-0            1/1     Running   0          68m
keng03-dev01-ath98-odj-rut98-1            1/1     Running   0          72m
keng03-dev01-ath98-odj-usr98-0            1/1     Running   0          69m
keng03-dev01-ath98-odj-usr98-1            1/1     Running   0          74m
```

kubectl get pods -n namespace -o wide

it will give extra info like on which node particular pod is deployed, ip of pod etc

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl get pods -n keng03-dev01-ath98-1649862354 -o wide
NAME                                        READY   STATUS    RESTARTS   AGE   IP           NODE                                                NOMINATED NODE   READIN
ESS GATES
keng03-dev01-ath98-oam98-0                  1/1     Running   0          20h   10.72.4.32   gke-keng01-use1-r100-use1-dev01-cl01--fc45fa2d-rgak   <none>           <none>

keng03-dev01-ath98-oam98-1                  1/1     Running   0          13h   10.72.0.179  gke-keng01-use1-r100-use1-dev01-cl01--439809b4-fbgl   <none>           <none>

keng03-dev01-ath98-oam98-ic-9cd757bd5-46hvv 1/1     Running   0          26h   10.72.1.52   gke-keng01-use1-r100-use1-dev01-cl01--52498f70-lq3j   <none>           <none>

keng03-dev01-ath98-oam98-ic-9cd757bd5-h4gpw 1/1     Running   0          26h   10.72.2.224  gke-keng01-use1-r100-use1-dev01-cl01--fc45fa2d-hud2   <none>           1/1
keng03-dev01-ath98-oam98-ic-9cd757bd5-wzljn 1/1     Running   0          26h   10.72.3.156  gke-keng01-use1-r100-use1-dev01-cl01--439809b4-zimj   <none>           1/1
keng03-dev01-ath98-odj-cfg98-0              1/1     Running   0          71m   10.72.1.117  gke-keng01-use1-r100-use1-dev01-cl01--52498f70-duf2   <none>           1/1
keng03-dev01-ath98-odj-cfg98-1              1/1     Running   0          77m   10.72.2.229  gke-keng01-use1-r100-use1-dev01-cl01--fc45fa2d-hud2   <none>           1/1
keng03-dev01-ath98-odj-cts98-0              1/1     Running   0          70m   10.72.1.60   gke-keng01-use1-r100-use1-dev01-cl01--52498f70-lq3j   <none>           1/1
keng03-dev01-ath98-odj-cts98-1              1/1     Running   0          76m   10.72.2.230  gke-keng01-use1-r100-use1-dev01-cl01--fc45fa2d-hud2   <none>           1/1
keng03-dev01-ath98-odj-rcs98-0              1/1     Running   0          71m   10.72.1.116  gke-keng01-use1-r100-use1-dev01-cl01--52498f70-duf2   <none>           1/1
keng03-dev01-ath98-odj-rcs98-1              1/1     Running   0          75m   10.72.0.181  gke-keng01-use1-r100-use1-dev01-cl01--439809b4-fbgl   <none>           1/1
keng03-dev01-ath98-odj-rut98-0              1/1     Running   0          70m   10.72.1.118  gke-keng01-use1-r100-use1-dev01-cl01--52498f70-duf2   <none>           1/1
keng03-dev01-ath98-odj-rut98-1              1/1     Running   0          74m   10.72.0.182  gke-keng01-use1-r100-use1-dev01-cl01--439809b4-fbgl   <none>           1/1
keng03-dev01-ath98-odj-usr98-0              1/1     Running   0          70m   10.72.1.119  gke-keng01-use1-r100-use1-dev01-cl01--52498f70-duf2   <none>           1/1
keng03-dev01-ath98-odj-usr98-1              1/1     Running   0          76m   10.72.0.180  gke-keng01-use1-r100-use1-dev01-cl01--439809b4-fbgl   <none>           1/1
```

9. kubectl apply -f yamlFileName.yaml --record

go to folder where yaml file present and type above command. If yaml contain pod definition it will run pod following config mention in file.

```
C:\kush\study\kubernetes\practicek8s>kubectl apply -f pod.yaml
pod/nginx1 created

C:\kush\study\kubernetes\practicek8s>kubectl get pods
NAME     READY   STATUS    RESTARTS   AGE
nginx1   1/1     Running   0          11s
```

You can also use create instead of apply keyword to create new pod. But apply can be used to run modified deployment definition.

You can also specify optional --record option. In such case in history of rollout it will show detail also in change-cause->

```
C:\kush\study\kubernetes\practicek8s>kubectl apply -f deployment-def.yaml --record
deployment.apps/myapp-deployment created

C:\kush\study\kubernetes\practicek8s>kubectl rollout history deployment/myapp-deployment
deployment.apps/myapp-deployment
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=deployment-def.yaml --record=true
```

10. kubectl get replicaset

kubectl get rs

kubectl get replicaset

it list down all the rs in cluster under namespace specified.

11. kubectl delete replicaset

kubectl delete rs rsname

kubectl delete replicaset rsname

12. kubectl scale replicaset

kubectl scale --replicas=4 -f replicasetDefinitionfile.yaml

kubectl scale rs --replicas=4 rsName

both will make rs to 4.

13. kubectl edit rs rsname

it opens rsname replicaset in vim editor by default and there you can make changes and when you save it will automatically restart rs with new config. Please note that file opened in vi has more fields then original one as it is created by Kubernetes for us to edit things using our original yaml definition file. By it you can edit service/deployment/pod/sts anything dynamically.

Specify --record option if want to record history where it will show command edit.

14. kubectl get all

it list down all the Kubernetes objects present in the system under given namespace.

```
C:\kush\study\kubernetes\practicek8s>kubectl get all
NAME                                    READY   STATUS    RESTARTS   AGE
pod/myapp-deployment-68c5c84d7-kd7jc    1/1     Running   0          12s
pod/myapp-deployment-68c5c84d7-r892m    1/1     Running   0          12s

NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP   10.44.0.1    <none>        443/TCP   234d

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/myapp-deployment    2/2     2            2           13s

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/myapp-deployment-68c5c84d7     2         2         2       13s
```

15. kubectl set image deployment

if we want to change image in already created deployment use command->

kubectl set image deployment myapp-deployment nginx=nginx:1.9.1

```
C:\kush\study\kubernetes\practicek8s>kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
deployment.apps/myapp-deployment image updated
```

16. kubectl rollout status

suppose you have done version upgrade and now you want check status.

to check rollout status ->

kubectl rollout status deploy myapp-deployment

```
C:\kush\study\kubernetes\practicek8s>kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
deployment.apps/myapp-deployment image updated
```

```
C:\kush\study\kubernetes\practicek8s>kubectl rollout status deployment/myapp-deployment
deployment "myapp-deployment" successfully rolled out
```

17. kubectl rollout history

suppose you have done version upgrade and now you want check history.

to check rollout history ->

kubectl rollout history deployment myapp-deployment

```
C:\kush\study\kubernetes\practicek8s>kubectl rollout history deployment/myapp-deployment
deployment.apps/myapp-deployment
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
```

To get change-cause value, add **--record** attribute while performing action. It will display now command executed in that step.

```
C:\kush\study\kubernetes\practicek8s>kubectl apply -f deployment-def.yaml --record
deployment.apps/myapp-deployment created

C:\kush\study\kubernetes\practicek8s>kubectl rollout history deployment/myapp-deployment
deployment.apps/myapp-deployment
REVISION  CHANGE-CAUSE
1         kubectl apply --filename=deployment-def.yaml --record=true
```

- **Using the --revision flag:**
  Here the revision 1 is the first version where the deployment was created.

  You can check the status of each revision individually by using the **--revision flag**:

```
1. master $ kubectl rollout history deployment nginx --revision=1
2. deployment.extensions/nginx with revision #1
3.
4. Pod Template:
5.  Labels:    app=nginx    pod-template-hash=6454457cdb
6.  Containers:  nginx:  Image:   nginx:1.16
7.   Port:    <none>
8.   Host Port: <none>
9.   Environment:    <none>
10.  Mounts:   <none>
11. Volumes:   <none>
12. master $
```

18. kubectl rollout undo

to rollback to previous version. i.e. go back to older replica set.

to undo rollout ->

kubectl rollout undo deployment myapp-deployment

1. before undo command->

```
C:\kush\study\kubernetes\practicek8s>kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-68c5c84d7    0         0         0       5m47s
myapp-deployment-85c966c768   2         2         2       4m31s
```

2. after undo command->

```
C:\kush\study\kubernetes\practicek8s>kubectl rollout undo deployment/myapp-deployment
deployment.apps/myapp-deployment rolled back
```

```
C:\kush\study\kubernetes\practicek8s>kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-68c5c84d7    2         2         2       9m44s
myapp-deployment-85c966c768   0         0         0       8m28s
```

It can be seen old rs is now ready.

```
C:\kush\study\kubernetes\practicek8s>kubectl rollout history deployment/myapp-deployment
deployment.apps/myapp-deployment
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

You can also see revision 1 is not present in history as it is same as rev 3 and hence Kubernetes deleted rev 1.

19. kubectl get service

kubectl get svc

kubectl get service

```
C:\kush\study\kubernetes\practicek8s>kubectl get service
NAME            TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
kubernetes      ClusterIP   10.44.0.1      <none>        443/TCP   235d
myapp-service   ClusterIP   10.44.12.125   <none>        80/TCP    52s
```

20. kubectl describe service ServiceName

kubectl describe service ServiceName

kubectl describe svc ServiceName

```
C:\kush\study\kubernetes\practicek8s>kubectl describe service myapp-service
Name:              myapp-service
Namespace:         default
Labels:            tier=frontend
Annotations:       cloud.google.com/neg: {"ingress":true}
Selector:          app=myapppod
Type:              ClusterIP
IP Families:       <none>
IP:                10.44.12.125
IPs:               10.44.12.125
Port:              <unset>  80/TCP
TargetPort:        80/TCP
Endpoints:         10.72.0.147:80,10.72.1.140:80
Session Affinity:  None
Events:            <none>
```

21. kubectl get svc,pods

separate comma with each k8s objects you want to list down

```
C:\kush\study\kubernetes\practicek8s\votingapp>kubectl get svc,pod
NAME                     TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
service/db               ClusterIP   10.44.6.166    <none>        5432/TCP       20m
service/kubernetes       ClusterIP   10.44.0.1      <none>        443/TCP        235d
service/redis            ClusterIP   10.44.0.106    <none>        6379/TCP       35m
service/result-service   NodePort    10.44.14.253   <none>        80:30005/TCP   34m
service/voting-service   NodePort    10.44.13.221   <none>        80:30004/TCP   37m

NAME                 READY   STATUS    RESTARTS   AGE
pod/postgres-pod     1/1     Running   0          20m
pod/redis-pod        1/1     Running   0          35m
pod/result-app-pod   1/1     Running   1          34m
pod/voting-app-pod   1/1     Running   0          37m
pod/worker-app-pod   1/1     Running   0          20m
```

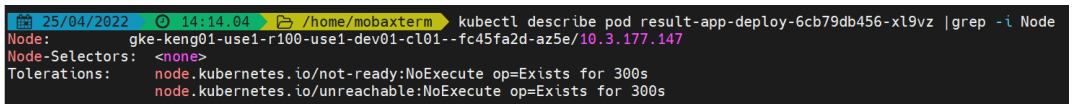
22. kubectl delete pod -l app=my-app

it will delete all pods with label app: my-app

```
C:\kush\study\kubernetes\practicek8s\votingapp>kubectl delete pods -l app=demo-voting-app
pod "postgres-pod" deleted
pod "redis-pod" deleted
pod "result-app-pod" deleted
pod "voting-app-pod" deleted
pod "worker-app-pod" deleted
```

23. kubectl get pod podName | grep -i Node

to get the text search where Node keyword is used. It will give node ip.



24. Formatting Output with kubectl -o

kubectl [command] [TYPE] [NAME] -o <output_format>
Here are some of the commonly used formats:

1. `-o json` Output a JSON formatted API object.
2. `-o name` Print only the resource name and nothing else.
3. `-o wide` Output in the plain-text format with any additional information.
4. `-o yaml` Output a YAML formatted API object.

Example->

```
master $ kubectl create namespace test-123 --dry-run -o json
{
    "kind": "Namespace",
    "apiVersion": "v1",
    "metadata": {
        "name": "test-123",
        "creationTimestamp": null
    },
    "spec": {},
    "status": {}
}
```

25. --all-namespace OR -A

To display k8s objects like rs, svc, pods, deployment, sts running under all namespaces we use this attribute.

Example ->

kubectl get pods --all-namespaces

kubectl get all -A

26. Kubectl create configmap

kubectl create configmap name1 --from-literal=APP_COLOR=blue --from-literal=type=frontend

kubectl create configmap name2 --from-file=sample.properties

Where sample.properties contains key: value entries

27. Kubectl get configmap

kubectl get configmap

kubectl get cm

28. Attach label to node.

kubectl label node nodeName key1=value1

```
root@controlplane:~# kubectl label node node01 color=blue
node/node01 labeled
```

29. Show all label present at node/pod.

kubectl get node node01 --show-labels
kubectl get pod --show-labels

30. Exec into pod to check logs.

kubectl exec -it podName -- command

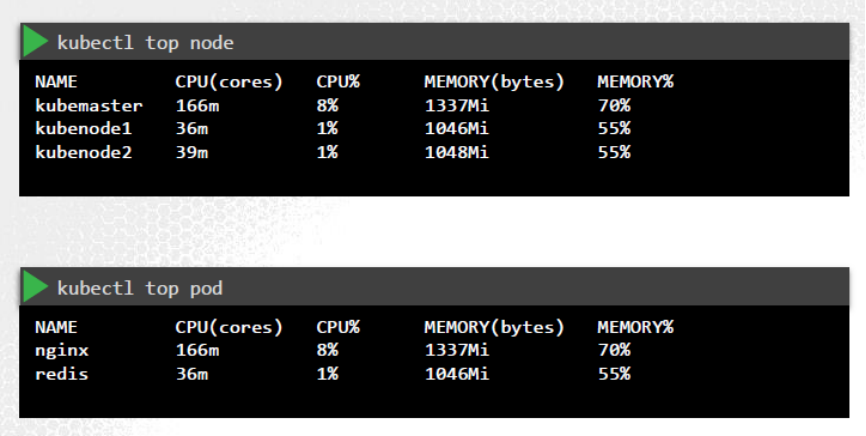
kubectl exec -it myapp -- cat /log/app.log

31. check container logs.

- kubectl logs podname containername
- kubectl logs podname -> if 1 container is running in a pod, then container name not needed.
- kubectl logs -f podname -> print live log of app.
- Kubectl describe pod podname command can be used to get container names.

32. check performance metric of pod and node

- Use the kubectl top pod/node command to view performance metrics of pods in kubernetes.



33. kubectl get resources via matching label

kubectl get pods --selector app=APP1

kubectl get pods -l app=APP1,tier=db,color=blue

it will return only those pod which has label app=APP1 and tier=db and color=blue

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl run nginx --image=nginx -l app=myapp
pod/nginx created

C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl get pods
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          7s
```

```
C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl get pods --show-labels
NAME    READY   STATUS    RESTARTS   AGE   LABELS
nginx   1/1     Running   0          17s   app=myapp

C:\Users\kush.gupta\AppData\Local\Google\Cloud SDK>kubectl delete pod -l app=myapp
pod "nginx" deleted
```

Get pods count with matching label env=dev

```
controlplane ~ ➜ kubectl get pods -l env=dev --no-headers | wc -l
7
```

34. kubectl create ingress

Format - kubectl create ingress <ingress-name> --rule="host/path=service:port"

Example - kubectl create ingress ingress-test --rule="wear.my-online-store.com/wear*=wear-service:80"

35. kubectl get networkpolicy

kubectl get netpol

kubect get networkpolicy

36. kubectl replace -f webapp.yaml --force

It will remove the existing resource and will replace it with the new one from the given manifest file.

37. kubectl get pvc pvcname

It is same as kubectl get persistentvolumeclaim command

```
root@controlplane ~ ➜ kubectl get pvc
NAME          STATUS        VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
claim-log-1   Terminating   pv-log   100Mi      RWX                           5m47s
```

38. kubectl get pv pvname

It is same as kubectl get persistentvolume command

```
root@controlplane ~ ➜ kubectl get pv
NAME     CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM                 STORAGECLAS
S   REASON   AGE
pv-log   100Mi      RWX            Retain           Released   default/claim-log-1
             16m
```

39. how to update something in kube-apiserver

1. kube-apiserver file is present in master node under /etc/Kubernetes/manifests.
2. below example shows addition of flag --runtime-config=rbac.authorization.k8s.io/v1alpha1 in existing kube-apiserver manifest definition file.

As a good practice, take a backup of that `apiserver` manifest file before going to make any changes.
In case, if anything happens due to misconfiguration you can replace it with the backup file.

```
root@controlplane:~# cp -v /etc/kubernetes/manifests/kube-apiserver.yaml
/root/kube-apiserver.yaml.backup
```

Now, open up the `kube-apiserver` manifest file in the editor of your choice. It could be `vim` or `nano`.

```
root@controlplane:~# vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

Add the `--runtime-config` flag in the `command` field as follows :-

```
 - command:
    - kube-apiserver
    - --advertise-address=10.18.17.8
    - --allow-privileged=true
    - --authorization-mode=Node,RBAC
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --runtime-config=rbac.authorization.k8s.io/v1alpha1 --> This one
```

After that `kubelet` will detect the new changes and will recreate the `apiserver` pod.
It may take some time.

```
root@controlplane:~# kubectl get po -n kube-system
```

Check the status of the `apiserver` pod. It should be in running condition