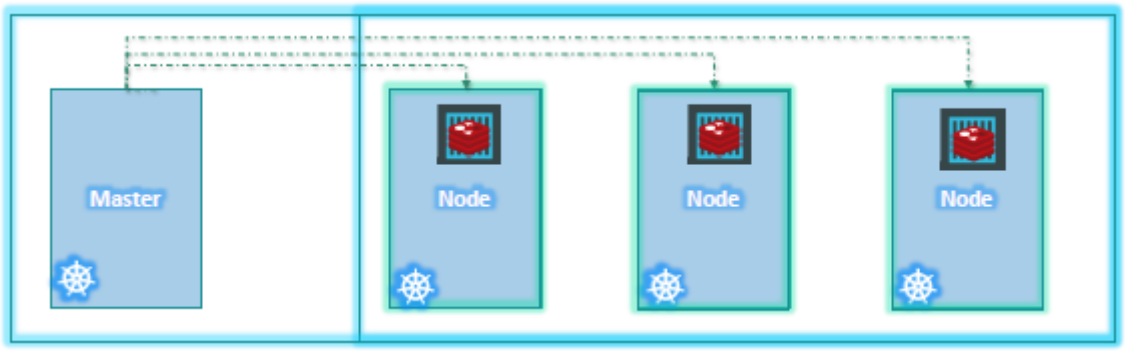
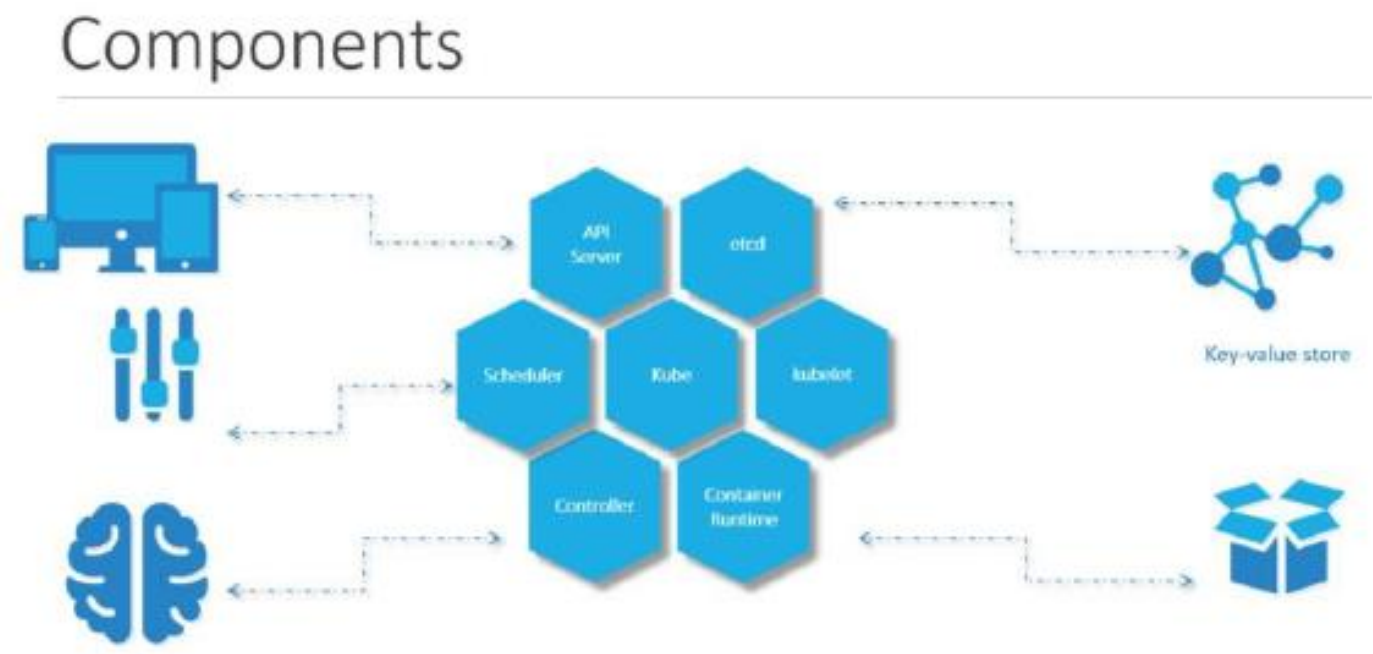
K8S

Architecture





When you install Kubernetes on a System, you are actually installing the following components. An API Server. An ETCD service. A kubelet service. A Container Runtime, Controllers and Schedulers.

°The **API server** acts as the front-end for kubernetes. The users, management devices, Command line interfaces all talk to the API server to interact with the kubernetes cluster.

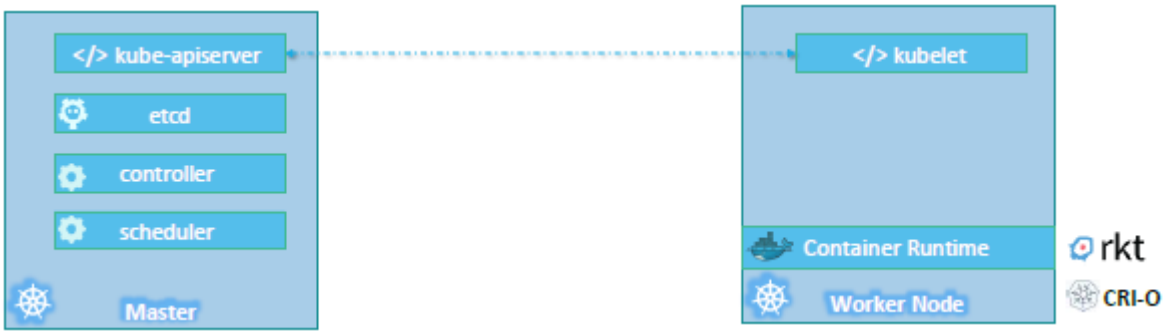
°Next is the **ETCD key store**. ETCD is a distributed reliable key-value store used by kubernetes to store all data used to manage the cluster. Think of it this way, when you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster in a distributed manner. ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.

°The **scheduler** is responsible for distributing work or containers across multiple nodes. It looks for newly created containers and assigns them to Nodes.

°The **controllers** are the brain behind orchestration. 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.

°The **container runtime** is the underlying software that is used to run containers. In our case it happens to be Docker (it can also be containerd or other).

°And finally **kubelet** is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.



You can use **kubeadm** to setup your k8s cluster instead of configuring each container one by one.

For that you will first have to configure docker (or another container manager) and install kubeadm and kubelet (which will start pods and containers etc…) on all your machines (master and worker). Then you will have to initialize the master node with kubeadm => this will download and setup all the requisite containers/pods for the master node.

Before joining the workers you will have to setup network prerequisite for the pods to communicate through all the nodes (calico, flannel…).

And finally the worker nodes can join the k8s cluster.

Usefull commands:

```
kubectl get nodes

kubectl cluster-info

kubectl run my-image
```

Install microK8s

```
apt install snapd

sudo snap set system proxy.http="url_proxy "

sudo snap set system proxy.https="url_proxy"
```

```
sudo snap install core
sudo snap install microk8s --classic --channel=1.25
sudo usermod -a -G microk8s $USER
sudo chown -f -R $USER ~/.kube
```

Restart your vm and test (the start may take some time):

```
microk8s start
microk8s kubectl get nodes
microk8s stop
```

Configure proxy :

```
sudo nano /etc/environment
```

Then paste :

HTTPS_PROXY=url_proxy

HTTP_PROXY=url_proxy

https_proxy=url_proxy

http_proxy=url_proxy

NO_PROXY=10.0.0.0/8,192.168.0.0/16,127.0.0.0/8,172.16.0.0/16

no_proxy=10.0.0.0/8,192.168.0.0/16,127.0.0.0/8,172.16.0.0/16

Follow the procedures to clean all warnings and be sure that microk8s is running properly :

```
microk8s inspect
```

Enable cgroups :

```
sudo nano /etc/default/grub
```

edit :

GRUB_CMDLINE_LINUX="cgroup_enable=memory cgroup_memory=1 swapaccount=1 systemd.unified_cgroup_hierarchy=0"

Usefull commands :

```
microk8s ctr images list
microk8s ctr images rm “your-image”
```

Shortcut/Aliases

po for PODs
rs for ReplicaSets
deploy for Deployments
svc for Services
ns for Namespaces
netpol for Network policies
pv for Persistent Volumes
pvc for PersistentVolumeClaims
sa for service accounts

You can get the shortcuts an all your resources of the kub version you are using with the command:

```
kubectl api-resources
```

Formatting Output with kubectl

The default output format for all **kubectl** commands is the human-readable plain-text format.

The -o flag allows us to output the details in several different formats.

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.

Here are some useful examples:

- Output with JSON format:

1. master $ kubectl create namespace test-123 --dry-run -o json
2. {
3. "kind": "Namespace",
4. "apiVersion": "v1",

```
    "metadata": {
      "name": "test-123",
      "creationTimestamp": null
    },
    "spec": {},
    "status": {}
}
master $
```

- Output with YAML format:

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

- Output with wide (additional details):

Probably the most common format used to print additional details about the object:

```
master $ kubectl get pods -o wide
NAME     READY   STATUS    RESTARTS   AGE    IP          NODE     NOMINATED
NODE   READINESS GATES
busybox  1/1     Running   0          3m39s  10.36.0.2   node01   <none>          <none>
ningx    1/1     Running   0          7m32s  10.44.0.1   node03   <none>          <none>
redis    1/1     Running   0          3m59s  10.36.0.1   node01   <none>          <none>
```

Pods

A pod is the smallest object you can create with kubernetes. They are deployed in the worker nodes.

A pod is a single instance of an application and encapsulate containers.

Create pods :

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

Check pods :

```
kubectl get pods
```

All namespaces :

```
kubectl get pods –A
```

More informations :

```
kubectl get pods –o wide
```

Pod details :

```
kubectl describe pod « mypod »
```

Delete Pod :

```
kubectl delete pod your-pod
```

Edit image of a container in a pod :

```
kubectl set image pod/nginx nginx=nginx:9.9.1
```

yaml example file (nginx.yml):

```
apiVersion: v1
kind: Pod


metadata:
  name: nginx-pod
  labels:
    type: front-end
    app: nginx

spec:
  containers:
  - name: nginx
    image: nginx
    env:
      - name: PASSWORD
        value: mypassword
```

Take note that the labels dictonary can contain any kind of key value, it’s only to tag your object and make it easier to manipulate later.

Create from yaml :

```
kubectl apply –f nginx.yaml
```

Generate a yaml file example with dry run :

```
kubectl run redis -–image=redis -–dry-run –o yaml > redis.yaml
```

Generate a file from an existing pod :

```
kubectl get pod <pod-name> -o yaml > pod-definition.yaml
```

Edit an existing pod :

```
kubectl edit pod < pod-name >
```

Note: this will not modify the existing yaml file.

Open a container port:

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
  - name: webserver
    image: nginx:latest
    ports:
    - containerPort: 80
```

OR

```
kubectl run webserver --image=nginx:latest --port=80
```

Commands and Arguments:

If you want to override or execute a command on the image you use (override or create a CMD instruction) :

```
apiVersion: v1
kind: Pod
metadata:
 name: ubuntu-sleeper
spec:
 containers:
 - name: ubuntu
```

```
    image: ubuntu
    command: [“sleep”, “5000”]
```

OR

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    command: [“sleep”]
    args: [“5000”]
```

OR

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper
spec:
  containers:
  - name: ubuntu
    image: ubuntu
    command:
      - “sleep”
      - “5000”
```

If you already have an ENTRYPOINT and a CMD in a Dockerfile :

Let’s take a simple example of a docker image (see entrypoint part of the docker course):

```
FROM ubuntu
```

```
ENTRYPOINT [“sleep”]
CMD [“5”]
```

Quick reminder : you can precise the number of seconds you want the sleep command to run (ENTRYPOINT), otherwise it will be 5 by default (CMD):

```
docker run ubuntu-sleeper 10
```

Using kubernetes pods, you can specify the arguments (here the number of seconds) with the args instruction:

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
  labels:
    app: ubuntu-sleeper
specs:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      args: [“10”]
```

If we want to override the ENTRYPOINT, we have to use the command instruction :

```
apiVersion: v1
kind: Pod
metadata:
  name: ubuntu-sleeper-pod
  labels:
    app: ubuntu-sleeper
specs:
  containers:
    - name: ubuntu-sleeper
      image: ubuntu-sleeper
      command: [“my_new_command”]
```

```
    args: [“10”]
```

NOTE: in the case the image used has an entrypoint, the command instruction in kubernetes yaml file will override the entrypoint instruction of the docker image used and the args kubernetes.

As in docker, you can run a pod and override the command and the arguments used by the image (using the --command option followed by -- to separate the kubectl command options from the args and command passed to the image):

```
kubectl run nginx --image=nginx --command -- my_cmd my_arg1 my_arg2 … my_argN
```

Ex (cmd and args):

```
kubectl run webapp-green --image=kodekloud/webapp-color --command -- python app2.py --color green
```

Execute command inside Pod or inside container

```
kubectl exec -it pod-name -- /bin/bash
```

`kubectl exec webapp -- cat /log/app.log`

We need to use -c to specify the container name:

`kubectl exec tomcat-nginx-78d457fd5d-446wx -c tomcat8 -- ls -lrt /opt/tomcat/webapps`

Environment variables :

You can define environment variable for your containers.

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

```
    - name: APP_COLOR
      value: pink
```

OR

```
docker run --env APP_COLOR=pink simple-webapp-color
```

ReplicationController, ReplicaSet and Deployment

ReplicationController

Used to specify the number of pods (replicas) with a pod template. The controller will automaticly ensure that a correct number of replicas are running (if one pod fails, it will create a new one).

The selector dictionnary is optionnal, it is used to only manage ressources labeled with specific metadatas (here app : nginx). If you don't precise any selector, it will only work with the given template.

Example :

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    app: nginx
  template:
    metadata:
      name: nginx
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        command:
        -  sh
        -  "-c"
        -  echo Hello Kubernetes! && sleep 3600
```

Usefull commands :

```
kubectl get replicationcontroller
```

ReplicaSet

ReplicaSets works the same way as ReplicationController excepts that the selector option is mandatory. It is a more recent way to control pods replicas (replication controller is outdated).

If ressources with the same label are already running, it will be taken into account. The template is used only to create fresh pods (in case the number of replicas isn't reached or if a pod fails).

If you delete a pod of a replicaset, a new one will automaticly be created, and if you try to create a new pod with the same labels than the replicaset selector, it will be terminated instantaneously to keep the right number of pods running.

Example :

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
```

```
containers:
- name: php-redis
  image: gcr.io/google_samples/gb-frontend:v3
```

Usefull commands :

°Create ReplicaSet :

```
kubectl create –f replicaset-definition.yml
```

°Check ReplicaSets :

```
kubectl get replicaset
```

°Describe ReplicaSet :

```
kubectl describe replicaset my-replicaset
```

°Delete ReplicaSet :

```
kubectl delete replicaset myapp-replicaset
```

°If you update your yaml file, to apply changes :

```
kubectl replace -f replicaset-definition.yml
```

°To scale your running ReplicaSet (this won't modify the yaml file) :

```
kubectl scale -–replicas=6 –f replicaset-definition.yml
```

°Same but targeting the labels :

```
kubectl scale -–replicas=6 replicaset myapp-replicaset
```

°Same but editing a file generated by k8s with the configuration of a running replicaset :

```
kubectl edit replicaset myapp-replicaset
```

Deployment

Rolling updates

A deployment is defined the exact same way than a ReplicaSet.

The created deployment will generate a ReplicaSet wich will manage our Pods.

Deployment resource makes it easier for updating your pods to a newer version.

Lets say you use *ReplicaSet-A* for controlling your pods, then You wish to update your pods to a newer version, now you should create *Replicaset-B*, scale down *ReplicaSet-A* and scale up *ReplicaSet-B* by one step repeatedly (This process is known as **rolling update**). Although this does the job, but it's not a good practice and it's better to let K8S do the job.

A **Deployment resource** does this automatically without any human interaction and increases the abstraction by one level.

Note: Deployment doesn't interact with pods directly, it just does rolling update using ReplicaSets.

When you update a deployment file (let's say update the image of one of the container), the rolling update strategy will create a new ReplicaSet, delete one of the running pod of the old replicaset, create a new one with the updated image on the new ReplicaSet and so on until all the pods of the old replicaset are deleted and all the new pods are running in the new replicaset. That one by one strategy allow to always maintain running pods for our application.

Example :

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
```

- **containerPort**: 80

Usefull commands :

°Create deployment :

```
kubectl create deployment httpd-frontend -–image=httpd:latest –-replicas=2
```

Check out the ReplicaSet commands and apply same kind of instructions to manipulate deployments.

```
kubectl create –f deployment-definition.yml

kubectl get deployments

kubectl apply –f deployment-definition.yml
```

°Caution for the following command, it will not update the deployment file if you have one :

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

°Check the current status of your rolling update :

```
kubectl rollout status deployment/myapp-deployment
```

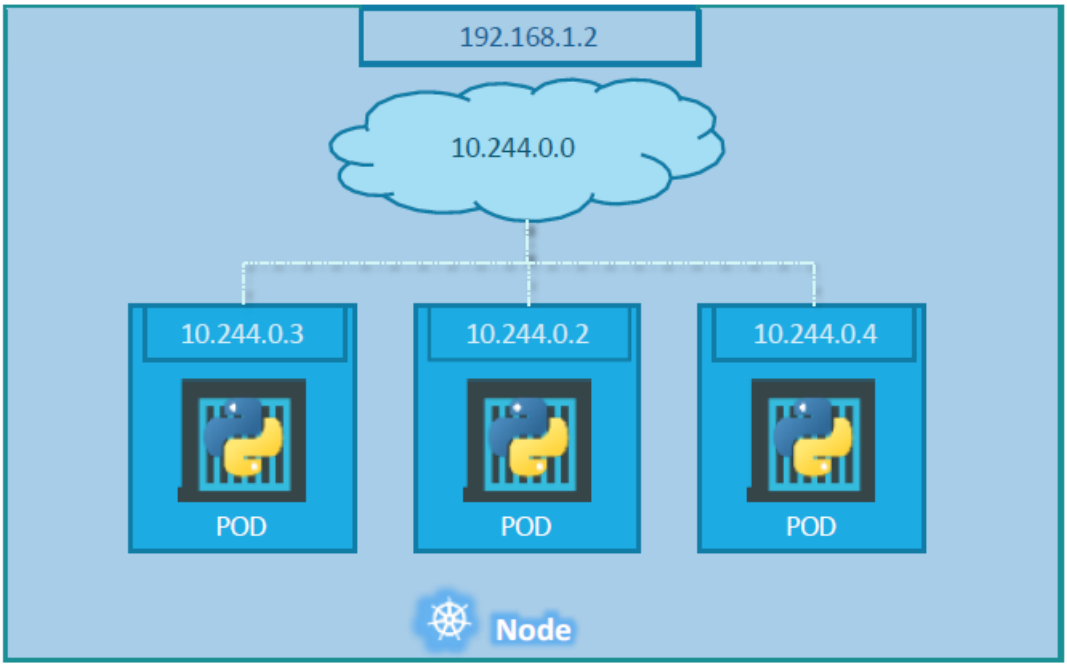
°See the history of your deployments :

```
kubectl rollout history deployment/myapp-deployment
```

°Go back to your previous deployment version :

```
kubectl rollout undo deployment/myapp-deployment
```

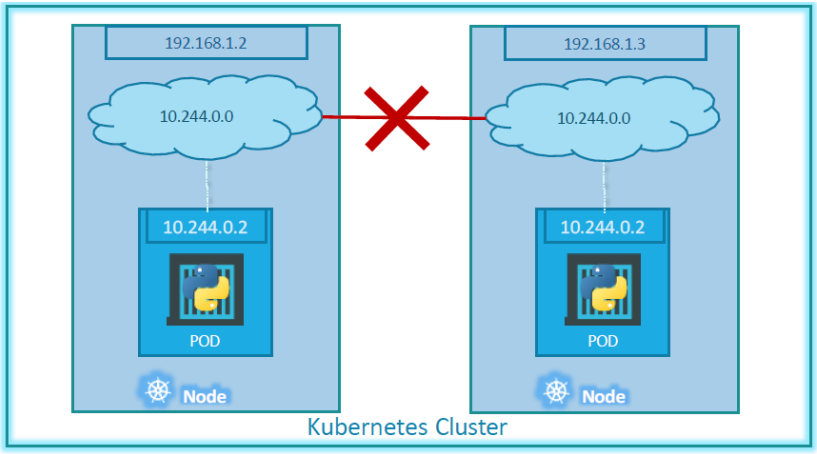
Networking in k8S



Unlike in the docker world were 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 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.

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?

- All containers/PODs can communicate to one another without NAT
- All nodes can communicate with all containers and vice-versa without NAT



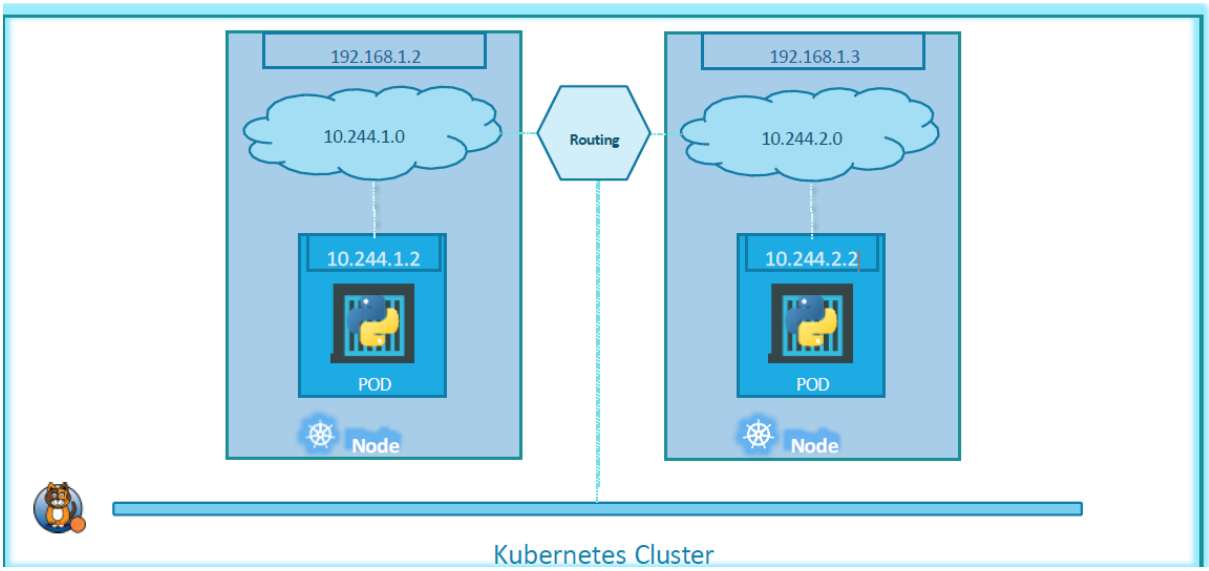
In this case we have two nodes running kubernetes and they have IP addresses 192.168.1.2 and 192.168.1.3 assigned to them. Note that they are not part of the same cluster yet. Each of them has a single POD deployed. As discussed in the previous slide these pods are attached to an internal network and 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. If you look at the play-with-k8s labs they use WeaveNet. In our demos in the course we used Calico.

Depending on your environment and after evaluating the Pros and Cons of each of

these, you may chose the right networking solution.



So back to our cluster, with the Calico networking 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 were 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

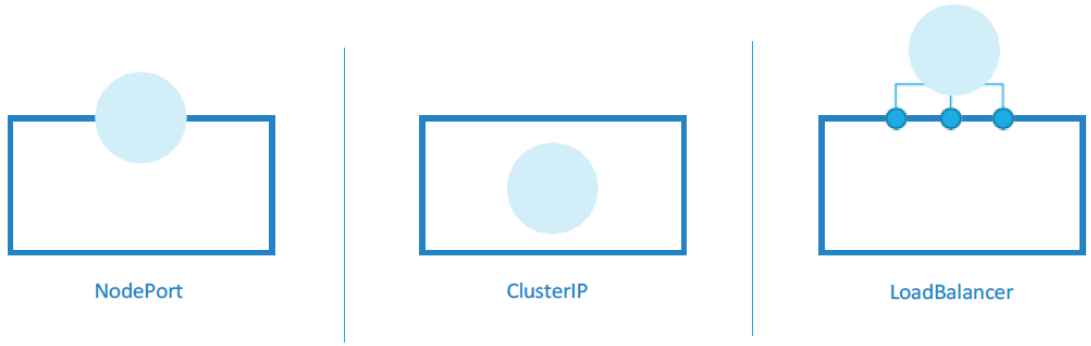
of kubernetes. Thus all PODs can now communicate to each other using the assigned IP addresses.

Services

Let's say 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 PODs. There are other kinds of services available which we will now discuss.

Services Types



°**NodePort** : the service makes an internal POD accessible on a Port on the Node (worker) => creates an external entry point.

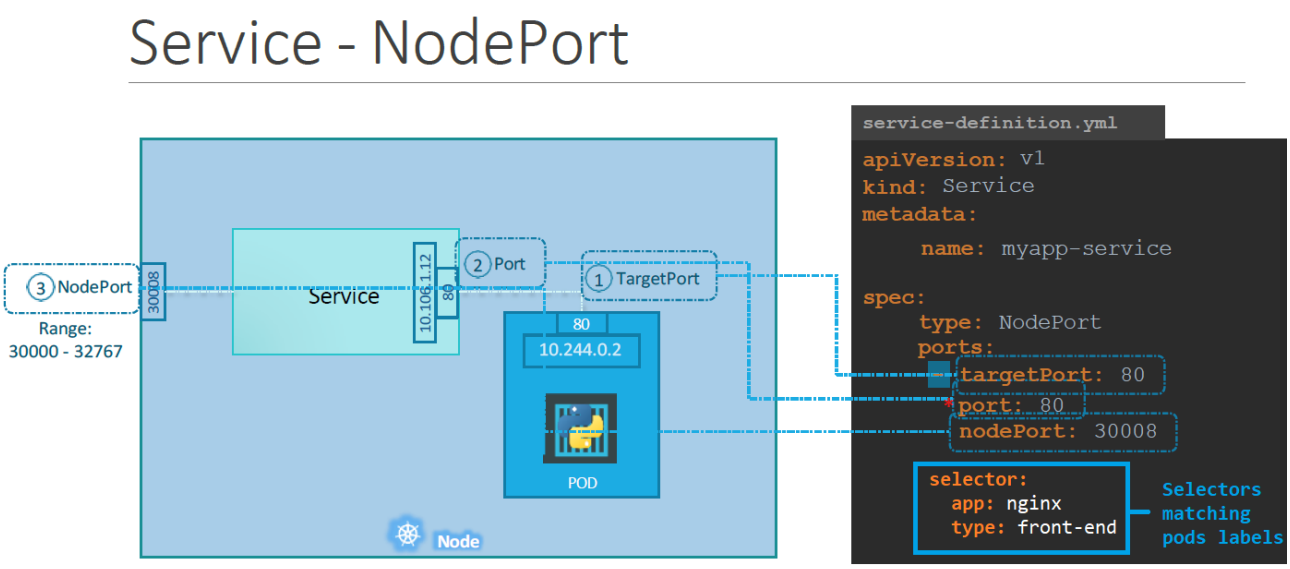
°**ClusterIP** : in this case the service creates a virtual IP inside the cluster to enable communication between different services such as a set of front-end servers to a set of backend- servers.

°**LoadBalancer** : it provisions a load balancer for our service in supported cloud providers. A good example of that would be to distribute load across different web servers.

NodePort

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 were 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.

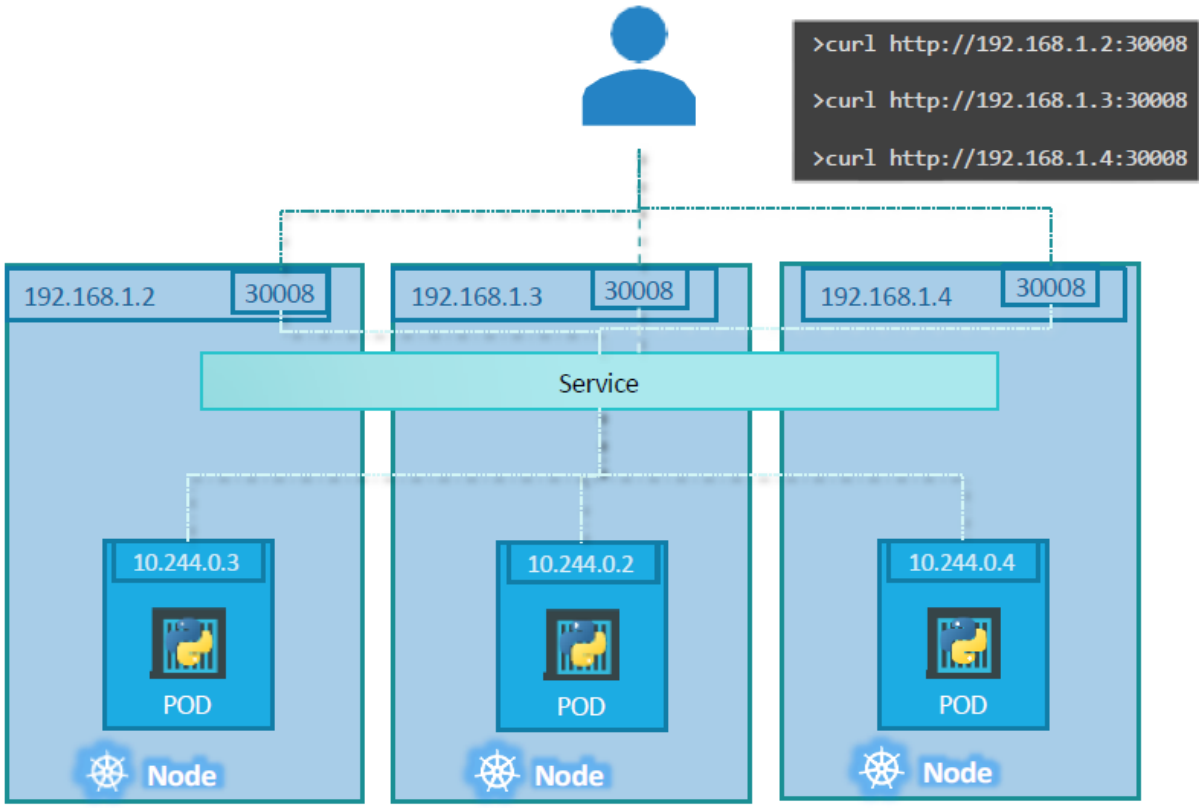
Remember that out of these, the only mandatory field is port . If you don't provide a targetPort it is assumed to be the same as port and if you don't provide a nodePort a free port in the valid range between 30000 and 32767 is automatically allocated. Also note that ports is an array.



NodePort default range: 30000 – 32767.

So note the dash under the ports section that indicate the first element in the array.
You can have multiple such port mappings within a single service.
So we have all the information in, but something is really missing. There is nothing here in the definition file that connects the service to the POD. We have simply specified the targetPort but we didn't mention the targetPort on which POD. There could be 100s of other PODs with web services running on port 80. So how do we do that?
As we did with the replicasets previously and a technique that you will see very often in kubernetes, we will use labels and selectors to link these together. We know that the POD was created with a label. We need to bring that label into this service definition file.

If you have several ports running with the same labels, it twill automatically load balance between those pods using a random algorithm by default.



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.

Example :

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

type: NodePort

selector:

app.kubernetes.io/name: MyApp

ports:

By default and for convenience, the `targetPort` is set to the same value as the `port` field.

- **port**: 80

  targetPort: 80

  # Optional field

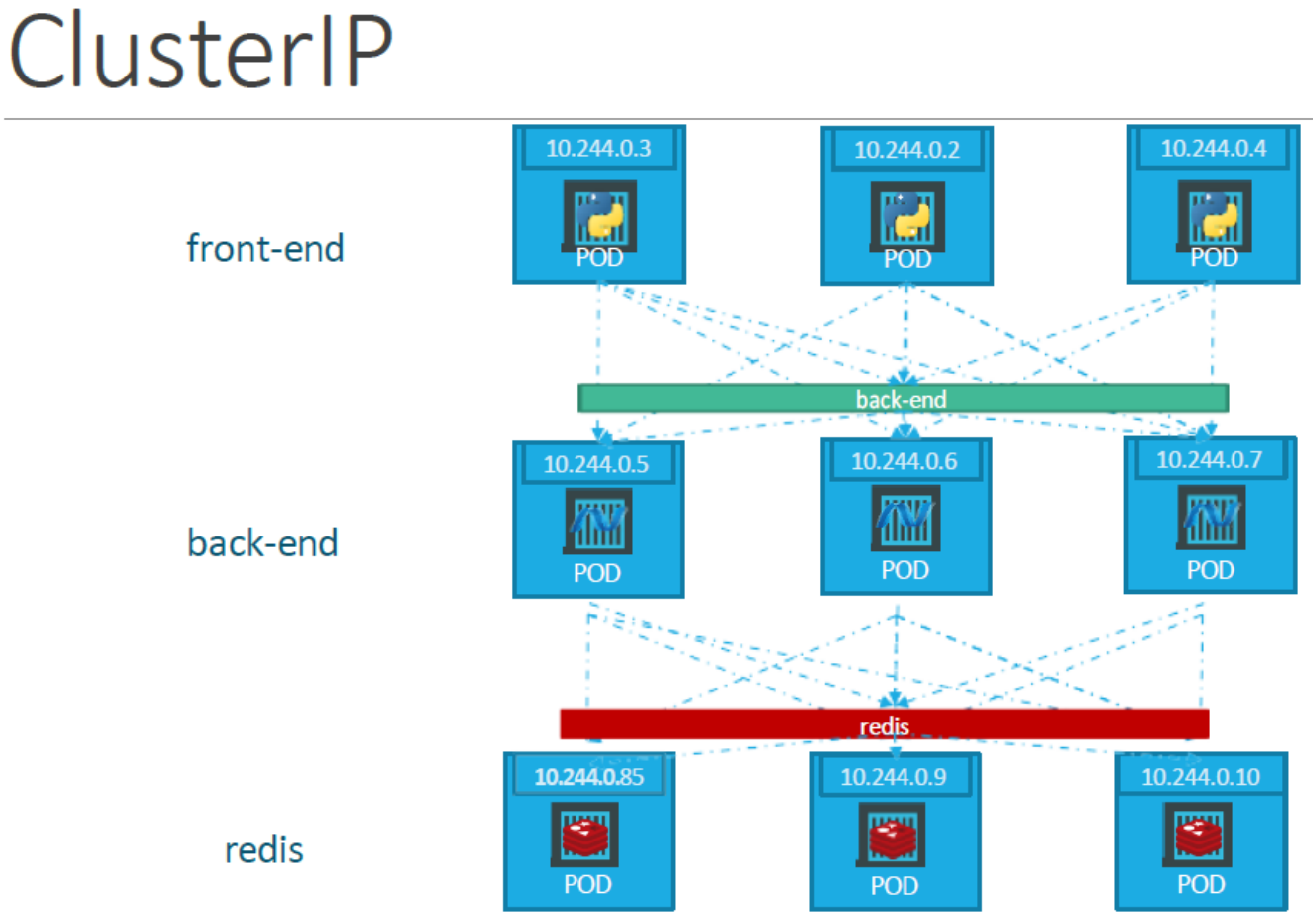
  # By default and for convenience, the Kubernetes control plane will allocate a port from a range (default: 30000-32767)

  nodePort: 30007

Usefull commands for all types of services:

```
kubectl create –f service-definition.yaml

kubectl get services
```

ClusterIp :



Used to manage internal communication between pods. You can link a pool of pods having the same labels to one ClusterIP service to automaticly load balance the internal communication between your pods.

Once the ClusterIP service is defined, the pods can use the ClusterIP adress or the name of the service to interact with a pool of nodes.

Example :

```
apiVersion: v1
kind: Service
metadata:
  name: back-end
spec:
  # ClusterIP is the default Service type if you don't precise it.
  type: ClusterIP
  clusterIP: 10.96.0.10
  selector:
    app: myapp
    type: back-end
  ports:
    # By default and for convenience, the `targetPort` is set to the same value as the `port` field.
    - port: 80
      targetPort: 80
```

LoadBalancer :

The NodePort will allow users to connect to the ips of the different nodes using the correct port, but what if I have several nodes with the same pod deployed ? I can give the users all the ips of my nodes with the open port but of course we will prefere to create a single entry point load balancing between those ips.

You can either set up an external server as a LoadBalancer (nginx or other), but it will be fastidious to maintain it (add manually the ips of the NodePort Services etc…) or you can use a cloud platform LoadBalancer (AWS, Azure, Google Cloud…).

If you use a LoadBalancer service on a non adapted environment (like directly on your VirtualBox VM), it will work exactly as a NodePort. On AWS, GC, Azure etc… managed k8s, using the LoadBalancer will create an external IP (you can get it with kubectl get services).

Example :

```
apiVersion: v1
```

kind: Service

metadata:

name: my-service

spec:

type: LoadBalancer

selector:

app.kubernetes.io/name: MyApp

ports:

- **protocol**: TCP

port: 80

targetPort: 9376

clusterIP: 10.0.171.239

status:

loadBalancer:

ingress:

- **ip**: 192.0.2.127

Multiple port configuration :

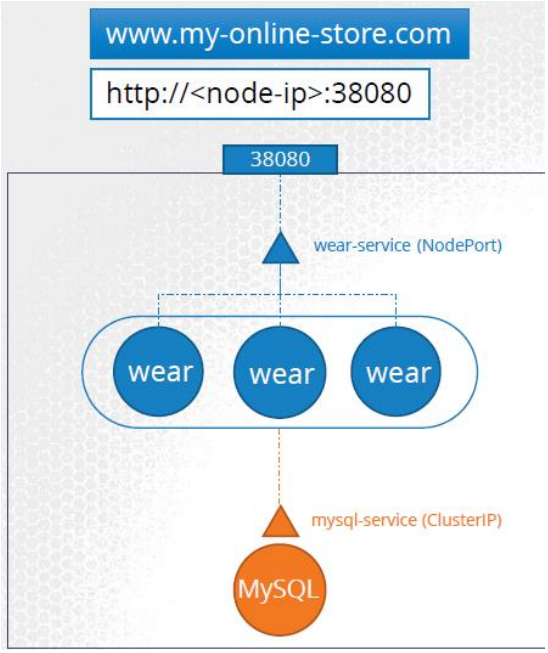
For all the service types :

```yaml
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 9376
    - name: https
      protocol: TCP
      port: 443
      targetPort: 9377
```

Ingress Controller

Let us first briefly revisit services and work our way towards ingress. We will start with a simple scenario. 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.

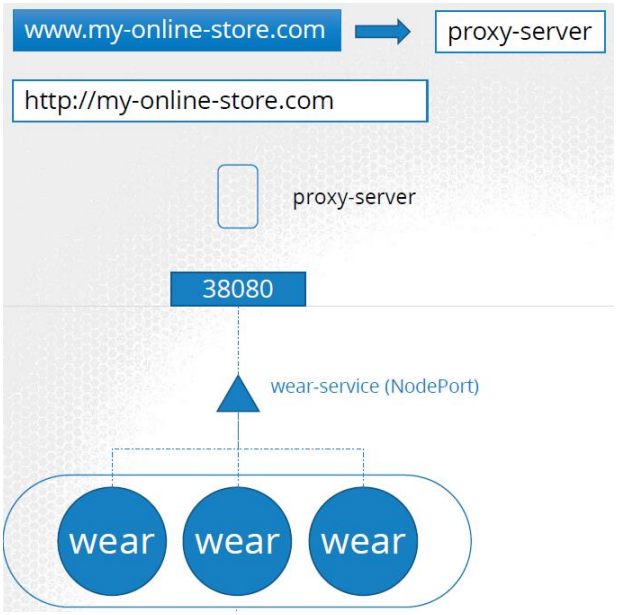
You build the application into a Docker Image and deploy it on the kubernetes cluster as a POD in a Deployment. Your application needs a database so you deploy a MySQL database as a POD and create a service of type ClusterIPcalled mysql-service to make it accessible to your application. Your application is now working. To make the application accessible to the outside world, you create another service, this time of type NodePortand 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.
Whenever traffic increases, we increase the number of replicas of the POD to handle the additional traffic, and the service takes care of splitting traffic between the PODs.



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.

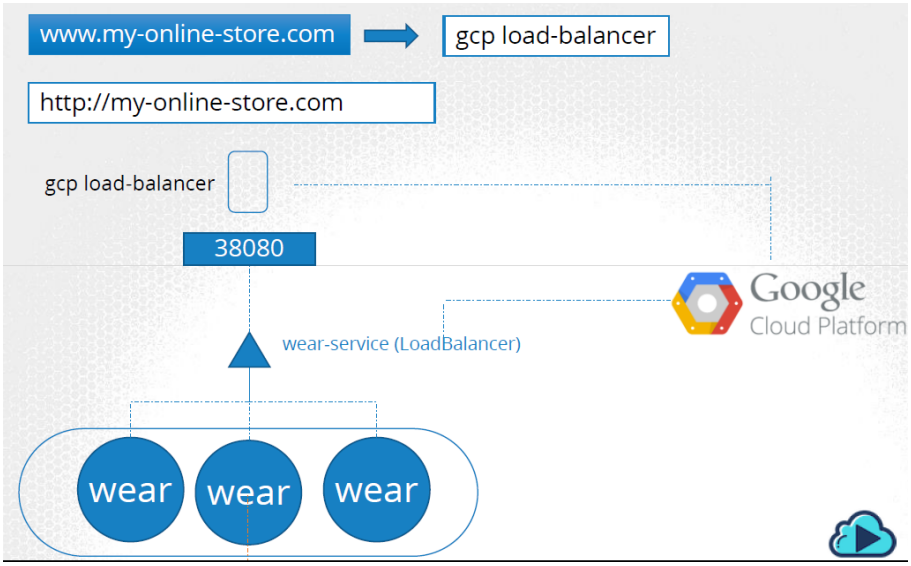
For example, we do not want the users to have to type in IP address everytime. So you configure your DNS server to point to the IP of the nodes. Your users can now access your application using the URL http://my-online-store.com:38080.

Now, you don't want your users to have to remember port number either. However, Service NodePortscan 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.

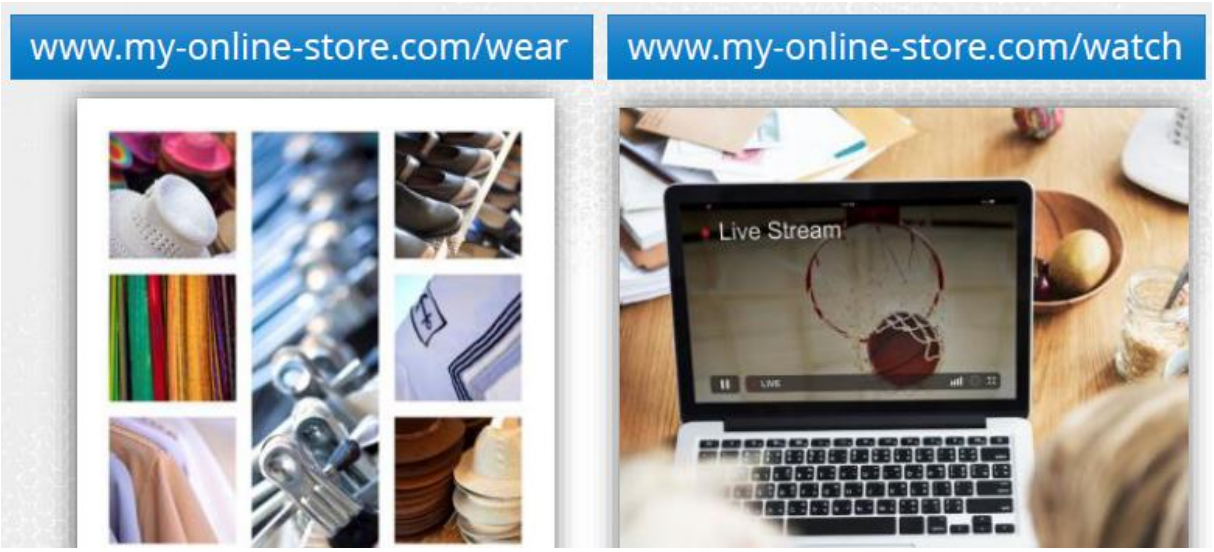


Let's take a step back and see what you could do it you were on a public cloud environment like Google Cloud Platform.
In that case, instead of creating a service of type NodePortfor your wear application, you could set it to type LoadBalancer. When you do that Kubernetes would still do everything that 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 LoadBalancerconfigured to route traffic to the service ports on all the nodes and return its information to kubernetes. The LoadBalancerhas 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.

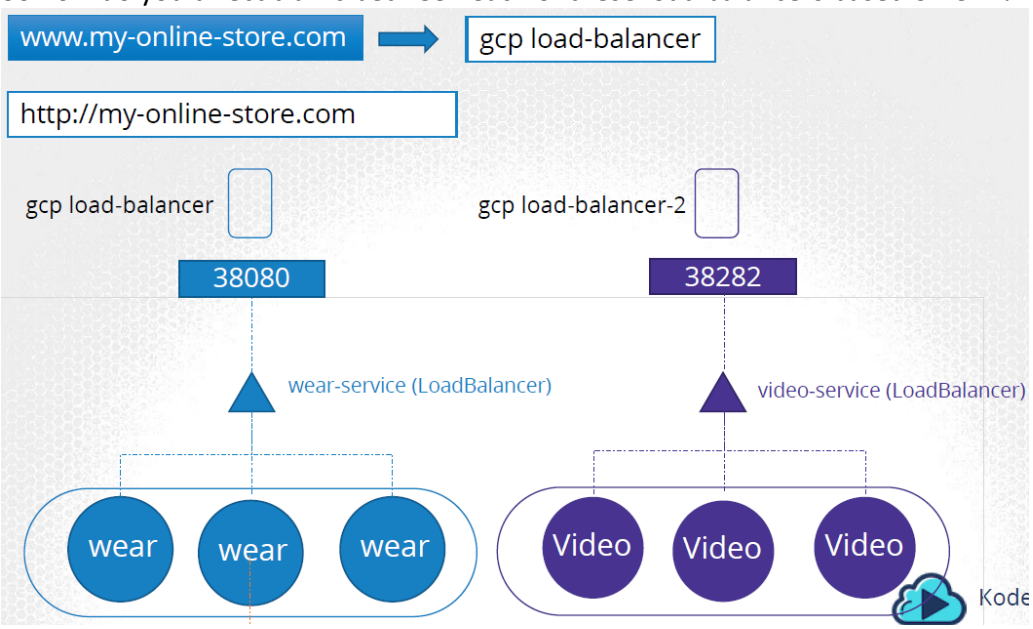
On receiving the request, GCP would then automatically deploy a LoadBalancerconfigured to route traffic to the service ports on all the nodes and return its information to kubernetes. The LoadBalancerhas an external IP that can be provided to users to access the application. In this case we set the DNS to point to this IP and users access the application using the URL my-online-store.com. Perfect!!



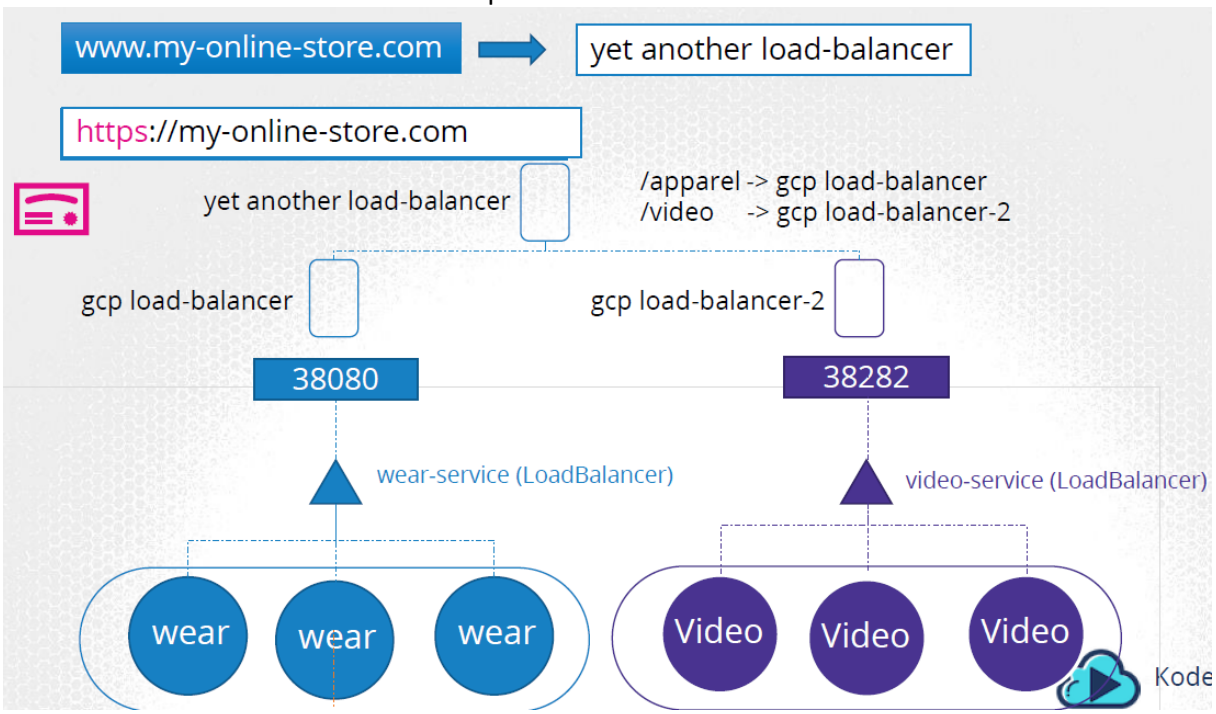
Your companies business grows and you now have new services for your customers. For example, a video streaming service. You want your users to be able to access your new video streaming service by going to my-online-store.com/watch. You'd like to make your old application accessible at my-online.store.com / wear.



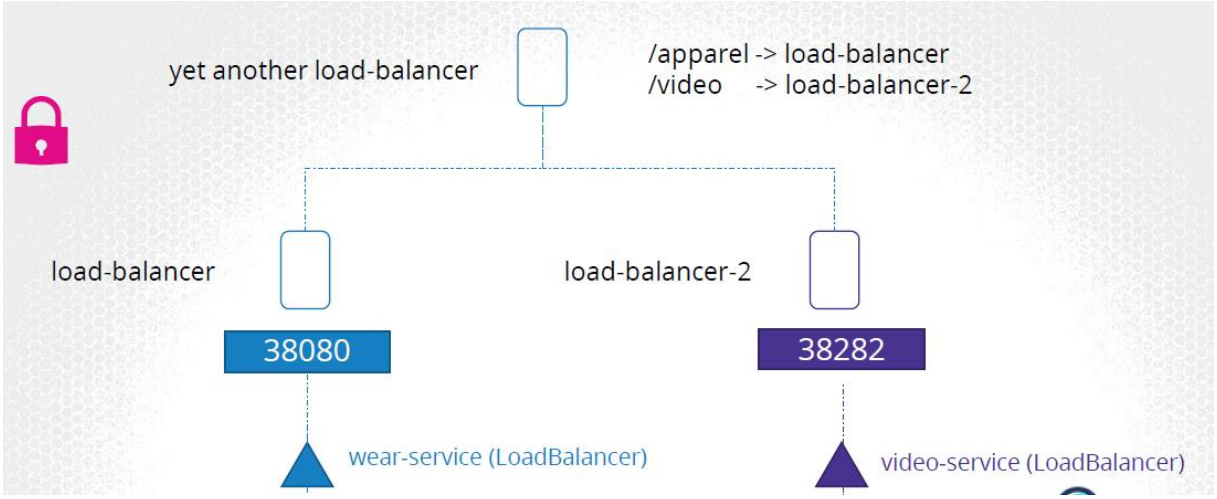
Your developers developed the new video streaming application as a completely different application, as it has nothing to do with the existing one. However to share the cluster resources, you deploy the new application as a separate deployment within the same cluster. You create a service called video-service 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?



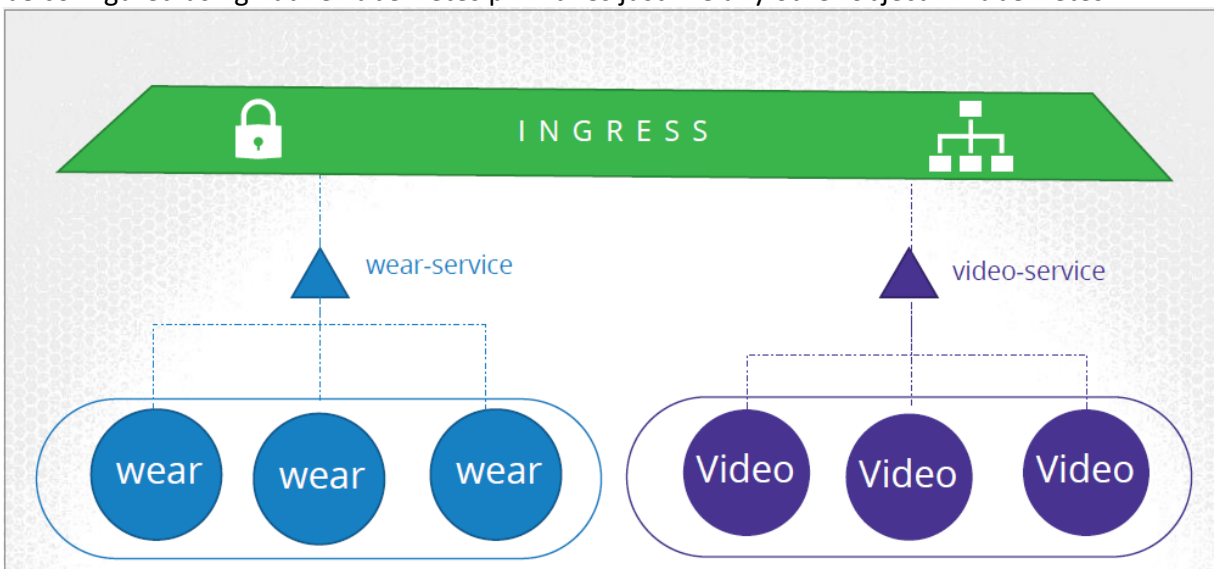
You need yet another proxy or load balancer that can re-direct traffic based on URLs to the different services. Every time you introduce a new service you have to re-configure the load balancer.
And finally you also need to enable SSL for your applications, so your users can access your application using 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.

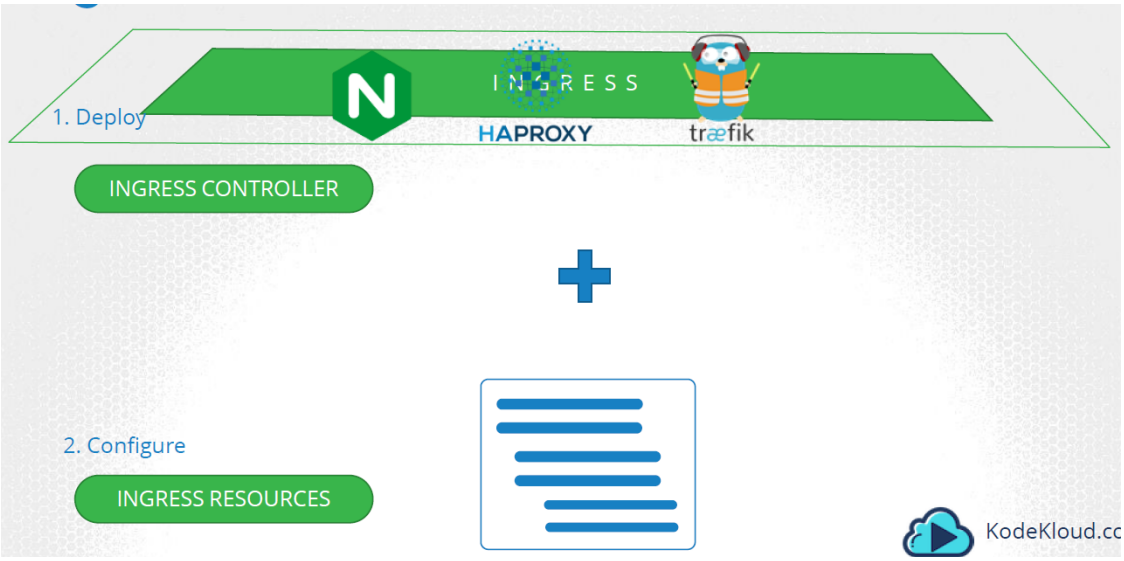


Simply put, think of ingress as a layer 7 load balancer built-in to the kubernetes cluster that can be configured using native kubernetes primitives just like any other object in kubernetes.



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 NodePortor 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.

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 HAProxy or 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 listed here, 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. If you setup a cluster following the demos in this course, you won't have an ingress controller. 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, sslsettings, session timeout etc. In order to decouple these configuration data from the nginx-controller image, you must create a ConfigMapobject and pass that in. Now remember the ConfigMapobject 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.

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
  matchLabels:
    name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - name: nginx-ingress-controller
          image: quay.io/kubernetes-ingresscontroller/nginx-ingress-controller:0.21.0
          args:
            - /nginx-ingress-controller
            - --configmap=$(POD_NAMESPACE)/nginx-configuration
          env:
            - name: POD_NAME
              valueFrom:
                fieldRef:
                  fieldPath: metadata.name
            - name: POD_NAMESPACE
              valueFrom:
                fieldRef:
                  fieldPath: metadata.namespace
```

```
        ports:
          - name: http
            containerPort: 80
          - name: https
            containerPort: 443
```

```
kind: ConfigMap
apiVersion: v1
metadata:
  name: nginx-configuration
```

We then need a service to expose the ingress controller to the external world. So we create a service of type NodePortwith the nginx-ingress label selector to link the service to the deployment. So with these three objects we should be ready with an ingress controller in its simplest form.

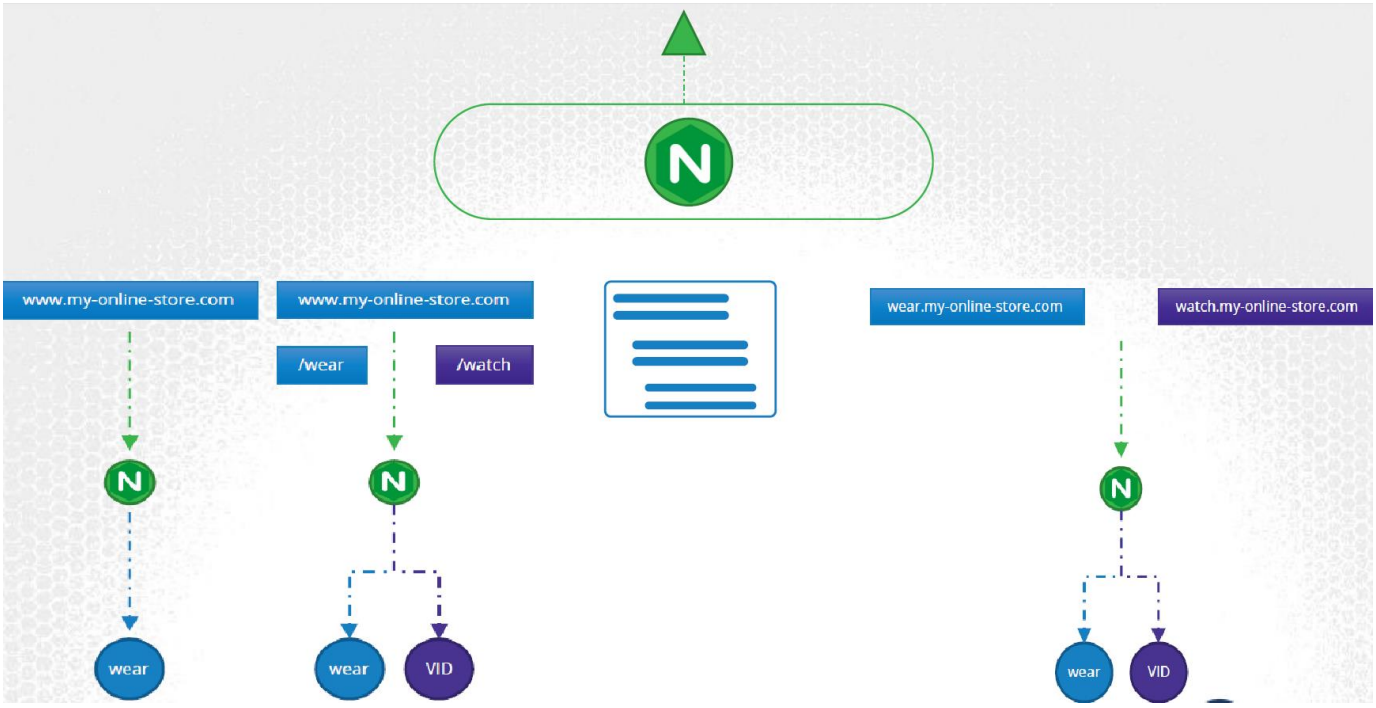
```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
  - port: 80
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    targetPort: 443
    protocol: TCP
    name: https
  selector:
    name: nginx-ingress
```

We will also need a service account for the ingress-controller Deployment if we are in another namespace than the default namespace:

kubectl create serviceaccount ingress-serviceaccount -n my-namespace

So with these three objects we should be ready with an ingress controller in its simplest form.

Now on to the next part, of creating Ingress Resources. An Ingress Resource is a set of rules and configurations applied on the ingress controller. You can configure rules to say, simply forward all incoming traffic to a single application, or route traffic to different applications based on the URL. So if user goes to my-online-store.com/wear, then route to one app, or if the user visits the /watch URL then route to the video app. Or you could route user based on the domain name itself.



Let us look at how to configure these in a bit more detail. The Ingress Resource is created with a Kubernetes Definition file. In this case, ingress-wear.yaml. As with any other object, we have apiVersion, kind, metadata and spec. The apiVersion is extensions/v1beta1, kind is Ingress, we will name it ingress-wear. And under spec we have backend.

As you might have guessed already, traffic is routed to the application services and not PODs directly. The Backend section defines where the traffic will be routed to. So if it's a single backend, then you don't really have any rules. You can simply specify the service name and port of the backend wear service. Create the ingress resource by running the kubectl create command. View the created ingress by running the kubectl get ingress command. The new ingress is now created and routes all incoming traffic directly to the wear-service.

```
apiVersion: extensions/v1beta1

kind: Ingress

metadata:
```

```
  name: ingress-wear
spec:
  backend:
    serviceName: wear-service
    servicePort: 80
```

```
$ kubectl create –f Ingress-wear.yaml
ingress.extensions/ingress-wear created
$ kubectl get ingress
NAME           HOSTS              ADDRESS   PORTS
ingress-wear      *                  80       2s
```

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 4thrule 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.

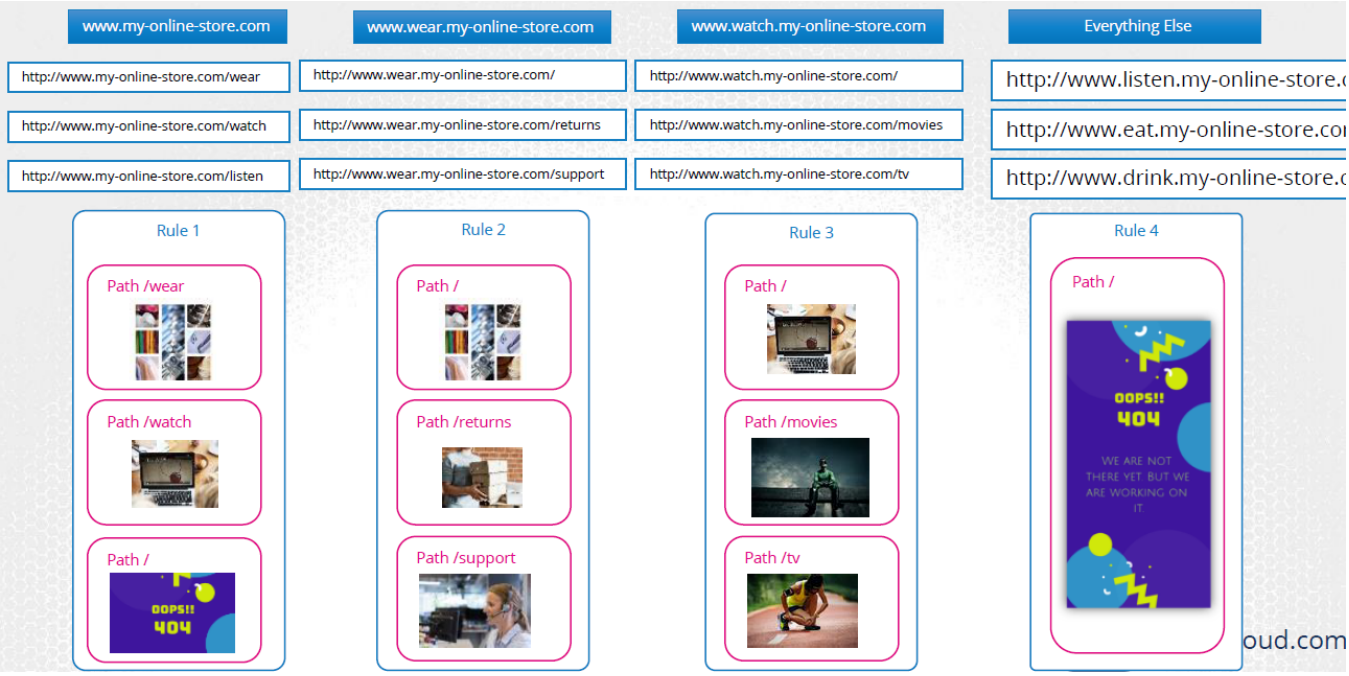
And just in case you didn’t know, you would typically achieve this, by adding multiple DNS entries, all pointing to the same Ingress controller service on your kubernetes cluster.

Now within each rule you can handle different paths. For example, within Rule 1 you can handle the wear path to route that traffic to the clothes application. And a watch path to route traffic to the video streaming application. And a third path that routes anything other than the first two to a 404 not found page.

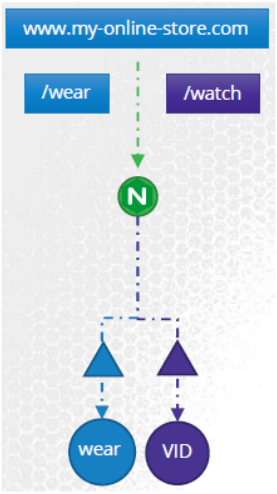
Similarly, the second rule handles all traffic from wear.my-online-store.com. You can have path definition within this rule, to route traffic based on different paths. For example, say you have different applications and services within the apparel section for shopping, or returns, or support, when a user goes to wear.my-online.store.com/, by default they reach the shopping page. But if they go to exchange or support, they reach different backend services.

The same goes for Rule 3, where you route traffic to watch.my-online-store.com to the video streaming application. But you can have additional paths in it such as movies or tv.

And finally anything other than the ones listed will got to the 4thRule, that would simply show a 404 Not Found Error page. So remember, you have rules at the top for each host or domain name. And within each rule you have different paths to route traffic based on the URL.



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 service name and service port. 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 command.



Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

Once created, view additional details about the ingress by running the kubectl 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.

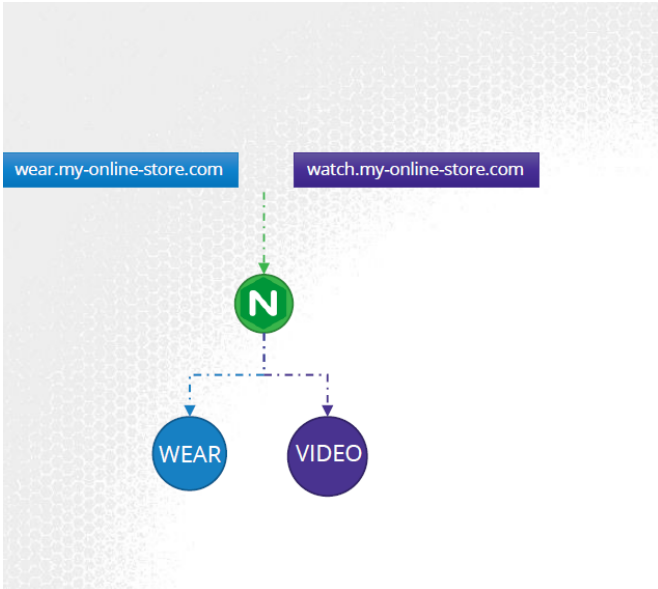
Now, If you look closely in the output of this command, you see that there is something about a Default-backend. Hmmm. What might that be?

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.

```
kubectl describe ingress ingress-wear-watch
Name:             ingress-wear-watch
Namespace:        default
Address:
Default backend:  default-http-backend:80 (<none>)
Rules:
  Host  Path  Backends
  ----  ----  --------
  *
        /wear    wear-service:80 (<none>)
        /watch   watch-service:80 (<none>)
Annotations:
Events:
  Type    Reason  Age   From                      Message
  ----    ------  ----  ----                      -------
  Normal  CREATE  14s   nginx-ingress-controller  Ingress default/ingress-wear-watch
```

Back in your application, say a user visits the URL my-online-store.com/listen or /eat and you don't have an audio streaming or a food delivery service, you might want to show them a nice message. You can do this by configuring a default backend service to display this 404 Not Found error page.

The third type of configuration is using domain names or hostnames. We start by creating a similar definition file for Ingress. 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
```

Let's compare the two. Splitting traffic by URL had just one rule and we split the traffic with two paths. To split traffic by hostname, we used two rules and one path specification in each rule.

Ingress-wear-watch.yaml

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - http:
      paths:
      - path: /wear
        backend:
          serviceName: wear-service
          servicePort: 80
      - path: /watch
        backend:
          serviceName: watch-service
          servicePort: 80
```

Ingress-wear-watch.yaml

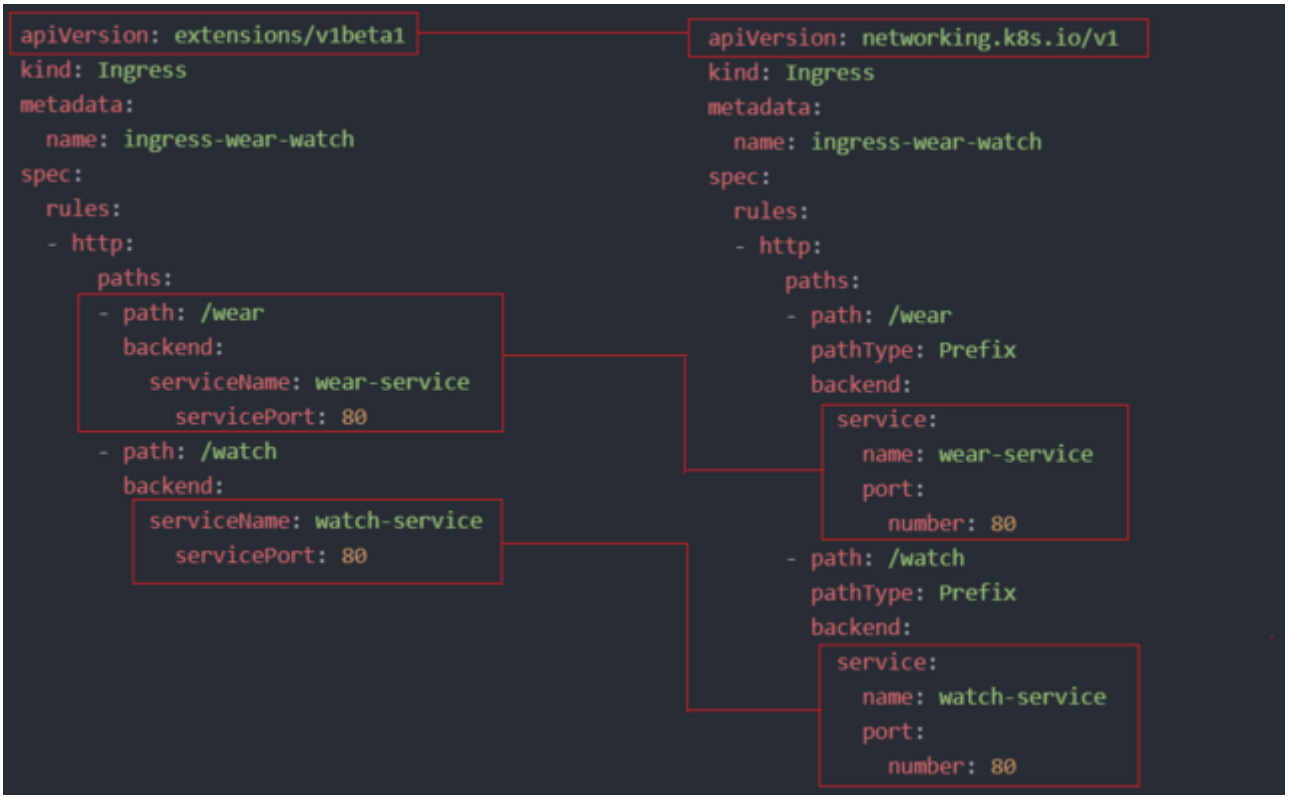
```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear-watch
spec:
  rules:
  - host: wear.my-online-store.com
    http:
      paths:
      - backend:
          serviceName: wear-service
          servicePort: 80
  - host: watch.my-online-store.com
    http:
      paths: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

UPDATES

As we already discussed **Ingress** in our previous lecture. Here is an update.

In this article, we will see what changes have been made in previous and current versions in **Ingress**.

Like in **apiVersion**, **serviceName** and **servicePort** etc.



Now, in k8s version **1.20+** we can create an Ingress resource from the imperative way like this:-

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"

Find more information and examples in the below reference link:-

https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-em-ingress-em-

References:-

https://kubernetes.io/docs/concepts/services-networking/ingress

https://kubernetes.io/docs/concepts/services-networking/ingress/#path-types

NOTE :

If you use a path /something, the ingress rule will forward the path to the pod hosting the app and will try to get the resource /something inside the Pod.

But what if that resource does not exists and you just want to redirect to the root of the pod?

Well you can use annotations for that :

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
  namespace: app-space
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

Another example :

In this ingress definition, any characters captured by `(.*)` will be assigned to the placeholder `$2`, which is then used as a parameter in the `rewrite-target` annotation.

For example, the ingress definition above will result in the following rewrites:

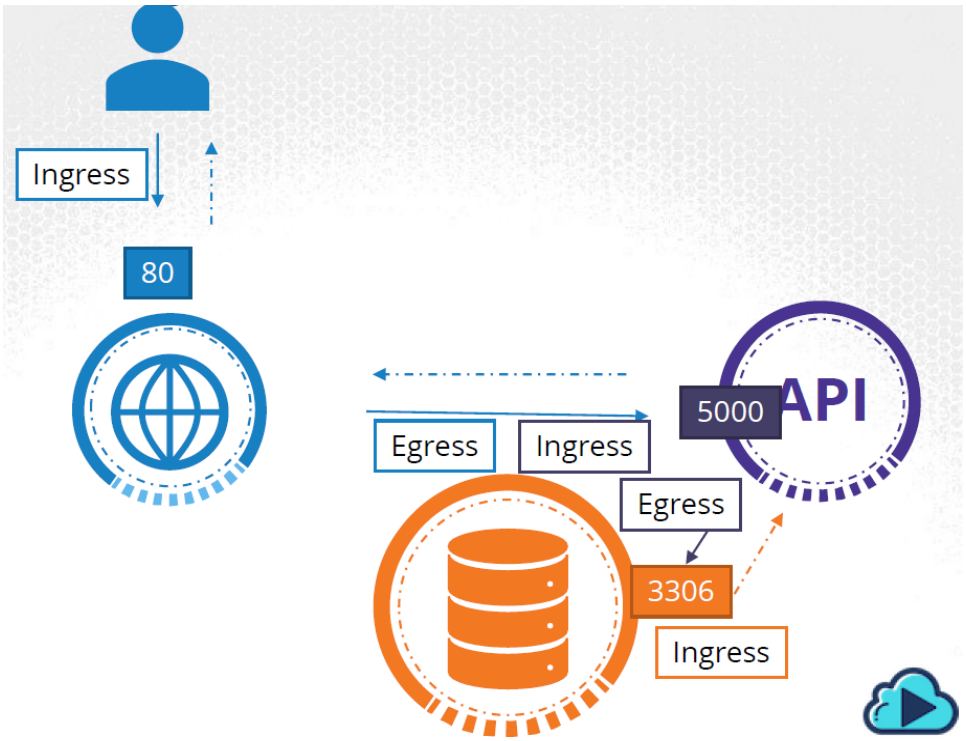
- `rewrite.bar.com/something` rewrites to `rewrite.bar.com/`
- `rewrite.bar.com/something/` rewrites to `rewrite.bar.com/`
- `rewrite.bar.com/something/new` rewrites to `rewrite.bar.com/new`

```
replace("/something(/|$)(.*)", "/$2")
```

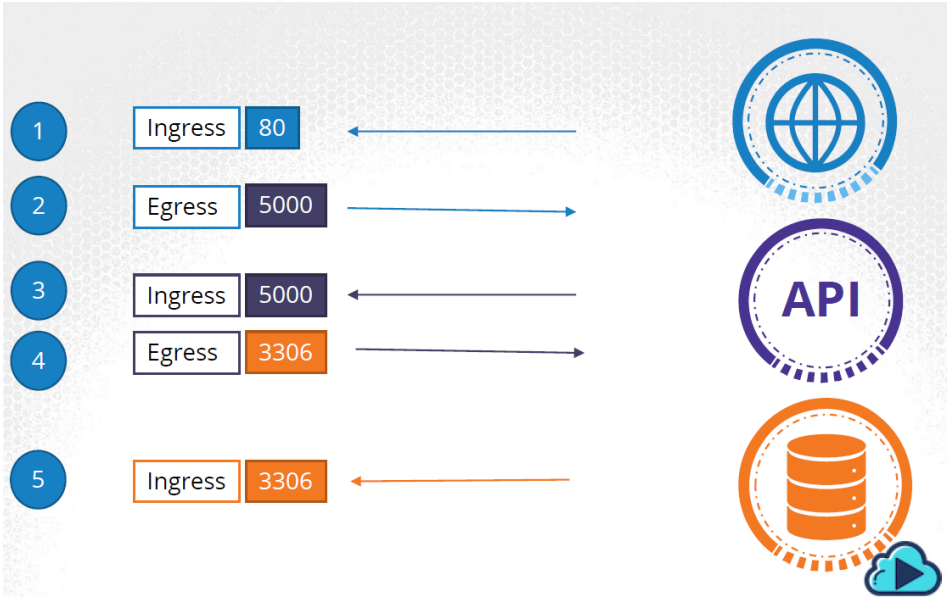
```
1.  apiVersion: extensions/v1beta1
2.  kind: Ingress
3.  metadata:
4.    annotations:
5.      nginx.ingress.kubernetes.io/rewrite-target: /$2
6.    name: rewrite
7.    namespace: default
8.  spec:
9.    rules:
10.   - host: rewrite.bar.com
11.     http:
12.       paths:
13.       - backend:
14.           serviceName: http-svc
15.           servicePort: 80
16.         path: /something(/|$)(.*)
```

Network Policies Ingress and Egress rules

We will start with a simple example of a traffic through a web, app and database server. So you have a web server serving front-end to users, an app server serving backend API's and a database server. 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. And that is denoted by the straight arrow. 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 the API server.



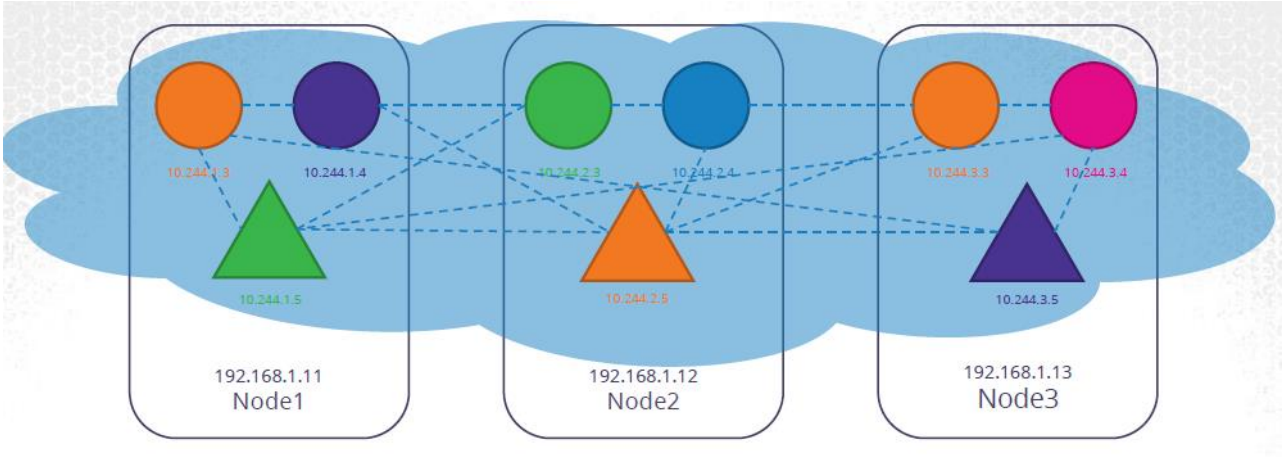
If we were to list the rules required to get this working, we would have an Ingress rule that is required to accept HTTP traffic on port 80 on the web server. An Egress rule to allow traffic from the web server to port 5000 on the API server. An ingress rule to accept traffic on port 5000 on the API server and an egress rule to allow traffic to port 3306 on the database server. And finally an ingress rule on the database server to accept traffic on port 3306. So that's traffic flow and rules basics.



Let us now look at Network Security in Kubernetes. So we have a cluster with a set of nodes hosting a set of pods and services. Each node has an IP address and so does each pod as well as service. One of the pre-requisite for networking in kubernetes, is whatever solution you implement, the pods should be able to communicate with each other without having to configure any additional settings, like routes.

For example, in this network solution, all pods are on a virtual private network that spans across the nodes in the kubernetes cluster. And they can all by default reach each other using the IPs or

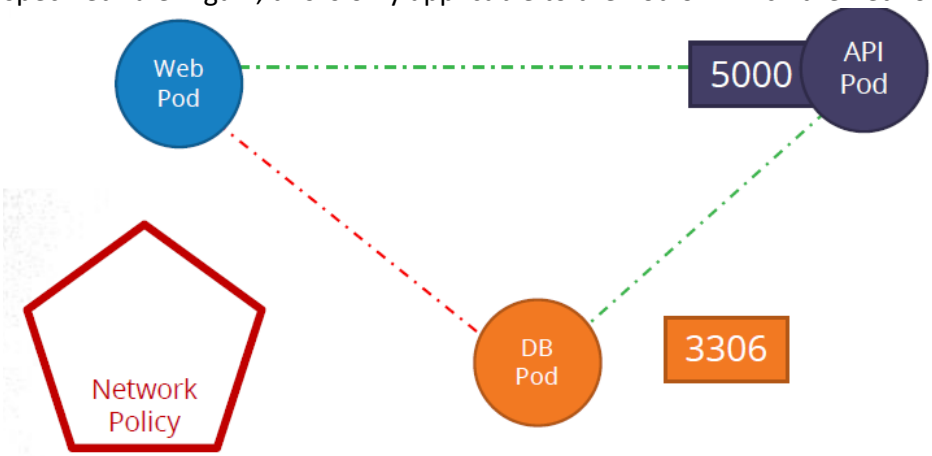
pod names or services configured for that purpose. Kubernetes is configured by default with an "All Allow" rule that allows traffic from any pod to any other pod or services.

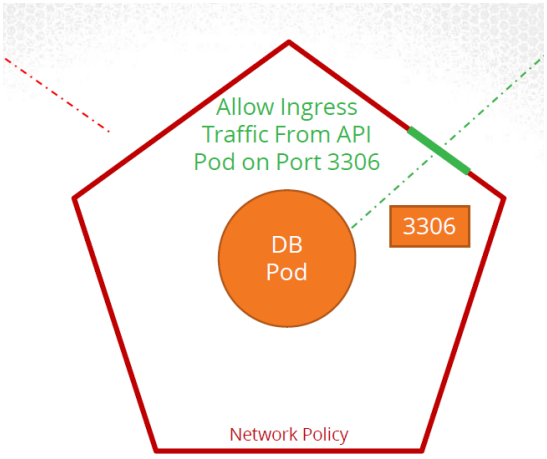


Let us now bring back our earlier discussion and see how it fits in to kubernetes. For each component in the application we deploy a POD. One for the front-end web server, for the API server and one for the database. We create services to enable communication between the PODs as well as to the end user. Based on what we discussed in the previous slide, by default all the three PODs can communicate with each other within the kubernetes cluster.

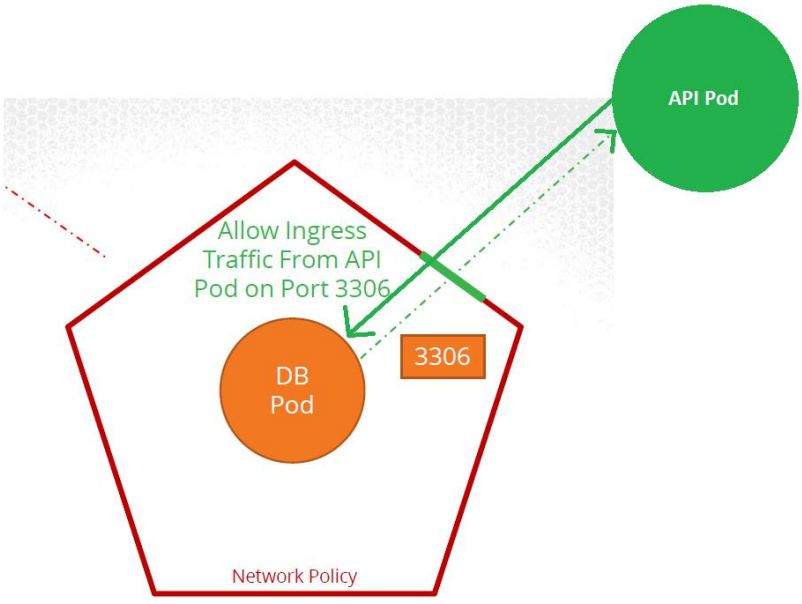
What if we do not want the front-end web server to be able to communicate with the database server directly? Say for example, the security teams and audits require you to prevent that from happening? That is where you would implement a Network Policy to allow traffic to the dbserver only from the apiserver. Let's see how we do that.

A Network policy is another object in the kubernetes namespace. Just like PODs, ReplicaSetsor Services. You link a network policy to one or more pods. You can define rules within the network policy. In this case I would say, only allow Ingress Traffic from the API Pod on Port 3306. Once this 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.





So how do you apply or link a network policy to a Pod? We use the same technique that was used before to link ReplicaSetsor Services to Pods. Labels and Selectors. We label the Pod and use the same labels on the pod selector field in the network policy.
And then we build our rule. Under policyTypesspecify weather the rule is to allow ingress or egress traffic or both. In our case we only want to allow ingress traffic to the db-pod. So we add Ingress. Next, we specify the ingress rule, that allows traffic from the API pod. And you specify the api pod, again using labels and selectors. And finally the port to allow traffic on, which is 3306.
°First we need to protect the desired Pod, here the db Pod, by creating a NetworkPolicy.
We can associate the concerned pod to our policy using labels (**podSelector**) in our case it is the label role: db.
°We next have to specify the kind of policies **Ingress and/or Egress**.
Once you create the network policy with the policyType, it will block all Ingress and/or Egress on the Pod except for the rules that you will define.
Once you create a rule to allow incoming traffic on a Pod, you don't have to create a rule for the returning traffic (returning the results from the db to the API), it will be automatically allowed (dotted line). **However,** that does not mean that the db Pod can query the API on a specific port, we will need a specific Egress rule for that.



°By default the NetworkingPolicy will only allow Pod from the same namespace to communicate (you can precise the namespace of the NetworkingPolicy in the metadata section), but you can expand the communication to Pods from different namespaces on the ingress or egress part of the definition file. For that, under the - podSelector section of the ingress part, you can use the section namespaceSelector.
Important note : the - namespaceSelector can also be used at the same level than the - podSelector, if you do that all the Pods on that namespace will be allowed to use the custom rule.
°You can also allow external servers (from outside your kub cluster) to communicate with your Pods with the NetworkingPolicies. For that you won't use selectors but the ip address of the external server. For this you can use the - ipBlock section and define ip **ranges** allowed to use the policy.
°Egress rules works the same way than ingress rules.

```
apiVersion: networking.k8s.io/v1
kind: NetworkingPolicy
metadata:
  name: db-policy
  namespace: prod
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - podSelector:
      matchLabels:
        name: api-pod
      namespaceSelector:
        matchLabels:
          name: staging
    - namespaceSelector:
```

```
    matchLabels:
      name: pre-prod
  - ipBlock:
    cidr: 192.168.5.10/32
  ports:
  - protocol: TCP
    port: 3306
 egress:
 -to:
  - ipBlock:
    cidr: 192.168.5.10/32
  ports:
  - protocol: TCP
    port: 80
```

You can get networking policies with the command (works with describe too):

```
kubectl get networkpolicies
```

Here is a version of the official kub documentation :

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.17.0.0/16
            except:
              - 172.17.1.0/24
        - namespaceSelector:
            matchLabels:
              project: myproject
        - podSelector:
            matchLabels:
              role: frontend
```

```
    ports:
      - protocol: TCP
        port: 6379
  egress:
    - to:
      - ipBlock:
          cidr: 10.0.0.0/24
      ports:
      - protocol: TCP
        port: 5978
```

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. Always refer to the network solution's documentation to see support for network policies. Also remember, even in a cluster configured with a solution that does not support network policies, you can still create the policies, but they will just not be enforced. You will not get an error message saying the networking solution does not support network policies.

Solutions that Support Network Policies:
- Kube-router
- Calico
- Romana
- Weave-net

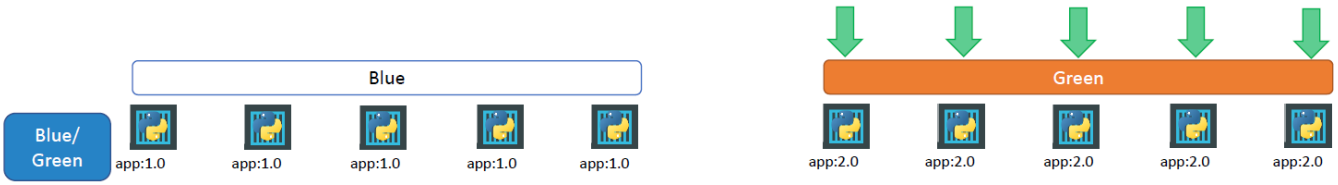
Solutions that DO NOT Support Network Policies:
- Flannel

Deployment strategies

Blue/Green deployment

With the blue/green strategy, the new version is deployed alongside the older one.

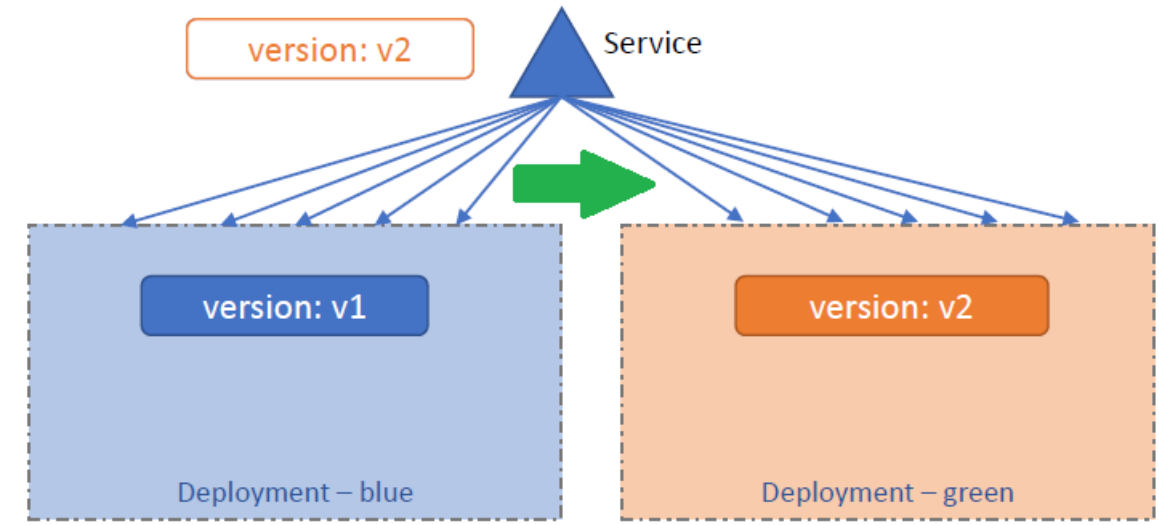
Then a number of tests are runned on the new version pods and if they are successful, the traffic is switched to the new version all at once.



So how do we implement a blue/green deployment natively in kubernetes?

So let's say we have a V1 deployment (blue deployment) linked (through selectors and labels) to a Service exposing the app.

We then deploy our new version V2 (green) on our kub and once all the tests are successful, we switch the selector of our service in order to make it point to the newer version (we can define an application version tag and use it as a selector).

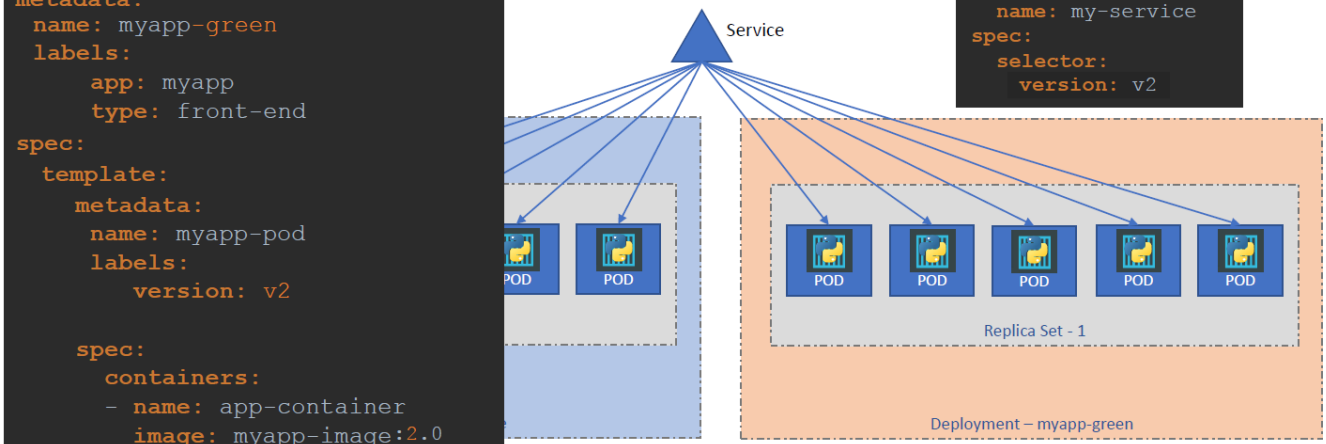


myapp-green.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-green
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        version: v2

    spec:
      containers:
      - name: app-container
        image: myapp-image:2.0

  replicas: 5
  selector:
    matchLabels:
      type: front-end
```

service-definition.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    version: v2
```



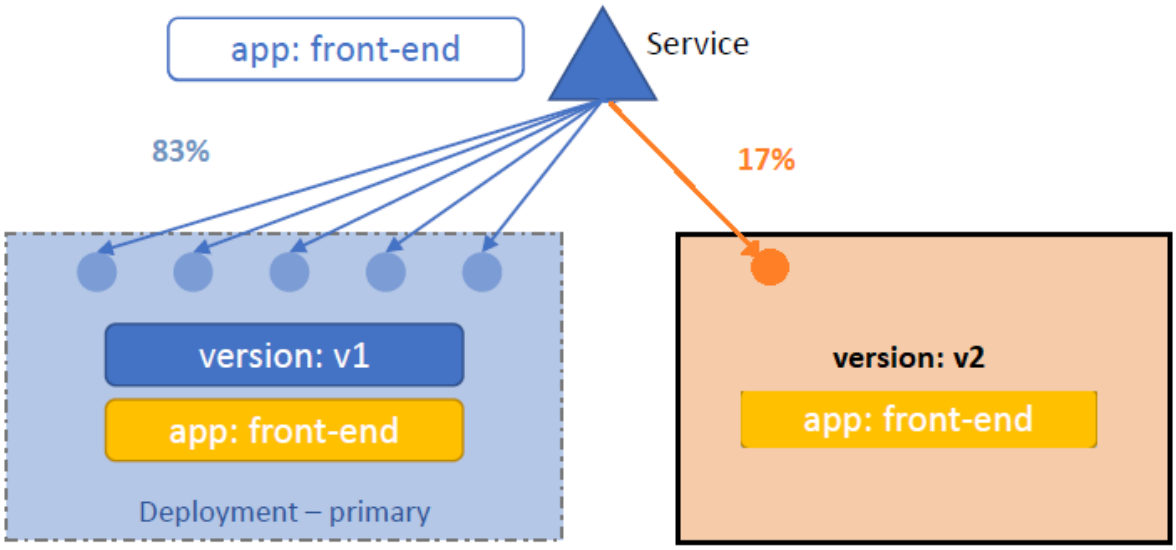
Canary updates

With the canary deployment, we want to test our new deployment by routing first only a small part of the traffic to the new version. Once the testing is done, the traffic is fully routed to the new version.

So how do we weight the traffic to our different deployments?

If we deploy a new version with the same number of pod replicas than the old version and we use a selector on the service which is shared by the two deployments (old and new), the traffic will be equally distributed to both versions (50/50).

If we want less traffic to be routed to the new version, we need to decrease the number of Pods deployed for the new version:



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
```

Once the new version is tested, we can increase the number of Pods of the new deployment and delete the old version.

This is one of native Kubernetes weakness, we cannot define a certain percentage of traffic to be routed to the desired selector, it will always be equally distributed so we have to act on the pods numbers (but if we want only 1% of the traffic to be routed to a deployment, we will need at least 100 Pods to be deployed).

This can be overcome with components like Istio (service mesh), which can manage the traffic in a more detailed fashion:

https://istio.io/v1.2/docs/setup/kubernetes/

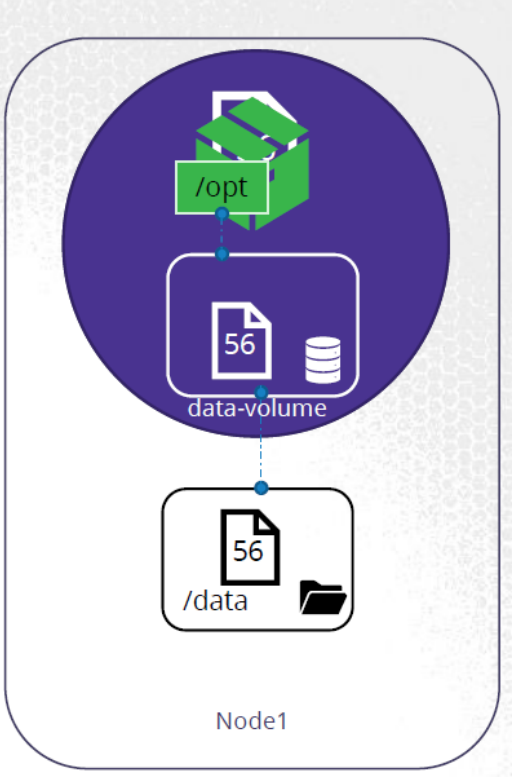
State Persistence

Source : https://www.youtube.com/watch?v=0swOh5C3OVM

Volume

Let's look at a simple implementation of volumes. We have a single node kubernetes cluster. We create a simple POD that generates a random between 1 and 100 and writes that to a file at /data/number.outand then gets deleted along with the random number. To retain the number generated by the pod, we create a volume. And a Volume needs a storage. 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 volumeMountsfield in each container to mount the data-volume to the directory /opt within the container. The random number 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 with the random number still lives on the host.

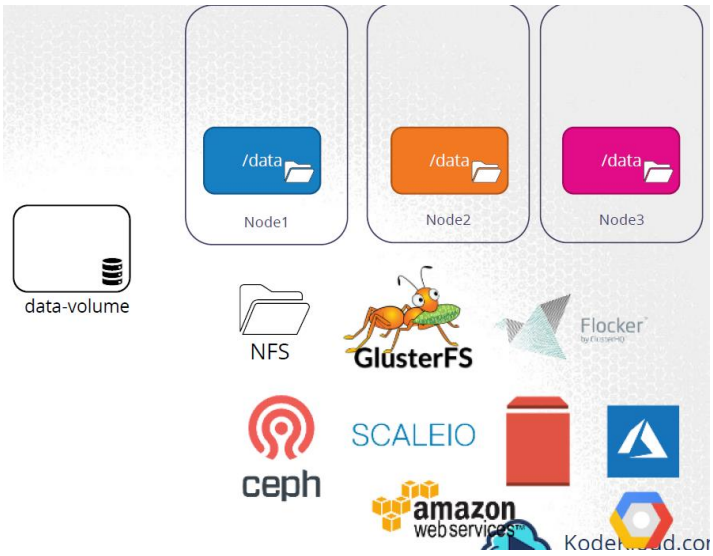
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume

  volumes:
  - name: data-volume
    hostPath:
      path: /data
      type: Directory
```



Let's take a step back and look at the Volume Storage option. We just used the hostPathoption 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.

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



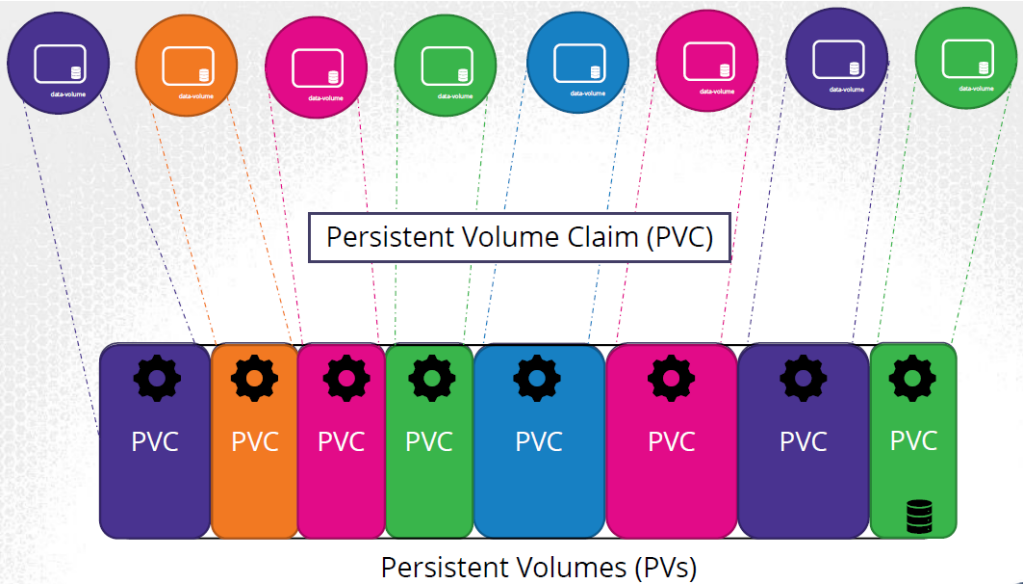
For example, to configure an AWS Elastic Block Store volume as the storage or the volume, we replace hostPathfield of the volume with awsElasticBlockStorefield along with the volumeIDand filesystem type. The Volume storage will now be on AWS EBS.
Well, that's it about Volumes in Kubernetes. We will now head over to discuss Persistent Volumes next.

```
volumes:
- name: data-volume
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

Persistent Volume

When we created volumes in the previous section we configured volumes within the POD definition file. So every configuration information required to configure storage for the volume goes within the pod definition file.
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. Instead, you would like to manage storage more centrally.
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 ReadOnly mode, or ReadWrite mode. The supported values are ReadOnlyMany, ReadWriteOnce or ReadWriteMany mode.
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. We will start with the hostPath option that uses storage from the node's local directory. Remember this option is not to be used in a production environment.
To create the volume run the kubectl create command and to list the created volume run the kubectl get persistentvolumes command.
Replace the hostPath option with one of the supported storage solutions as we saw in the previous lecture like AWS Elastic Block Store.

pv-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
   name: pv-vol1
spec:
  accessModes:
      - ReadWriteOnce
  capacity:
      storage: 1Gi
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

ReadOnlyMany

ReadWriteOnce

ReadWriteMany

Persistent Volume (PV)

```
kubectl create -f pv-definition.yaml
```

```
kubectl get persistentvolume
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON	AGE
pv-vol1	1Gi	RWO	Retain	Available				3m

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
 name: pv-log
 labels:
  type: pv
spec:
 capacity:
  storage: 100Mi
 accessModes:
 - ReadWriteMany
 hostPath:
    # directory location on host
    path: /pv/log
    # this field is optional
    type: Directory
 persistentVolumeReclaimPolicy: Retain
```

Persistent Volume YAML Example

NFS Storage

Use that physical storages in the **spec** section

How much:

Additional params, like access:

Nfs parameters:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-name
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.0
  nfs:
    path: /dir/path/on/nfs/server
    server: nfs-server-ip-address
```

Persistent Volume YAML Example

Google Cloud

How much:

Google Cloud parameters:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-volume
  labels:
    failure-domain.beta.kubernetes.io/zone: us-central1-a__us-central1-b
spec:
  capacity:
    storage: 400Gi
  accessModes:
  - ReadWriteOnce
  gcePersistentDisk:
    pdName: my-data-disk
    fsType: ext4
```

Persistent Volume YAML Example

Depending on storage type,
spec attributes differ

Node Affinity:

Local storage

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-pv
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/ssd1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - example-node
```

Persistent Volume YAML Example

Depending on storage type,
spec attributes differ

Complete list of storage backends supported by K8s:

the Pod must independently specify where to mo

Types of Volumes

Kubernetes supports several types of Volumes:

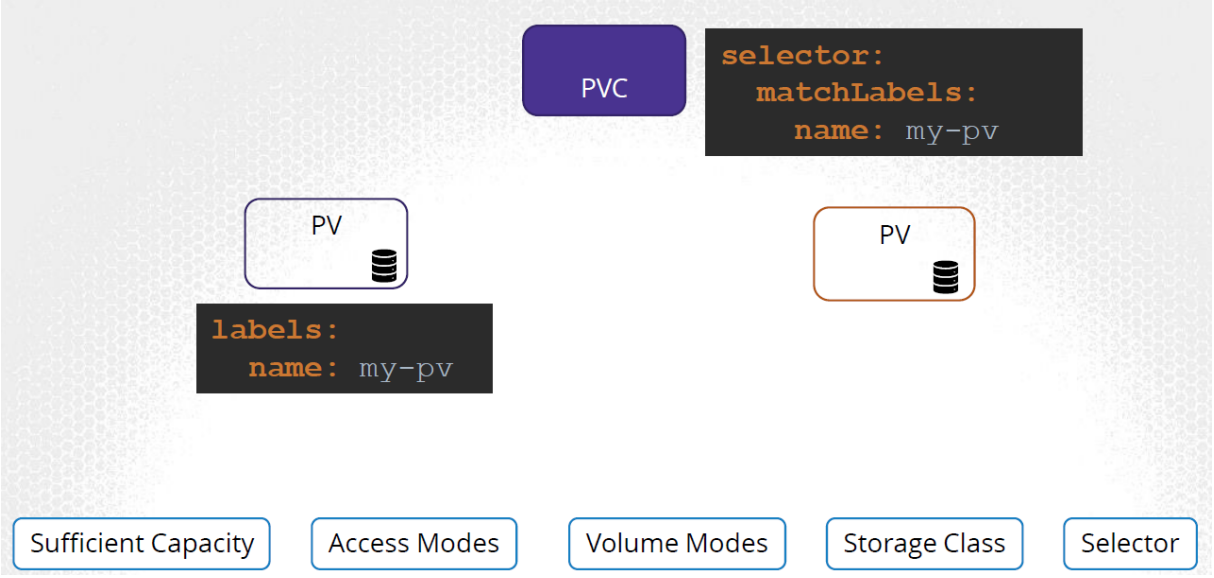
- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs
- cinder
- configMap
- csi
- downwardAPI
- emptyDir
- fc (fibre channel)
- flexVolume
- flocker

Persistent volume claim

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. Note that whichever PersistentVolume matching the criterias (accessModes and storage capacity for example) will be used by the claim, you don't have to precise a specific PV.

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.

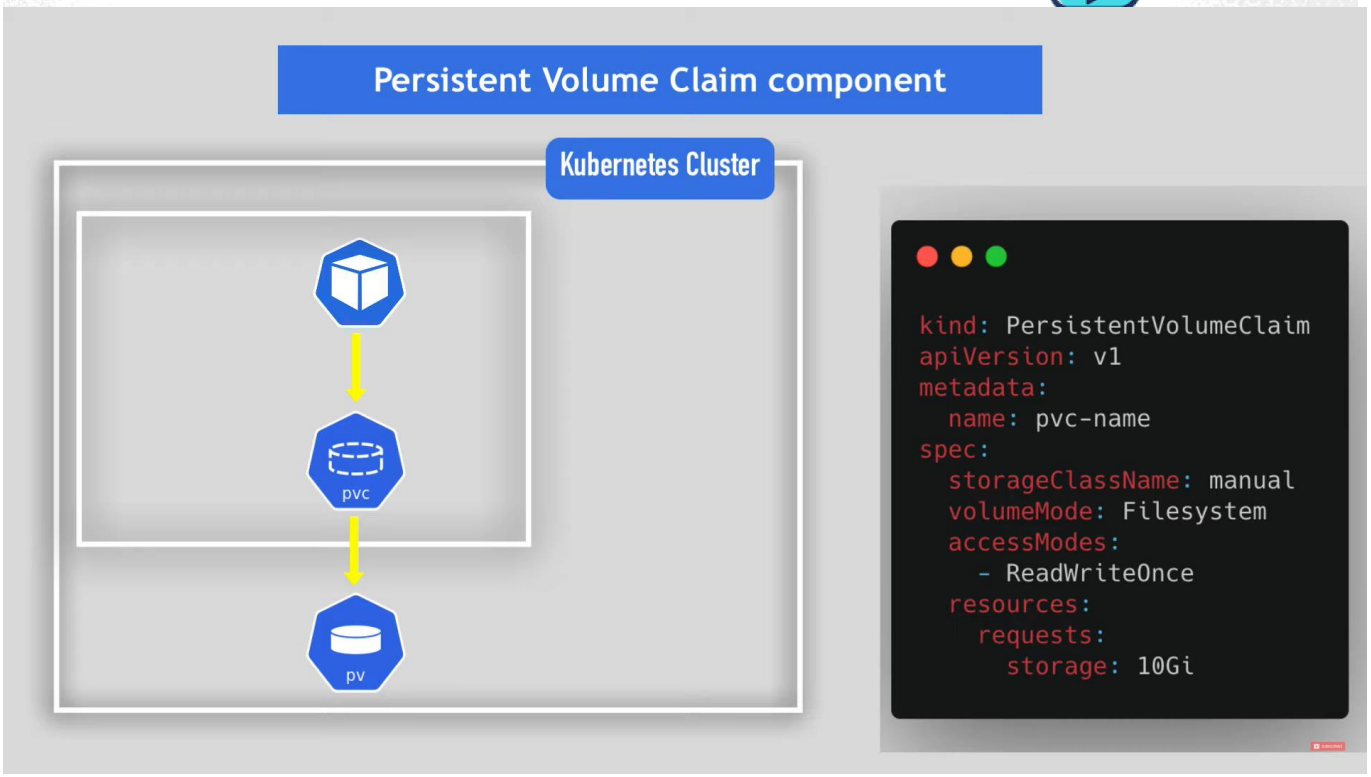
pvc-definition.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
     - ReadWriteOnce
  resources:
     requests:
       storage: 500Mi
```

```
kubectl get persistentvolumeclaim
NAME        STATUS    VOLUME    CAPACITY   ACCESS MODES
myclaim     Pending
```

```
kubectl create -f pvc-definition.yaml
```

KodeKloud.com



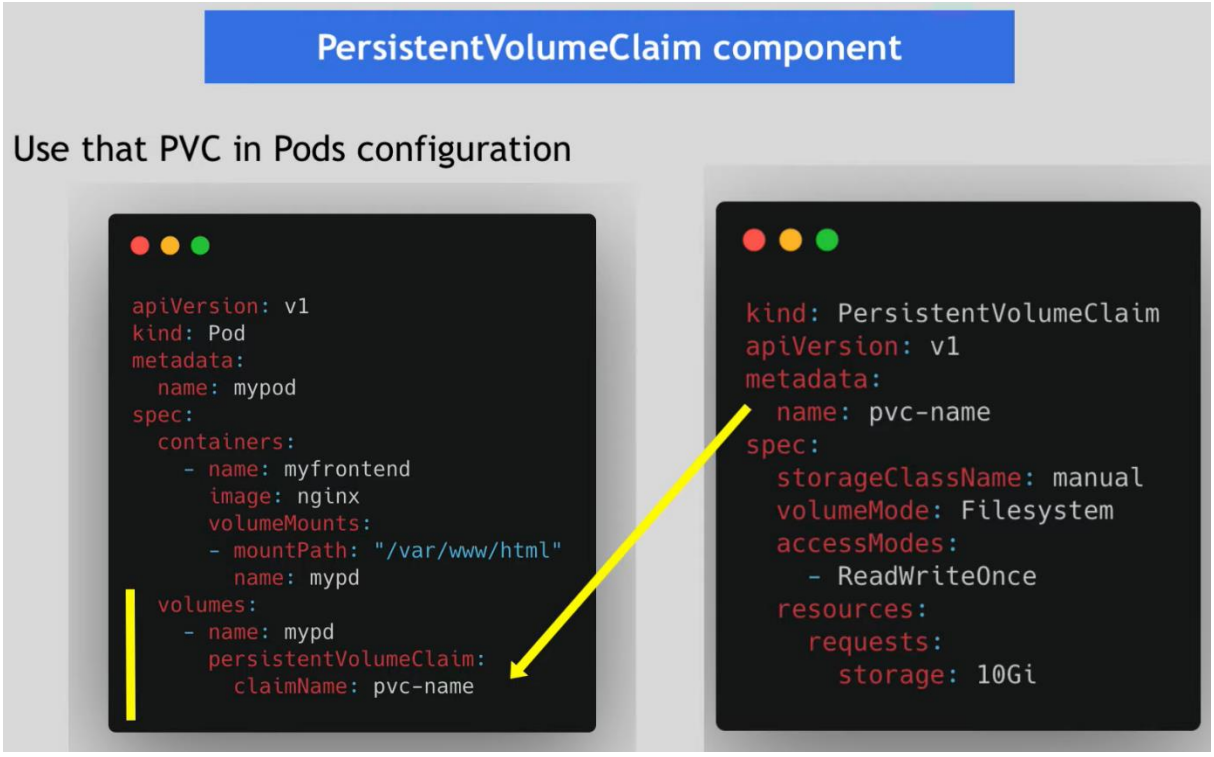
When the claim is created, kubernetes looks at the volume created previously. The access Modes match. The capacity requested is 500 Megabytes but the volume is configured with 1 GB of storage. Since there are no other volumes available, the PVC is bound to the PV.

pv-definition.yaml

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
      - ReadWriteOnce
  capacity:
      storage: 1Gi
 awsElasticBlockStore:
   volumeID: <volume-id>
   fsType: ext4
```

To delete a PVC run the kubectl delete persistentvolumeclaim command. But what happens to the Underlying Persistent Volume when the claim is deleted? You can chose 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.

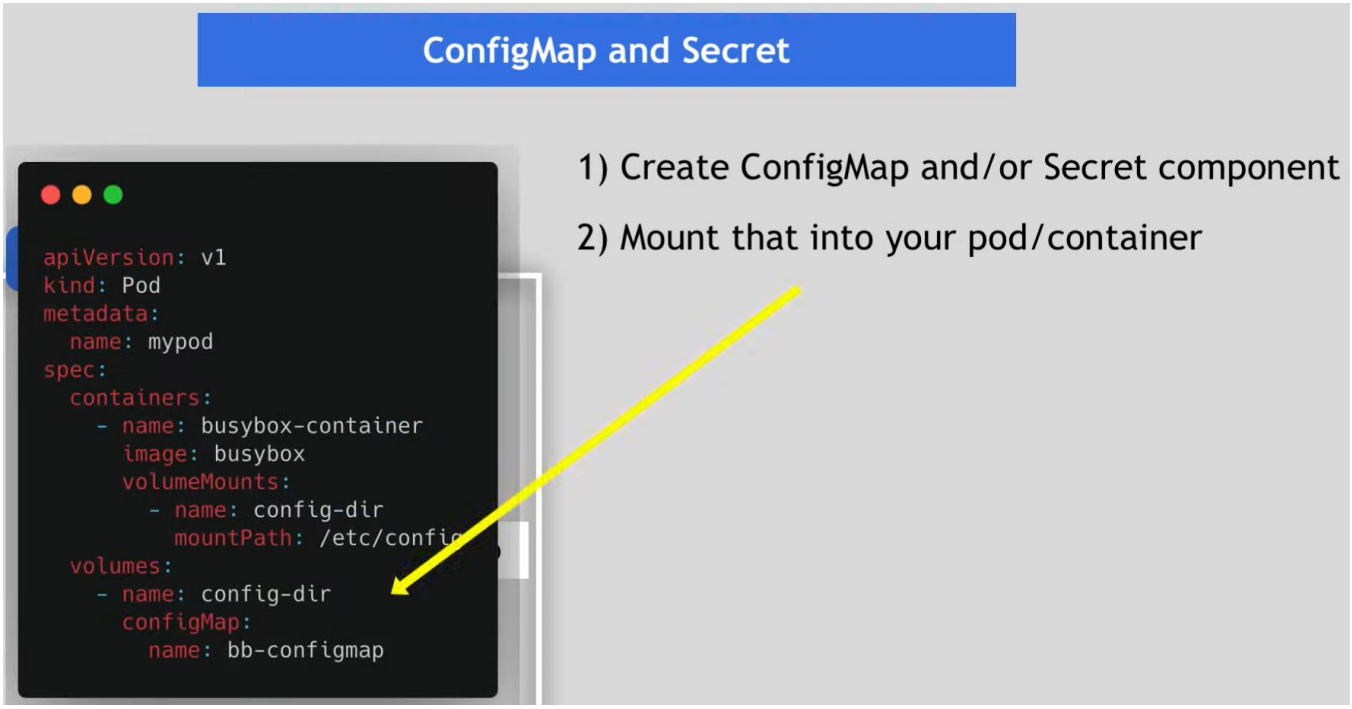
Using the PVC in Pod config :



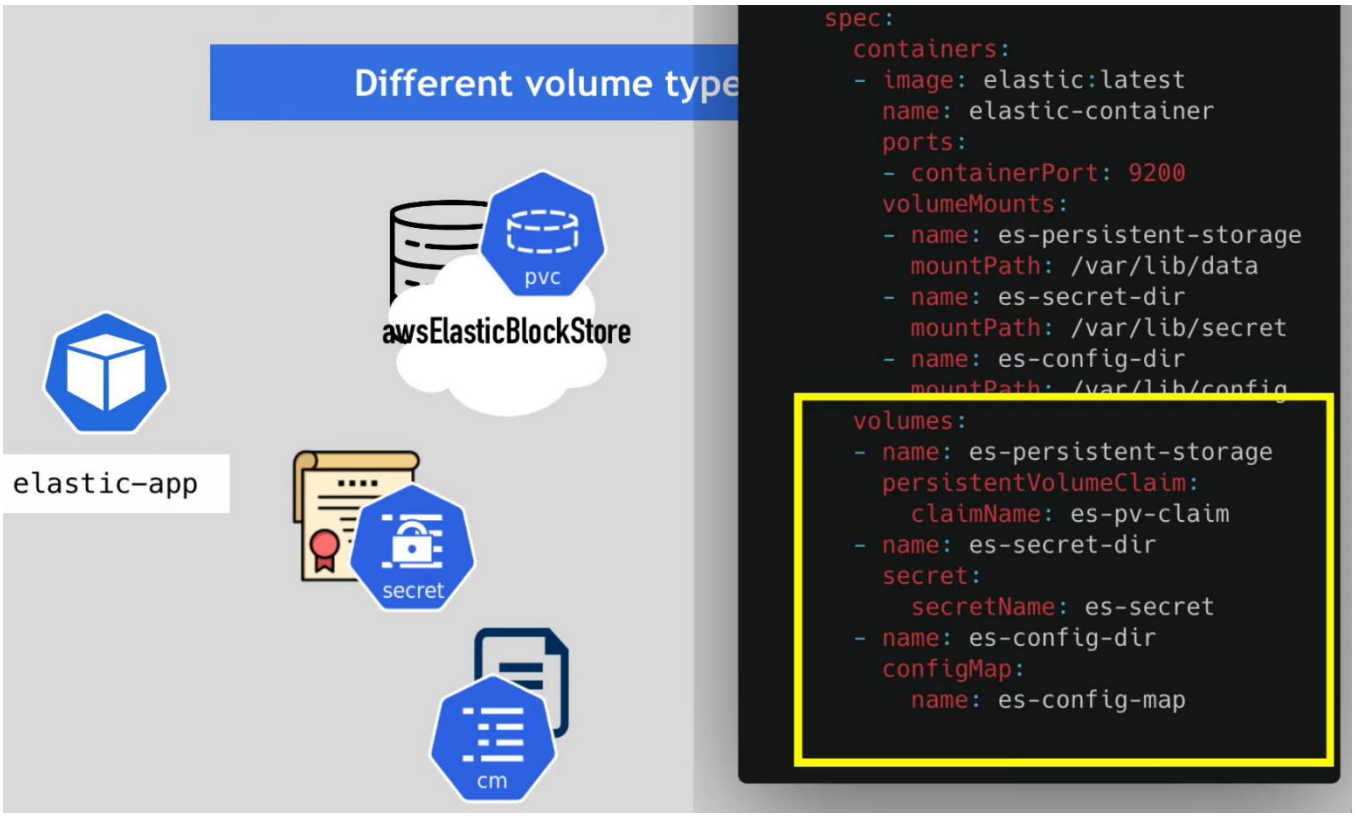
The claim and the pod using it must be in the same namespace.

ConfigMap and Secret components

Ex for certificates etc…

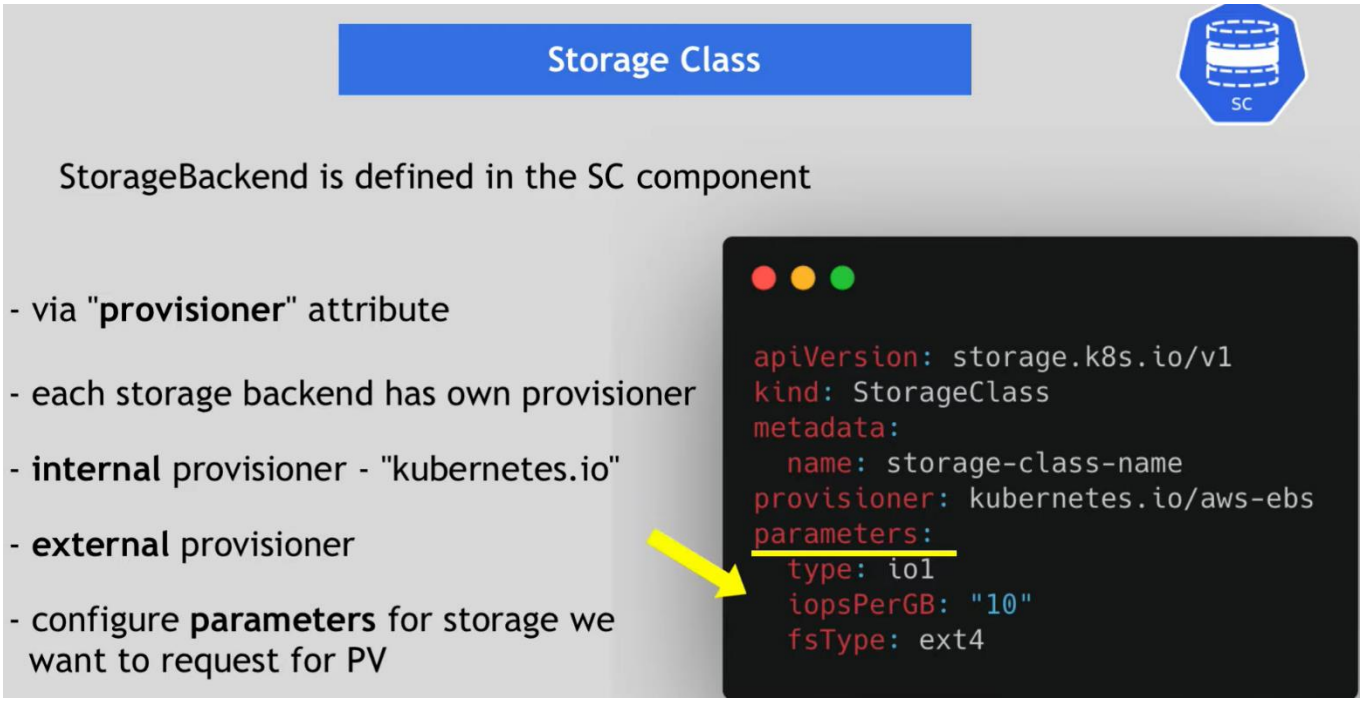


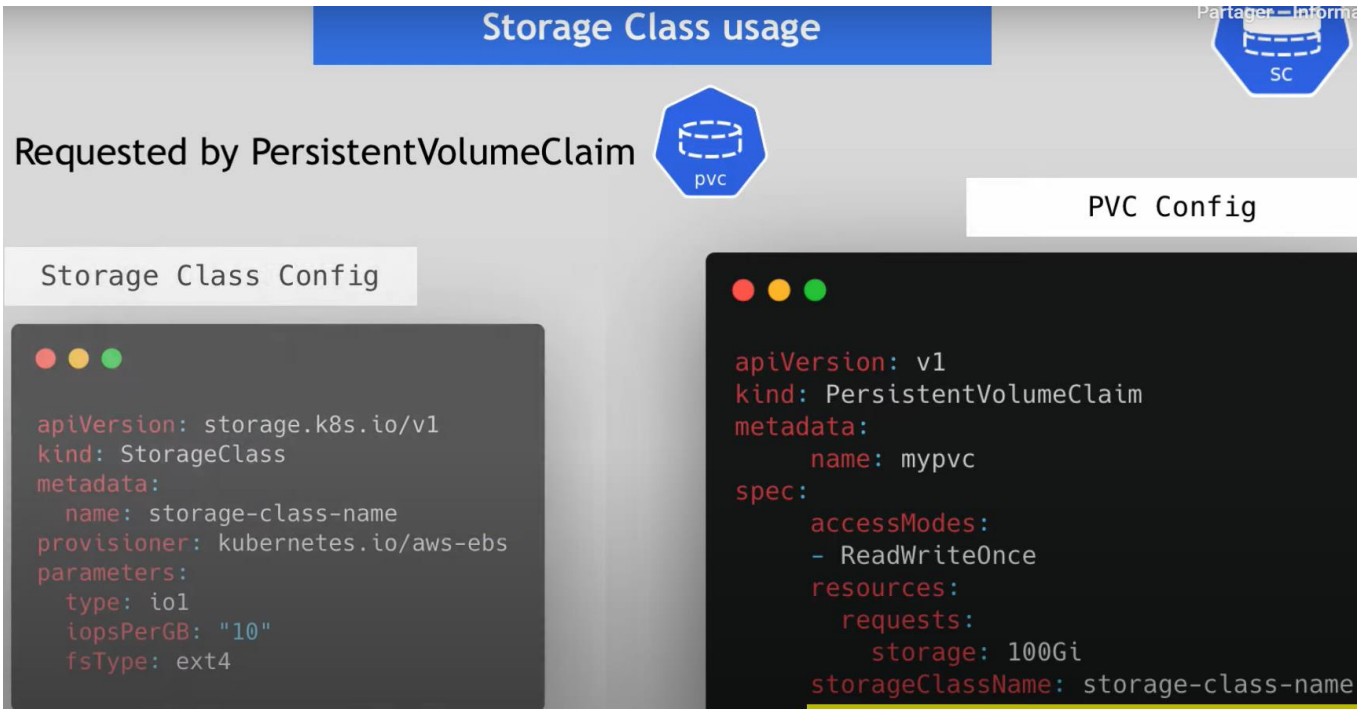
Example of multiple volumes types used by a pod



Storage Class

The storage class allows to dynamically create volumes to store datas (on a cloud for example, here aws provisionner, note that you can also create local volume) when a PersistentVolumeClaim is created (note : this will automatically create persistent volumes on your Kubernetes cluster).





Stateful Sets (optional for certification)

A stateful set object is similar to a deployment object in Kubernetes, but there are few differences:

The PODs are created in a sequential order. The Pods will wait for the previous Pod to be in a ready state to be deployed. Stateful sets assign a unique original index to each Pod starting from zero. Each Pod gets a unique name derived from this index combined with the stateful set name. Ex : mysql-0, mysql-1, mysql-2…

If for example we want to setup a replication for mysql db Pods, usually we have one master and several slaves. First, the data of the master has to be copied to the first slave (mysql-0 to mysql-1) and then the replication between the slave and the master has to be established.

Then to lighten the master workload, the data is copied from the first slave to the second one and after that the replication is set from the master to the second slave.

With a stateful set, the copy instruction can be done from the last Pod up to the next one sequentially, like that even if we add a new replica (mysql-3), the data will be copied from mysql-2 to mysql-3.

We can now easily define the master as the Pod mysql-0 for the different operation we want to execute.

A stateful set has the exact same file construction than a Deployment file but you **must** define a serviceName of a headless service (documentation about headless services bellow) :

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  minReadySeconds: 10 # by default is 0
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
```

When you delete StatefulSets, the pods will be deleted in the reverse order.

You can use an option to make the Pod deployment not sequential (in parallel like a classic deployment), and keep the benefits of StatefulSets (stable network ID).

Headless Services (optional for certification)

Usually if you want to refer another Pod inside a Pod, you use Services (ClusterIp), to associate a DNS name to the Service.

But what if we want different behaviours between our Pods inside a deployment or a StatefulSet? For example, we want an application to read a mysql db Pod information on slaves only and to do read and write on the master (see StatefulSets). By default the app will contact the service (ClusterIp) exposing the StatefulSet and that service will load balance between our port without any distinction.

For example to only contact the master in case of write request, on our app we have to use it's DNS (different from the DNS of the service ex : for the service DNS will be : mysql.default.svc.cluster.local, for the Pod it will be 10.40.2.8.default.pod.cluster.local) or it's IP. The problem is that the IP address is dynamically allowed so we won't be able to reach the Pod this way (if the Pod change, the IP will also change).

A headless service is used when you wan't to allocate a unic DNS entry for **each** Pod in a StatefulSet.

It is created like a normal service, but it does not have an IP of it's own and it does not perform any load balancing. It only creates a DNS entry for each pod using the pod name and a subdomain.

It will be defined following this rule :

```
podname.headless-service-name.namespace.svc.cluster-domain.local
```

Ex :

```
mysql-pod-1.mysql-headless-service.default.svc.cluster.local
```

Defining a nameless service is the same than defining a classic service, excepts that you have to set the clusterIP to 'None' :

```
apiVersion: v1
kind: Service
metadata:
  name: mysql-headless-service
spec:
  ports:
    - port: 3306
```

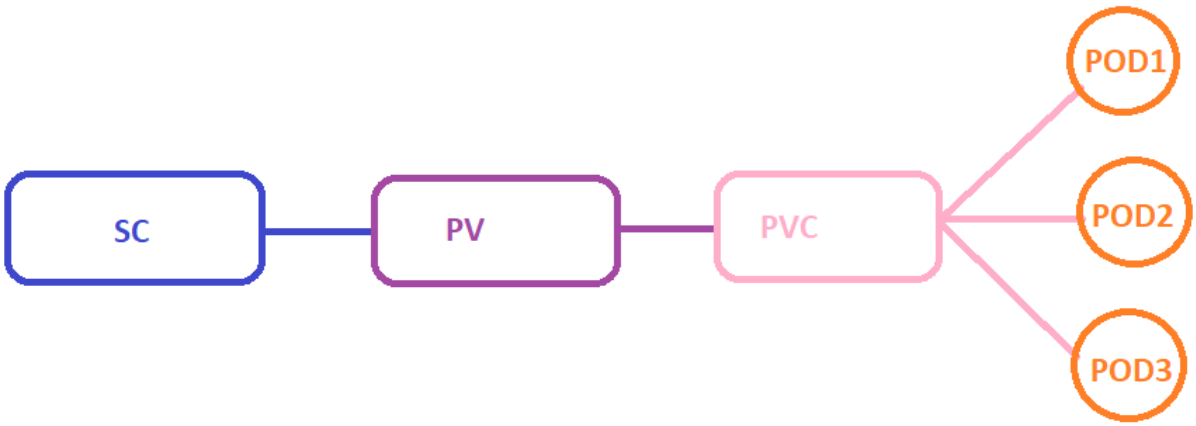
```
  selector:
    app: mysql
  clusterIP: None
```

The link is made on the StatefulSet file with the ‘serviceName’ instruction.

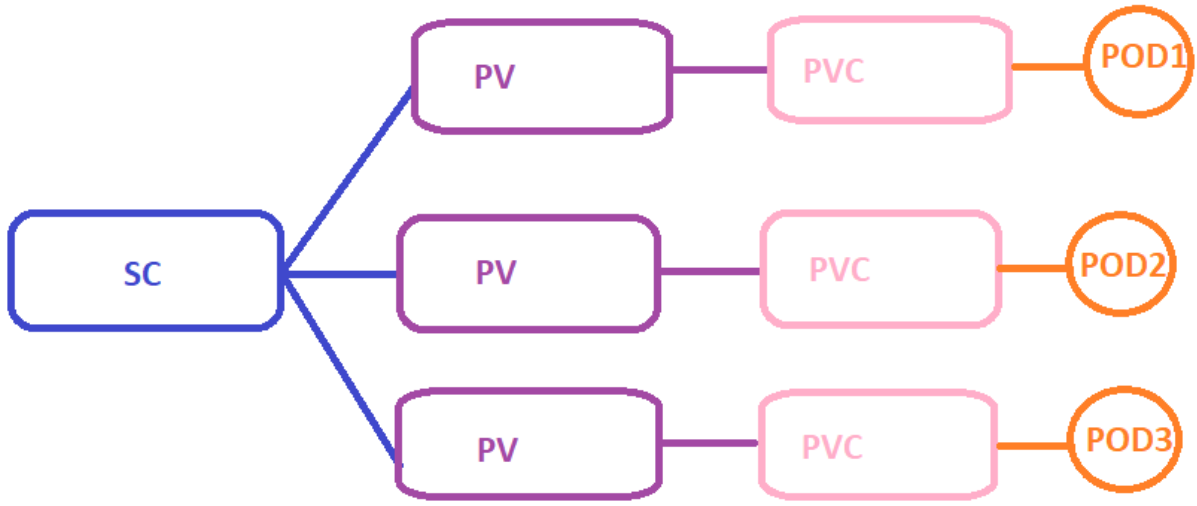
Storage in StatefulSets

If you define a persistentVolumeClaim in the Pod template of your StatefulSet (or your deployment), if the desired volume is a StorageClass (so it will automatically create a PV for your Claim), well only one PV (and one volume in the cloud) will be created for all your Pods replicas :

```
spec:
 containers:
 - name: mysql
   image: mysql
   volumeMounts:
   - mountPath: /var/lib/mysql
     name: data-volume
 volumes:
   - name: data-volume
     persistentVolumeClaim:
       claimName: data-volume
```



Now this can be the desired behavior, but what if we want to create a separate volume for each pod (for our mysql master and slave case for example)?



To achieve that, you can use a volumeClaimTemplate.

It is the same principle than a Pod template, instead of creating separates volumeClaim definition files, you will create a volumeClaim template inside the StatefulSet file.

Example from official kub doc :

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx # has to match .spec.template.metadata.labels
  serviceName: "nginx"
  replicas: 3 # by default is 1
  minReadySeconds: 10 # by default is 0
  template:
    metadata:
      labels:
        app: nginx # has to match .spec.selector.matchLabels
```

```
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

StatefulSet don't automatically delete the PVCs created or the associated volumes, instead if a Pod fails, it ensures that the newly created Pod is attached to the existing PVC, thus ensuring stable storage for Pods.

Namespaces

General information

When you create pods, deployments and services in a k8s cluster, they will be assigned to a namespace. If you don't precise any namespace, they will be assigned to the **default namespace** (automatically created by kubernetes).

Several pods and services are automatically created when running k8s to manage the networking solution, the DNS service etc… => to isolate them from the user and avoid accidental deletions, those pods are running under the **namespace kube-system**.

A third namespace is created automatically by kubernetes and is called **kube-public** => it manages the resources available to all users.

Namespaces are useful to isolate different kind of environments (as dev and prod etc…).

Each namespace can have different policies that define who can do what.

You can also manage (limit) the resources allocated for each namespace.

Ressources references

Resources under the same namespace can simply refers to each other by their names.

If required, resources in a namespace can reach resources from another namespace :

For this, you must append the name of the namespace to the name of the service :

Ex on the same namespace: to connect a db service to a pod, you will have to add the db service name on the configuration files of that pod :

```
mysql.connect("db-service")
```

On a different namespace :

```
mysql.connect("db-service.dev.svc.cluster.local")
```

This address is automatically added in kubernetes DNS when you create a service on a namespace, let's break it down a little bit :

°cluster.local : is the default domain name of the local k8s cluster

°svc : is the subdomain for services

°dev : is the namespace where the service is located

°db-service : is the name of the db service

Create a resource on the desired namespace in a yaml file (metadata)

```
apiVersion: v1
kind: Pod

metadata:
  name: nginx-pod
  namespace: dev
  labels:
    type: front-end
    app: nginx

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

Create a namespace from a yaml file

```
apiVersion: v1
kind: Namespace
metadata:
```

```
  name: dev
```

Configure resources quota for a namespace

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name:compute-quota
  namespace: dev
spec:
  hard:
    pods: “10”
    requests.cpu: “4”
    requests.memory: 5Gi
    limits.cpu: “10”
    limits.memory: 10Gi
```

Usefull commands

Create a namespace

```
kubectl create namespace dev
```

Get resource (pod, deployment, service…) by namespace :

```
kubectl get resource-type –-namespace=your-namespace
```

Note : if you don’t precise any namespace you will get the resources of the default namespace.

Create a resource from a yaml file in desired namespace :

```
kubectl create –f pod-definition.yml –-namespace=dev
```

Change namespace context (automatically be in a specific namespace instead of the default one):

```
kubectl config set-context $(kubectl config current-context) --namespace=dev
```

View resources in all namespaces :

```
kubectl get resource_type --all-namespaces
```

Certification Tip: Imperative Commands cheat sheet

While you would be working mostly the declarative way - using definition files, imperative commands can help in getting one time tasks done quickly, as well as generate a definition template easily. This would help save considerable amount of time during your exams.

Before we begin, familiarize with the two options that can come in handy while working with the below commands:

`--dry-run`: By default as soon as the command is run, the resource will be created. If you simply want to test your command , 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.

Use the above two in combination to generate a resource definition file quickly, that you can then modify and create resources as required, instead of creating the files from scratch.

POD

Create an NGINX Pod and a clusterip service at the same time (--expose=true)

```
kubectl run nginx --image=nginx --port=80 --expose=true
```

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

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

Service

Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379

```
kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml
```

(This will automatically use the pod's labels as selectors)

Or

`kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml` (This will not use the pods labels as selectors, instead it will assume selectors as **app=redis.** You cannot pass in selectors as an option. So it does not work very well if your pod has a different label set. So generate the file and modify the selectors before creating the service)

Create a Service named nginx of type NodePort to expose pod nginx's port 80 on port 30080 on the nodes:

```
kubectl expose pod nginx --port=80 --name nginx-service --type=NodePort --dry-run=client -o yaml
```

(This will automatically use the pod's labels as selectors, but you cannot specify the node port. You have to generate a definition file and then add the node port in manually before creating the service with the pod.)

Or

```
kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml
```

(This will not use the pods labels as selectors)

Both the above commands have their own challenges. While one of it cannot accept a selector the other cannot accept a node port. I would recommend going with the `kubectl expose` command. If you need to specify a node port, generate a definition file using the same command and manually input the nodeport before creating the service.

A quick note on editing PODs and Deployments

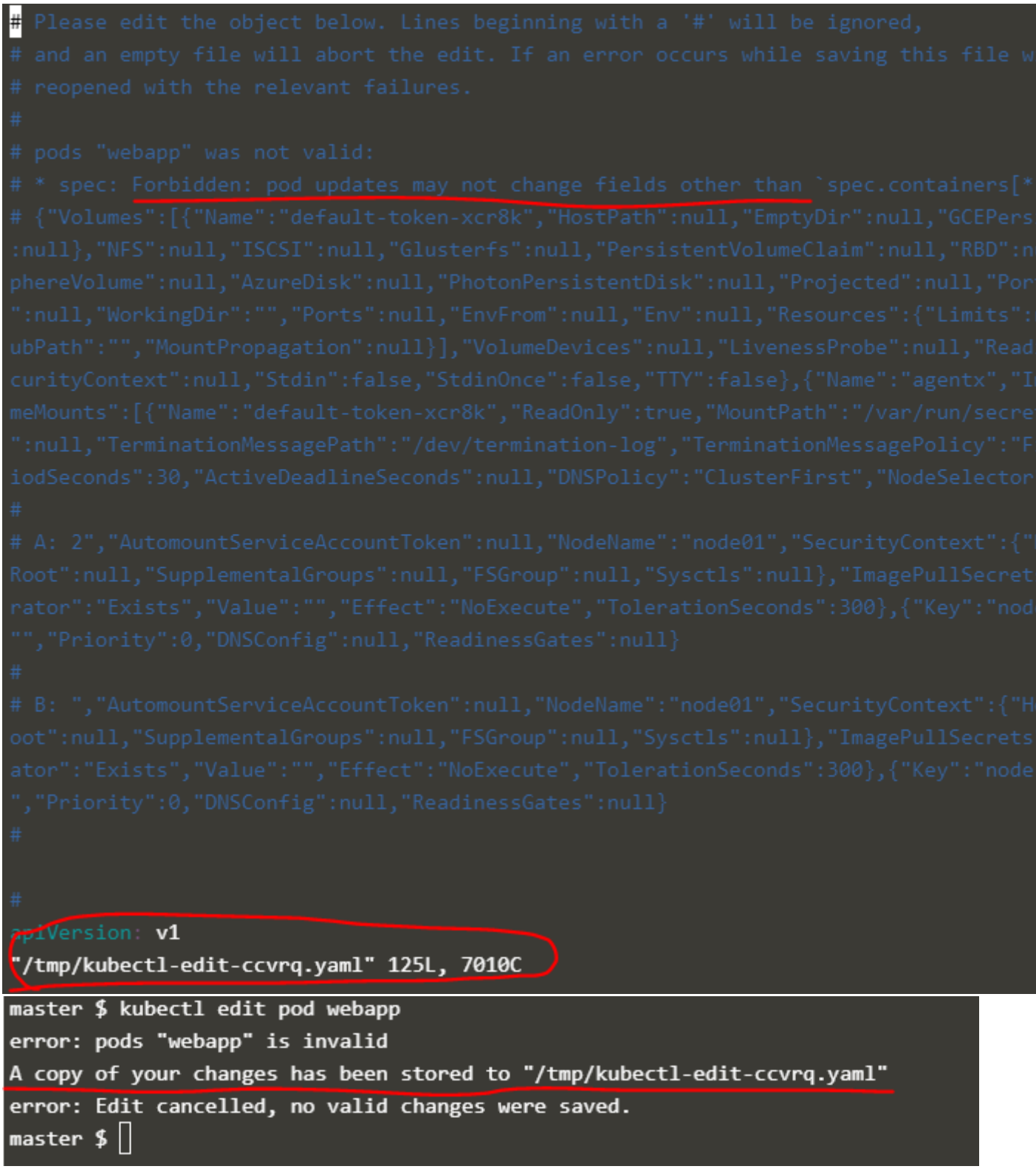
Edit a POD

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

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

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

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




```
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
```

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

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

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

```
vi my-new-pod.yaml
```

Then delete the existing pod

```
kubectl delete pod webapp
```

Then create a new pod with the edited file

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

Edit Deployments

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

```
kubectl edit deployment my-deployment
```

Security in kubernetes

ConfigMaps:

You can use configuration files (variables files) to define env variables => ConfigMap

Create a configmap

°Using imperative approach (command line) :

```
kubectl create configmap app-config \
        --from-literal=APP_COLOR=blue \
        --from-literal=APP_MOD=PROD
```

Here we create a configmap file named app-config with 2 env variables.

But this can be quite complicated when you have many configuration items.

You can precise a file to create the configmap this way:

```
kubectl create configmap app-config --from-file=app_config.properties
```

°Using declarative approach (file):

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_MODE: prod
```

Note: we use data and not specs.

```
kubectl apply -f configmapfile.yaml
```

Usefull commands:

```
kubectl get configmaps
kubectl describe configmaps
```

Use configmaps in Pod files (envFrom):

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

```
    envFrom:
      - configMapRef:
        name: app-config
```

You can also use only one value of the configmap instead of creating all the env variables :

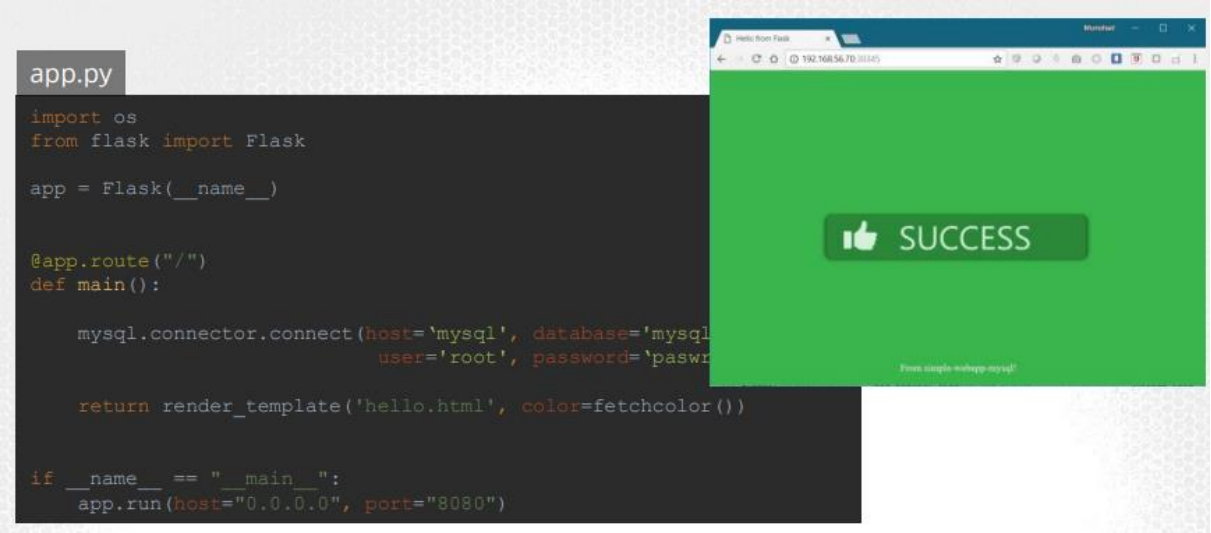
```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      ports:
        - containerPort: 8080
      env:
        - name: APP_COLOR
          valueFrom:
            configMapKeyRef:
              name: app-config
              key: APP_COLOR
```

You can also use the configmaps to define a volume (see volumes part of the doc):

```
volumes:
- name: app-config-volume
  configMap:
    name: app-config
```

Secrets

Create secrets



Here we have a simple python web application that connects to a mysql database. On success the application displays a successful message.

If you look closely into the code, you will see the hostname, username and password hardcoded. This is of-course not a good idea.

As we learned in the previous part, one option would be to move these values into a configMap. The configMap stores configuration data in plain text, so while it would be OK to move the hostname and username into a configMap, it is definitely not the right place to store a password.

This is were secrets come in. Secrets are used to store sensitive information, like passwords or keys. They are similar to configMaps, except that they are stored in an encoded or hashed format. As with configMaps, there are two steps involved in working with Secrets. First, create the secret and second inject it into Pod.

```
kubectl create secret generic app-secret \
        --from-literal=DB_Host=mysql \
        --from-literal=DB_User=root \
        --from-literal=DB_Password=paswrd
```

OR

```
kubectl create secret generic app-secret \
        --from-file=app_secret.properties
```

OR

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw=
  DB_User: cm9vdA
  DB_Password: cGFzd3Jk
```

```
kubectl create -f secret-data.yaml
```

Encode secrets

However, one thing we discussed about secrets was that they are used to store sensitive data and are stored in an encoded format. Here we have specified the data in plain text, which is not very safe. So, while creating a secret with the declarative approach, you must specify the secret values in a hashed format. So you must specify the data in an encoded form like this. But how do you convert the data from plain text to an encoded format? You have to use the base64 command :

```
echo –n ‘root’ | base64
cm9vdA==
```

View secrets

```
kubectl get secrets
kubectl describe secrets
kubectl get secret app-secret –o yaml
```

Decode secrets;

```
echo –n ‘cm9vdA==’ | base64 --decode
```

Use secrets in Pod files

°Using a secret file

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: simple-webapp-color
  labels:
    app: simple-webapp-color
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      ports:
        - containerPort: 8080
      envFrom:
      - secretRef:
        name: app-secret
```

°Value by value

```
env:
  - name: DB_Password
    valueFrom:
      secretKeyRef:
        name: app-secret
        key: DB_Password
```

°As volumes

```
volumes:
  -  name: app-secret-volume
    secret:
      secretName: app-secret
```

If you were to mount the secret as a volume in the Pod, each attribute in the secret is created as a file with the value of the secret as its content. In this case, since we have 3 attributes in our secret, 3 files are created.

```
ls /opt/app-secret-volumes
DB_Host         DB_Password           DB_User
```

And if we look at the contents of the DB_password file, we see the password inside it. That's it for this lecture, head over to the coding exercises and practice working with secrets.

°Quick note

Remember that secrets encode data in base64 format. Anyone with the base64 encoded secret can easily decode it. As such the secrets can be considered as not very safe.

The concept of safety of the Secrets is a bit confusing in Kubernetes. The kubernetes documentation page and a lot of blogs out there refer to secrets as a "safer option" to store sensitive data. They are safer than storing in plain text as they reduce the risk of accidentally exposing passwords and other sensitive data. In my opinion it's not the secret itself that is safe, it is the practices around it.

Secrets are not encrypted, so it is not safer in that sense. However, some best practices around using secrets make it safer. As in best practices like:

- Not checking-in secret object definition files to source code repositories.
- Enabling Encryption at Rest for Secrets so they are stored encrypted in ETCD.
- https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/

Also the way kubernetes handles secrets. Such as:

- A secret is only sent to a node if a pod on that node requires it.
- Kubelet stores the secret into a tmpfs so that the secret is not written to disk storage.
- Once the Pod that depends on the secret is deleted, kubelet will delete its local copy of the secret data as well.

Read about the protections and risks of using secrets here:

https://kubernetes.io/docs/concepts/configuration/secret/#risks

Having said that, there are other better ways of handling sensitive data like passwords in Kubernetes, such as using tools like Helm Secrets, HashiCorp Vault. I hope to make a lecture on these in the future.

Encrypting data at rest

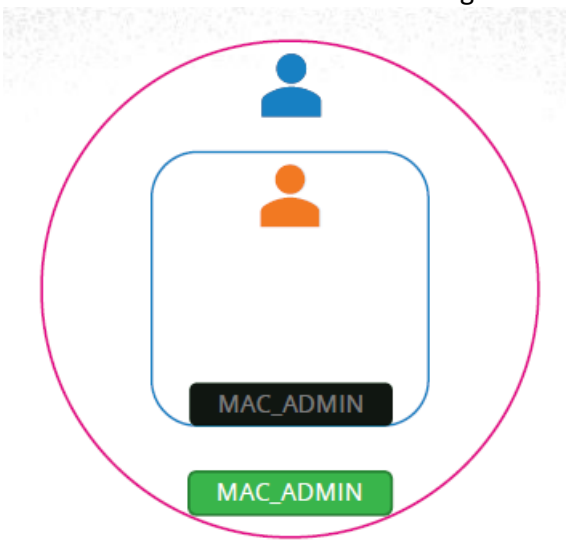
https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/

Security contexts

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 (see docker course).

As you know already, in Kubernetes containers are encapsulated in PODs. You may chose to configure the security settings at a container level….

… or at a POD level. If you configure it at a POD level, the settings will carry over to all the containers within the POD. If you configure it at both the POD and the Container, the settings on the container will override the settings on the POD.:



Let us start with a POD definition file. This pod runs an ubuntu image with the sleep command. To configure security context on the container, add a field called securityContextunder the spec section of the pod. Use the runAsUseroption to set the user ID for the POD.

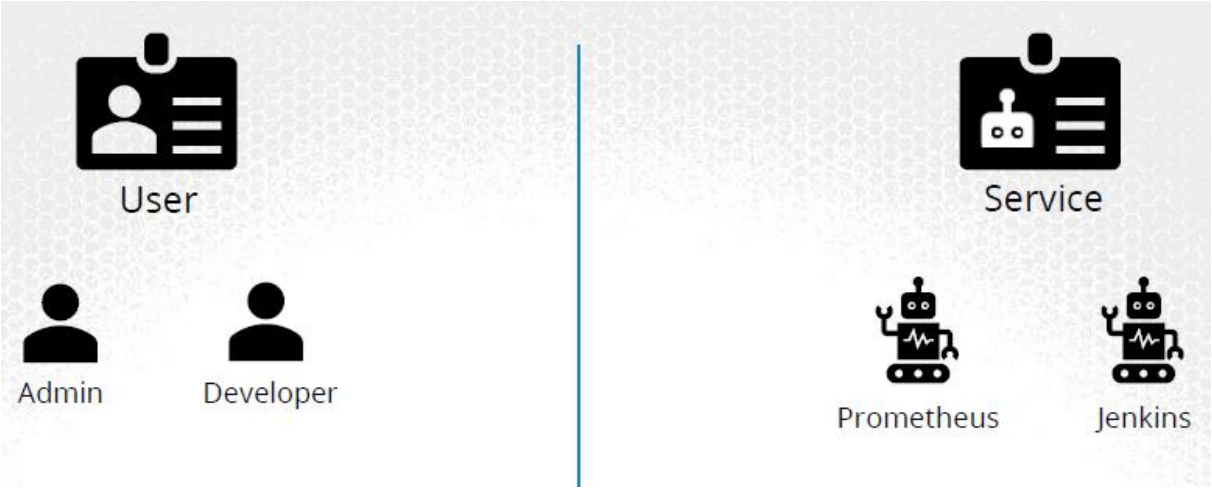
```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  securityContext:
    runAsUser: 1000
    capabilities:
      add: ["MAC_ADMIN"]
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "3600"]
```

To set the same configuration on the container level, move the whole section under the container specification like this.
To add capabilities use the capabilities option and specify a list of capabilities to add to the POD.

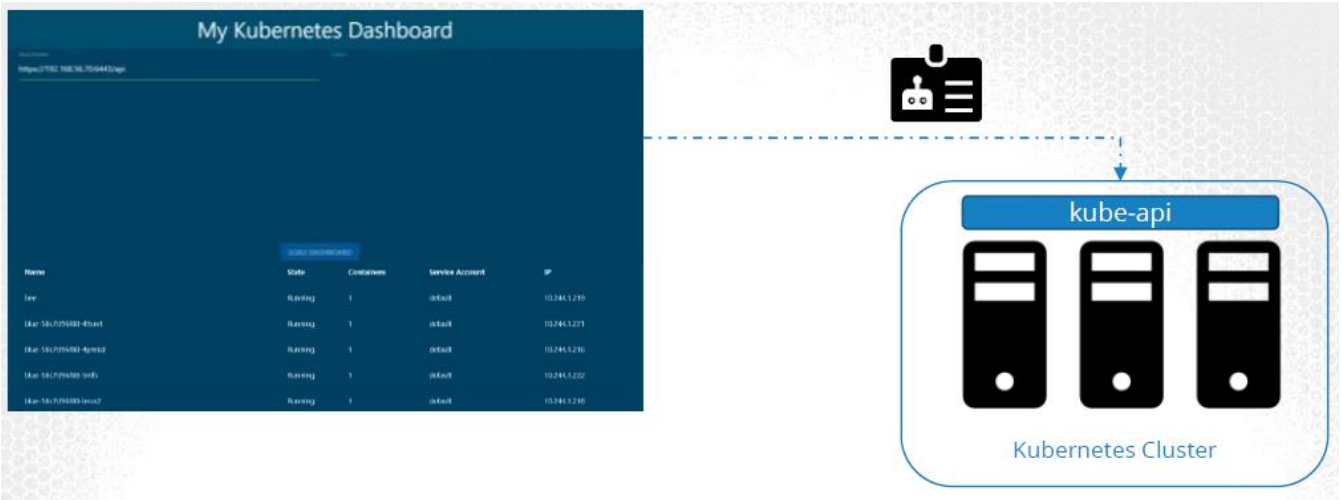
```
apiVersion: v1
kind: Pod
metadata:
  name: web-pod
spec:
  containers:
    - name: ubuntu
      image: ubuntu
      command: ["sleep", "3600"]
      securityContext:
        runAsUser: 1000
        capabilities:
          add: ["MAC_ADMIN"]
```

ServiceAccounts

There are two types of accounts in Kubernetes. A user account and a service account. As you might already know, the user account is used by humans. And service accounts are used by machines. A user account could be for an administrator accessing the cluster to perform administrative tasks, a developer accessing the cluster to deploy applications etc. A service account, could be an account used by an application to interact with the kubernetes 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.



Let's take an example. I have built a simple kubernetes dashboard application named, my-kubernetes-dashboard. It's a simple application built in Python and all that it does when deployed is retrieve the list of PODs on a kubernetes cluster by sending a request to the kubernetes API and display it on a web page. In order for my application to query the kubernetes API, it has to be authenticated. For that we use a service account.



To create a service account run the command kubectlcreate service account followed by the account name, which is dashboard-sain this case. To view the service accounts run the kubectlget serviceaccountcommand. This will list all the service accounts.
When the service account is created, it also creates a token automatically. The service account token is what must be used by the external application while authenticating to the Kubernetes

API. The token, however, is stored as a secret object. In this case its named dashboard-sa-token-kbbdm.

```
kubectl create serviceaccount dashboard-sa

serviceaccount "dashboard-sa" created
```

```
kubectl get serviceaccount

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

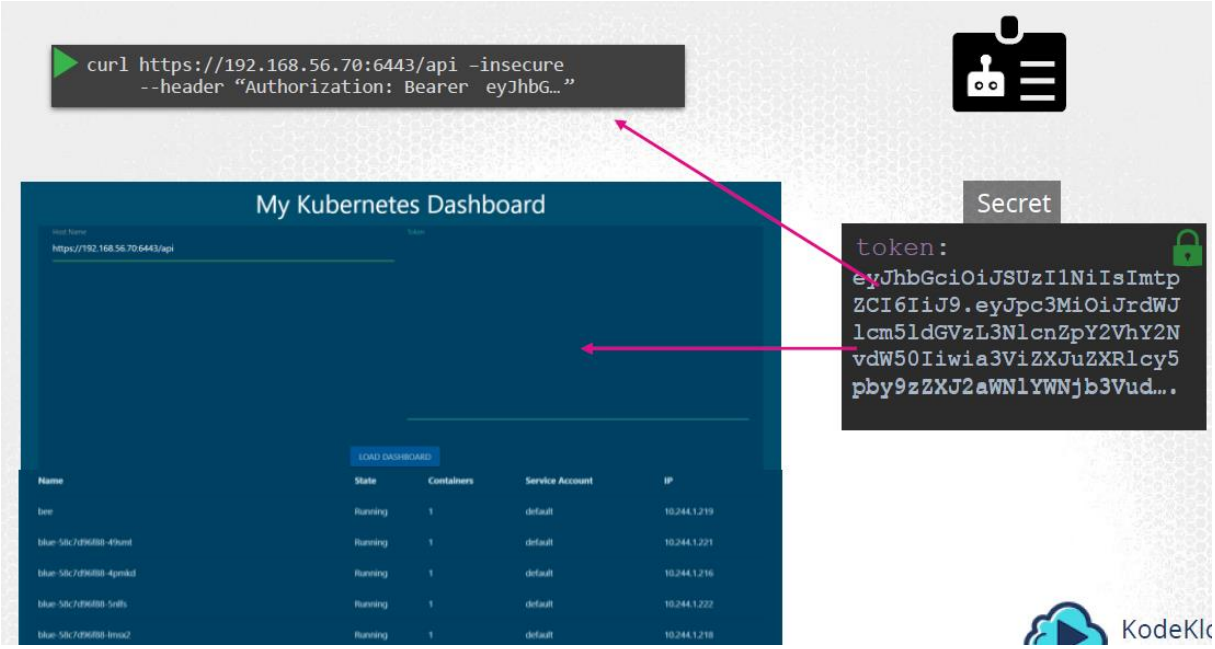
```
kubectl describe serviceaccount dashboard-sa

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.

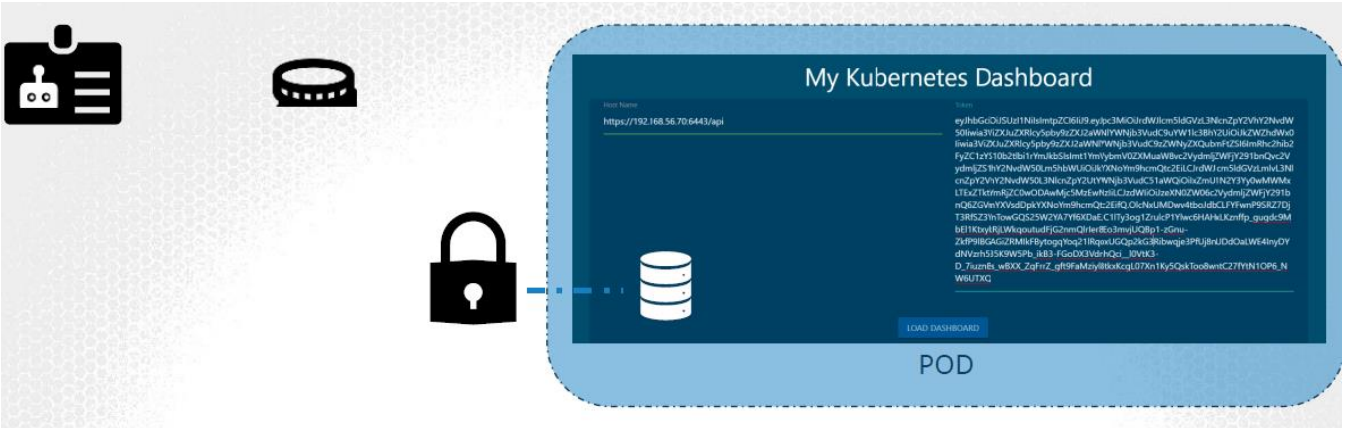
```
kubectl describe serviceaccount dashboard-sa

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>
```

```
kubectl describe secret dashboard-sa-token-kbbdm

Name:         dashboard-sa-token-kbbdm
Namespace:    default
Labels:       <none>

Type:  kubernetes.io/service-account-token

Data
====
ca.crt:     1025 bytes
namespace:  7 bytes
token:
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL
3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3V
udC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3
```

This token can then be used as an authentication bearer token while making a rest call to the kubernetes API. For example in this simple example using curl you could provide the bearer token as an Authorization header while making a rest call to the kubernetes API.
In case of my custom dashboard application, copy and paste the token into the tokens field to authenticate the dashboard application.

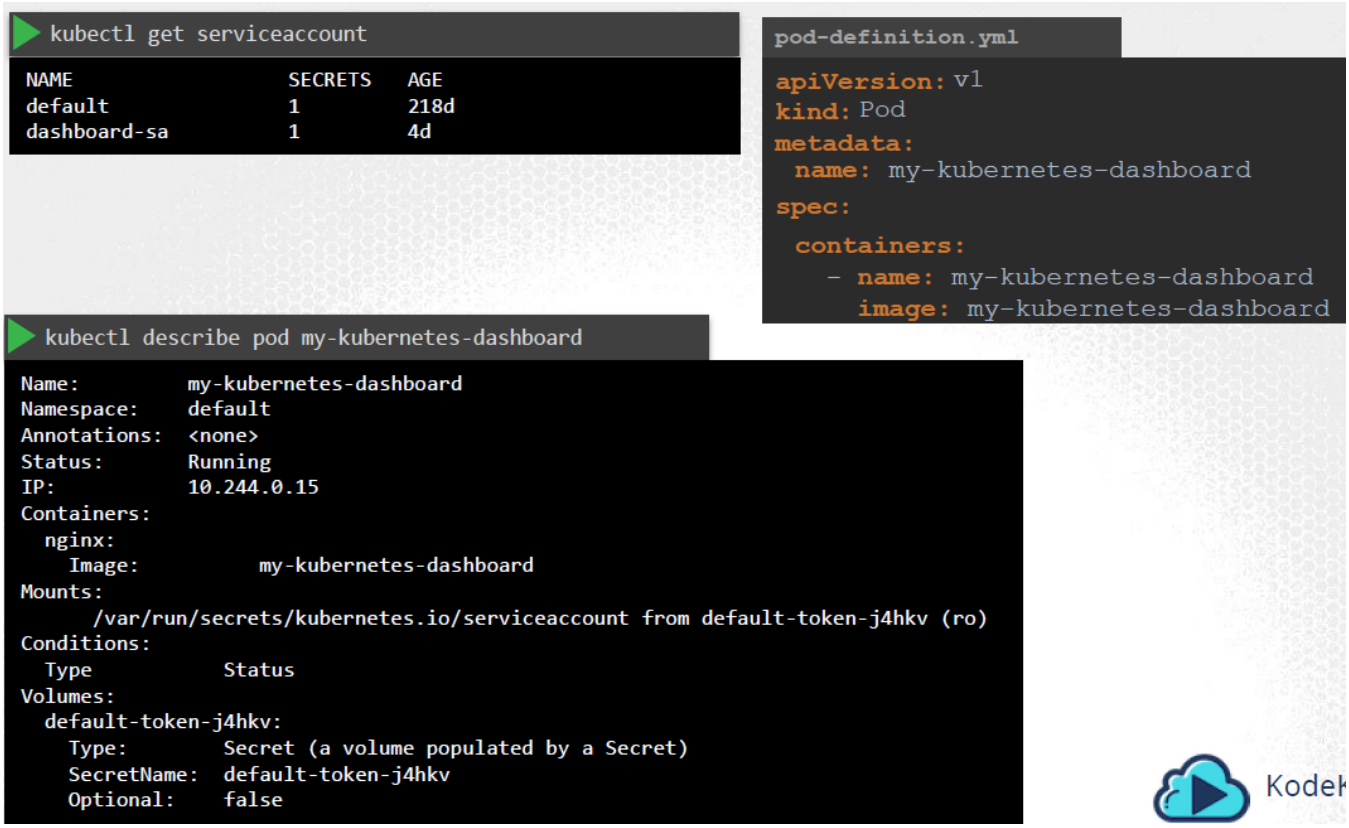


So, that’s how you create a new service account and use it. You can create a service account, assign the right permissions using Role based access control mechanisms (which is out of scope for this course) and export your service account tokens and use it to configure your third party application to 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.



If you go back and look at the list of service accounts, you will see that there is a default service account that exists already. For every namespace in kubernetes a service account named default is automatically created. Each namespace has its own default service account.

Whenever a POD is created the default service account and its token are automatically mounted to that POD as a volume mount. For example, we have a simple pod definition file that creates a POD using my `custom kubernetes dashboard` image. We haven't specified any secrets or volume mounts. However when the pod is created, if you look at the details of the pod, by running the kubectldescribe pod command, you see that a volume is automatically created from the secret named default-token-j4hkv, which is in fact the secret containing the token for the default service account. The secret token is mounted at location /var/run/secrets/kubernetes.io/serviceaccountinside the pod. So from inside the pod if you run the ls command.



If you list the contents of the directory inside the pod, you will see the secret mounted as 3 separate files. The one with the actual token is the file named token. If you list the contents of that file you will see the token to be used for accessing the kubernetes API. Now remember that the default service account is very much restricted. It only has permission to run basic kubernetes API queries.

```
kubectl describe pod my-kubernetes-dashboard
```

```
Name:         my-kubernetes-dashboard
Namespace:    default
Annotations:  <none>
Status:       Running
IP:           10.244.0.15
Containers:
  nginx:
    Image:          my-kubernetes-dashboard
Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-j4hkv (ro)
Conditions:
  Type           Status
Volumes:
  default-token-j4hkv:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-j4hkv
    Optional:    false
```

```
kubectl exec -it my-kubernetes-dashboard ls /var/run/secrets/kubernetes.io/serviceaccount
```

```
ca.crt  namespace  token
```

```
kubectl exec -it my-kubernetes-dashboard cat /var/run/secrets/kubernetes.io/serviceaccount/token
```


```
eyJhbGciOiJSUzI1NiIsImtpZCI6IiJ9.eyJpc3MiOiJrdWJlcm5ldGVzL3NlcnZpY2VhY2NvdW50Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3V
udC9uYW1lc3BhY2UiOiJkZWZhdWx0Iiwia3ViZXJuZXRlcy5pby9zZXJ2aWNlYWNjb3VudC9zZWNyZXQubmFtZSI6ImRlZmF1bHQtdG9rZW4tajRoa3Y
iLCJrdWJlcm5ldGVzLmlvL3NlcnZpY2VhY2NvdW50L3NlcnZpY2UtYWNjb3VudC5uYW1lIjoiZGVmYXVsdCIsImt1YmVybmV0ZXMuaW8vc2VydmljZWF
jY291bnQvc2VydmljZS1hY2NvdW50LnVpZCI6IjcxZGM4YWExLTU2MGMtMTFlOC04YmI0LTA4MDAyNzkzMTA3MiIsInN1YiI6InN5c3RlbTpzZXJ2aWN
```


If you decode this token using the command :

```
jq -R 'split(".")' | select(length > 0) | .[0],.[1] | @base64d | fromjson' <<< eyJhbGci....
```

OR

Pasting it on the jwt.io website (jwt token).

You will see that there is no expiration date on this token. So the jwt is valid as long as the service account exists.

If you'd like to use a different serviceAccount, such as the ones we just created, modify the pod definition file to include a serviceAccountfield 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. When you look at the pod details now, you see that the new service account is being used.

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

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  serviceAccount: dashboard-sa
```

```
kubectl describe pod my-kubernetes-dashboard
Name:         my-kubernetes-dashboard
Namespace:    default
Annotations:  <none>
Status:       Running
IP:           10.244.0.15
Containers:
  nginx:
    Image:          my-kubernetes-dashboard
Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from dashboard-sa-token-kbbdm (ro)
Conditions:
  Type           Status
Volumes:
  dashboard-sa-token-kbbdm:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  dashboard-sa-token-kbbdm
    Optional:    false
```

So remember, kubernetes automatically mounts the default service account if you haven't explicitly specified any. You may choose not to mount a service account automatically by setting the automountServiceAccountTokenfield to false in the POD spec section.

Well that's it for this lecture. Head over to the practice exercises section and practice working with service accounts. We will configure the custom kubernetes dashboard with the right service account.

```
pod-definition.yml
apiVersion: v1
kind: Pod
metadata:
  name: my-kubernetes-dashboard
spec:
  containers:
    - name: my-kubernetes-dashboard
      image: my-kubernetes-dashboard
  automountServiceAccountToken: false
```

The jwt token has a number of security issues, thus the TokenRequestAPI was introduced to enhance security in v1.22 of kubernetes.

Token generated by the TokenRequestAPI are audient bound, time bound and object bound and so, more secure.

Since v1.22, when a pod is created, it no longer uses the service account secret token that we just saw. Instead, a token with a defined lifetime is generated through the token request API and this token is then mounted as a projected volume into the pod.

With version 1.24, another improvement was made. In the past when a serviceaccount was created, it automatically created a secret with a token that has no expiration date and is not bound to any audience. This was automatically mount as a volume to pods using the

serviceaccount (as we saw before). Since the new version, when a serviceaccount is created, it no longer automatically creates a secret token. You must run the command :

```
kubectl create token your-service-account
```

Now, this time the token has an expiration date (usually one hour after running the command, you can pass the lifetime of the token as an option).

To now create a secret token with no lifetime limit, you have to create a secret object with the type set to kubernetes.io/service-account-token and the name of the service account passed as a metadata:

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: mysecretname
  annotations:
    kubernetes.io/service-account.name: dashboard-sa
```

But you should use this method ONLY if you can't use the service token API to generate tokens because it is less secure.

Network Policies

See Network Policies part of the Services section.

Authentication

The kube-apiserver is the center of all operations within Kubernetes. We interact with the kube-apiserver via the kuctl utility or through direct api calls. You can perform almost any operation on your Kube Cluster through the apiserver.

Controlling the access to the apiserver is the first line of defence of your kube cluster.

You have to define who can access the cluster (authentication) and what can they do (authorization).

How to setup authentication:

° Files - Username and Passwords

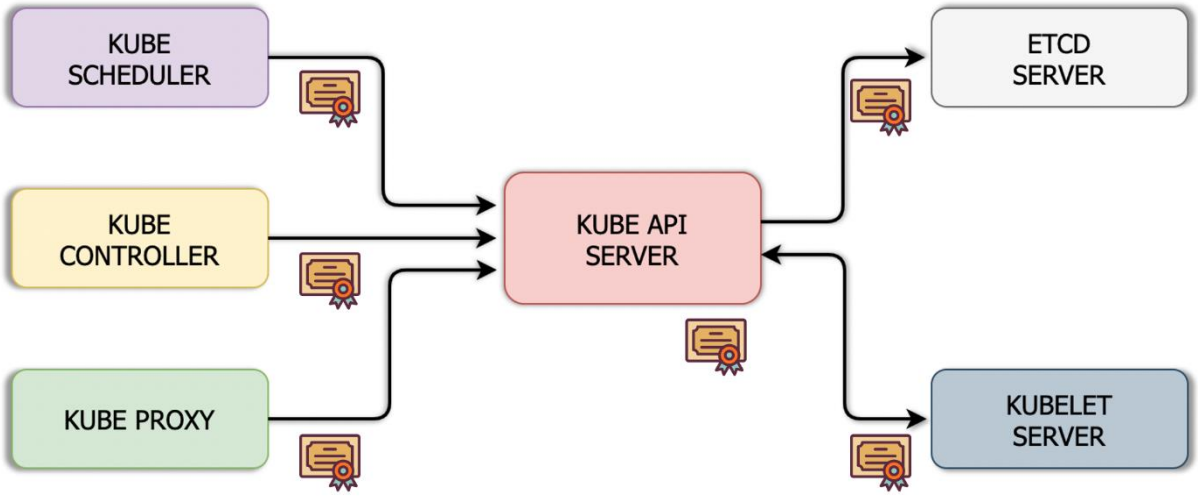
° Files - Username and Tokens

° Certificates

° External Authentication providers - LDAP

° Service Accounts (for machines)

All communications between the different components of the cluster (including master and worker nodes, for ex kube proxy and kubelet for workers) are secured using TLS encryption.



So who are the users who have access to our Kubernetes cluster?

°Admins

°Developers

°End Users (using the app exposed on our cluster)

°Third party applications bots for integration purposes

Security of the end users is managed by the used applications internally, so we don't have to take them into account for the authentication topic.

So we still have the humans user who can access our cluster and robots such as other processes, services or applications that requires access to the cluster.

Natively Kub doesn't manage user accounts natively, it relies on an external source like a file with users details or certificates or a third party identity service like LDAP to manage these users.

But Kubernetes can manage service accounts using kubernetes api to manage bots (see service accounts).

So we will focus on users, whether they are accessing the cluster through kubectl tool or the API directly => all these requests go through the Kube API server.

The kube-apiserver authenticate the request before processing it.

There are different authentication mechanisms that can be configured.

Authentication with static password and token files

°User file:

You can create a list of users and password in a csv file and use it as a source for user information. The file has three or four columns : password, username, userid and user-group (optional):

password123,user1,u0001,group1

password123,user2,u0002,group2

…

We can then pass the file to our kube-apiserver.service file or using kubeadm in the /etc/kubernetes/manifests/kube-apiserver.yaml Pod definition file and add the line in the command section, then restart the Pod:

```
--basic-auth-file=user-details.csv
```

Now to externally contact the api using curl command, we have to specify a username and a password :

```
curl -v -k https://manster-node-ip:6443/api/v1/pods -u “user1:password123”
```

°Bearer Token file:

Similarly, to a static user file, we can have a static token file. The file will be the same, but instead of a password, we will pass a token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9,user1,u0001,group1

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmF,user2,u0002,group2

…

Same than for the userfile, you will have to add that line on kube-apiserver.service file or using kubeadm in the /etc/kubernetes/manifests/kube-apiserver.yaml Pod definition file command section:

```
--token-auth-file=user-details.csv
```

Now to contact the api:

```
curl -v -k https://manster-node-ip:6443/api/v1/pods --header “Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9”
```

NOTE: THOSE AUTH MECHANISMS ARE NOT RECOMMENDED AS THEY STORE SENSITIVE INFORMATION IN CLEAR TEXT, IT IS INSECURE (BUT WORTH KNOWING THAT THEY EXISTS).

Article on Setting up Basic Authentication

Setup basic authentication on Kubernetes (Deprecated in 1.19)

Note: This is not recommended in a production environment. This is only for learning purposes. Also note that this approach is deprecated in Kubernetes version 1.19 and is no longer available in later releases

Follow the below instructions to configure basic authentication in a kubeadm setup.

Create a file with user details locally at `/tmp/users/user-details.csv`

```
# User File Contents
password123,user1,u0001
password123,user2,u0002
password123,user3,u0003
password123,user4,u0004
password123,user5,u0005
```

Edit the kube-apiserver static pod configured by kubeadm to pass in the user details. The file is located at `/etc/kubernetes/manifests/kube-apiserver.yaml`

```
apiVersion: v1
kind: Pod
metadata:
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-apiserver
      <content-hidden>
    image: k8s.gcr.io/kube-apiserver-amd64:v1.11.3
    name: kube-apiserver
    volumeMounts:
    - mountPath: /tmp/users
      name: usr-details
      readOnly: true
  volumes:
  - hostPath:
      path: /tmp/users
      type: DirectoryOrCreate
    name: usr-details
```

Modify the kube-apiserver startup options to include the basic-auth file

```
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  name: kube-apiserver
  namespace: kube-system
spec:
  containers:
  - command:
```

```
    - kube-apiserver
    - --authorization-mode=Node,RBAC
      <content-hidden>
    - --basic-auth-file=/tmp/users/user-details.csv
```

Create the necessary roles and role bindings for these users:

```
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]

---
# This role binding allows "jane" to read pods in the "default" namespace.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: user1 # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to bind to
  apiGroup: rbac.authorization.k8s.io
```

Once created, you may authenticate into the kube-api server using the users credentials

```
curl -v -k https://localhost:6443/api/v1/pods -u "user1:password123"
```

Certificates and KubeConfig

NOTE : How to generate certificates for different Kubernetes components and for a user and use them in the Kubernetes cluster is not in the scope of the official CKAD exam. These are part of the official CKA exam.

You can create certificates for your users (out of the scope) and then contact the api this way:

```
curl -v -k https://localhost:6443/api/v1/pods \
--key admin.key \
```

```
--cert admin.crt \
--cacert ca.crt
```

You can also do that using the kubectl utility:

```
kubectl get pods \
--server my-kube-playground:6443 \
--client-key admin.key \
--client-certificate admin.crt \
--certificate-authority ca.crt
```

Obviously, tipping all those server infos and certificates is a repetitive task, so you can put all those certificates information in a kubeconfig file and then specify this file as a kubeconfig option in your command, ex file $HOME/.kube/config (.kube is the default directory for your kube config files).

Then:

```
kubectl get pods --kubeconfig config
```

The kubeconfig file has a specific format ex:

```
apiVersion: v1
kind: Config
current-context: dev-user@development
clusters:
- name: production
  cluster:
    certificate-authority: /etc/kubernetes/pki/ca.crt
    proxy-url: http://proxy.example.org:3128
    server: https://my-kube-playground:6443
- name: development
  cluster:
    …

users:
- name: admin
  user:
    client-certificate: /etc/kubernetes/pki/users/admin.crt
    client-key: /etc/kubernetes/pki/users/admin.key
- name: dev-user
  user:
    client-certificate: /etc/kubernetes/pki/users/dev.crt
    client-key: /etc/kubernetes/pki/users/dev.key

contexts:
- name: admin@production
  context:
    cluster: production
    user: admin
```

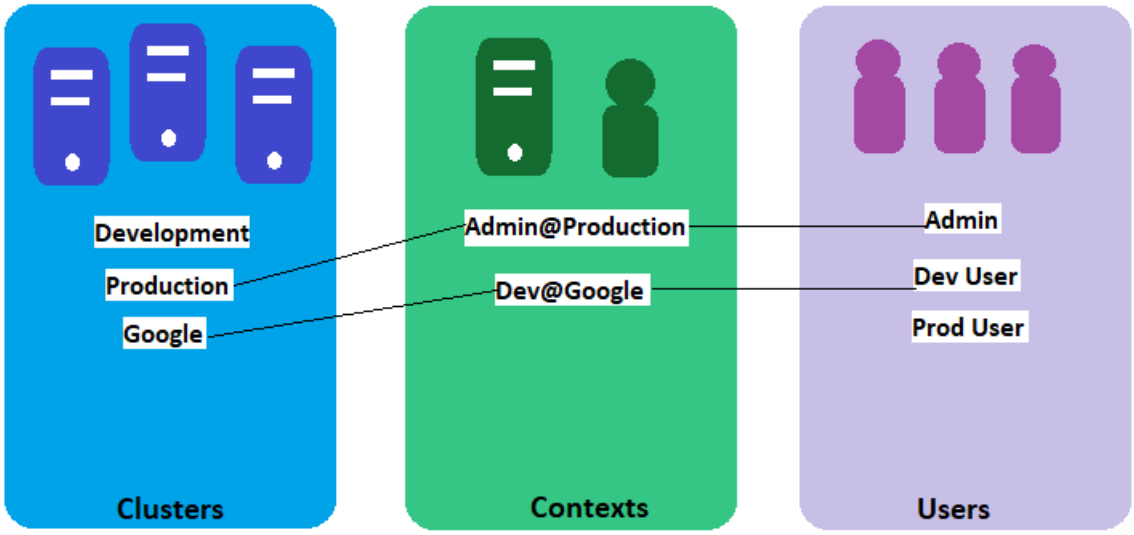
```
- name: dev-user@development
  context:
    cluster: development
    user: dev-user
    namespace: development

…
```

There are three sections => cluster, users and contexts.

°The clusters part will describe your different kubernetes clusters that you need access to, with different information including the server url, the server certificates, the proxy if needed…

°The users part will describe the different users and the user-certificates and user-keys needed to access the different clusters.

°The contexts part will make the bridge between the users and the clusters and allow the communication between them (if the link is not done they won't be able to communicate), **you can also precise the allowed namespace for the user**.



But when we use a kubectl command, how does it know which context to use?

Well, we can specify the default context to use for a user in the kubeconfig file with the "current-user" field.

On our example, kubectl will always use the context dev-user@development to access the development cluster using the dev-user credentials.

See below how to use other contexts.

You can have several config files, by default the $HOME/.kube/config is used.

There are commands to view and edit config files:

°View current file being used:

```
kubectl config view
```

°Use a specific context (other than the default one):

```
kubectl config use-context admin@production
```

°Use a specific context in another config file than the default config file:

```
kubectl config use-context admin@production --kubeconfig /root/my-other-kube-config
```

API groups

All the operations made on the cluster interact with the kube-apiserver either through kubectl or directly via REST.

We can access the API server at the master node address followed by the API port (by default 6443).

Ex get the cluster version and get pods:

```
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
          }
        ]
```

Several api groups are available to interact with the cluster:

/metrics /healthz /version /api /apis /logs

°/version: version of the cluster

°/metrics and /healthz: used to monitor the health of the cluster

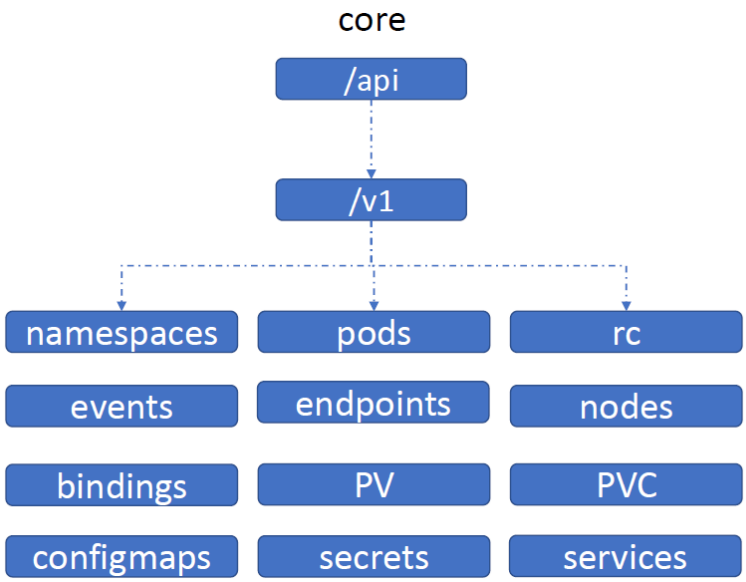
°/logs: used for integrating with third party logging applications

°/api and /apis: responsible for cluster functionality

/api is the core group and /apis is the named group.

Core group

The core group regroups the core functionalities (v1) of the cluster:

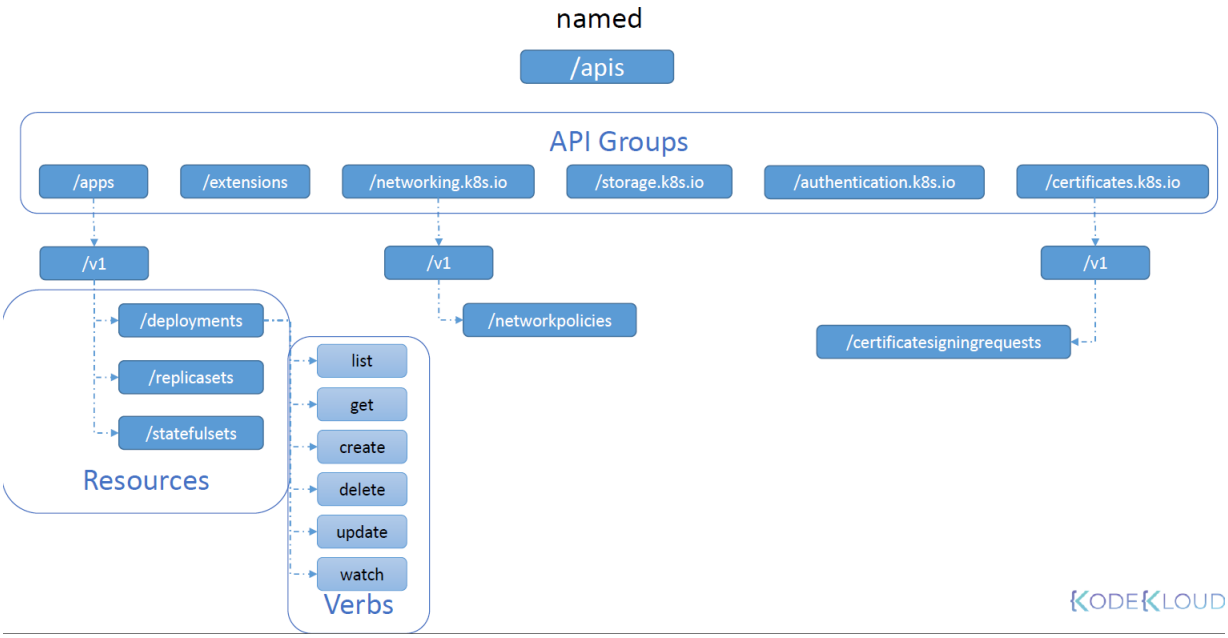


Named group

The named group APIs are more organized, and going forward, all the newer features are going to be made available through these named groups.

All the resources which are not strictly from the v1 API are available here (remember the apiVersion in the definition files).

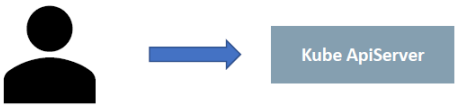
Each resource has a set of actions to manipulate them (list, get, create, update….).



Accessing the REST APIs

If you try to access the kube-apiserver, your requests will be denied excepts for certain APIs like version as you haven't specify any authentication mechanisms.

So you have to use your certificates to access the API :




```
curl http://localhost:6443 -k
```

```
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {

  },
  "status": "Failure",
  "message": "forbidden: User \"system:anonymous\" cannot get path \"/\"",
  "reason": "Forbidden",
  "details": {

  },
  "code": 403
}
```

```
curl http://localhost:6443 -k
      --key admin.key
      --cert admin.crt
      --cacert ca.crt
```

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
    "/metrics"
```

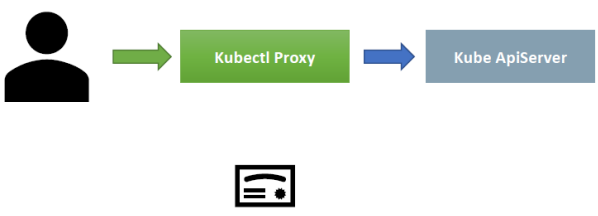
An alternate solution is to start a kubectl proxy client.

The kubectl proxy command launches a proxy service locally on port 8001 and uses credentials and certificates from your kube config file to access the cluster.

That way you won't have to specify the certificates and keys etc in the curl command.

You can then just pass by the kubectl proxy and you requests will be forwarded to the kube-apiServer using the kube config file (instead of the port 6443 of the api, you will use the port of the kubectl proxy).

kubectl proxy




```
kubectl proxy
```

```
Starting to serve on 127.0.0.1:8001
```

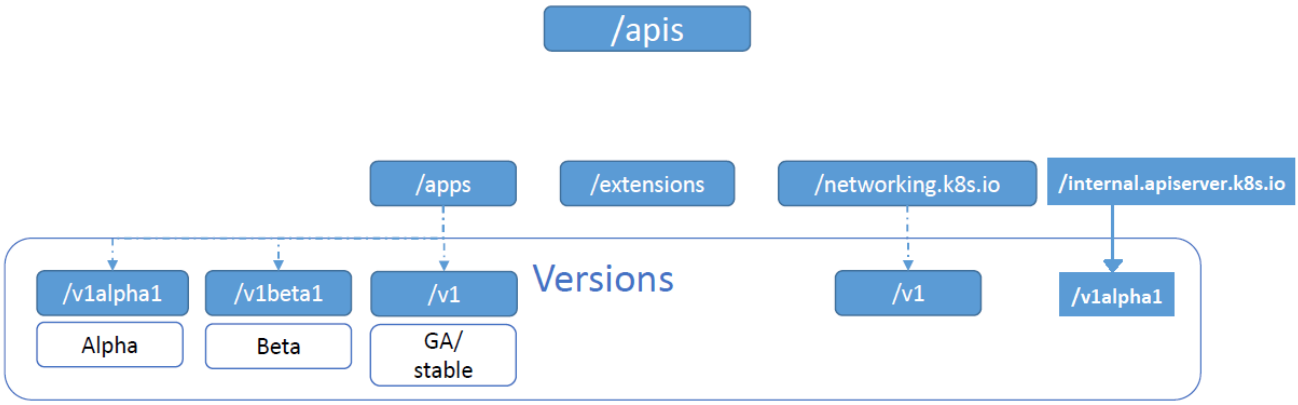
```
curl http://localhost:8001 -k
```

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/",
    "/healthz",
    "/logs",
    "/metrics",
    "/openapi/v2",
    "/swagger-2.0.0.json",
```

API versions

When an api group is at version v1, it means that it is in the GA (generally available) stable version. But the api group may have other version such as /v1alpha1 or /v1beta1.

Alpha is when an api is first developed and merged to the Kubernetes code base and becomes part of the Kubernetes release for the very first time. This api group is not enabled by default, so it means that if you want to create a resource from an alpha api group, you will not be able to do that. For example the StorageVersion resource is only available in the api group interal.apiserver.k8s.io/v1alpha1, which means that you cannot create it by default.



The alpha version are not very reliable and might have bugs. As they might be dropped later, there isn't any support for them. Alpha version might not have end to end tests. In addition, there is no guarantee that this API will be available in the future releases. Experts interested in giving early feedbacks might use those API groups.

Once the version is patched and stable and the end to end tests are ok, the api is available on it's beta version.

The beta api groups are enabled by default, but as it's not GA, they might still have minor bugs. But in beta version, you have the insurance that they will one day reach the GA version and be available in future kub versions. Users interested in beta testing and provisioning feedbacks can use the beta versions.

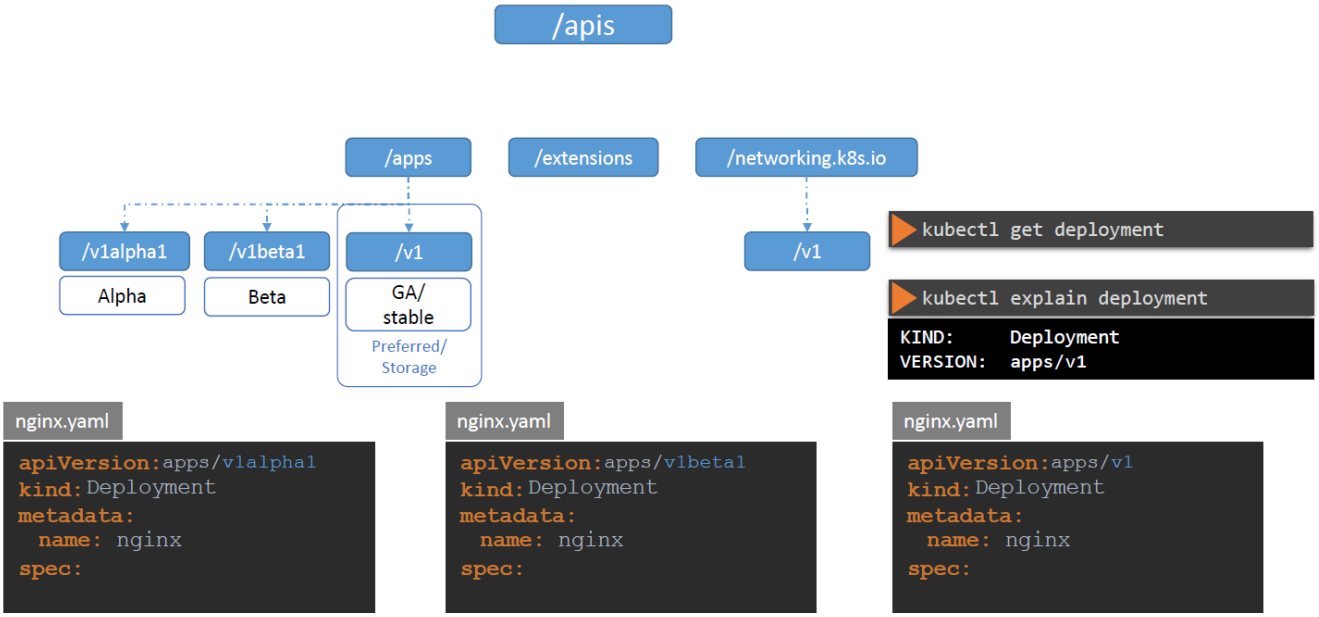
The GA stable version has conformance tests, is highly reliable and will be present in many future releases.

When resources have several API groups, the kubectl command (ex kubectl exec deployment), will use the preferred version of the API to use.

You also have a storage version, precising which version is used to store the object in the etcd database.

Only one version can be the storage version, this means that if any object is created with the API version set to anything other than the storage version, then those will be converted to the storage version, which is v1, before storing them into the etcd database.

Usually the preferred version and the storage version are the same, but that's not always the case.



To get the preferred version, you have to get the information of the api group you want to check using it's url, for example:

For the storage version, there are no commands to check which one is set so you will have to query the etcd database directly ex :

```
ETCDCTL_API=3 etcdctl
  --endpoints=https://[127.0.0.1]:2379
  --cacert=/etc/kubernetes/pki/etcd/ca.crt
  --cert=/etc/kubernetes/pki/etcd/server.crt
  --key=/etc/kubernetes/pki/etcd/server.key
  get "/registry/deployments/default/blue" --print-value-only
```

```
k8s

apps/v1
Deployment


bluedefault"*$cf8dcd55-8819-4be2-85e7-bb71665c2ddf2ZB
successfully progresse8"2
```

To enable or disable a specific version, you must add it to the runtime config parameter of the kube api server service (you can add several separating them by “,”).

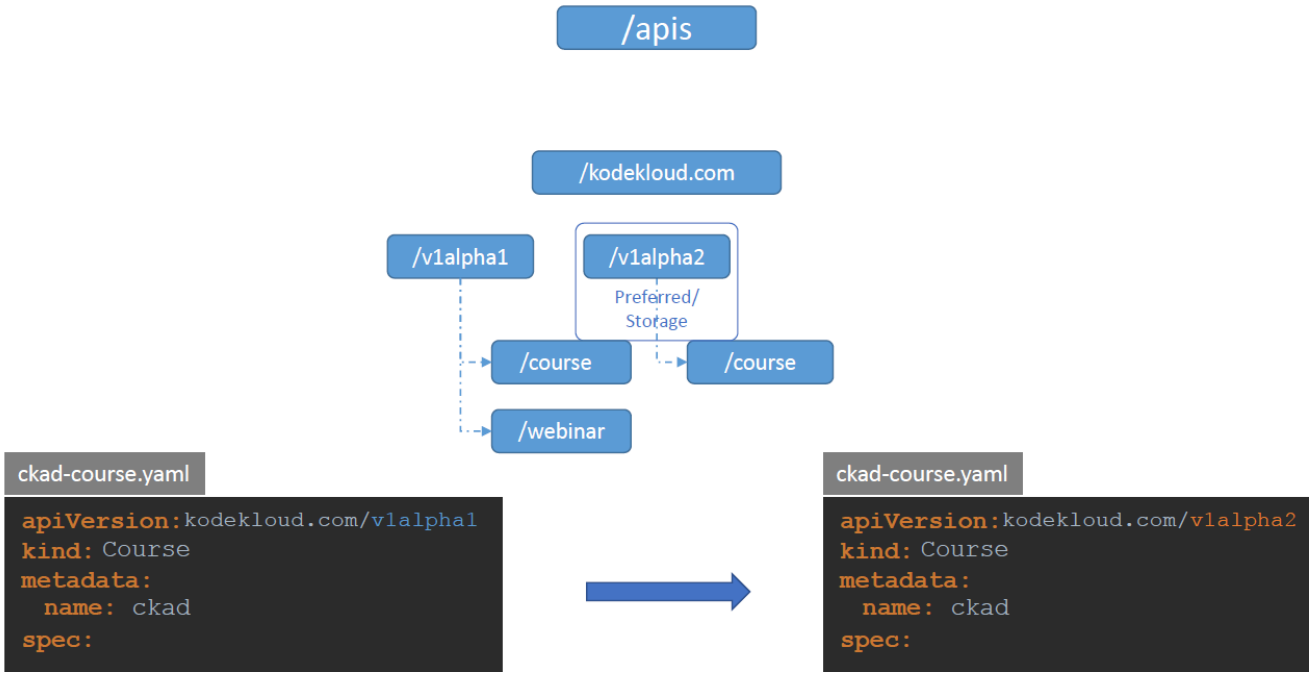
```
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=${INTERNAL_IP} \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/apiserver-etcd-client.crt \\
  --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
```

Then you will have to restart the api server service.

API deprecation

When we creates a new api group (developer team), when we want to remove an element from, for example, the v1alpha1 (which is buggy), we need to increment the version of the

API group if we do that modification. Ex, to remove the /webinar object of the /kodecloud.com api group /v1alpha1 :

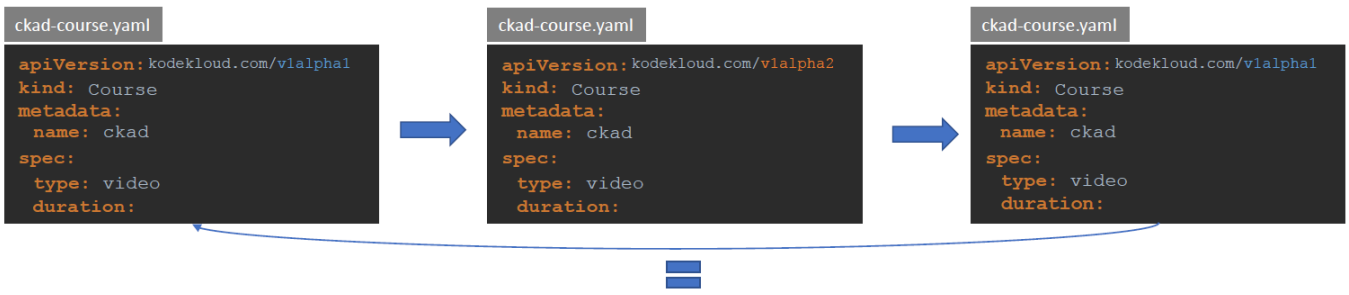


So the /webinar object still exists in /v1alpha1 but not in /v1alpha2.

But what if now the preferred version is v1alpha2? Well the user can still create objects using the v1alpha1 api group, but they will be converted and stored as v1alpha2.

If we add a field to the Course object, for ex duration: , we upgrade our version but that field is not present in v1alpha1. Well we need to add an equivalent field in the v1 version as well, like that we can still convert objects to previous version without information loss.

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.

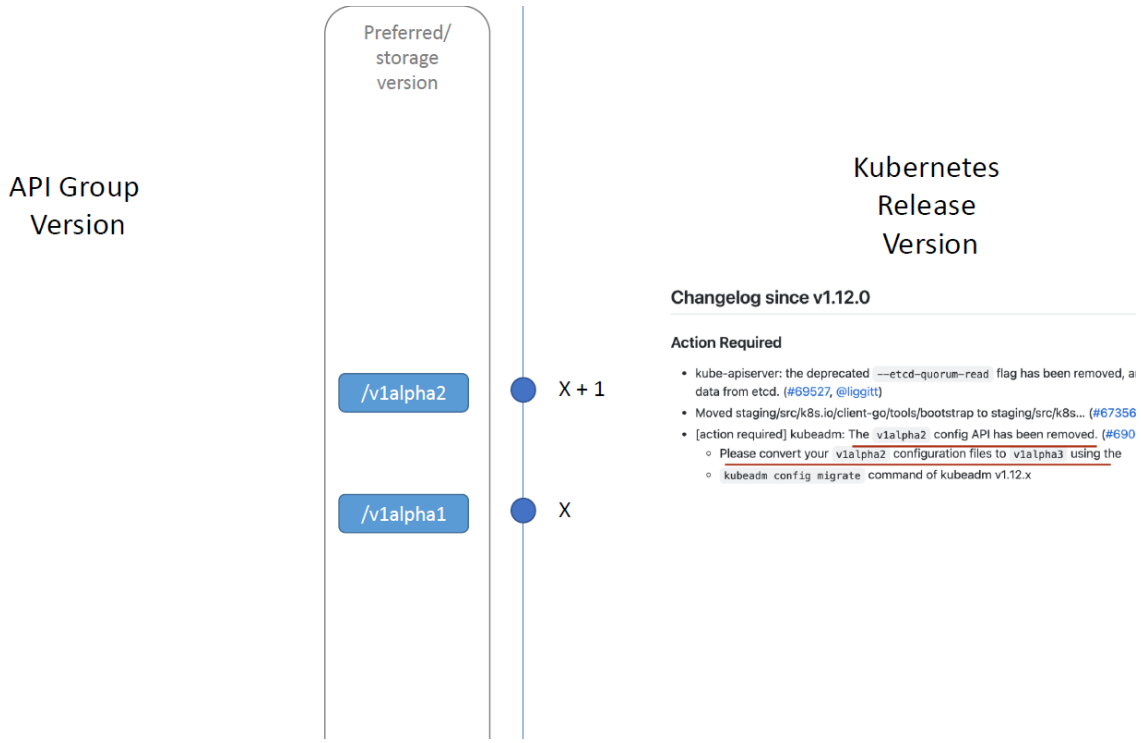


As the time pass, our number of version grows until it reach the GA version v1.

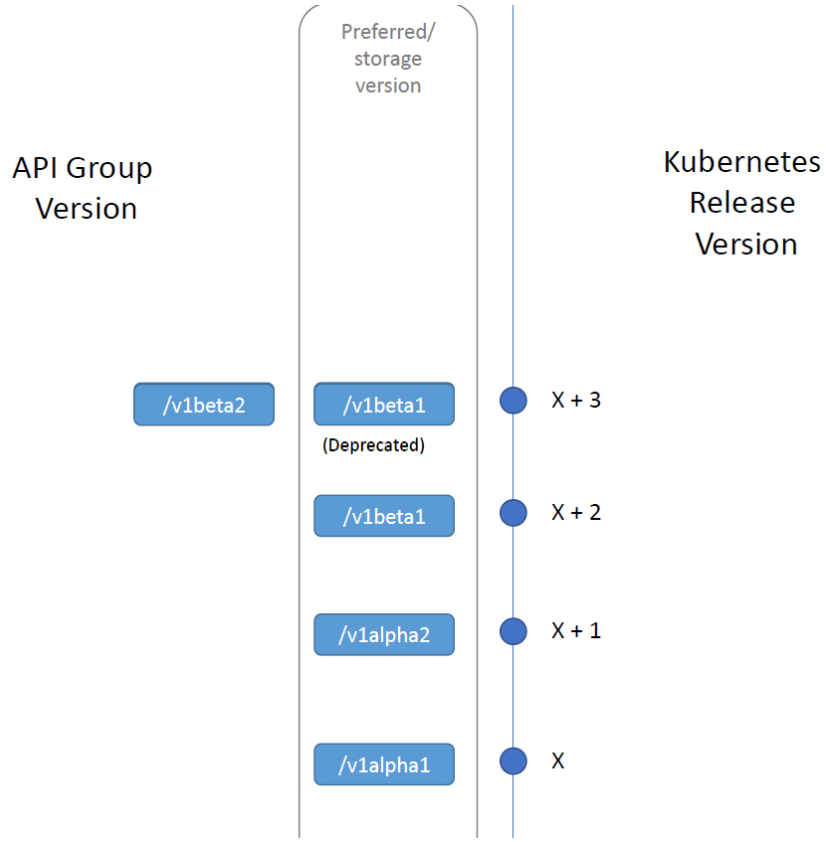
But what do we do about older version, we can't keep them all, so we must deprecate them to avoid having them in future versions.

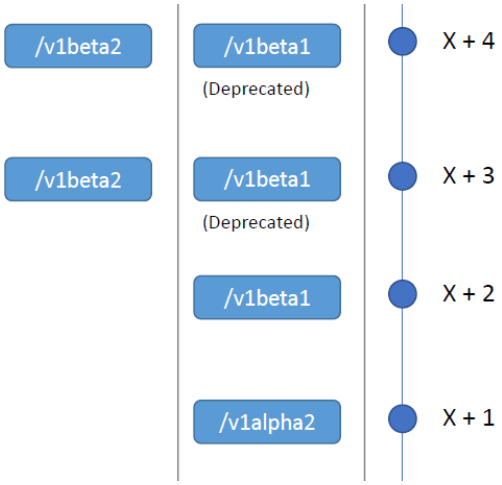
Other than the most recent API versions in each track, older API versions must be supported after their announced deprecation for a duration of no less than:

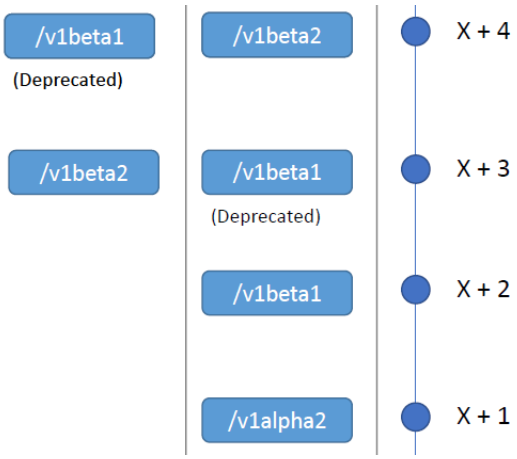
- **GA: 12 months or 3 releases (whichever is longer)**
- **Beta: 9 months or 3 releases (whichever is longer)**
- **Alpha: 0 releases**

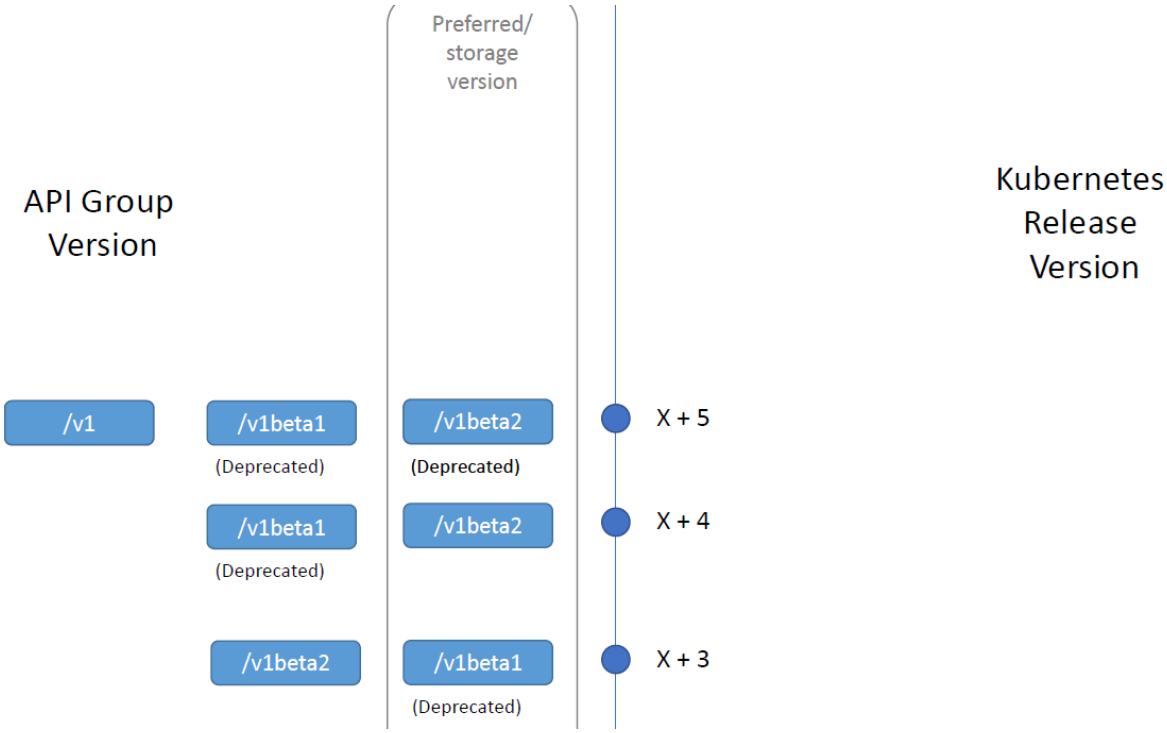


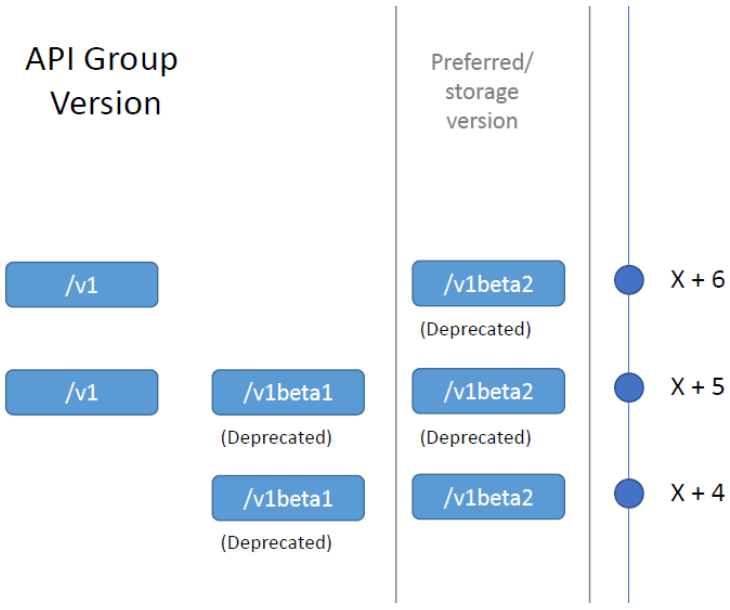
The "preferred" API version and the "storage version" for a given group may not advance until after a release has been made that supports both the new version and the previous version

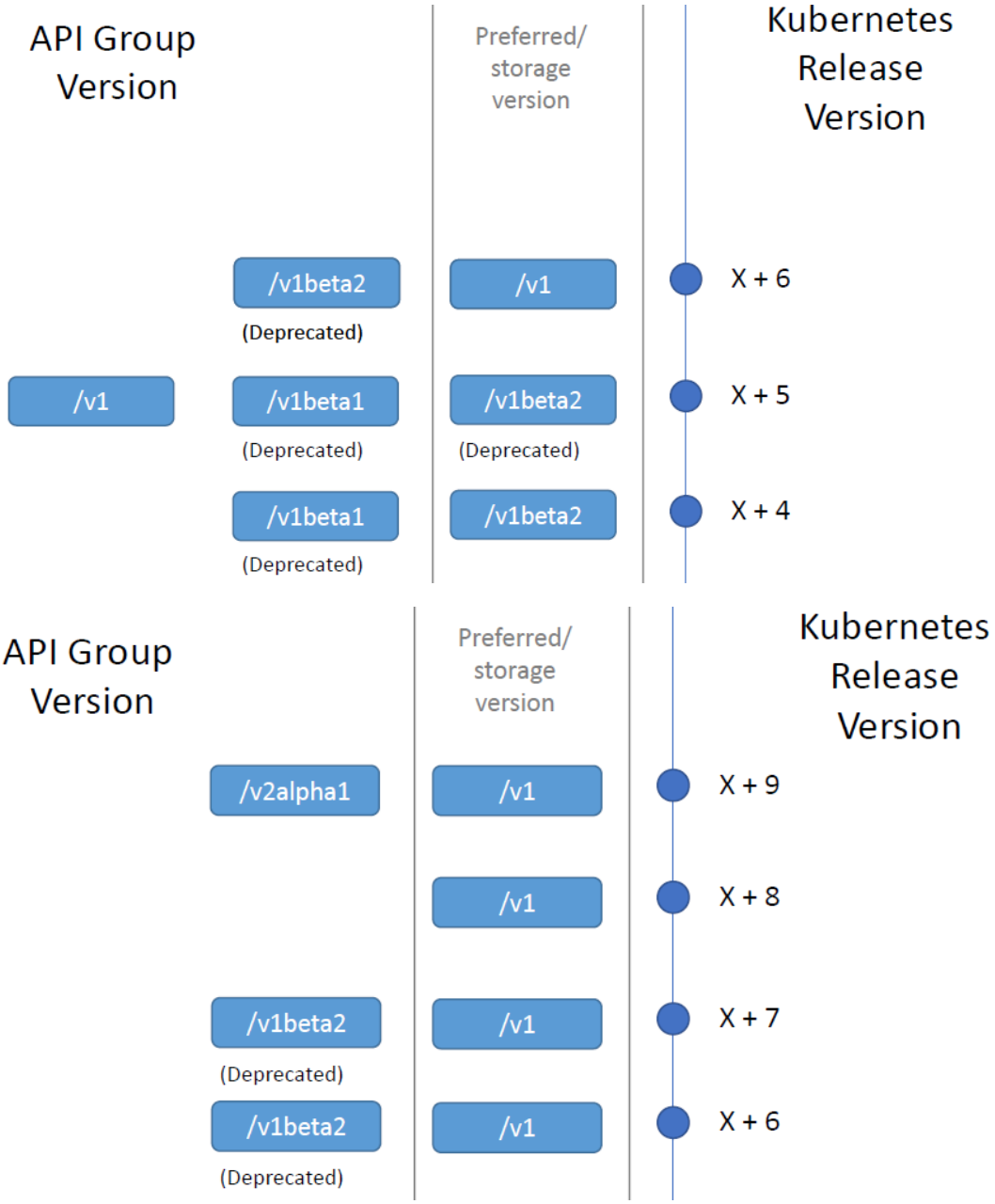

API Group Version
Preferred/ storage version
Kubernetes Release Version
/v1beta2
/v1beta1
(Deprecated)
X + 3
/v1beta1
X + 2
/v1alpha2
X + 1
/v1alpha1
X


/v1beta2
/v1beta1
(Deprecated)
X + 4
/v1beta2
/v1beta1
(Deprecated)
X + 3
/v1beta1
X + 2
/v1alpha2
X + 1

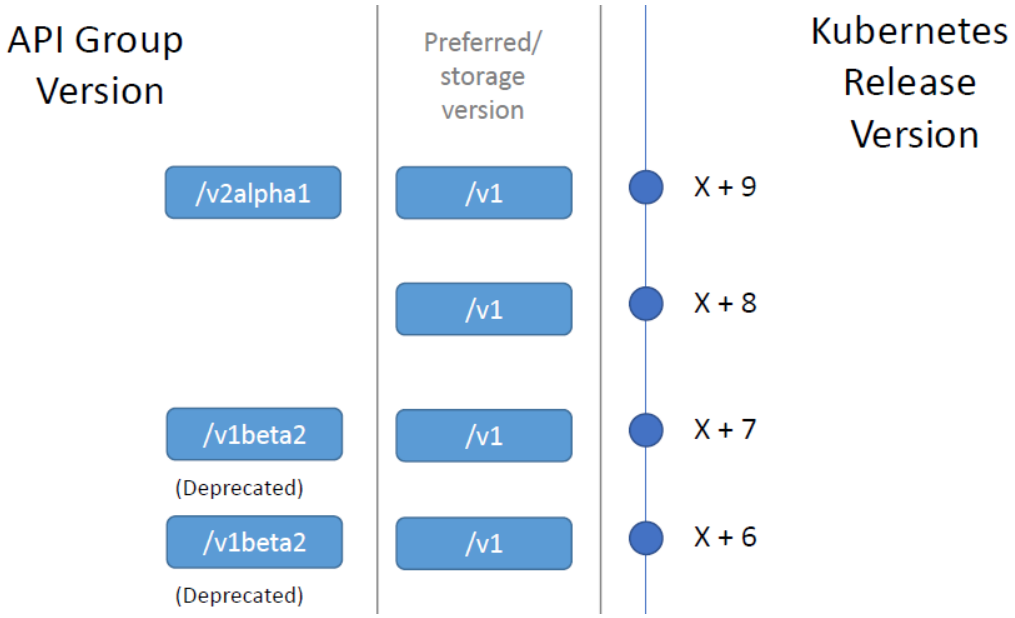

/v1beta1
(Deprecated)
/v1beta2
/v1beta2
/v1beta1
(Deprecated)
/v1beta1
/v1alpha2
X + 4
X + 3
X + 2
X + 1


API Group Version
Preferred/ storage version
Kubernetes Release Version
/v1
/v1beta1
(Deprecated)
/v1beta2
(Deprecated)
/v1beta1
(Deprecated)
/v1beta2
/v1beta2
/v1beta1
(Deprecated)
X + 5
X + 4
X + 3


API Group Version
Preferred/ storage version
Kubernetes Release Version
/v1
/v1
/v1beta1
(Deprecated)
/v1beta1
(Deprecated)
/v1beta2
(Deprecated)
/v1beta2
(Deprecated)
/v1beta2
X + 6
X + 5
X + 4

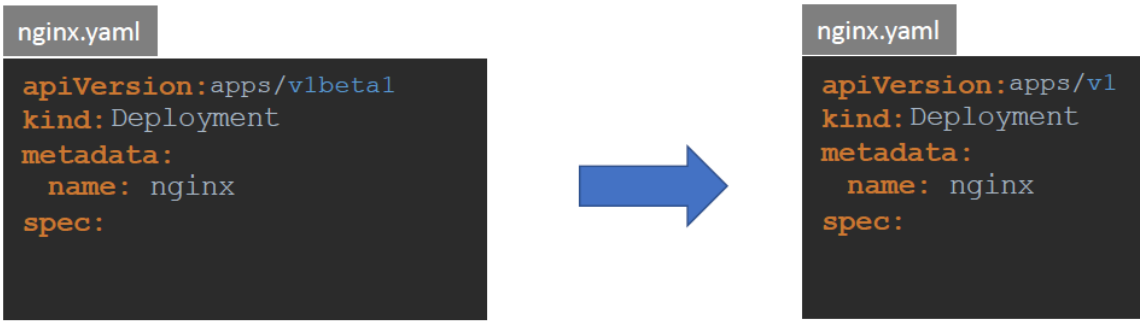


An API version in a given track may not be deprecated until a new API version at least as stable is released.



To update a definition file and change it's api group version, you can use the kubectl convert command. Note that it's a plugin so you might have to install it:

https://kubernetes.io/docs/tasks/tools/install-kubectl-linux/#install-kubectl-convert-plugin




```
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
```

Authorization

Now that we have seen how to allow users or machine to gain access to our cluster, we have to define what they are authorized or restricted to do.

There are several authorization mechanisms:

°Attribute based access control (ABAC)

°Role based access control (RBAC)

°Webhook

° Always allow or always deny

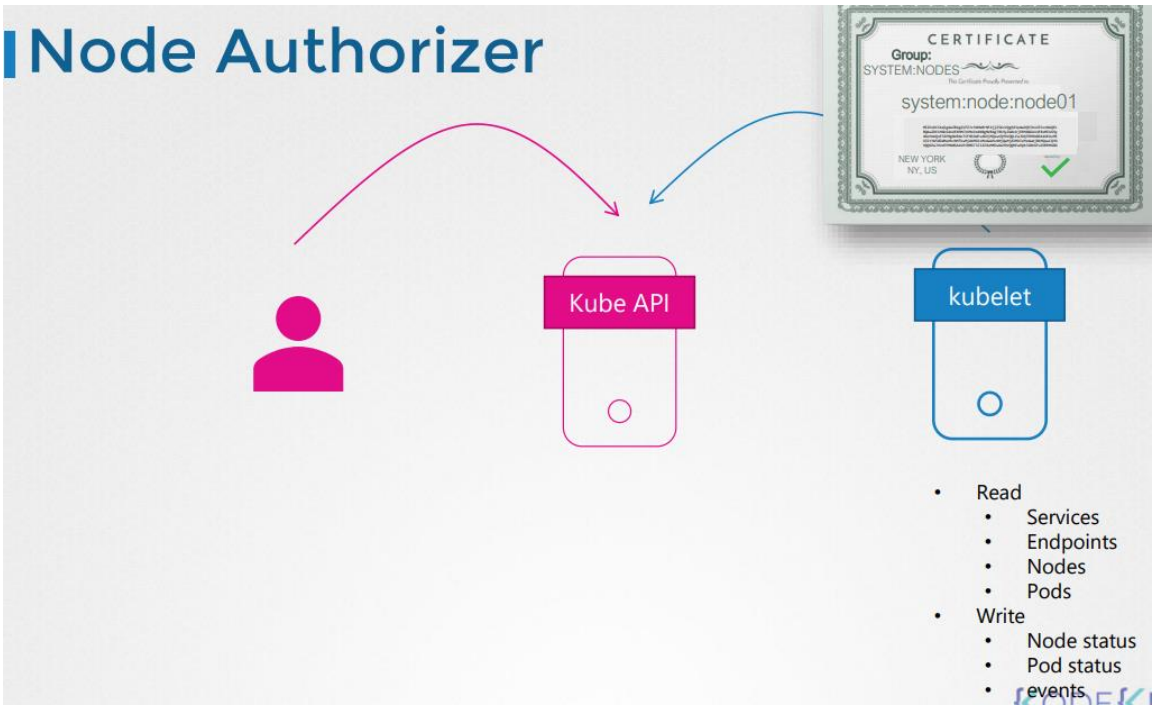
Attribute based authorization (ABAC)

We know that the kube-apiserver is accessed by users and as well by the kubelet on nodes within the cluster for cluster management purposes.

Kubelet accesses the API server to read information about services, endpoints, nodes and pods. Kubelet also reports to the apiServer information about node status, pods status and events.

These requests are handled by a special authorizer known as the node autohorizer.

About certificates : Kubelet should be part of the system nodes group and have a name prefixed with system:node, so any request coming from a user with the name system node and part of the system nodes group is authorized by the node authorizer and are granted these privileges. So that's access within the cluster.



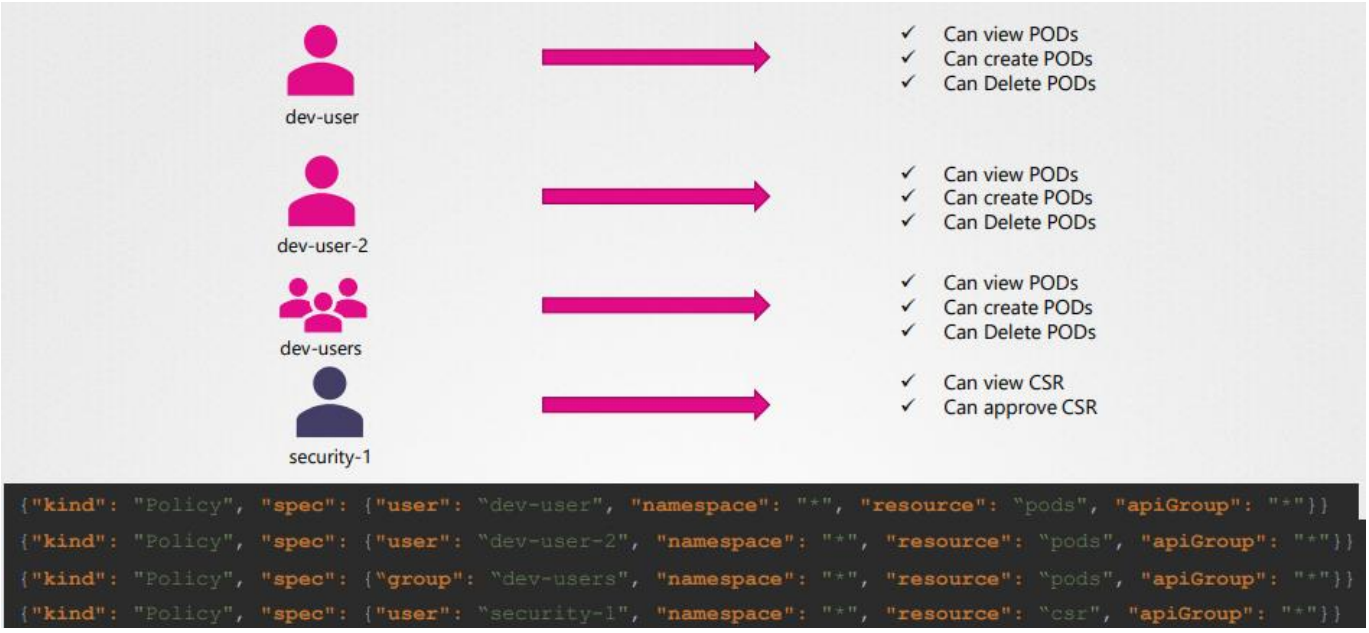
But what about access from outside the cluster, for instance a user or a group of users? That's where the attribute based authorization is used.

You can associate a user or a group of users with a set of permissions (create, delete view edit pods for example).

This can be defined with a set of policies defined in adjacent format (see below).

You have to pass this policies file into the API server.

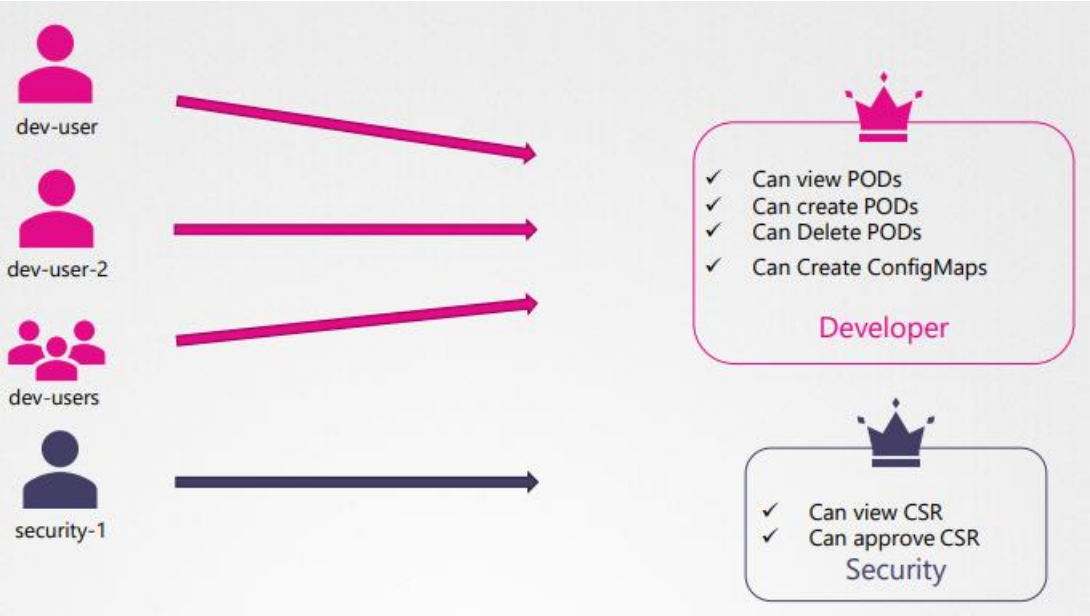
Every time you have to add a new, delete or edit rights of a user or a group of users you will have to modify that policy file manually so it is kind of difficult to manage. This is why there is a role based authorization mechanism, which makes it easier to manage.



Role based access control (RBAC)

Namespaced roles

Roles are created separately and then associated with users or a group of users.



This way the permissions can be modified for several users or group of users at once by modifying the roles.

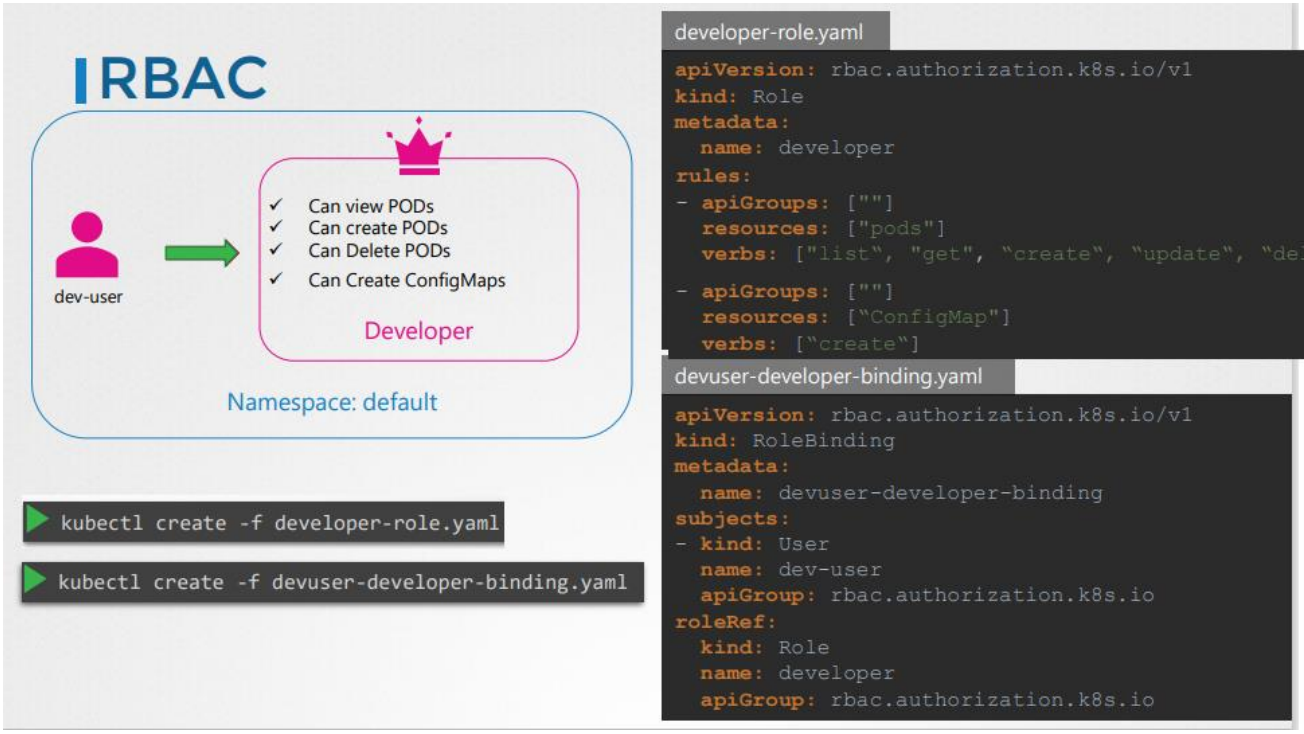
RBAC role and user/group bindings are kube objects which can be created through definition files.

We need to use the api group rbac.authorization.k8s.io/v1.

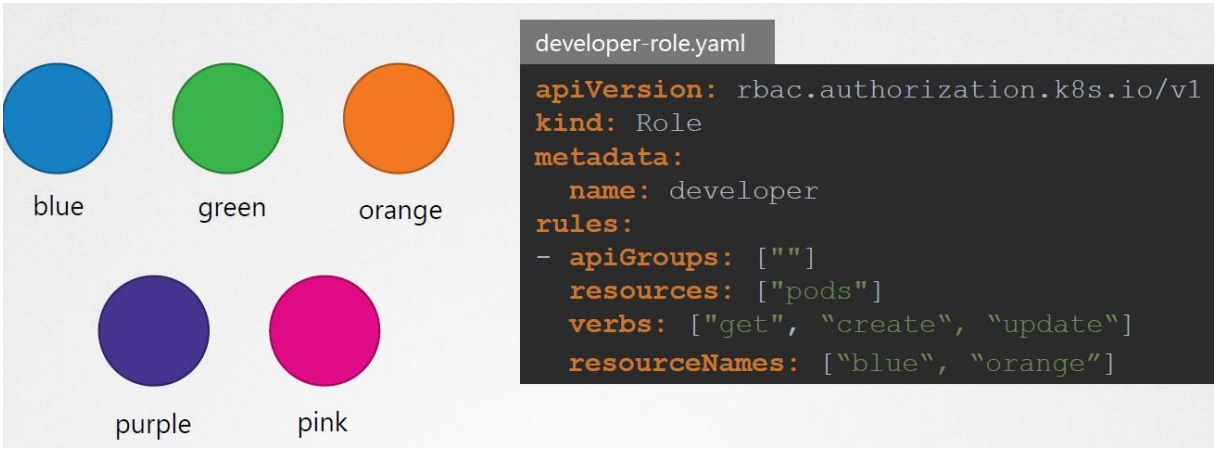
On the role definition file we can precise the different kind of api groups (by default the core group v1) of each resource we want to make accessible and the type of the resource. Then the authorization rules (list, get, create, update, delete) concerning the resource can be defined. You can add multiple rules in one role.

To link one user or one group to that role we use a RoleBinding definition file (same api group). Ther we can precise several subjects to bind, the type of the subject (user, group…), the api group (as it is to bind the user to a role, we use the same api group) and the reference to the role we just created.

Note that role and RoleBinding are limited to the namespace where they are created (default namespace if none), if you want to limit the users access to a specific namespace, you will have to add the namespace information in the metadata section of the definition files.



You can even restrict access to specific resources (like specific pods, deployments etc…) using their resourceName:




```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  creationTimestamp: "2023-02-10T10:16:38Z"
  name: developer
  namespace: blue
  resourceVersion: "577"
  uid: 5d8e3487-c68d-4beb-9dc9-d2c5fa677310
```

```
rules:
- apiGroups:
  - ""
  resourceNames:
  - blue-app
  - dark-blue-app
  resources:
  - pods
  verbs:
  - get
  - watch
  - create
  - delete
- apiGroups:
  - "apps"
  resources:
  - deployments
  verbs:
  - create
```

```
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]
```

```
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
```

How to check if you can perform some actions as a user? And as an administrator can I impersonate a user to check if the user has the rights? (--as user)

```
kubectl auth can-i create deployments
yes
```

```
kubectl auth can-i delete nodes
no
```

```
kubectl auth can-i create deployments --as dev-user
no
```

```
kubectl auth can-i create pods --as dev-user
yes
```

```
kubectl auth can-i create pods --as dev-user --namespace test
no
```

Cluster Roles

We saw with RBAC that we can specify authorizations based on resources created in namespaces.

But what if we want to create authorizations to access resources based on the different nodes in our cluster?

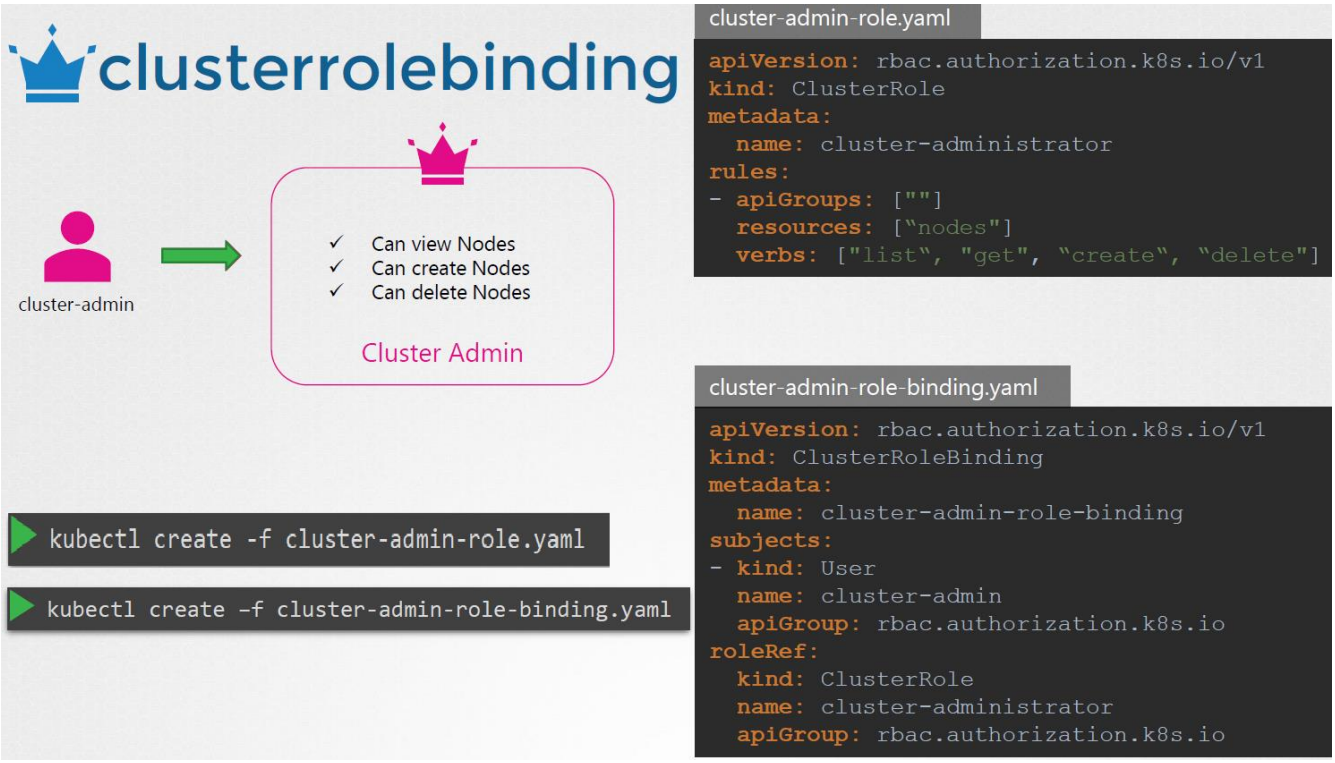
Nodes can't be associated with particular namespaces so the resources can be cluster scope.

And other resources like PVs and PVCs and, in fact, namespaces are not created specifying a namespace.

Here is a non exhaustive list of the resources classified by their scope : namespace or cluster.

So how do we authorize users to use cluster scope resources like nodes or persistent volumes?

In fact it is very similar than the RBAC role and role binding procedure for the definition files: you use ClusterRole and CluesterRoleBinding



Important note : you can create CluesterRoleBinding for the namespaced resources as well, it will authorize the user to access the resources across all namespaces.

Admission Controller

We can go further than authorizing users to use resources, we can also limit other options, for example :

°Allow user to create Pods on a namespace but using only images from a specific internal registry.

°Enforce that we are not allowed to use the :latest tag for any images.

°Never allow the containers to run as user root.

°Only permit some capabilities.

°Always label Pods.

You can't achieve all those specific security measures using the existing RBAC.

And that is where Admission Controllers comes in.

There are a number of admission controllers that come pre build with Kubernetes such as :

°AlwaysPullImages: ensures that every time a Pod is created, the images are always pulled.

°DefaultStorageClass: observes the creation of PVCs and automatically adds a default storage class to them if one is not specified.

°EventRateLimit: can be used to put a limit to the number of requests authorized to the API server (prevents the API to be flooded with requests).

°NamespaceExists: rejects requests to namespaces that does not exists.

°And many more…

Ex NamespaceExists:

If I do a request to an un existing namespace:

```
kubectl run nginx --image nginx --namespace blue
```

I'll get the return:

```
Error from server (NotFound): namespaces "blue" not found
```

What's happening here is that my request gets authenticated, then authorized and then gets through the AdmissionControllers. The NamespaceExists admission controller handles the request and check if the blue namespace exists. If it doesn't exists, the request is rejected.

Note: the NamespaceExists admission controller is deprecated, the new one is called NamespaceLifecycle admission controller.

Another AdmissionController which is not enable by default is the NamespaceAutoProvision controller. This will automatically create the namespace if it does not exists.

To see the admission controllers enabled by default we can use the command:

```
kube-apiserver -h | grep enable-admission-plugins
```

Or if you are running the command in a kubeadm based setup:

```
kubectl exec kube-apiserver-controlplane -n kube-system -- kube-apiserver -h | grep enable-admission-plugins
```

To enable admission controllers, you have to edit the kube-apiserver.service file or the Pod definition file (kubeadm based setup) and add in the container command section that line:

```
- --enable-admission-plugins=NodeRestriction,NamespaceAutoProvision…
```

Similarly to disable admission controllers use the - --disable-admission-plugins flag:

```
- --disable-admission-plugins=DefaulStorageClass
```

There are different kinds of Admission Controllers:

°Some are validating admission controllers, like the NamespaceLifecycle controller (they validate the requests).

°Others can make operations on your behalf and are called mutating validation controllers.

For example, the DefaultStorageClass will check if a storage class is associated to your PVC and if not, will add the storageClassName: default option on you PVC object (which will allow

the creation of a default storage class on your cluster), even if you didn't specified that option. The mutating admission controllers can change or mutate the objects themselves before they are created.

°Some admission controller can both validate requests and mutate objects.

Usually the requests will first go through the mutating admission controllers and then to the validating admission controllers, this way the mutations operated by the mutation admission controllers can be validated (or rejected) as well.

What if we want our own admission controller? For that, two special admission controller are available:

°MutatingAdmission Webhook

°ValidatingAdmission Webhook

We can configure those webhooks to point to an Admission Webhook server inside or outside the Kub cluster. This server will have our own Admission Webhook service running with our own code and logic.

Once the requests have past all the built in (kub native) Admission Controllers of the Kub cluster, it hits the configured Admission Webhoooks.

Once the request hits the webhook, it makes a call to the admission webhook server by passing in an admission review object in a json format. This object has all the details about the request such as the user that makes the request and the type of operation the user is trying to perform on what kubernetes object and the details about the object itself.

This example shows the data contained in an `AdmissionReview` object for a request to update the `scale` subresource of an `apps/v1` `Deployment (the request itself will have a json format)`:

```yaml
apiVersion: admission.k8s.io/v1
kind: AdmissionReview
request:
  # Random uid uniquely identifying this admission call
  uid: 705ab4f5-6393-11e8-b7cc-42010a800002

  # Fully-qualified group/version/kind of the incoming object
  kind:
    group: autoscaling
    version: v1
    kind: Scale

  # Fully-qualified group/version/kind of the resource being modified
  resource:
```

```
    group: apps
    version: v1
    resource: deployments

  # subresource, if the request is to a subresource
  subResource: scale

  # Fully-qualified group/version/kind of the incoming object in the original
request to the API server.
  # This only differs from `kind` if the webhook specified `matchPolicy:
Equivalent` and the
  # original request to the API server was converted to a version the webhook
registered for.
  requestKind:
    group: autoscaling
    version: v1
    kind: Scale

  # Fully-qualified group/version/kind of the resource being modified in the
original request to the API server.
  # This only differs from `resource` if the webhook specified `matchPolicy:
Equivalent` and the
  # original request to the API server was converted to a version the webhook
registered for.
  requestResource:
    group: apps
    version: v1
    resource: deployments

  # subresource, if the request is to a subresource
  # This only differs from `subResource` if the webhook specified
`matchPolicy: Equivalent` and the
  # original request to the API server was converted to a version the webhook
registered for.
  requestSubResource: scale

  # Name of the resource being modified
  name: my-deployment

  # Namespace of the resource being modified, if the resource is namespaced
(or is a Namespace object)
  namespace: my-namespace

  # operation can be CREATE, UPDATE, DELETE, or CONNECT
  operation: UPDATE

  userInfo:
    # Username of the authenticated user making the request to the API server
    username: admin

    # UID of the authenticated user making the request to the API server
    uid: 014fbff9a07c

    # Group memberships of the authenticated user making the request to the
API server
    groups:
```

```
      - system:authenticated
      - my-admin-group
    # Arbitrary extra info associated with the user making the request to the
API server.
    # This is populated by the API server authentication layer and should be
included
    # if any SubjectAccessReview checks are performed by the webhook.
    extra:
      some-key:
        - some-value1
        - some-value2

  # object is the new object being admitted.
  # It is null for DELETE operations.
  object:
    apiVersion: autoscaling/v1
    kind: Scale

  # oldObject is the existing object.
  # It is null for CREATE and CONNECT operations.
  oldObject:
    apiVersion: autoscaling/v1
    kind: Scale

  # options contains the options for the operation being admitted, like
meta.k8s.io/v1 CreateOptions, UpdateOptions, or DeleteOptions.
  # It is null for CONNECT operations.
  options:
    apiVersion: meta.k8s.io/v1
    kind: UpdateOptions

  # dryRun indicates the API request is running in dry run mode and will not
be persisted.
  # Webhooks with side effects should avoid actuating those side effects when
dryRun is true.
  # See http://k8s.io/docs/reference/using-api/api-concepts/#make-a-dry-run-
request for more details.
  dryRun: False
```

On receiving the request, the admission webhook server responds with an admission review object with a result indicating whether the request is allowed or not. If the allowed field in the response is set to true, then the request is allowed.

Example of a minimal response from a webhook to allow a request:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true
  }
}
```

To set this up, we first have to deploy our admission webhook server, and then we can configure the webhook using an webhook configuration object.

The webhook server can be a simple API server built on any kind of platform, language etc… it only must accept the json format send from the webhook and send the response in the right json format.

Example of a server written in go:

https://github.com/kubernetes/kubernetes/blob/v1.13.0/test/images/webhook/main.go

Ex of a python server with a simple validation route rejecting requests of users creating resources with their own name in the metdata section of the resource:

```
@app.route("/validate", methods=["POST"])
def validate():
 status = true
 message = ""
 object_name = request.json["request"]["object"]["metadata"]["name"]
 user_name = request.json["request"]["userInfo"]["name"]
 if object_name == user_name
  message = "You can't create resources with your own name"
  status = false
 return jsonify(
  {
   "response": {
    "allowed": status,
    "uid": request.json["request"]["uid"]
    "status": {"message": message},
   }
  }
 )
```

For a mutating route you have to precise the kind of mutations you want to do to the object in your response, here for example to add a user name label in the metadata: labels: part of the object.

You can precise the action (op for operation) add, delete, update to change an option in the configuration of the resource, where you want to do the change (path) and the value of the modification (value), those infos have to be specified on the patch key of the json return encoded in base 64:

```
@app.route(“/mutate”, methods=[“POST”])
def mutate():
 user_name = request.json[“request”][“userInfo”][“name”]
 patch = [{“op”:”add”, “path”: “/metadata/labels/users”, “value”: user_name}]
 return jsonify(
  {
   “response”: {
    “allowed”: True,
    “uid”: request.json[“request”][“uid”]
    “patch”: base64.b64encode(patch),
    “patchtype”: “JSONPatch”,
   }
  }
 )
```

°Then you will have to configure the admission webhook on your kubernetes cluster.

The admission webhook is a kub resource:

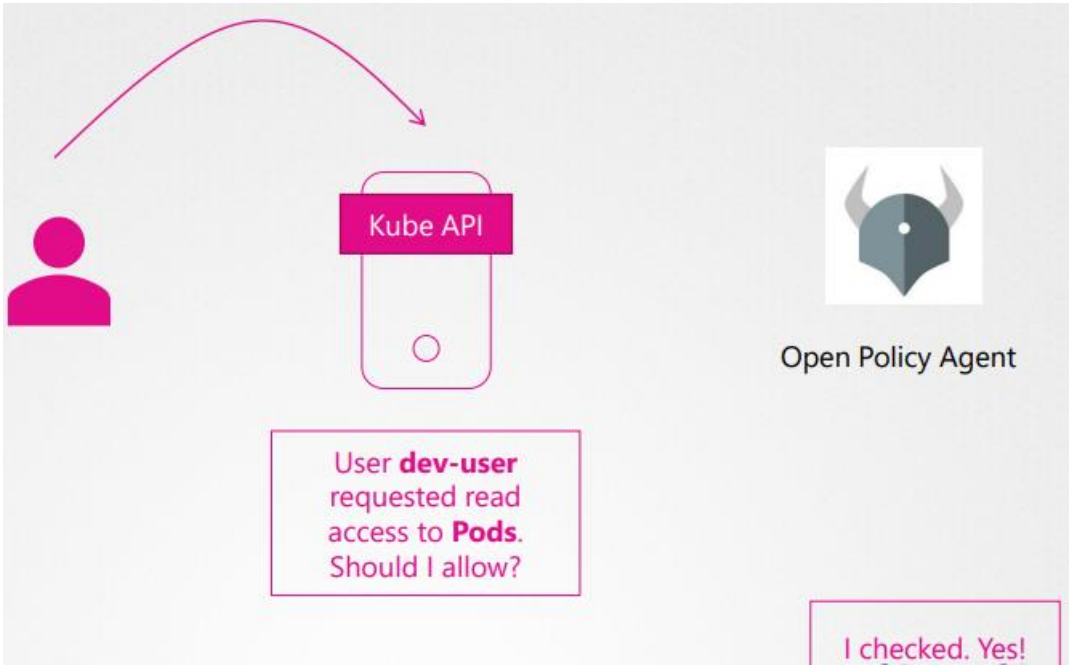
The following is an example `ValidatingWebhookConfiguration`, a mutating webhook configuration is similar. See the webhook configuration section for details about each config field.

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  #You have to precise the rules to trigger the webhook, here it’s on Pod
creation (precise the api group used, the version, the resource and the scope
```

```
  rules:
  - apiGroups:   [""]
    apiVersions: ["v1"]
    operations:  ["CREATE"]
    resources:   ["pods"]
    scope:       "Namespaced"
  clientConfig:
    #If it's an external server use the url option at that level
    #url: https://external-server.example.com
    #If it's an internal api server just use the service configured to reach
it
    service:
      namespace: "example-namespace"
      name: "example-service"
    #Certificate bundle because it's TLS communication
    caBundle: <CA_BUNDLE>
  admissionReviewVersions: ["v1"]
  sideEffects: None
  timeoutSeconds: 5
```

Webhook

You can use third party tools to externalize the authorizations outside the cluster. When the user will make a request, the kube-apiServer will ask to that third party tool if the user is authorized or not to access the requested resource.



Always allow or always deny

You can configure the kube-apiServer to allow or deny all requests without performing any authorization checks.

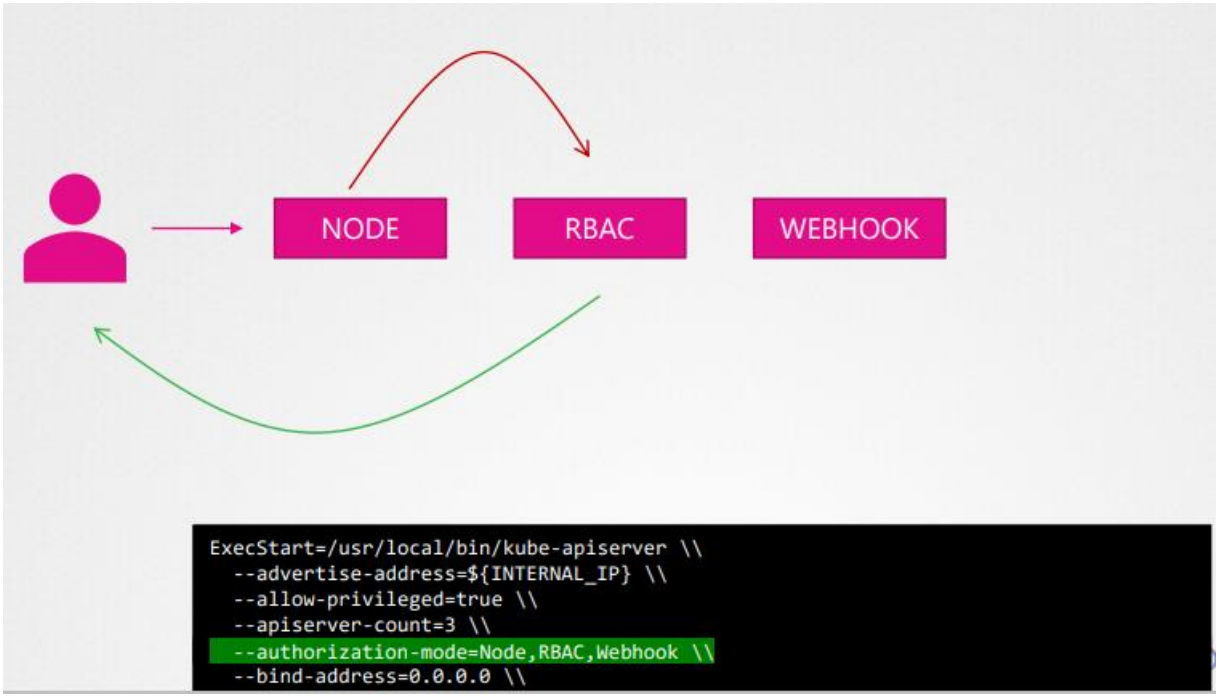
Configure an authorization mode on the kube-apiServer

You can configure one of the different kind of authorization mode we saw on the kube-apiServer pod definition file with the option --authorization.

You can also use the command:

```
ps -aux | grep authorization
```

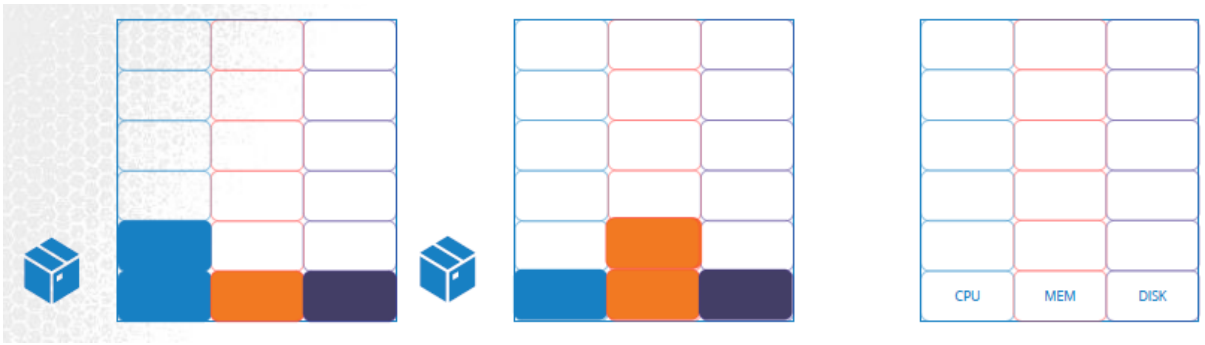
If you have multiple modes configured, your request will try to pass each authorization mode, in the order they are defined, until one of the mode authorize the request. If none of the authorization mode allows the request to pass, the request is denied.



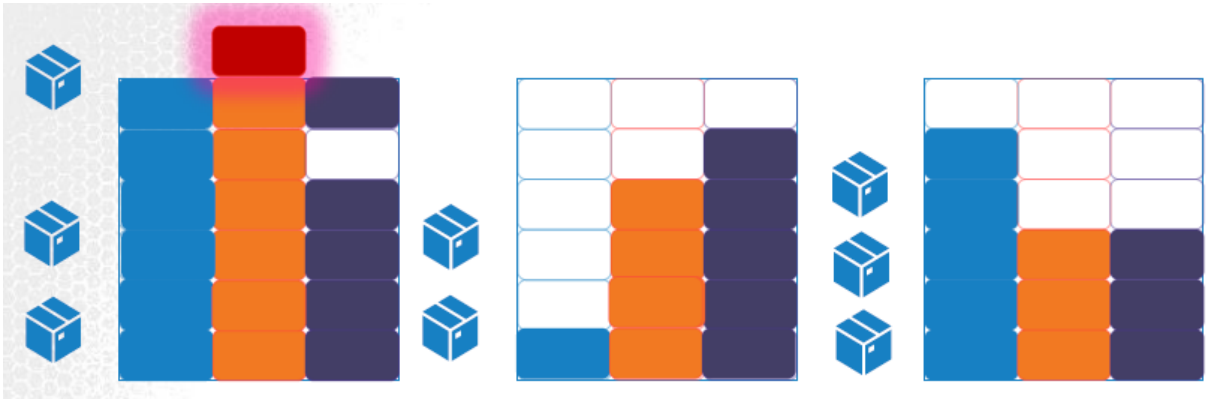
Resources requirement

Let us look at a 3 Node Kubernetes cluster. Each node has a set of CPU, Memory and Disk resources available. Every POD consumes a set of resources. In this case 2 CPUs , one Memory and some disk space. Whenever a POD is placed on a Node, it consumes resources available to that node.

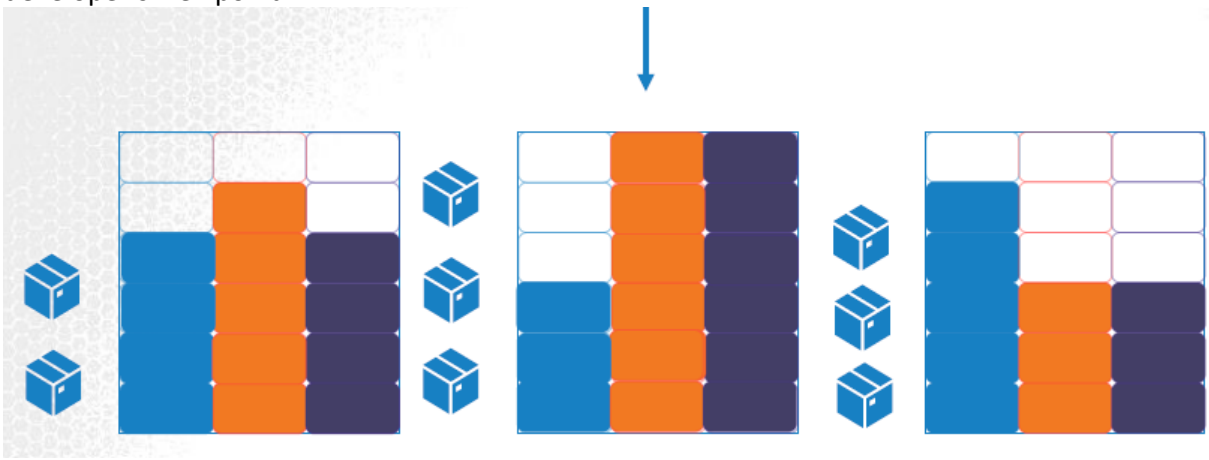
As we have discussed before, it is the kubernetes scheduler that decides which Node a POD goes to. The scheduler takes into consideration, the amount of resources required by a POD and those available on the Nodes. In this case, the scheduler schedules a new POD on Node 2.



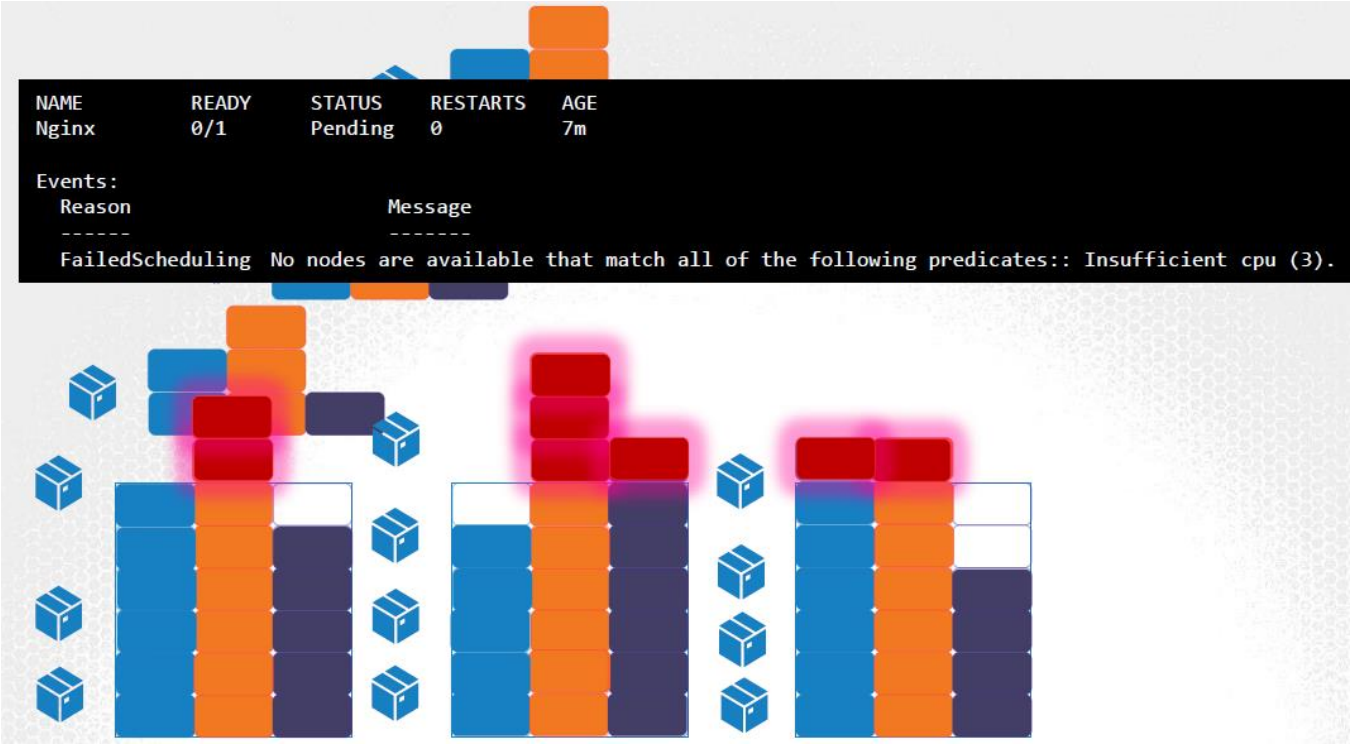
If the node has no sufficient resources, the scheduler avoids placing the POD on that node…



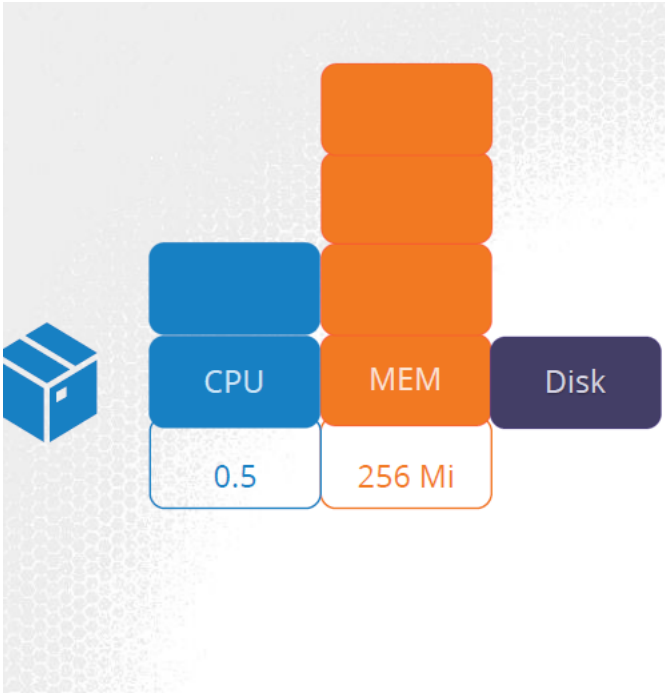
.. Instead places the POD on one were sufficient resources are available. Some of the related topics such as scaling and auto-scaling PODs and Nodes in the cluster and how the scheduler itself works are out of scope for this course and the Kubernetes Application Developer certification. These are discussed in much more detail in the Kubernetes Administrators course. In this course we focus on setting resource requirements for PODs from an application developer's viewpoint.



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 –insufficient cpu.



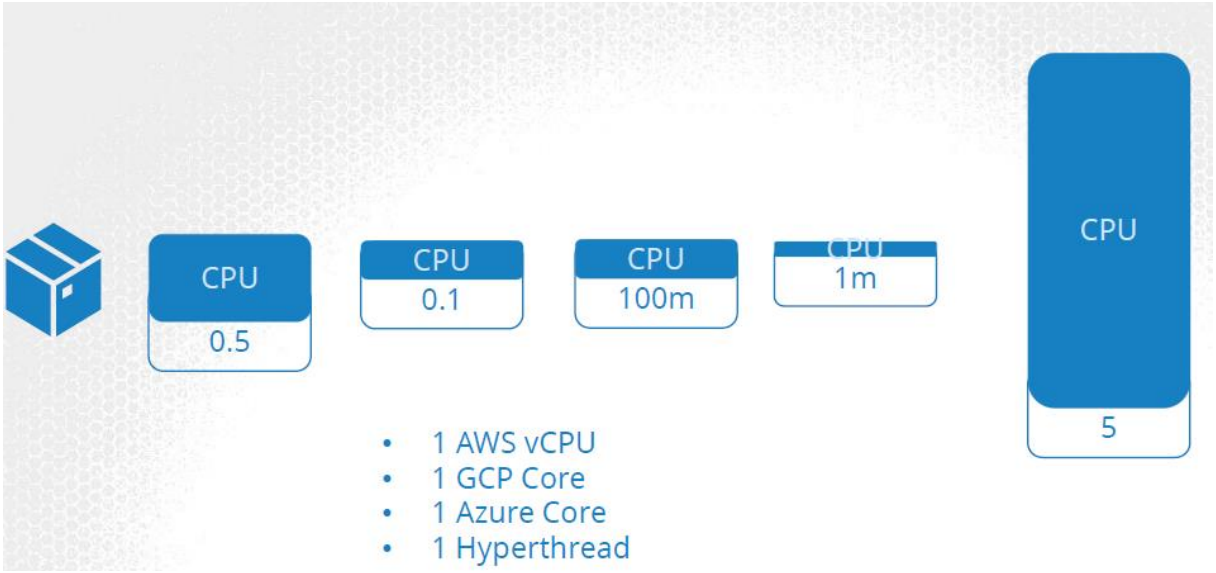
Let us now focus on the resource requirements for each POD. What are these blocks and what are their values? By default, kubernetes assumes that a POD or a container within a POD requires 0.5 CPU & 256 Mebibyte of memory. This is known as the resource request for a container. The minimum amount of CPU or Memory requested by the container. When the scheduler tries to place the POD on a Node, it uses these numbers to identify a Node which has sufficient amount of resources available. Now, if you know that your application will need more than these, you can modify these values, by specifying them in your POD or deployment definition files. In this sample pod definition file, add a section called resources, under which add requests and specify the new values for memory and cpu usage. In this case I set it to 1GB of memory and 1 count of vCPU.



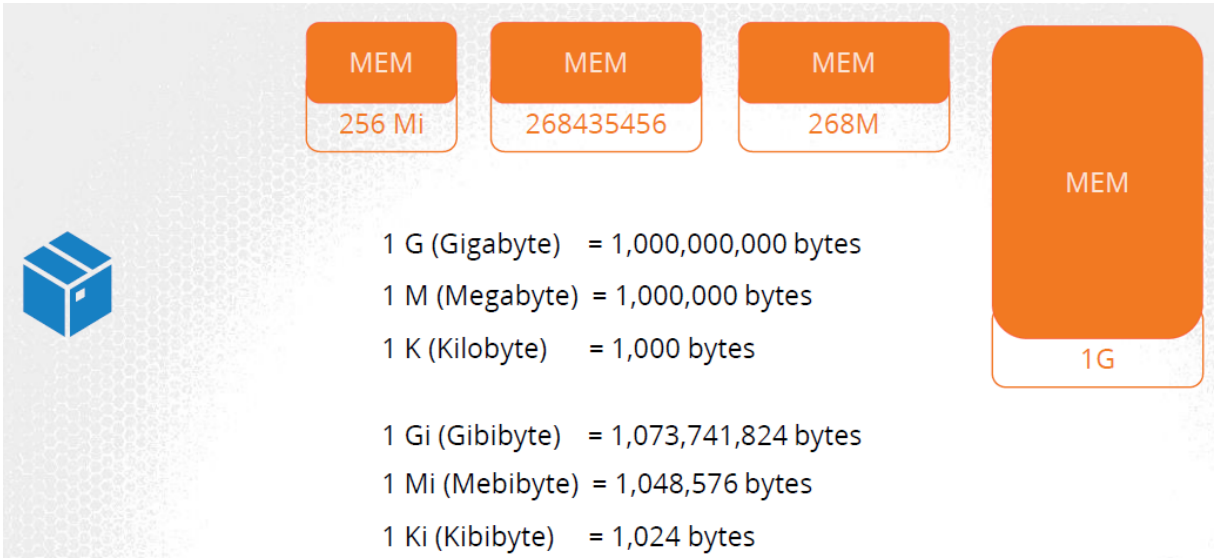
pod-definition.yaml

```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
```

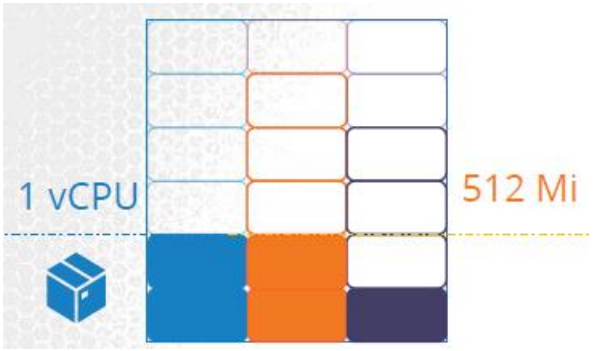
So what does 1 count of CPU really mean? Remember these blocks are used for illustration purpose only. It doesn't have to be in the increment of .5. You can specify any value as low as 0.1. 0.1 CPU can also be expressed as 100m were m stands for milli. You can go as low as 1m, but not lower than that. 1 count of CPU is equivalent to 1 vCPU. That's 1 vCPU in AWS, or 1 Core in GCP or Azure or 1 Hyperthread. You could request a higher number of CPUs for the container, provided your Nodes are sufficiently funded.



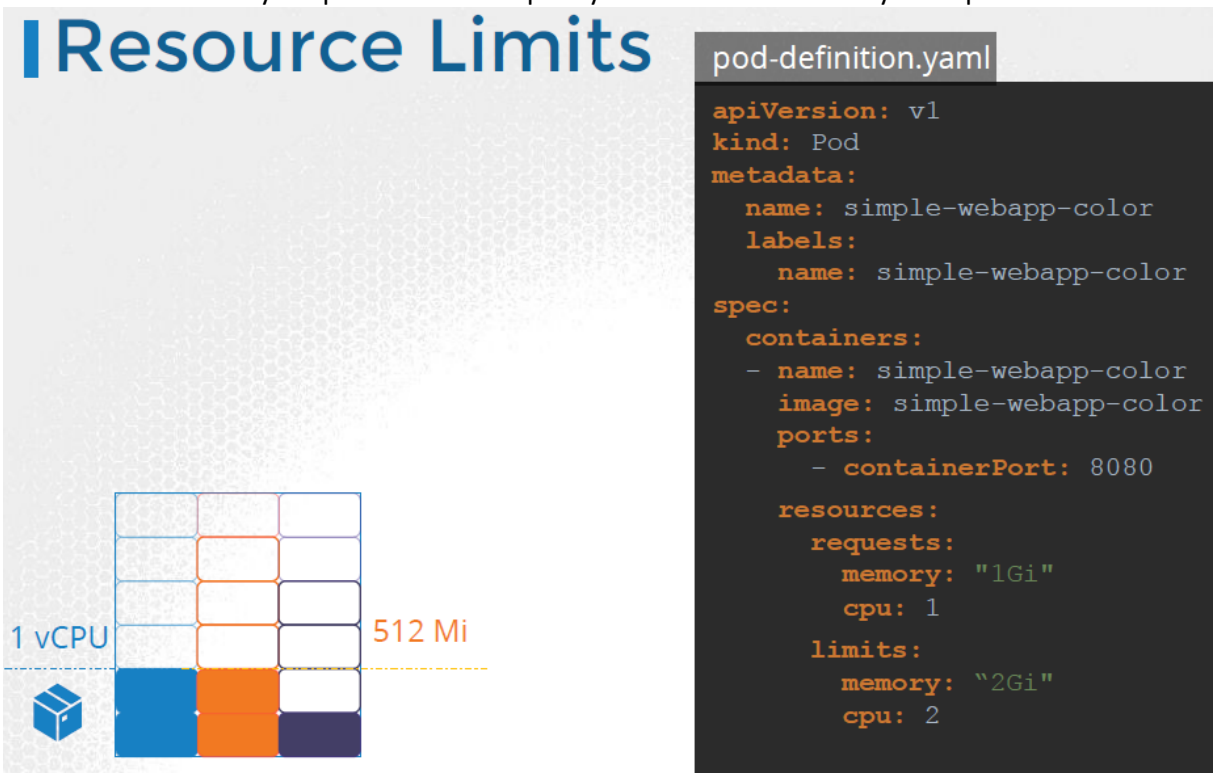
Similarly, with memory you could specify 256 Mebibyte using the Mi suffix. Or specify the same value in Memory like this. Or specify the same value in Memory like this. Or use the suffix G for Gigabyte. Note the difference between G and Gi. G is Gigabyte and it refers to a 1000 Megabytes, whereas Gi refers to Gibibyte and refers to 1024 Mebibyte. The same applies to Megabyte and Kilobyte.



Let's now look at a container running on a Node. In the Docker world, a docker container has no limit to the resources it can consume on a Node. Say a container starts with 1 vCPU on a Node, it can go up and consume as much resource as it requires, suffocating the native processes on the node or other containers of resources. However, you can set a limit for the resource usage on these PODs. By default Kubernetes sets a limit of 1vCPU to containers. So if you do not specify explicitly, a container will be limited to consume only 1 vCPU from the Node.
The same goes with memory. By default, kubernetessets a limit of 512 Mebibyte on containers.



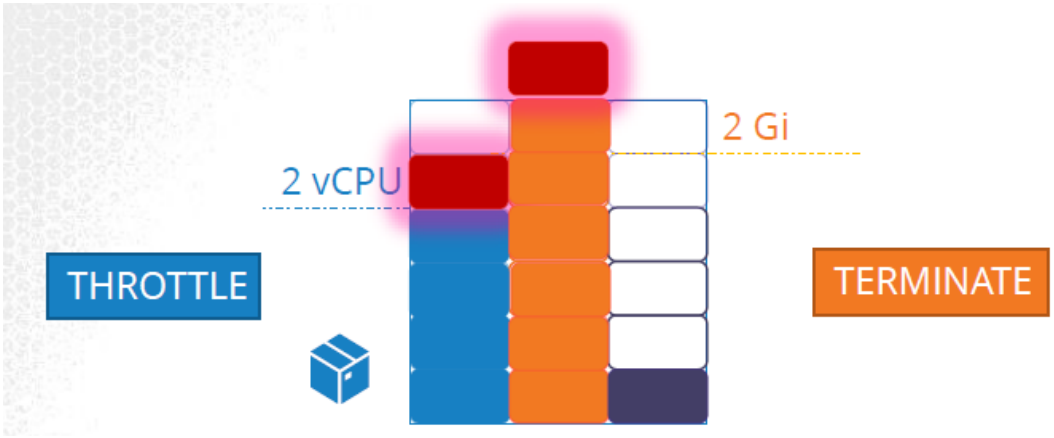
If you don’t like the default limits, you can change them by adding a limits section under the resources section in your pod definition. Specify new limits for memory and cpu.



:

When the pod is created, kubernetes sets new limits for the container. Remember that the limits and requests are set for each container.

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.



Note on default resource requirements and limits

In the previous lecture, I said - "When a pod is created the containers are assigned a default CPU request of .5 and memory of 256Mi". For the POD to pick up those defaults you must have first set those as default values for request and limit by creating a LimitRange in that namespace.

```
1.  apiVersion: v1
2.  kind: LimitRange
3.  metadata:
4.    name: mem-limit-range
5.  spec:
6.    limits:
7.    - default:
8.        memory: 512Mi
9.      defaultRequest:
10.       memory: 256Mi
11.     type: Container
```

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/

```
1.  apiVersion: v1
2.  kind: LimitRange
3.  metadata:
4.    name: cpu-limit-range
5.  spec:
6.    limits:
7.    - default:
8.        cpu: 1
9.      defaultRequest:
10.       cpu: 0.5
11.     type: Container
```

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/cpu-default-namespace/

References:

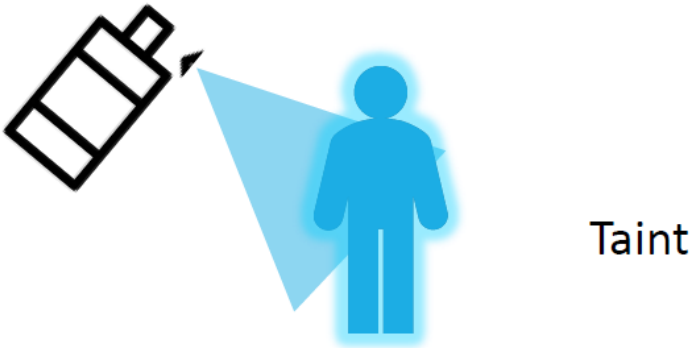
https://kubernetes.io/docs/tasks/configure-pod-container/assign-memory-resource

Taints and tolerations

Let's say we have a bug (a mosquito), intolerant to an odor (lemongrass).

The lemongrass on a person will repel the mosquito but not another bug like a fly, which will land on the person.

In our case, the lemongrass is the 'taint' on the person and the bugs are considered 'tolerant' or 'intolerant' to that 'taint'.



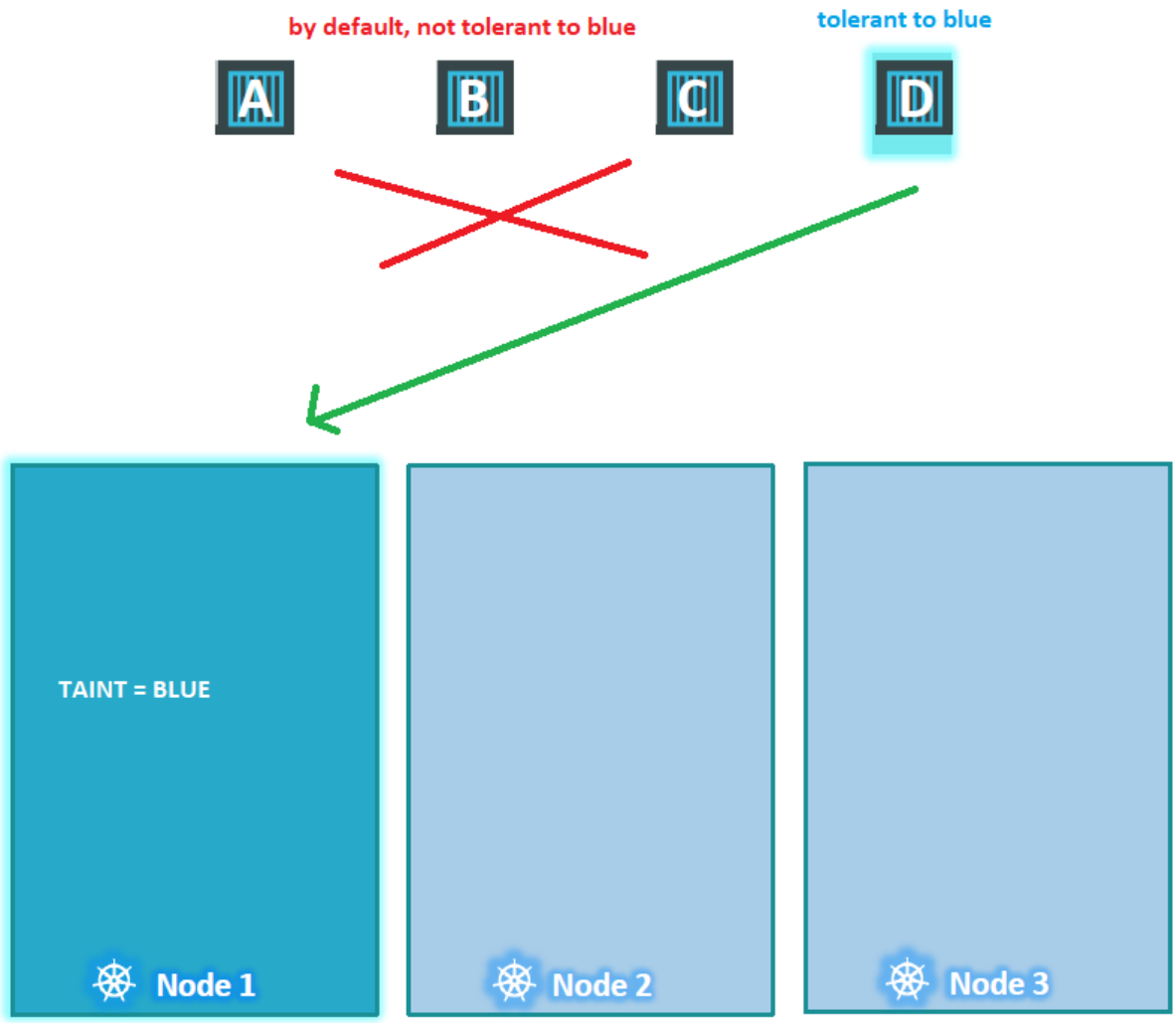
Now in the context of kubernetes, the person is a node and the bugs are pods.

Taints and tolerations are restrictions or limitations that can be used to define what kind of pod can be scheduled on a node.

Let's say that we have several nodes and that we want only certain pods which are parts of one application to run on a specific node.

We will first apply a taint on a node (ex, taint = blue). By default, a taint will repel all pods, preventing them to be runned on that node => nodes are intolerant to taints by default.

If we want the scheduler to schedule a pod to run on that tainted node (WooOohOoh), you will have to configure the pod to be tolerant to blue.



To taint a node, use the command :

```
kubectl taint nodes node-name key=value:taint-effect
```

The taint effect will describe what happens to Pods that do not tolerate this taint.

There are 3 taints effect:

°NoSchedule: the scheduler won't schedule intolerant Pods on that node.

°PreferNoSchedule: the system will try to avoid placing a Pod on that node but it is not guaranteed.

°NoExecute: new Pods will not be scheduled on the node and existing Pods on the node (if any) will be evicted if they are taint intolerant (this can happen if you taint a node with already running Pods).

Ex :

```
kubectltaint nodes node1 app=myapp:NoSchedule
```

Add toleration to a Pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: nginx-container
    image: nginx
  tolerations:
  - key: ”app”
    operator: “Equal”
    value: “blue”
    effect: “NoSchedule”
```

If you wan’t to untaint a node (add the - at the end of the command):

```
kubectltaint nodes node1 app=myapp:NoSchedule-
```

You can also edit the node (kubectl edit node my-node) and remove the taint section.

Note : taints and tolerations don’t tell the Pod to go on a particular node, a Pod tolerant to a tainted node won’t necessarily run on that node. It only tells the node to only accept pods with certain tolerations. If you wan’t to restrict Pods to certain nodes, it is achieved through the node affinity concept.

Note : The master node is tainted at his creation to prevent any other Pods than the ones needed for kub management to be scheduled on it. You can change that behaviour but a best practice is to avoid running application workloads on the master node.

To see the taint run :

```
kubectl describe node kubemaster | grep Taint
```

Node Selectors

Let's say that we have three nodes with different hardware resources. The Node 1 has the highest performances level in term of hardware resource, so we wan't that our biggest workload applications Pods to be run on that node.

If nothing is configured, our workload could be runned on a less performant worker.

We can set limitations on the Pods to solve this problem and force them to be runned on specific nodes.

To do that we can use Node Selectors :

First you have to label your node :

```
kubectl label nodes <node-name> <label-key>=<label-value>
```

```
kubectl label nodes node-1 size=Large
```

Then you can use those selectors on your Pod definition file with the nodeSelector label :

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
```

```
 - name: data-processor
   image: data-processor
 nodeSelector:
   size: Large
```

Note: Node Selectors are useful but they have their limitations. For example if we want to configure a Pod to run on a Large OR a Medium node, or something like : “place the Pod on any node which is not small”, well we can’t achieve that with nodeSelector.

For this, Node Affinity and Anti-Affinity features are used.

Node Affinity

The purpose of Node Affinities is to ensure that pods are hosted on particular nodes.

We can achieve the same effect than nodeSelector, but note that, because it is more flexible, the notation becomes more complex.

Let’s take the previous nodeSelector example using node Affinity instead:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: size
            operator: In
            values:
```

```
        - Large
```

It means that the node affinity is taken into account when the Pod is scheduled on it's creation but not when the Pod is already running.

Then to select the correct nodes, under nodeSelectorTerms, the Pod will run on a node matching the key size which value is IN the array values (in that case only Large but we can add more values like medium, small etc...).

```
        operator: In
        values:
        - Large
        - Medium
```

For the operator, you can also use the NotIn value to exclude certain labels:

```
        operator: NotIn
        values:
        - Small
        - Nano
```

You can use the operator Exists to only check if the key of the label exists and run the Pod on the node whatever the value of that key:

```
        - key: size
        operator: Exists
```

Other operators exists, you can check the official documentation.

But what happen if there are no nodes matching the criteria of the nodeAffinity rules?

Or if somebody changes the node label with already running pods on it?

Well all that is answered in the type of the nodeAffinity, wich is the long sentence right under the nodeAffinity section.

There are currently only two types available:

Available:

requiredDuringSchedulingIgnoredDuringExecution

preferredDuringSchedulingIgnoredDuringExecution

	DuringScheduling	DuringExecution
Type 1	Required	Ignored
Type 2	Preferred	Ignored

The “preferred” sentence will tell kubernetes “you can try to find a node with labels matching the criteria I give you on the Pod creation, but if none, run it anyway on another node”.

The “required” sentence will not run a Pod if the label criteria don’t matches with any node.

For now there is only one type for running Pods (execution) which is “ignored”. That means that if the label of a node change or is deleted, even if the nodeAffinity criteria don’t matches anymore, the Pods will continue to run on that node.

A new type of nodeAffinity is expected for the future:

Type 3	Required	Required

With that type, a Pod, running on a node, which is no more matching the nodeAffinity criteria, will be terminated.

Taints and tolerations + Node Affinity

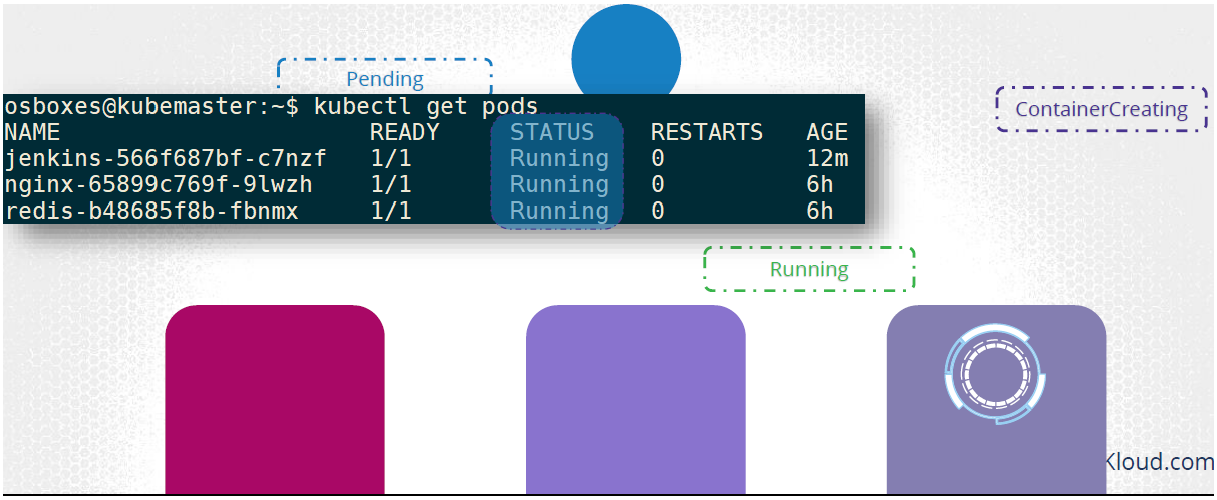
You can combine both of them to ensure that only desired pods runs on desired nodes.

Observability

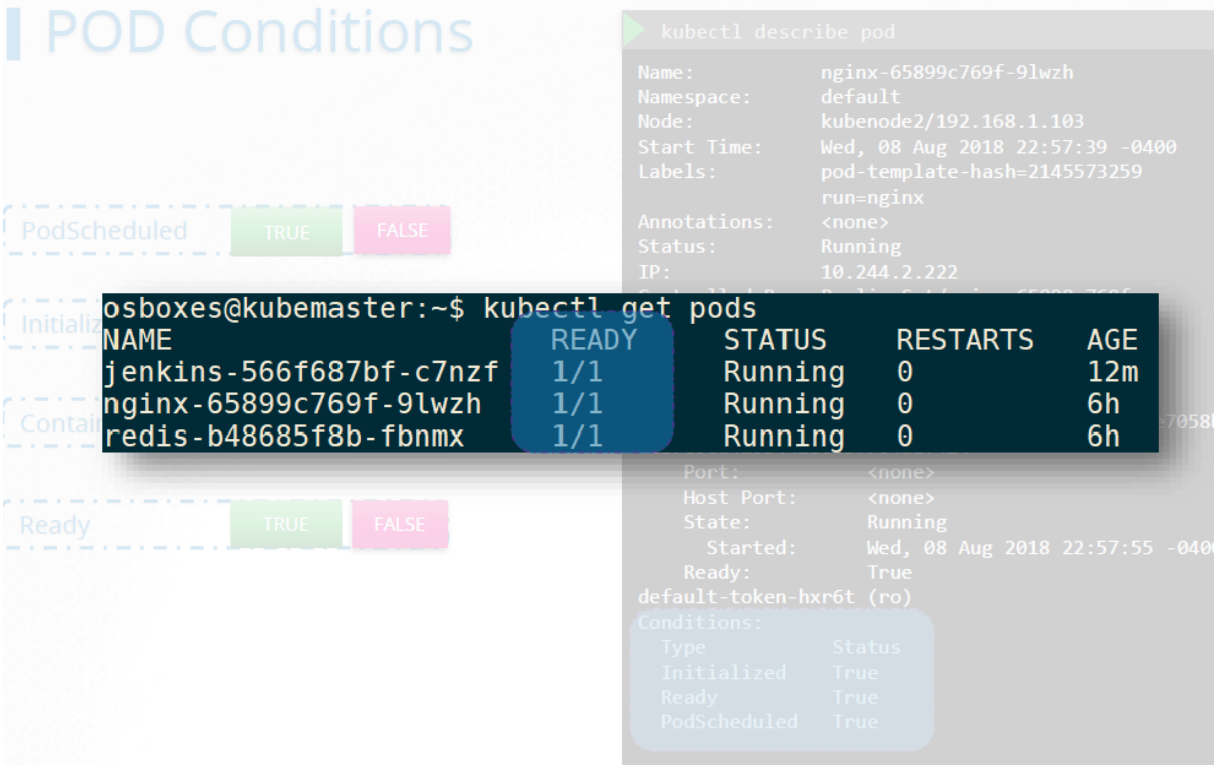
Readiness and Liveness Probes

The POD status tells us were the POD is in its lifecycle. When a POD is first created, it is in a Pending state. This is when the Scheduler tries to figure out were 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 kubectldescribe pod command, and it will tell you exactly why. Once the POD is scheduled, it goes into a ContainerCreating status, 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, were it continues to be until the program completes successfully or is terminated.

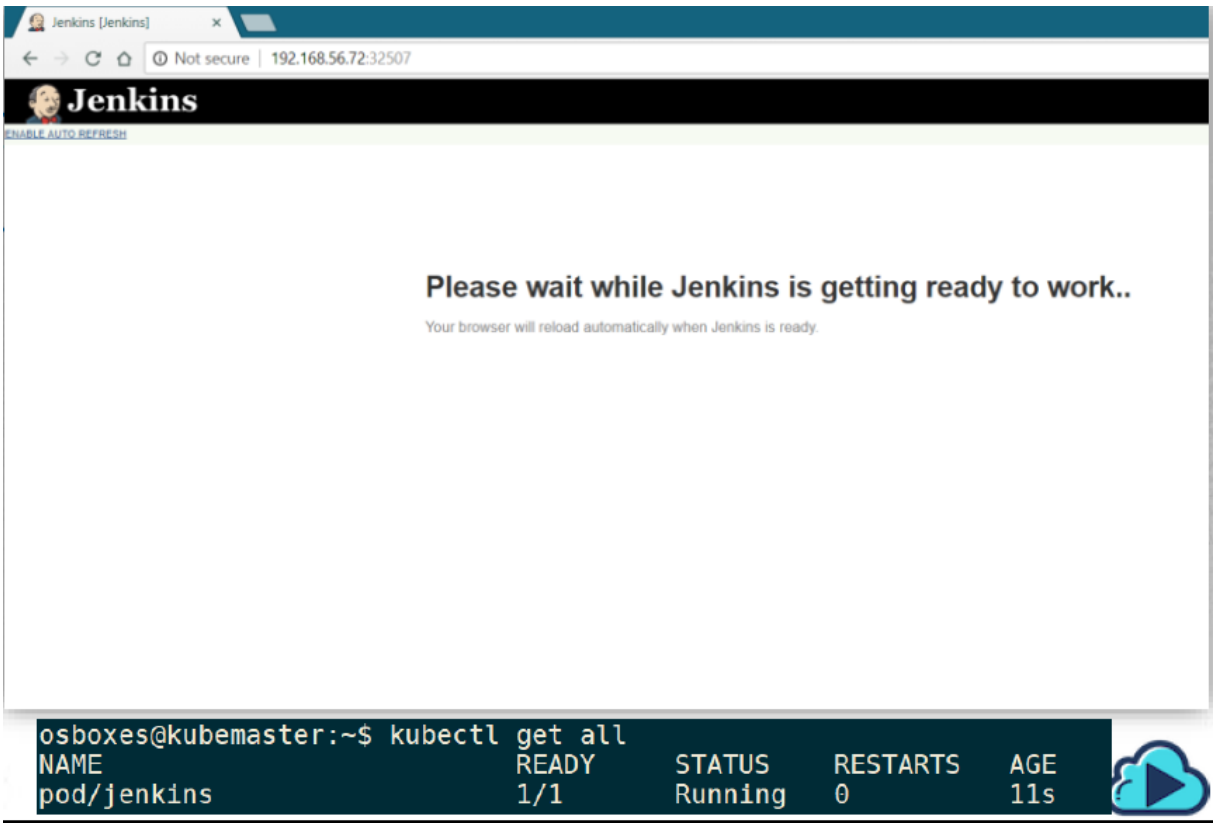
You can see the pod status in the output of the kubectlget pods command. So remember, at any point in time the POD status can only be one of these values and only gives us a high level summary of a POD. However, at times you may want additional information.



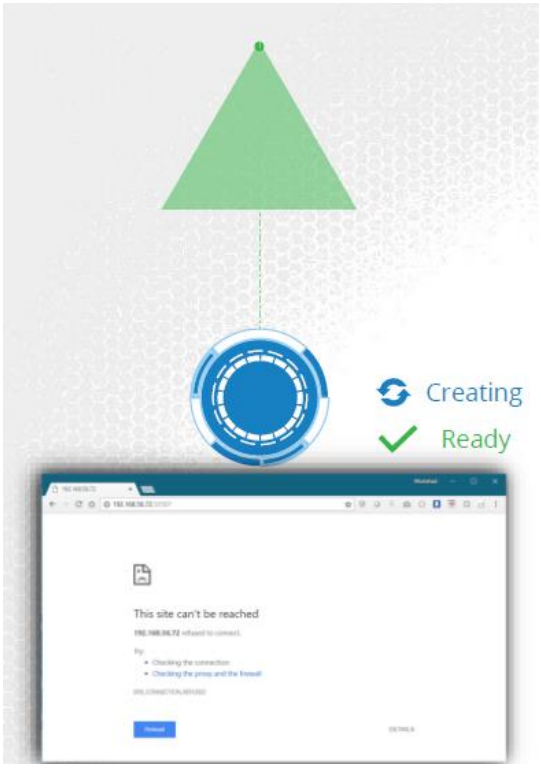
Conditions complement POD status. It is an array of true or false values that tell us the state of a POD. When a POD is scheduled on a Node, the PodScheduledcondition is set to True. When the POD is initialized, it's value is set to True. We know that a POD has multiple containers. When all the containers 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 kubectldescribe POD command and look for the conditions section.
You can also see the Ready state of the POD, in the output of the kubectlget pods command. And that is the condition we are interested in for this lecture.



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.
So why is that happening and how does kubernetes know whether that the application inside the container is actually running or not? But before we get into that discussion, why does it matter if the state is reported incorrectly.

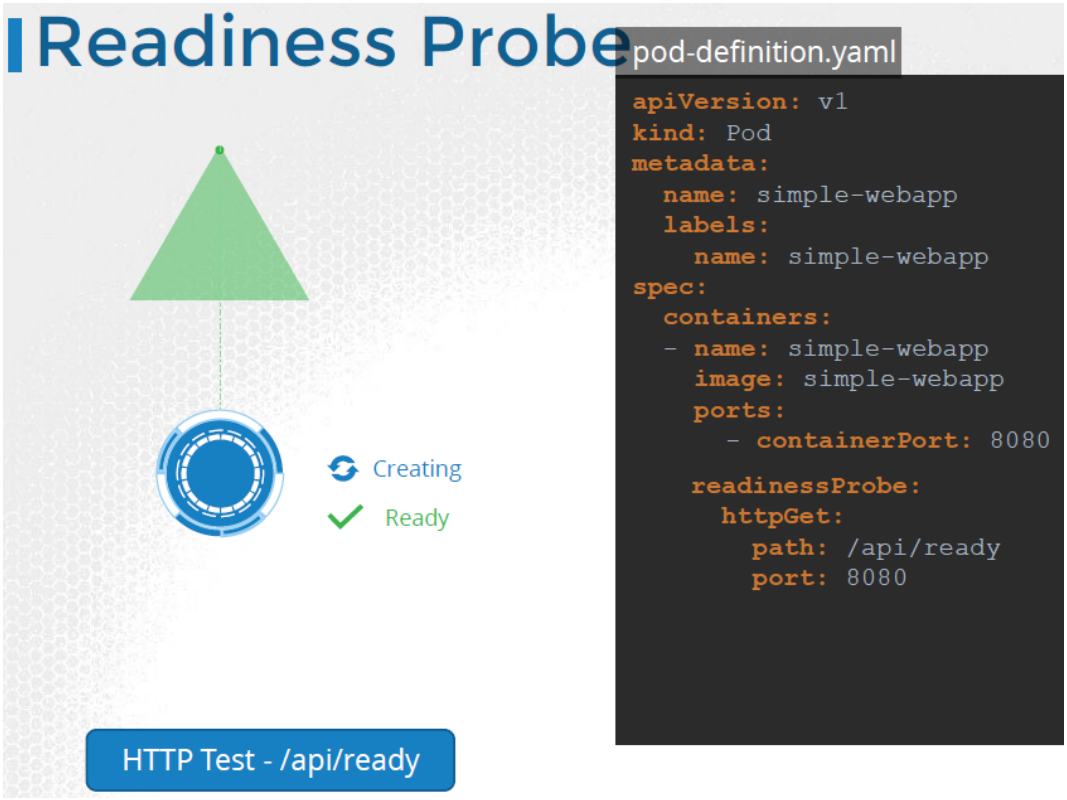


Let us look at a simple scenario were you create a POD and expose it to external users using a service. The service will route traffic to the POD immediately. The service relies on the pod's READY condition to route traffic.
By default, Kubernetes assumes that as soon as the container is created, it 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.

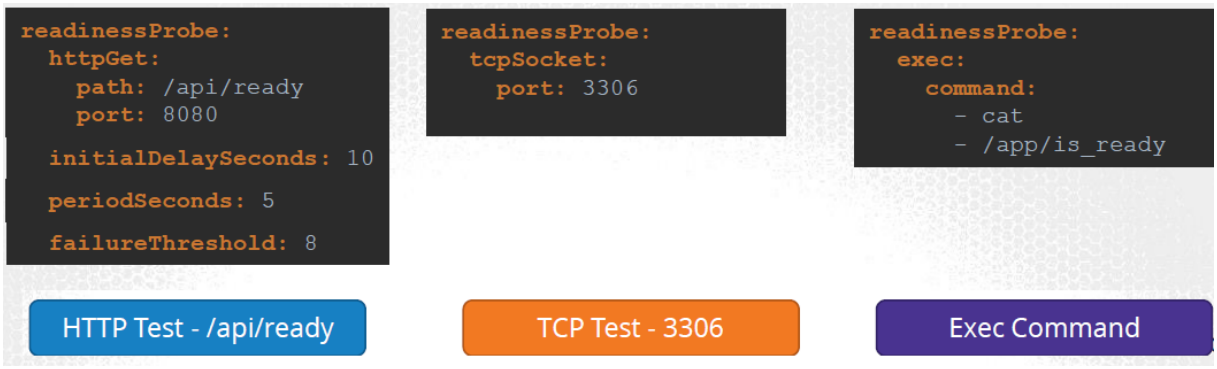


There are different ways that you can define if an application inside a container is actually ready. You can setup different kinds of tests or Probes, which is the appropriate term. In case of a web application it could be when the API server is up and running. So you could run a HTTP test to see if 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.

So how do you configure that test? In the pod definition file, add a new field called readinessProbeand use the httpGetoption. 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 apiresponds positively. Until then the service does not forward any traffic to the pod, as it sees that the POD is not ready.



There are different ways a probe can be configured. For http, use the httpGetoption with the path and port. For TCP use the tcpSocketoption with port. And for executing a command specify the exec option with the command and options in an array format. 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 periodSecondsoption. By default if the application is not ready after 3 attempts, the probe will stop. If you'd like to make more attempts, use the failureThresholdoption. We will look through more options in the Documentation Walkthrough.



Finally, Let us look at how readinessProbesare useful in a multi-pod setup. 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 readinessProbeconfigured 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

Let's start from the basics. You run an image of NGINX using docker and it starts to serve users. For some reason the web server crashes and the nginx process exits. The container exits as well. And you can see the status of the container when you run the docker ps command. Since docker is not an orchestration engine, the container continues to stay dead and deny services to users, until you manually create a new container.

```
docker ps -a
CONTAINER ID        IMAGE       CREATED              STATUS                          PORTS
45aacca36850        nginx       43 seconds ago       Exited (1) 41 seconds ago
```

Enter Kubernetes Orchestration. You run the same web application with kubernetes. 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.

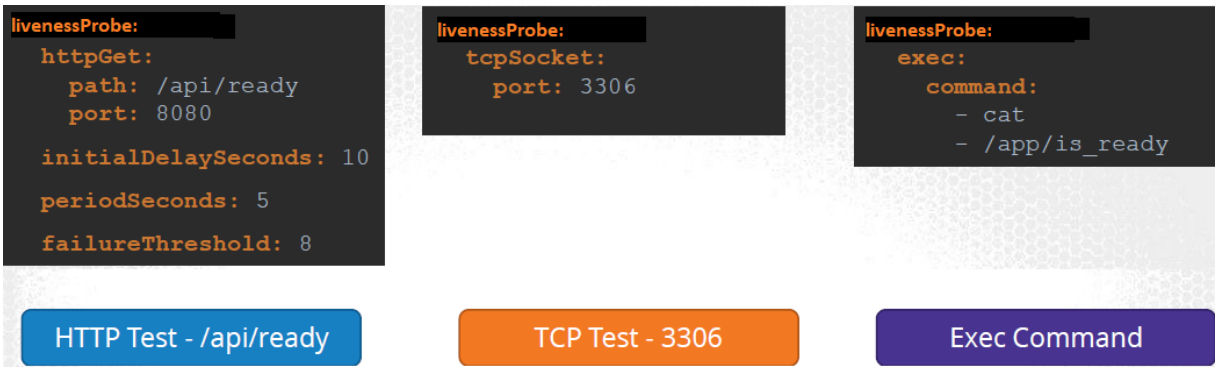
```
kubectl get pods
NAME                READY     STATUS      RESTARTS   AGE
nginx-pod           0/1       Completed   2          1d
```

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.
In case of a web application it could be when the API server is up and running. In case of database, you may test to see if a particular TCP socket is listening. Or You may simply execute a command to perform a test.

The liveness probe is configured in the pod definition file as you did with the readinessProbe. Except here you use liveness instead of readiness.

Similar to readiness probe you have httpGetoption for apis, tcpSockerfor ports and exec for commands. As well as additional options like initialDelay before the test is run, periodSeconds to define the frequency and success and failure thresholds.



Container logging

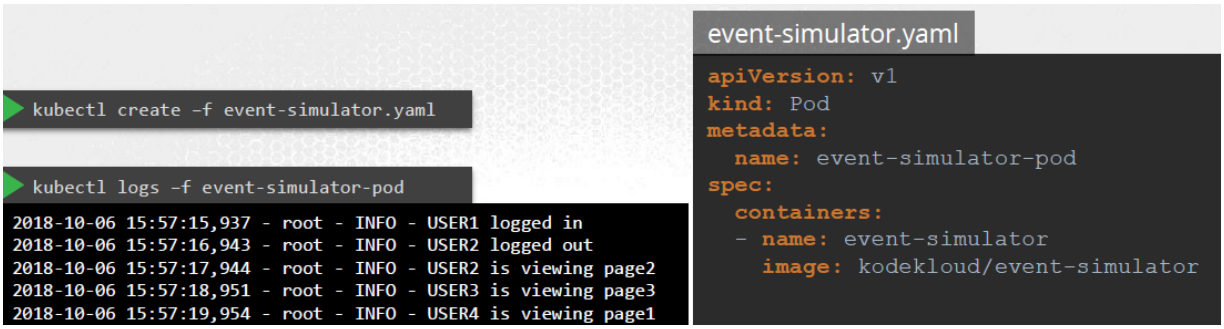
Let us start with logging in Docker. I run a docker container called event-simulator and all that it does is generate random events simulating a web server. These are events streamed to the standard output by the application.

```
docker run kodekloud/event-simulator
2018-10-06 15:57:15,937 - root - INFO - USER1 logged in
2018-10-06 15:57:16,943 - root - INFO - USER2 logged out
2018-10-06 15:57:17,944 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:18,951 - root - INFO - USER3 is viewing page3
2018-10-06 15:57:19,954 - root - INFO - USER4 is viewing page1
2018-10-06 15:57:20,955 - root - INFO - USER2 logged out
```

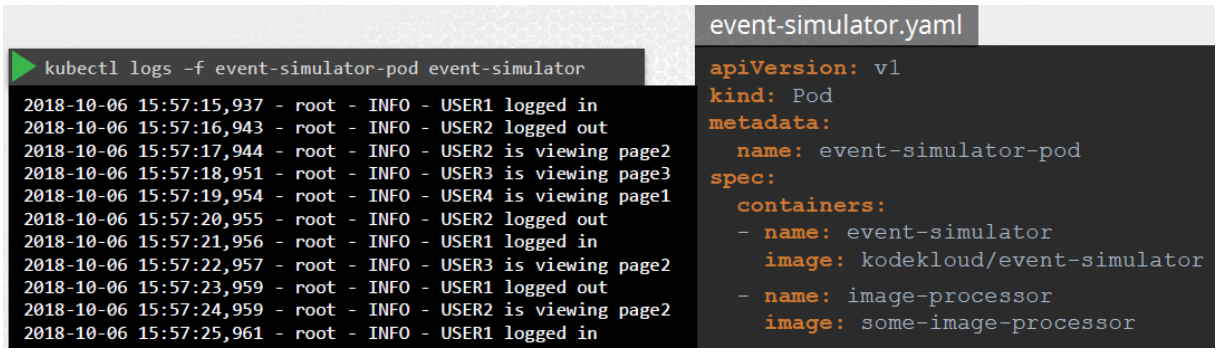
Now, if I were to run the docker container in the background, in a detached mode using the –d option, I wouldn't see those logs. If I wanted to view the logs, I could use the docker logs command followed by the container ID. The –f option helps us see the live log trail.

```
docker run -d kodekloud/event-simulator

docker logs -f ecf
2018-10-06 15:57:15,937 - root - INFO - USER1 logged in
2018-10-06 15:57:16,943 - root - INFO - USER2 logged out
2018-10-06 15:57:17,944 - root - INFO - USER2 is viewing page2
2018-10-06 15:57:18,951 - root - INFO - USER3 is viewing page3
```

Now back to Kubernetes. We create a pod with the same docker image using the pod definition file. Once it's the pod is running, we can view the logs using the kubectl logs command with the pod name. Use the –f option to stream the logs live.



Now, these logs are specific to the container running inside the POD. As we learned before, Kubernetes PODs can have multiple docker containers in them. In this case I modify my pod definition file to include an additional container called image-processor. If you ran the kubectllogs command now with the pod name, which container's log would it show? If there are multiple containers within a pod, you must specify the name of the container explicitly in the command, otherwise it would fail asking you to specify a name. In this case I will specify the name of the first container event-simulator and that prints the relevant log messages.



Monitor and debug applications

So how do you monitor resource consumption on Kubernetes? Or more importantly what would you like to monitor? I'd 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.

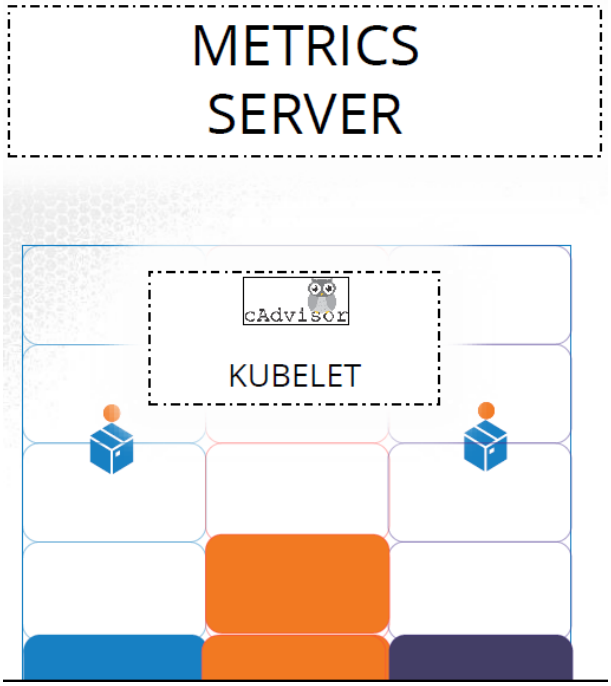
As well as POD level metrics such as the number of PODs, and performance metrics of each POD such the CPU and Memory consumption. So we need a solution that will monitor these metrics, store them and provide analytics around this data.
As of this recording , Kubernetes does not come with a full featured 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.

So in the scope of this course, we will discuss about the Metrics Server only.

Heapsterwas one of the original projects that enabled monitoring and analysis features for Kubernetes. You will see a lot of reference online when you look for reference architectures on monitoring Kubernetes. However, Heapsteris now Deprecated and a slimmed down version was formed known as the Metrics Server.
You can have one metrics server per kubernetes cluster.

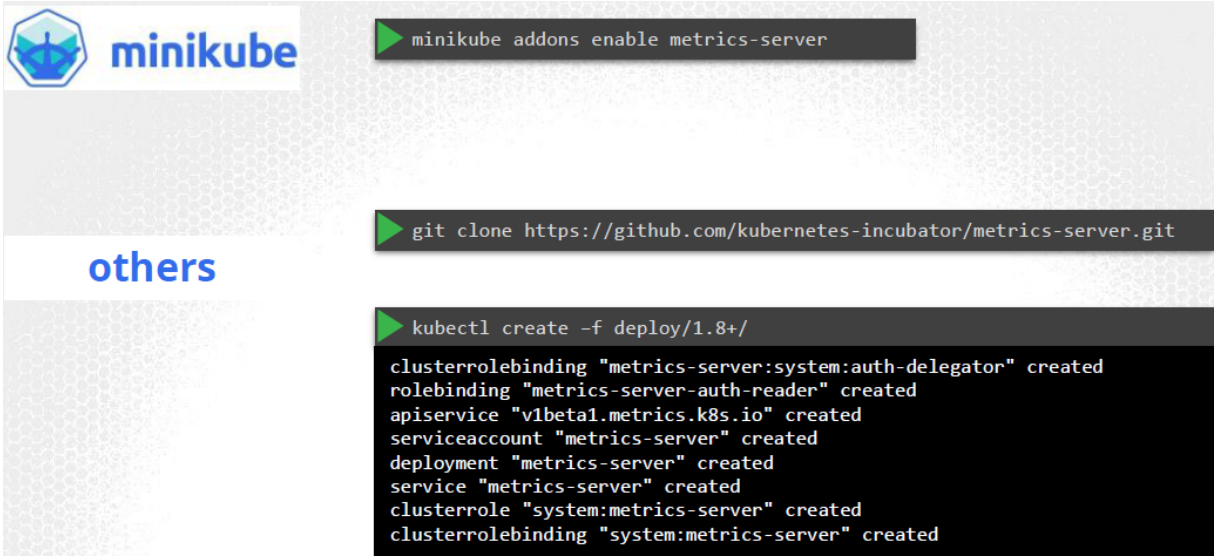
The metrics server retrieves metrics from each of the kubernetes nodes and pods, aggregates them and stores them in memory. Note that the metrics server is only an in-memory monitoring solution and does not store the metrics on the disk, and as a result you cannot see historical performance data. For that you must rely on one of the advanced monitoring solutions we talked about earlier in this lecture.

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 kubeletalso contains a subcomponent known as ascAdvisoror Container Advisor. cAdvisoris responsible for retrieving performance metrics from pods, and exposing them through the kubeletAPI to make the metrics available for the Metrics Server.



If you are using minikubefor your local cluster, run the command minikubeaddons enable metrics-server. For all other environments deploy the metrics server by cloning the metrics-

server deployment files from the github repository. And then deploying the required components using the kubectl create command. This command deploys a set of pods, services and roles to enable metrics server to poll for performance metrics from the nodes in the cluster (you have to launch the kubectl create -f command precising the folder containing all the yaml files you just downloaded).



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. As you can see 8% of the CPU on my master node is consumed, which is about 166 milli cores.
Use the kubectl top pod command to view performance metrics of pods in kubernetes.

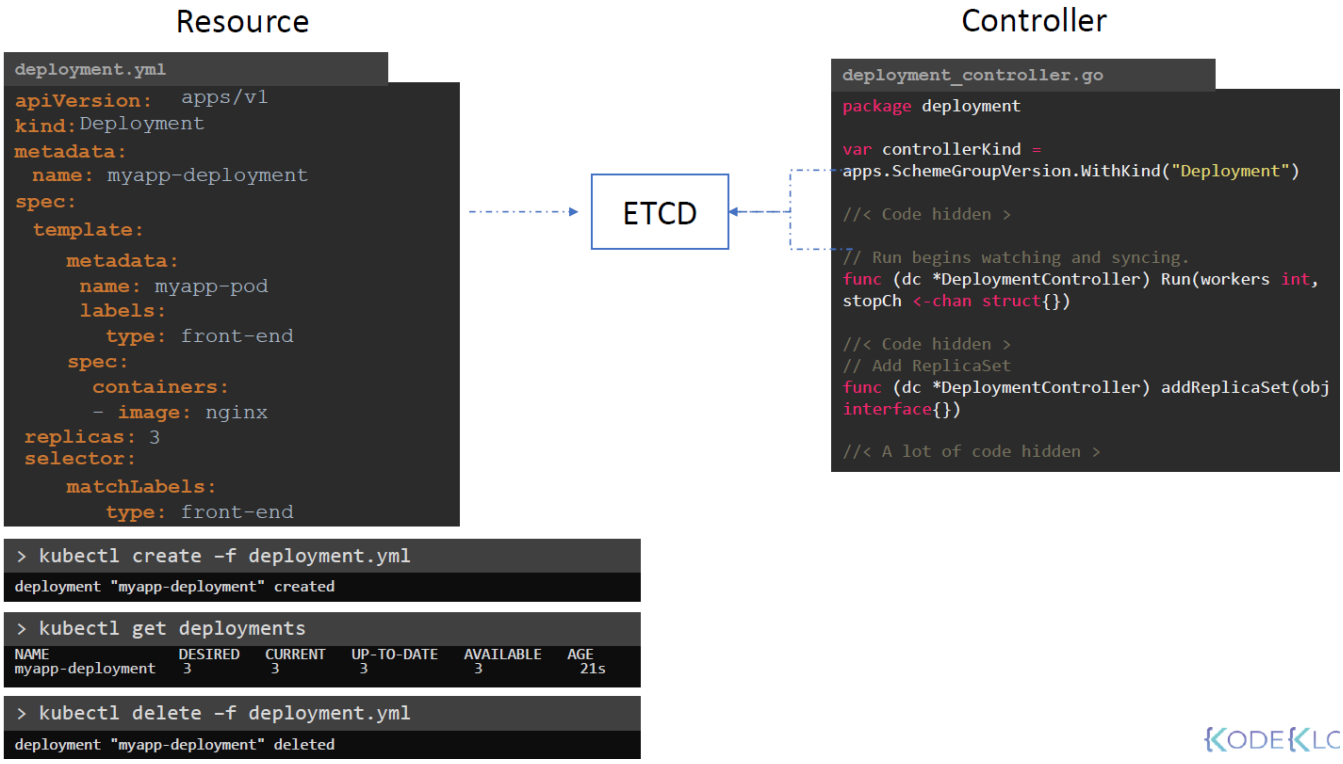
```
kubectl top node
NAME         CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
kubemaster   166m         8%        1337Mi          70%
kubenode1    36m          1%        1046Mi          55%
kubenode2    39m          1%        1048Mi          55%
```

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

Custom resource and Controller definition

Custom resource definition

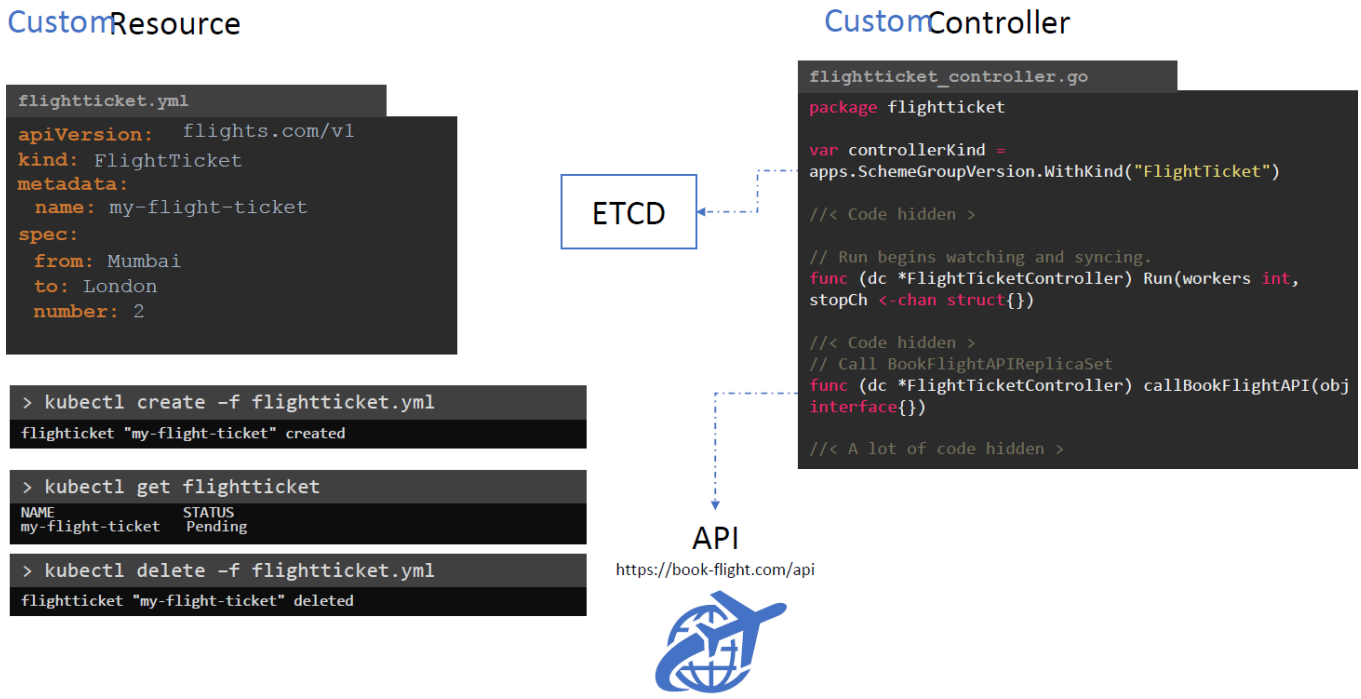
When you create a resource, let's say a deployment, kubernetes stores it's information in the etcd data store. All the commands you do to get details about the deployment, delete the deployment, edit the deployment etc… will read, delete or modify the object or resource stored in the etcd data store. But when we create a deployment with 3 replicas, what is the component which is creating the ReplicaSets and the Pods? Well that's the job of a controllers (in this case the deployment controller). The controller is a process that runs in the background and it's job is to continuously monitor the statuses of resources, in the etcd data store, that it's supposed to manage (ex deployments) and when we create, delete or edit that kind of resources, it makes the necessary changes on the cluster to match the action done. For example, the deployment controller will create a ReplicaSet and in turn the ReplicaSet controller will creates the Pods. The controllers are written in go language and are part of the kubernetes source code:



There are controllers for each kind of resources:

We can create our own controllers to manage our own resources/objects.

For example I want to create a FlighTicket resource, which means that I can create it using kubectl, store it in etcd data store and use a custom FlightTicket Controller to manage my objects. The CustomController will contact an external API to buy, delete or edit real flight tickets.



Of course you can't simply try to create your object like that, the kube-apiserver will throw an error because it doesn't know that resource.

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

To create that resource, we need to create a Custom Resource Definition file (CRD).

The CustomResourceDefinition is a kub object, we can create it using the api group "apiextensions.k8s.io/v1".

In the spec section, we can precise the scope of the object (namespaced or not), the api group (api group that we provide in the apiVersion of our object, flights.com), and the names of the resource (the kind field: FlightTicket, the singular and plural names used by the kubectl command and the short names).

Then we can also precise the different api versions available (and which version is the storage and which one is the served version).

Then we pass the schema of our object, this will define the fields of our resource.

When defining fields we can also precise limits (like a maximum or a minimum number), this way if it's out of scope, an error message will be thrown and the resource won't be created.

CustomResource

Custom Resource Definition (CRD)


```
    shortNames:
      - ft
  versions:
    - name: v1
      served: true
      storage: true

      schema:
        openAPIV3Schema:
            type: object
            properties:
              spec:
                type: object
                properties:
                  from:
                    type: string
                  to:
                    type: string
                  number:
                    type: integer
                    minimum: 1
                    maximum: 10
```

```
> kubectl create -f flightticket-custom-definition.yml
customresourcedefinition created
```

Now that our CustomResource exists, we can save and edit the definition file in the etcd data store, the next step is to create our resource Controller to define the actions relative to that resource manipulation.

You can also define a scope where your custom resource can be used and accessed (at the same level than scope, group, names…), either namespaced or cluster wide (Cluster).

```
# either Namespaced or Cluster
  scope: Namespaced
```

Custom Controller definition

You can create a custom controller which will monitor the resource of your choice and act the way you decide.

There is a github template written in go (kubernetes is coded in go language) that you can use:

https://github.com/kubernetes/sample-controller

To test it you'll have to install go on your machine.

```
> 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
```

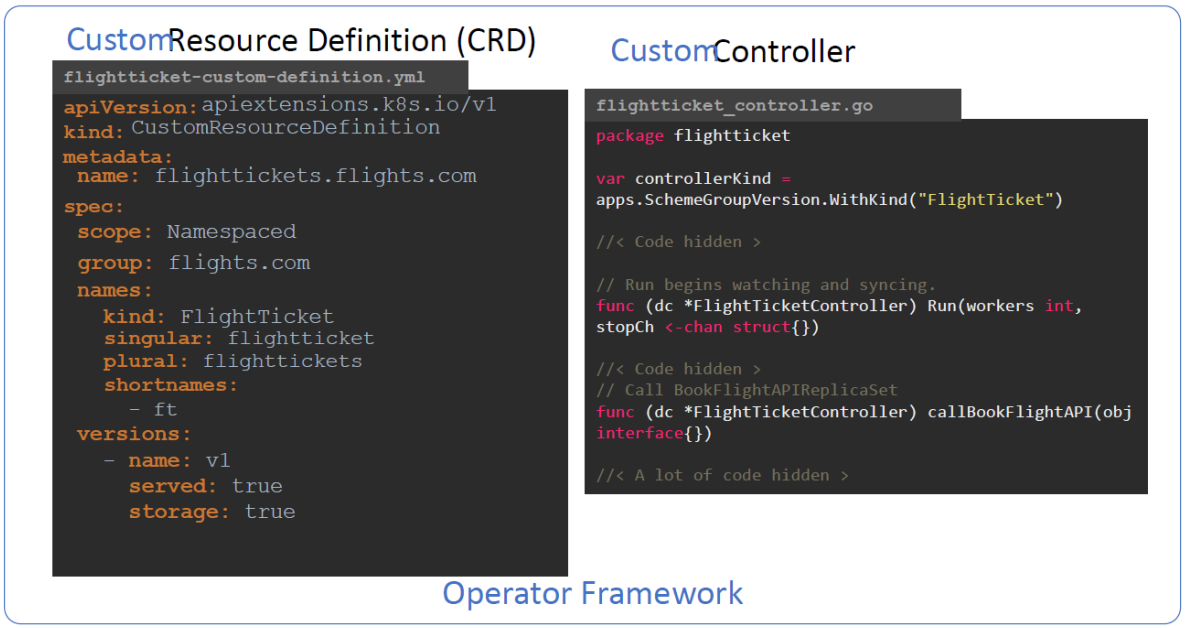
```
> ./sample-controller -kubeconfig=$HOME/.kube/config
I1013 02:11:07.489479  40117 controller.go:115] Setting up event handlers

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

controller.go

```
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 >
```

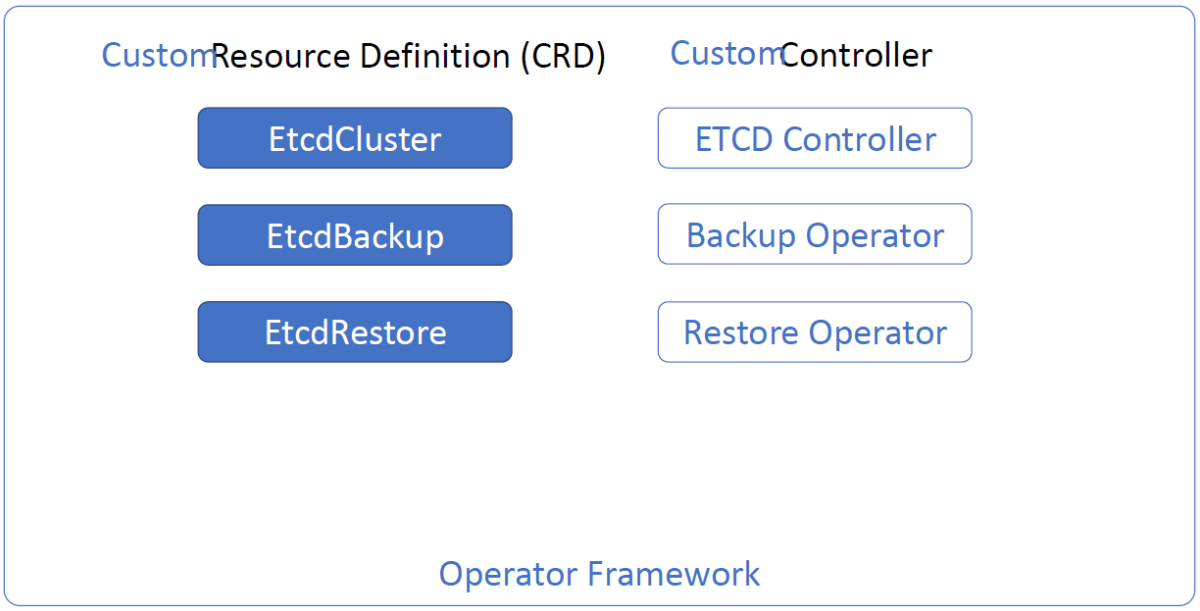
Operator Framework (optional for certification)

The operator framework is used to package together the custom resource definition CRD and the Custom Controller to be deployed as a single entity.

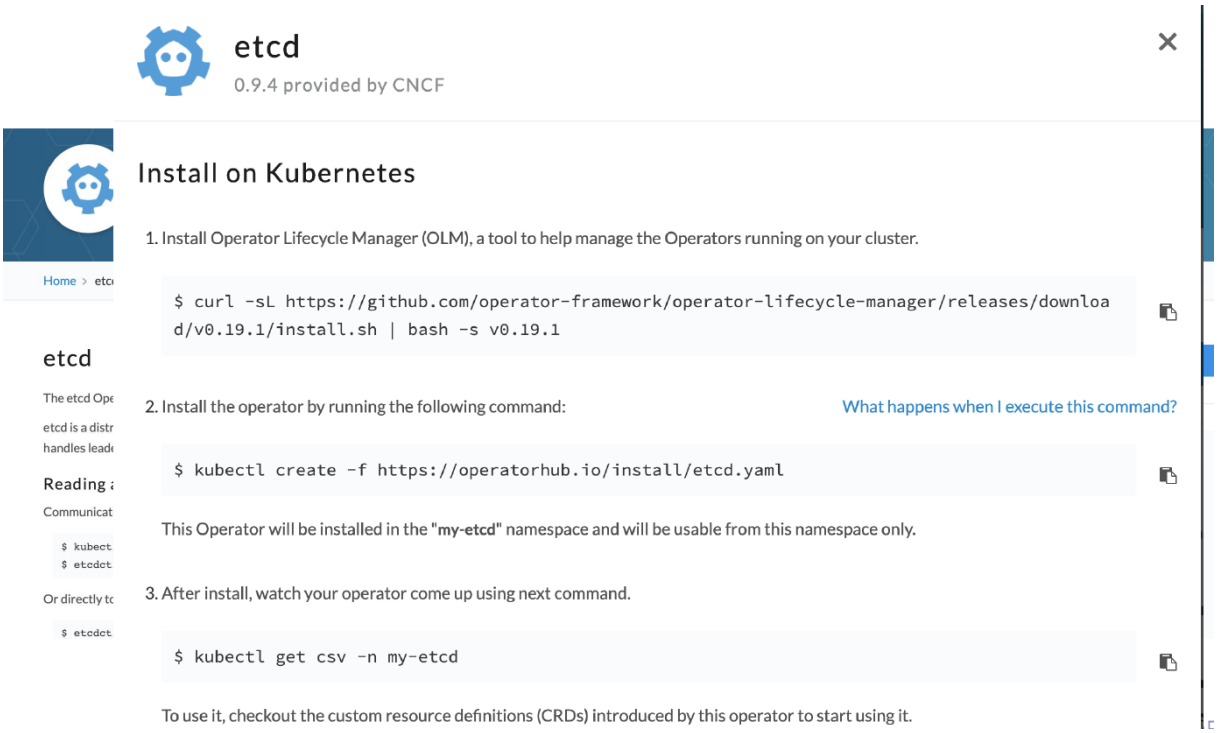



```
> kubectl create -f flight-operator.yaml
```

Real use case example : One of the most popular framework operator is the etcd operator, which is used to deploy and manage etcd cluster. This operator is used to manage backups and restoration of the ectd component in case of disasters, fixing issues that may come across the application etc…



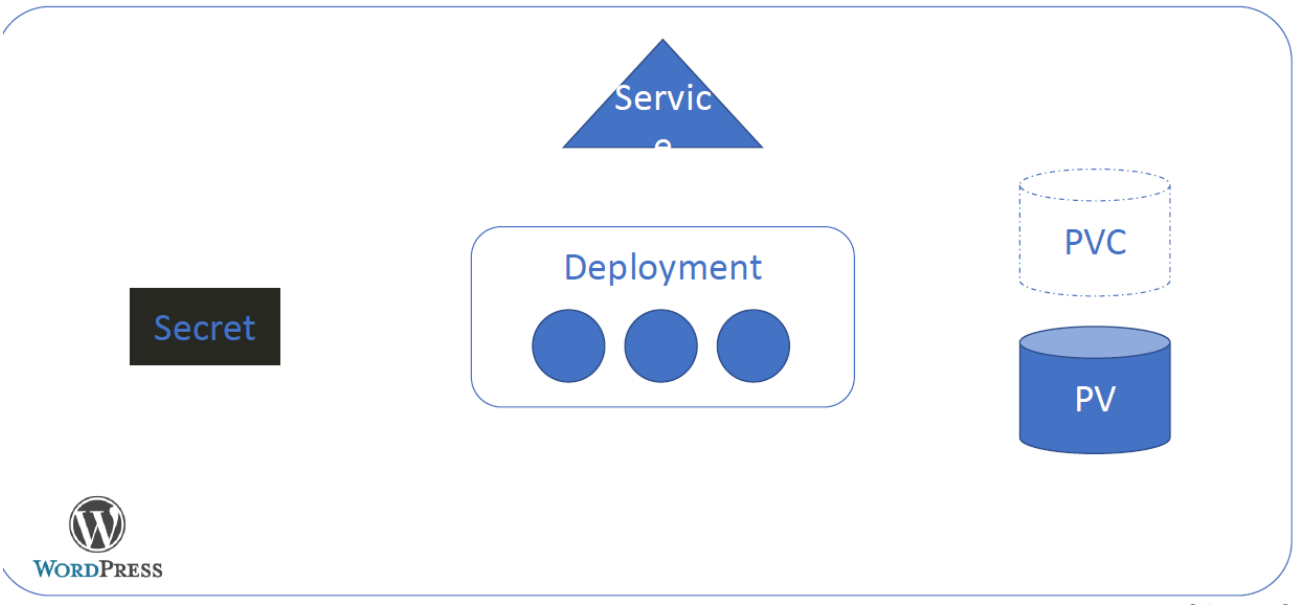
All operators are available at the operatorhub.io. You can find operators for a lot of usefull technologies like Prometheus etc…



HELM

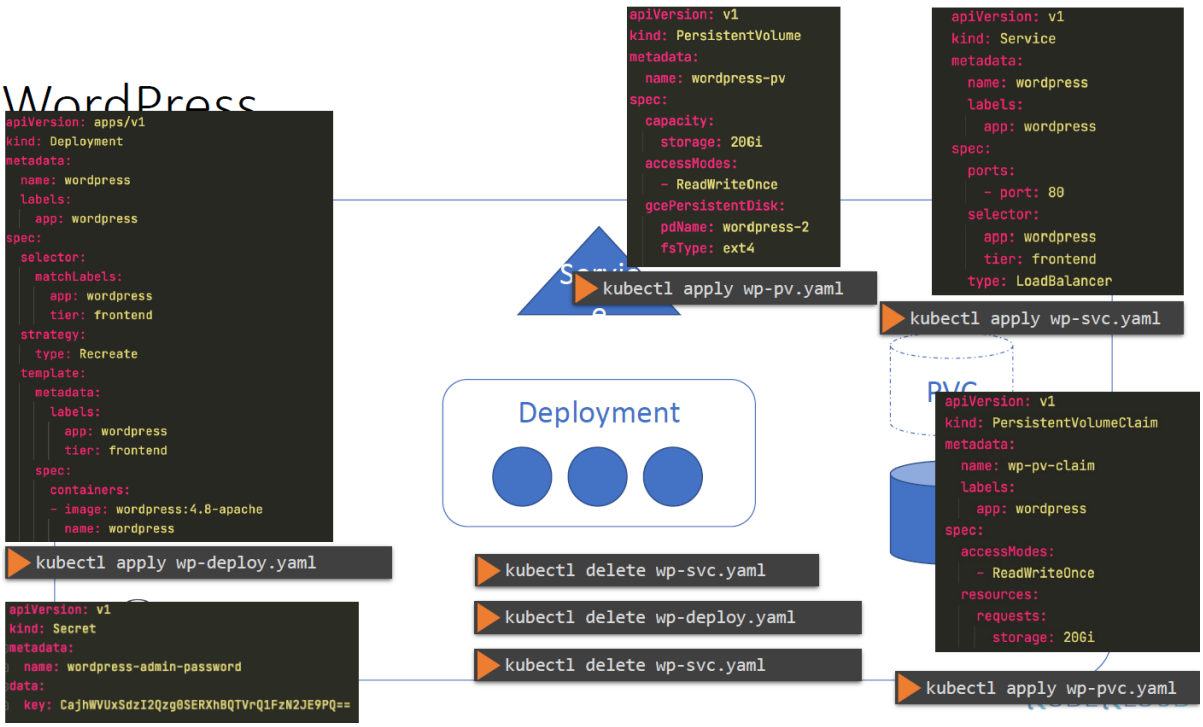
What is Helm?

It usually takes several kub objects to manage even a simple app like wordpress:



As kubernetes does not manage an entire app but only manage kub objects, applying/updating/deleting each of this configuration files is a tedious task as we have to go through each files.

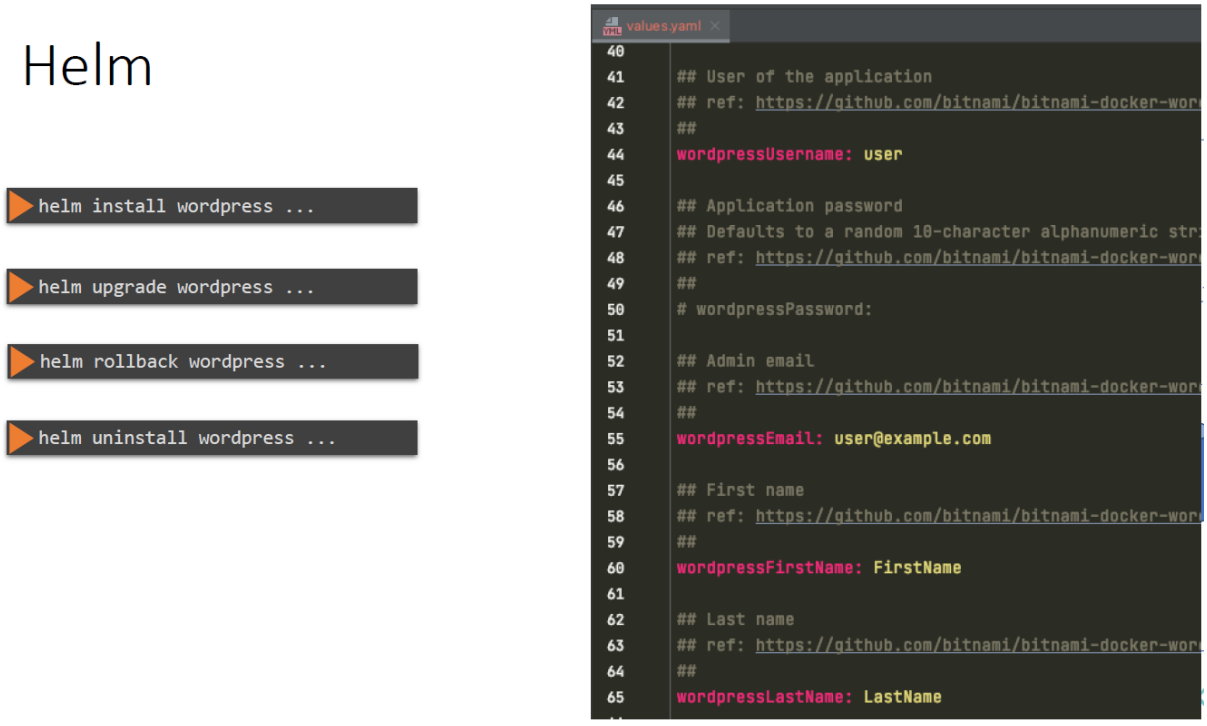
Even if there is a single yaml file for the entire app, it can be a harsh task to find a specific parameter through hundreds or thousands of code lines.



And that's why HELM is a really useful tool.

HELM is used to install whole "packages", it will manage a group of connected kub objects to easily manage an app (services, Pods, Secrets, Deployments, Volumes…).

Using Helm is really easy, you will have a config file that you can use to personalize your app and HELM will do the rest:



Thanks to Helm, you don't have to micromanage each object one by one anymore.

Install Helm

On linux, depending of your os, you have several ways to install Helm:


```
sudo snap install helm  --classic
```

```
curl https://baltocdn.com/helm/signing.asc | sudo apt-key add -
sudo apt-get install apt-transport-https --yes
echo "deb https://baltocdn.com/helm/stable/debian/ all main" | sudo tee /etc/apt/sources.list.d/helm-stable-debian.list
sudo apt-get update
sudo apt-get install helm
```

```
pkg install helm
```

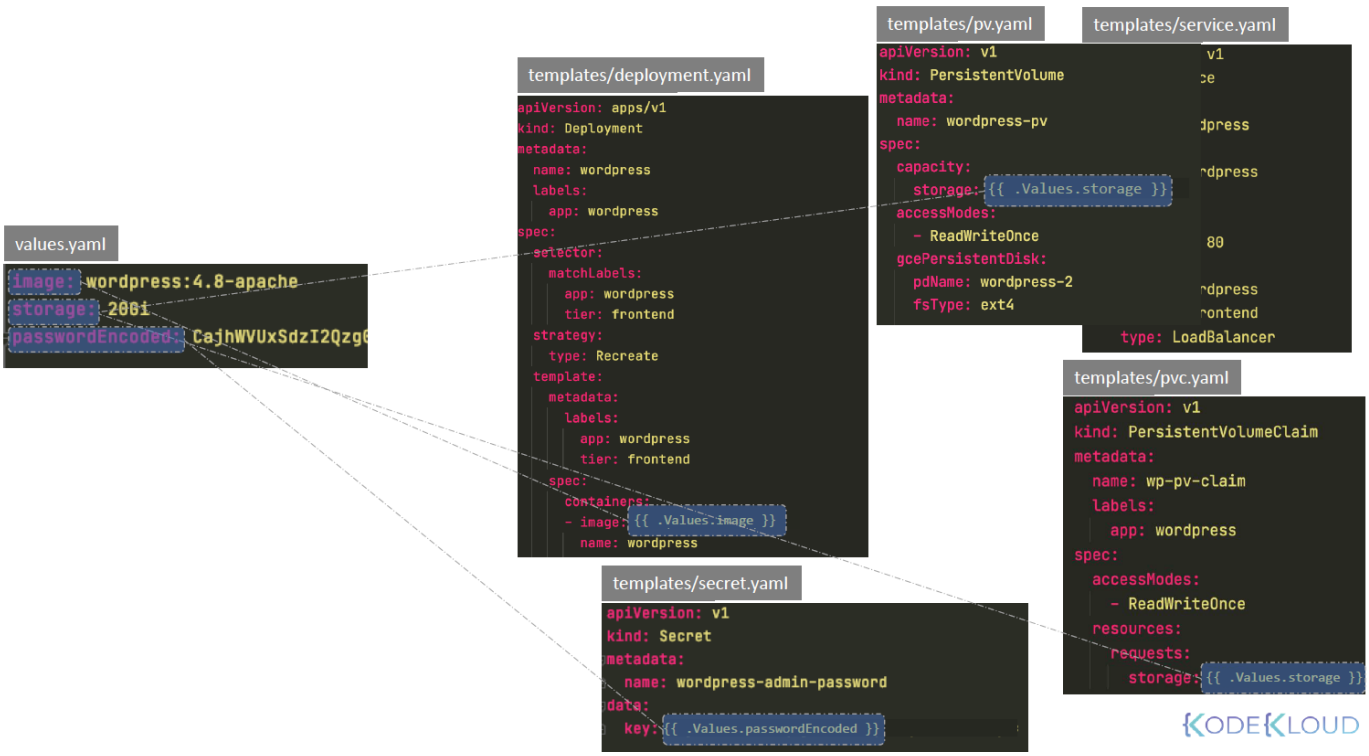
https://helm.sh/docs/intro/install/

For all OS:

https://helm.sh/docs/intro/install/

Use Helm

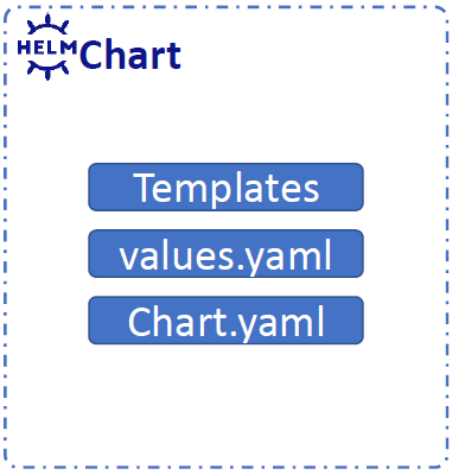
Helm use variabilized yaml files instead of classic ones to deploy applications. Those files are called templates. The values of those variables will be written in a values.yaml file.



A simple Helm Chart will contain all the templates needed for the desired app to run, the values.yaml file and a Chart.yaml file containing all the information about the Chart (contact, source, name and version of the chart, description, keywords…):

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
```

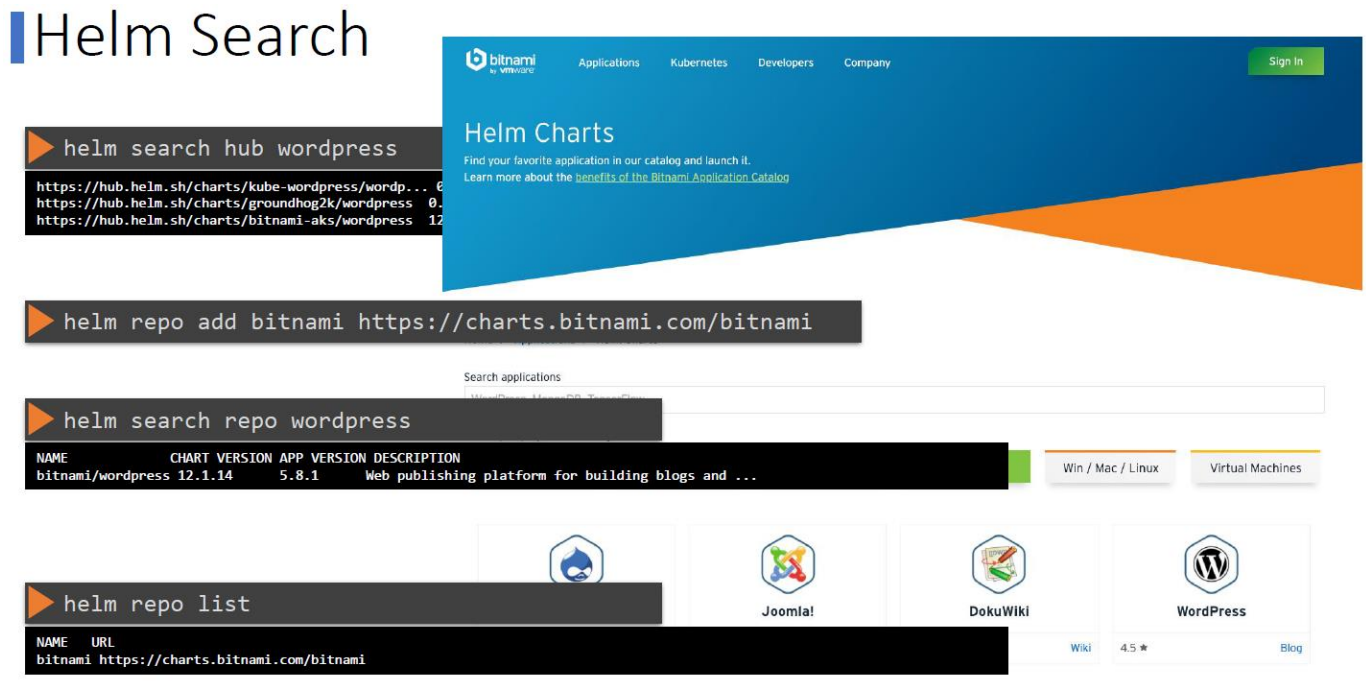


You can find desired Charts on:

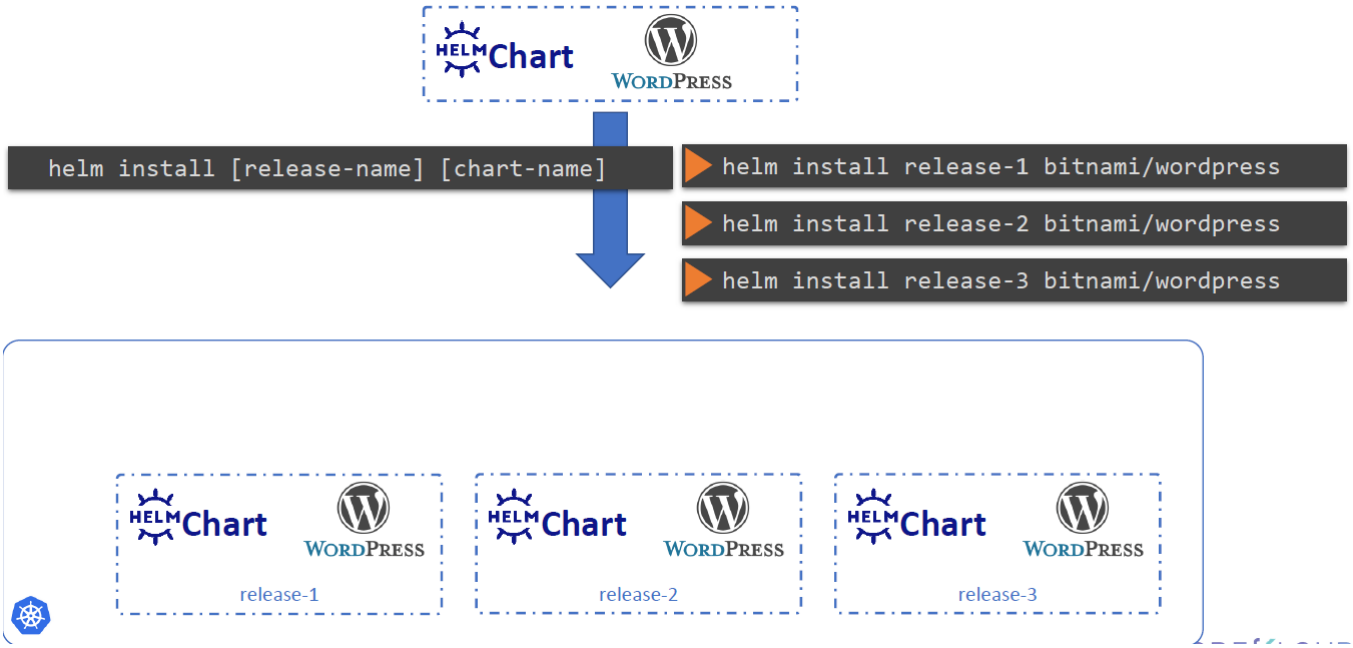
https://artifacthub.io/

OR, you can use the “helm search hub” command to search the desired Chart.

You can add repositories (like mirrors in package managers), to extend your Charts search, for example the bitnami repository:



To install an helm chart package use helm install. You have to precise a release-name for each package you install, for example if you install several time the same package, each release-name must be different (separate apps, a little bit like docker containers needs different names). Each release is completely independent from the others.



More commands:

°List helm packages that you currently have:

```
helm list
```

```
NAME        NAMESPACE   REVISION   UPDATED                                   STATUS     CHART                APP VERSION
my-release  default     1          2021-05-30 09:52:38.33818569 -0400 EDT    deployed   wordpress-11.0.12    5.7.2
```

°Uninstall a release:

```
helm uninstall my-release
```

°Pull a chart but don't install it (the --untar will extract the package and give you access to Chart files):

```
helm pull --untarbitnami/wordpress
```

°List the files of a chart:

```
ls wordpress
```

```
Chart.lock    README.md    ci           values.schema.json
Chart.yaml    charts       templates    values.yaml
```

°Once you have a chart installed on your computer or if you find it on an external repo (like docker), use helm install with the release name of your choice:

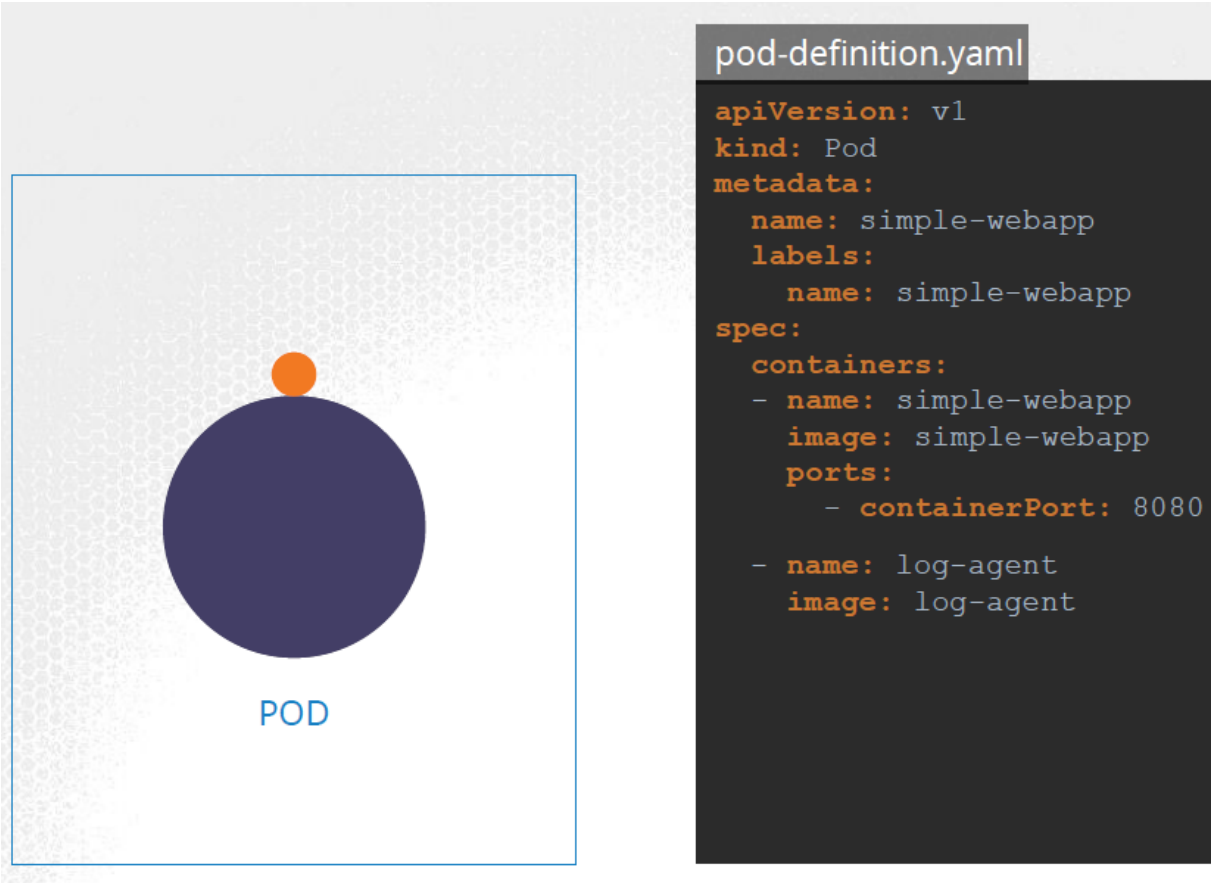
```
helm install release-4 ./wordpress
```

Multi container Pods

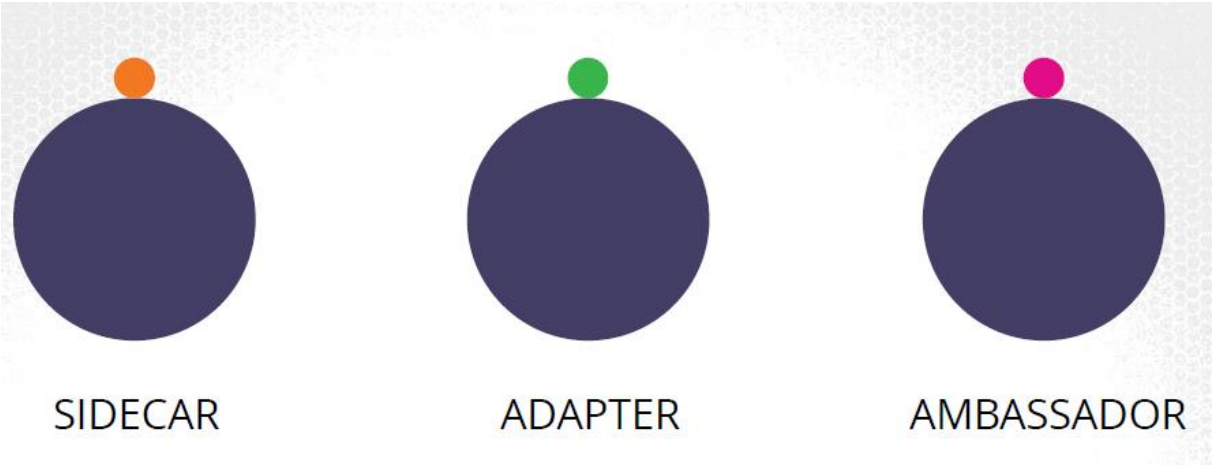
Sometimes you will need to create multi container Pods.

The containers in a multi container Pod 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.

You can have for example a container running one application and a container running the logging agent of that app.

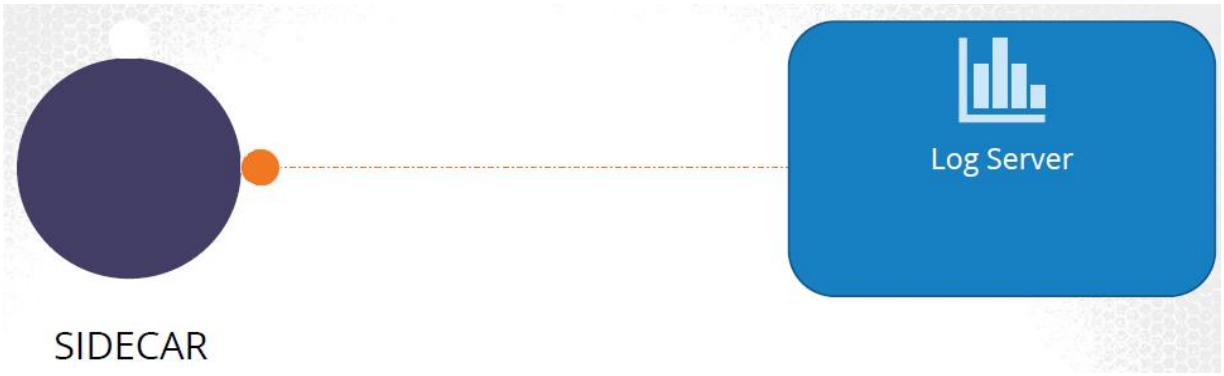


Three design patterns exists for multi container Pods:



Sidecar

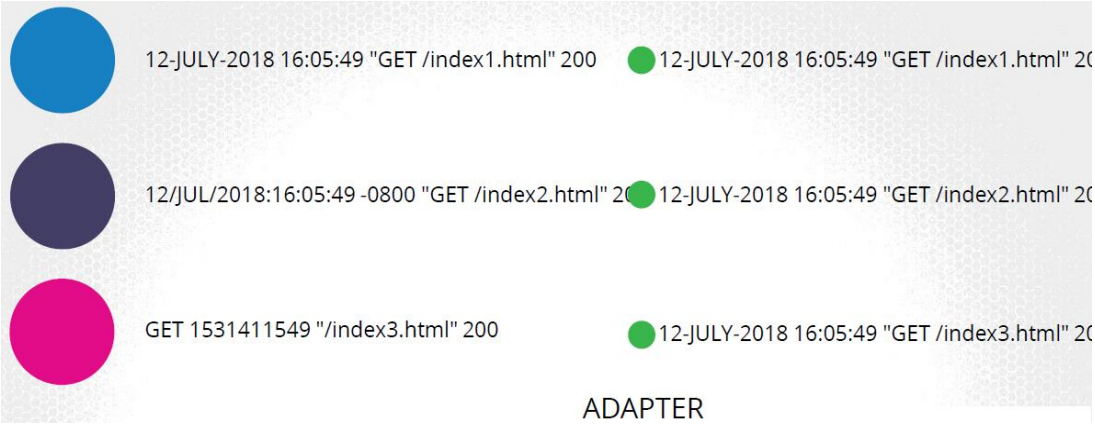
A good example of a side car pattern is deploying a logging agent along side a web server to collect logs and forward them to a central log server.



Adapter

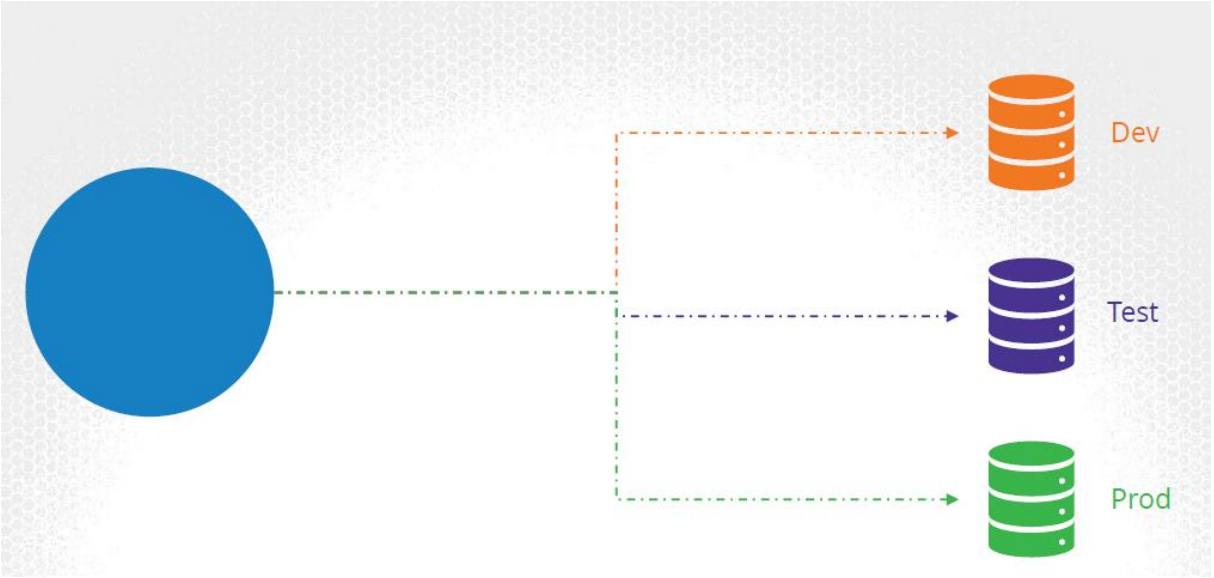
Building on the previous example, say we have multiple applications generating logs in different formats. It would be hard to process the various formats on the central logging server.

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.

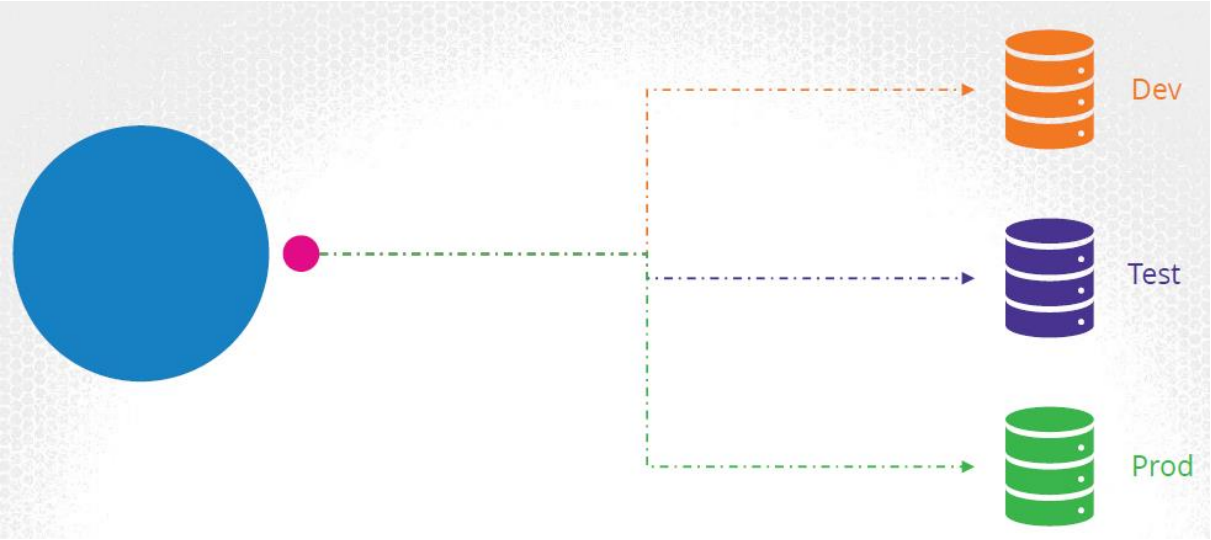


Ambassador

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.



Again, remember that these are different patterns in designing a multi-container pod. When it comes to implementing them using a pod-definition file, it is always the same. You simply have multiple containers within the pod definition file.

Init Containers

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. For example in the multi-container pod that we talked about earlier that has a web application and logging agent, both the containers are expected to stay alive at all times. The process running in the log agent container is expected to stay alive as long as the web application is running. If any of them fails, the POD restarts.

But at times you may want to run a process that runs to completion in a container. For example a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only one time when the pod is first created. Or a process that waits for an external service or database to be up before the actual application starts. That's where **initContainers** comes in.

An **initContainer** is configured in a pod like all other containers, except that it is specified inside a `initContainers` section, like this:

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: myapp-pod
5.   labels:
6.     app: myapp
7. spec:
8.   containers:
9.   - name: myapp-container
10.    image: busybox:1.28
11.    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
12.  initContainers:
13.  - name: init-myservice
14.    image: busybox
15.    command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-
   application> ;']
```

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts.

You can configure multiple such initContainers as well, like how we did for multi-pod containers. In that case each init container is run **one at a time in sequential order**.

If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds.

```
1. apiVersion: v1
2. kind: Pod
3. metadata:
4.   name: myapp-pod
5.   labels:
6.     app: myapp
7. spec:
8.   containers:
9.   - name: myapp-container
10.    image: busybox:1.28
11.    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
12.  initContainers:
13.  - name: init-myservice
14.    image: busybox:1.28
15.    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for
   myservice; sleep 2; done;']
16.  - name: init-mydb
17.    image: busybox:1.28
18.    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2;
   done;']
```

Read more about initContainers here.

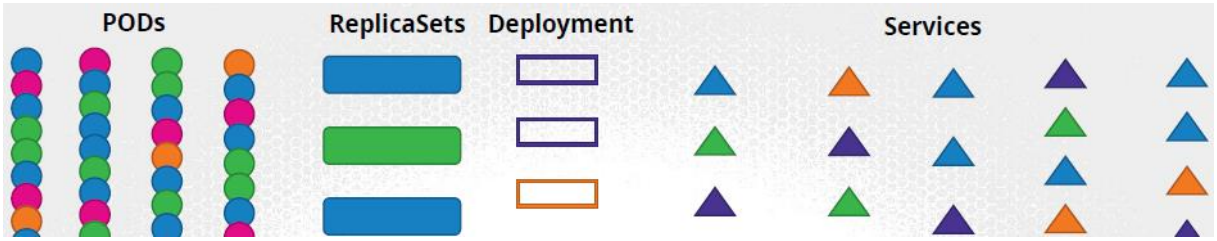
https://kubernetes.io/docs/concepts/workloads/pods/init-containers/

Pod design

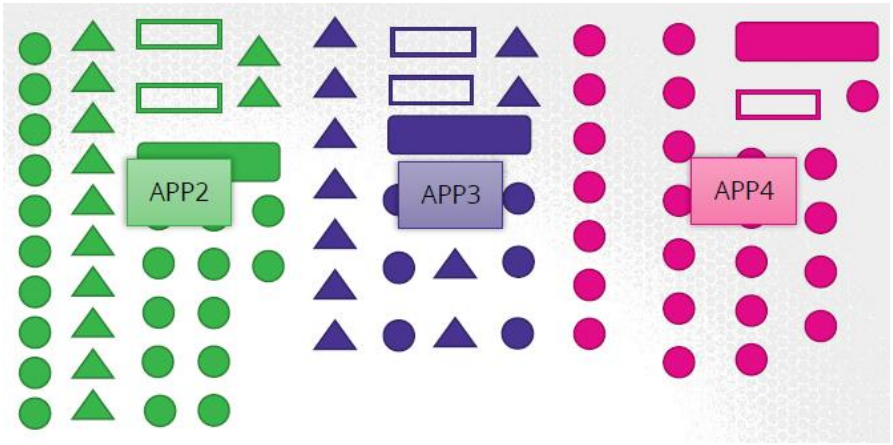
Labels, Selectors and Annotations

So how are labels and selectors used in Kubernetes? We have created a lot of different types of Objects in Kuberentes. Pods, Services, ReplicaSetsand Deployments. For Kubernetes, all of these are different objects. Over time you may end up having 100s and 1000s of these objects in your cluster. Then you will need a way to filter and view different objects by different categories.

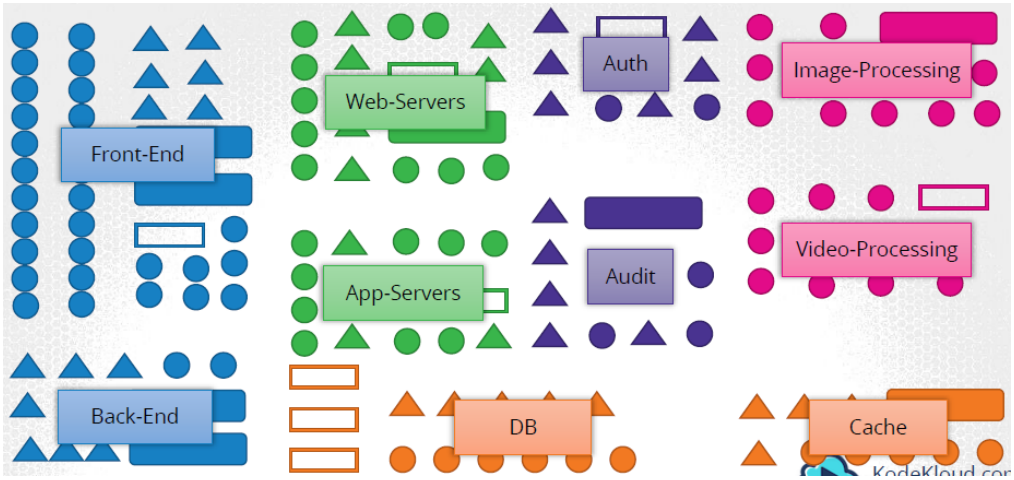
Such as to group objects by their type.



Or view objects by application.



Or by their functionality. Whatever it may be, you can group and select objects using labels and selectors.



For each object attach labels as per your needs, like app, function etc.

Then while selecting, specify a condition to filter specific objects. For example app == App1.

So how exactly do you specify labels in kubernetes. In a pod-definition file, under metadata, create a section called labels. Under that add the labels in a key value format like this. You can add as many labels as you like.

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
      app: App1
      function: Front-end
```

Once the pod is created, to select the pod with the labels use the kubectl get pods command along with the selector option, and specify the condition like app=App1.

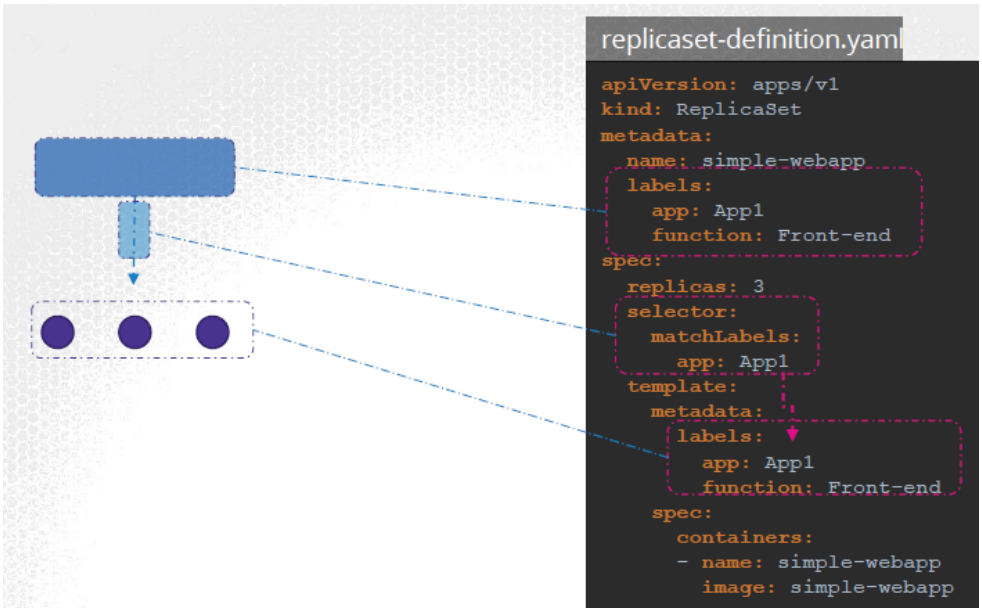
```
kubectl get pods --selector app=App1
NAME                READY     STATUS      RESTARTS   AGE
simple-webapp       0/1       Completed   0          1d
```

To get all the objects labelled the same way :

```
kubectl get all --selector app=App1,tier=front-end
```

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 not really concerned about that for now, because we are trying to get the replicaset to discover the pods. The labels on the replicaset will be used if you were configuring some other object to discover the replicaset. In order to connect the replica set to the pods, we configure the selector field under the replicaset specification to match the labels defined on the

pod. A single label will do if it matches correctly. However if you feel there could be other pods with that same label but with a different function, then you could specify both the labels to ensure the right pods are discovered by the replicaset.



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 file.

service-definition.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: App1
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
```

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 informatory purpose. For example tool details like name, version build information etcor contact details, phone numbers, email ids etc, that may be used for some kind of integration purpose.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: simple-webapp
  labels:
    app: App1
    function: Front-end
  annotations:
      buildversion: 1.34
spec:
```

Rolling Updates & Rollbacks in Deployments

See deployment part to have more info about RollingUpdate strategy and about the useful commands.

Here are some handy examples related to updating a Kubernetes Deployment:

- **Creating a deployment, checking the rollout status and history:**

In the example below, we will first **create** a simple deployment and inspect the **rollout status** and the **rollout history**:

```
master $ kubectl create deployment nginx --image=nginx:1.16
deployment.apps/nginx created

master $ kubectl rollout status deployment nginx
Waiting for deployment "nginx" rollout to finish: 0 of 1 updated replicas are available...
deployment "nginx" successfully rolled out

master $


master $ kubectl rollout history deployment nginx
deployment.extensions/nginx
REVISION CHANGE-CAUSE
1     <none>

master $
```

- **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**:

```
master $ kubectl rollout history deployment nginx --revision=1
deployment.extensions/nginx with revision #1

Pod Template:
 Labels:    app=nginx    pod-template-hash=6454457cdb
 Containers:  nginx:  Image:   nginx:1.16
  Port:    <none>
  Host Port: <none>
  Environment:    <none>
```

```
10.  Mounts:   <none>
11. Volumes:   <none>
12.master $
```

- **Using the --record flag:**

You would have noticed that the "**change-cause**" field is empty in the rollout history output. We can use the **--record flag** to save the command used to create/update a deployment against the revision number.

```
1. master $ kubectl set image deployment nginx nginx=nginx:1.17 --record
2. deployment.extensions/nginx image updated
3. master $master $
4.
5. master $ kubectl rollout history deployment nginx
6. deployment.extensions/nginx
7.
8. REVISION CHANGE-CAUSE
9. 1     <none>
10.2     kubectl set image deployment nginx nginx=nginx:1.17 --record=true
11.master $
```

You can now see that the **change-cause** is recorded for the revision 2 of this deployment.

Let's make some more changes. In the example below, we are editing the deployment and changing the image from **nginx:1.17** to **nginx:latest** while making use of the --record flag.

```
1. master $ kubectl edit deployments. nginx --record
2. deployment.extensions/nginx edited
3.
4. master $ kubectl rollout history deployment nginx
5. REVISION CHANGE-CAUSE
6. 1     <none>
7. 2     kubectl set image deployment nginx nginx=nginx:1.17 --record=true
8. 3     kubectl edit deployments. nginx --record=true
9.
10.
11.
12.master $ kubectl rollout history deployment nginx --revision=3
13.deployment.extensions/nginx with revision #3
14.
15.Pod Template: Labels:    app=nginx
16.    pod-template-hash=df6487dc Annotations: kubernetes.io/change-cause: kubectl
   edit deployments. nginx --record=true
17.
18. Containers:
19.  nginx:
20.  Image:   nginx:latest
```

```
    Port:    <none>
    Host Port: <none>
    Environment:    <none>
    Mounts:    <none>
  Volumes:    <none>

master $
```

- **Undo a change:**

Lets now rollback to the previous revision:

```
controlplane $ kubectl rollout history deployment nginx
deployment.apps/nginx
REVISION  CHANGE-CAUSE
1         <none>
3         kubectl edit deployments.apps nginx --record=true
4         kubectl set image deployment nginx nginx=nginx:1.17 --record=true


controlplane $ kubectl rollout history deployment nginx --revision=3
deployment.apps/nginx with revision #3
Pod Template:
  Labels:       app=nginx
        pod-template-hash=787f54657b
  Annotations:  kubernetes.io/change-cause: kubectl edit deployments.apps nginx --
  record=true
  Containers:
   nginx:
    Image:      nginx:latest
    Port:      <none>
    Host Port:  <none>
    Environment: <none>
    Mounts:     <none>
  Volumes:

controlplane $ kubectl describe deployments. nginx | grep -i image:
    Image:        nginx:1.17

controlplane $
```

With this, we have rolled back to the previous version of the deployment with the **image = nginx:1.17.**

```
controlplane $ kubectl rollout history deployment nginx --revision=1
deployment.apps/nginx with revision #1
Pod Template:
  Labels:       app=nginx
        pod-template-hash=78449c65d4
  Containers:
   nginx:
```

```
8.      Image:      nginx:1.16
9.      Port:       <none>
10.     Host Port:  <none>
11.     Environment: <none>
12.     Mounts:     <none>
13.   Volumes:
14.
15. controlplane $ kubectl rollout undo deployment nginx --to-revision=1
16. deployment.apps/nginx rolled back
```

To rollback to specific revision we will use the `--to-revision` flag.
With `--to-revision=1`, it will be rolled back with the first image we used to create a deployment as we can see in the `rollout history` output.

```
1. controlplane $ kubectl describe deployments. nginx | grep -i image:
2. Image: nginx:1.16
```

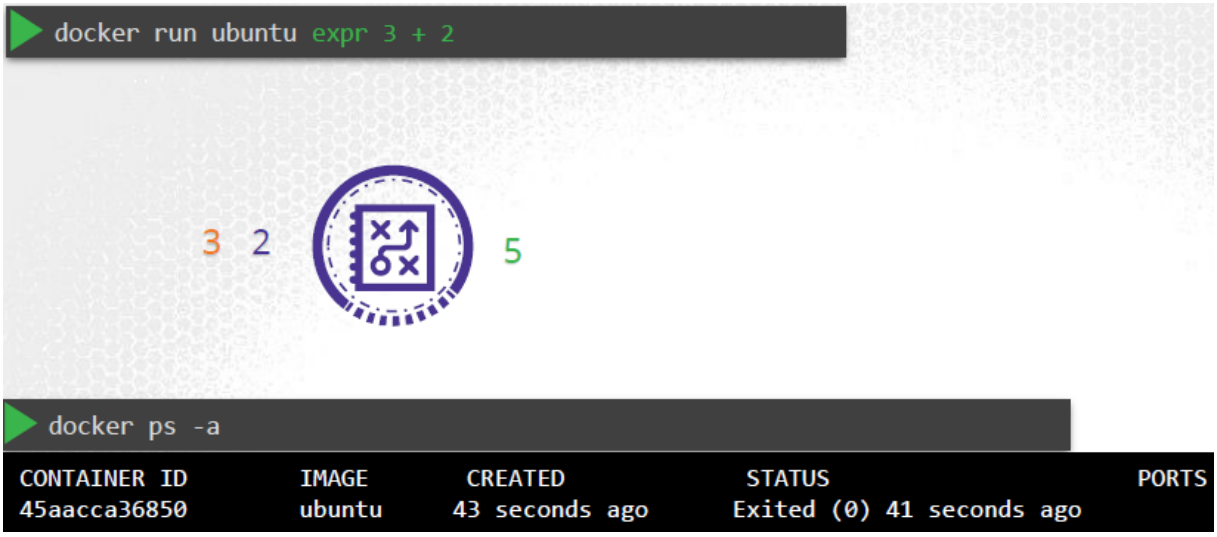
Jobs

There are different types of workloads that a container can serve. A few that we have seen through this course are Web, application and database. We have deployed simple web servers that serve users. These workloads are meant to continue to run for a long period of time, until manually taken down. There are other kinds of workloads such as batch processing, analytics or reporting that are meant to carry out a specific task and then finish.

For example, performing a computation, processing an image, performing some kind of analytics on a large data set, generating a report and sending an email etc. These are workloads that are meant to live for a short period of time, perform a set of tasks and then finish.

Single Pod Job

Let us first see how such a workload works in Docker and then we will relate the same concept to Kubernetes. So I am going to run a docker container to perform a simple math operation. To add two numbers. The docker container comes up, performs the requested operation, prints the output and exits. When you run the docker pscommand, you see the container in an exited state. The return code of the operation performed is shown in the bracket as well. In this case since the task was completed successfully, the return code is zero.



Let us replicate the same with Kubernetes. We will create a pod definition to perform the same operation. When the pod is created, it runs a container performs the computation task and exits and the pod goes into a Completed state. But, It then recreates the container in an attempt to leave it running. Again the container performs the required computation task and exits. And kubernetesbrings it up again. And this continuous to happen until a threshold is reached. So why does that happen?

Kubernetes wants your applications to live forever. The default behavior of PODs is to attempt to restart the container in an effort to keep it running. This behavior is defined by the property restartPolicyset on the POD, which is by default set to Always. And that is why the POD ALWAYS recreates the container when it exits. You can override this behavior by setting this property to Never or OnFailure. That way Kubernetes does not restart the container once the job is finished. Now, that works just fine.

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
```

We have new use cases for batch processing. We have large data sets that requires multiple pods to process the data in parallel. 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. But we have learned about ReplicaSetshelping us creating multiple PODs. While a ReplicaSetis 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.:

We create a JOB using a definition file. So we will start with a pod definition file. To create a job using it, we start with the blank template that has apiVersion, kind, metadata and spec. The apiVersionis batch/v1 as of today. But remember to verify this against the version of Kubernetes release that you are running on. The kind is Job of course. We will name it math-add-job. Then under the spec section, just like in replicasetsor deployments, we have template. And under template we move all of the content from pod definition specification.

job-definition.yaml

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: math-add-job
spec:
  template:
    spec:
      containers:
        - name: math-add
          image: ubuntu
          command: ['expr', '3', '+', '2']

      restartPolicy: Never
```

```
kubectl create -f job-definition.yaml
```

```
kubectl get jobs
NAME           DESIRED   SUCCESSFUL   AGE
math-add-job   1         1            38s
```

```
kubectl get pods
NAME                 READY   STATUS      RESTARTS   AGE
math-add-job-l87pn   0/1     Completed   0          2m
```

```
kubectl logs math-add-job-ld87pn
5
```

```
kubectl delete job math-add-job
job.batch "math-add-job" deleted
```

Once done create the job using the kubectl create command. Once created, use the kubectlget jobs command to see the newly created job. We now see that the job was created and was completed successfully. To see the pods created by the kubectlget pods command you run kubectlget pods command. We see that it is in a completed state with 0 Restarts, indicating that kubernetes did not try to restart the pod. Perfect! But, what about the output of the job? In our

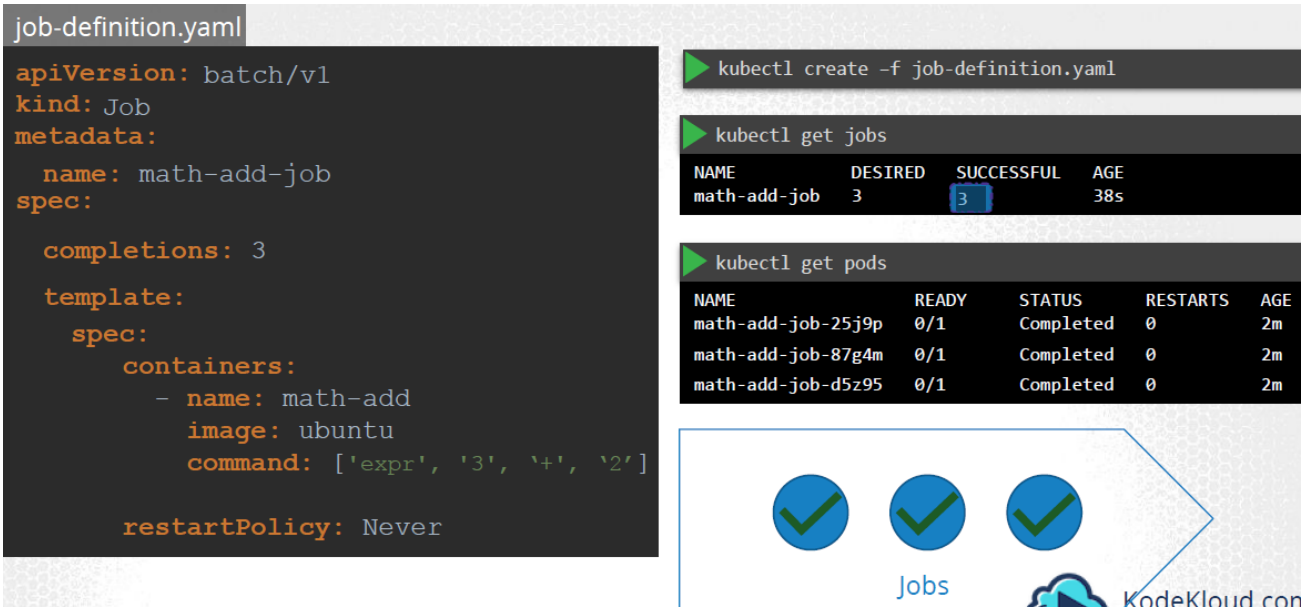
case, we just had the addition performed on the command line inside the container. So the output should be in the pods standard output. The standard output of a container can be seen using the logs command. So we run the kubectllogs command with the name of the pod to see the output. Finally, to delete the job, run the kubecl delete job command. Deleting the job will also result in deleting the pods that were created by the job.

backoffLimit: The backoffLimit specifies how many times a pod can fail and restart before the job is considered failed

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl:5.34.0
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

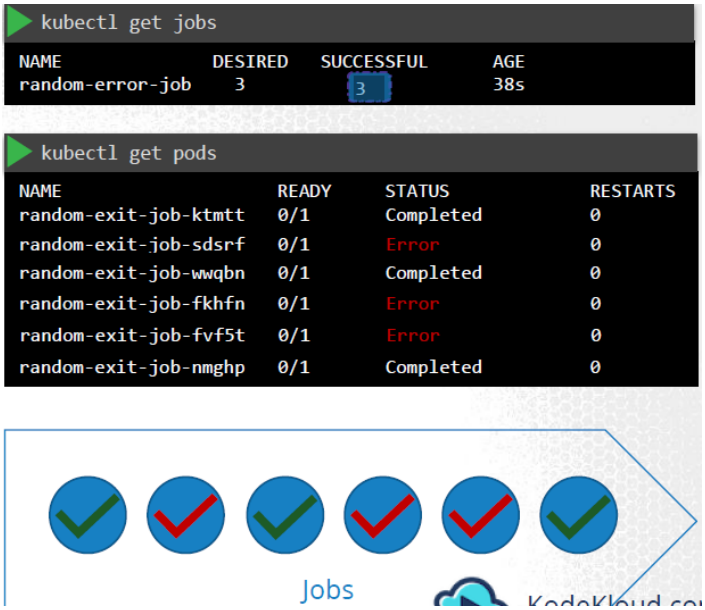
Multiple Pods Job

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. This time, when we create the job, We see the Desired count is 3, and the successful count is 0. Now, by default, the PODs are created one after the other. The second pod is created only after the first is finished.

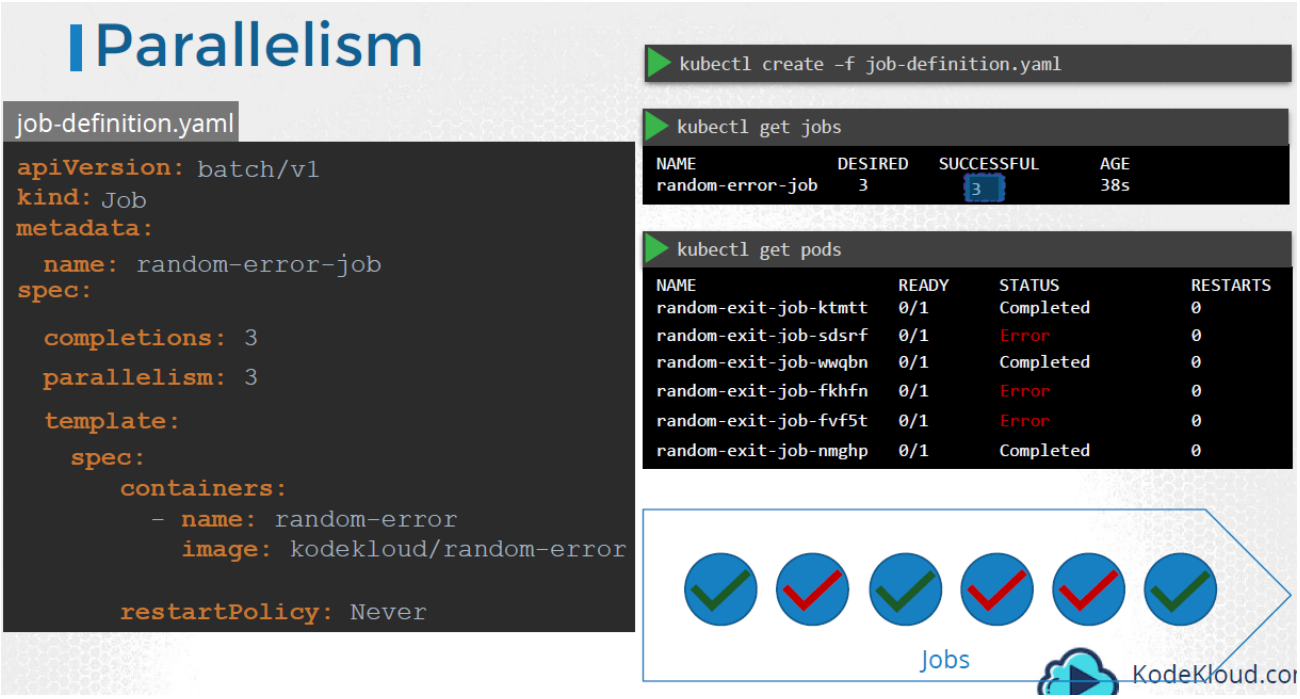


That was straight forward. But what if the pods fail? For example, I am now going to create a job using a different image called random-error. It's a simple docker image that randomly completes or fails. When I create this job, first pod completes successfully, the second one fails, so a third one is 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.



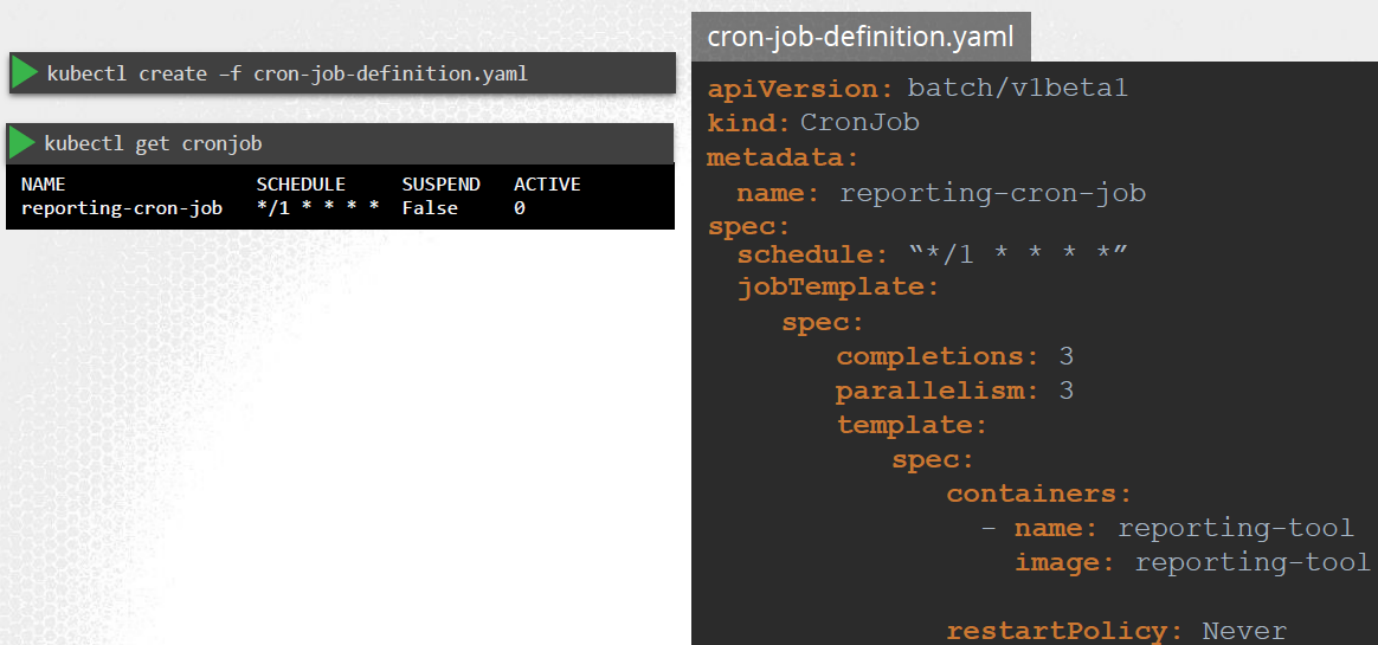
Instead of getting the pods created sequentially we can get them created in parallel. For this add a property called parallelism to the job specification. We set it to 3 to create 3 pods in parallel. So the job first creates 3 pods at once. Two of which completes successfully. So we only need one more, so it's intelligent enough to create one pod at a time until we get a total of 3 completed pods.



CronJobs

A cronjob is a job that can be scheduled. Just like crontab in Linux, if you are familiar with it. Say for example you have a job that generates a report and sends an email. You can create the job

using the kubectlcreate command, but it runs instantly. Instead you could create a cronjob to schedule and run it periodically. To create a cronjob we start with a blank template. The apiVersionas of today is batch/v1beta1. The kind is CronJobwith a capital C and J. I will name it reporting-cron-job. Under spec you specify a schedule. The schedule option takes a cronlike 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.



Once the file is ready run the kubectl create command to create the cron-job and run the kubectl get cronjob command to see the newly created job. It would in turn create the required jobs and pods.

Micro services architecture example

Example for Docker ecosystem

For a voting app let's use 5 containers :

- Front of the voting app allowing users to vote
- In-memory redis database storing the answers
- A worker container which will get the votes from the redis container and save them permanently into a postgres sql db
- Postgresql container
- Front of the result app displaying the results using the postgres container

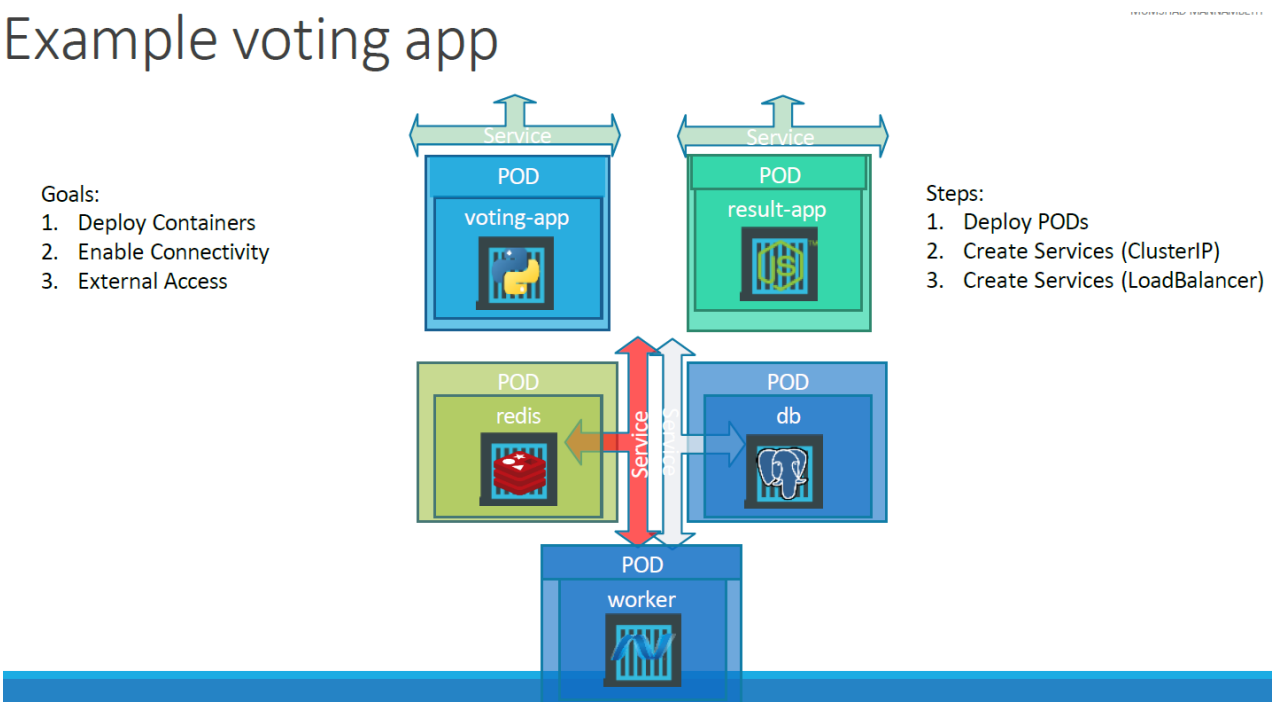
We will have to launch a serie of 5 docker commands :

```
docker run –d -–name=redis redis

docker run –d -–name=db postgres:9.4

docker run –d -–name=vote –p 5000:80 –-link redis:redis voting-app

docker run –d -–name=result –p 5001:80 –-link db:db voting-app

docker run –d –-name=worker –-link db:db –link redis:redis worker
```

Here the -–link are used to link the different containers thanks to their name, for example the vote application is expecting to have a redis host usable to contact (here the name will create a line in the hosts file associating the ip of the redis container with the redis hostname). The format is defined-hostname:name-of-the-expected-variable-in-the-container.

Example for K8s ecosystem

Let's do the same thing for k8s ecosystem :



Example of yaml files for the voting app deployment, the redis deployment and their respective services.

Keep in mind that the different pods are using the services name that we define to communicate between them, here in the code of the voting app, the redis host is contacted on the internal network, this redis host is defined with the name of the redis ClusterIP service.

Link with all the files :

[GitHub - kodekloudhub/example-voting-app-kubernetes]

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: voting-app-deployment
  labels:
    app: demo-voting-app
    name: voting-app-deployment

spec:
  replicas: 3
  selector:
    matchLabels:
      name: voting-app-pod
      app: demo-voting-app
  template:
    metadata:
      name: voting-app-pod
      labels:
        name: voting-app-pod
        app: demo-voting-app

    spec:
      containers:
        - name: voting-app
          image: bretfisher/examplevotingapp_vote
          ports:
            - containerPort: 80
apiVersion: v1
kind: Service
```

```
metadata:
  name: voting-app
  labels:
    name: voting-app-service
    app: demo-voting-app
spec:
  type: NodePort
  selector:
    name: voting-app-pod
    app: demo-voting-app
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30004
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-deployment
  labels:
    app: demo-voting-app
    name: redis-deployment

spec:
  replicas: 3
  selector:
    matchLabels:
      name: redis-pod
      app: demo-voting-app
```

```
  template:
    metadata:
      name: redis-pod
      labels:
        name: redis-pod
        app: demo-voting-app

    spec:
      containers:
        - name: redis
          image: redis
          ports:
            - containerPort: 6379
```

```
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    name: redis-service
    app: demo-voting-app
spec:
  type: ClusterIP
  selector:
    name: redis-pod
    app: demo-voting-app
  ports:
    - port: 6379
      targetPort: 6379
```

Prerequisites

To pass your Kubernetes Certified Application Developer, you will need to have a basic knowledge of building docker images to use them on Kubernetes. CF Docker course.

Resources for training

https://github.com/dgkanatsios/CKAD-exercises

https://github.com/lucassha/CKAD-resources