**Kubernetes**

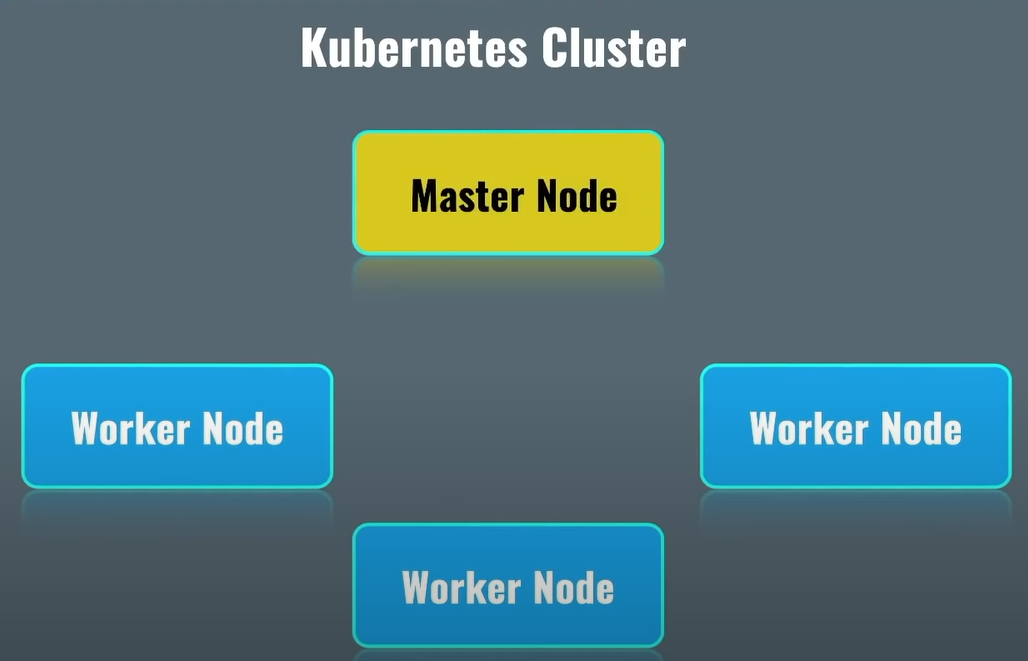
Kubernetes, also known as K8s, is an open source system for managing containerized applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications. Your app is never down.

The smallest unit of K8 is the pod. It is the abstraction over container. It creates a running environment for containers. We can run multiple apps inside one pod.

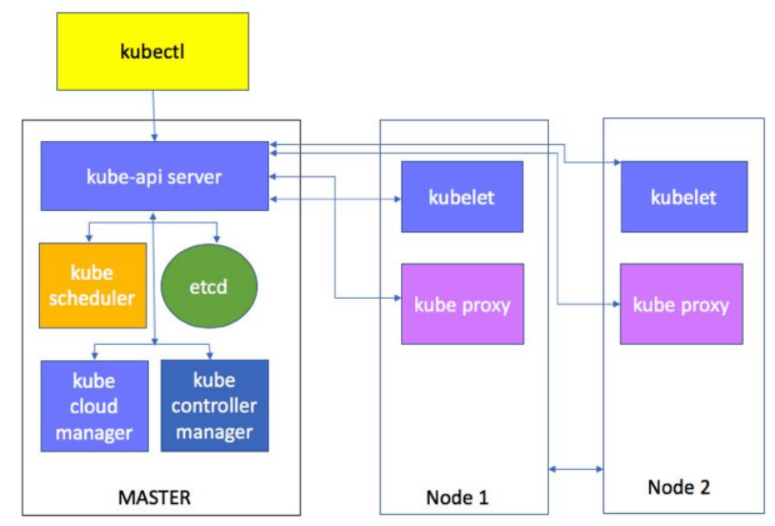
K8 offers a virtual network so each pod has its own IP address. So each pod can communicate with each other using their ips.

If a container inside a pod dies, a new one will be created and new ip address will be recreated for the pod which is inconvenient. Because of regular change of an ip address of the pod it is difficult to communicate with pods using their ips since they change.

Because of this another componenet is introduced, namely Service. Service is basically a permanent ip address that can be attached to each pod. The lifecycle of service and pod is not connected! So even if a pod dies the service and its ip address stay.



Master node or control plane manages worker nodes.



Kubectl (aka kube control) is a command line tool which allows us to connect to a specific kubernetes cluster and manage it remotely.

Every request is handled by Api server. So, for instance, if you wanted to get info from etcd key value base, then you would have to first send request to the api then, it would go to the etcd.

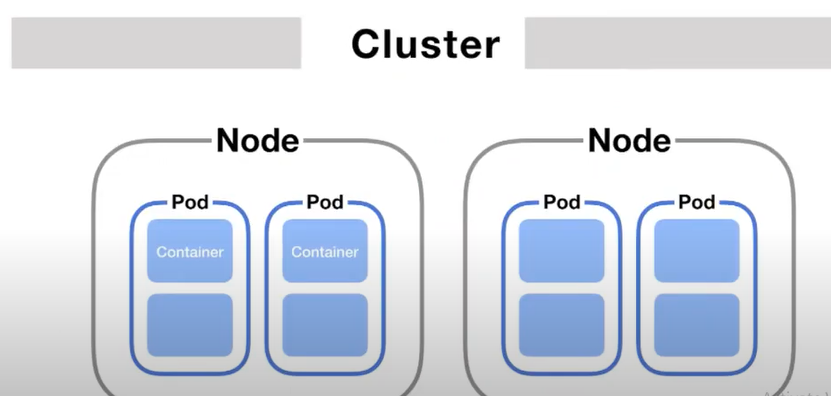
**minikube** is a tool that lets you run Kubernetes locally. minikube runs a single-node Kubernetes cluster on your personal computer (including Windows, macOS and Linux PCs) so that you can try out Kubernetes, or for daily development work.

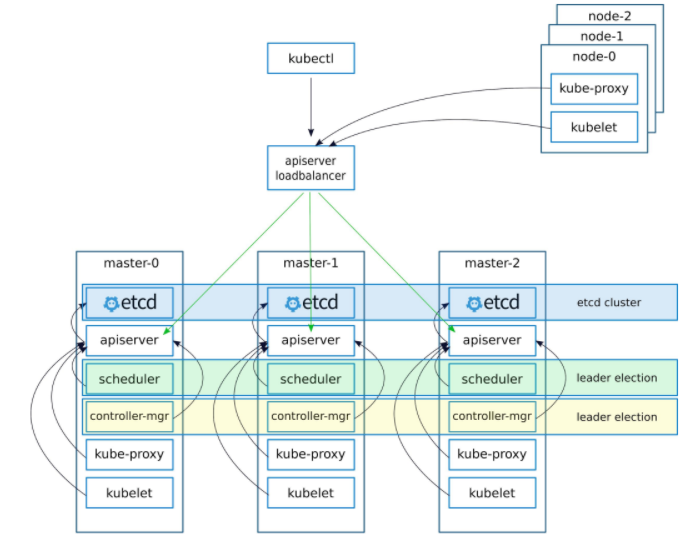
Kube-scheduler decides where a container has to rise. So it doesn’t raise anything but just decides. It decides where to run a container by analyzing nods’ resources (cpu and ram), if they have the most space then the container is decided to run in that node.

Kubelet sends information about node and containers, that are run under that node, to the etcd via kube-api-server. And when a container is decided to run on a node, that nodes’ kublet received command and sends this command to docker so that a container is realized.

Kube-proxy sends the received traffic to a particular container.

One master can die so that’s why it is better to have 3 master nodes. When there are more master nodes, one controller manager leader and one scheduler leader are selected.





**Common commands**

kubectl run podName --image imageName

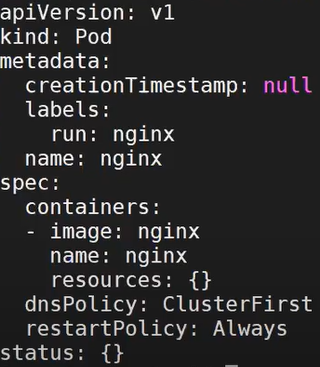
kubectl run nginx --image=nginx

**Applying common commands with one command**

To create a file that has all the specs to make our lives easier so that we don’t type commands, we use --dry-run=client –o yaml command to print the object that would be sent, without sending it🡺



If the command needs to run so that yaml file can be displayed, we use --dry-run option to imitate run command.



It is the manifest file of the pod. Then to apply this yaml file we use the following command 🡺

kubectl apply –f pod.yaml 🡪 -f is to specify filename

--dyr-run=client just runs the command without actually running it for real and when used together with “-o yaml”, then the all the specs to run the command are put in a yaml format and printed to the console.

-o format- output it in a specific format (json,yaml) 🡺 -o json

Dry run mode gives you the possibility of issuing a command without side effects for testing an actual command that you intend to run

-f option🡪 --filename=[]: that contains the configuration to apply,

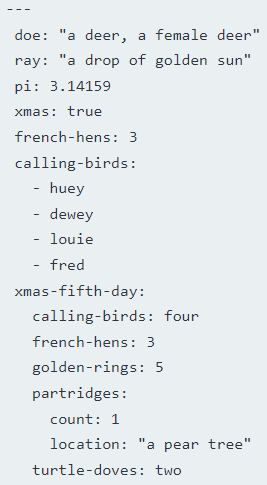
--force is true (force deletion).

**YAML FILE**

YAML is a data serialization language that is often used for writing configuration files. Depending on whom you ask, YAML stands for yet another markup language or YAML ain’t markup language (a recursive acronym), which emphasizes that YAML is for data, not documents.

It's often used as a format for configuration files, but its object serialization abilities make it a viable replacement for languages like JSON.

YAML uses Python-style indentation to indicate nesting. Tab characters are not allowed, so whitespaces are used instead. There are no usual format symbols, such as braces, square brackets, closing tags, or quotation marks. YAML files use a .yml or .yaml extension.



The file starts with three dashes. These dashes indicate the start of a new YAML document. Next, we see the construct that makes up most of a typical YAML document: a key-value pair.

**Kubectl logs**

kubec logs podName



This will print all the logs of the single container in that pod. If there are more containers in the pod, then the name of the container must be specified.

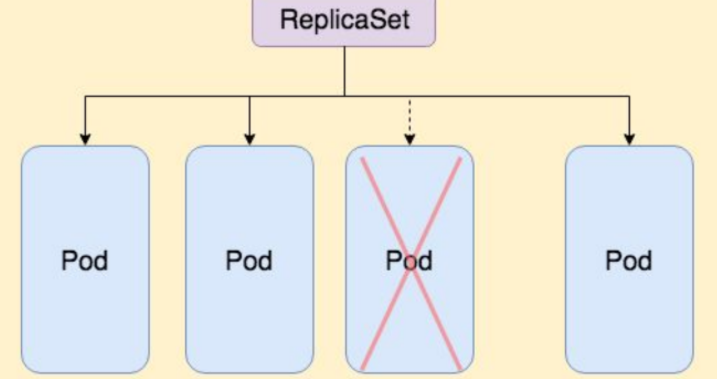
If we wanted to watch logs live then we use -f 🡺



**Raising deleted pods (Replicaset)**

If we delete a pod, pods are not raised again because the controller manager (main component) doesn’t manage any pods. Contoller manager manages replica sets. To actually automatically restart pods, we need replica sets.

Inside replica set we have pods. Pods in a replicaSet are all the same.



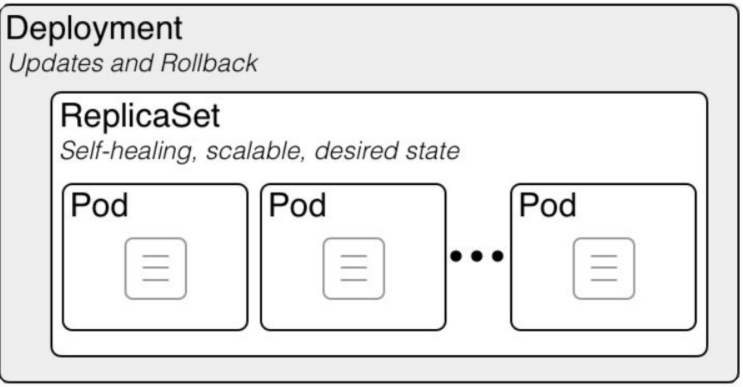
So we do like this 🡺 create a pod whose replicaSet is 3 and inside that pod raise this container/containers.

We can’t create replicaSet manually by simply using pod because they will not be able to be updated. We won’t be able to scale replicas, for instance (change the numbe of replicas). So we need something called deployment to be able to update.

**Deployment**

We create a deployment with a specific image and number of replicas for the replicaset.

If a pod dies inside the replicaSet then a new pod is started from the replicaSet.



So we basically create a deployment with a specific image and replicaSet number.



Now there will be 3 pods of the image nginx.

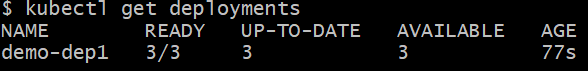
Now if we delete a pod, it will raise a new pod.

Basically, when we delete a pod, controller manager sees that the replica number is 3 but one pod died so now it is 2, so controller manager has to raise 1 more pod. It is called self-healing.

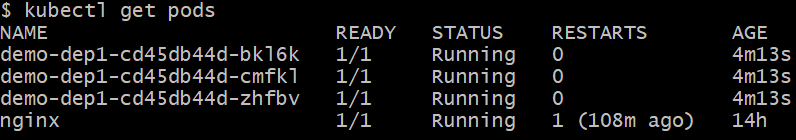
So controller manager always keeps however many pods we specified alive. So if we delete all 3 of the pods, it will just restart raise new 3 pods.

**Normally the maximum number of replicas is 3.**

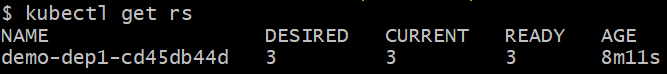
 or  to print all the deploys



Ready shows that there are 3 pods in this deployment so the number of replicas is 3.



The first part of the pod name is deployment name and the second part is the replicaSet 🡺



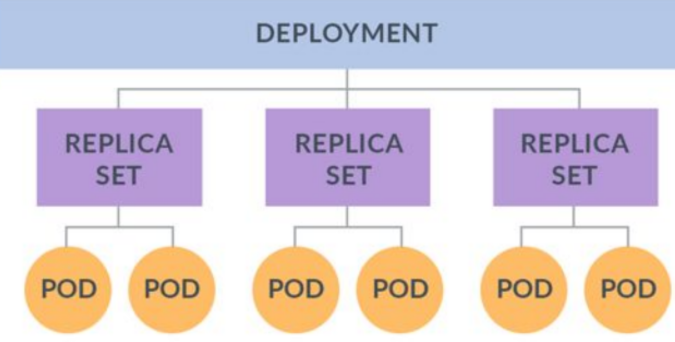
The third part is the unique hash to distinguish pods.

Now we can also scale replicaSet. 🡺



Now we increased the number of replicas.

This means in one deployment we can have multiple replicaSets. Controller manager manages replicaSets and deployment provides us updating replicaSets and having old replicaSets for rollback purposes.



Let’s create a manifest file of the deployment and use that file to create a specific deployment when needed.

 to get the manifest of the command in the yaml format. Then we can put everything from here to .yaml file 🡺



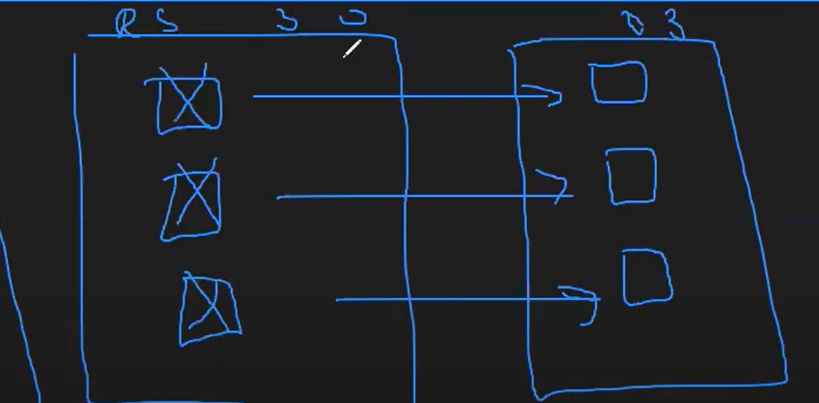


With deployment, we can set a new image for the pod meaning that we can update the deployment.



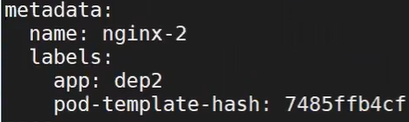
This will terminate the old rs and create a new replicaSet. The process goes like this 🡺

1. First a new replicaSet is created with no pods
2. Then a new pod is created in that replicaSet with the specified image.
3. When the pod is on ready status which means it is ready then one replica from the old replicaSet is terminated because controller manager tries to keep the number of replicas 3



The old replicaSet is not deleted for rollback purposes.

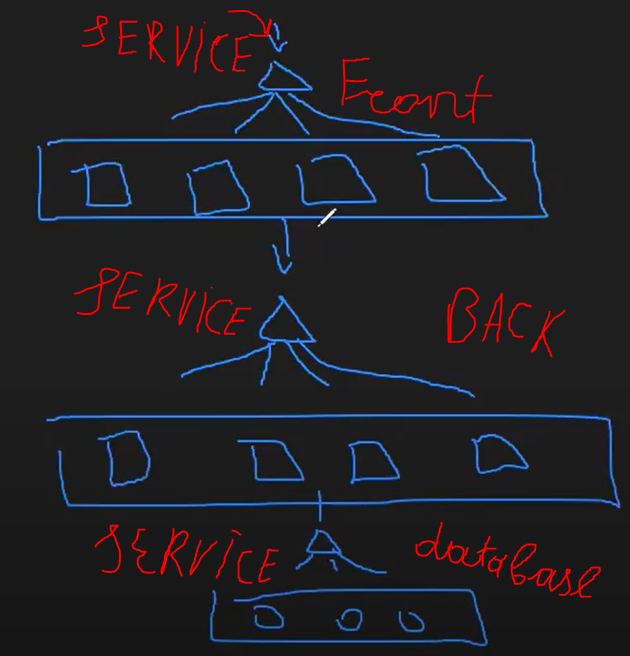
When a pod dies in a deployment, a new pod is raised for that deployment so that the number of pods is the same as the replicas number. We can have lots of deploys. So it figures out to which deploy it has to send this newly raised pod by the labels metadata of the pod 🡺

 it is the pods metadata when created. So it references dep2 with the specific hash so that it is not confused with another deployment. We are making sure that this newly raised pod is delivered where it needs to be delivered.

**Services**

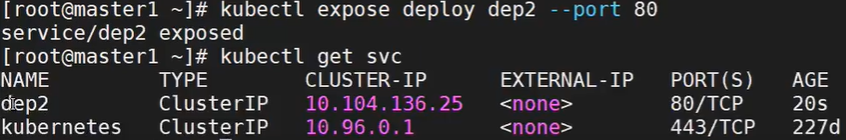
There are different types of services: ClusterIP, NodePort service

ClusterIp services work only inside kubernetes clusters.

As we know, pods ips are dynamic, meaning that the change. So we use services to communicate amon deployments🡺  


So here we have 3 deployments: front, back and some database. Front will send request to the database so it needs a service. BackEnd also needs another service to send a reqeust to the database.

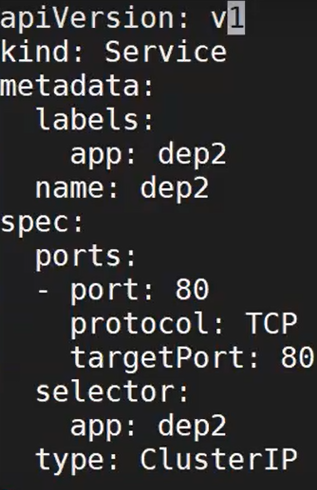
So we create a service by exposing something with a specific port🡺



kubernetes service is the api-server. These services type is ClusterIp.

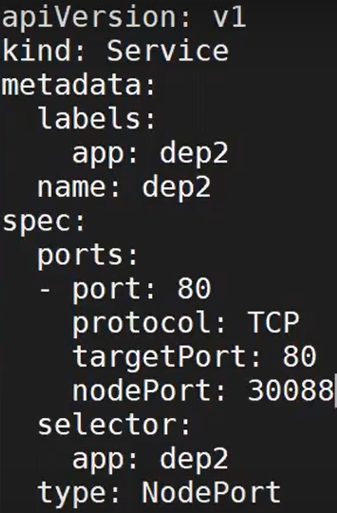
Here we say that we want to expose the dep2 deployment’s port 80.

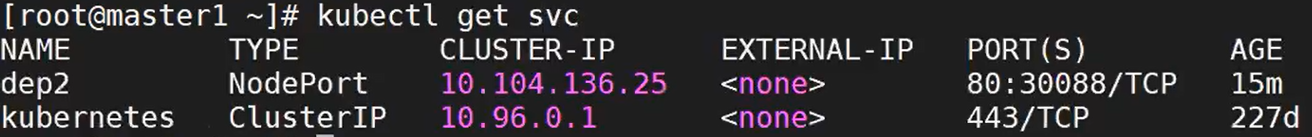
In ClusterIP service, we send a request to the service’s ip address and then it is forwarded to the specific pods that have the specified port (80 in our case).



Targetport is the port of the deployment pods. Port is the service’s port.

With **nodePort** service, we can send a request from a node. When we create a nodePort service, one port is specified for all the worker nodes to which if the request is sent, it forwards the request to the ClusterIP service which will then direct the request to the pod.

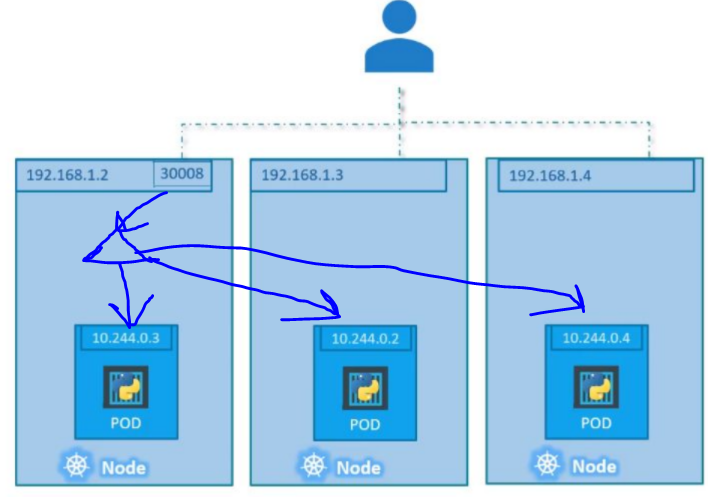




From the selector sections k8s knows which pod/deploy this service is bound to.

As we can see, if we send a request to the nodes’ 30088 port then it will be mapped to 80’s port of the ClusterIP service.

So nodeport inside itself has a ClusterIP svc. When we create a nodePort service, one port from 30\_000 range activated for all worker nodes and when it is sent to this port then it’ll forward the request to the ClusterIP service which will then direct the request to the pod.



That one svc inside the pod is the ClusterIP service and this entire thing is inside nodeport Service.

Now we send requests to worker nodes with the port 30008, for example, and after receiving this request from this port it will forward this request to the ClusterIP service which then forwards the request to the specific pod.

**Update deployment**

We can update any deployments image, for instance, by 3 means:

1. We can change the yaml file of the deploy and apply the modified yaml file by using the following command kubectl apply –f deploy.yaml
2. We can use set image command: kubectl set image deployName oldImageName:newImageName
3. The best way is to use edit command: kubectl edit deploy deployName and this will take us to the live yaml file of the deployment. If we make changes to this yaml file, they will be automatically applied.

**Namespace**

Namespaces help us to group our deployments, services. We can create a namespace with the following command🡺

kubectl create ns name

By default, if we get pods by this command: kubectl get pods It will get pods from the default namespace. If we want to get pods from a specific namespace then we need to specify it 🡺

kubectl get pods –n namespaceName.

With Namespaces, we can use the same yaml file which was used to create deployments and services again. Meaning that it will not mix anything. Because when we send a request to a service, for example, it will attach namespace name to it and it will know to where the request has to be sent (curl serviceAddress.Namespace).

kubectl get pods –A –o wide (-A or –all-namespaces)

This command, will get all pods with their namespaces and also some extra info (in which node the pods are running).

All main components of k8s (control manager, etcd, kube-proxy) are in the kube-system namespace. When we send a request, it always adds the name of the namespace at the end of the command (serviceIp.namespaceName).

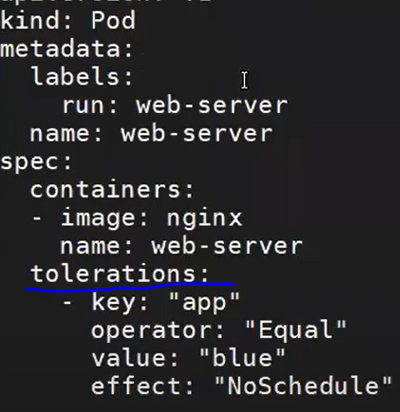
**Taint**

We can taint nodes to have some limits on them such as don’t schedule anything on this node. Each taint has an effect as well as some key value pair. For example, master node by default is tainted and has the effect of NoSchedule so that nothing (pod) can be run in that node. Taint prevents it from hapenning. Taint effects:

1. NoSchedule – when one pod is to be run, don’t schedule anything on this node. This effect applies to only new ones. So it doesn’t do anything to running pods in that node.
2. PreferNoSchedule – It will look for other possible nodes so that it can schedule there. If it can’t find any node to schedule a new pod, only then it can schedule this node.
3. NoExecute – this node cannot have any containers, pods. So even if there are running pods in the node, it will shut them all down.

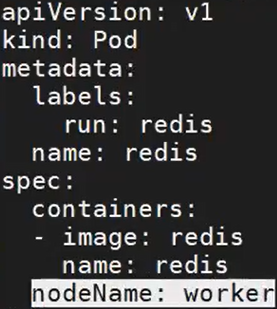
**Toleration**

We can use toleration to stands up to taints. In other words, if we have the same toleration as the taint then the pod can be raised in the node.

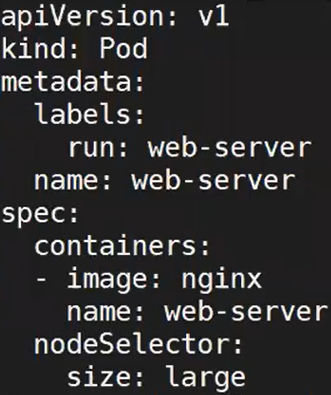


We can put some rules to pods with **operators**. For example, here it is “Equal” which means both the key value pair as well as the effect of the toleration must be equal to the ones of the taint in order for the pod to be able to be run in the node. There is also “Exists” which is like “or” operator.

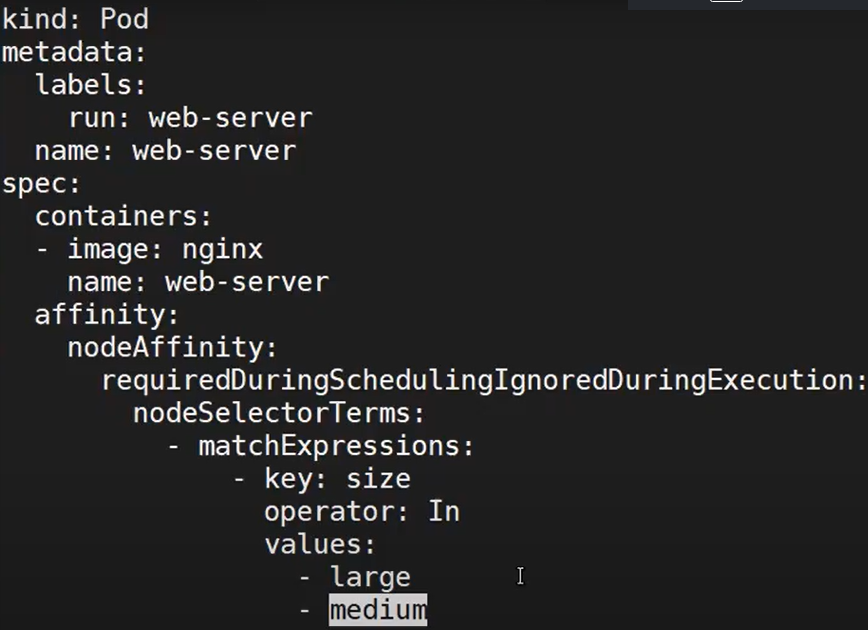
If we want a pod to run in a specific node, then we gotta specify the **nodeName** in the yaml file of the pod. Nodename is unique 🡺



There is also **nodeSelector** which is similar to normal selectors that have labels. Basically, nodeSelectors looks for the node that has the same key value pair🡺



If we have multiple statements to select a node then we can use **affinity**🡺



nodeAffinity in our case is requiredDuringScheduling IgnoredDuringExecution basically tells k8s, use this nodeSelectorTerms only when scheduling a new pod.

matchExpressions is similar to labels. This expression’s key is size and values are large and medium, the operator is “In”.

So this expression tells k8s that this pod can run in a pod that has a key “size” whose value is either large or medium. We can also have following expressions 🡺

