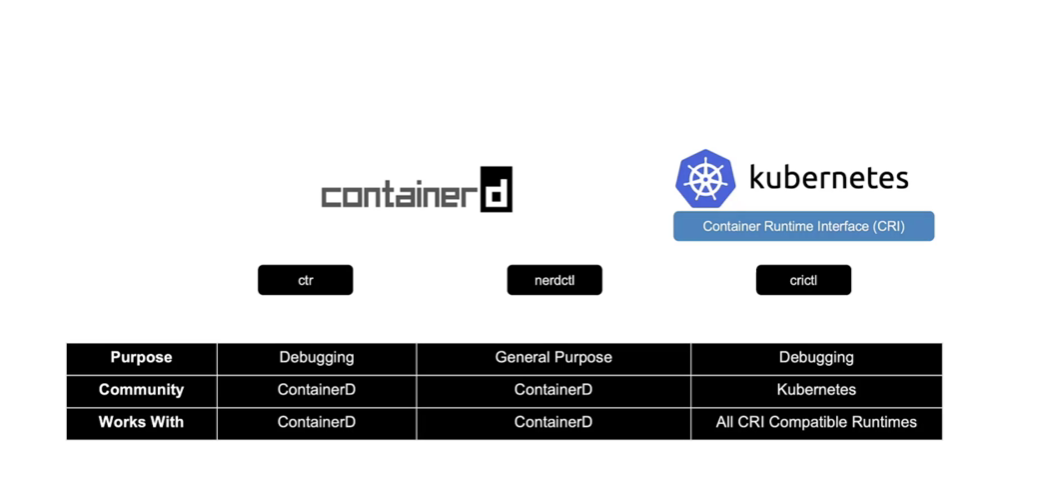
**Kubernetes**

* Master node:
  + ETCD cluster - Stores information about cluster.
  + kube-scheduler - Responsible for scheduling applications/containers on nodes.
  + Kube Controller Manager - Responsible for controlling node controller, replication controller.
  + kube-apiserver - Responsible for orchestrating all operations within the cluster.
* Worker node:
  + kubelet - Listens to the instructions from kube-apiserver and managers containers.
  + kube-proxy - Responsible for enabling communication between the services.

CLI tools:

* As support for docker itself was removed with the arrival of version 1.24, containerD was used to interact with Container Runtime Interface (CRI).
* This is so because initially Kubernetes was built solely to orchestrate docker images, but with time many other players like rkt wanted to join Kubernetes, so they introduced CRI.
* For a tool to work with CRI, it should follow the Open Container Initiative (OCI) standard.
* For containerD, you can either use ctr (used for debugging containerD, not user friendly) or nerdctl (recommended).
* For other tools, you can use crictl, but it is ideally not used to create containers. It is used only for debugging purposes and getting into containers.



***ETCD - Commands (Optional)***

ETCDCTL is the CLI tool used to interact with ETCD.  
  
ETCDCTL can interact with ETCD Server using 2 API versions - Version 2 and Version 3.  By default, it’s set to use Version 2. Each version has different sets of commands.

For example, ETCDCTL version 2 supports the following commands:

1. etcdctl backup
2. etcdctl cluster-health
3. etcdctl mk
4. etcdctl mkdir
5. etcdctl set

Whereas the commands are different in version 3

1. etcdctl snapshot save
2. etcdctl endpoint health
3. etcdctl get
4. etcdctl put

To set the right version of API set the environment variable ETCDCTL\_API command.

export ETCDCTL\_API=3

When API version is not set, it is assumed to be set to version 2. And version 3 commands listed above don't work. When API version is set to version 3, version 2 commands listed above don't work.

Apart from that, you must also specify path to certificate files so that ETCDCTL can authenticate to the ETCD API Server. The certificate files are available in the etcd-master at the following path. We discuss more about certificates in the security section of this course. So don't worry if this looks complex:

1. --cacert /etc/kubernetes/pki/etcd/ca.crt
2. --cert /etc/kubernetes/pki/etcd/server.crt
3. --key /etc/kubernetes/pki/etcd/server.key

So, for the commands I showed in the previous video to work you must specify the ETCDCTL API version and path to certificate files. Below is the final form:

1. kubectl exec etcd-master -n kube-system --sh -c "ETCDCTL\_API=3 etcdctl get / --prefix --keys-only --limit=10 --cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key"

***Kube-Apiserver***

* It is the primary management component is Kubernetes.
* It is the centre of the Kubernetes. All the commands go through it.
* It has following job:
  1. Authenticate User
  2. Validate Request
  3. Retrieve Data
  4. Update ETCD
  5. Scheduler
  6. Kubelet
* It is the only component which interacts with ETCD component directly.
* Some useful commands:
  1. To update the configuration after file update:

*kubectl replace --force -f <filename>*

* 1. To watch the change in deployment status without running the command again & again:

*kubectl get pods --watch*

* 1. If an object (rs, rc, deployment, pod, svc) is running and you want to edit its parameters like replica, but you don’t know the location of the yaml file or it hasn’t been created then run:

*kubectl edit rs/rc/deployment/po/svc -l “<key=value>”*

*Or*

*kubectl get all -l “<key=value>”*

* 1. To ssh into another node:

*ssh <node\_name>*

***Kube-Controller Manager***

* Manages various controllers in Kubernetes.
* It continuously monitors the state of various components within the system.
* The node-controller is responsible for monitoring the nodes and takes the necessary actions to keep the application running.
* The replication-controller is responsible for monitoring the replication set.
* ***How to install & view the Kuberbetes-Controller Manager?***
  1. Download the kube-controller-manager
     + wget <https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-controller-manager>
  2. Extract it & run it as a service
     + kube-controller-manager.service
  3. You can provide additional parameters here to customise the controller.
  4. You can enable or disable controllers from here.
  5. To view the kube-controller server manager options
     + If you have set it up using kube admin tool then kube admin sets it up as a pod in the kube-system namespace on the master node.
       - kubectl get pods -n kube-system
     + To inspect the options in a kube admin setup
       - cat /etc/kubernetes/manifests/kube-controller-manager.yaml
     + To inspect the options in a non-kube admin setup
       - cat /etc/system/system/kube-controller-manager.service

***Kube-Scheduler***

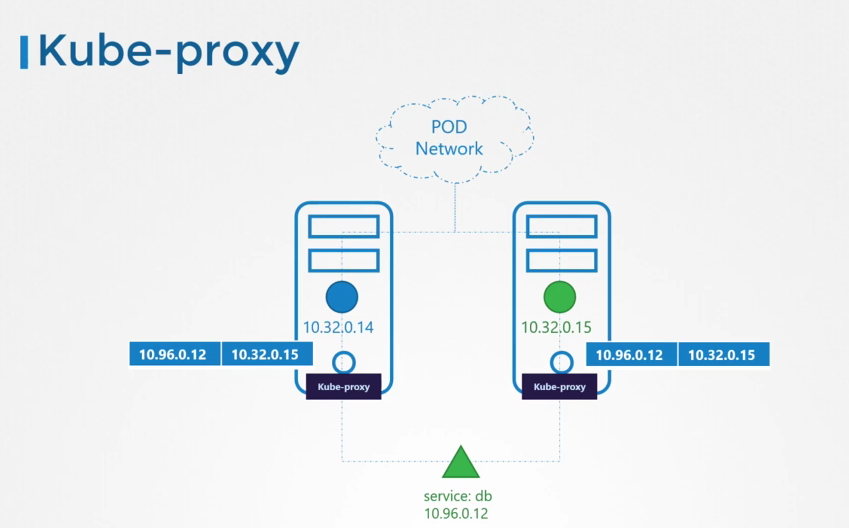
* Responsible for scheduling pods or nodes.
* It is only responsible for deciding which pod goes to which node.
* ***How to install kube-scheduler?***
  1. Download the kube-scheduler binary
     + wget <https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-scheduler>
  2. Extract it & run it as a service
     + kube-scheduler.service

***Kubelet***

* They load/unload containers as instructed by the scheduler or the master.
* They send status to the master.
* They register the node with the Kubernetes cluster.
* It creates pods.
* It monitors the pods.
* You need to manually install the kubelet on the worker node.

***Kube-Proxy***

* An application service can access the database service using the service name instead of IP, because the IP may change.
* This service name then exposes the IP of the database.
* Kube-proxy is a process that runs on each node in the Kubernetes cluster.
* Its job is to look for new services, and every time a new service is created, it creates the appropriate rules on each node to forward traffic to those service to the backend pods.
* It uses IP tables rule.



* ***How to install kube-proxy?***
  1. Download the kube-proxy:
     + *wget* [*https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-proxy*](https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-proxy)
  2. Extract it & run it as a service:
     + kube-proxy.service
  3. The kubeadm tool deploys kube-proxy as a pod on each node:
     + kubelet get pods -n kube-system
  4. It is deployed as a daemonset

***pod-definition.yaml file in Kubernetes***

* pod-definition.yaml file is used to create a pod in a Kubernetes cluster/node.
* These files are used as inputs by the Kubernetes for the creation of objects such as pods, services, etc.
* It always contains 4 top level fields:
  1. apiVersion - Version of the Kubernetes api, like v1
  2. kind - Type of object we are trying to create, e.g., pod, replica set, deployment, service.
  3. metadata - Data about the object, like name, labels
  4. spec - Specification of the object.



* To create the pod, run:

*kubectl create -f pod-definition.yaml*

*or*

*kubectl apply -f pod-definition.yaml*

***Replication-Controller***

* It helps in running multiple pods in a Kubernetes cluster.
* It is basically for load balancing & scaling.
* Replica set is the new technology.
* It has a yaml file as well with same components:



* Run the command:

*kubectl create -f rc-definition.yaml*

* To view the number of replications:

*kubectl get replicationcontroller*

***Replica Set:***



* Selector labels and container labels should be same. ReplicationController’s label may differ. It is to make sure that the number of pods always remain same as number of replicas mentioned.
* Run the command:

*kubectl create -f replicaset-definition.yaml*

* To view the number of replications:

*kubectl get replicaset*

**

If a replicaset is running and you want to update some value of it (suppose scale), but you don’t know where is the yaml file, you can use the following command:

*kubectl edit rs(or replicaset) <replicasetName>*

***Deployment***

* It’s a Kubernetes object.
* It provides us the capability to undo changes, pause, rolling updates, resume changes.
* Create a deployment-definition.yaml file:



* To run the deployment, run:

*kubectl create -f deployment-definition.yaml*

* Get the deployment:

*kubectl get deployment*

* The deployment then creates the replicaset, which creates the pods.

*You can make the kubectl create the yaml file for you.*

*Run the command e.g*

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

**Create a NGINX Pod**

kubectl run nginx --image=nginx

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

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

**Create a deployment**

kubectl create deployment --image=nginx nginx

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

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

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

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

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

kubectl create -f nginx-deployment.yaml

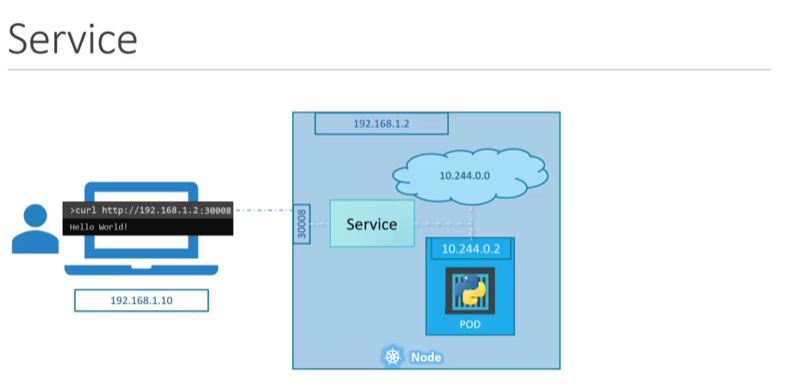
**OR**

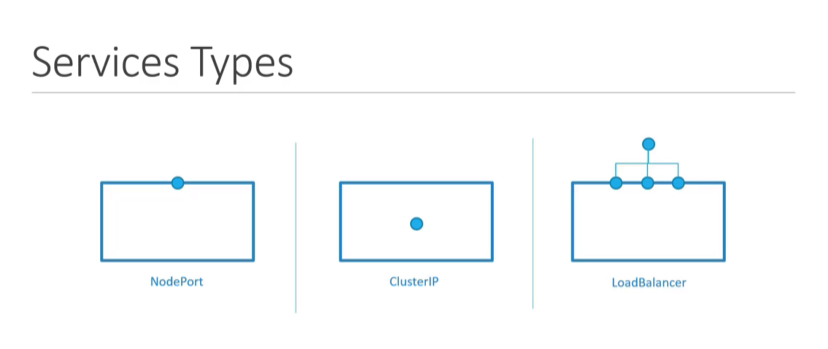
**In k8s version 1.19+, we can specify the --replicas option to create a deployment with 4 replicas.**

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

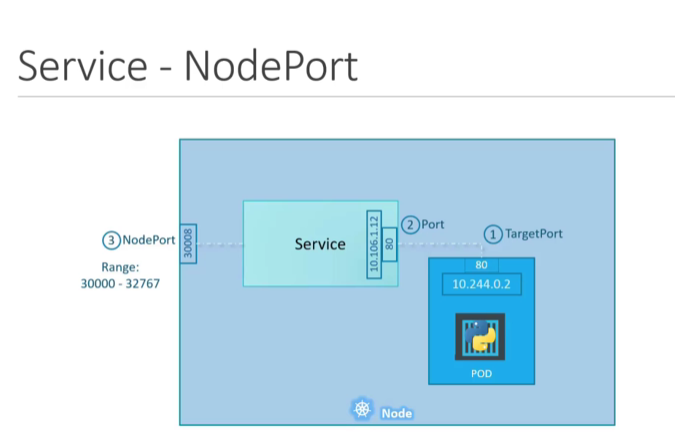
***Services:***

* It is a Kubernetes object.
* Services help connect to different pods.
* They help connect to external database services.
* They enable loose coupling in our application.
* It listens to a port on the node and forward request on that port to a port running the web application. It is known as node-port service.





* ***NodePort Service:***



* + Node ports have a valid range from 30000-32767.
  + Create a service:



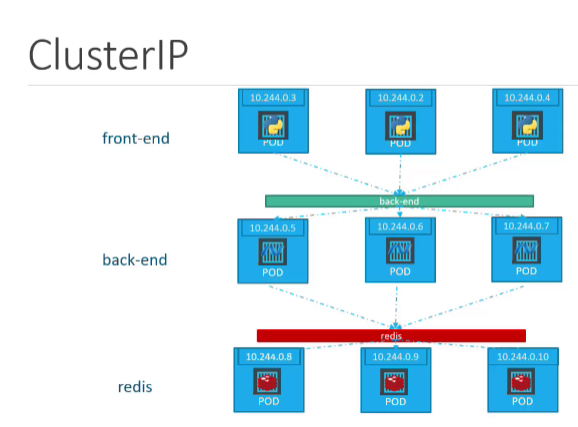
* + Run the create command:

*kubectl create -f service-definition.yaml*

* + View the running services:

*kubectl get services*

* ***Cluster IP Service:***



* + Create a clusterIP-service-definition.yaml file:

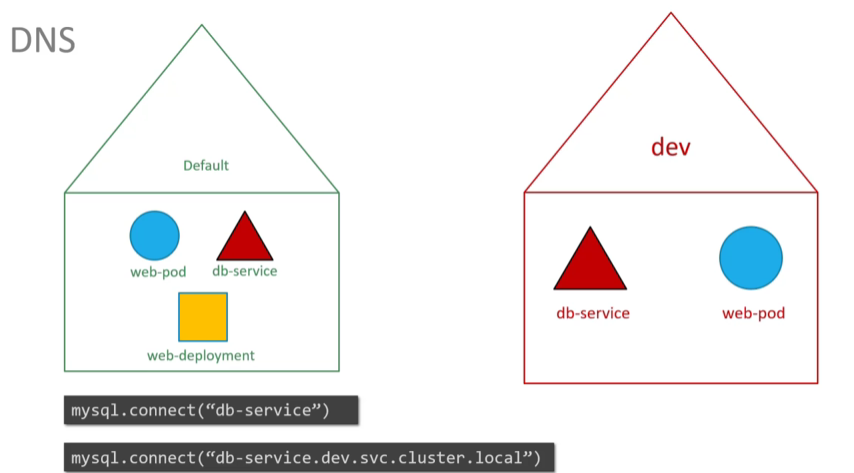


* ***Load Balancer Service:*** 
  + Create a loadBalancer-service-definition.yaml file:



***Namespace:***

* Namespace is like a house inside which Kubernetes objects resides.
* The “Default” namespace is created automatically by the Kubernetes during its initial set up.
* The Kubernetes creates the following namespaces automatically:
  1. Default
  2. kube-system - All the resources essential for Kubernetes are inside this.
  3. kube-public - Resources that are to be shared with the public are to be put inside this namespace.





* To create a pod inside a namespace then run:

*kubectl create -f pod-definition.yaml --namespace=dev*

***Or***

*Provide the namespace key & value in the metadata in the podWithNamespace-definition.yaml file*



*kubectl -f create podWithNamespace-definition.yaml*

* To create a namespace, you can use namespace-definition.yaml file:



***Or***

*kubectl create namespace dev*

* You can view the pods in the namespace by running the command:

*kubectls get pods --namespace=dev*

* If you want to permanently switch to a namespace then run:

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

* To view all the pods in all the namespaces:

*kubectl get pods --all-namespaces*

***Or***

*kubectl get pods --all -A*

* To limit resources in a namespace, create resource-quota.yaml file

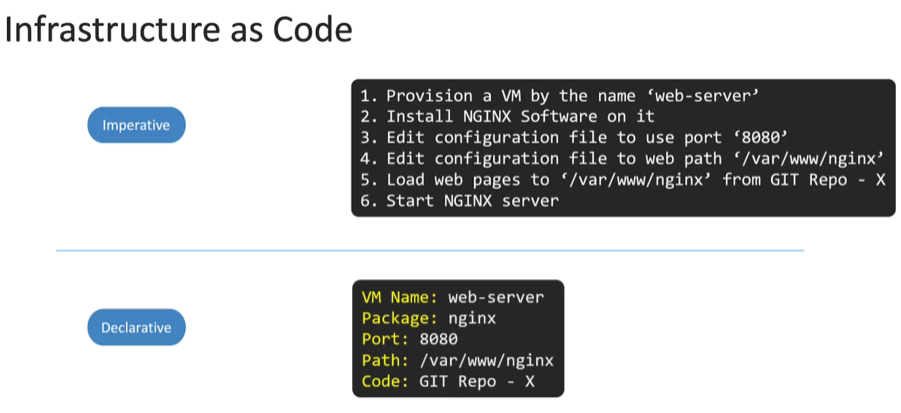


* Run the command:

*kubectl create -f resource-quota.yaml*

***Imperative vs Declarative:***

* Imperative approach means specifying each step about what to do and how to do.
* Declarative approach means just telling what to do to the system and it will be done.





* The apply command will look at the file and determine what changes are needed to be made to the system.

***Certification Tips - Imperative Commands with Kubectl***

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 a NGINX Pod:**

kubectl run nginx --image=nginx

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

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

#### *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=client -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.](https://github.com/kubernetes/kubernetes/issues/46191) 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 --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml

(This will automatically use the pod's labels as selectors, [but you cannot specify the node port](https://github.com/kubernetes/kubernetes/issues/25478). You must 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)

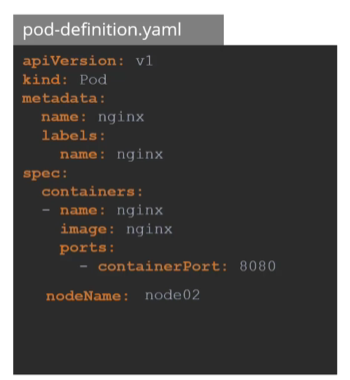
Both the above commands have their own challenges. While one of it cannot accept a selector the other cannot accept a node port. I would recommend going with the kubectl expose command. If you need to specify a node port, generate a definition file using the same command and manually input the nodePort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

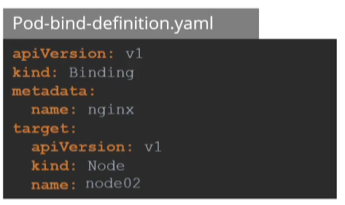
#### ****Reference:****

<https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands>

<https://kubernetes.io/docs/reference/kubectl/conventions/>

***Scheduler:***

* Scheduler is used to schedule the pods. It assigns the node to the nodeName parameter in the pod-definition.yaml file.
* If the scheduler is not present in the kube-system then the pods will be in pending state unless you manually enter the nodeName value.
* ***Manual Scheduling:***
  + Every pod-definition.yaml file has a field named nodeName in it. Usually, we don’t specify it, Kubernetes adds it automatically.
  + The scheduler goes through all the files which doesn’t have this property and identifies them as the candidates for the scheduling.
  + The scheduler identifies the right node for this pod but running the scheduling algorithm.
  + Once identified, it schedules the pod on the node, by setting the nodeName property to the name of the node by creating a binding object.
  + In manual scheduling, simply set the nodeName property by yourself while creating the pod.
  + We can only assign the nodeName while creating the pod-definition.yaml file.
  + If the pod is already created, then Kubernetes won’t allow you to assign the pod to a node. In this case, create a binding object and send a post request to the pod’s binding API, thus mimicking what the actual scheduler does.



* + In the binding object, specify the target node with the name of the node. Then send a post request to the pod’s binding API.

*curl --header "Content-Type:application/json" --request POST --data \ ‘{“apiVersion”: “v1”, “kind”: “Binding”, …..}’ \* [*http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding/*](http://$SERVER/api/v1/namespaces/default/pods/$PODNAME/binding/)

***Labels & Selectors:***

* They are used to group things together.
* Labels are properties attached to each item.
* Selectors help to filter out the items.

metada:

labels:

appName: my-app

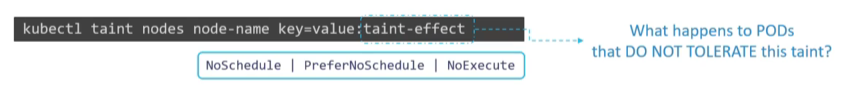
tier: frontend

* Or you can run the command:

*kubectl label <object> <object\_name> <key>=<value>*

***Taints & Tolerations:***

* Taints are for nodes, and tolerations are for pods.
* Taints on nodes put restrictions on pods whether it can be schedules on this node of not.
* Toleration on pod allows a pod to be schedules on a node with a taint, it’s like a passkey for the pod.
* To put taint on a node:



*kubectl taint nodes node1 app=blue:NoSchedule*

* To put a toleration on a pod put the toleration in pod-toleration-definition.yaml file:



* Types of taint effects:
  + NoSchedule: The pods without the toleration won’t be scheduled on the node with taint.
  + PreferNoSchedule: In this even though the pod won’t have toleration to the taint, but it is not guaranteed that it won’t be scheduled on the node with the taint.
  + NoExecute: In this, suppose there are 2 pods running on a node prior to its taint application. Now the taint is applied, so it’ll evict/kill the pod without the toleration.
* Taints & tolerations doesn’t tell the pods to be scheduled on which node, but it tells the node to accept which pod. Meaning, a pod can be scheduled on any node as long as it has a toleration, even on the nodes which doesn’t have any taint on it.
* To get the taint details on a node:

*kubectl describe nodes <node\_name> | grep Taint*

***Node Selectors:***

* It is used to make the pod run on the desired node only.
* For this, we add a new parameter called nodeSelector in the podWithNodeSelector-definition.yaml file:



* You also need to add a label to the node:

*kubectl label nodes <node\_name> <label-Key>=<label-value>*

* We can use only a single label in NodeSelector, that is its limitation. That’s why we use Node affinity & anti-affinity.

***Node Affinity:***

* Its function is to ensure pods are hosted on a particular node.



* Node affinity types:
  1. requiredDuringSchedulingIgnoredDuringExecution
  2. prefferedDuringSchedulingIgnoredDuringExecution
  3. requiredDuringSchedulingRequiredDuringExecution (Planning to introducing this)

***Taints & Tolerants vs. Node Affinity:***

* Only with taints & tolerant, even though other than the required pod no other pod will be assigned to the node, but the desired pod might itself get assigned to a different node.
* And only with affinity, the desired pod will be assigned to the desired node only, but any random pod can also be assigned to the desired node.
* Hence, we need to use a combination of both.
* Use taints & tolerants for other pods not to be placed on your nodes and use affinity for your pods not to be placed on other nodes.

|  |  |  |
| --- | --- | --- |
|  | Pods | Nodes |
| Taints & Tolerants | Having taint tolerant doesn’t mean with always be assigned to the tainted node only. It can be assigned to any untained node as well. | Won’t accept other pods. Only the pods with the required tolerants will be scheduled. |
| Affinity | Pods will be scheduled only on the desired node with the label. | Having a label doesn’t mean it’ll accept only the pods with the affinity. It can accept any pod without an affinity. |

***Resource Requirements & Limits:***

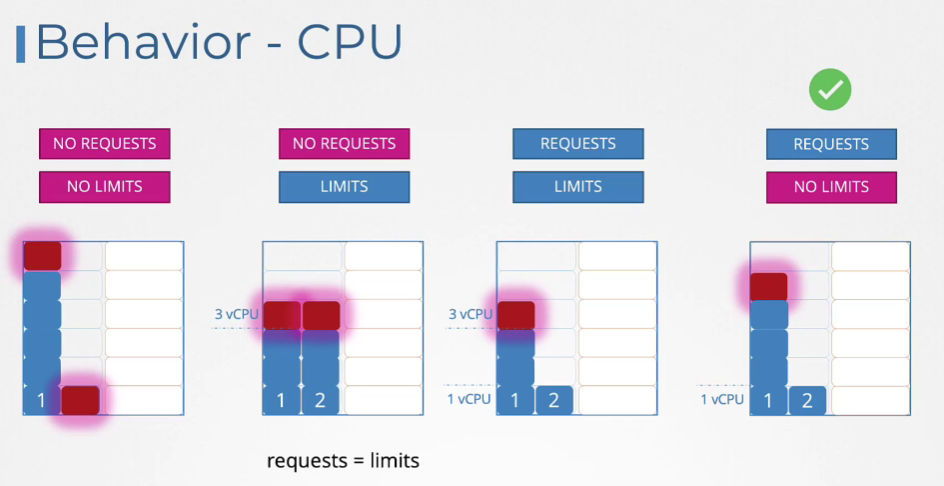
* Resource requirements block on a pod tells the scheduler how much resource this pod requires so that the scheduler can schedule it to a proper node.
* By default, Kubernetes doesn’t set any limit on the resources,
* Create a podWithResourceRequirements-definition.yaml file:

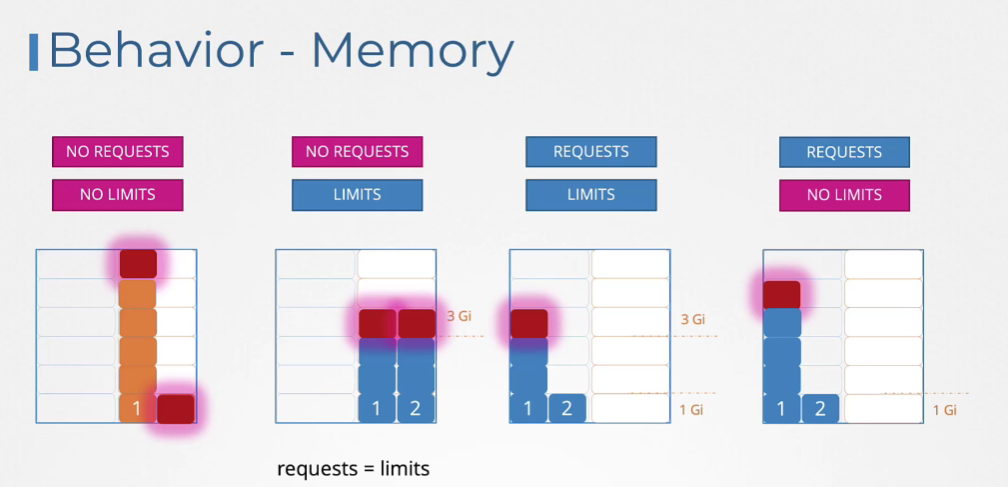


* You may also would like to limit the resources of the pod. Create podWithResourceRequirememntsAndLimits-definition.yaml file:



* If the container’s CPU usage goes beyond the limit, then the CPU starts throttling and the pod gets killed.
* If the memory goes beyond a limit then the pod doesn’t get instantly, rather it stays up for a while and then it gets killed with “Out Of Memory” error.





* In case you want to enforce resource limits on every pod created, you can use **LimitRange** object.
* LimitRange is a namespace level object which sets the default limits of the containers in a pod in a namespace.
* Create limitRange-definition.yaml file:



* To limit resources in a namespace, create resource-quota.yaml file

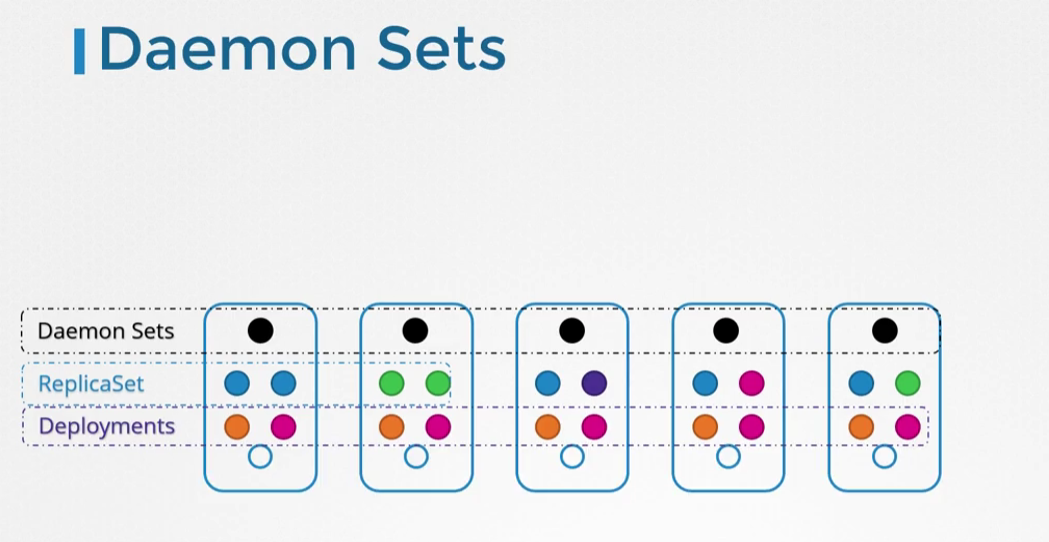


* Run the command:

*kubectl create -f resource-quota.yaml*

***Daemon Sets:***

* Daemon Sets are like ReplicaSets, as it helps in running multiple instances in your cluster, but it runs only 1 pod in 1 node.
* Daemon sets ensure that 1 copy of the pod is running on all the nodes of the cluster.

**

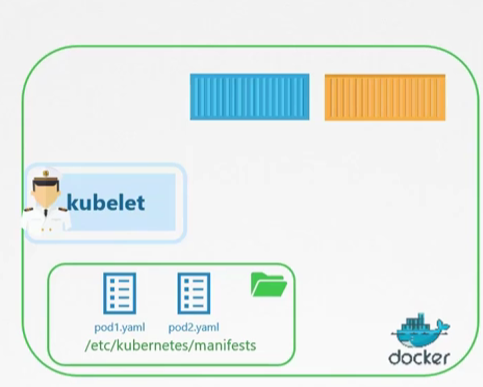
* Use cases:
  + Monitor agent
  + Log viewer
  + Kube-proxy
  + Weave-net
* Create a daemonSet-definition.yaml file:

**

* Then run ***kubectl create -f daemonSet-definition.yaml***
* Run***kubectl get daemonsets***to get daemonset details*.*

***Static Pods:***

* Kubelet can manage a node independently. Ideally, kube-scheduler schedules the pods on the node. But let’s imagine a scenario where there is no master node, no cluster, no kube-scheduler, no kube-apiserver, and there is only 1 node which has docker & kubelet in it.
* You can configure the kubelet to read the pod-definition files from a directory(***/etc/kubernetes/manifests)***, the kubelet will periodically check for new pod definition file and will update, add or remove pods based on the pod-definition.yaml file.
* Such pods which are created by solely by the kubelet are known as static pods.



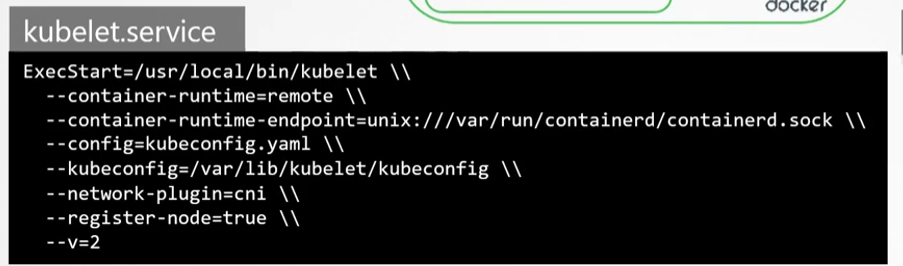
* Only pods can be created in such a way, no other Kubernetes objects can be created this way.
* You can change the directory location if you wish to. It is stored in:



Run the command:

***cat /etc/system/system/kubelet.service.d/<config\_file\_name>***

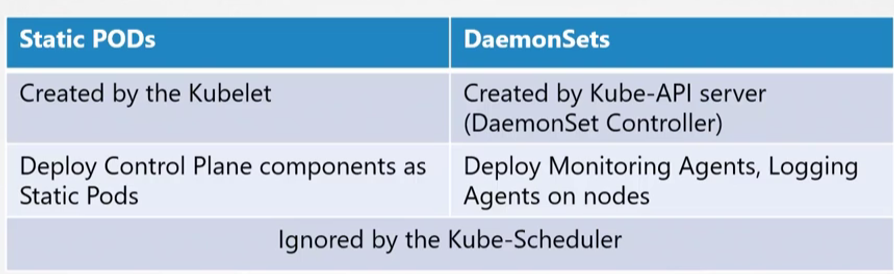
* You can also provide the directory path using a config file by making the following changes:
  + Provide the config file path (kubeconfig.yaml) in the config parameter.



* + Then add staticPodPath in that file (kubeconfig.yaml).



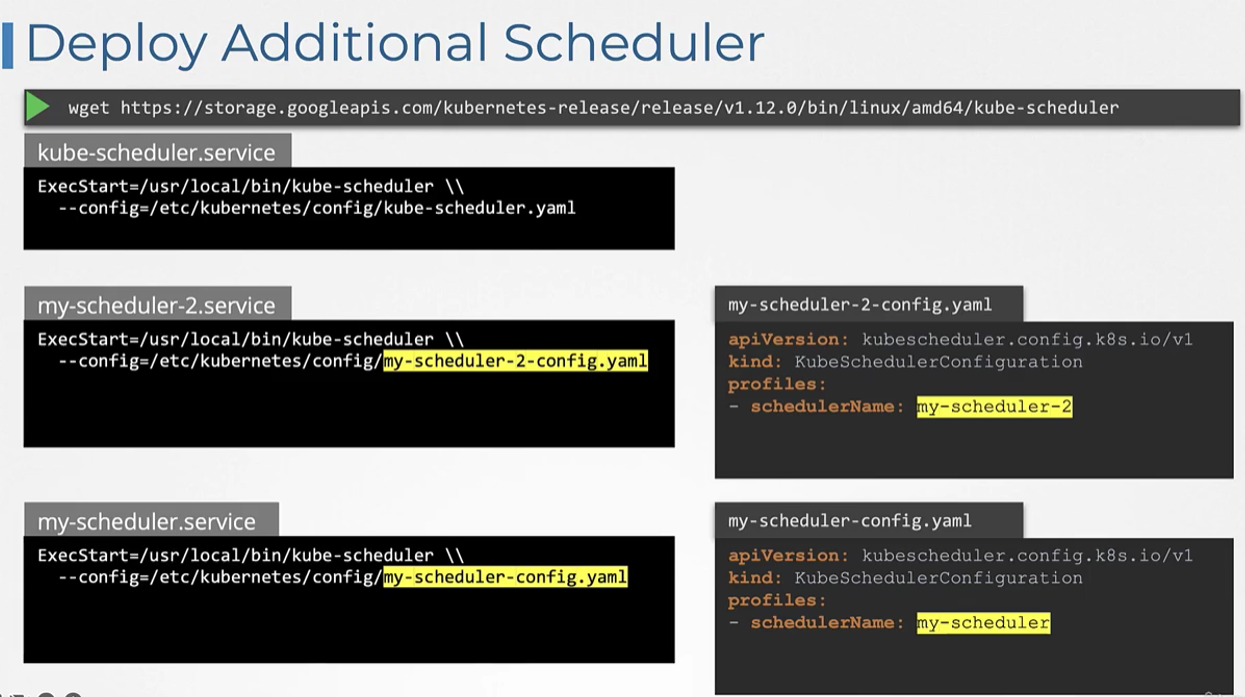
* To delete a static pod, delete the yaml file from the staticPodPath location. Same goes for update as well.



* A quick way to identify a deployed static pod is by noticing the node name at its end.

***Multiple Schedulers:***

* Kubernetes does the scheduling using its own algorithm to schedule the pods, but if you want to create your own scheduler then Kubernetes allow you to do that also.
* When creating a pod, you can instruct Kubernetes to be scheduled by a specific scheduler.
* Create a scheduler-config.yaml file: 
* Now to point out the scheduler to the custom scheduler, point out the config file to the file that you have created.



* ***What if you were to deploy the kube-scheduler as a pod?***



* Leader Elect Option:
  + Suppose you have setup multiple master nodes for high availability, and each of these master nodes will have the same copies of the custom scheduler that you have setup.
  + In such cases where multiple copies of the same scheduler are running on different master nodes, only one scheduler can be active at a time.
  + That’s why we have the leader elect option, which helps in maintaining who will lead the scheduling activity.
  + Create a schedulerWithLeaderElect-config.yaml file:



* + ***How to use the custom scheduler for a pod?***
    - Add a property called schedulerName in the podWithScheduler-definition.yaml file.



WNUB4KB5CPFI7B7EEENRLY6254GFEPTRWWGFAHLLR3TAQZBKR6M3RM6QM47DBNZV