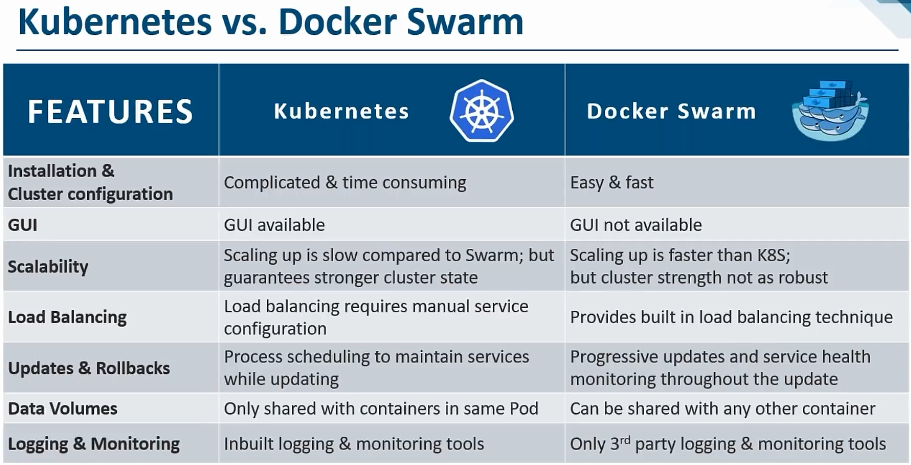
**KUBERNETES:**

* It is a container management technology developed in Google labs.
* It is an open-source system for scaling, automated deployment and maintaining containerized applications in cluster environment.
* Kubernetes referred as k8s.



**FEATURES:**

* Load balancing
* Self-healing (automatic restarts)
* Scheduling
* Scaling
* Rolling updates

**MASTER COMPONENTS:**

* **Etcd** - it is a key-value data store used to store configuration info (state) which can be used by each of node in the cluster. It is high availability key value store that can be distributed among nodes. It is accessed only by kubernetes api server, because of sensitive info. It stores info about service discovery, pods, services etc.
* **Api** **server** - it is an rest api, which manages all operations of clusters. It allows a user to configure workloads and organizational units. **Kubeconfig** is a package in the server side used for communication. It is the only master component that is user accessable. There is a client called **kubectl** which is a default method for interacting cluster from local computer.
* **Controller** **manager** - Basically a controller watches over state of clusters and when its get notified, it makes necessary changes to cluster to bring back to desired state. It has node controller, service controller, replication controller, namespace controller, endpoint controller. Each of these controllers works separately to maintain desired state.
* **Scheduler** - it is responsible for assigning workload among nodes. The scheduler watches newly created pods and assigned them to nodes.

**NODE:**

* Nodes are also called as minions, where all our work is happening.

**NODE COMPONENTS:**

* **Kubelet** - it is the most important component in kubernetes. It’s an agent running on nodes which watches api server for the pods that are bound to its node. It makes sure that pods are running. It reports to api server about status of pods. It determines state of pod containers based on the manifest provided by master. Exposes endpoint on: **10255**
* **Container** **Engine** - it does container management. Like pulling images, start/stop containers. It is pluggable, you can use docker, rocket etc.
* **Kube proxy**  - It is a proxy service running on all nodes, makes service available to external hosts. It forwards requests to correct containers. it assigns ip address to pods. Pod will have one ip address. All containers in pod will share one ip. It **load balances** across all pods.
* We will write a manifest file which has our desired state and give it to api server. The api will take care to get the desired state.
* The desired state is maintained by api. If the nodes are not in desired state, api will get the state by scaling.

**SERVICES:**

* Pods are ephermal and bound to get killed and started by replication controllers dynamically. Thus, communicating with pods might be difficult.
* A service is a grouping of pods that are running on a cluster. Service will get an ip and dns name. service is much like a **load balance** between your pods and they use random load balance.
* With the help of service, even if your pod ip changed there would be no issue because we are referring to service ip not pod ip. The service ip won’t change. Thus, there would be no disrupt in the flow.
* Services internally uses a concept called as “**lables**”. We will tell to service to load balance based on the label.
* We will assign lables to pods and specify that label to a service. So, the service will load balance based on the label.
* Service will load balance only to healthy pods.
* We can create another service with same lable and it load balances the pods.

**INSTALLATION:**

* Take 3 nodes one acts as master and other 2 acts as nodes.
* We need to install docker **v1.12, 1.13** on all nodes to work with kubernetes. Latest versions of docker is not supported.
* After installing docker we have to install kubernetes on 3 nodes.
* We ned to install kubeadm, kubelet and kubectl
* **Kubeadm** - command to bootstrap the cluster.
* **Kubelet** - agent which runs on all nodes, reports to master.
* **Kubectl** - cli to talk to your cluster.
* We have to allow ports to work with these.

**MASTER:**

* **6443\*** - api server
* **2379-2380** - etcd server client api
* **10250** - kubelet api
* **10251** - kube scheduler
* **10252** - kube controller

**NODE:**

* **10250** - kubelet api.
* **30000-32767** - nodeport services\*\*
* Create an yum repo file. And add the following content in the file

To see versions of k8s = apt-cache madison kubeadm

**[kubernetes]**

**name=Kubernetes**

**baseurl=https://packages.cloud.google.com/yum/repos/kubernetes-el7-x86\_64**

**enabled=1**

**gpgcheck=1**

**repo\_gpgcheck=1**

**gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg**

**exclude=kube\***

* you have to disable selinux to allow containers to access host file system.
* set selinux to permissive. Until selinux support is improved by kubelet.
* Create a file **/etc/sysctl.d/k8s.conf**. paste the following content. There are some issues with traffic being routed incorrectly due to iptables being bypassed.

**net.bridge.bridge-nf-call-ip6tables = 1**

**net.bridge.bridge-nf-call-iptables = 1**

* once saving the file, update it…..**sysctl --system**
* Do yum update and Install kubeadm, kubelet, kubectl with yum.

**yum install kubeadm kubelet kubectl**

* Once it is installed, start the kubelet service.
* All these above installation must be done in all nodes.
* Now choose any node as master and we have to initialize kubeadm to act as master.
* Before initialize, you have to turn off **swap** space, if any. The minimum numbers of cpus required is **2**.

**kubeadm init** - to initialize kubeadm.

* You can also initialize kubeadm with specific network.

**kubeadm init --pod-network-cidr=cidrblock -apiserver--advertise-address=ip**

* Once it is initialized it will give you some tasks and a **url** for nodes to join master.
* Perform the tasks as **normal** **user**.
* The tasks are:

**mkdir -p $HOME/.kube**

**sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config**

**sudo chown $(id -u):$(id -g) $HOME/.kube/config**

* the node join url is :

**kubeadm join 172.31.20.215:6443 --token fjjk0j.6g81t4qapnsy8kry --discovery-token-ca-cert-hash sha256:ae24e1a595f7dd38de0bf3bbb0f495848cf703d713def2852cf169bd4140edad**

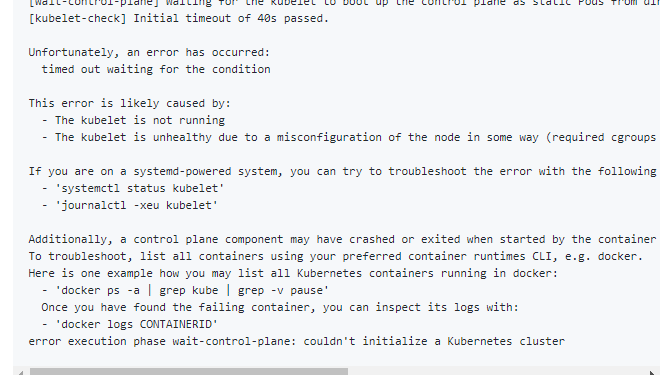
kubectl taint nodes --all node-role.kubernetes.io/master-

**taint is used to run all commands in master**

* Once kubeadm init is done, you have to install a **pod network** add-on so that your pods can communicate each other.
* There are multiple types of pod networks, you can choose any one of them. In my case, I am using **calico**. Here I am executing this command to create calico pod network.

**kubectl apply -f** [**https://docs.projectcalico.org/v3.8/manifests/calico.yaml**](https://docs.projectcalico.org/v3.8/manifests/calico.yaml)

* the url will be different based on pod network.
* Once pod network has been installed, you can confirm by checking the **core-DNS** pod is running or not with this command. **Kubectl get pods –all-namespaces**
* Thus you have setup Kubernetes cluster successfully.
* If you are facing any issues while **kubeadm init** like in the image below. Follow the below steps.



**sudo kubeadm reset**

**sudo kubeadm init phase certs all**

**sudo kubeadm init phase kubeconfig all**

**sudo kubeadm init phase control-plane all**

**sudo sed -i 's/initialDelaySeconds: [0-9][0-9]/initialDelaySeconds: 240/g' /etc/kubernetes/manifests/kube-apiserver.yaml**

**sudo sed -i 's/failureThreshold: [0-9]/failureThreshold: 18/g' /etc/kubernetes/manifests/kube-apiserver.yaml**

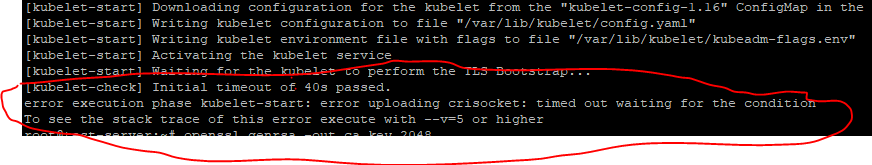
**sudo sed -i 's/timeoutSeconds: [0-9][0-9]/timeoutSeconds: 20/g' /etc/kubernetes/manifests/kube-apiserver.yaml**

**sudo kubeadm init --v=1 --skip-phases=certs,kubeconfig,control-plane --ignore-preflight-errors=all**

* By executing all the commands above, you can successfully initialize kubeadm.
* After this you will get a token and other configuration as per process.
* Now, you have to add nodes to master by executing the token url in the nodes.
* This token will get experies for every 24 hours, you have to create a new token by this command if you want to join a node after token expiration.

**kubeadm token create –print-join-command.**

* Once you executed token in all nodes. Check whether the nodes are attached to master (or) not with **kubectl get nodes** command.
* If you are facing any issues while adding the node to master like in the image below. Follow these steps.



* **swapoff -a**
* **kubeadm reset**
* **systemctl daemon-reload**
* **systemctl restart kubelet**
* once you execute the above commands, you can execute the node token url to add node to cluster successfully.
* By default, scheduler will run pods only on nodes not on master. To run pods on master also, execute the below command.

**kubectl taint nodes --all node-role.kubernetes.io/master-**

**PODS:**

* A pod is a group of containers that are deployed together on a single host.
* Pods are resides under nodes. More than one pod can share the same node.
* We have to write manifest files to create and run pods. These manifest file are executed by api server in nodes.
* **Kubectl get pods** - to get list of pods.
* **Kubectl create –f yml-file** - to create a pod (-f = file)
* **Kubectl describe pods** - get detailed info.
* **Kubectl get pods/podname** - to see specific pod.
* **Kubectl delete pods/pod-name** - to delete a pod.
* **Kubectl delete pods --all** - to delete all pods.
* **Kubectl apply –f yml-file** - to update the pods (replication controller).
* **Kubectl get pods --all-namespaces** - to get al pods started by default.
* Updating the pods means, when you update the manifest file to scale up (or) down the pods you have to use apply command. Because you have already created those pods.
* By default, k8s pull images form docker hub. But if you want to use images from local pc or from private repo, you have to configure a secret and specify that secret in pod or rc.
* To use images from local pc. Add this in container spec.

**imagePullPolicy: Never**

**imagePullPolicy: IfNotPresent**

* **Note**: you should that image in all the nodes.
* To pull image from a private repo, you have to login to your private repo in nodes and to create a secret .
* When you login to private repo, docker will update config.json with the new creds. We have to create secret by specifying path og the config.json.

**sudo kubectl create secret generic <secret-name> --from-file=.dockerconfigjson=~/.docker/config.json --type=kubernetes.io/dockerconfigjson**

* once you have created this secret , you have to specify imagePullSecrets in pod spec.

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: test-pvt**

**spec:**

**containers:**

**- name: nginx**

**image: ozd1861u:8083/nginx**

**imagePullPolicy: IfNotPresent**

**volumeMounts:**

**- mountPath: /cache**

**name: local**

**ports:**

**- containerPort: 80**

**imagePullSecrets:**

**- name: regcred**

**volumes:**

**- name: local**

**hostPath:**

**path: /root**

**NODE SELECTOR:**

* we can assign a pod to run on specific node by using node selectors.
* Assign a label to the node and use that label in pod spec.

**kubectl get nodes –show-labels** – to get labels of all nodes

**kubectl label node <node-name> key=value** – to assign label.

* Now if see node labels will be updated. You can use this node label in poc spec.

**nodeSelector:**

**key: “value”**

* You have to specify same key and value as you have assigned for a node.

**CONTROLLERS:**

* We specify replicas in controllers. And we specify our container in template section with a label.
* **Kubectl get rc** - to get replication controllers
* **Kubectl get rc/rc-name** - to get specific rc.
* **Kubectl describe rc/rc-name** - shows details of pod status, replicas etc.

**SERVICES:**

* Service discovery

Dns based

Environment variables

* Service type
* Cluster ip - stable internal cluster ip
* Node port - exposes the app outside of cluster by adding a cluster-wide open port on top of cluster. (the container port is exposed to outside world via node port).
* Load balancers - integrates nodeport with cloud based lb’s.
* Once you created the service, it will load balance across your pods and you can access pods outside of your cluster.
* You can also use cloud based load balancers with kubernetes.
* **Kubectl get svc** - to get services
* **Kubectl describe svc/svc-name** - to get detailed info of a service.
* **Kubectl delete svc/vc-name** - to delete a service.

**REPLICA SETS:**

* It makes sure that desired number of pods running all the time just like replication controller.
* If there are less pods than desired, it will create new ones. If there are excess than desired, it will delete them to match the desired pods.
* Replica set is next-generation of replication controller both doing the same thing.
* The one difference between them is **rs** works with **set-based selectors**, while **rc** works with **equality-based selectors**.
* **Equality-based**: three kind of operators we will be using **(=,** **==,** **!=)**

**env = prod**

**tier != frontend**

* **Set-based**: three kind of operators as well **(In , NotIn, exists)**
* the primary advantage of set-based is we can specify multiple objects in options.

**env in (prod, qa)**

**tier notin (frontend,backend)**

* you can use these selectors in cmd line like below:

**kubectl get pods -l env=prod** - equality

**kubectl get pods -l ‘env in (prod,qa)** - set-based

* you can use these selectorsa in manifest file like this:
* equality-based:

**selector:**

**env: prod**

**tier: frontend**

* Set-based:

**Selector:**

**matchExpressions:**

**- {key: env, operator: In, values: [prod, qa]}**

**- {key: tier, operator: NotIn, values: [frontend, backend]}**

* One more thing to remember in selector where to use **labels** and **matchlabels**.
* **Labels** under selector will be used for **old** resources **services**, **rc**.
* **Matchlabels** under selector will be used for **new** resources **deployments**, **rs**, **ds**, **jobs**.

**DEPLOYMENTS:**

* Deployments use **replica** **sets**, which have an unique feature called **revision**.
* It stores version of your manifest file and you can go back to any version you want to rolling updates.
* The only thing that u need to remember is the version number.
* There are no deployments in apiversion 1. You have to change the api version which is suitable for deployments (**apps/v1**).
* You have to specify replica sets in deployments instead of replication controllers,where you can specify **minreadyseconds**, which means you have to specify a time and if the pod is not up in that time, it will consider the pod as unhealthy.
* You have to specify **strategy**, where you can specify rolling updates.
* There are 3 types of deployments:
* **Recreate**  – re-create the deployments
* **rollingUpdate** - one by one
* **canary** - gradually shifts traffic from old to new version as some subset of users are using canary version and rest using old version.
* **blue/green** - both deployed at same time. Once new version is successfully working after all the tests, old one will be deleted.
* **Kubectl get deploy** - displays all deployments.
* **Kubectl describe deploy** **deploy-name** - displays details of deployment.
* **Kubectl get rs** - displays replica sets
* **Kubectl describe rs** - displays details of replica sets.
* **Kubectl apply –f deploy-file --record** - records revisions of deployments.
* **Kubectl rollout history deploy deploy-name** - displays deployment revisions.
* **Kubectl rollout status deploy deploy-name** - to see the status of last rolledout deployment.
* **Kubectl rollout undo deploy deploy-name --to-revision=3** - to roll back to specific revision number.

**Daemon Set:**

* A daemonset make sure that all nodes in cluster runs a copy of pod. If we add nodes to cluster, pods will added to them, if nodes are remove, pods will be garbage collected. All pods will be deleted if we delete daemonset.
* You have to write a manifest file for daemonSet and execute it.
* Just like all the other k8s components, thisd also have all the specifications in manifest file like Kind - DaemonSet, metadata, spec, template and selector under spec.
* **Kubectl create -f ds.yml** - to create ds
* **Kubectl get ds/ds-name** - to see ds status
* **Kubectl describe ds/ds-name** - to see detailed info about ds.
* **Kubectl delete ds/ds-name** - to delete ds

**CONFIGMAPS:**

* These are dictionaries of configuration settings. We can set a key-value pairs in configmaps and use them in containers as volumes.
* Using a configmap lets you separate application code form configuration. So that you can easily change configuration based on environment.
* configmaps can be used in two different ways….as volumes or env.
* We can create configmaps using
* **Directories** - It will create configmaps with all the files inside the dir.

**kubectl create configmap <cm-name> --from-file=dir-path**

* **Files** - Creates configmap for the specified file only.

**kubectl create configmap <cm-name> --from=file=file-path**

* **Literal** **values** - can create configmaps straight from the cmd line

**kubectl create configmap <cm-name> --from-literal=z=3443576**

* Once literal configmap is created, you can use this in pod spec in env section.

**env:**

**- name: some-name**

**valueFrom:**

**configMapKeyRef:**

**name: cm-name**

**Key: key u have specified**

* Once configmap is created using files, you can use this in pod spec.

**volumes:**

**- name: vol-name**

**configMap:**

**- name: cm-name (that u have created earlier)**

**Items:**

**- key: file-name (which u have used in configmap)**

**Path: file-to-mount in pods**

* Once you create pod it will create a file based on the path config in the respective path.

**SECRETS:**

* Secrets Is just like configmaps used to separate configuration from code. But the difference is we can store sensitive-data with secrets like passwords, ip address, token akd keys whereas with configmaps the sensitive data can be seen by everybody.
* Secrets can be used in two different ways….as volumes or env.
* Secrets will be stored in etcd datastore on k8s master.
* The size of secrets shouldn’t be not more than 1MB.
* There are three types of secrets. You have to specify type while creating secret.
* **Generic** - use if you want to create secret from files or dirs..
* **Docker**-**regstry** - use if you want to create secret for any pvt registry.
* **tls**
* We can create secrets for
* **Directories** - It will create secrets with all the files inside the dir.

**kubectl create secret type <st-name> --from-file=dir-path**

* **Files** - Creates secrets for the specified file only.

**kubectl create secret** **type** **<st-name> --from=file=file-path**

* **Literal** **values** - can create secrets straight from the cmd line

**kubectl create secret type <st-name> --from-literal=z=3443576**

* Once literal secrets is created, you can use this in pod spec in env section.

**env:**

**- name: some-name**

**valueFrom:**

**secretKeyRef:**

**name: st-name**

**Key: key u have specified**

* Once secrets is created using files, you can use this in pod spec.

**volumes:**

**- name: vol-name**

**secret:**

**- name: some-name**

**secretName: st-name**

* Once you create pod it will create a file based on the path config in the respective path.
* You can create secret by writing an manifest file as below. But you have to encode your data before specifying in manifest file.

**echo -n “username” | base64**

**echo -n “password” | base64**

* Copy the given output from endocing cmds to use it in manifest file.

**apiVersion: v1**

**kind: Secret**

**metadata:**

**name: some-name**

**Type: Opaque**

**Data:**

**Username: encoded-value**

**Password: encoded-value**

* Once you have created a pod and used this secret in those pods, data will be automatically decoded inside the pods.

**To create secret key for nexus repo:**

kubectl create secret docker-registry suresh --docker-server=ec2-100-26-218-137.compute-1.amazonaws.com:8083 --docker-username=admin --docker-password=123

**to use .docker/config.json form all what we logined repos:**

kubectl create secret generic all\

--from-file=.dockerconfigjson=.docker/config.json \

--type=kubernetes.io/dockerconfigjson

**JOBS:**

* A job is a controller which supervisor pods for carrying out certain tasks like a backup.
* It meant to be short-lived and wait for completion. Job runs until the task specified in the jobs are completed. If pod gives exit code 0, job will be stopped which means pods created by job will be shutdown.
* There are two types of jobs:
* Run to completion (jobs) - primarily used for batch process. When we submit job manifest file, pod will be created and execute a task. Once tasks finished, it will shutdown (not delete) by itself. Here controller waits for exit code 0
* Scheduled (cron) -

**VOLUMES:**

* Volumes are used for consistent storage and to share among all the pods and nodes.
* The data of the pods will be deleted once the pod is deleted. To persist the data beyond pod lifecycle we need to use volumes.
* Types of volumes:
* Ephemeral (same as pod’s life)
* Durable (beyond pod’s life)
* K8s supports these following volume types:

**Aws Ebs**

**ConfigMap**

**Emptydir**

**Gce persistentdisk - mount from google cloud**

**Hostpath - mount a file or dir from ur host system. nfs, efs**

**Persistentvolumeclaim**

**Secret**

* Emptydir - creates an empty dir on the node where pod is created. Same as pod’s life.
* You have to specify emptyDir volume section in containers spec as per below example.

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: nginx**

**spec:**

**containers:**

**- name: nginx**

**image: nginx**

**imagePullPolicy: IfNotPresent**

**volumeMounts:**

**- name: first-vol**

**mountPath: /cache**

**ports:**

**- containerPort: 80**

**volumes:**

**- name: first-vol**

**emptyDir: {}**

* Once pod is created, test whether volume is created or not in pod by running the below cmd.

**kubectl exec pod-name df mountpath**

* Hostpath - mount an host sir in pods.
* You have to specify hostpath section in containers spec as per below ex

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: nginx**

**spec:**

**containers:**

**- name: nginx**

**image: nginx**

**imagePullPolicy: IfNotPresent**

**volumeMounts:**

**- mountPath: /cache**

**name: first-vol**

**ports:**

**- containerPort: 80**

**volumes:**

**- name: first-vol**

**hostPath:**

**path: /root**

* Persistent Volumes - In companies, we may use different types of storage types, but to integrate with Kubernetes it will be a headache to configure.
* In this case, k8s provides persistent volumes with an standard config(api) to connect to all types of storage provided from and how it is consumed.
* K8s provides 2 types of api objects for sdtorage provisioning.
* **persistent** **volumes** is a pre-provisioned storage in the cluster. We have to create pv before creating pod.
* persistent volumes claim is a storage request by an user (developer).
* Lifecycle of persistent volumes are

**Provisioning** - static/dymanic

**Binding** - binds user request storage with a pool.

**Using** - can umount and use.

**Reclaiming** - can get back storage by deleting pvc.

**STATIC VOLUMES:**

* It is a type of persistent volume provisioning.
* First we have to create pv and request volume storage using pvc. If the request size matches with pv size, it will bound to pod. If the request size doesn’t match, it will wait until the size matches.
* We can use nfs(efs) to share volumes across nodes and mount it in pods.
* First create a persistent volume file with the nfs shared ip and path.

**apiVersion: v1**

**kind: PersistentVolume**

**metadata:**

**name: first-pv**

**spec:**

**capacity:**

**storage: 1Gi**

**accessModes:**

**- ReadWriteMany**

**storageClassName: manual**

**nfs:**

**server: 10.226.224.65**

**path: "/root/app"**

* Now we have to create persistent volume claim to claim the storage from pv.

**apiVersion: v1**

**kind: PersistentVolumeClaim**

**metadata:**

**name: first-pvc**

**spec:**

**storageClassName: manual**

**accessModes:**

**- ReadWriteMany**

**resources:**

**requests:**

**storage: 500Mi**

* Here in the pvc, you have to request how much storage you want from pv and you have to specify same **storageclassname** in pvc just as pv. Otherwise pvc won’t bound to pv.
* To use pvc You have to specify the pvc in pods/depoyments volumes section in config files like below.

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: nginx**

**spec:**

**volumes:**

**- name: some-name**

**persistentVolumeClaim:**

**Claimname: pvc name**

**containers:**

**- name: nginx**

**image: nginx**

**imagePullPolicy: IfNotPresent**

**volumeMounts:**

**- name: some-name (just as above one)**

**mountPath: /cache**

**ports:**

**- containerPort: 80**

**DYNAMIC VOLUMES:**

* In dynamic volumes, we don’t need to create a pv before creating pvc. We can automatically create while creating pvc.
* Users don’t need to worry about the correct size of the pv to specify in pvc. Users can create pvc with their specific size which create pv and bound to it.

Change node port range

/etc/kubernetes/manifests/kube-apiserver.yaml

--service-node-port-range=0-9000.