**Kubernetes**

URL: <https://github.com/adhig93/k8sinstall>

* **kubectl get nodes** 🡪 To check all the nodes in the setup
* **kubectl label node <name> node-role.kubernetes.io/worker1=worker1** 🡪 To label a node[give role]

k8s Master --> t2.medium

k8s Worker Nodes --> t2.micro

**Installation:**

1. We need 2 CPU, 2gb RAM min requirement
2. We should install docker as container run time
3. We need to download the packages from cloud.google.com
4. We need to install k8s components such as kubelet kubeadm kubectl kubernetes-cmi
5. Then add these config file to user so that user can run commands
6. At last, we will run kubeadm init which generates a token
7. With above token we connect to the worker nodes

**Container runtime:**

* A container runtime, also known as container engine, is a software component that can run containers on a host operating system. In a containerized architecture, container runtimes are responsible for loading container images from a repository, monitoring local system resources, isolating system resources for use of a container, and managing container lifecycle.
* K8S is a tool for container orchestration
  + previously developed by google
  + now by CNCF - cloud native computing foundation
  + uses GOLANG
* There is a server, and in it multiple java applications. Generally, we create multiple containers for java apps. And we have **nginx load balancer** to maintain. But when the server goes down all the containers go down. We will have multiple containers running. Docker cannot maintain multiple servers and also all servers should be in the same stage, sync.
* for this we need **Kubernetes** or **Docker swarm**, **Openshift** etc.
* **Pod** is a basic unit in kubernetes. Pods can contain one or more containers. (Other objects include controllers, probes, volumes, services, affinity, RBAC, ingress)

1. **Controllers** to deploy and maintain pods. watch the state of your cluster
2. **Probes** to maintain pods
3. **Volumes** to mount volumes
4. **Services** for networking
5. **Affinity** - where we want our pods to run on which server
6. **RBAC** - how access a user can have
7. **Helm** - package installer
8. **Scheduler**: control plane process which assigns Pods to Nodes.

* In general we have 3 masters in sync, and work nodes are connected to the master servers. nginx for load balancing

From Kubernetes 🡪 nginx 🡪 masters

* If we have swap memory, Kubernetes will **not** work. If you run nodes with (traditional to-disk) swap, you lose a lot of the isolation properties that make sharing machines viable. You have no predictability around performance or latency or IO

**Kubernetes Architecture**

**K8S master/control plane**

1. **API server**

* Total controlling part in the k8s cluster. any component in k8s should go through API server. even in **kubectl get nodes** 🡪 kubectl will connect to the API server.
* first it will check for **authentication and authorization** (refer the difference) then to concerned place and get back to console

1. **etcd**

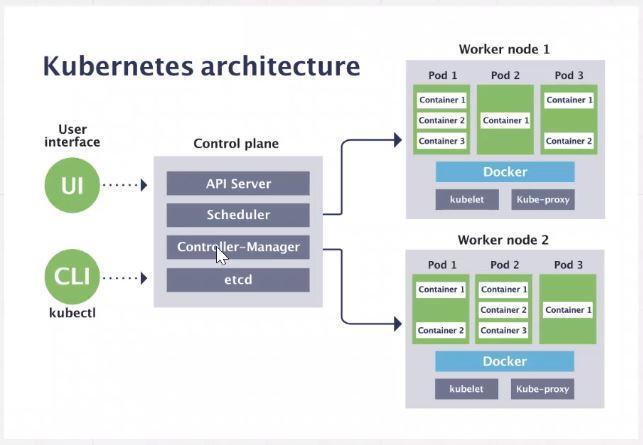
* storage point in k8s. it stores in form of key=value format
* stores configuration data, its state, and its metadata
* comes pre-installed
* if we want backup of k8s cluster, we can do it by backing up etcd
* Kubernetes uses etcd as its database.
* It also stores the *actual* state of the system and the *desired* state of the system in etcd. It then uses etcd’s *watch* functionality to monitor changes to either of these two things. If they diverge, Kubernetes makes changes to reconcile the actual state and the desired state

1. **Scheduler**

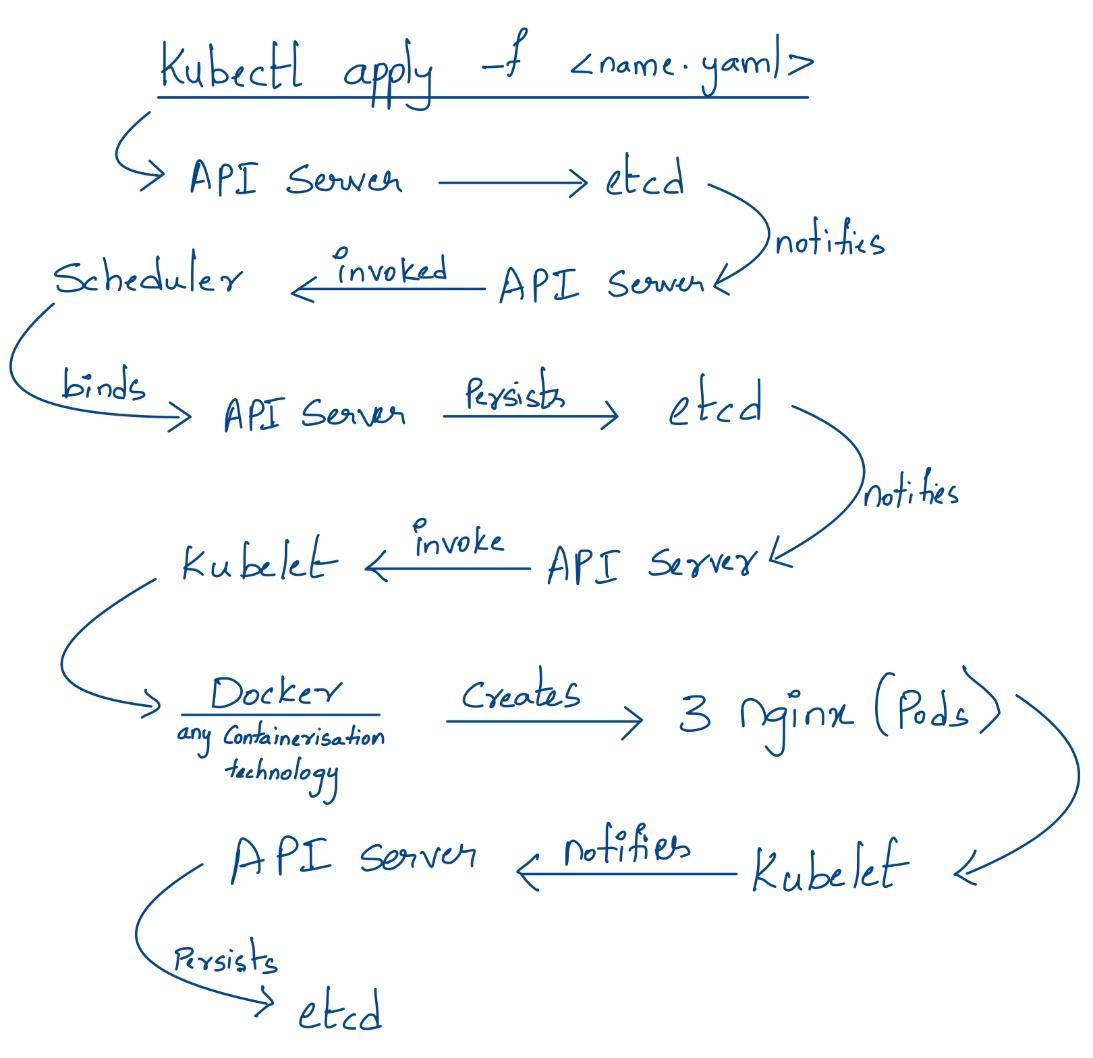
* It looks at all the nodes and checks which node is more available to k8s and will assign those pods to better nodes.
* Watch for any unassigned pods and bind them to those particular nodes

1. **Controller-Manager**
   * it looks for changes. If we ask for some changes, the controller will be at the backend. The controller helps in moving our k8s cluster from actual state to desired state.
   * types - replica set, deployment, stateful set, daemon set

**kubectl apply -f <name.yaml>**



API server --> etcd --> notifies API server --> invokes scheduler and binds --> API server --> persists etcd --> notifies API server --> invoke kubelet --> connects to Docker --> 3/any no. of nginx --> kubelet --> notifies API server --> persists etcd



**Example of a Deployment**

It creates a ReplicaSet to bring up three nginx Pods:

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

type: load-balancer

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

**Assignment** 🡪 Command to backup of etcd

Backing up an etcd cluster can be accomplished in two ways:

1. **built-in snapshot:**

A snapshot may either be taken from a live member with the **etcdctl snapshot save** command or by copying the **member/snap/db** file from an etcd data directory that is not currently used by an etcd process. Taking the snapshot will not affect the performance of the member.

*ETCDCTL\_API=3 etcdctl --endpoints $ENDPOINT snapshot save snapshotdb* -🡪 by $ENDPOINT to the file snapshotdb

ETCDCTL\_API=3 etcdctl --write-out=table snapshot status snapshotdb 🡪verify the snapshot

1. **volume snapshot**

If etcd is running on a storage volume that supports backup, such as Amazon Elastic Block Store, back up etcd data by taking a snapshot of the storage volume.

*ETCDCTL\_API=3 etcdctl -h* 🡪 will list various options available from etcdctl

*ETCDCTL\_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \*

*--cacert=<trusted-ca-file> --cert=<cert-file> --key=<key-file> \*

*snapshot save <backup-file-location>* 🡪 snapshot by specifying the endpoint, certificates etc

**Manifest file fields**

1. **API version:**

* Specifies the version of K8S API that we will be using to create any K8S object.

Ex: v1, apps/v1 etc

**Commands:**

* **kubectl api-versions** --> To check all the api versions in K8S
* **kubectl api-resource --api-group <api-version>** --> To check the objects that can be created under a api version

1. **Kind:**

* It specifies the type of K8S object we are trying to create
* Ex: Pod, Deploy, ReplicationController etc.

1. **Metadata:**

* It provides the information about the K8S objects like name of the pod, namespace under which the pod is running

1. **Spec:**

* It consists of core information about the pod and it specifies the desired state of our object

1. **Labels:**

* They are Key=Value pairs attached to an object that helps us select and group in K8S objects
* Ex:

Labels:

-app: nginx

-type: wed

1. **Selectors:**

* By using the matching labels, selectors can be used to identify and select the Kubernetes objects.
* Currently the API supports 2 types of selectors
  + Equity-based selectors 🡪 =, !=, ==
  + Set-based selectors 🡪 in, notin, exists.
    - * + ex: environment in (production, qa),
        + environment notin (dev)

**Commands:**

* **kubectl apply -f <.yaml\_file>** --> To create any K8S object
* **kubectl get pods** --> To display all the pods in the cluster [default-namespace]
* **kubectl exec <pod\_name> -it <command>** --> To get inside a pod
* **kubectl describe <object\_kind> <object\_name>** --> To get the information about K8S objects 🡪 in docker it is docker inspect 🡨
* **kubectl delete <object\_kind> <object\_name>** --> To delete an object in K8S

**Controller Types:**

1. **Replication controller/Replica Set:**

* They are used to run or create multiple instances of a single pod to achieve load balancing and high availability.
* if any pod dies or deleted then the controller will recreate a new pod

apiVersion: v1

kind: ReplicationController

metadata:

name: nginx-rc

spec:

replicas: 4

selector:

app: nginx

template:

metadata:

name: nginx

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

**kubectl scale --replicas=6 rc nginx-rc 🡪** to increase the number of replicas

**Difference between Replica set (RS) and Replica Controller(RC):**

* Both RS and RC are used to create replica of pods.
* while RC only supports equity-based selectors, the replica set (RS) supports both equity-based as well as set based selectors
* Replica Set is newly introduced or newer version

1. **Deployment controller:**

* deployment is the most common controller used for deploying the pods in Kubernetes
* we can effortlessly without breaking the user experience we roll out the application or roll back the update if anything breaks using deployment controller
* (ex: let’s say we have 3 replicas of the pod, if developer want to update a pod to version 1.1, then it creates new pod if any issue with new pod (1.1) it will be kept in waiting, V1.0 will still be running so that there will be available 3 pods at any point of time. If V1.1 works perfectly then it runs with that pod.)

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deploy

labels:

app: java

spec:

replicas: 4

selector:

matchLabels:

app: nginx

template:

metadata:

name: nginx

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.20

ports:

- containerPort: 80

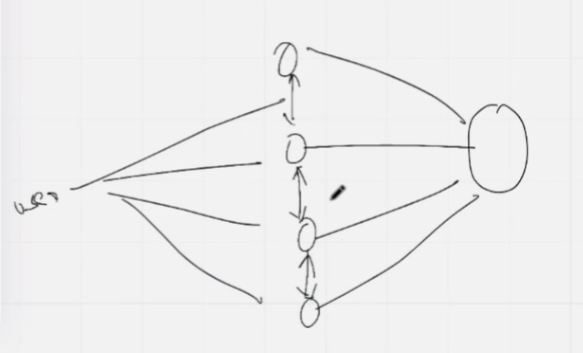
**kubectl set image deploy nginx-deploy nginx=nginx:1.21 --record** --> To update the image

**kubectl set image deploy nginx-deploy nginx=nginx:1.6 --record**

**kubectl rollout history deploy nginx-deploy** --> To check the total history of the controller

**kubectl rollout undo deploy nginx-deploy** --> To roll back to a previous version

**kubectl rollout undo deploy nginx-deploy --to-revision=2** --> To roll back to a specific version



**Deployment Strategies:**

A Deployment strategy defines how to create, upgrade, or downgrade different versions of Kubernetes applications. Deployment strategies that let you perform rolling updates to multiple application instances, and avoid or minimize downtime.

1. **Rolling deployment—**replaces pods running the old version of the application with the new version, one by one, without downtime to the cluster.
2. **Recreate—**terminates all the pods and replaces them with the new version.
3. **Ramped slow rollout**—rolls out replicas of the new version, while in parallel, shutting down old replicas.
4. **Best-effort controlled rollout—**specifies a “max unavailable” parameter which indicates what percentage of existing pods can be unavailable during the upgrade, enabling the rollout to happen much more quickly.
5. **Canary deployment—**uses a progressive delivery approach, with one version of the application serving most users, and another, newer version serving a small pool of test users. The test deployment is rolled out to more users if it is successful.
6. **DaemonSet Controller:**

* This controller ensures an instance of a pod is running on all the nodes of the cluster.
* if a node is added or removed from the cluster, daemon set automatically adds or deletes the pod
* use cases:
  + **Monitoring purpose**: for monitoring all the nodes of a cluster we can run tools like NodeExporter and Prometheus and deploy them using DaemonSet
  + **Logs Collection**: tools like Fluentd deployed using a DaemonSet, we can export logs from all the nodes of our cluster.

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: nginx-daemon

spec:

selector:

matchLabels:

app: nginx

template:

metadata:

name: nginx

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:1.20

ports:

- containerPort: 80

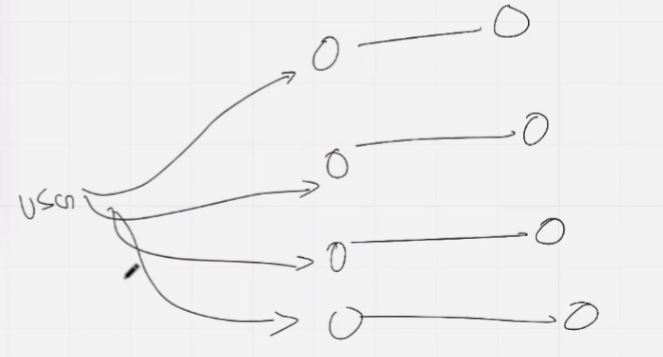
(cases where we want to run the pods in all the nodes. generally, Replica set or others check for availability of nodes and push the pod to that node (may be inefficient). But for monitoring and log purposes we might need a particular pod running on all nodes for example, monitoring tools like **Prometheus**, **NodeExporter**. Prometheus collects info, and use **Grafana** for Kubernetes visual dashboard, for log collection: **fluentd**)

<https://medium.com/devops-dudes/install-prometheus-on-ubuntu-18-04-a51602c6256b>

1. **Stateful Set:**

The common thing between the replica set, deployment controller and daemon set is the pods that are running are exactly identical to each other. That is the same state, same base volume etc. but in cases like databases, cache or database to pull info faster, we have a stateful set. Stateful set allows us to have different state, each pod can have their own state and their own volumes

* like deployment controller, stateful set manages pods that are based on an identical container specification
* unlike deployment controller, stateful set maintains unique identity for each of their pods, that is they have their own state and own volume
* we **cannot roll back** to a previous version. we can only delete, scale up, scale down or update the stateful set
* uses headless service to connect. (refer services)



**Pod Phases/Pod Lifecycle:**

1. **Pending**: pod has been accepted by k8s cluster but one or more of the containers has not been set up and made ready to run
2. **running**: pod has been bound to a node and all of the containers have been created and are up and running
3. **succeeded**: all the containers in the pod have terminated in success and will not be restarted
4. **failed:** all the containers in the pod have been terminated but atleast one of them got terminated in failure. that is the container exited with non zero status
5. **Unknown**: For some reason if the state of the pod could not be obtained then the pod goes into an unknown phase. this typically occurs due to an error in communicating with the node where the pod is running

**Probes:**

* They are the actions **performed by kubelet** to monitor the state or the condition of the pods
* At times pod response may slow down or may not respond at all. Prometheus etc. would monitor at higher level but if we want to monitor the application within a container or pod, or do health check we use Probes
* Probes will always be at container level (refer code)

**Configuration for probes:**

1. **Initial delay seconds**: (default: 0 sec, minimum: 0) amount of time (no. of seconds) we want to give to pod after container has started (to setup all its dependencies) before a probe action for the first time. Ex: if it is 10 sec, it waits for 10 sec and probe action starts.
2. **Period seconds:** (default: 10 sec, minimum: 1) it is how often you want your probe to perform its action. Duration for your probes between each probe action. Ex: probe action1 - 5sec - probe action2 - 5 sec - probe action3 and goes on
3. **Timeout seconds:** (default: 1 sec, minimum: 1) no. of seconds the probe waits for a response before assuming a failure. how much time we want our probe to wait to get the response
4. **success threshold:** (default: 1 , minimum: 1) minimum consecutive success times for the probe to be considered to be successful how many consecutive times we want our pod to give response that it is working properly
5. **Failure threshold:** (default: 3 , minimum: 1) when a probe check fails, the probe will try failure threshold times before marking the pod as failed. It is how many consecutive times we want our pod to give response that it is not working properly

**Types of Probes:**

**Readiness Probe**

* Are used to check whether the applications inside the container are ready and able to receive external requests. If the probe check fails, applications are not allowed to serve any request.
* when a pod is not responding properly then the request from the user will not be sent to that unhealthy pod. it sends to other pods.

**Liveness Probe**

* In addition to checking the container response, liveness probe causes the container to be killed and restarted if the probe check fails
* if any pod is not responding to the user's request, it will not only stop the pod to take request, but also it will delete the pod and recreate or restarts a new pod

**Startup Probe**

* latest introduction to Kubernetes v1.16
* Startup probe and Liveness probe has same features that is in case of failure, they cause the container to be restarted
* if both Startup and Liveness probes are set, K8S will execute Startup probe first, if it succeeds then the liveness probe will be executed normally as configured
* it's an extra layer of check for the pods
* It provides additional configuration over readiness and liveness probe

spec:

containers:

startupProbe:

httpGet:

path: <path\_to\_application>

port: <port>

initialDelaySeconds: 10

failureThreshold: 20

successThreshold: 10

livenessProbe:

httpGet:

path: <path\_to\_application>

port: <port>

initialDelaySeconds: 5

failureThreshold: 3

**Probe Actions:**

**Shell [exec]:**

* shell probe action lets us run shell commands in the pods and response is considered failure if the executed command exits with non-zero value

**Syntax:**

<probe\_type>

exec:

command:

- <command>

- <arguments>

apiVersion: v1

kind: Pod

metadata:

name: Shell

labels:

app: alpine

spec:

containers:

- name: alpine

image: alpine

args:

- /bin/sh

- -c

- touch /home/probe; sleep 20; rm /home/probe; sleep 10000

livenessProbe:

exec:

command:

- cat

- /home/probe

initialDelaySeconds: 5

failureThreshold: 5

**HTTP request (httpGet):**

* HTTP get sends a get API request to the path defined in the probe
* the HTTP response code will determine whether the probe check is successful or not

**Syntax:**

<probe\_type>

httpGet:

path: <path\_to\_application>

port: <port>

apiVersion: v1

kind: Pod

metadata:

labels:

test: liveness

name: liveness-http

spec:

containers:

- name: liveness

image: k8s.gcr.io/liveness

args:

- /server

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 3

periodSeconds: 3

1. [Informational responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#information_responses) (100–199)
2. [Successful responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#successful_responses) (200–299)
3. [Redirection messages](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#redirection_messages) (300–399)
4. [Client error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#client_error_responses) (400–499)
5. [Server error responses](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status#server_error_responses) (500–599)

**TCP port check [tcpSocket]**

* It is used to check whether a port is open and if the kubelet is able to connect to the specific port.

**Syntax:**

<probe\_type>

tcpSocket:

port: <port\_number>

**Assignment: Pod Failures**

### Wrong Container Image / Invalid Registry Permissions

### Application Crashing after Launch

### Missing ConfigMap or Secret

### Liveness/Readiness Probe Failure

### Exceeding CPU/Memory Limits

1. Resource Quotas
2. Insufficient Cluster Resources
3. Persistent Volume fails to mount
4. Validation Errors
5. Container Image Not Updating

<https://kukulinski.com/10-most-common-reasons-kubernetes-deployments-fail-part-1/>

<https://kukulinski.com/10-most-common-reasons-kubernetes-deployments-fail-part-2/>

**SERVICES:**

* services are a way of defining network configuration for pods
* services always point to the pods not the deployment or other controllers
* Note: by default, pods can communicate with each other without any configuration by using internal ip-addresses assigned to them by kubeproxy

**Need for K8S services:**

* The pods in Kubernetes are short lived. that is every time a pod dies, a new pod comes in its place and the ip address is most likely to change. therefore, we cannot rely on the internal ip address for communication
* the pods can be horizontally scaled and each new pod gets its own ip address
* there won't be a way for us to know the ip address of these new pods in advance

<https://medium.com/@deepeshtripathi/all-about-kubernetes-port-types-nodeport-targetport-port-containerport-e9f447330b19>

**Types of services:**

[**https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0**](https://medium.com/google-cloud/kubernetes-nodeport-vs-loadbalancer-vs-ingress-when-should-i-use-what-922f010849e0)

1. **Cluster IP**

* It is the default service type in k8s
* And it is only used to EXPOSE only internally within the cluster

apiVersion: v1

kind: Service

metadata:

name: nginx-clusterip

spec:

type: ClusterIP

selector:

app: nginx

ports:

- targetPort: 80

port: 90

* + Instead of connecting to the pods individually, k8s can be used to set up services (cluster IP) to connect to the pod. Pods are generally short lived (by liveness probe etc) so every time whenever the pod restarts, we cannot check the IP. For this we have cluster IP. When we use this IP, it accesses all the IPs in the backend. Cluster IP to connect internally in the cluster. Cluster IP is basic so it is connected to different services

**2. Node Port**

* it allows us to EXPOSE our pods running inside the cluster on a static port to the outside world
* nodeport range: 30000 - 32767

apiVersion: v1

kind: Service

metadata:

name: nginx-nodeport

spec:

type: NodePort

selector:

app: nginx

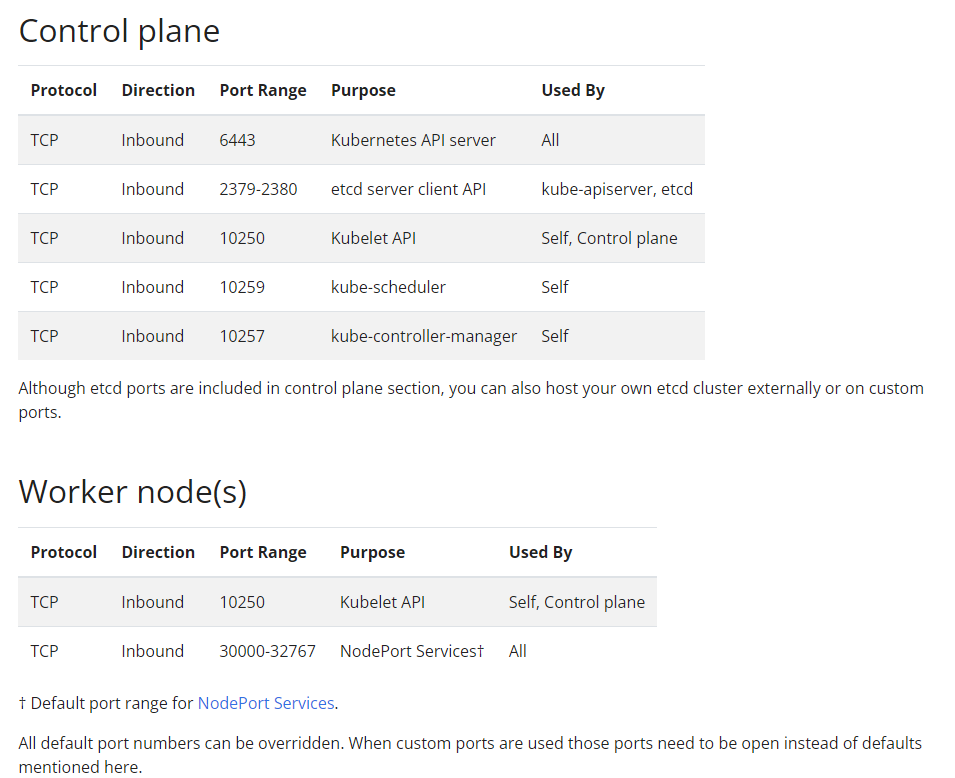
ports:

- targetPort: 80

port: 90

nodePort: 30500

* + If we want to connect to an outside server then we use a node port. Default range of ports **30000 – 32767**. Node port is connected to the Cluster IP.



1. **Load Balancer**

* way to EXPOSE services externally and it is useful when combined with cloud provider’s load balancer (ELB etc)
* Load balancer is connected to the node port. Can be any load balancer. Nginx Ingress - mostly used load balancer

1. **Headless**

* it is same as default clusterIP service but lacks load balancing or proxying allowing us to connect to a pod directly
* headless service will return all the IPs of the pod which are associated with it
* this is especially used **for stateless controller**

(**nslookup**: gives dns addresses etc 🡪 system admin command)

apiVersion: v1

kind: Service

metadata:

name: nginx-headless

spec:

clusterIP: None

selector:

app: nginx

ports:

- targetPort: 80

port: 90

**Load Balancer:** A kubernetes LoadBalancer service is a service that points to external load balancers that are NOT in your kubernetes cluster, but exist elsewhere. They can work with your pods, assuming that your pods are externally routable. Google and AWS provide this capability natively. In terms of Amazon, this maps directly with ELB and kubernetes when running in AWS can automatically provision and configure an ELB instance for each LoadBalancer service deployed.

**Ingress:** An ingress is really just a set of rules to pass to a controller that is listening for them. You can deploy a bunch of ingress rules, but nothing will happen unless you have a controller that can process them. A LoadBalancer service could listen for ingress rules, if it is configured to do so.

You can also create a **NodePort** service, which has an externally routable IP outside the cluster, but points to a pod that exists within your cluster. This could be an Ingress Controller.

An Ingress Controller is simply a pod that is configured to interpret ingress rules. One of the most popular ingress controllers supported by kubernetes is nginx. In terms of Amazon, ALB [can be used](https://github.com/kubernetes/ingress/tree/master/controllers/nginx) as an ingress controller.

For an example, [this](https://github.com/kubernetes/ingress/tree/master/controllers/nginx) nginx controller is able to ingest ingress rules you have defined and translate them to an nginx.conf file that it loads and starts in its pod.

**Namespaces:**

* allows us to partition our cluster into separate virtual subdivisions
* we can create multiple namespaces within one Kubernetes cluster and they are logically isolated from one another
* We can create a namespace for different teams. Projects or project stages

Kubernetes has 4 Initial Namespaces by Default:

1. **default**: any object that we create k8s without specifying namespace, they will be created under default namespace

2. **kube-system**: it is used by the k8s system to run its components. We usually never reconfigure or create objects in kube-system namespace

3. **kube-public:** resources inside this namespace are readable by all the users including those who are not authenticated

4**. kube-node-lease:** it is a new addition in k8s and it has resources which provides tools to monitor objects associated with each node

**Creating Namespace**

1. kubectl create ns <namespace\_name>

2.

apiVersion: v1

kind: Namespace

metadata:

name: dev

**To create objects under a specific namespace**

1.

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

namespace: dev

labels:

app: nginx

type: load-balancer

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

2. kubectl apply -f <yaml\_file> -n <namespace\_name>

**Scheduling Pods Manually:**

<https://medium.com/kubernetes-tutorials/learn-how-to-assign-pods-to-nodes-in-kubernetes-using-nodeselector-and-affinity-features-e62c437f3cf8>

In general pod scheduling is handled by the scheduler and it ensures that the right node is selected by checking the node's capacity for CPU and RAM. However, there are some scenarios where we want our pods to end up on a specific node.

**Ex:**

* when it is required for pods to end up on a node with some particular resources available
* when it is required to co-locate a pod from 1 service with a pod from another service on the same node due to strong dependency on each other.

**How to schedule our pods manually?**

* We cannot keep all the nodes at higher specs; simple applications can have small specs. Scheduler doesn't differentiate among the specs.
* it only looks for availability

**Types:**

Three ways to schedule our pods to specific nodes or not schedule

1. **Node Selector:**

* simplest way of scheduling pods to particular nodes by using its labels

**syntax:**

spec:

nodeselector:

<key>: <value>

**Example:**

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

nodeSelector:

instance: worker1

**Commands:**

**kubectl get nodes --show-labels** --> To display all the labels of the node

**kubectl label nodes <key>=<value>** --> To add a label to a node

1. **Affinity:**

* It greatly expands the node selector. It also provides soft and hard scheduling rules.
* If the soft rule is not met, the scheduler can still schedule a pod on any node
* Hard Rule: requiredDuringSchedulingIgnoredDuringExecution
* Soft Rule: preferredDuringSchedulingIgnoredDuringExecution

**2.a. Node Affinity:**

* similar to node selector, node affinity allows scheduling pods to a specific node

**Syntax:**

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: <key>

operator: In

values:

- <value>

**Ex (Hardrule):**

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: instance

operator: In

values:

- worker2

**Ex(Soft Rule):**

apiVersion: v1

kind: Pod

metadata:

name: nginx-affinity

labels:

app: nginx

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 1

preference:

matchExpressions:

- key: instance

operator: In

values:

- worker2

**2.b. Pod Affinity:**

apiVersion: v1

kind: Pod

metadata:

name: pod-affinity

labels:

app: pod

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

-labelSelector:

matchExpressions:

- key: app

operator: In

values:

- nginx

topologyKey: Kubernetes.io/arch

**2.c. Pod Anti Affinity**

apiVersion: v1

kind: Pod

metadata:

name: pod-antiaffinity

labels:

app: pod

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

-labelSelector:

matchExpressions:

- key: app

operator: In

values:

- nginx

topologyKey: Kubernetes.io/hostname

**Pod affinity and anti-affinity:**

* + With pod affinity and anti-affinity we can define whether a pod should or should not be scheduled on to a particular node based on labels of other pods that are already running on the cluster

1. **Repel(no schedule):**
   1. **Taints and Tolerations:**
   * Affinity is a property of pods that attracts them to a set of nodes
   * while the taint allows a node to repel the pods.
   * To schedule the pods on tainted nodes tolerations has to be added in the pod config file
   * Generally, on master node we will not schedule, to avoid any failure

**Taint effect:**

1. **NoSchedule**: unless the pod has matching toleration, pods won’t be able to schedule onto the nodes
2. **PreferredNoSchedule**: it is a soft version of NoSchedule
3. **NoExecute**: k8s will immediately evict all the pods that are running without the matching toleration

**How to Taint a Node**

* **Kubectl taint nodes <node\_name> <key>=<value>:<taint\_effect> 🡪** to taint a node
* **Kubectl taint nodes <node\_name> <key>=<value>:<taint\_effect>-** 🡪 adding a “-“ at the end will remove taint

**How to apply Toleration to a pod**

apiVersion: v1

kind: Pod

metadata:

name: taint-toleration

labels:

app: nginx

type: load-balancer

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

Tolerations:

- key: instance

operator: “Equal”

value: taint

effect: NoSchedule

**Kubernetes Volumes:**

In K8s, a volume can be thought of as a directory which is accessible to the containers in a pod.

**Types**:

1. **Ephemeral Volumes:**

* Ephemeral Volumes are short lived, they are tightly dependant with the lifetime of the Pod and they are deleted if the Pod goes down

1. **Persistent Volumes:**

* Persistent Volumes are meant for long term storage and are independent of the Pods or Nodes life-cycle.
* Persistent volumes can be created in two ways:
  + manually by an administrator (static) or
  + dynamically using storage classes (dynamic).
* Once created, persistent volumes can be bonded to a pod using a Persistent Volume Claims (PVC)

1. **Static (Manually created by the administrator)**

* Manual or Static provisioning consists of manually creating a persistent volume pool.
* This step is usually done by an administrator.
* PV 🡪 PVC 🡪 Pod

1. **Dynamic (Storage Classes)**

* In Dynamic Volumes we configure **storage classes** to dynamically create persistent volumes. PV 🡪 SC 🡪 PVC 🡪 Pod
* The main goal of storage classes is to eliminate the need for administrators to pre-provision storage and volumes are created dynamically on-demand.
* Users can then request storage by specifying the storage class in PVC.

**Uses (Persistent Volumes):**

* Persistent Volumes, when mounted, allow data to persist even when the pods are deleted.
* Persistent Volumes allow data sharing between pods by mounting the same Persistent Volume to various pods.

StorageClass --> PVC --> Pod

**Persistent Volumes:**

apiVersion: v1

kind: PersistentVolume

metadata:

name: pv

spec:

storageClassName: static

capacity:

storage: 2Gi

accessModes:

- ReadWriteOnce

hostPath:

path: /tmp/static

**Persistent Volumes Claim:**

* To claim a Persistent Volume, we use a Persistent Volume Claim.
* Persistent Volume Claims or PVCs are a way for an application developer to request storage for the application.

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: pv-claim

spec:

storageClassName: static

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

**Mounting on Container:**

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

volumes:

- name: volume-static

persistentVolumeClaim:

claimName: pv-claim

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

volumeMounts:

- mountPath: /home

name: volume-static

The **access modes** are:

* **ReadWriteOnce** -- the volume can be mounted as read-write by a single node
* **ReadOnlyMany** -- the volume can be mounted read-only by many nodes
* **ReadWriteMany** -- the volume can be mounted as read-write by many nodes

----------------------------------------------------------------------------------------------------------

**Reclaim Policies:**

1. **Retain**: When the PersistentVolumeClaim is deleted, the PersistentVolume still exists and the volume is considered "released". But it is not yet available for another claim because the previous claimant's data remains on the volume. An administrator can manually reclaim the volume.
2. **Delete**: Delete reclaim policy removes both the PersistentVolume object from Kubernetes, as well as the associated storage
3. **Recycle**: recycle reclaim policy performs a basic scrub (rm -rf /thevolume/\*) on the volume and makes it available again for a new claim.

----------------------------------------------------------------------------------------------------------

**Config Maps & Secrets:**

[**https://medium.com/google-cloud/kubernetes-configmaps-and-secrets-68d061f7ab5b**](https://medium.com/google-cloud/kubernetes-configmaps-and-secrets-68d061f7ab5b)

* ConfigMaps and Secrets are Kubernetes objects to store data in key value pairs.
* Pods can then use these key values Environment Variables or as configuration files in a volume.
* ConfigMaps and Secrets are ways of separating the configuration data from the Deployment manifests, which makes them more portable.

**Config Maps:**

* Kubernetes config maps are namespaced objects and they are used to store data in terms of plain text, which is open and readable to all the users who have access to the cluster.

**kubectl create configmap <configmap\_name> --from-literal <key>=<value>**

**kubectl create configmap username --from-literal user=abc**

apiVersion: v1

kind: ConfigMap

metadata:

name: username

data:

user: abc

**Secrets:**

* Kubernetes secrets are secure objects to store sensitive data such passwords, keys etc which are encrypted in base 64

**kubectl create secret generic <secret\_name> --from-literal <key>=<value>**

**kubectl create secret generic password --from-literal pass=test**

apiVersion: v1

kind: Secret

metadata:

name: password

type: Opaque

data:

pass: test

**Mounting on Containers as ENV:**

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

env:

- name: USERNAME

valueFrom:

configMapKeyRef:

name: username

key: user

- name: PASSWORD

valueFrom:

secretKeyRef:

name: password

key: pass

**Resource Quota:**

* Kubernetes namespaces help create logically isolated work environments. But namespaces do not enforce limitations or quotas.
* For this purpose, we use Kubernetes quotas to specify strict quota limits for around 15 Kubernetes API resources

apiVersion: v1

kind: ResourceQuota

metadata:

name: resourcequota

namespace: dev

spec:

hard:

pods: 1

configmaps: 2

requests.cpu: 0.5

limits.cpu: 1

requests.memory: 256Mi

limits.memory: 512Mi

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

labels:

app: nginx

spec:

containers:

- name: webserver

image: nginx

ports:

- containerPort: 80

resources:

requests:

cpu: 0.25

memory: 128Mi

limits:

cpu: 0.5

memory: 256Mi

**Securing Kubernetes:**

**Context:**

* A context is a group of access parameters to enable secure connection to a specific cluster’s API server.
* Each context contains a Kubernetes cluster, a user, and a namespace.

**Types of Users in Kubernetes:**

1. **User Account:**

* We can create users and groups who can connect to the Kubernetes API server.
* kubernetes-admin is the default user

1. **Service-Account:**

* Service accounts are used to give access to processes inside pods to interact with the Kubernetes API.
* They can also be used by applications outside the cluster.
* For example, Prometheus monitoring tool which is used to monitor the cluster can be given access using Service-Account.

**Creating a Service Account in Kubernetes**

**Step1**: Creating Service Account

**kubectl create sa test**

**Step** 2: Get the secret token of the Service Account

**kubectl describe sa test**

**Step** **3**: Get the token value of the secret

**kubectl describe secret <token\_name>**

**Step** **4**: Assign the token value to a variable

**TOKEN="<value>"**

(or)

**TOKEN="$(kubectl describe secret $(kubectl describe sa test | grep "Tokens" | awk '{print$NF}') | grep "token:" | awk '{print $NF}')"**

**Step** **5**: Set the credential for the SA

**kubectl config set-credentials test --token=$Token**

**Step** **6**: Create a new context for the SA

**kubectl config set-context <context\_name> --cluster=kubernetes --user=test**

**Step** **7**: Switch to newly created context

**kubectl config use-context <context\_name>**

<https://www.adaltas.com/en/2019/08/07/users-rbac-kubernetes/>

**RBAC:**

<https://medium.com/devops-mojo/kubernetes-role-based-access-control-rbac-overview-introduction-rbac-with-kubernetes-what-is-2004d13195df>

* RBAC or Role-Based Access Control is an approach in Kubernetes used to add constraints for users, groups and applications to access Kubernetes resources.
* RBAC basically adds security to the Kubernetes cluster and we can apply it for a specific namespace or to the total cluster.
* It was introduced in version 1.8 and uses rbac.authorization.k8s.io API group.

**3 important concepts in RBAC.**

1. **Subject**: Subject is the entity that needs access. It could be user or group or a service account
2. **Resources**: Resource is the K8s object that a subject wants to access. It could be pods, deployments, services etc
3. **Verbs**: Verbs are the actions that a subject can do on resources. It could be the list, delete, create, watch etc

----------------------------------------------------------------------------------------------------------

**Role & Cluster Role:**

* Role and ClusterRole contain a set of rules to access & modify Kubernetes resources.
* Difference between them is that Role works in a particular namespace while ClusterRole is cluster wide.
* Basically, we use Role If we want to define permissions inside a namespace and use ClusterRole for cluster wide.

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: sa-cluster-role

rules:

- apiGroups: [" "]

resources: ["pods", "secrets"]

verbs: ["list", "get", "delete"]

**kubectl create clusterrole sa-cr --resource=pods --resource=secrets --verb=get --verb=delete --verb=list**

**Role Binding & Cluster Role Binding**

* **Rolebinding** binds the Role to a Subject to access the Resources within a namespace while
* **ClusterRoleBinding** binds the ClusterRole to a Subject to access the resources cluster-wide.

*Role/ClusterRole* only say what can be done, while who can do what is defined in a *RoleBinding/ClusterRoleBinding*

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: sa-crb

subjects:

- kind: ServiceAccount

name: test

namespace: default

roleRef:

kind: ClusterRole

name: sa-cr

apiGroup: rbac.authorization.k8s.io

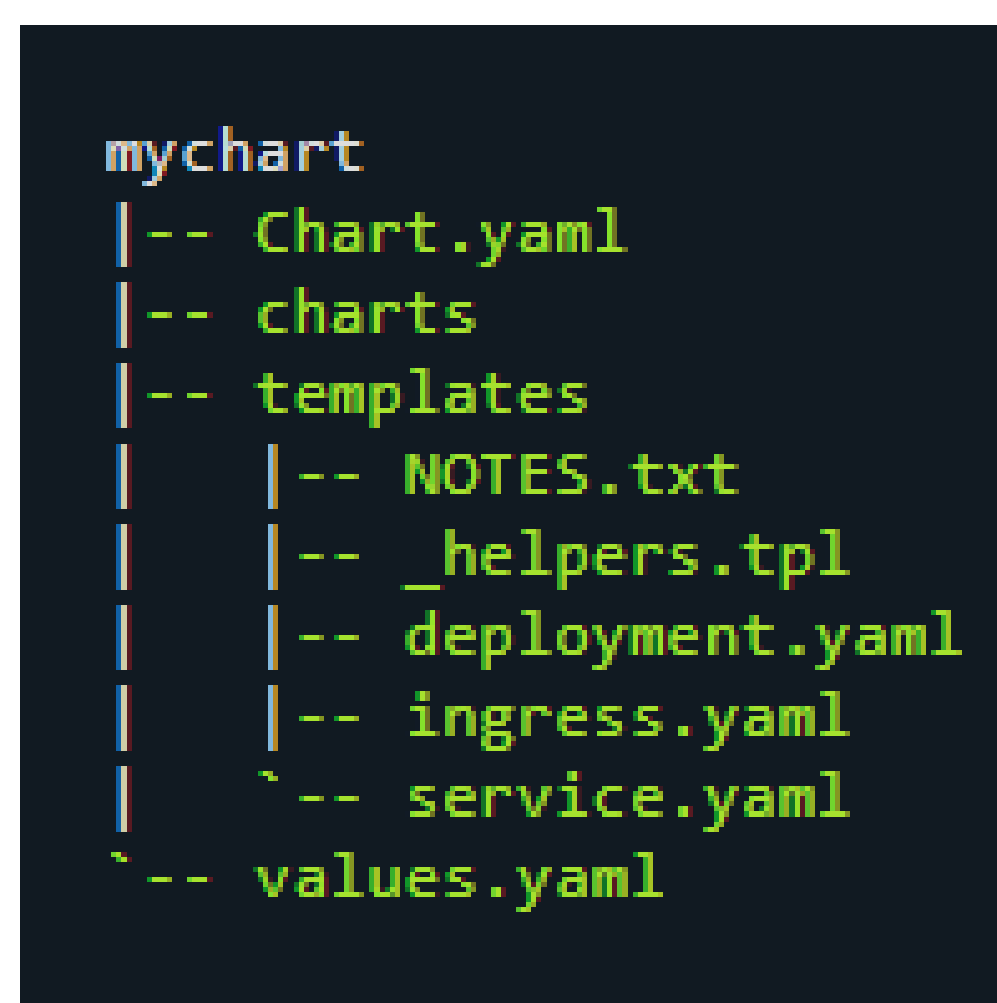
**kubectl create clusterrolebinding sa-crb --serviceaccount=default:test --clusterrole=sa-cr**

**Helm Charts:**

It is a package manager for Kubernetes which helps in combining the YAML definition to single package. Once packaged, installing a Helm Chart into your cluster is as easy as running a single helm install.

**Advantages:**

* Boosts productivity:
* Reduces duplication & complexity
* Enables the easy integration to ci/cd
* Simplifies deployments



**helm create <chart\_name>** 🡪 To create a helm template

**helm install <chart\_name> <folder\_name>** 🡪 To install a Helm chart

**helm upgrade <chart\_name> <folder\_name>** 🡪 To upgrade a Helm chart

**helm upgrade <chart\_name> --set replicacount=5 <folder\_name>** 🡪 To set values from CLI

**.helmignore:** This holds all the files to ignore when packaging the chart. Similar to .gitignore, if you are familiar with git.

**Chart.yaml:** This is where you put all the information about the chart you are packaging. So, for example, your version number, etc. This is where you will put all those details.

**Values.yaml:** This is where you define all the values you want to inject into your templates. If you are familiar with terraform, think of this as helms variable.tf file.

**Charts:** This is where you store other charts that your chart depends on. You might be calling another chart that your chart needs to function properly.

**Templates:** This folder is where you put the actual manifest you are deploying with the chart. They will all get their values from values.yaml from above.

**\_hepers.tpl:** This is file is used to create a template which we can refer in the manifest files.

**NOTES.txt:** This file helps in displaying any any information at the time of helm installation.

**Chart.yaml File:**

apiVersion: v2

name: nodejshelm

description: A Helm chart for Deploying NodeJS application in Kubernetes

type: application

version: 2.1

appVersion: "version13"

**deployment.yaml File**

apiVersion: apps/v1

kind: Deployment

metadata:

labels:

app: webapp

name: webapp-deployment

spec:

replicas: {{ .Values.replicaCount }}

selector:

matchLabels:

app: webapp

template:

metadata:

labels:

app: webapp

spec:

containers:

- image: {{ .Values.image.repository }}

name: nodejs

ports:

- containerPort: {{ .Values.ports.containerPort }}

imagePullSecrets:

- name: dockerhub

**service.yaml File**

apiVersion: v1

kind: Service

metadata:

name: webapp-svc

spec:

ports:

- port: {{ .Values.ports.containerPort }}

targetPort: {{ .Values.ports.containerPort }}

nodePort: {{ .Values.ports.nodePort }}

selector:

app: webapp

type: NodePort

**values.yaml File**

replicaCount: 3

image:

repository: adhig93/kubernetes:version13

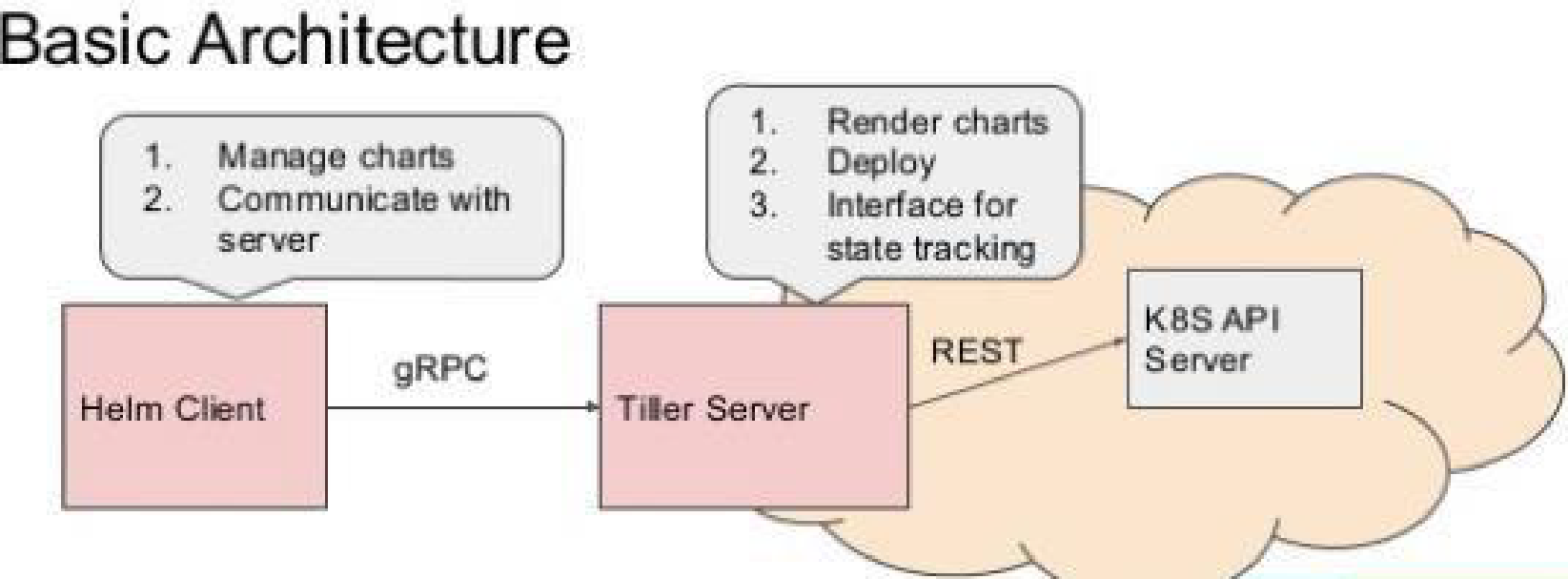
ports:

containerPort: 3080

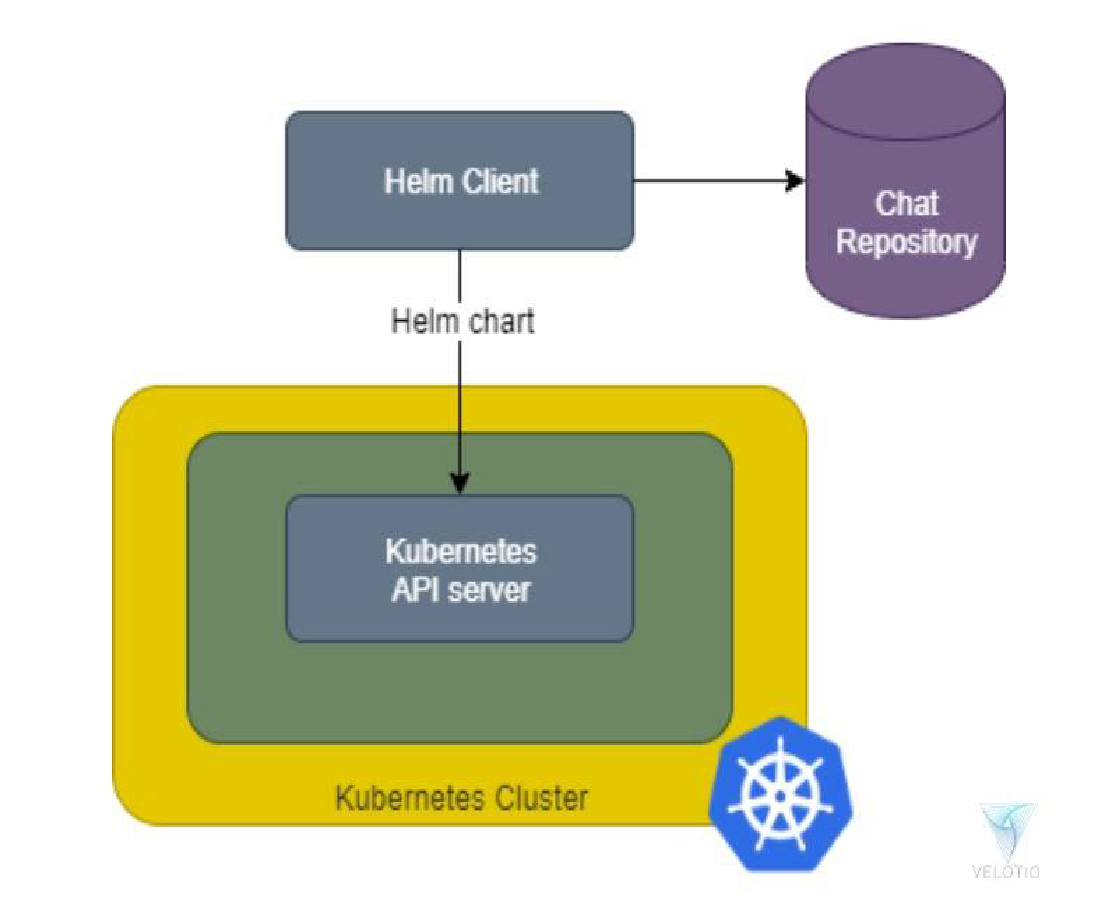
nodePort: 31200

**Architecture**

**Helm2**

****

**Helm3**

****

|  |  |  |
| --- | --- | --- |
| **Helm2** |  | **Helm3** |
| Tiller server present |  | Tiller server is removed |
| Chart dependencies should be mentioned in requirement.yaml |  | Chart dependencies can be mention in chart.yaml / requirement.yaml |
| Auto-genarate name is present |  | Auto-genarate name is absent, unless the auto-generate |
|  |  |  |
| Release details saved in configMaps |  | Release details saved in secrets. |
| Namespace will be created automatically |  | Namespace will not get created. |
| (CLI changes)  helm delete helm inspect helm fetch  .. |  | (CLI changes) Helm remove Helm show Helm pull  .. |

**Helm default objects**

* Object which we can refer in the chart template
* Release.Name : To get the release name
* Chart.Name : To get the chart name
* Chart.Version : To get the chart version

**Helm control structure**

**If/else:** for creating the conditional block

**range:** for each loop

**with:** to specify a scope

**Helm Hooks**

Helm provides a hook mechanism to allow chart developers to intervene at certain points in a release's life cycle example at pre-install, post-install, pre-upgrade, post- delete

**Helm commands**

* **helm create** : create a new chart with the given name
* **helm dependency** : manage a chart's dependencies
* **helm lint** : examine a chart for possible issues
* **helm env** : helm client environment information
* **helm install** : install a chart
* **helm upgrade** : upgrade a release
* **helm history** : fetch release history
* **helm rollback**: Rollback to any revisions
* **helm list** : List the releases
* **helm uninstall** : Remove the release
* **helm pull** :To pull the charts to local from remote repository

References

<https://helm.sh/docs/chart_template_guide/getting_started/>

**EKS Cluster:**

* EKS provides **an integrated console for Kubernetes clusters.**
* Cluster operators and application developers can use EKS as a single place to organize, visualize, and troubleshoot your Kubernetes applications running on Amazon EKS.
* The EKS console is hosted by AWS and is available automatically for all EKS clusters.

**Step 1:** Install AWS ClI

<https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2-windows.html>

**Step 2**: Open **cmd.exe** as Administrator

**Step 3:** Instal Chocolatey

<https://docs.chocolatey.org/en-us/choco/setup>

**Step 4**: Install eksctl

**choco install -y eksctl**

**Step 5:** Install kubectl

**choco install kubernetes-cli**

**Step 6:** Create Cluster

eksctl create cluster \

--name my-cluster \

--version 1.21 \

--with-oidc \

--nodegroup-name worker \

--region us-east-2 \

--node-type t3.medium \

--managed \

--ssh-access \

--ssh-public-key eks

Command to get all the kubeconfig files on to the local machine is →

**aws eks --region us-east-2 update-kubeconfig --name my-cluster**

**Jenkins Shared Library:**

* JSL is the concept of having a common code in your pipeline, which could be distributed in several applications yet, being versioned and referenced. Let’s assume you have to take responsibility for multiple applications, some of them are NodeJS, others are Python, or Maven even.with JSL you could segregate different components of your pipeline, which are called Libraries. They stay in a remote repository, being versioned, under your control. Every time you wish to add another test, or integrate another software, just create a new Library to do this.
* A Jenkins Shared Library is exactly what it says on the tin: a library with code you share between Jenkins jobs. Developers use libraries to share code and avoid duplication. Jenkins has libraries that let you do the same thing.
* If you’ve written code, you’ve heard about the DRY principle of software development: Don’t repeat yourself. It’s one of the cardinal laws of software engineering, and it applies to Jenkins, too. Copying build steps between build jobs isn’t just inefficient—it’s potentially dangerous. If you need to change one of those steps, you’ll have to track it down to every place it’s been copied.
* Jenkins loads Shared Libraries from SCM repositories. So, you write your code, push it to a repo, and then Jenkins loads it on demand. If you need to make a change, you update the code, push the new version to your source control and Jenkins picks it up the next time an associated build job runs.
* If you have more than one build job, then you have common tasks or variables. It may be as simple as the email where you need to send build results. Or it might be as complicated as updating a Docker container before using it to run a build

**@Library('shared-library')\_**

**jenkinsKubernetes2('dockerhub', 'bharathreddy06/k8s', 'version', \**

**'https://github.com/bharathreddy06/nodejs-k8s.git', \**

**'main','git')**