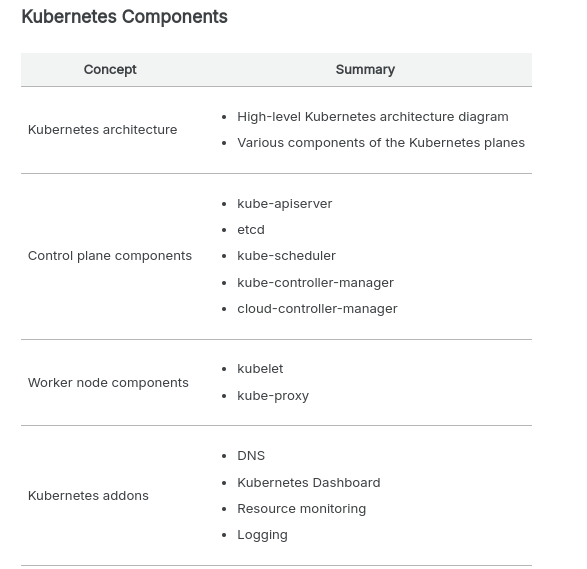
**Kubernetes**



h

**kube-api server**

At its core, the Kubernetes API server acts as an intermediary between user requests and the underlying components of a given cluster. It works by receiving requests from users via an HTTP or HTTPS request, validating those requests using predefined rules, and then forwarding them to other components (such as nodes) to perform an action such as deploying or scaling an application.

The API server also interfaces with other components in the control plane. For example, it can receive requests from the scheduler to create new pods in a cluster or talk to the controller manager to adjust an application’s resource utilization.

**etcd**

etcd is an open-source storage system that stores data as key/value pairs across multiple servers in a cluster. CoreOS developed it as part of its distributed systems platform.

Kubernetes uses etcd as its primary storage mechanism for storing all its cluster state information, such as configurations, endpoint information, service definitions, secrets, etc.

etcd can be set up in a highly distributed manner across multiple nodes so that no single node can become a point of failure. If one node fails or becomes unavailable, other nodes still have access to the same consistent set of data, making it possible to maintain availability and high performance even under extreme load conditions or during unexpected outages.

etcd is an indispensable component of any Kubernetes deployment because it provides reliable storage for vital application data while enabling reliable communication among components within and between clusters.

**kube-scheduler**

Kubernetes has a built-in cluster scheduler that is responsible for assigning workloads to available nodes in the cluster. The kube-scheduler determines which node should run each application within the cluster. The node selection process includes deciding which nodes have enough resources to meet an application’s requirements and which are most appropriate based on user preferences.

The kube-scheduler uses an algorithm known as bin packing to determine where to place containers in the cluster. This algorithm considers all available resources (CPU, memory, etc.) and maximizes utilization while ensuring that no single node is overloaded.

In addition, the scheduler will consider other factors, such as node labels (for example, if certain nodes need to be used only for specific purposes) or affinity rules (which allow you to specify which nodes a particular container should run on). The scheduler also keeps track of any changes in resource usage over time, so it can scale the pods in or out (if pod scaling is configured using solutions such as Horizontal Pod Autoscaler).

If any changes occur in terms of resource availability or user preferences, the scheduler will react by changing existing workload assignments or creating new assignments as needed.

**kube-controller-manager**

Kubernetes uses a built-in controller framework that ensures that all the nodes in your cluster are running as expected. It includes several controllers, each of which is responsible for a specific aspect of the cluster’s behavior:

**Node controller:** The node controller is responsible for ensuring that all nodes in the cluster are healthy and ready to accept workloads. It monitors the status of nodes on an ongoing basis and takes corrective action if any node fails or becomes unresponsive. For example, it can remove the unresponsive node from the cluster if necessary.

**Replication controller:** The replication controller runs continuously in the background to ensure that all pods (containers) are running as expected. It can detect when new pods have been added or removed from the cluster and adjust accordingly by starting or stopping additional pods to maintain desired pod count levels.

Service account and token controllers: These two controllers work together to manage authentication within a Kubernetes cluster by creating and managing service accounts and tokens for users to securely access resources within their environments.

**Endpoints controller:** This controller manages communication among services on different cluster nodes. It creates endpoints on those services to allow other services to communicate with them directly without going through an external load balancer. This helps improve application performance by reducing latency across services in a distributed system like Kubernetes clusters.

The kube-controller-manager also provides other essential functions, such as managing the namespace lifecycle, handling service accounts, and managing configuration changes. Overall, the kube-controller-manager plays a critical role in ensuring the desired state of the Kubernetes cluster.

**cloud-controller-manager**

The cloud-controller-manager is an essential component of Kubernetes that manages communication between the application and infrastructure layers. It provides a way for applications running on Kubernetes to interact with cloud services such as Amazon Web Services (AWS) or Google Compute Engine (GCE). Specifically, it enables you to provision resources from your preferred cloud provider and configure them within the Kubernetes cluster. Additionally, it allows you to access features such as load balancers or other networking capabilities offered by cloud providers.

The cloud-controller-manager runs as part of the control plane in Kubernetes, translating user requests into API calls that the underlying cloud provider can understand. For example, when an administrator wants to create a new load balancer in Amazon Elastic Kubernetes Service (EKS) or Google Kubernetes Engine (GKE), the admin will define the configuration in the Kubernetes cluster, which gets relayed to the appropriate API call via the cloud controller manager. The cloud controller manager also handles tasks such as scaling resources in or out based on user demand and ensuring that all resources are configured correctly before allowing applications to use them.

The cloud controller manager is a crucial component of Kubernetes because it enables you to harness the powerful features of cloud providers while managing cluster resources. This makes deploying complex applications much more straightforward than if Kubernetes administrators had to manage each resource manually. In addition, because the cloud controller manager operates as part of the control plane in Kubernetes, it ensures that all cluster nodes manage resources reliably. This makes it considerably simpler for administrators to ensure that clusters constantly operate at peak performance.

**Worker node components**

A worker node is a physical or virtual machine in Kubernetes responsible for running containerized apps, represented as pods and managed by the control plane. Each worker node has a kubelet that connects with the control plane to retrieve the necessary state for the pods operating on that node and a container runtime responsible for running the containers within the pods.

The worker node also runs additional system components, such as the kube-proxy, which aid in managing networking in the cluster. In a Kubernetes cluster, the worker nodes provide the computing power required to run containerized applications.

**kubelet**

The kubelet is essential to any Kubernetes worker node, acting as a bridge between the master and worker nodes. It runs on each node in the Kubernetes cluster to manage containers, monitoring their health and ensuring that they have access to enough resources for optimal performance.

The kubelet connects with other cluster components, such as the API server, scheduler, and replication controller, to coordinate tasks. Periodically, the kubelet interacts with the API server to retrieve the desired state of the containers and pods running on its node.

The desired state includes information such as which containers and pods must be running on the node, which images must be deployed, and how they must be configured. After retrieving the intended state, the kubelet compares it to the present state of containers and pods on the node. If the current state does not match the desired state, the kubelet takes action to reconcile the two, such as starting or stopping containers or creating or deleting pods.

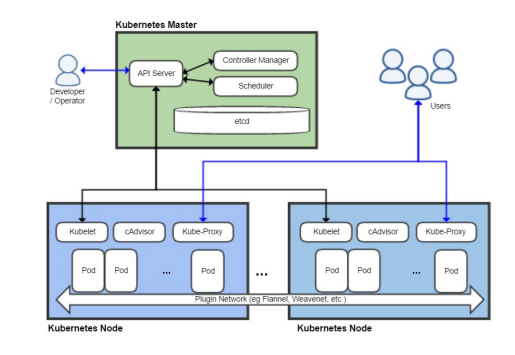
Kubelet also monitors container health by checking CPU/memory use and notifying the API server of any anomalies related to a specific container’s performance or health status.

**kube-proxy**

The worker nodes of Kubernetes are responsible for operating the cluster’s applications. The kube-proxy component is essential to these nodes since it functions as a proxy between cluster services and external clients. It controls which services can access each other and ensures that requests from external clients reach the correct service to manage network traffic.

Kube-proxy offers load-balancing features by dispersing incoming requests over different service endpoints. This ensures that all requests are efficiently handled and prevents any single endpoint from overloading.

Kubernetes kube-proxy also manages networking across pods, ensuring communication among containers within and between pods. This enables containers running in different pods to connect without using public IP addresses or directly exposing themselves to the Internet.



kubectl:

kubectl use to deploy and manages applications on kubernetes cluster.

***Kubectl run hello-minikube*** command is use to deploy the application.

***Kubectl cluster-info*** is use to view information about the clusters.

***Kubectl get nodes*** is use to list all nodes part of the cluster.

## Example crictl commands

The following examples show some crictl commands and example output.

### List pods

List all pods:

crictl pods

The output is similar to this:

POD ID CREATED STATE NAME NAMESPACE ATTEMPT

926f1b5a1d33a About a minute ago Ready sh-84d7dcf559-4r2gq default 0

4dccb216c4adb About a minute ago Ready nginx-65899c769f-wv2gp default 0

a86316e96fa89 17 hours ago Ready kube-proxy-gblk4 kube-system 0

919630b8f81f1 17 hours ago Ready nvidia-device-plugin-zgbbv kube-system 0

List pods by name:

crictl pods --name nginx-65899c769f-wv2gp

The output is similar to this:

POD ID CREATED STATE NAME NAMESPACE ATTEMPT

4dccb216c4adb 2 minutes ago Ready nginx-65899c769f-wv2gp default 0

List pods by label:

crictl pods --label run=nginx

The output is similar to this:

POD ID CREATED STATE NAME NAMESPACE ATTEMPT

4dccb216c4adb 2 minutes ago Ready nginx-65899c769f-wv2gp default 0

### List images

List all images:

crictl images

The output is similar to this:

IMAGE TAG IMAGE ID SIZE

busybox latest 8c811b4aec35f 1.15MB

k8s-gcrio.azureedge.net/hyperkube-amd64 v1.10.3 e179bbfe5d238 665MB

k8s-gcrio.azureedge.net/pause-amd64 3.1 da86e6ba6ca19 742kB

nginx latest cd5239a0906a6 109MB

List images by repository:

crictl images nginx

The output is similar to this:

IMAGE TAG IMAGE ID SIZE

nginx latest cd5239a0906a6 109MB

Only list image IDs:

crictl images -q

The output is similar to this:

sha256:8c811b4aec35f259572d0f79207bc0678df4c736eeec50bc9fec37ed936a472a

sha256:e179bbfe5d238de6069f3b03fccbecc3fb4f2019af741bfff1233c4d7b2970c5

sha256:da86e6ba6ca197bf6bc5e9d900febd906b133eaa4750e6bed647b0fbe50ed43e

sha256:cd5239a0906a6ccf0562354852fae04bc5b52d72a2aff9a871ddb6bd57553569

### List containers

List all containers:

crictl ps -a

The output is similar to this:

CONTAINER ID IMAGE CREATED STATE NAME ATTEMPT

1f73f2d81bf98 busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47 7 minutes ago Running sh 1

9c5951df22c78 busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47 8 minutes ago Exited sh 0

87d3992f84f74 nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c55d8ac139d5f 8 minutes ago Running nginx 0

1941fb4da154f k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f253d7fdd714b30a714260a 18 hours ago Running kube-proxy 0

List running containers:

crictl ps

The output is similar to this:

CONTAINER ID IMAGE CREATED STATE NAME ATTEMPT

1f73f2d81bf98 busybox@sha256:141c253bc4c3fd0a201d32dc1f493bcf3fff003b6df416dea4f41046e0f37d47 6 minutes ago Running sh 1

87d3992f84f74 nginx@sha256:d0a8828cccb73397acb0073bf34f4d7d8aa315263f1e7806bf8c55d8ac139d5f 7 minutes ago Running nginx 0

1941fb4da154f k8s-gcrio.azureedge.net/hyperkube-amd64@sha256:00d814b1f7763f4ab5be80c58e98140dfc69df107f253d7fdd714b30a714260a 17 hours ago Running kube-proxy 0

### Execute a command in a running container

crictl exec -i -t 1f73f2d81bf98 ls

The output is similar to this:

bin dev etc home proc root sys tmp usr var

### Get a container's logs

Get all container logs:

crictl logs 87d3992f84f74

The output is similar to this:

10.240.0.96 - - [06/Jun/2018:02:45:49 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"

10.240.0.96 - - [06/Jun/2018:02:45:50 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"

10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"

Get only the latest N lines of logs:

crictl logs --tail=1 87d3992f84f74

The output is similar to this:

10.240.0.96 - - [06/Jun/2018:02:45:51 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.47.0" "-"

PODS

Command to create pod with nginx container:

kubectl run nginx –image nginx

Command to list all pods

kubectl get pods

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Kubernetes – YAML file creation:

Below are the top level/root level properties, these are must have fields :

apiVersion: V1

kind: Pod #type of object we are trying to create

metadata: #Data about the object

name: myapp-pod

labels:

spec: #This is to mention the container for pod

file – pod-definition.yml

|  |
| --- |
| apiVersion: V1  kind: Pod  metadata:  name: myapp-pod  labels:  app: myapp  type: front-end  spec:  containers:  - name: nginx-container  image: nginx |

**To create the above pod, hit the below command:**

kubectl create -f pod-definition.yml

**Command to get the list of pods**

kubectl get pods

**Command to get the information about the pod**

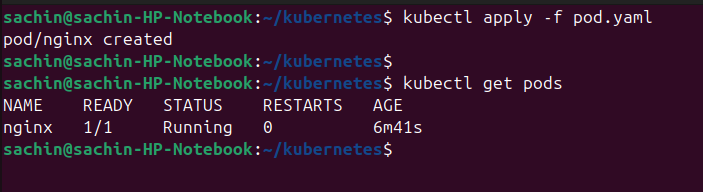
kubectl describe pod myapp-pod

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Installing the kubectl tool before miniqube, will allow miniqube to configure the kubectl utility to work with the cluster.

Kubectl utility can work multiple cluster. Like local or remote.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



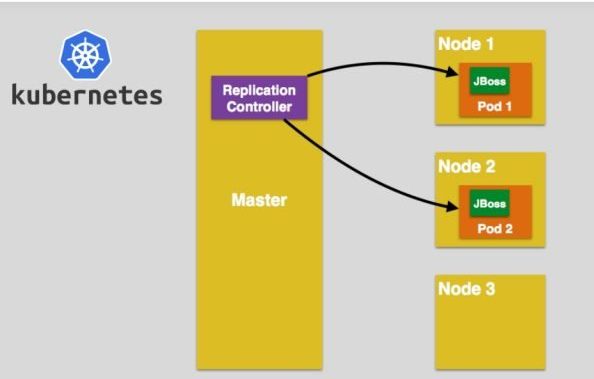
We also have **replication controller** to mitigate the pod failure, if any pod gets failed/crash, another pod will come into the picture with the help of replication controller.

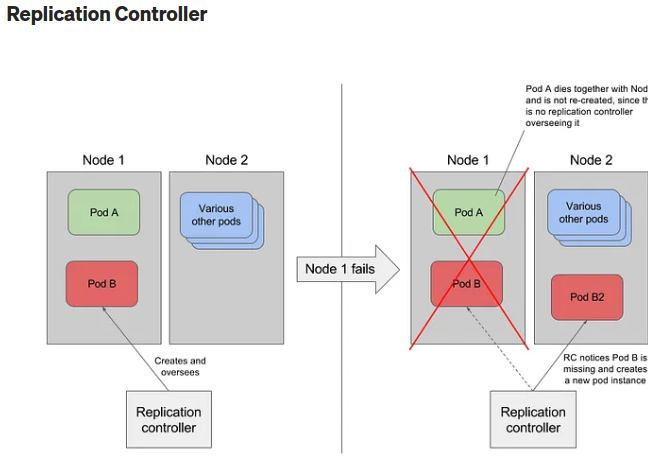
The **replication controller** ensures that the specific number of pods are running at all times even if it’s just 1 or 100.

A ****ReplicationController**** is a Kubernetes resource that ensures its pods are always kept running.

* If the pod disappears for any reason, such as in the event of a node disappearing from the cluster or because the pod was evicted from the node, the ReplicationController notices the missing pod and creates a replacement pod.
* The ReplicationController in general, are meant to create and manage multiple copies (replicas) of a pod

It is possible that you Node is out of resources while creating new pods with Replication controllers or replica sets, in such case it will automatically create new pods on another available cluster node

 s



Replica set – it is a upgraded version of replication controller.

## **Replication Controller Vs Replica Set:**

| ****Replication Controller**** | ****Replica Set**** |
| --- | --- |
| The Replication Controller is the original form of replication in Kubernetes | ReplicaSets are a higher-level API that gives the ability to easily run multiple instances of a given pod |
| The Replication Controller uses **equality-based selectors** to manage the pods. | ReplicaSets Controller uses **set-based selectors** to manage the pods. |
| The rolling-update command works with Replication Controllers | The rolling-update command won’t work with ReplicaSets. |
| Replica Controller is deprecated and replaced by ReplicaSets. | Deployments are recommended over ReplicaSets. |

|  |
| --- |
| #Below the yaml file for ReplicationController  apiVersion: v1  kind: ReplicationController --------------------------> 1  metadata:  name: Tomcat-ReplicationController --------------------------> 2  spec:  replicas: 3 ------------------------> 3  template:  metadata:  name: Tomcat-ReplicationController  labels:  app: App  component: neo4j  spec:  containers:  - name: Tomcat- -----------------------> 4  image: tomcat: 8.0  ports:  - containerPort: 7474 ------------------------> 5 |

****Setup Details****

****Kind: ReplicationController**** → In the above code, we have defined the kind as replication controller which tells the kubectl that the YAML file is going to be used for creating the replication controller.

****name: Tomcat-ReplicationController**** → This helps in identifying the name with which the replication controller will be created. If we run the kubctl, get rc < Tomcat-ReplicationController > it will show the replication controller details.

****replicas: 3**** → This helps the replication controller understand that it needs to maintain three replicas of a pod at any point in the pod life cycle.

****name: Tomcat**** → In the spec section, we have defined the name as tomcat which will tell the replication controller that the container present inside the pods is tomcat.

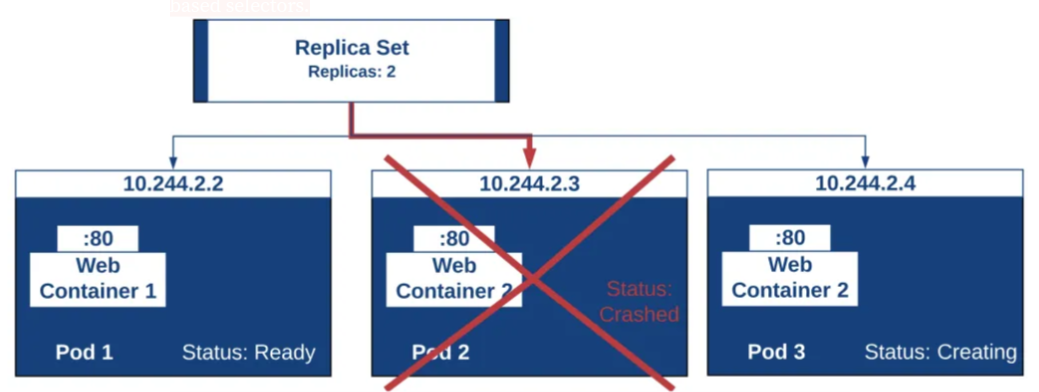
****containerPort: 7474**** → It helps in making sure that all the nodes in the cluster where the pod is running the container inside the pod will be exposed on the same port 7474.

Description: Kube Service for Replicas

Here, the Kubernetes service is working as a load balancer for three tomcat replicas.

## **Kubernetes — Replica Sets**

Replica Set ensures how many replicas of the pod should be running. It can be considered as a replacement or replication controller. The replica set and the replication controller's key difference is that the replication controller only supports equality-based selectors whereas the replica set supports set-based selectors.



|  |
| --- |
| #replicaSet.yaml  apiVersion: extensions/v1beta1 --------------------->1  kind: ReplicaSet --------------------------> 2  metadata:  name: Tomcat-ReplicaSet  spec:  replicas: 3  selector:  matchLables:  tier: Backend ------------------> 3  matchExpression:  { key: tier, operation: In, values: [Backend]} --------------> 4  template:  metadata:  lables:  app: Tomcat-ReplicaSet  tier: Backend  labels:  app: App  component: neo4j  spec:  containers:  - name: Tomcat  image: tomcat: 8.0  ports:  - containerPort: 7474 |
| #replicaSet.yaml  apiVersion: apps/v1  kind: ReplicaSet  metadata:  labels:  app: webapp  tier: frontend  name: webapp  spec:  replicas: 3  selector:  matchLabels:  tier: frontend  template:  metadata:  labels:  tier: frontend  spec:  containers:  -  image: "webapp:2.0"  name: webapp |

****Setup Details****

****apiVersion: extensions/v1beta1**** → In the above code, the API version is the advanced beta version of Kubernetes which supports the concept of the replica set.

****kind: ReplicaSet****→ We have defined the kind as the replica set which helps kubectl to understand that the file is used to create a replica set.

****tier: Backend****→ We have defined the label tier as backend which creates a matching selector.

****{key: tier, operation: In, values: [Backend]}****→ This will help match expression to understand the matching condition we have defined and in the operation which is used by match label to find details.

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

Run the above file using kubectl and create the backend replica set with the provided definition in the yaml file.

-------------------------------------------------------------------------------------------Description: Kube Service Backend Replicaset

## **ReplicaSets**

When you use ****Deployments****, you don’t have to worry about managing the ReplicaSets that they create.

Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don’t require updates at all.

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

****Create**** a replica set:

kubectl apply -f replicaSet.yaml

Output- replicaset.apps "webapp" created

kubectl get rs

NAME DESIRED CURRENT READY AGE

webapp 3 3 3 49s

Now check the pods:

kubectl get pods

NAME READY STATUS RESTARTS AGE

webapp-jcs7v 1/1 Running 0 32s

webapp-kpl9h 1/1 Running 0 32s

webapp-rxwr5 1/1 Running 0 32s

****Display**** your replica sets:

kubectl get replicasets

Get ****details****of a replica set:

kubectl describe replicaset webapp

****Edit**** a replica set:

kubectl edit replicaset webapp

****Scale**** a replica set:

kubectl scale --replicas=5 rs webapp

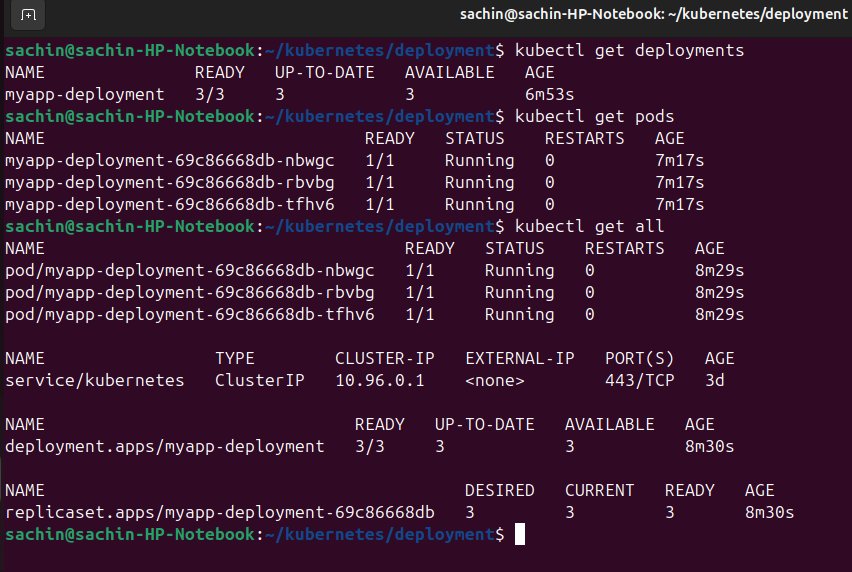
output of above command - replicaset.apps "webapp" scaled

Delete a replica set:

kubectl delete replicaset/webapp

Output of above command: replicaset.apps "webapp" deleted

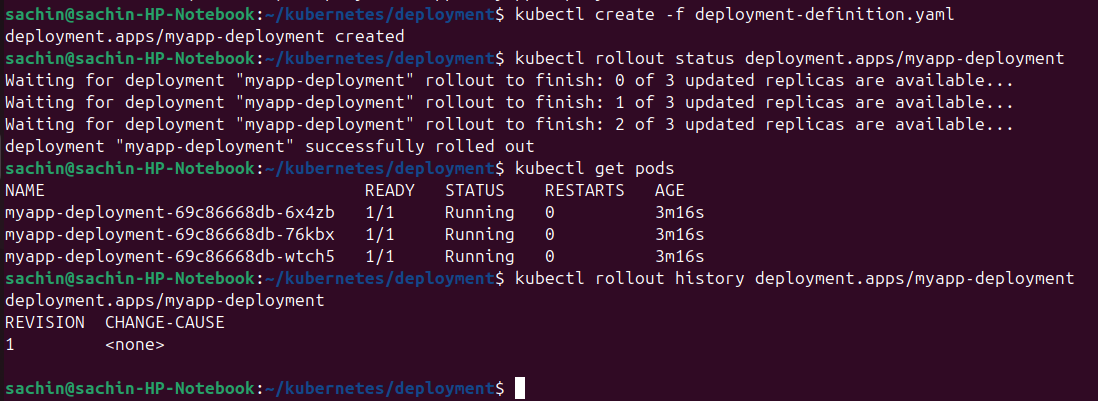
Deployment:

h

sachin@sachin-HP-Notebook:~/kubernetes/deployment$ kubectl get rs

NAME DESIRED CURRENT READY AGE

myapp-deployment-69c86668db 3 3 3 22h



As we create a deployment with replicas, we can see the replicas being created 1 by 1 by using the command:

*kubectl rollout status deployment.apps/myapp-deployment*

Command to check the deployment history, whether there is any old version of deployment:

|  |
| --- |
| kubectl rollout history deployment.apps/myapp-deployment |

|  |
| --- |
| deployment.apps/myapp-deployment  REVISION CHANGE-CAUSE  1 <none> |

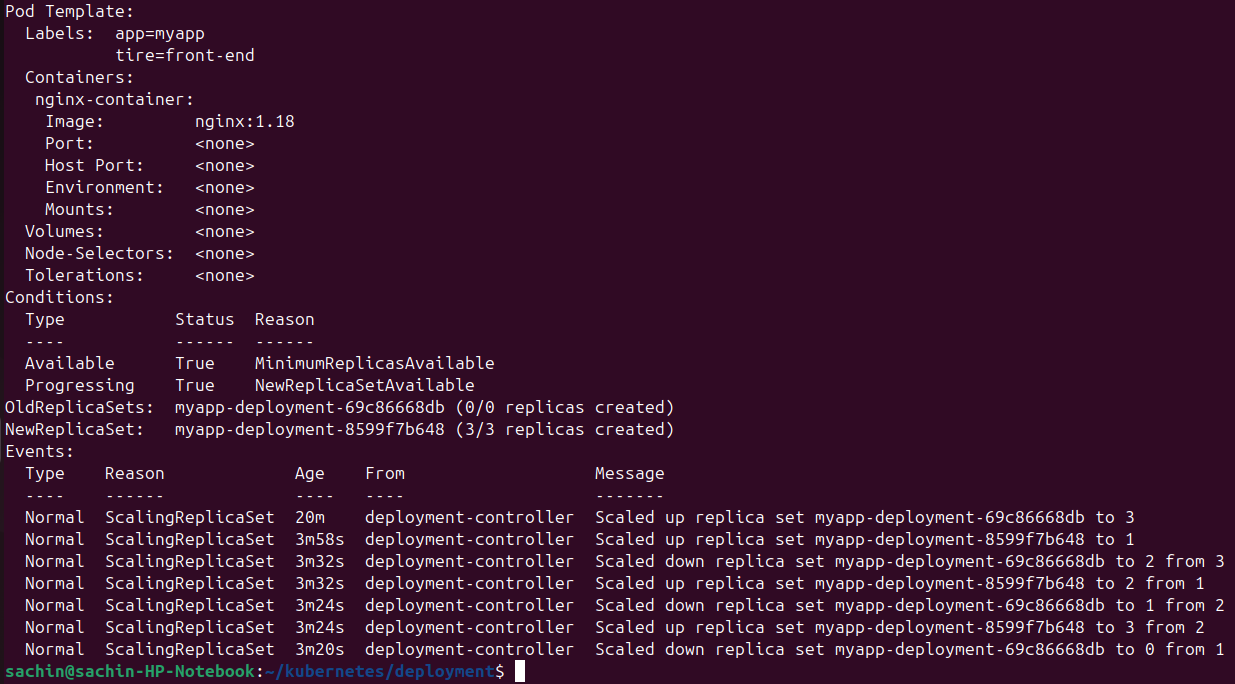
*As* per the above output of deployment history, there is no change in deployment.

**Default update strategy is rolling update, kubernetes deployments are intelligent enough to make sure that, there are some running pods in the system. So that end users are not affected and does the update 1 after the other.**

I updated the image as nginx:1.18 using deployment edit command:

**kubectl edit deployment myapp-deployment –record**

Above command update the deployment. And by doing the describe command on deployment command , we can see the changes like image update and new pods rollsout one by one.

**

We can rollout back to the previous version too, using the below command:

**kubectl rollout undo deployment/<deployment-name>**

Above command will rollback to the previous version.

Kubernetes service:

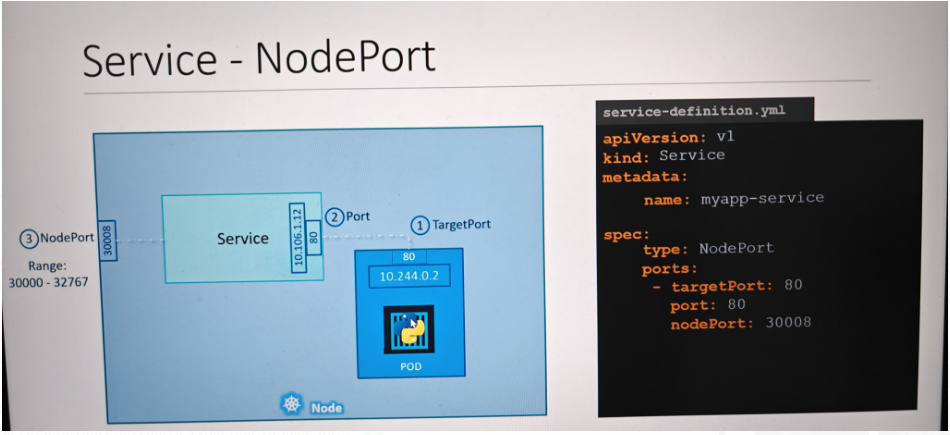
There are 3 types of services in kubernetes:

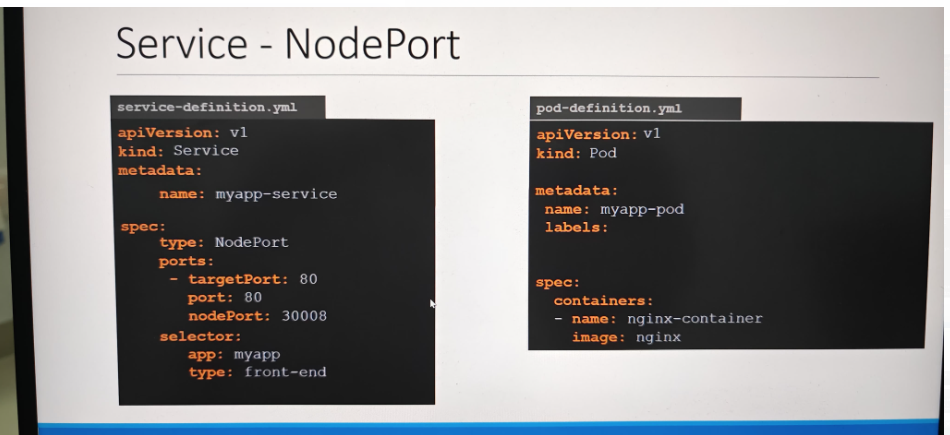
* NodePort,
* ClusterIP(Service creates a virtual IP inside the cluster, to enable comunications between different services. Such as front-end server to a backend server).
* LoadBalancer.

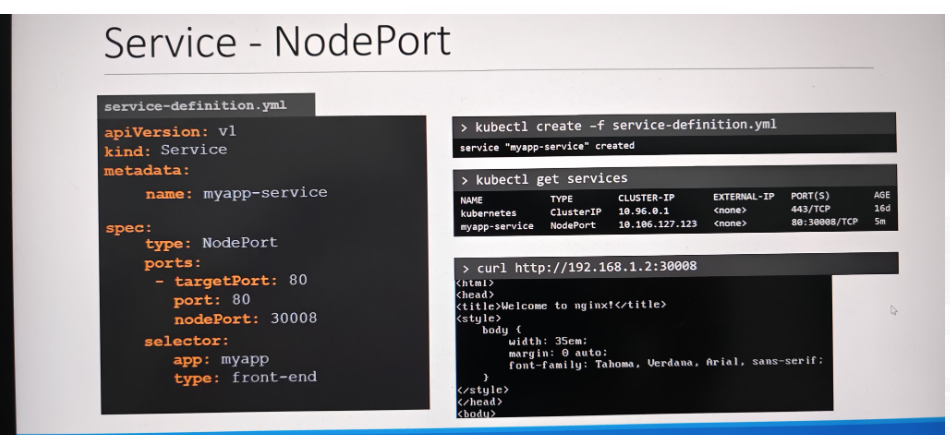
A Kubernetes **NodePort** is a type of Service that allows you to expose a specific application running in a cluster to external traffic. It does this by assigning a static port on each node in the cluster, which can then be accessed from outside the cluster.

### How NodePort Works

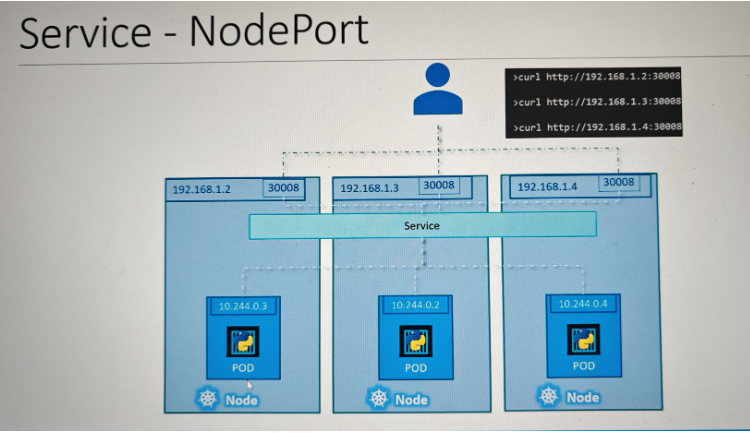
1. **Service Creation**: When you create a NodePort service, Kubernetes allocates a port from a default range (usually 30000–32767) on each node in the cluster.
2. **Routing Traffic**: When a request is sent to any node's IP address on the allocated NodePort, Kubernetes routes that request to one of the Pods that the service is targeting, based on its internal load-balancing mechanism.
3. **Pod Communication**: The Pods behind the service can be accessed via the cluster's internal network without needing to know their specific IP addresses.



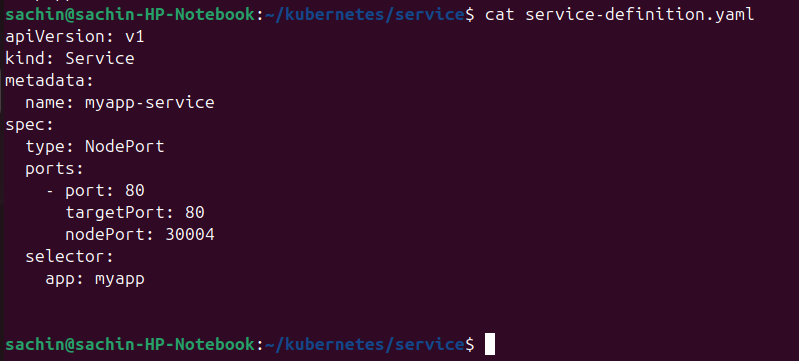


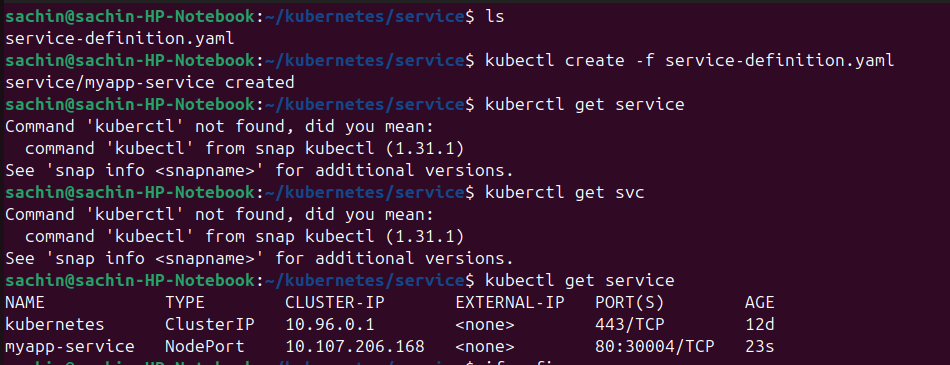
s

When we have multi-node kubernetes cluster, then we can connect using any IP address of POD(refer below screenshot).

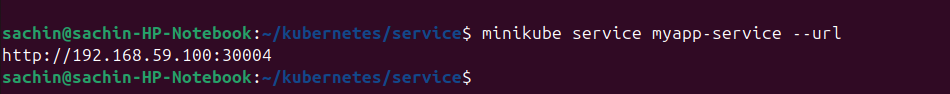


NodePort under spec section is a port for kubernetes node.

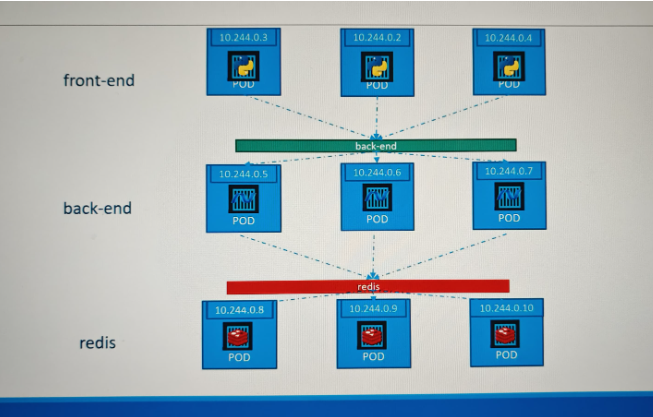




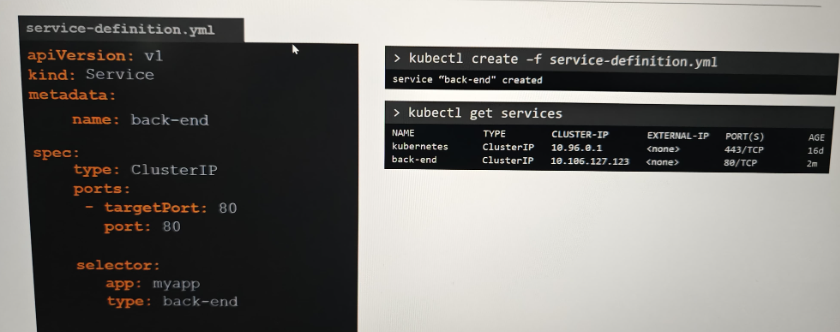
Now get the URL using minikube command to use nginx server



services – ClusterIP



d

Now the problem comes, that we can not rely on the IP addresses of our applications(frontend and backend), since IP addresses are not static. PODs can go down and new PODs can replace them anytime.

To Resolve this issue, a service created for the backend pods will help group all the backend pods together and provide a single interface for other pods to access this service.

The requests are forwarded to one of the pods under the service randomly.

Similarly, create additional services for redis and allow the backend pods to access the redis system.

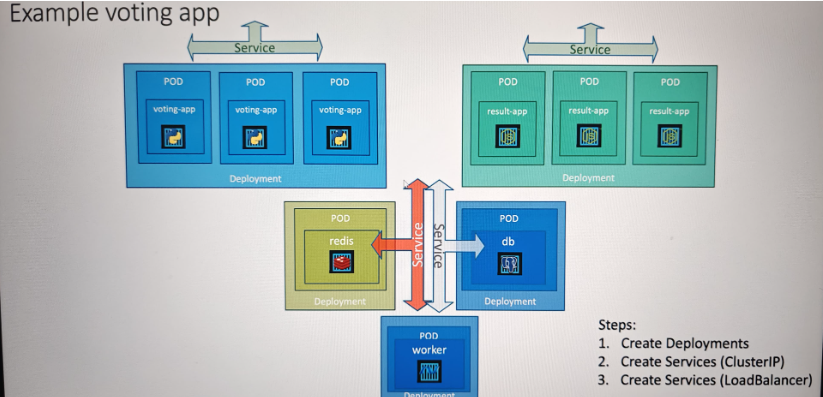
This enables us easily and effectively deploy a microservices based application on kubernetes cluster.

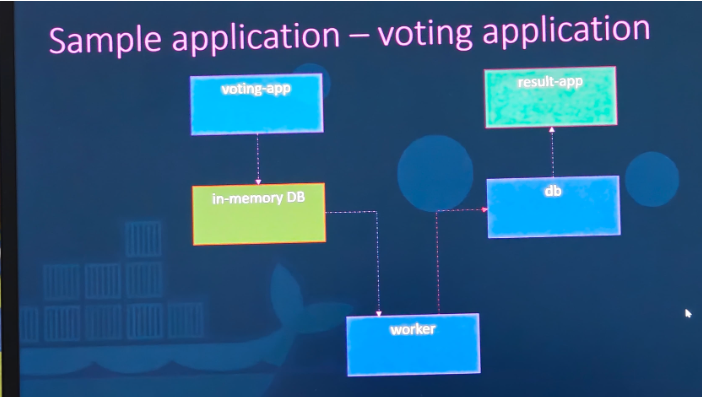
Each layer can now scale or move as required without impacting communication between the various servies.

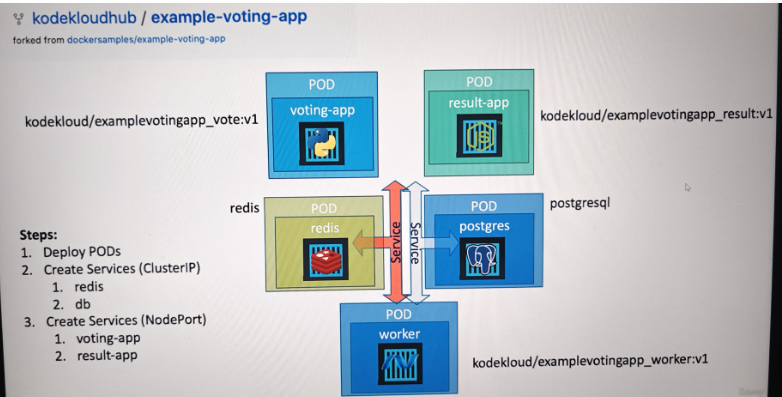
Each service gets an IP and name assigned to it inside the cluster, and that is the name that should be used by other paths to access the service.

This type of service is called clusterIP.

End to end project.

S

s

a