Core Concepts Section Introduction

Course Objectives

- Core Concepts
  - Cluster Architecture
  - API Primitives
  - Services & Other Network Primitives
- Scheduling
- Logging Monitoring
- Application Lifecycle Management
- Cluster Maintenance
- Security
- Storage
- Networking
- Installation, Configuration & Validation
- Troubleshooting

Cluster Architecture

1. Kubernates Architecture
2. ETCD For Beginners
3. ETCD in Kubernates
4. Kube-api server
5. Controller Manager
6. Kube Scheduler
7. Kubelet
8. Kube-proxy

1. Kubernates Architecture

For this course we would be using an analogy of Ships to understand the components of a Kubernates architecture.

Thus Lets consider an example of two ships at sea, one being the cargo ship and the other being the control ship, now the control ship manages the cargo ships, its stipulates which ships goes where and the cargo it should carry, it generally monitors, all the cargo ships running.

In a similar vein relating this to our Kubernates architecture, we have several nodes (computers) that carry and run our container applications **(worker nodes)**, this nodes are controlled by a component of kubernates called the control plane **(master node)**.

Generally the master node, manages, plan, schedule and monitor worker nodes.

Back to our ship analogy, we understand that the control ship is in charge of assigning the loading and unloading of cargo, on and off the cargo ships, specifically assigning which cargo, goes on which ship.

Because of this it is important that we keep track of this information, somewhere.

Relating this back to the kubernates architecture, **ETCD**, inside the master node is charge of keeping tracking of the information about the containers in key value store. Basically its a database in our kubernates cluster. To keep track of this information.

Again, Following this pattern, we know that the control ship assigns and loads cargo to a cargo ship based on its Capacity, size e.t.c

For kubernates, we Load containers onto nodes in similar way, with a component called **kube-scheduler,** kube-scheduler is in charge of loading cargo onto the a particular node, based on the capacity (cpu), memory, tolerance, or any other constrains e.t.c

Now just like a normal ship, we normally have several Departments to deals with several aspect of the ship for example the operations team, is in charge of ship handling, traffic control, e.t.c, while other department such as the cargo ship deals with containers, making sure they are replaced when one is damaged e.t.c

In a similar vein, We have **controllers** in kubernates that handles different Parts of operations in the kubernates cluster, the controller manager has components such as node controller, replication controller all handing nodes (if one is down) and getting the desired amount of nodes available (replication) to match the desired state.

One important question to ask thou, is how all these components interact with one another, and who manages them in a broad scope.

This is done by the **kube-api-server,** which manges all these components on a broad level

Now lets Turn our attention back to the cargo ships

Every Cargo ship, has a captain, that monitors activties on each ship, talking to the control ship and getting information e.t.c

In a kubernates cluster, a **kubelet** is a process that runs on a container and interacts with the kube-api-server, getting information and sending information back

But the kubelet is limited because each one runs on a specific container, what happens when two containers want to communicate with one another. For example in a three tier achetcture, with each server either running the client (frontend), backend and the database.

How those the client communicate or connect to the backend and the backend to the database, e.t.c, this communication in a kubernates cluster is enabled by the **kube-proxy** running on the worker nodes

2. ETC FOR BEGINERS

Objectives

- What is ETCD
- What is key value store
- How to get started quickly
- How to operate ETCD

Later

- What is a distributed System
- How ETCD works
- RAFT Protocol
- Best Practice on number of nodes

What is ETCD


ETCD is a distributed reliable key-value store that is Simple, Secure & Fast

What is key value store

A key value store, is like a relational database where values are stored and retrieved.

Given say, a table with a key called name and a value John Doe, whenever we want to access “John Doe” all we do is get the key name, and the we get “john Doe back”

key-value store

Key	Value
Name	John Doe
Age	45
Location	New York
Salary	5000

Put Name “John Doe”

Get Name

“John Doe”

Tabular/Relational Databases

Name	Age	Location
John Doe	45	New York
Dave Smith	34	New York
Aryan Kumar	10	New York
Lauren Rob	13	Bangalore
Lily Oliver	15	Bangalore

Install ETCD

1. Download Binaries

```
curl -L https://github.com/etcd-io/etcd/releases/download/v3.3.11/etcd-
v3.3.11-linux-amd64.tar.gz -o etcd-v3.3.11-linux-amd64.tar.gz
```

2. Extract

```
tar xzvf etcd-v3.3.11-linux-amd64.tar.gz
```

3. Run ETCD Service

```
./etcd
```

After installation, And Running the ETCD service

This starts a Process that runs on port **2379**

To save a key in the etcd Database use the command

- ./etcdctl set key1 value1

To Retrive a value in the etcd Database use the command

- ./etcdctl get key1

To get all the list of commands use

- ./etcdctl

Operate ETCD

3. Run ETCD Service

```
./etcd
```

```
./etcdctl set key1 value1
```

```
./etcdctl get key1
value1
```

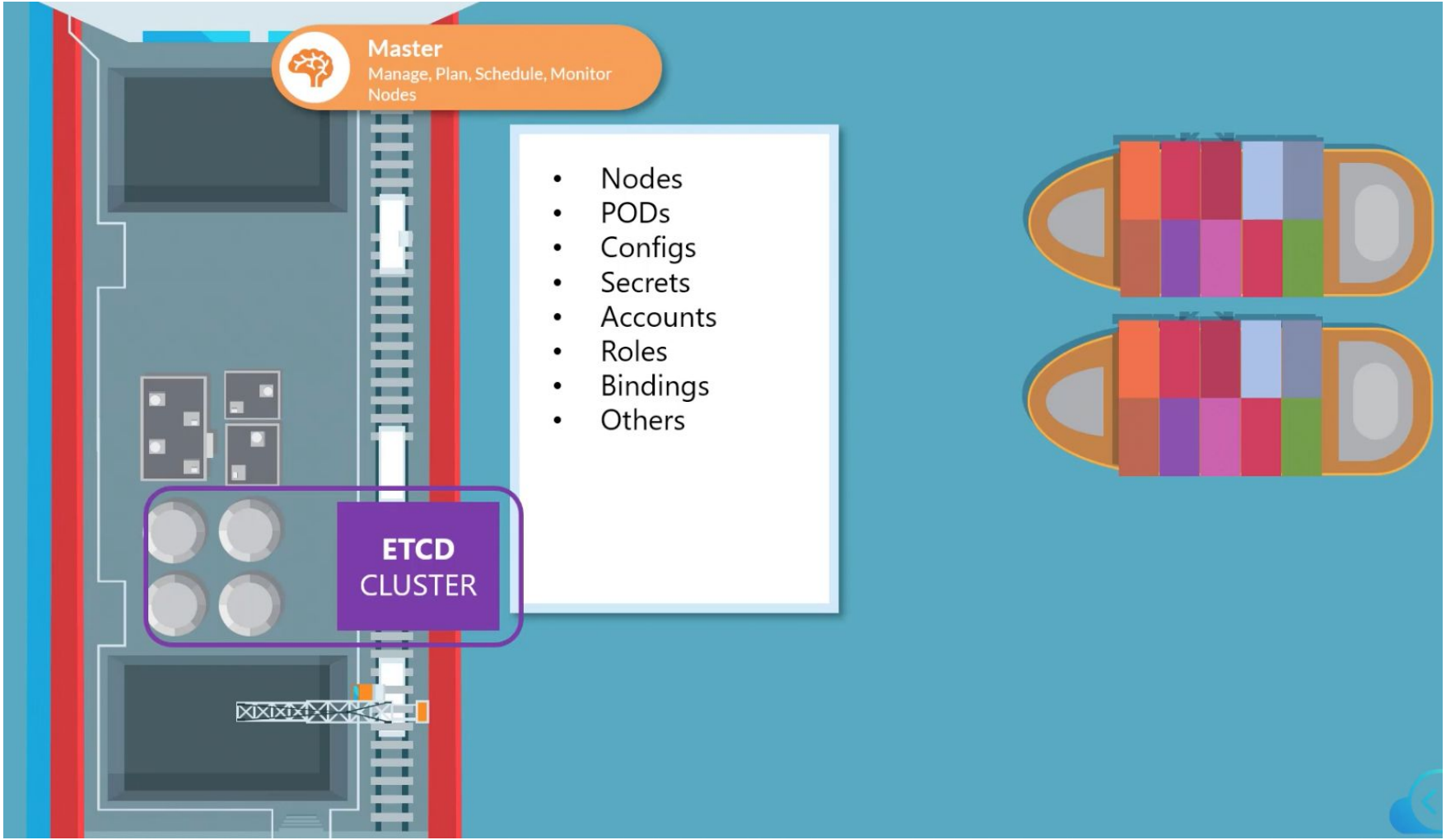
```
./etcdctl
NAME:
   etcdctl - A simple command line client for etcd.

COMMANDS:
     backup          backup an etcd directory
     cluster-health  check the health of the etcd cluster
     mk              make a new key with a given value
     mkdir           make a new directory
     rm              remove a key or a directory
     rmdir           removes the key if it is an empty directory or a key-value pair
     get             retrieve the value of a key
```

3. ETCD in Kubernates

The ETCD cluster stores several information regarding our kubernates cluster.

Informations such as the

- Pods
- Nodes
- Config
- Secrets
- Accounts
- Roles
- Bindings. e.t.c


Master
Manage, Plan, Schedule, Monitor Nodes
Nodes
PODs
Configs
Secrets
Accounts
Roles
Bindings
Others
ETCD
CLUSTER

There are Two ways to get started with ETCD in kubernates

1. Is If you are setting up a cluster by you self, in which you would have to set up and install the ECTD binaries your self

Setup - Manual

```
wget -q --https-only \
    "https://github.com/coreos/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz"
```

etcd.service

```
ExecStart=/usr/local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \\
  --advertise-client-urls https://${INTERNAL_IP}:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster controller-0=https://${CONTROLLER0_IP}:2380,controller-1=https://${CONTROLLER1_IP}:2380 \\
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
```

2. Is If you are setting up a cluster with Kubeadm, kubeadm install the ECTD binaries in the kube-sytem namespace

You can access the ectd database with the ectd control utility in this pod, to list all keys in the ectd database, write

Kubectl exec etcd-master -n kube-system etcdctl get / - - pre

Setup - kubeadm

```
kubectl get pods -n kube-system
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-78fcdf6894-prwvl	1/1	Running	0	1h
kube-system	coredns-78fcdf6894-vqd9w	1/1	Running	0	1h
kube-system	etcd-master	1/1	Running	0	1h
kube-system	kube-apiserver-master	1/1	Running	0	1h
kube-system	kube-controller-manager-master	1/1	Running	0	1h
kube-system	kube-proxy-f6k26	1/1	Running	0	1h
kube-system	kube-proxy-hnzsw	1/1	Running	0	1h
kube-system	kube-scheduler-master	1/1	Running	0	1h
kube-system	weave-net-924k8	2/2	Running	1	1h
kube-system	weave-net-hzfcz	2/2	Running	1	1h

Explore ETCD

```
kubectl exec etcd-master -n kube-system etcdctl get / --prefix -keys-only
/registry/apiregistration.k8s.io/apiservices/v1.
/registry/apiregistration.k8s.io/apiservices/v1.apps
/registry/apiregistration.k8s.io/apiservices/v1.authentication.k8s.io
/registry/apiregistration.k8s.io/apiservices/v1.authorization.k8s.io
/registry/apiregistration.k8s.io/apiservices/v1.autoscaling
/registry/apiregistration.k8s.io/apiservices/v1.batch
/registry/apiregistration.k8s.io/apiservices/v1.networking.k8s.io
/registry/apiregistration.k8s.io/apiservices/v1.rbac.authorization.k8s.io
/registry/apiregistration.k8s.io/apiservices/v1.storage.k8s.io
/registry/apiregistration.k8s.io/apiservices/v1beta1.admissionregistration.k8s.io
```

Run inside the etcd-master POD

- Registry
  - minions
  - pods
  - replicasets
  - deployments
  - roles
  - secrets

In a highly available Environment ECTD, where they are multiple master nodes in the kubernates cluster,

then we will have multiple ETCD instances in the Kubernates cluster.

ETCD in HA Environment

etcd.service

```
ExecStart=/usr/local/bin/etcd \\
  --name ${ETCD_NAME} \\
  --cert-file=/etc/etcd/kubernetes.pem \\
  --key-file=/etc/etcd/kubernetes-key.pem \\
  --peer-cert-file=/etc/etcd/kubernetes.pem \\
  --peer-key-file=/etc/etcd/kubernetes-key.pem \\
  --trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-trusted-ca-file=/etc/etcd/ca.pem \\
  --peer-client-cert-auth \\
  --client-cert-auth \\
  --initial-advertise-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-peer-urls https://${INTERNAL_IP}:2380 \\
  --listen-client-urls https://${INTERNAL_IP}:2379,https://127.0.0.1:2379 \\
  --advertise-client-urls https://${INTERNAL_IP}:2379 \\
  --initial-cluster-token etcd-cluster-0 \\
  --initial-cluster controller-0=https://${CONTROLLER0_IP}:2380,controller-1=https://${CONTROLLER1_IP}:2380 \\
  --initial-cluster-state new \\
  --data-dir=/var/lib/etcd
```

ETCD – Commands (Optional)

(Optional) Additional information about ETCDCTL UtilityETCDCTL is the CLI tool used to interact with ETCD.ETCDCTL can interact with ETCD Server using 2 API versions – Version 2 and Version 3. By default it's set to use Version 2. Each version has different sets of commands

For example, ETCDCTL version 2 supports the following commands:

```
etcdctl backup

etcdctl cluster-health

etcdctl mk

etcdctl mkdir

etcdctl set
```

Whereas the commands are different in version 3

```
etcdctl snapshot save
etcdctl endpoint health
etcdctl get
etcdctl put
```

ETCD In Kubernates – Commands (Optional)

Get etcd installed in kubernates

- Kubectl get pods -n kube-system
- Or
- Kubectl get pods --namespace=kube-system

Where -n is namespace

To set the right version of API set the environment variable ETCDCTL_API command

```
export ETCDCTL_API=3
```

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

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

```
--cacert /etc/kubernetes/pki/etcd/ca.crt
```

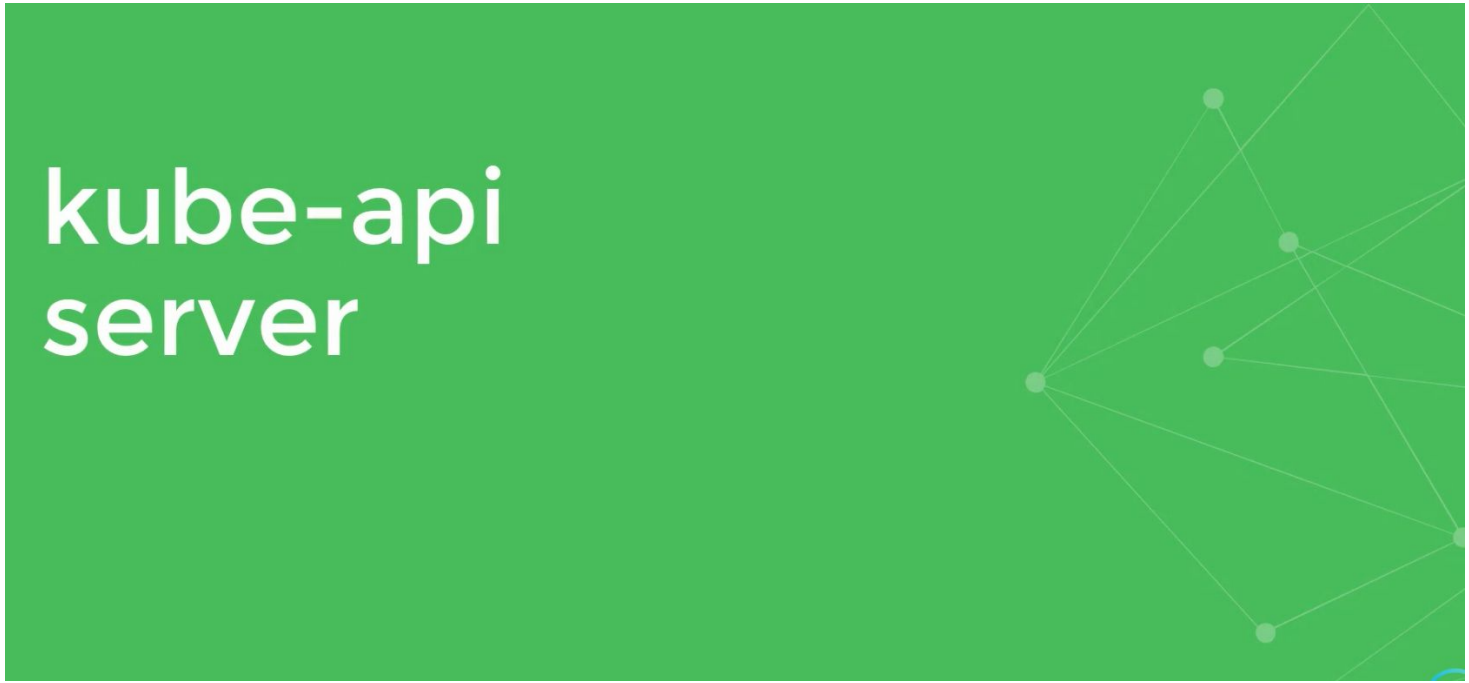
```
--cert /etc/kubernetes/pki/etcd/server.crt
```

```
--key /etc/kubernetes/pki/etcd/server.key
```

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

```
kubectl exec etcd-controlplane -n kube-system -- sh -c "ETCDCTL_API=3 etcdctl get /
--prefix --keys-only --limit=10 --cacert /etc/kubernetes/pki/etcd/ca.crt --cert
/etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key"
```

4. Kube-api server

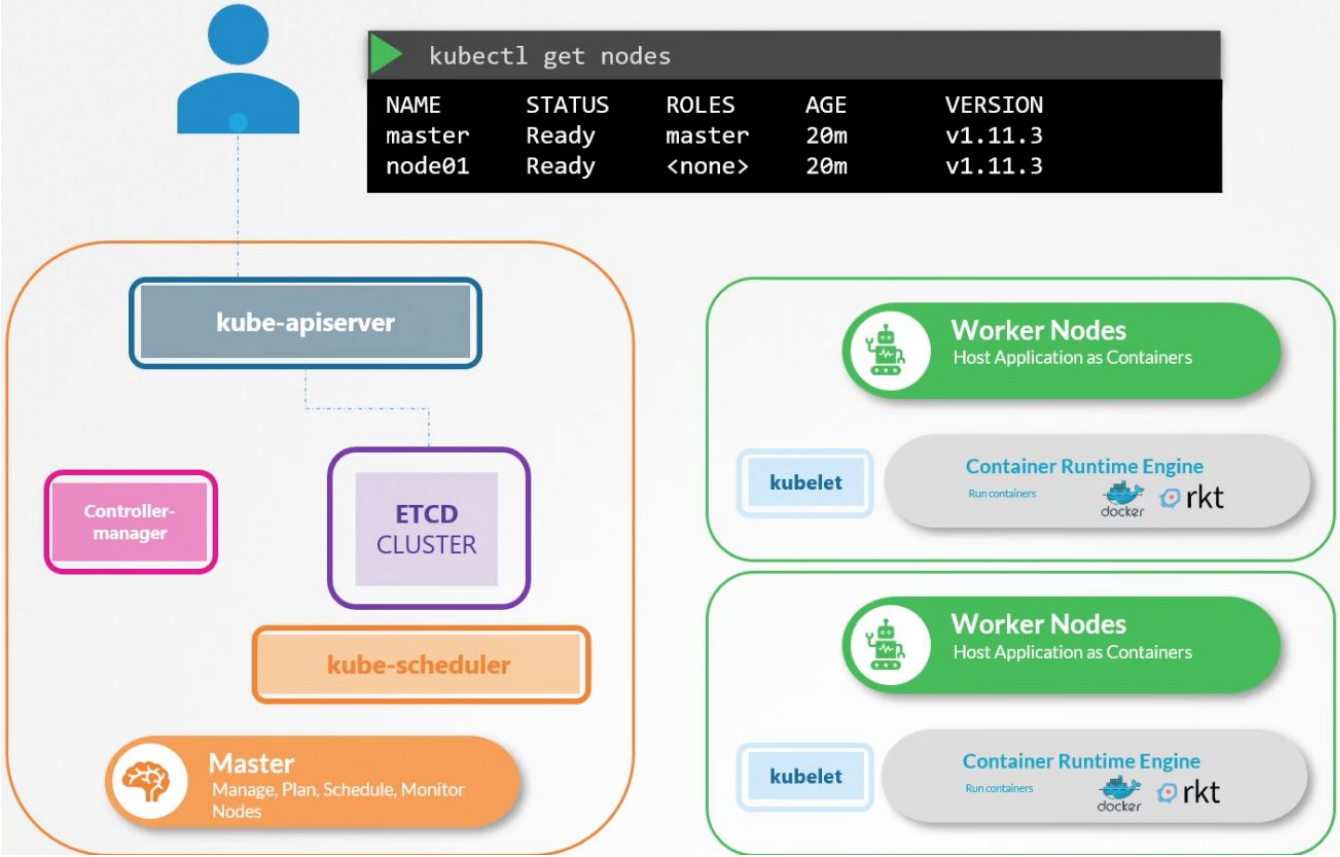


Like we discussed the kube-api-server, is the Primary Management Component in kubernates, it is the entry point to the kubernates cluster

For example when we make an kubectl command to get information from the cluster

The Kube-api-server, gets this information from ETCD, and sends a response back to the client

Kubernetes Architecture



The kube-api-server also performs the following functions

Kube-api Server

1. Authenticate User
2. Validate Request
3. Retrieve data
4. Update ETCD
5. Scheduler
6. Kubelet

Kube Controller Manager

4 - **Kube Controller Manager:** is a process that manages and monitors the state of various components in the system, and works on bringing the entire system to the desired stated state

For Example the node controller, manges and monitors the nodes in a kubernates cluster, performing health checks every 5 seconds to make sure, the node is Reachable

If it finds a Node unreachable, it terms it unhealthy.

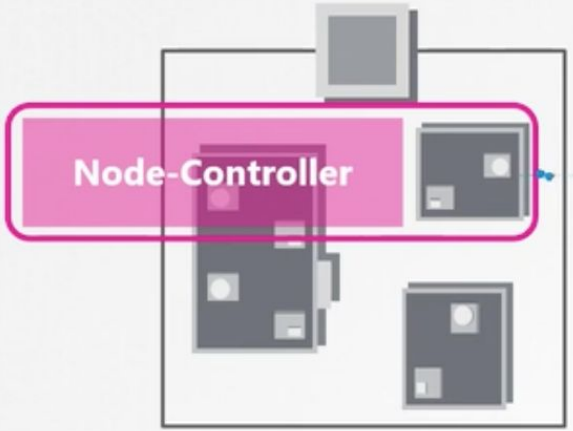
Controller

Watch Status

Remediate Situation

Node Monitor Period = 5s

Node Monitor Grace Period = 40s



kube-apiserver

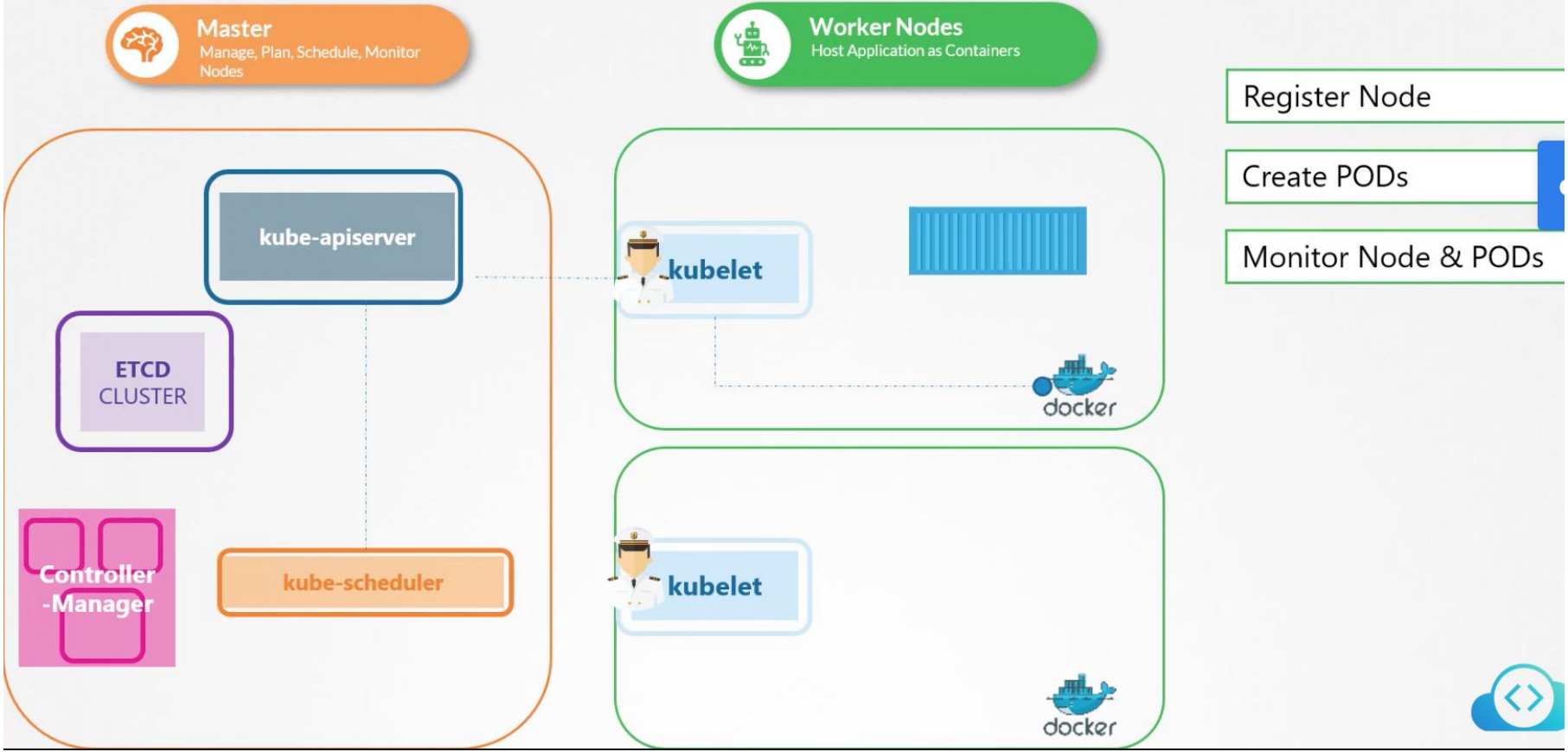
UNREACHABLE

```
kubectl get nodes
NAME       STATUS     ROLES    AGE   VERSION
worker-1   Ready      <none>   8d    v1.13.0
worker-2   NotReady   <none>   8d    v1.13.0
```

5. Kubelet

As stated Earlier the kubelet, is like the captain of the ship, he in charge of communicating with the API server, sending and receiving Information, from the kube-api-server. E.T.C

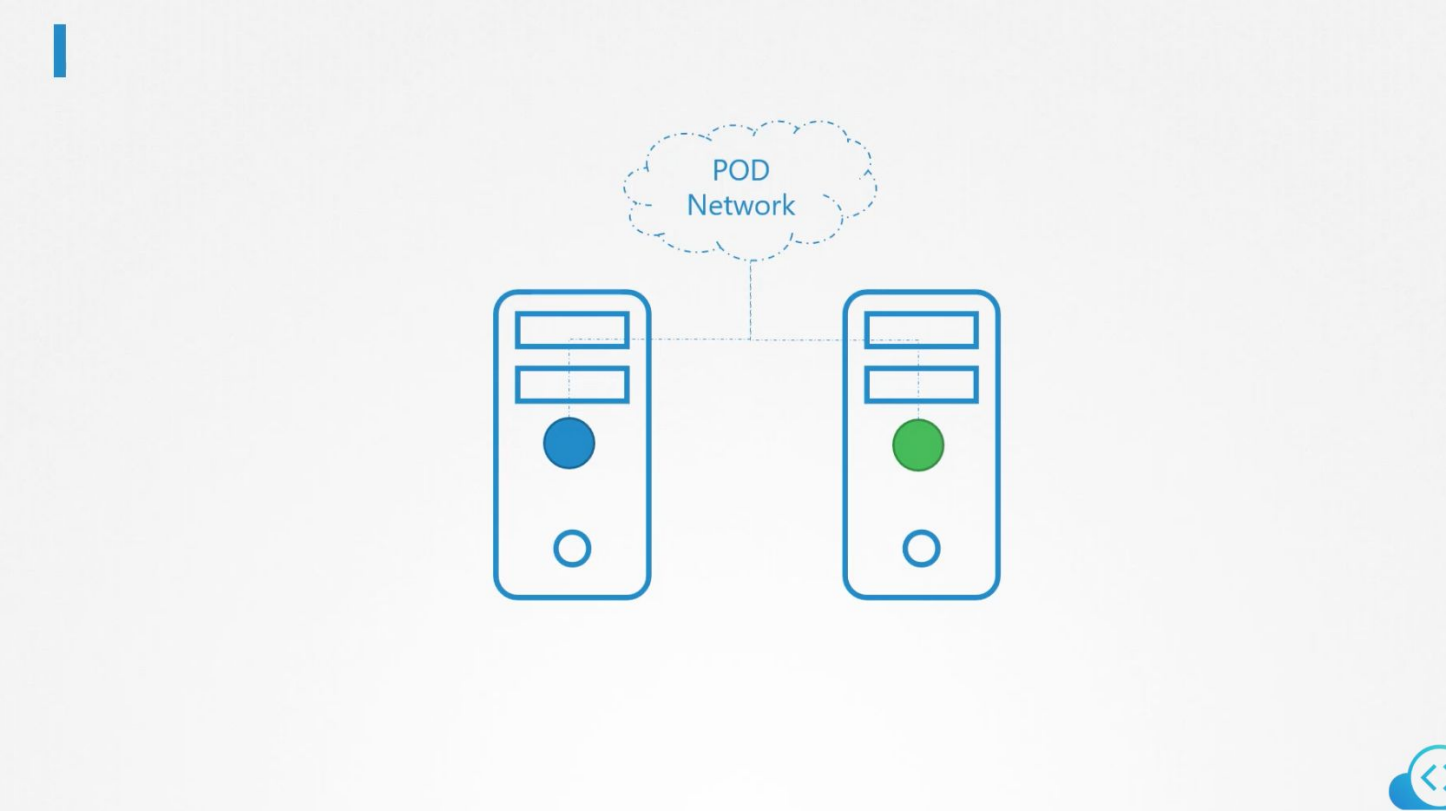
Kubernetes Architecture



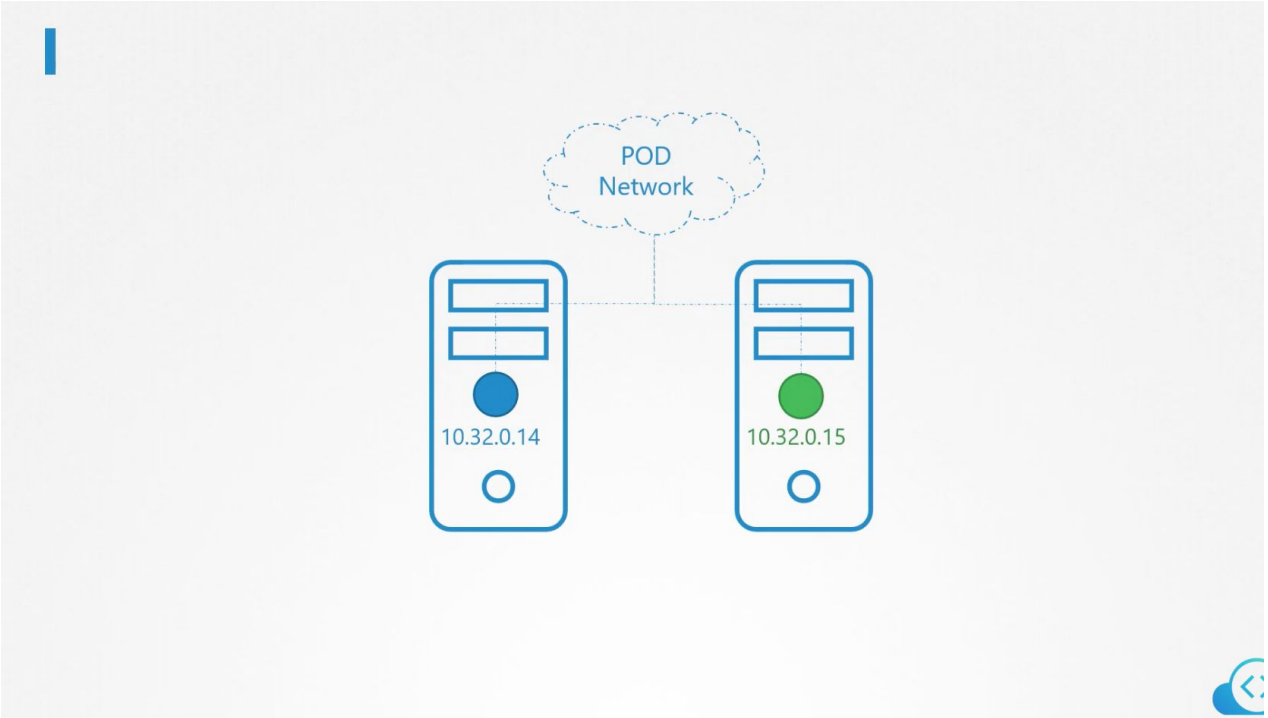
Kube-proxy

Within a kubernates cluster, Every Pod can reach another Pod, but how dose it achieve this?

This is done, via a virtual Network existing in the cluster i.e (Pod Network)

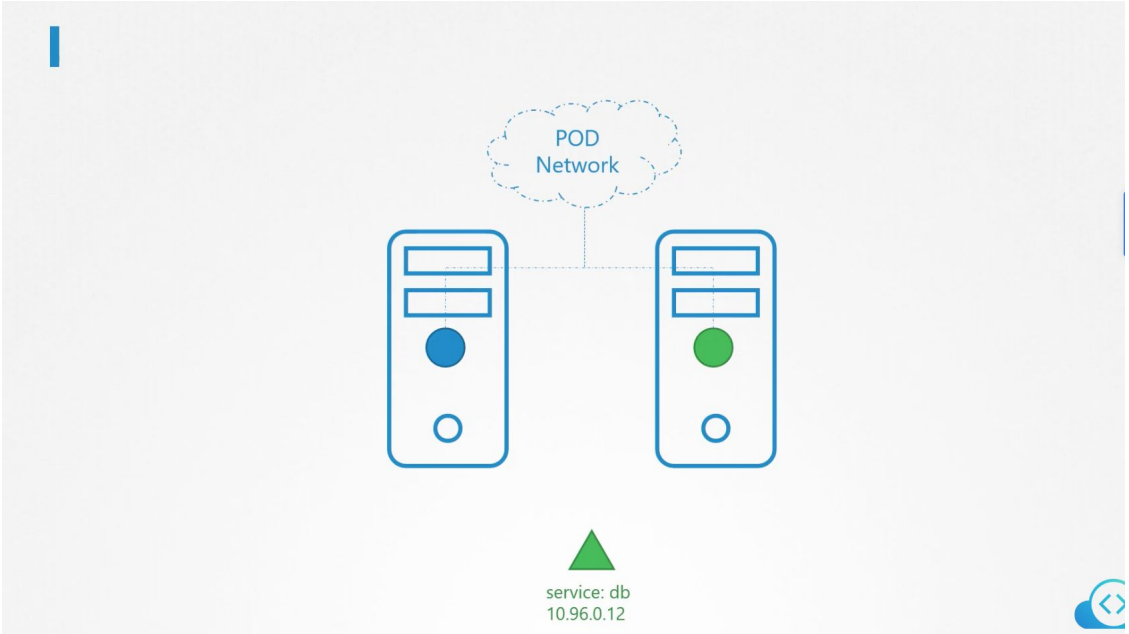


Let say For example in the First Pod, am having a web application running on that server (pod), and on the Second Pod, a database Application, the first Pod(web application) can connect to the second pod (Database application) through its IP address



But because Pods are liable to Fail and be Replaced. This IP Address might not be the best way to connect to an application, cause when it gets replaced, we wont be connecting to the database via the same IP any More.

A way to solve this problem is to use a service

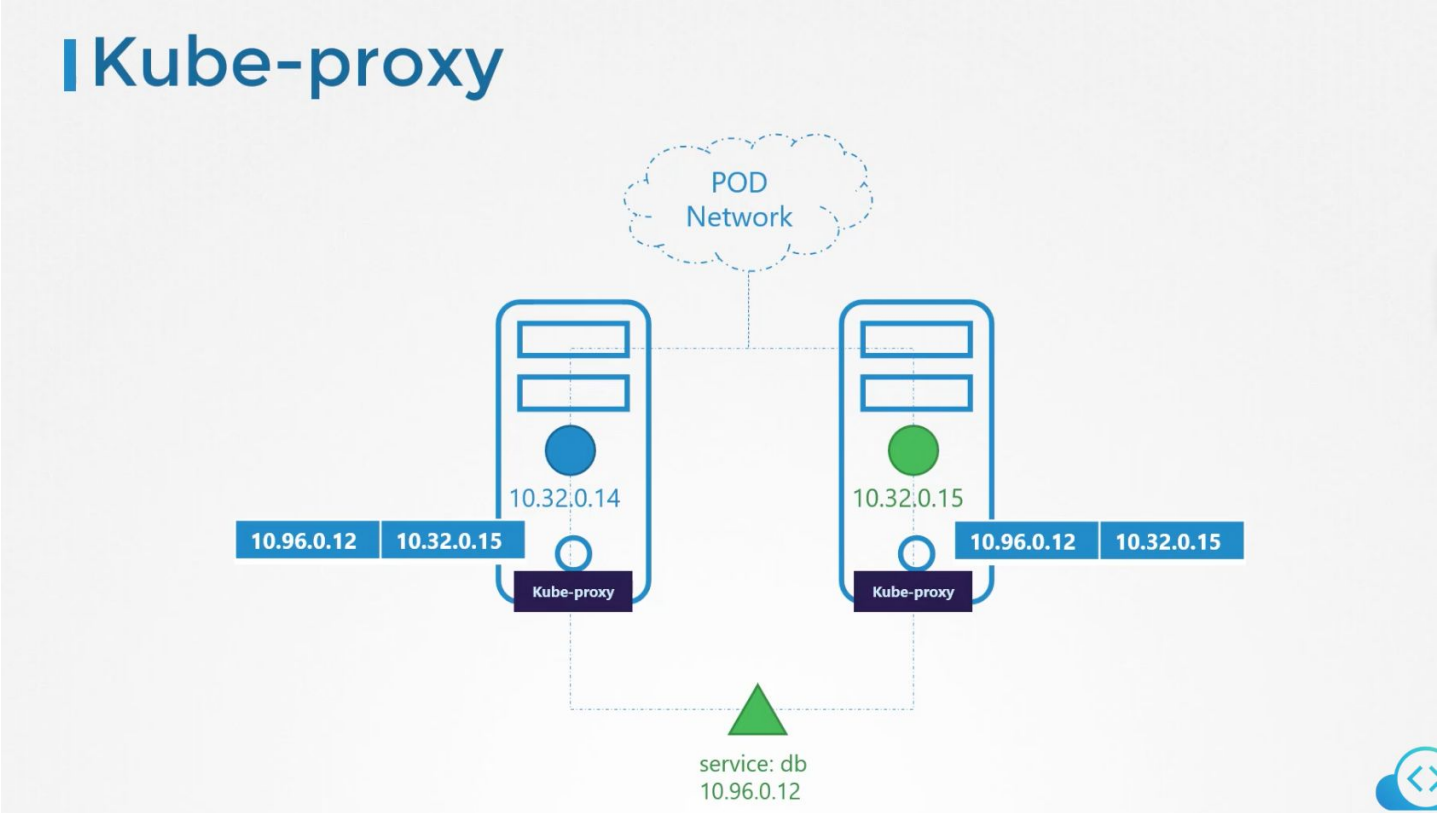


Now the Application can connect to another Pod via the service

The service is simple an abstraction that always has access to the access Pods IP address, even when a Pod is Replaced or changed

Also important to note that the service is an abstract component, that just lives in the clusters Memory, its not like a Pod that we can actually access or share, how do we then make the service available across the cluster

This is achieved with the kube-proxy component, which is a process that runs on a worker node, and always checks and access these services



PODS

POD

Assumptions

Docker Image

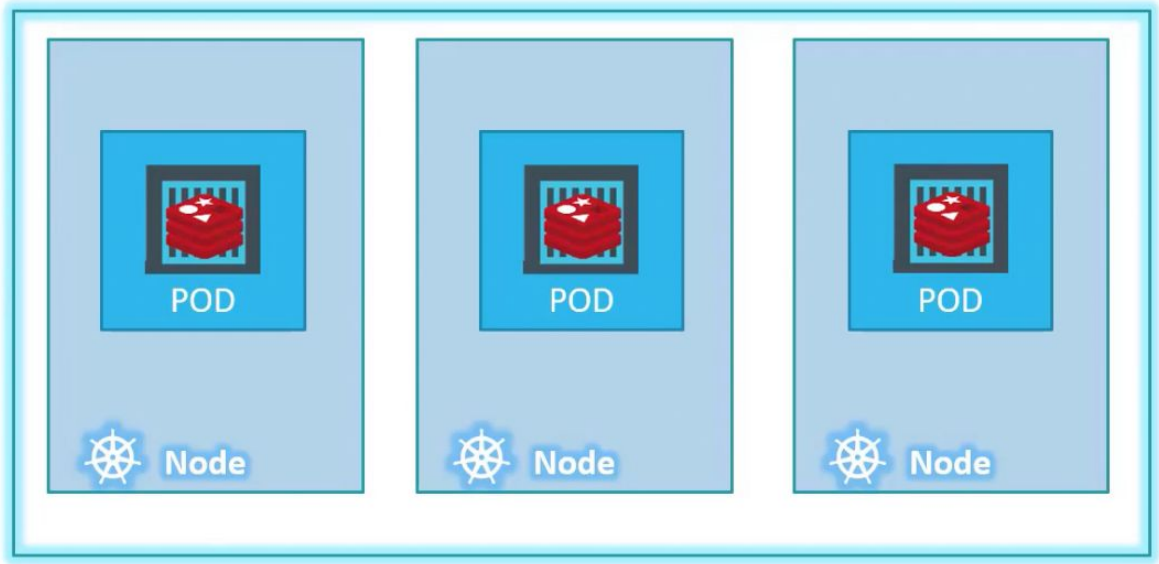
Kubernetes Cluster

A Pod is the smallest unit in a Kubernates cluster.

Our Goal with kubernates is to deploy our applications as containers, but in kubernates we usually intract with pods

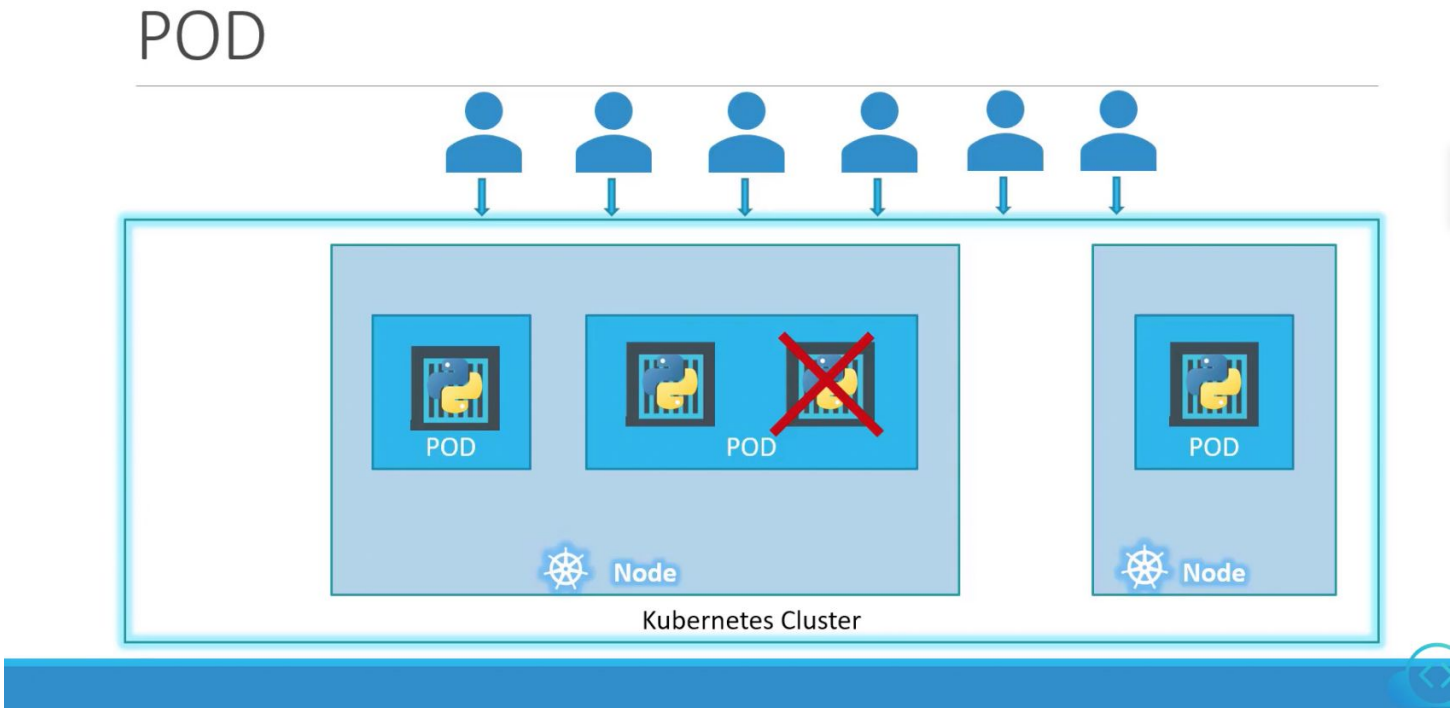
And these Pods runs a containers, which is what runs our application

POD



Inside of a Pod, we can have multiple containers running, but this is not recommended as it is,

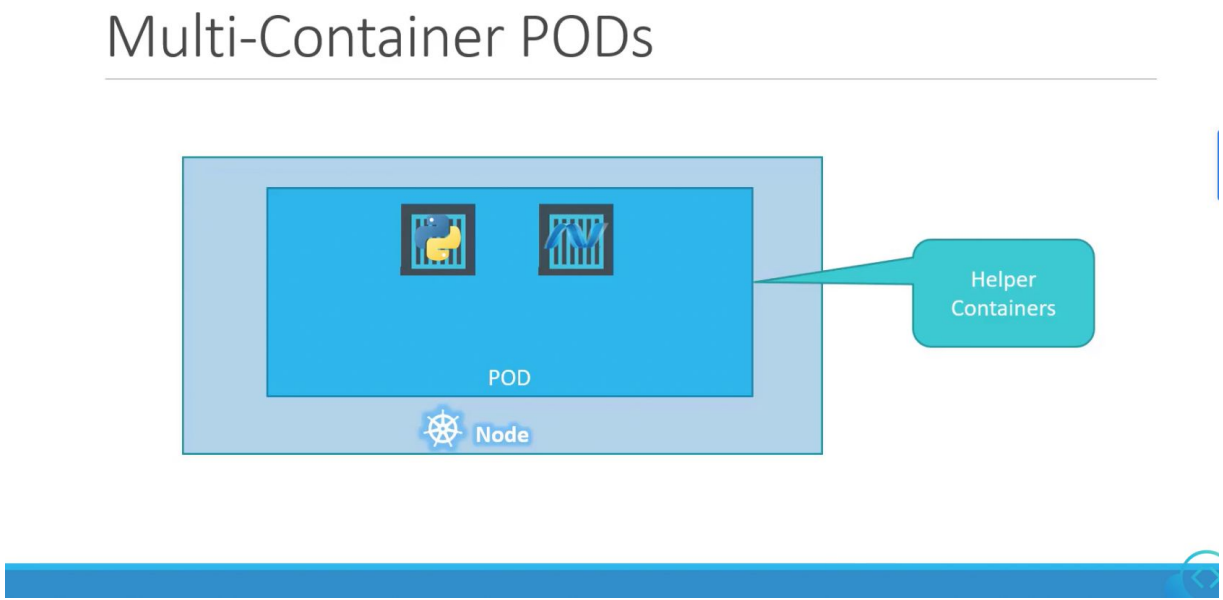
Usually a container runs in a single Pod, and More instances are generated as user Traffic increases



As stated Previously, we should not run two containers in a single,

But there might be cases where this is effienient, as Example is when one container, is acting as a helper (providing services to the main application).

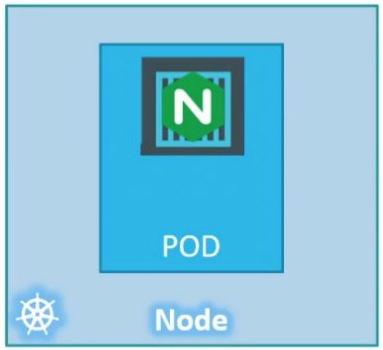
It might be better to have both running in the same Pod



Simple Example running a pod in kubernates

kubectl

```
kubectl run nginx --image nginx
```

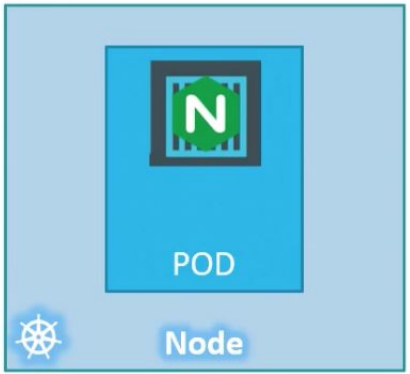


To see the list of pod in a particular namespace type

Kubectl get Pods

```
kubectl get pods
NAME    READY   STATUS              RESTARTS   AGE
nginx   0/1     ContainerCreating   0          6s
```

```
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0          34s
```



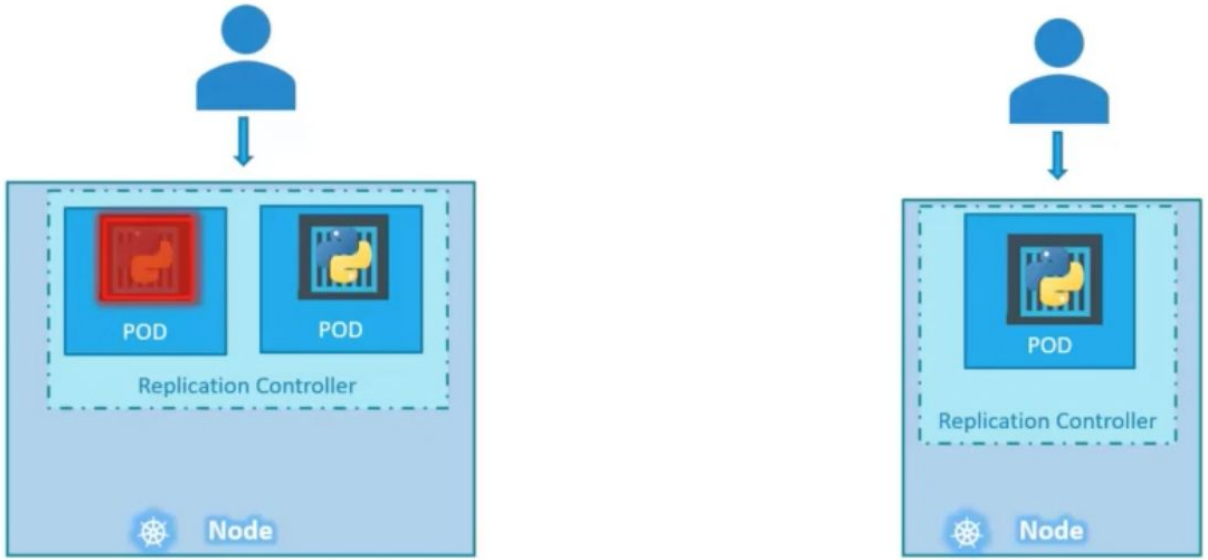
ReplicaSets

Replication Controller

Replica sets are simply copies of our pods and that keep the desired state of running Pods in the cluster to be the actual state of the pods

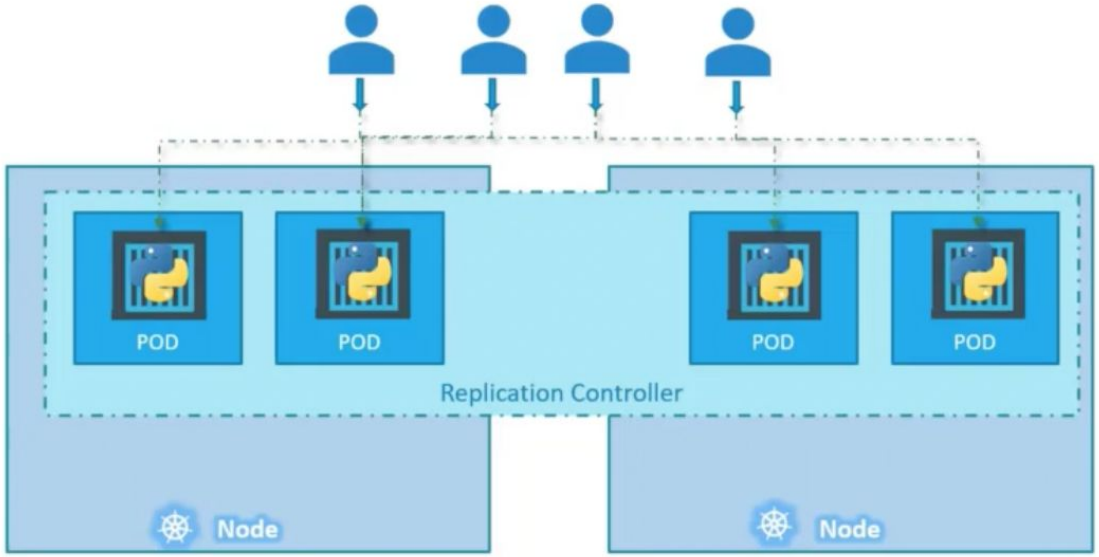
This is important because the Pods are replaced with one is down or unhealthy

High Availability

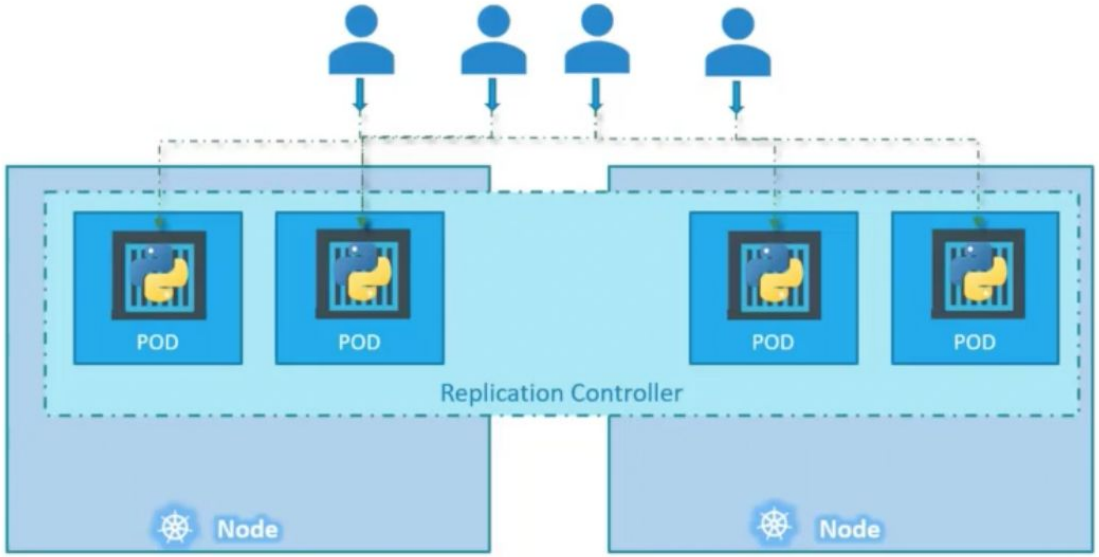


This is also Important for scaling and Load balancing when Traffic increases

Load Balancing & Scaling



Load Balancing & Scaling



Replication Controller

Replica Set

In the simplest Terms

A Replication Controller is an Older way to create Replicas

And a replica set is a newer way to do the same thing

How to create a Replication controller

We do this with a yaml, configuration file and in there we state the details of the pod to be replicated

rc-definition.yml

```yaml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
```

Replication Controller

POD

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
```

Underneath the Specs for the Pod, Add the number of desired replicas, that you want

```
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
```

To create a Repica set run the following command

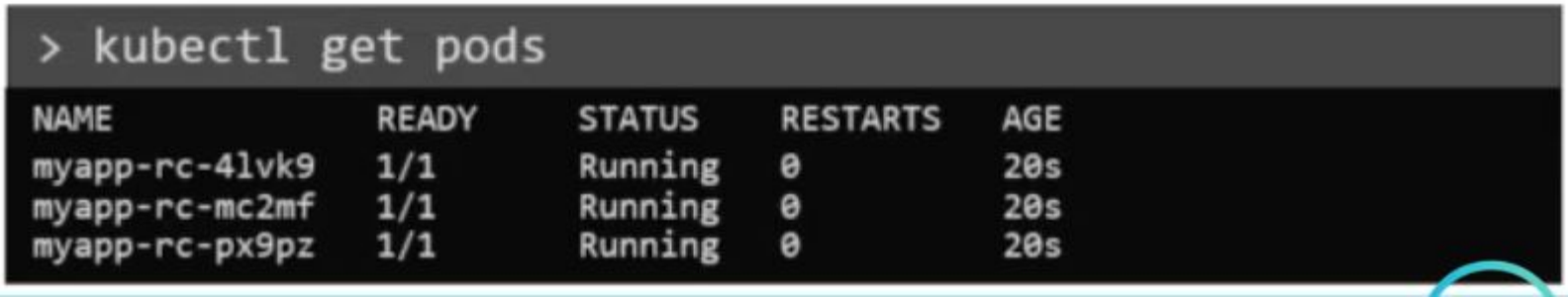
```
> kubectl create -f rc-definition.yml
replicationcontroller "myapp-rc" created
```

```
> kubectl get replicationcontroller
NAME       DESIRED   CURRENT   READY     AGE
myapp-rc   3         3         3         19s
```

Finally run

Kubectl get pods

To get all the replicas



How to create a Replica set

We still do this with a yaml, configuration file and in there we state the details of the pod to be replicated,

But with some minor differences from a replication Controller

replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
     name: myapp-pod
     labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
  selector:
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod
```

Some changes you should notice would be from the

- apiVersion
  - This is set to apps/v1
- Kind
  - ReplicaSet
- Selector: this is to state the pods in this replica set

```
selector:
    matchLabels:
        type: front-end
```

To create a replica set from a yam file

Run the following.

```
> kubectl create -f replicaset-definition.yml
replicaset "myapp-replicaset" created
```

```
> kubectl get replicaset
NAME              DESIRED   CURRENT   READY     AGE
myapp-replicaset   3         3         3         19s
```

```
> kubectl get pods
NAME                    READY     STATUS    RESTARTS   AGE
myapp-replicaset-9dd19   1/1       Running   0          45s
myapp-replicaset-9jtpx   1/1       Running   0          45s
myapp-replicaset-hq84m   1/1       Running   0          45s
```

Why Do we need Replica sets

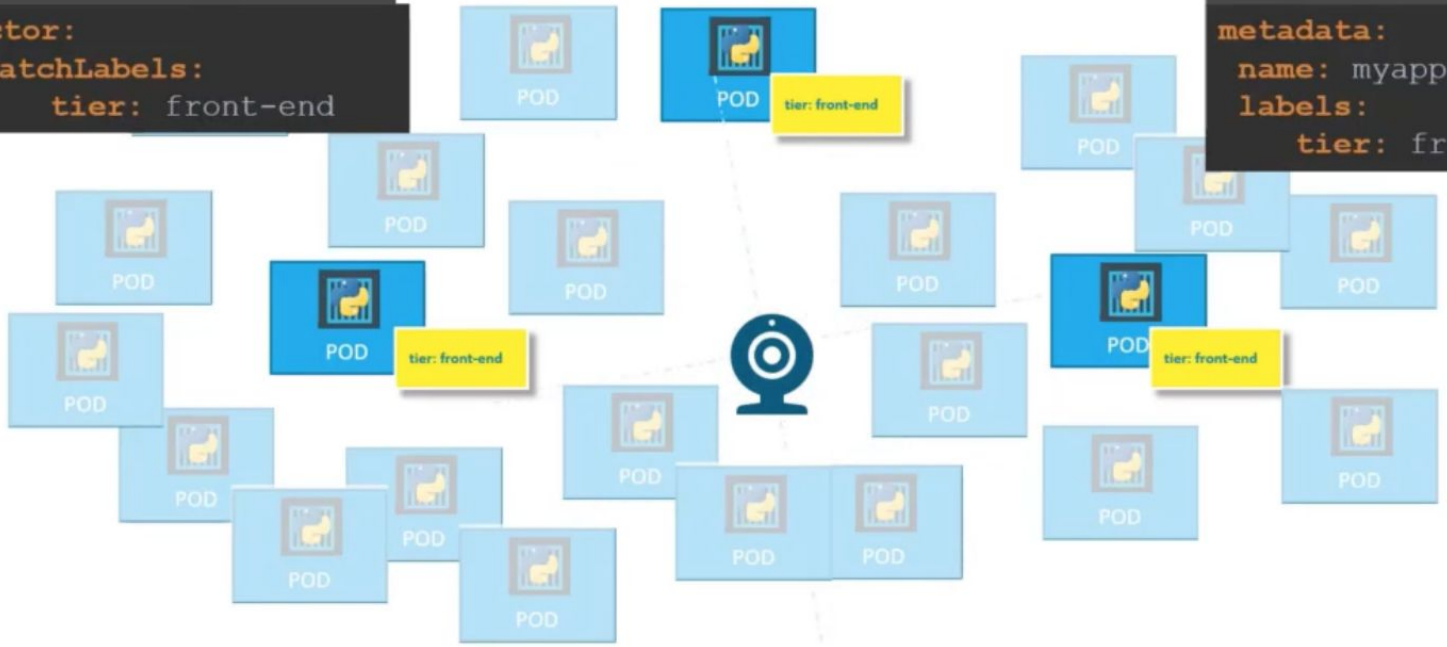
Labels and Selectors

replicaset-definition.yml

```
selector:
    matchLabels:
        tier: front-end
```

pod-definition.yml

```
metadata:
  name: myapp-pod
  labels:
    tier: front-end
```



Scale

```
> kubectl replace -f replicaset-definition.yml
```

```
> kubectl scale --replicas=6 -f replicaset-definition.yml
```

```
> kubectl scale --replicas=6 replicaset myapp-replicaset
```

TYPE NAME

replicaset-definition.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx
  replicas: 6
  selector:
    matchLabels:
      type: front-end
```

commands

```
> kubectl create –f replicaset-definition.yml
```

```
> kubectl get replicaset
```

```
> kubectl delete replicaset myapp-replicaset
```

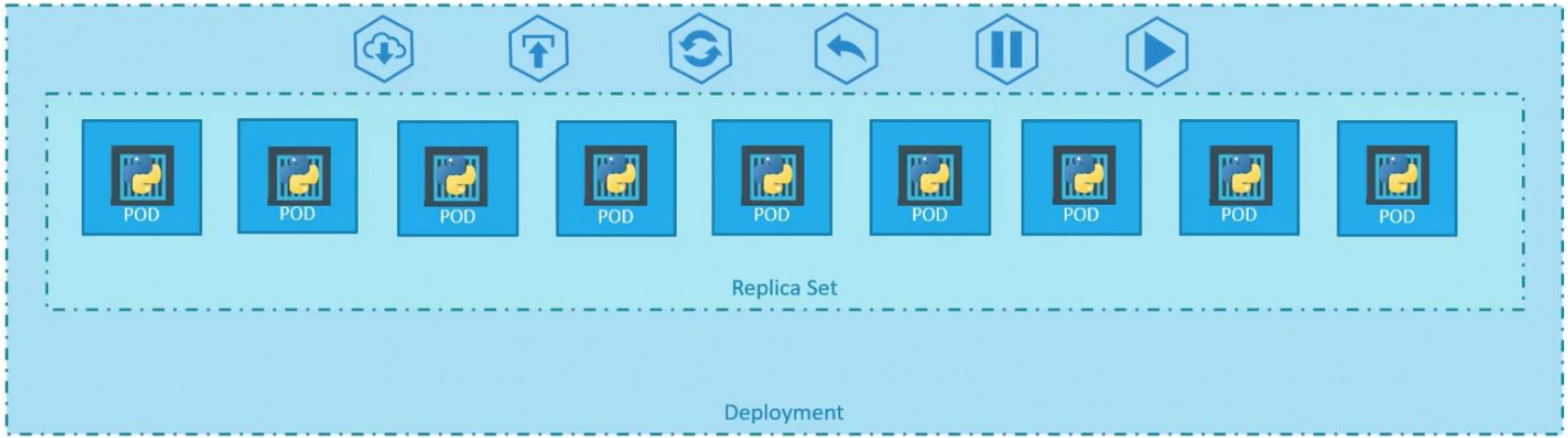
*Also deletes all underlying PODs

```
> kubectl replace -f replicaset-definition.yml
```

```
> kubectl scale –replicas=6 -f replicaset-defi
```

Deployments

Deployment


v1
v2
POD
POD
POD
POD
POD
POD
POD
POD
POD
Replica Set
Deployment

Definition

deployment-definition.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
      containers:
      - name: nginx-container
        image: nginx

  replicas: 3
  selector:
    matchLabels:
      type: front-end
```

Pause

commands

```
> kubectl get all
NAME                        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/myapp-deployment     3         3         3            3           9h

NAME                                DESIRED   CURRENT   READY     AGE
rs/myapp-deployment-6795844b58      3         3         3         9h

NAME                                    READY     STATUS    RESTARTS   AGE
po/myapp-deployment-6795844b58-5rbjl    1/1       Running   0          9h
po/myapp-deployment-6795844b58-h4w55    1/1       Running   0          9h
po/myapp-deployment-6795844b58-lfjhv    1/1       Running   0          9h
```

Certification Tip!

Here's a tip!

As you might have seen already, it is a bit difficult to create and edit YAML files. Especially in the CLI. During the exam, you might find it difficult to copy and paste YAML files from browser to terminal. Using the `kubectl run` command can help in generating a YAML template. And sometimes, you can even get away with just the `kubectl run` command without having to create a YAML file at all. For example, if you were asked to create a pod or deployment with specific name and image you can simply run the `kubectl run` command.

Use the below set of commands and try the previous practice tests again, but this time try to use the below commands instead of YAML files. Try to use these as much as you can going forward in all exercises

Reference (Bookmark this page for exam. It will be very handy):

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

Create an NGINX Pod

```
kubectl run nginx --image=nginx
```

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

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

Create a deployment

```
kubectl create deployment --image=nginx nginx
```

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

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

Generate Deployment YAML file (-o yaml). Don’t create it(–dry-run) with 4 Replicas (–replicas=4)

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

Save it to a file, make necessary changes to the file (for example, adding more replicas) and then create the deployment.

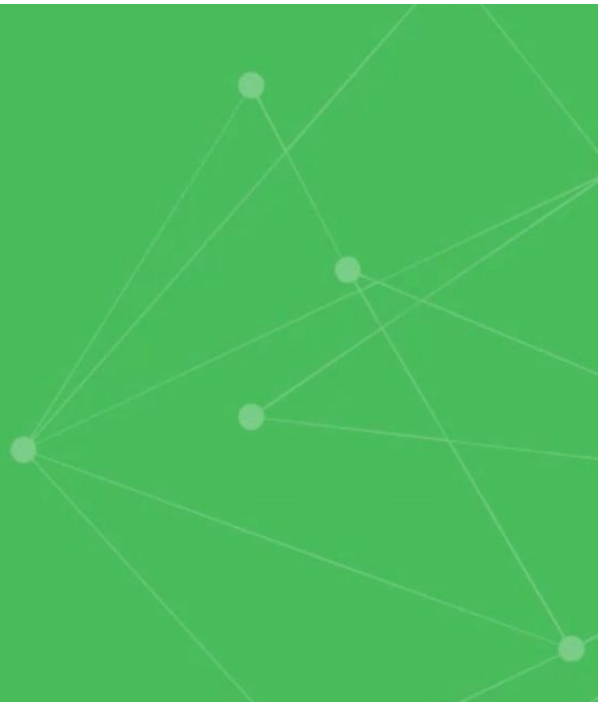
```
kubectl create -f nginx-deployment.yaml
```

OR

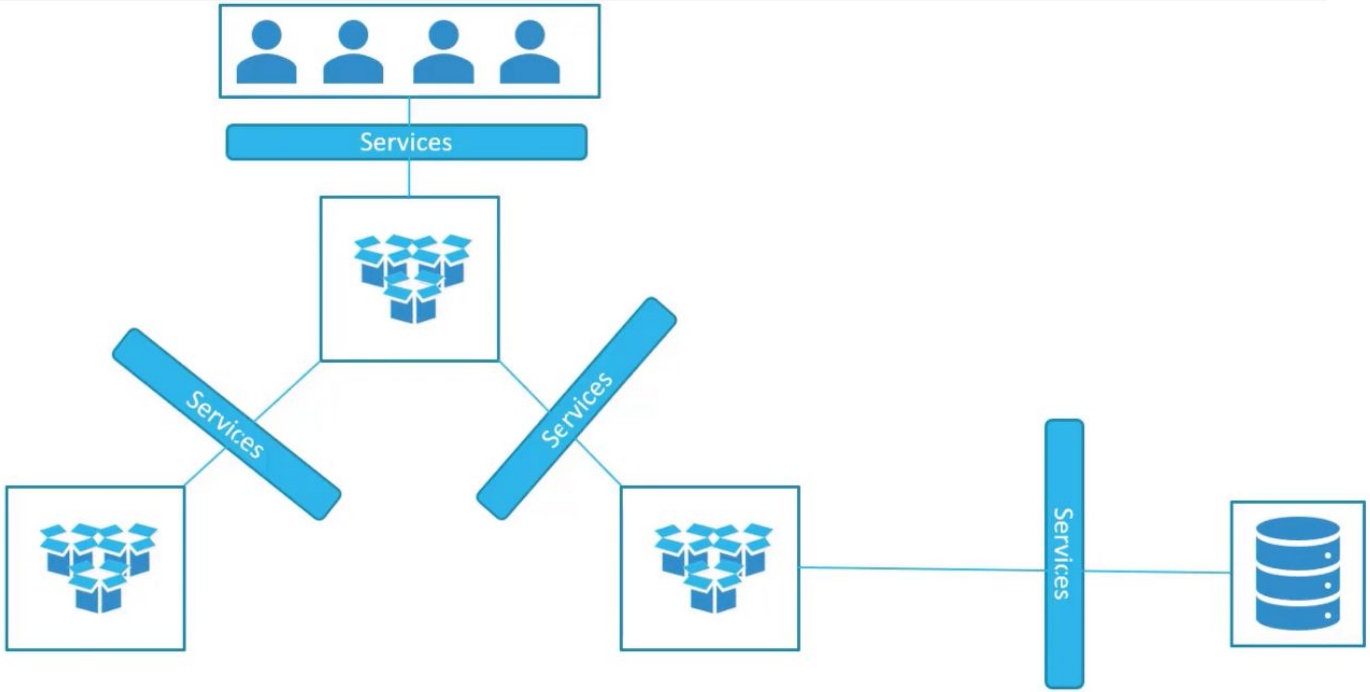
In k8s version 1.19+, we can specify the –replicas option to create a deployment with 4 replicas.

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

SERVICES

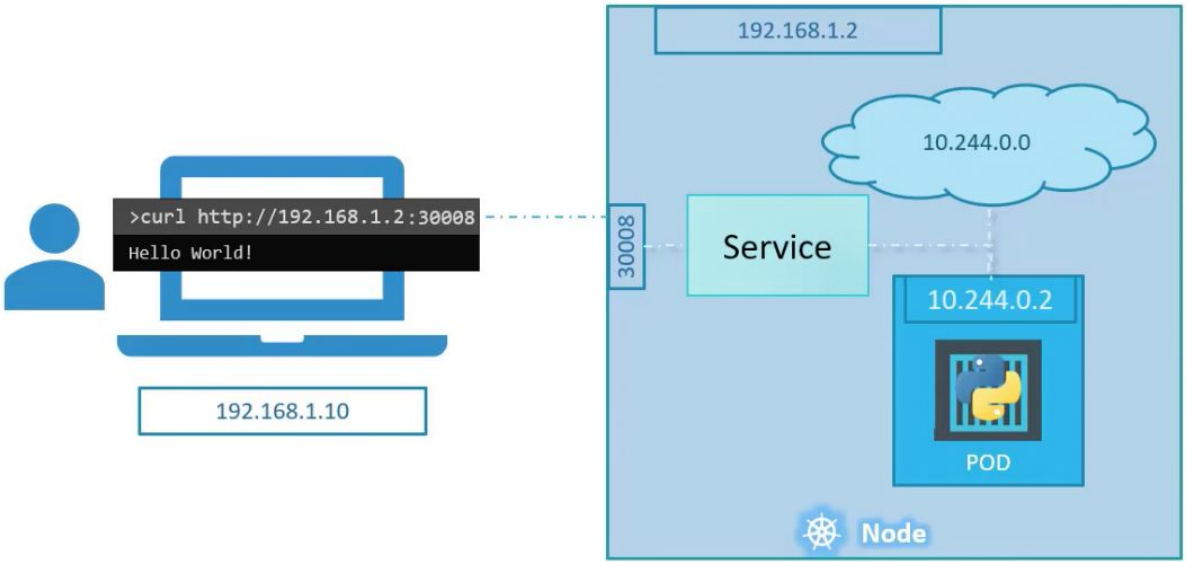


Services



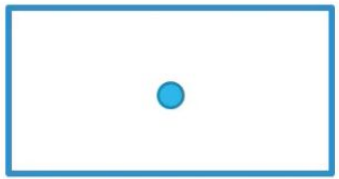
Services Makes available an Internal network that, users can interact with the Pods and a Pod (Application) can connect To another Pod (Application).

Service

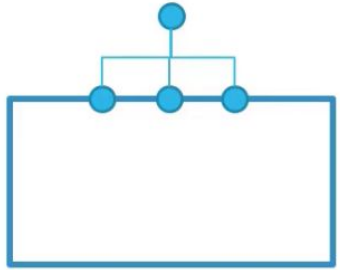

>curl http://192.168.1.2:30008
Hello World!
192.168.1.10
192.168.1.2
10.244.0.0
30008
Service
10.244.0.2
POD
Node

Services Types

NodePort



ClusterIP

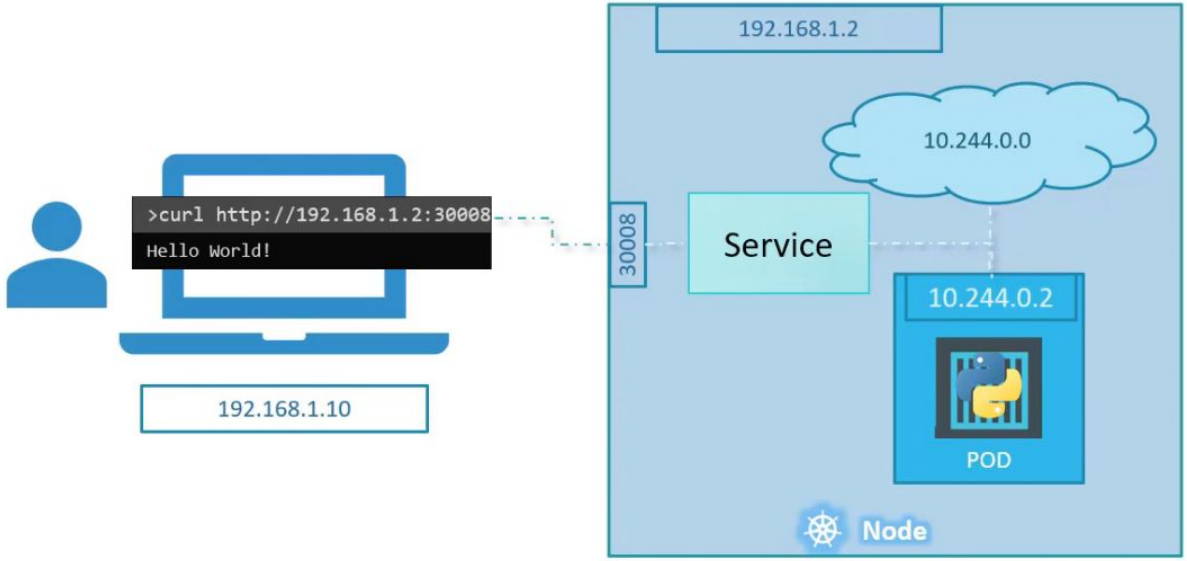


LoadBalancer

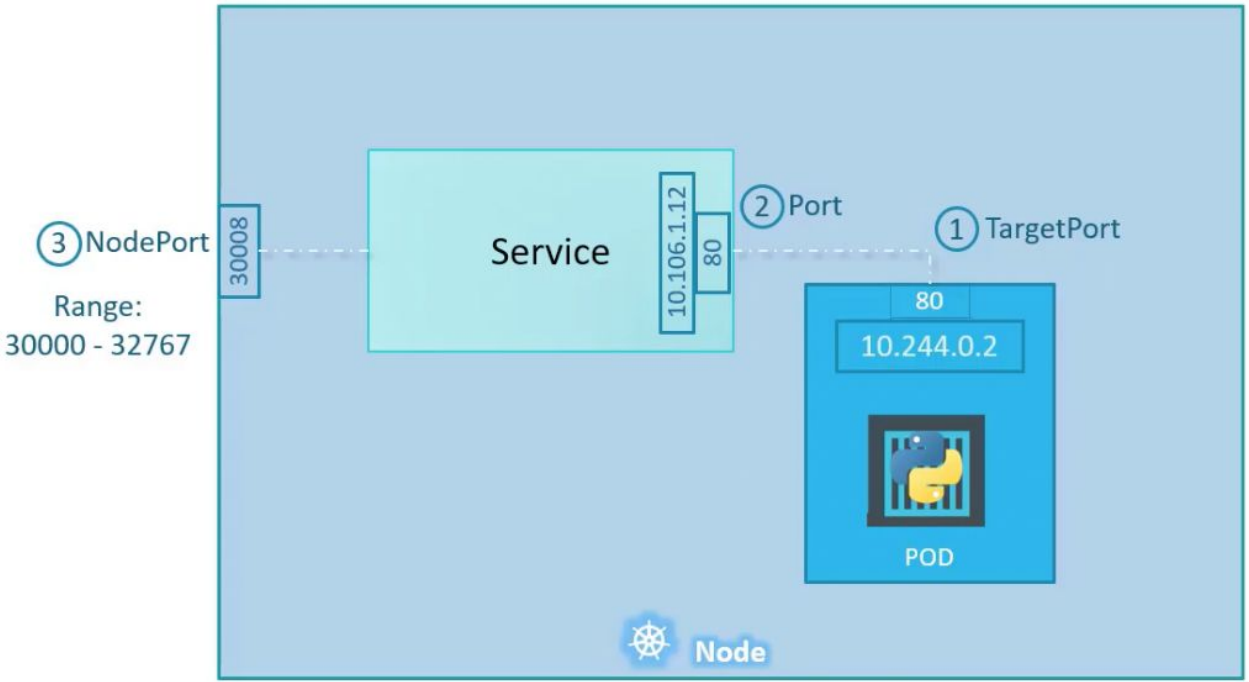
NodePort: This Maps a Port on the Pod to a Port on the containers, so external users can access the Application via the node port IP

NodePort

Service - NodePort



Service - NodePort



service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
```

Creating a Service

Kubectl create -f service-defination.yaml

Get the list of services

Kubectl get services

Kubectl get svc

Adding Selectors And Labels

Service - NodePort

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

Add labels from Pod

Service - NodePort

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
       port: 80
       nodePort: 30008
    selector:
       app: myapp
       type: front-end
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:


spec:
  containers:
  - name: nginx-container
    image: nginx
```

```
> kubectl create -f service-definition.yml
service "myapp-service" created
```

```
> kubectl get services
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes      ClusterIP   10.96.0.1       <none>        443/TCP        16d
myapp-service   NodePort    10.106.127.123  <none>        80:30008/TCP   5m
```

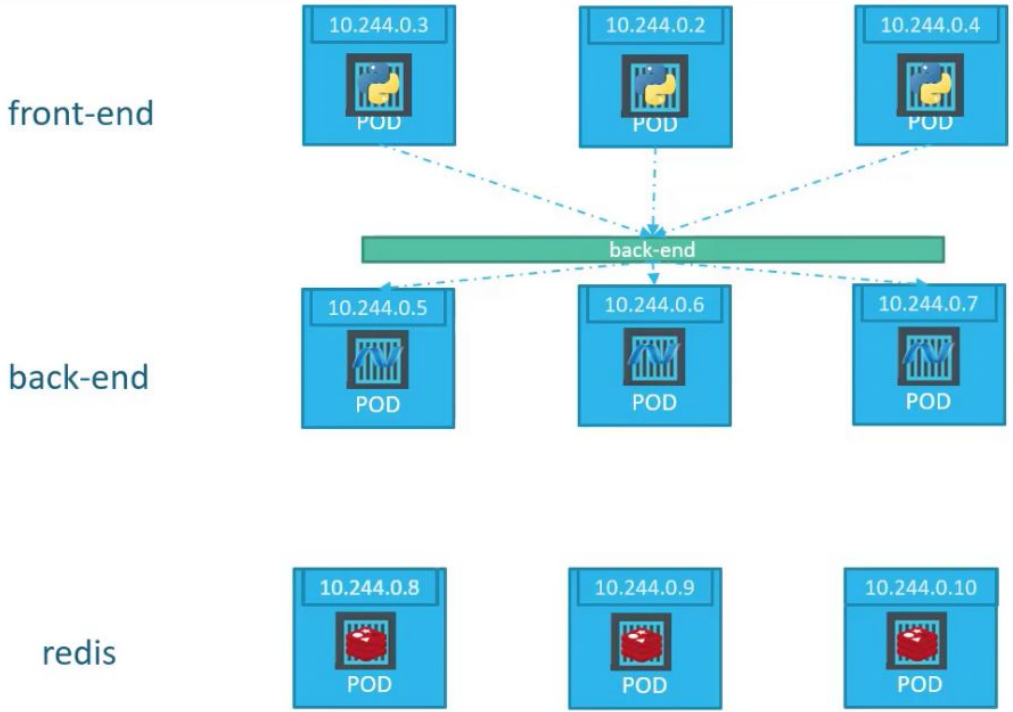
```
> curl http://192.168.1.2:30008
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
```

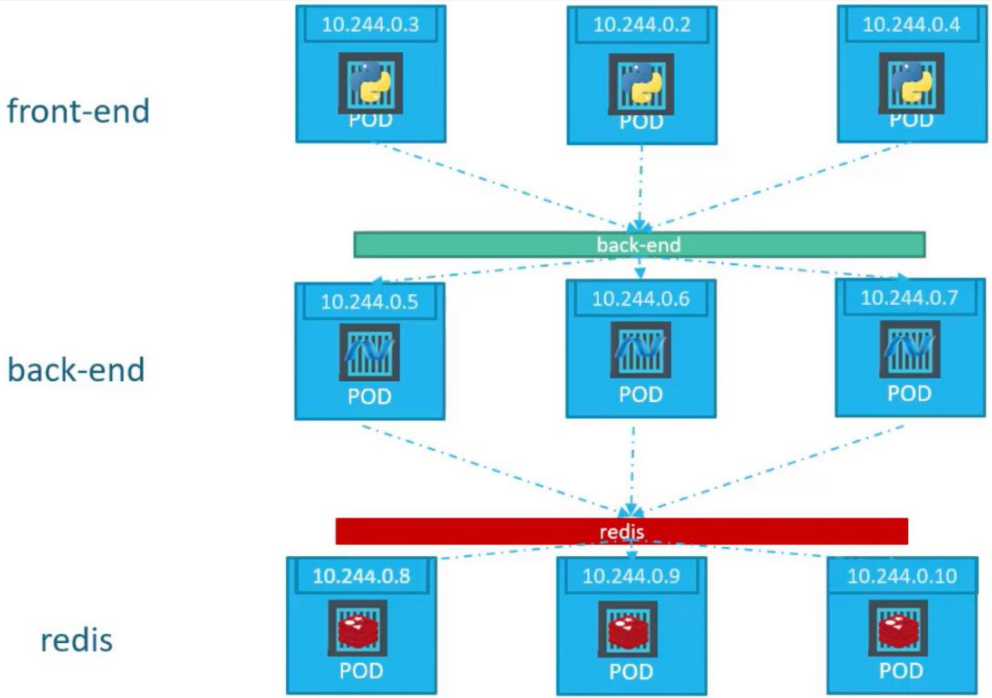

CLUSTER IP

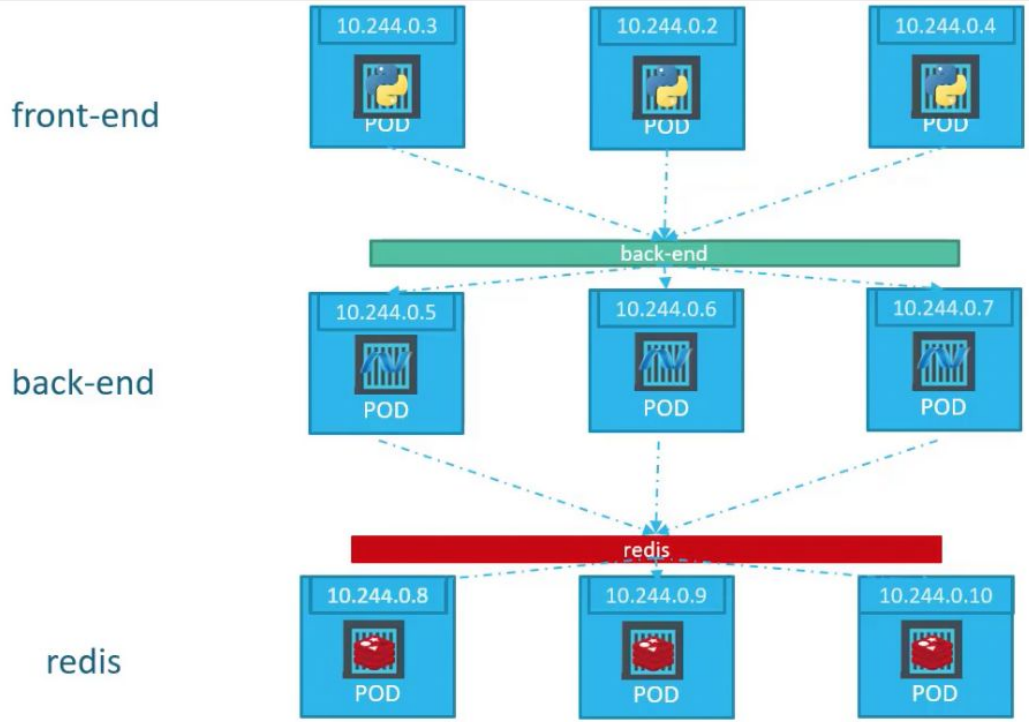
In a multi-tier acheticure where we having multiple applications on Multiple Pods, trying to connect to Each other over the network, a service Cluster IP is an Ideal Approach to handle, networking

A kubernates Cluster IP Service, can help us to group the

Pods together, so we can access them in a Group


front-end
10.244.0.3
POD
10.244.0.2
POD
10.244.0.4
POD
back-end
back-end
10.244.0.5
POD
10.244.0.6
POD
10.244.0.7
POD
redis
10.244.0.8
POD
10.244.0.9
POD
10.244.0.10
POD


front-end
back-end
redis
10.244.0.3
POD
10.244.0.2
POD
10.244.0.4
POD
back-end
10.244.0.5
POD
10.244.0.6
POD
10.244.0.7
POD
redis
10.244.0.8
POD
10.244.0.9
POD
10.244.0.10
POD


front-end
back-end
redis
10.244.0.3
10.244.0.2
10.244.0.4
POD
POD
POD
back-end
10.244.0.5
10.244.0.6
10.244.0.7
POD
POD
POD
redis
10.244.0.8
10.244.0.9
10.244.0.10
POD
POD
POD

Creating A Cluster IP Service

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: back-end

spec:
    type: ClusterIP
    ports:
     - targetPort: 80
       port: 80

    selector:
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: back-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

service-definition.yml

```yaml
apiVersion: v1
kind: Service
metadata:
    name: back-end

spec:
    type: ClusterIP
    ports:
     - targetPort: 80
       port: 80

    selector:
       app: myapp
       type: back-end
```

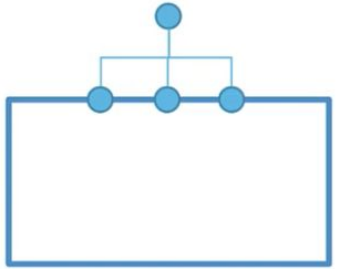
pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:


spec:
  containers:
  - name: nginx-container
    image: nginx
```

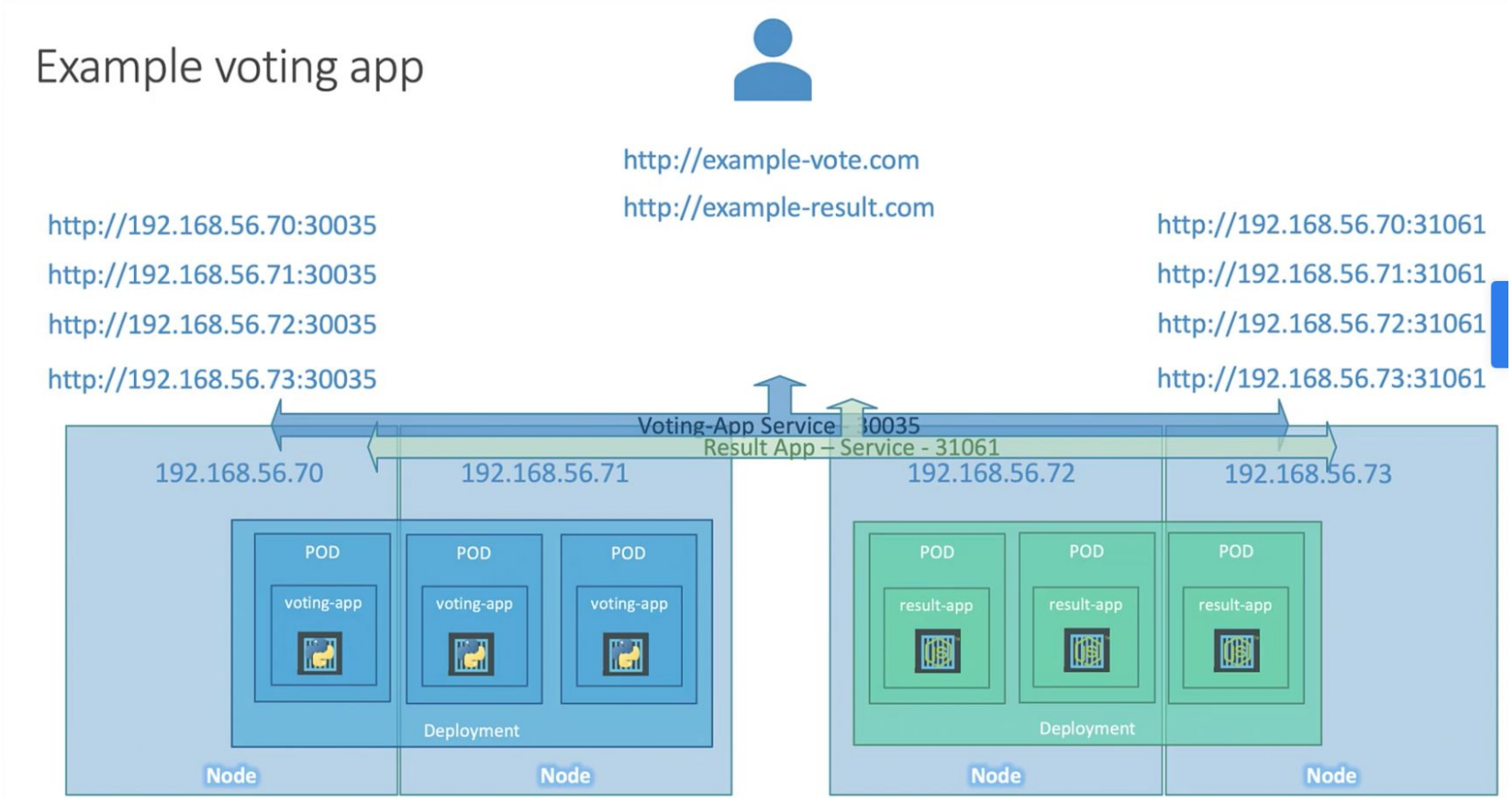
```
> kubectl create -f service-definition.yml
service “back-end" created
```

```
> kubectl get services
NAME          TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
kubernetes    ClusterIP   10.96.0.1        <none>        443/TCP    16d
back-end      ClusterIP   10.106.127.123   <none>        80/TCP     2m
```

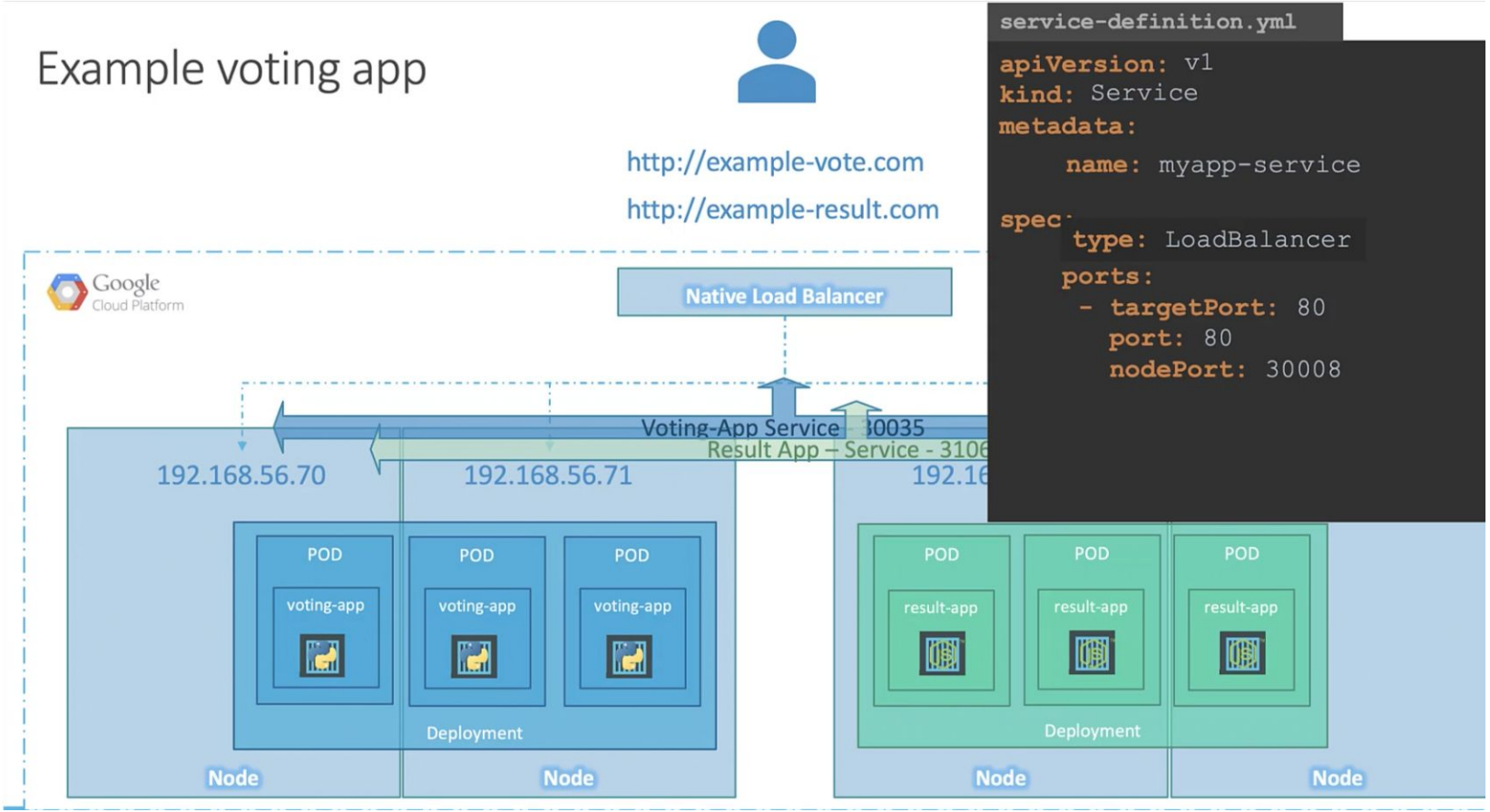
Service-
Load Balancer



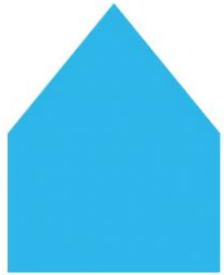
Services – Loadbalancer



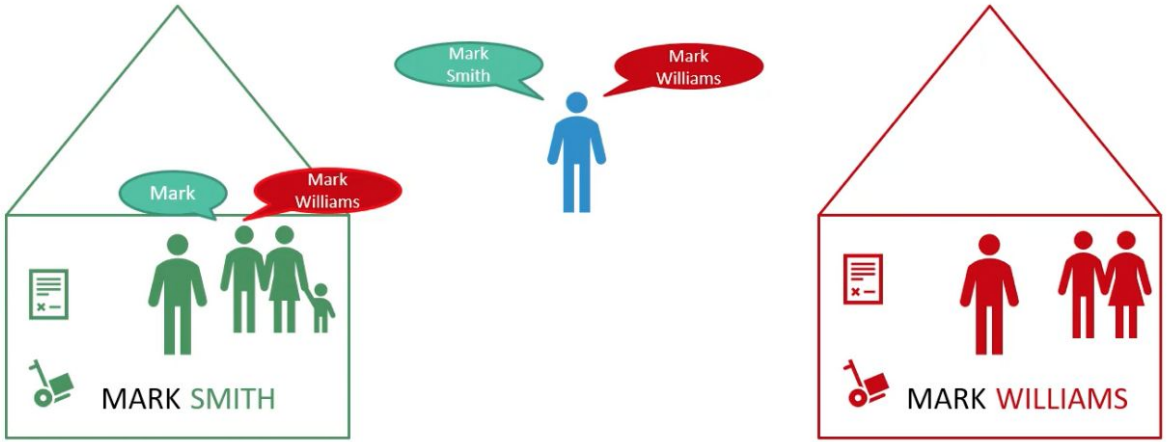
Services – Loadbalancer



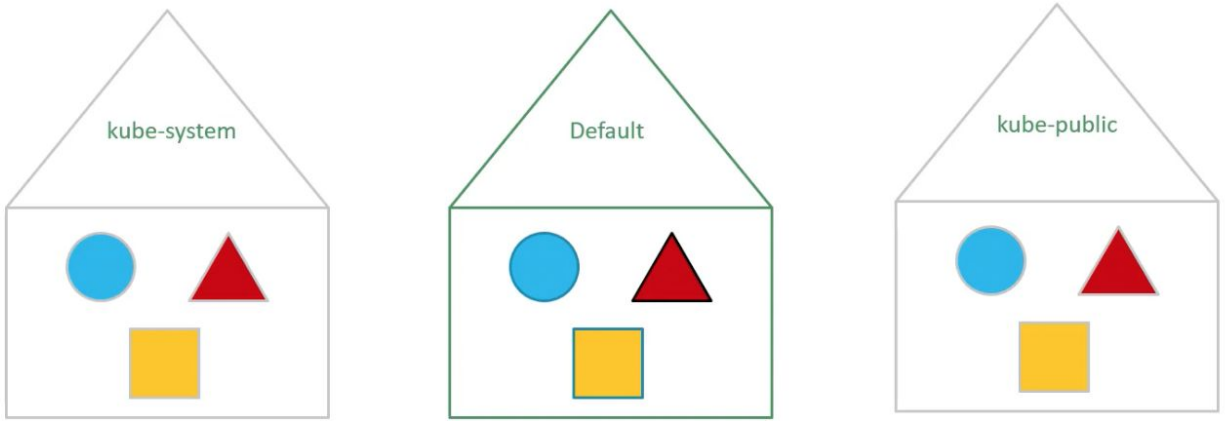
Namespaces



Namespaces

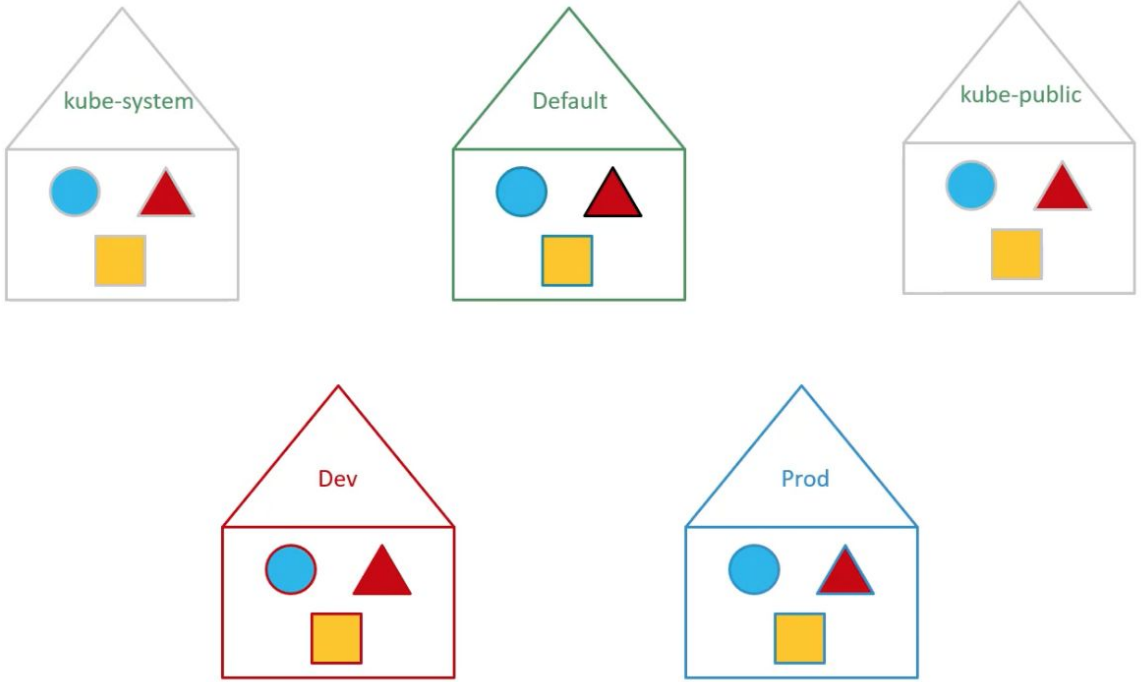

Mark
Mark Williams
MARK SMITH
Mark Smith
Mark Williams
MARK WILLIAMS

Namespaces



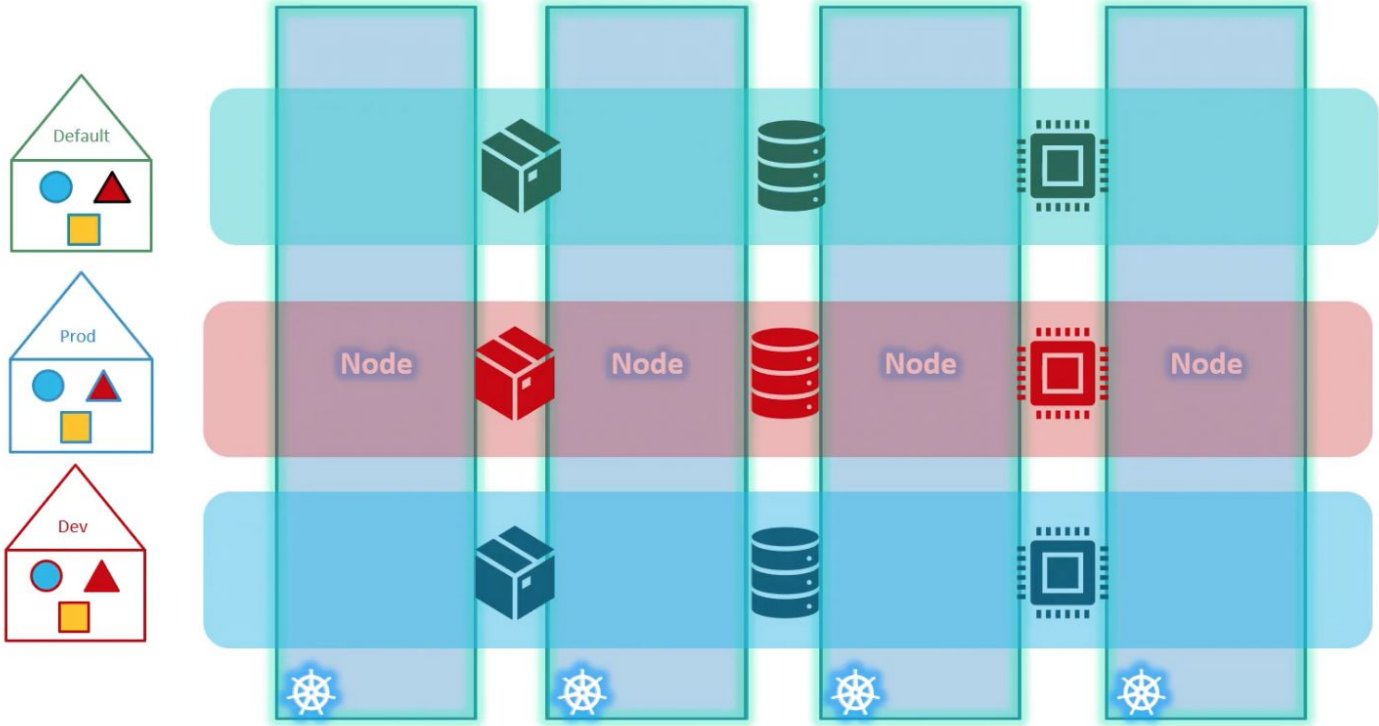
Namespaces

Namespace - Isolation

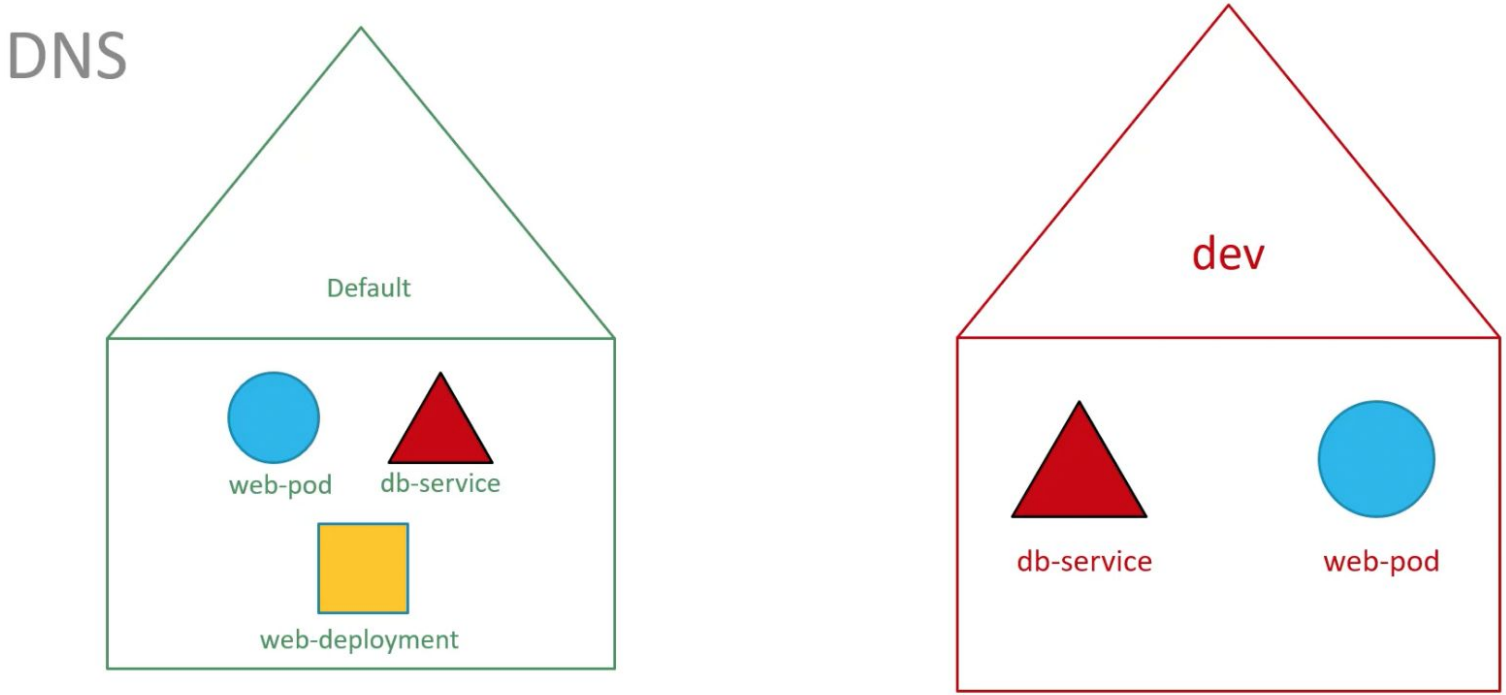


Namespaces

Namespace – Resource Limits



Namespaces



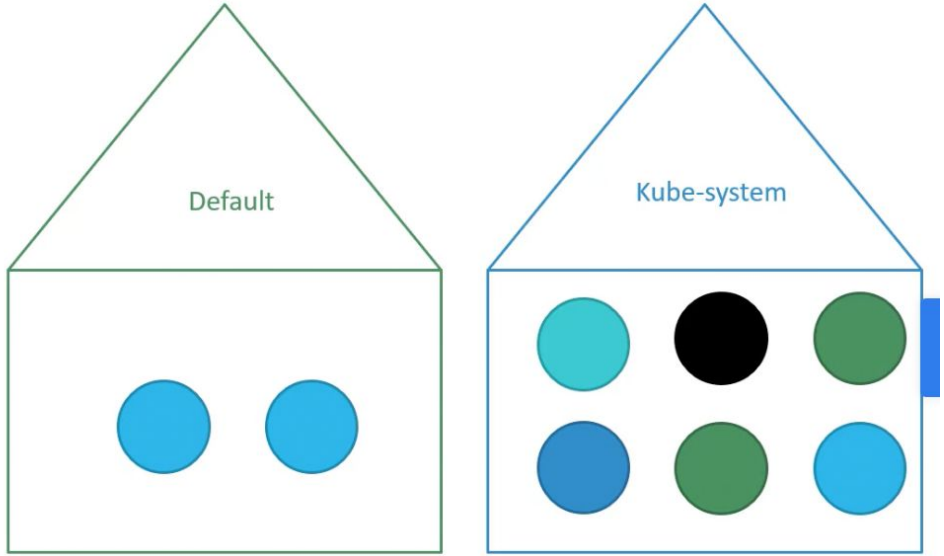

```
mysql.connect("db-service")
```

```
mysql.connect("db-service.dev.svc.cluster.local")
```

Namespaces

```
> kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
Pod-1     1/1       Running   0          3d
Pod-2     1/1       Running   0          3d
```

```
> kubectl get pods --namespace=kube-system
NAME                              READY     STATUS    RESTAR
coredns-78fcdf6894-92d52          1/1       Running   7
coredns-78fcdf6894-jx25g          1/1       Running   7
etcd-master                       1/1       Running   7
kube-apiserver-master             1/1       Running   7
kube-controller-manager-master    1/1       Running   7
kube-flannel-ds-amd64-hz4cf       1/1       Running   14
kube-proxy-4b8tn                  1/1       Running   7
kube-proxy-98db4                  1/1       Running   7
kube-proxy-jjrbs                  1/1       Running   7
kube-scheduler-master             1/1       Running   7
```

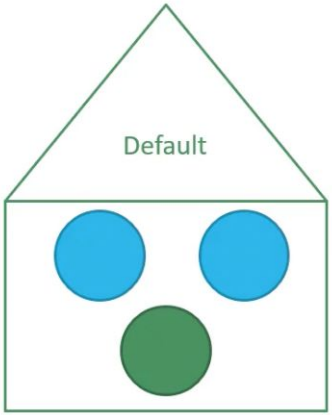


Namespaces

```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```



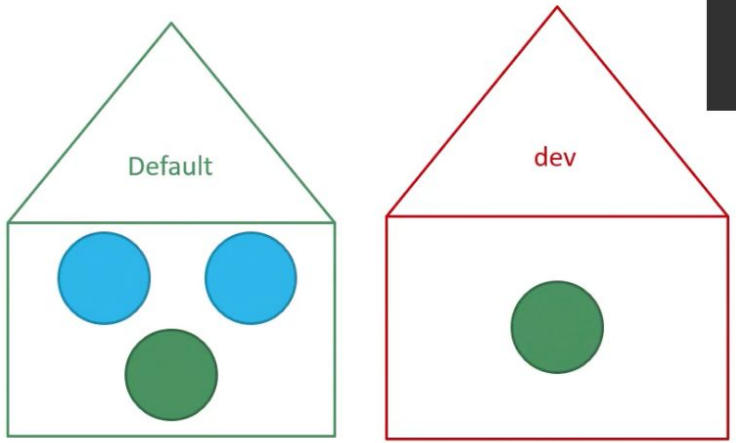
Namespaces

```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

```
> kubectl create -f pod-definition.yml --namespace=dev
pod/myapp-pod created
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

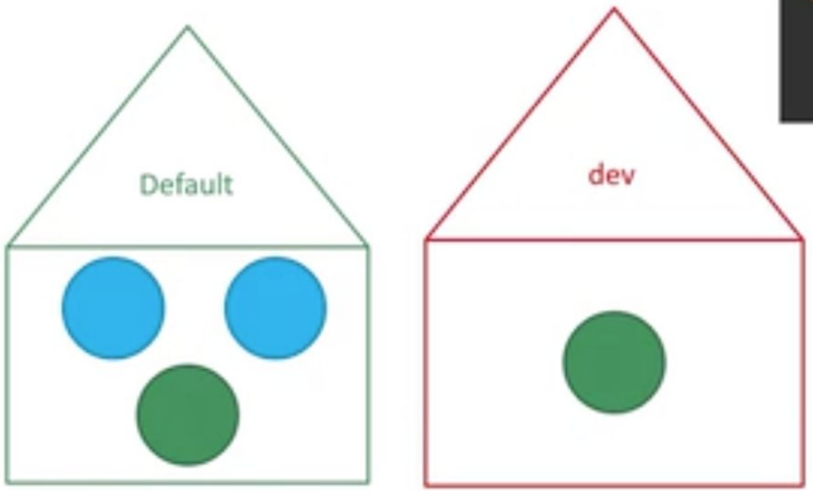


```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

```
> kubectl create -f pod-definition.yml
pod/myapp-pod created
```

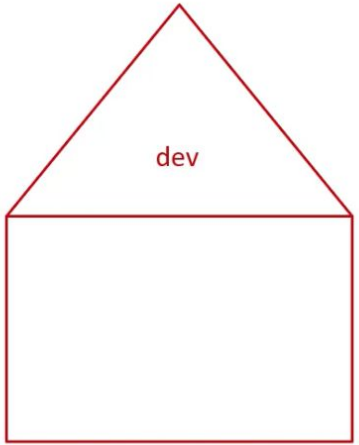
pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 namespace: dev

 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```



Namespaces

Create Namespace



namespace-dev.yml

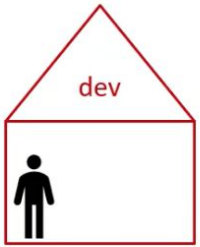
```yaml
apiVersion: v1
kind: Namespace
metadata:
    name: dev
```

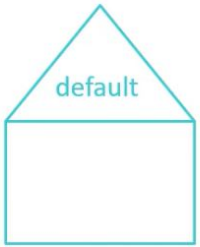
```
> kubectl create -f namespace-dev.yml
namespace/dev created
```

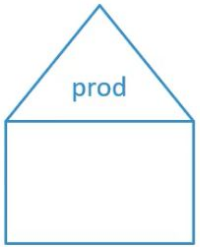
```
> kubectl create namespace dev
namespace/dev created
```

Namespaces

Switch








```
> kubectl get pods --namespace=dev
```

```
> kubectl get pods
```

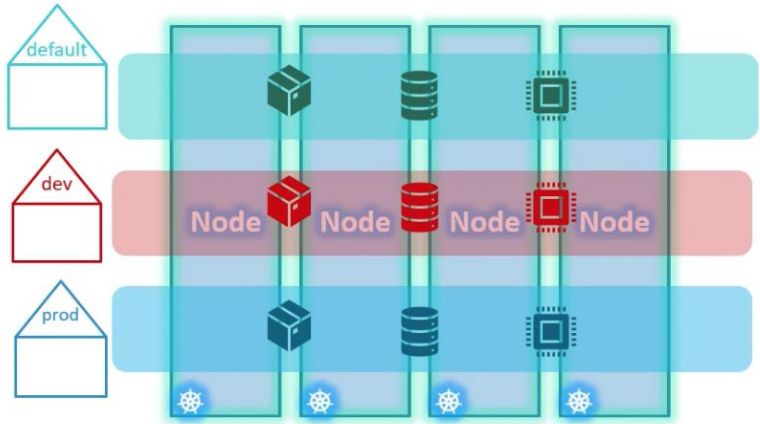
```
> kubectl get pods --namespace=prod
```

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

```
> kubectl get pod
```

Namespaces

Resource Quota



Compute-quota.yaml

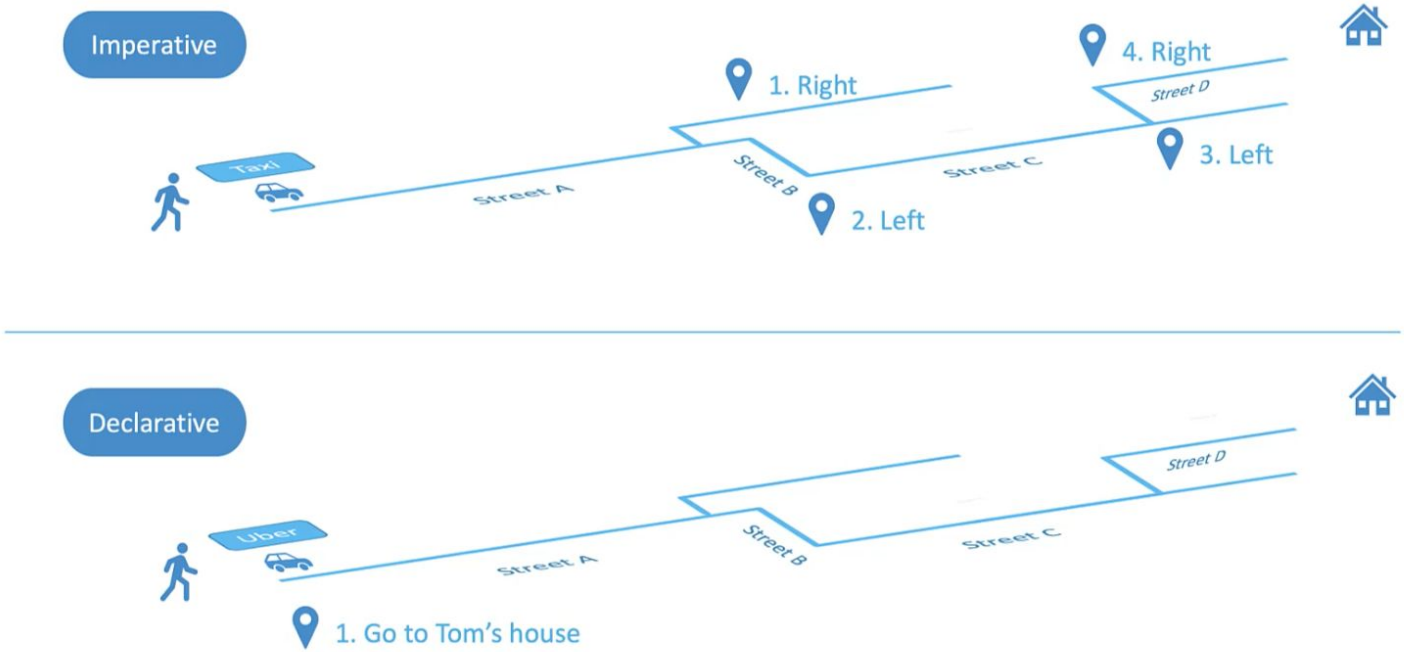
```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
    name: compute-quota
    namespace: dev
spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi
```

```
> kubectl create -f compute-quota.yaml
```

Imperative and Declarative

Imperative vs Declarative

Imperative vs Declarative



Imperative vs Declarative

Infrastructure as Code

Imperative

```
1. Provision a VM by the name ‘web-server’
2. Install NGINX Software on it
3. Edit configuration file to use port ‘8080’
4. Edit configuration file to web path ‘/var/www/nginx’
5. Load web pages to ‘/var/www/nginx’ from GIT Repo - X
6. Start NGINX server
```

Declarative

```
VM Name: web-server
Package: nginx
Port: 8080
Path: /var/www/nginx
Code: GIT Repo - X
```

Imperative vs Declarative

Kubernetes

Imperative

```
> kubectl run --image=nginx nginx
> kubectl create deployment --image=nginx nginx
> kubectl expose deployment nginx --port 80
> kubectl edit deployment nginx
> kubectl scale deployment nginx --replicas=5
> kubectl set image deployment nginx nginx=nginx:1.18
> kubectl create –f nginx.yaml
> kubectl replace –f nginx.yaml
> kubectl delete –f nginx.yaml
```

Declarative

Declarative

```
> kubectl apply -f nginx.yaml
```

Imperative vs Declarative

Imperative Object Configuration Files

Create Objects

```
> kubectl create -f nginx.yaml
```

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

Update Objects

```
> kubectl edit deployment nginx
```

```
> kubectl replace -f nginx.yaml
```

```
> kubectl replace --force -f nginx.yaml
```

```
> kubectl create -f nginx.yaml
Error from server (AlreadyExists): error when creating "nginx.yaml": pods ”myapp-pod" already exists
```

```
> kubectl replace -f nginx.yaml
Error from server (Conflict): error when replacing "nginx.yaml": Operation cannot be fulfilled on pods
”myapp-pod"
```

Imperative vs Declarative

Imperative Object Configuration Files

Create Objects

```
> kubectl create -f nginx.yaml
```

Update Objects

```
> kubectl edit deployment nginx
```

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx
```

pod-definition

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end
spec:
  containers:
  - name: nginx-container
    image: nginx

status:
  conditions:
  - lastProbeTime: null
    status: "True"
    type: Initialized
```

Imperative vs Declarative

Imperative Object Configuration Files

Create Objects

```
> kubectl create -f nginx.yaml
```

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

Update Objects

```
> kubectl edit deployment nginx
```

```
> kubectl replace -f nginx.yaml
```

```
> kubectl replace --force -f nginx.yaml
```

```
> kubectl create -f nginx.yaml
Error from server (AlreadyExists): error when creating "nginx.yaml": pods ”myapp-pod" already exists
```

```
> kubectl replace -f nginx.yaml
Error from server (Conflict): error when replacing "nginx.yaml": Operation cannot be fulfilled on pods
”myapp-pod"
```

Imperative vs Declarative

Declarative

Create Objects

```
> kubectl apply -f nginx.yaml
```

```
> kubectl apply -f /path/to/config-files
```

Update Objects

```
> kubectl apply -f nginx.yaml
```

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

Certification Tips – Imperative Commands with Kubectl

While you would be working mostly the declarative way – using definition files, imperative commands can help in getting one time tasks done quickly, as well as generate a definition template easily. This would help save considerable amount of time during your exams.

Before we begin, familiarize with the two options that can come in handy while working with the below commands:

`--dry-run`: By default as soon as the command is run, the resource will be created. If you simply want to test your command , use the `--dry-run=client` option. This will not create the resource, instead, tell you whether the resource can be created and if your command is right.

`-o yaml`: This will output the resource definition in YAML format on screen.

Use the above two in combination to generate a resource definition file quickly, that you can then modify and create resources as required, instead of creating the files from scratch.

POD

Create an NGINX Pod

```
kubectl run nginx --image=nginx
```

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

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

Deployment

Create a deployment

```
kubectl create deployment --image=nginx nginx
```

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

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

Generate Deployment with 4 Replicas

```
kubectl create deployment nginx --image=nginx --replicas=4
```

You can also scale a deployment using the `kubectl scale` command.

```
kubectl scale deployment nginx --replicas=4
```

Another way to do this is to save the YAML definition to a file and modify

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

You can then update the YAML file with the replicas or any other field before creating the deployment.

Service

Create a Service named redis-service of type ClusterIP to expose pod redis on port 6379

```
kubectl expose pod redis --port=6379 --name redis-service --dry-run=client -o yaml
```

(This will automatically use the pod's labels as selectors)

Or

`kubectl create service clusterip redis --tcp=6379:6379 --dry-run=client -o yaml` (This will not use the pods labels as selectors, instead it will assume selectors as app=redis. You cannot pass in selectors as an option. So it does not work very well if your pod has a different label set. So generate the file and modify the selectors before creating the service)

Create a Service named nginx of type NodePort to expose pod nginx's port 80 on port 30080 on the nodes:

```
kubectl expose pod nginx --type=NodePort --port=80 --name=nginx-service --dry-run=client -o yaml
```

(This will automatically use the pod's labels as selectors, but you cannot specify the node port. You have to generate a definition file and then add the node port in manually before creating the service with the pod.)

Or

```
kubectl create service nodeport nginx --tcp=80:80 --node-port=30080 --dry-run=client -o yaml
```

(This will not use the pods labels as selectors)

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

Reference:

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

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

Kubectl Apply

Kubectl Apply

Local file

Last applied Configuration

Kubernetes

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

```
> kubectl apply -f nginx.yaml
```

Live object configuration

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18

status:
  conditions:
  - lastProbeTime: null
    status: "True"
    type: Initialized
```

Kubectl Apply

Local file

nginx.yaml

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18
```

```
> kubectl apply -f nginx.yaml
```

Last applied Configuration

```json
{
  "apiVersion": "v1",
  "kind": "Pod",
  "metadata": {
    "annotations": {},
    "labels": {
      "run": "myapp-pod",
      "type": "front-end-service"
    },
    "name": "myapp-pod",
  },
  "spec": {
    "containers": [
      {
        "image": "nginx:1.18",
        "name": "nginx-container"
      }
    ]
  }
}
```

Kubernetes

Live object configuration

```yaml
apiVersion: v1
kind: Pod

metadata:
 name: myapp-pod
 labels:
    app: myapp
    type: front-end-service
spec:
  containers:
  - name: nginx-container
    image: nginx:1.18

status:
  conditions:
  - lastProbeTime: null
    status: "True"
    type: Initialized
```