Contents

[KUBERNETES 3](#_Toc129621818)

[CONTAINERS 3](#_Toc129621819)

[WHY WE NEED CONTAINERS 3](#_Toc129621820)

[SOLUTION 4](#_Toc129621821)

[WHAT IS CONTAINER? 4](#_Toc129621822)

[HOW DOCKER WORKS? 4](#_Toc129621823)

[CONTAINER VERSUS VM 5](#_Toc129621824)

[CONTAINER VERSUS IMAGE 5](#_Toc129621825)

[CONTAINER ADVANTAGE 5](#_Toc129621826)

[CONTAINER ORCHESTRATION 6](#_Toc129621827)

[ORCHESTRATION TECHNOLOGIES 6](#_Toc129621828)

[ADVANTAGE OF KUBERNETES 6](#_Toc129621829)

[KUBERNETES ARCHITECTURE 7](#_Toc129621830)

[NODES 7](#_Toc129621831)

[CLUSTER 7](#_Toc129621832)

[MASTER 7](#_Toc129621833)

[COMPONENTS OF KUBERNETES 8](#_Toc129621834)

[MASTER VERSUS WORKER NODES 8](#_Toc129621835)

[SETTING UP KUBERNETES - MINIKUBE 9](#_Toc129621836)

[MINIKUBE 9](#_Toc129621837)

[KUBERNETES PODS 11](#_Toc129621838)

[MULTICONTAINER POD 12](#_Toc129621839)

[PODS IN PRACTICE 13](#_Toc129621840)

[YAML 13](#_Toc129621841)

[PODS WITH YAML 14](#_Toc129621842)

[REPLICATION CONTROLLER AND REPLICASET 15](#_Toc129621843)

[HIGH AVAILIBILITY 15](#_Toc129621844)

[LOAD BALANCING & SCALING 16](#_Toc129621845)

[CREATING REPLICATION CONTROLLER 16](#_Toc129621846)

[CREATING REPLICATION SET 17](#_Toc129621847)

[DEPLOYMENTS 20](#_Toc129621848)

[CREATING DEPLOYMENT 21](#_Toc129621849)

[UPDATE AND ROLLBACK 21](#_Toc129621850)

[NETWORKING 27](#_Toc129621851)

[SINGLE NODE 27](#_Toc129621852)

[MULTIPLE NODES(CLUSTER NETWORKING) 28](#_Toc129621853)

[SERVICES 28](#_Toc129621854)

[SERVICE TYPES 29](#_Toc129621855)

[USE CASE OF SERVICES 29](#_Toc129621856)

[NODEPORT SERVICE 30](#_Toc129621857)

[CLUSTER-IP SERVICE 33](#_Toc129621858)

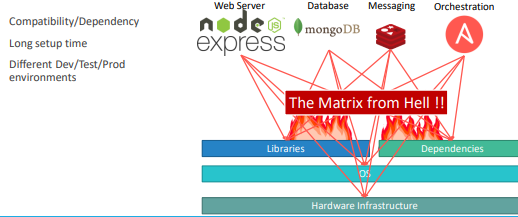
[LOADBALANCER SERVICE 34](#_Toc129621859)

# KUBERNETES

* Developed by Google also known as k8.
* It’s a container orchestration tool

## CONTAINERS

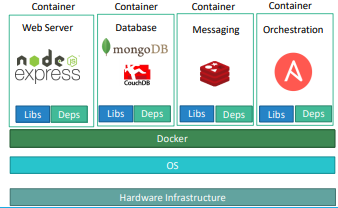
### WHY WE NEED CONTAINERS



To understand the concept of container. Let’s take an example

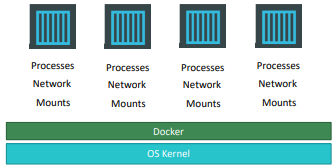
* Let say we have a requirement to setup an end-to-end stack including various technologies like a Web Server using NodeJS and a database such as MongoDB/CouchDB, messaging system like Redis and an orchestration tool like Ansible.
* We will have lot of issues developing this application with all these different components. First, their compatibility with the underlying OS. We must ensure that all these different services were compatible with the version of the OS we were planning to use.
* There is a possibility when certain version of these services was not compatible with the OS, and we have to go back and look for another OS that will be compatible with all of these different services. Secondly, we must check the compatibility between these services and the libraries and dependencies on the OS.
* The possibility can be where one service requires one version of a dependent library whereas another service required another version.
* Going forward if architecture of the application changed over time, when have an upgrade to newer versions of these components or change the database etc. and every time something changed, we have to go through the same process of checking compatibility between these various components and the underlying infrastructure. **This compatibility matrix issue is usually referred to as the matrix from hell**.
* Apart from that – every time to on board a new developer, it will be difficult to setup a new environment. The new developers must follow a large set of instructions and run 100s of commands to finally setup their environments. They must make sure they were using the right Operating System, the right versions of each of these components and each developer had to set all that up by himself each time.
* We also had different development test and production environments. One developer may be comfortable using one OS, and the others may be using another one and so we couldn’t guarantee the application that we were building would run the same way in different environments. And so, all of this made our life in developing, building and shipping the application really difficult.

### SOLUTION



* To solve the compatibility issue we need some tool that will allow us to modify or change these components without affecting the other components and even modify the underlying operating systems as required. The solution to the problem is Docker.
* With Docker we can be able to run each component in a separate container – with its own libraries and its own dependencies. All on the same VM and the OS, but within separate environments or containers.
* We just had to build the docker configuration once, and all our developers could now get started with a simple “docker run” command. Irrespective of what underlying OS they run, all they needed to do was to make sure they had Docker installed on their systems

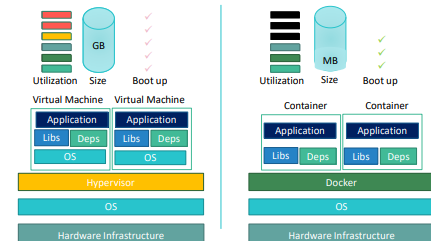
### WHAT IS CONTAINER?



* Containers are completely isolated environments, as in they can have their own processes or services, their own network interfaces, their own mounts, just like Virtual machines, **except that they all share the same OS kernel**.
* But it’s also important to note that containers are not new with Docker. Containers have existed for about 10 years now and some of the different types of containers are LXC, LXD, LXCFS etc. Docker utilizes LXC containers.
* Setting up these container environments is hard as they are very low level and that is where Docker offers a high-level tool with several powerful functionalities making it easy for end users like us.

### HOW DOCKER WORKS?

### CONTAINER VERSUS VM



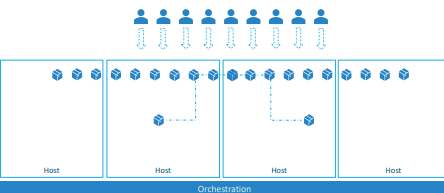
### CONTAINER VERSUS IMAGE

|  |  |
| --- | --- |
|  | * An image is a package or a template, just like a VM template. It is used to create one or more containers. * Containers are running instances off images that are isolated and have their own environments and set of processes. |

### CONTAINER ADVANTAGE

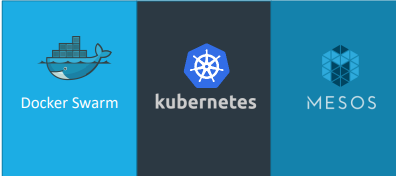
* For a traditionally developed application, developer hand it over to Ops team to deploy and manage it in production environments. They do that by providing a set of instructions such as information about how the hosts must be setup, what pre-requisites are to be installed on the host and how the dependencies are to be configured etc. Since the Ops team did not develop the application on their own, they struggle with setting it up. When they hit an issue, they work with the developers to resolve it.
* With Docker, a major portion of work involved in setting up the infrastructure is now in the hands of the developers in the form of a Docker file. The guide that the developers built previously to setup the infrastructure can now easily put together into a ***Dockerfile*** to create an image for their applications. This image can now run on any container platform and is guaranteed to run the same way everywhere.
* So the Ops team now can simply use the image to deploy the application. Since the image was already working when the developer built it and operations are not modifying it, it continues to work the same when deployed in production.

## CONTAINER ORCHESTRATION



* After the application packaged into a docker container. Now the question is
  + How do we run it in production?
  + What if the application relies on other containers such as database or messaging services or other backend services?
  + What if the number of users increase and we need to scale the application? And to scale down when the load decreases.
* ***To enable these functionalities, we need an underlying platform with a set of resources. The platform needs to orchestrate the connectivity between the containers and automatically scale up or down based on the load. This whole process of automatically deploying and managing containers is known as Container Orchestration.***

### ORCHESTRATION TECHNOLOGIES



* Kubernetes is thus a container orchestration technology. There are multiple such technologies available today

1. DOCKER SWARM.

Docker Swarm is really easy to setup and get started, it lacks some of the advanced autoscaling features required for complex applications.

1. KUBERNETES from Google
   * + Kubernetes - arguably the most popular of it all – is a bit difficult to setup and get started but provides a lot of options to customize deployments and supports deployment of complex architectures.
     + Kubernetes is now supported on all public cloud service providers like GCP, Azure and AWS and the kubernetes project is one of the top ranked projects in Github.
2. MESOS from Apache. While Mesos on the other hand is quite difficult to setup and get started but supports many advanced features.

## ADVANTAGE OF KUBERNETES

There are various advantages of container orchestration.

* Application will be highly available as hardware failures do not bring the application down because we have multiple instances of the application running on different nodes.
* The user traffic is load balanced across the various containers. When demand increases, deploy more instances of the application seamlessly and within a matter of second and we can do that at a service level.
* When we run out of hardware resources, scale the number of nodes up/down without having to take down the application. And do all these easily with a set of declarative object configuration files.
* **Kubernetes - It is a container Orchestration technology used to orchestrate the deployment and management of 100s and 1000s of containers in a clustered environment.**

## KUBERNETES ARCHITECTURE

### NODES

|  |  |
| --- | --- |
|  | * A node is a machine – physical or virtual – on which Kubernetes is installed. * A node is a worker machine, and this is where containers will be launched by Kubernetes) It was also known as Minions in the past). * But what if the node on which our application is running fails? Well, obviously our application goes down. So, you need to have more than one node. Then comes the concept of Cluster |

### CLUSTER

|  |  |
| --- | --- |
|  | * A cluster is a set of nodes grouped together. * This way even if one node fails the application still accessible from the other nodes. * Having multiple nodes helps in sharing load as well. |

### MASTER



Master node is Kubernetes are

* Is responsible for managing the cluster
* Master has the information about the members of the cluster stored
* Monitoring the Nodes – For example - when a node fails it moves the workload of the failed node to another worker node

**The master is another node with Kubernetes installed in it and is configured as a Master. The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.**

### COMPONENTS OF KUBERNETES



When we install Kubernetes on a System, following components get installed

1. AN API SERVER.

The API server acts as the front-end for Kubernetes. **The users, management devices, Command line interfaces all talk to the API server to interact with the Kubernetes cluster.**

1. AN ETCD SERVICE.

* ETCD key store is a distributed reliable key-value store used by Kubernetes to store all data used to manage the cluster. For example - when we have multiple nodes and multiple masters in our cluster, etcd stores all that information on all the nodes in the cluster in a distributed manner.
* ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.

1. A KUBELET SERVICE.

Kubelet is the agent that runs on each node in the cluster. The agent is responsible for making sure that the containers are running on the nodes as expected.

1. A CONTAINER RUNTIME

The container runtime is the underlying software that is used to run containers. For example - Docker

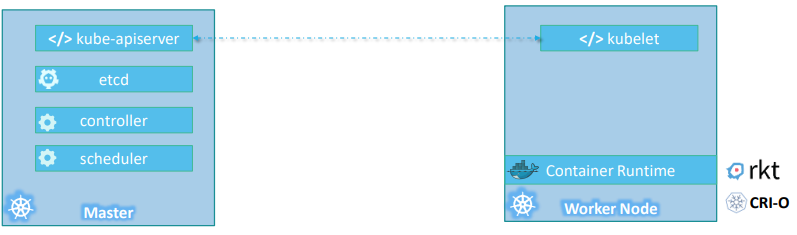
1. CONTROLLERS

The controllers are the brain behind orchestration. They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers make decisions to bring up new containers in such cases.

1. SCHEDULERS.

The scheduler is responsible for distributing work or containers across multiple nodes. It looks for newly created containers and assigns them to Nodes.

### MASTER VERSUS WORKER NODES



**So far we have seen two types of servers – Master and Worker** and a set of components that make up Kubernetes.

Question - **But how are these components distributed across different types of servers. In other words, how does one server become a master and the other slave?**

**WORKER**

* The containers are hosted by the worker node. Hence the worker node should have a container runtime e.g. Docker.
* There are other container runtime alternatives available such as Rocket or CRIO.

**MASTE**

* ***The master server has the kube-apiserver and that is what makes it a master***. Similarly, the worker nodes have the ***kubelet agent*** that is responsible for interacting with the master to provide health information of the worker node and carry out actions requested by the master on the worker nodes.
* All the information gathered are stored in a key-value store on the Master. The key value store on ***etcd framework***.
* The master also has the controller manager and the scheduler.

## SETTING UP KUBERNETES - MINIKUBE



* There are lots of ways to setup Kubernetes. We can setup it up ourselves locally on our laptops or virtual machines using solutions like z
  + **MINIKUBE** – Minikube is an open source tool used to setup a single instance of Kubernetes in an All-in-one setup
  + **KUBEADMIN** – Kubeadmin is a tool used to configure Kubernetes in a multi-node setup.
  + **HOSTED SOLUTIONS** available for setting up Kubernetes in a cloud environment such as **GCP and AWS**.
  + **ONLINE** - <https://labs.play-with-k8s.com/>

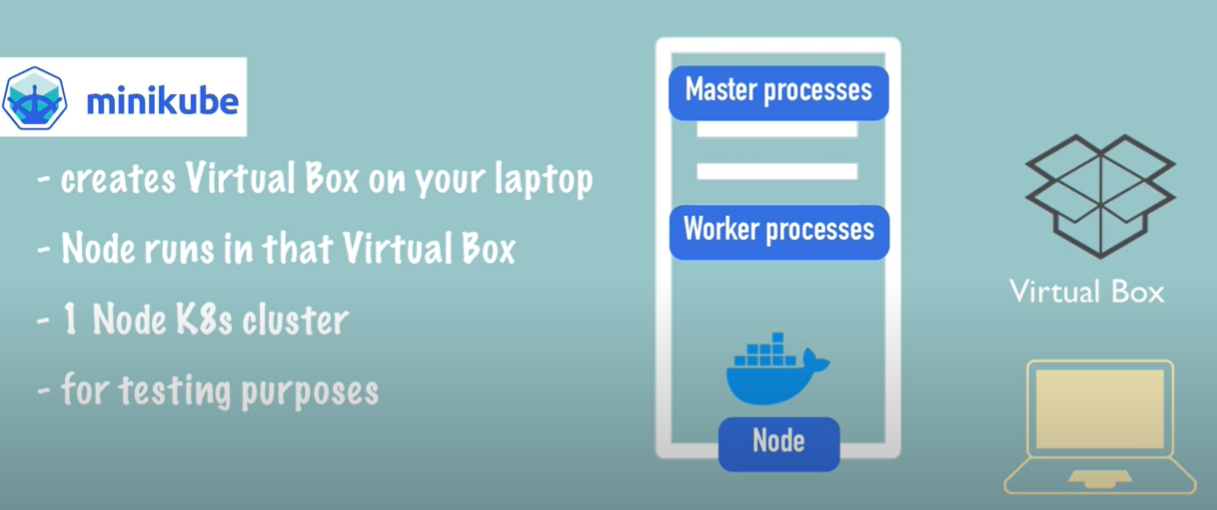
### MINIKUBE

* <https://minikube.sigs.k8s.io/docs/start/>

|  |  |
| --- | --- |
|  | For Kubernetes installation (Production Cluster Set up)   * The master node consists of - *apiserver, etcd - key value store, controllers, and scheduler* * Worker Node has – *kubelets and container runtime* * It would take a lot of time and effort to setup and install all these various components on different systems individually by ourselves. |

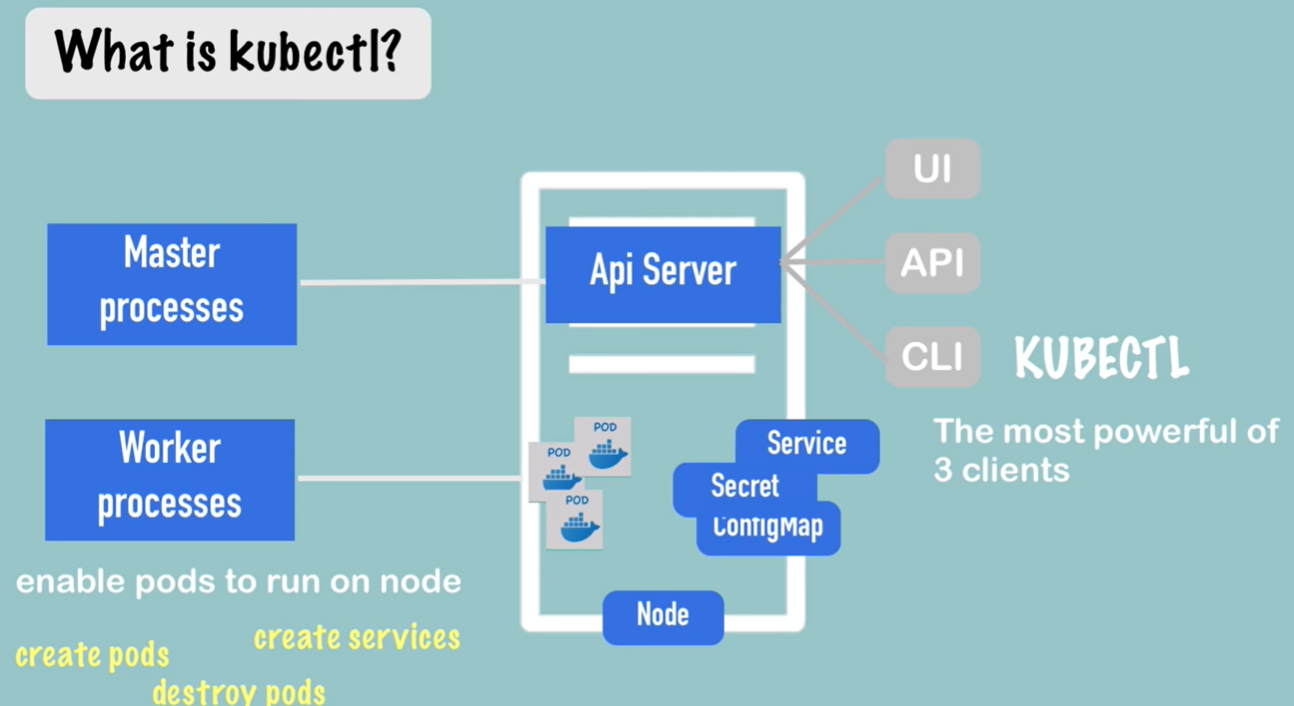
We need 3 things to get Kubernetes working on a local machine,

* + **HYPERVISOR** – Minikube installation needs virtualization. ***For windows we could use Virtualbox or Hyper-V and for Linux use Virtualbox or KVM*** *as a Hypervisor software.*
  + **KUBECTL -** Kubernetes command line tool
  + **MINIKUBE** executable
* Minikube provides an executable command line utility that will AUTOMATICALLY download the ISO and deploy it in a virtualization platform such as **Oracle Virtualbox or Vmwarefusion**.
* ***kubectl Kubernetes command line tool*** will be used to interact with the Kubernetes cluster.
* **Minikube bundles all these different components into a single image providing us a pre-configured single node Kubernetes cluster so we can get started in a matter of minutes. The whole bundle is packaged into an ISO image and is available online for download.**



* Minicube is one node cluster – where the master process and worker process run on one machine
* The node has docker container runtime preinstalled – which provides the runtime environment for the containers
* The way – it runs on local machine is –
  + Minikube creates a Virtualized environment using Virtual Box (Virtual box is a Type 2 hypervisor software)
  + The Single Cluster node runs on the Virtual Box (Virtualized Environment)

#### KUBECTL



* Once we have a virtual node on our local machine – we need a way to interact with Pod and create other Kubernetes components like Services, Secrets and ConfigMap Etc. The way to do it is kubectl(Command line tool for k8 clusters)

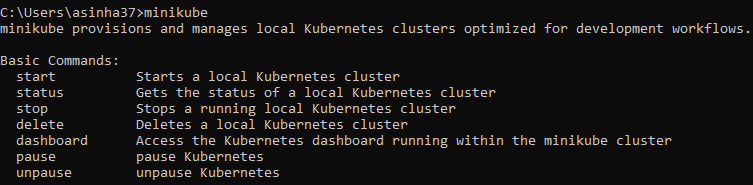
##### HOW KUBECTL INTERACT WITH MINIKUBE CLUSTER?

* Since Minikube run both master and worker process. One of the master processes called “API Server” is the main entry point into the Kubernetes cluster. Hence if we want to any create component or configuration we need to talk to API Server
* The way to interact with API server is using some sort of client. kubectl is one such client
* Once the kubectl submits a request to create / delete pods or service – the Worker process does the actual job.

#### MINIKUBE COMMAND

* To get the list of minikube command just type “minikube” in your command line

**BASIC COMMANDS**



|  |  |  |
| --- | --- | --- |
| COMMAND | | DESCRIPTION |
| **minikube start –vm-driver<*driverName*>**  Ex- minikube start –vm-driver=hyperkit | | * This will start minikube single node cluster for given hypervisor (hyperkit, Virtual box or Docker) * List of supported Minikube Drivers - <https://minikube.sigs.k8s.io/docs/drivers/> |
| START A CLUSTER USING THE DOCKER DRIVER: minikube start --driver=docker  TO MAKE DOCKER THE DEFAULT DRIVER: minikube config set driver docker | | |
| minikube status |  | |
| To Open the minikube dashboard | minikube dashboard | |

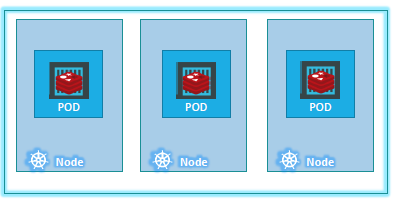
KUBECTL COMMANDS (TO INTERACT WITH NODE & PODS)

* To validate the installation of minikube – we can run couple of kubectl commands from the reference document - <https://kubernetes.io/docs/tutorials/hello-minikube/>

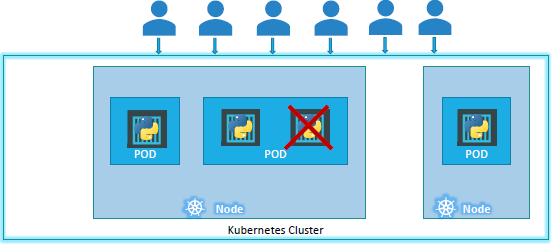
|  |  |
| --- | --- |
| **NODE INFORMATION**  ***kubectl get nodes***  This gives the information of single node minikube cluster | **NAME**- Name of the Node (minikube) which is in ready state.  **VERSION** – Kubernetes version  **AGE** – Since when the Node has been spun up |
| **CREATE DEPLOYMENTS** | kubectl create deployment hello-node --image=registry.k8s.io/echoserver:1.4 |

## KUBERNETES PODS

* To understand PODs let consider application is already developed and built into Docker Images, and it is available on a Docker repository like Docker hub, so that Kubernetes can pull it down.
* We also assume that the Kubernetes cluster has already been setup and is working. This could be a single-node setup or a multi-node setup, doesn’t matter. All the services need to be in a running state.

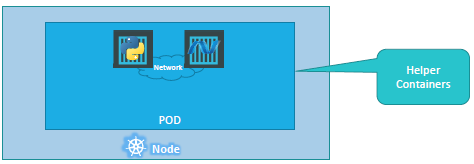


* With Kubernetes our aim is to deploy our application in the form of containers on a set of machines that are configured as worker nodes in a cluster.
* However, Kubernetes does not deploy containers directly on the worker nodes. The containers are encapsulated into a Kubernetes object known as PODs.
* **A POD is a single instance of an application. A POD is the smallest object, that we can create in Kubernetes.**



* Here we see the simplest of simplest cases where we can have a single node Kubernetes cluster with a single instance of the application running in a single docker container encapsulated in a POD.
* **If the number of users accessing the application increases and we need to scale our application. Then we create a new POD altogether with a new instance of the same application**,
* *Note – if load increases - we don’t add additional instances of the web application to share the load or a new container instance within the same POD*
* If the user base FURTHER increases and the current node has no sufficient capacity - THEN we can always deploy additional PODs on a new node in the cluster. You will have a new node added to the cluster to expand the cluster’s physical capacity.
* SO, PODs usually have a one-to-one relationship with containers running the application. To scale UP we create new PODs and to scale down we delete PODs. We do not add additional containers to an existing POD to scale your application.
* Reference - <https://kubernetes.io/docs/concepts/workloads/pods/>

### MULTICONTAINER POD



* PODs usually have a one-to-one relationship with the containers.
* But - we are not restricted to having a single container in a single POD.A single POD CAN have multiple containers, except for the fact that they are usually not multiple containers of the same kind.
* if our intention was to scale our application, then we would need to create additional PODs. But sometimes you might have a scenario where we have a helper container, that might be doing supporting task for our web application such as processing a user entered data, processing a file uploaded by the user etc. and we want these helper containers to live alongside our application container. In that case, we can have both of these container’s part of the same POD, so that when a new application container is created, the helper is also created and when it dies the helper also dies since they are part of the same POD.
* The two containers can also communicate with each other directly by referring to each other as ‘localhost’ since they share the same network namespace and the same storage space as well.

### PODS IN PRACTICE

|  |  |
| --- | --- |
| CREATING A POD WITH A GIVEN DOCKER IMAGE  Note   1. The POD name can be anything, but the image name must be the name of the Docker Image in the Docker-HUB 2. What if image is hosted in another registry apart from docker?? | **kubectl run <POD\_NAME> --image=<IMAGE\_NAME>**  **Example-**  **kubectl run nginx –image=nginx**  **Note – Using above command we can create a POD directly using Docker image. Another way to create image using Manifest file (Yaml configuration file)** |
| GET THE RUNNING PODS   * NAME – Name of the POD * READY (1/1) – **Number of containers in the POD which are in ready state** | **kubectl get pods** |
| DETAIL INFORMTAION OF A GIVEN POD(nginx) | **kubectl describe pod nginx** |
| 1. NAME - Name of the POD 2. IP – Internal IP Address of the POD 3. READY (1/2) – Number of containers in the POD Running / Total number of containers in a POD 4. NODE – The Node(worker node) in which the POD is deployed | |

## YAML

* YAML is a way to represent data like JSON and XML

QUICK COMPARISION

|  |  |  |
| --- | --- | --- |
| XML | JSON | YAML |
| <Servers>  <Server>  <name>Server1</name>  <owner>ABC</owner>  <created>21-01-22</created>  <status>Active</status>  </Server>  <Server>  <name>Server2</name>  <owner>ZYZ</owner>  <created>22-01-22</created>  <status>Inactive</status>  </Server>  </Servers> | {  "Servers": {  "Server": [  {  "name": "Server1",  "owner": "ABC",  "created": "21-01-22",  "status": "Active"  },  {  "name": "Server2",  "owner": "ZYZ",  "created": "22-01-22",  "status": "Inactive"  }  ]  }  } | Servers:  Server:  **-** name: Server1  owner: ABC  created: 21-01-22  status: Active  **-** name: Server2  owner: ZYZ  created: 22-01-22  status: Inactive |

|  |  |
| --- | --- |
| **KEY VALUE PAIR**   * The simplest data in YAML is key-value pair * There is always a “space” between “:” and the value | Fruit: Apple  Vegetables: Carrot  Liquid: Water |
| **ARRAY**  Array of Fruits. “-” indicates an element of an array | Fruits:  - Apple  - Banana  - Grapes |
| **DICTIONARY / MAP**  We should equal space for a property of each map item, so that they aligned together | Apple:  Calories: 105  Fat: 0.4 g  Carbs: 16 g  Banana:  Calories: 120  Fat: 0.5 g  Carbs: 18 g |
| **ALL COMBINED** | Fruits:  - Apple:  Calories: 105  Fat: 0.4 g  Carbs: 16 g  - Banana:  Calories: 120  Fat: 0.5 g  Carbs: 18 g |

## PODS WITH YAML

**Kubernetes uses YAML files as input for the creation of objects such as PODs, Replicas, Deployments, Services etc. The YAML file for k8 is Manifest file.**

|  |  |
| --- | --- |
| A Kubernetes definition file always contains 4 top level fields.   * **THE APIVERSION, KIND, METADATA AND SPEC.** * These are top level or root level properties. * These are all REQUIRED fields | SAMPLE MANIFEST FILE (pod-defination.yaml)  apiVersion: v1  kind: Pod  metadata:    name: my-first-pod    labels:      app: my-first-app      type: front-end  spec:    containers:      - name: nginx-container        image: nginx |
|  | |

|  |  |  |
| --- | --- | --- |
| apiVersion | * This is the version of the Kubernetes API we’re using to create the object. * Depending on what we are trying to create we must use the RIGHT apiVersion(refer below table). For POD - set the apiVersion as “v1” * Few other possible values for this field are apps/v1beta1, extensions/v1beta1 etc. | |
| kind | * The kind refers to the type of object we are trying to create, For POD - we will set it as Pod. * Other possible values here could be ReplicaSet or Deployment or Service. | |  |  | | --- | --- | | KIND | VERSION | | POD | v1 | | Service | v1 | | ReplicaSet | apps/v1 | | Deployment | apps/v1 | |
| metadata | * The metadata is data about the object like its name, labels etc. * Represented in form of dictionary/ Map * Labels (usability)- Say for example there are 100s of PODs running a front-end application, and 100’s of them running a backend application or a database, it will be DIFFICULT for you to group these PODs once they are deployed. If you label them now as front-end, back-end or database, you will be able to filter the PODs based on this label at a later point in time. * We can only specify name or labels or anything else that Kubernetes expects to be under metadata. We CANNOT add any other property under this. However, under labels we CAN have any kind of key or value pairs. | |
| spec | * Depending on the object we are going to create, this is where we provide additional information to Kubernetes pertaining to that object. This is going to be different for different objects, * Spec is a dictionary so add a property under it called containers, which is a list or an array. The reason this property is a list is because the PODs can have multiple containers within | |
| * Once the file is created, run the below command to creates a POD out of it   **kubectl create -f pod-definition.yml** | | |

## REPLICATION CONTROLLER AND REPLICASET

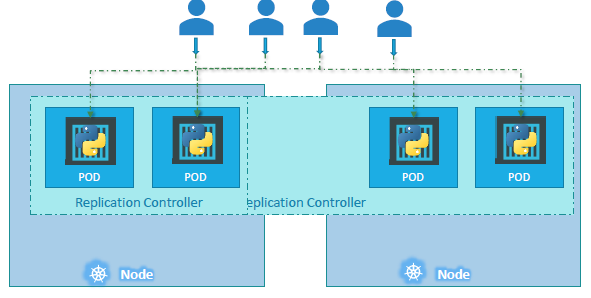
* Controllers are the brain behind Kubernetes. It’s a Kubernetes components associated with Master node.
* They are processes that monitor Kubernetes objects and respond accordingly.
* **NOTE** - Replication Controller and Replica Set are two similar terms. Both have the same purpose, but they are not the same. **Replication Controller is the older technology that is being replaced by Replica Set. Replica set is the new recommended way to setup replication**.
* RESPOSINILITIES OF REPLICATION CONTROLLER
  + **HIGH AVALIBILITY**
  + **LOAD BALANCING AND SCALING**

### HIGH AVAILIBILITY



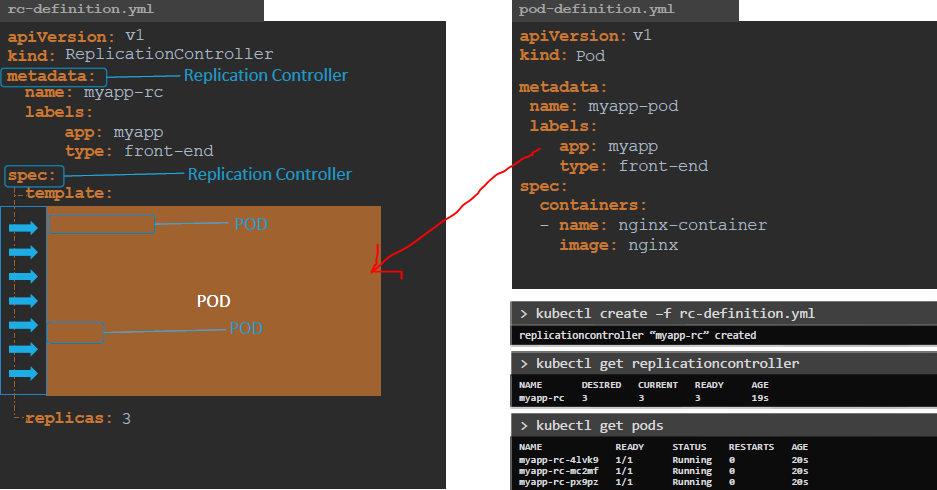
* Let’s say we have a scenario - where we had a single POD running our application. What if for some reason, our application crashes and the POD fails? Users will no longer be able to access our application. To prevent users from losing access to our application, we would like to have more than one instance or POD running at the same time. That way if one POD fails, we still have our application running on the other one.
* **The replication controller helps us run multiple instances of a single POD in the Kubernetes cluster thus providing High Availability.**
* Even if we have a single POD, the replication controller can help by automatically bringing up a new POD when the existing one fails. **Thus, the replication controller ensures that the specified number of PODs are always running.**

### LOAD BALANCING & SCALING



* Another reason we need replication controller is to create multiple PODs to share the load across them. For example, in this simple scenario we have a single POD serving a set of users. When the number of users increase, we deploy additional POD to balance the load across the two pods. If the demand further increases and if we were to run out of resources on the first node, we could deploy additional PODs across other nodes in the cluster.
* As in the above diagram - **the replication controller spans across multiple nodes in the cluster. It helps us balance the load across multiple pods on different nodes as well as we scale our application when the demand increases.**

### CREATING REPLICATION CONTROLLER



|  |  |  |  |
| --- | --- | --- | --- |
| apiVersion | The apiVersionis specific to what we are creating. In this case replication controller is supported in kubernetes apiVersion is v1. | | |
| kind | ReplicationController | | |
| metadata | Name- Name of the Replication Controller (myapp-rc) | | |
| specs | * The spec section defines what’s inside the object we are creating. In this case we know that the replication controller creates multiple instances of a POD. * We create a template section under spec to provide a POD template to be used by the replication controller to create replicas. * To DEFINE the POD template – It is similar to the pod-definition file. * We could re-use the contents of the same file to populate the template section.   + Move all the contents of the pod-definition file into the template section of the replication controller, except for the first two lines –which are apiVersion and kind. | | |
| EXAMPLE  apiVersion: v1  kind: ReplicationController  metadata:    name: my-rc    labels:      app: my-app-rc      type: front-end-app  spec:    template:      metadata:        name: my-app-pod        labels:          app: my-app          type: web-app      spec:        containers:          -  name: nginx-container             image: nginx    replicas: 3 | | * Looking at our file, we now have two metadata sections –one is for the Replication Controller and another for the POD and we have two spec sections –one for each. * We have nested two definition files together. The replication controller being the parent and the pod-definition being the child. * For replication controller, add another property to the spec called replicas and input the number of replicas we need under it. * COMMAND: **kubectl create -f <fileName>** * When the replication controller is created it first creates the PODs using the pod-definition template as many as required, which is 3 in this case. | |
| TO VIEW REPLICATION CONTROLLER | **kubectl get replicationcontroller** |
| TO VIEW THE CREATED PODS(3) | kubectl get pods |
|  | |
|  | | | |

QUESTION – EXPLAIN THE ROLE OF REPLICATION CONTROLLER AND REPLICASET?

* **We create a replication controller or replica set to ensure that we have 3 active PODs at anytime**
* We can use it to monitor existing pods, if the specified number of PODs (replicas) are not created the replica set will create them for us. Hence - the role of the replicaset is to monitor the pods and if any of them were to fail, deploy new ones.

### CREATING REPLICATION SET

* It is very similar to replication controller.

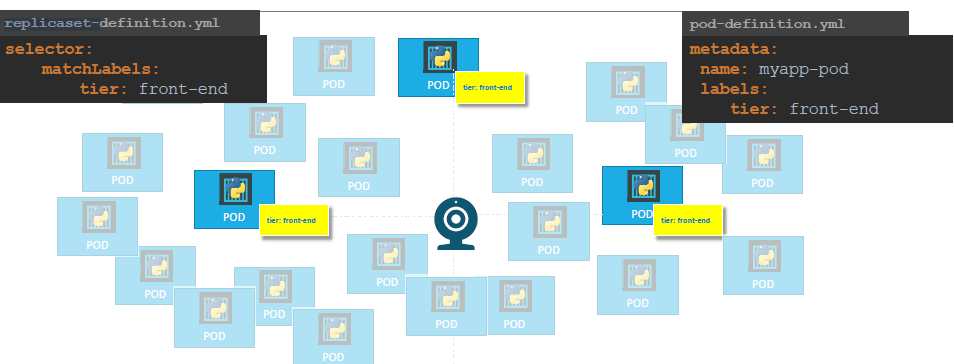
|  |  |
| --- | --- |
| apiVersion | apps/v1 |
| kind | ReplicaSet |

* There is one major difference between replication controller and replica set. Replica set requires a selector definition. The selector section helps the replicaset identify what pods fall under it.
* **QUESTION** - But why would we have to specify what PODs fall under it, if we have provided the contents of the pod-definition file itself in the template

**ANSWER**- It’s BECAUSE, replica set can ALSO manage pods that were not created as part of the replicaset creation. Say for example, there were pods created BEFORE the creation of the ReplicaSet that match the labels specified in the selector, the replica set will also take THOSE pods into consideration when creating the replicas.

|  |  |
| --- | --- |
| * **The selector is one of the major differences between replication controller and replica set. The selector is not a REQUIRED field in case of a replication controller, but it is still available**. * In Replication Controller – when we skip it, it assumes it to be the same as the labels provided in the pod-definition file. * In case of replica set a user input IS required for this property. It must be written in the form of matchLabels as shown here. The matchLabels selector simply matches the labels specified under it to the labels on the PODs.   Note - The replicaset selector also provides many other options for matching labels that were not available in a replication controller.   * To run the replica set definition file | apiVersion: apps/v1  kind: ReplicaSet  metadata:    name: my-replica-set    labels:      name: my-app      type: front-end-app  spec:    template:      metadata:        labels:          name: nginx-front-end-app      spec:        containers:          - name: nginx            image: nginx    replicas: 3    selector:      matchLabels:        name: nginx-front-end-app |

#### LABELS AND SELECTORS



* To understand the concept of label and selector - let us look at a simple scenario. Let say we deployed 3 instances of our frontend web application as 3 PODs.
* **The replica set is a process that monitors the pods. Now, how does the replicaset know what pods to monitor. There could be 100s of other PODs in the cluster running different application. This is where labelling our PODs during creation comes in handy. We could now provide these labels as a filter for replicaset. Under the selector section we use the matchLabels filter and provide the same label that we used while creating the pods. This way the replicaset knows which pods to monitor.**
* In the replicaset specification section we learned that there are 3 sections: Template, replicas and the selector. We need 3 replicas and we have updated our selector based on our discussion. Say for instance we have the same scenario as in the previous slide where we have 3 existing PODs that were created already, and we need to create a replica set to monitor the PODs to ensure there are a minimum of 3 running at all times. When the replication controller is created, it is NOT going to deploy a new instance of POD as 3 of them with matching labels are already created. In that case, do we really need to provide a template section in the replica-set specification, since we are not expecting the replicaset to create a new POD on deployment? Yes, we do, BECAUSE in case one of the PODs were to fail in the future, the replicaset needs to

create a new one to maintain the desired number of PODs. And for the replica set to create a new POD, the template definition section IS required.

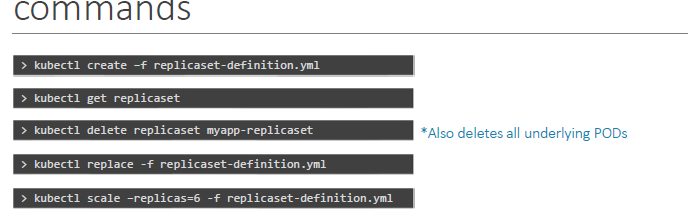
#### SCALLING A REPLICASET

1. Let’s say we started with 3 replicas and in the future, we decide to scale to 6. We have multiple ways to update the replicaset to scale to 6 replicas.
2. **UPDATING THE DEFINATION FILE**
   1. Step 1: update the number of replicas in the definition file to 6.
   2. Run the kubectl replace command – **kubeclt replace -f replicaset-defination.yml**
3. **USING KUBECTL SCALE COMMAND**

|  |  |
| --- | --- |
| **kubectl scale --replicas=10 -f replica-set.yaml** | Use the replicas parameter to provide the new number of replicas and specify the same file as input |
| kubectl scale –replicas=8 replicaset my-replica-set | You may either input the definition file or provide the replicaset name in the TYPE Name format. However, remember that using the file name as input will not result in the number of replicas being updated automatically in the file. In other words, the number of replicas in the replicaset-definition file will still be 3 even though you scaled your replicaset to have 6 replicas using the kubectl scale command and the file as input. |

|  |  |
| --- | --- |
| CREATING A REPLICASET | **kubectl create -f replica-set.yaml** |
| LIST ALL REPLICASET | **kubectl get replicaset** |
| DELETING REPLICASET | **kubectl delete replicaset <replicateSetName>**  (Note - This will delete underlying PODs too) |
| DELETING A POD IN REPLICASET |  |

Note - There are also options available for automatically scaling the replicaset based on load



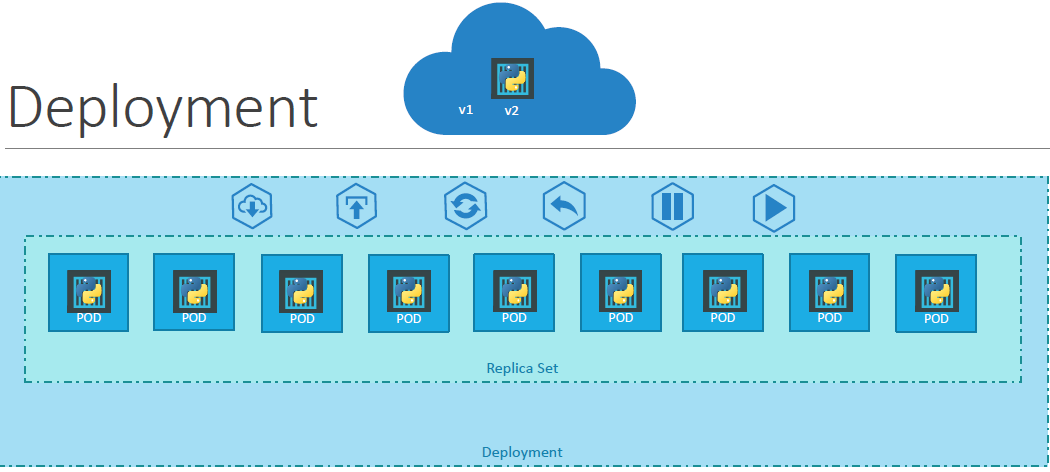
#### UPDATING THE YAML

|  |  |
| --- | --- |
| EDIT THE YAML | kubectl edit replicaset **my-replica-set** |
|  | * This will opens up running configuration of the replicate in an editor (like Notepad/ Vim Editor) * The is in memory file , which Kubernetes created for us to edit the configurations (it has lot of extra details in the file) * Make the changes (replica=10) and Save. This will immediately come in effect and spin up 10 pods in the replicaset |

## DEPLOYMENTS

In a traditional PROD environment - if want to deploy an application in a production environment. Say for example we have a web server that needs to be deployed in a production environment.

* We need multiple instances of webserver of the web server running for obvious reasons.
* Whenever newer versions of application builds become available on the docker registry, we need to UPGRADE our docker instances seamlessly.
* In PROD use-case, when we upgrade our instances, we never upgrade all of them at once. This may impact users accessing our applications, so we will be upgrading them one after the other. **This kind of upgrade is known as Rolling Updates**.
* Now, suppose one of the upgrades we performed resulted in an unexpected error and we can be able to undo the recent update(rollback).
* Hence In PROD - for the changes in the environment such as upgrading the underlying Webserver versions, scaling the environment, and modifying the resource allocations etc. We do not want to apply each change immediately after the command is run, instead we would like to apply a pause to our environment, make the changes and then resume so that all changes are rolled out together.
* **All above capabilities are available with the Kubernetes Deployments.**



* As we know, each container is encapsulated in PODs. Multiple such PODs are deployed using Replication Controllers or Replica Sets.
* And then comes Deployment which is a Kubernetes object that comes higher in the hierarchy. **The deployment provides us with capabilities to upgrade the underlying instances seamlessly using rolling updates, undo changes, and pause and resume changes to deployments.**

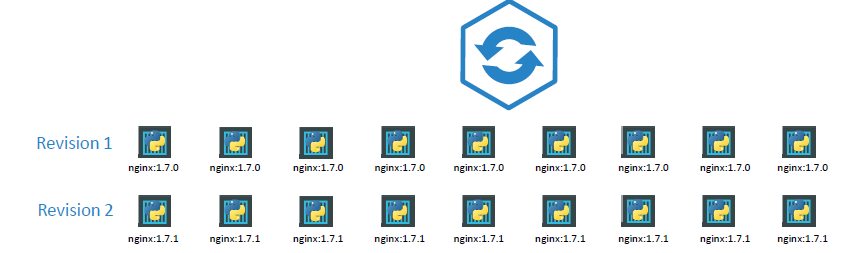
### CREATING DEPLOYMENT

* Once the file is ready run the kubectl create command and specify deployment definition file. Then run the kubectlget deployments command to see the newly created deployment.
* The deployment automatically creates a replica set. When we run the kubectl get replicaset command – we will be able to see a new replicaset in the name of the deployment.
* The replicaset ultimately create pods, so if we run the **kubectl get pods** command we will be able to see the pods with the name of the deployment and the replicaset.

|  |  |  |
| --- | --- | --- |
| apiVersion | apps/v1 | apiVersion: apps/v1  kind: Deployment  metadata:    name: myapp-deployment    labels:      tier: front-end      app: nginx  spec:    template:      metadata:       name: nginx-2       labels:        app: myapp      spec:        containers:          - name: nginx            image: nginx    replicas: 4    selector:      matchLabels:        app: myapp |
| kind | Deployment |
| The contents of the deployment-definition file are exactly like the replicaset definition file, except for the kind, which is now going to be **Deployment**.  QUESTION?  USING COMMAND  **kubectl create deployment http-frontend --image=httpd:2.4-alpine --replicas=3** | |
|  | | |

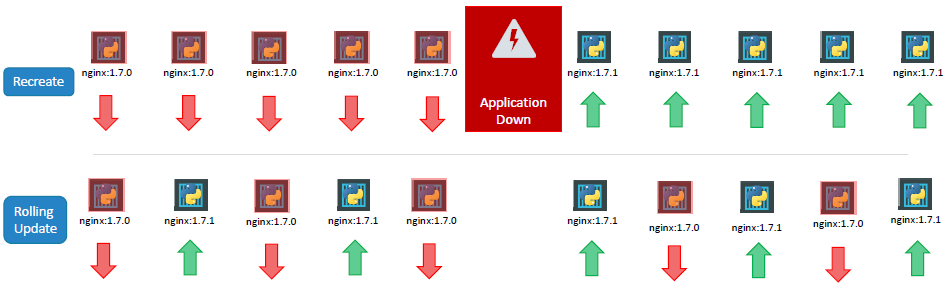
### UPDATE AND ROLLBACK

#### ROLLOUT AND VERSIONING

* Whenever we create a new deployment or upgrade the images in an existing deployment it triggers a Rollout.
* **A rollout is the process of gradually deploying or upgrading the application containers**.
* Let’s refer the below image - When we first create a deployment, it triggers a rollout. A new rollout creates a new Deployment revision. (Revision 1). In the future when the application is upgraded –meaning when the container version is updated to a new one –a new rollout is triggered, and a new deployment revision is created (Revision 2)
* The revisioning helps us keep track of the changes made to our deployment and enables us to roll back to a previous version of deployment if necessary

|  |  |
| --- | --- |
| **COMMAND** | **EXAMPLE** |
| kubectl rollout status deployment/<***deployment-name>*** | kubectl rollout status deployment/**myapp-deployment** |
| kubectl rollout history deployment/***<deployment-name>*** | kubectl rollout history deployment/myapp-deployment |

#### DEPLOYMENT STRATEGIES



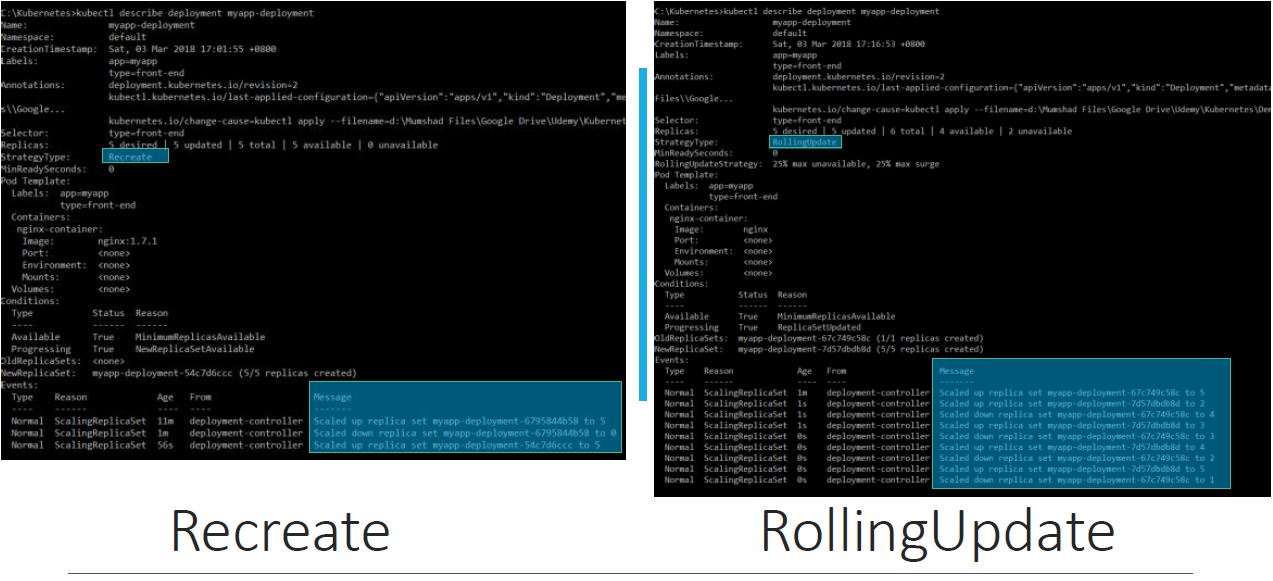
**There are two types of deployment strategies.**

1. **RECREATE STRATEGY**:
   1. Let’s say we have 5 replicas of the web application instance deployed. One way to upgrade these to a newer version is to destroy all of these and then create newer versions of application instances.
   2. Meaning first, destroy the 5 running instances and then deploy 5 new instances of the new application version.
   3. The problem with this- is that during the period after the older versions are down and before any newer version is up, the application is down and inaccessible to users. **This strategy is known as the Recreate strategy**
2. **ROLLING UPDATE STRATEGY:**
   1. The second strategy is where we do not destroy all of them at once. Instead, we take down the older version and bring up a newer version one by one. This way the application never goes down and the upgrade is seamless.
   2. Remember, if we do not specify a strategy while creating the deployment, it will assume it to be Rolling Update. In other words, **Rolling Update is the default Deployment Strategy.**

**COMMAND - kubectl describe deployment <*deployment\_name*>**

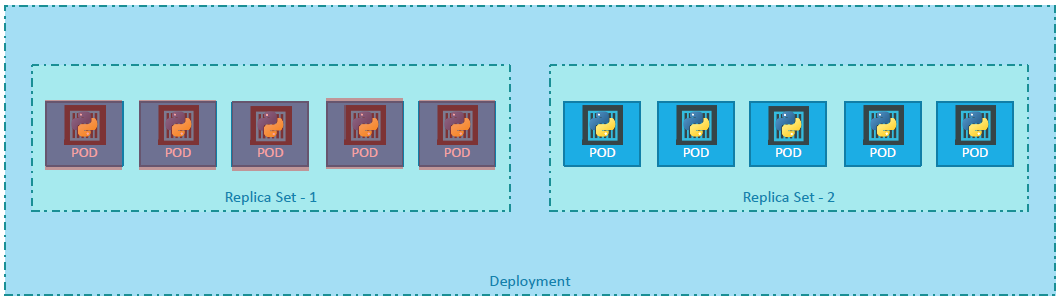
*When we refer the below deployment details -* T*he difference between the recreate and rolling update strategies can also be seen when we view the deployments in detail.*

* When the Recreate strategy was used the events indicate that the old replica set was scaled down to 0 first and the new replica set scaled up to 5.
* However, when the Rolling Update strategy was used the old replica set was scaled down one at a time simultaneously scaling up the new replica set one at a time.



#### UPGRADE AND ROLLOUT – UNDER THE HOOD

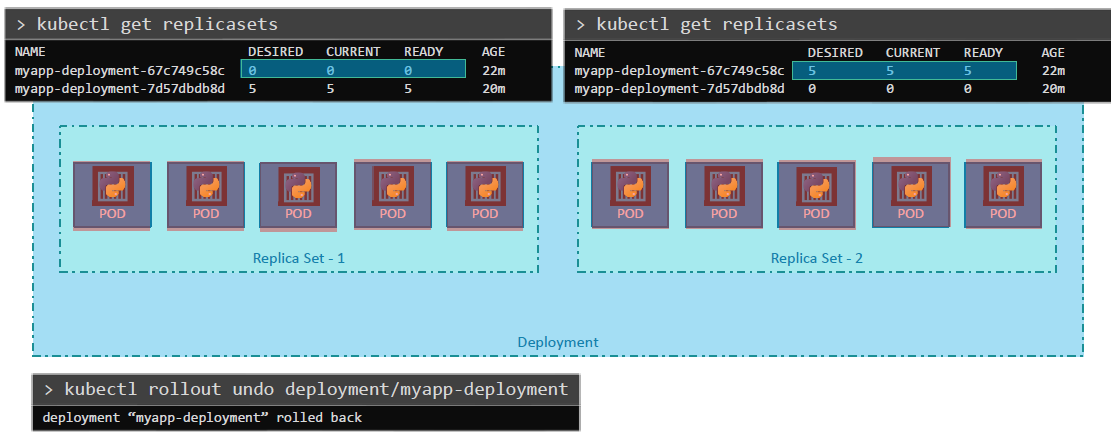
##### UPGRADE

Let’s look at how a deployment performs an upgrade under the hoods. 

* When a new deployment is created, say to deploy 5 replicas, **it first creates a Replicaset automatically**, which in turn creates the number of PODs required to meet the number of replicas.
* When we upgrade the application, **the Kubernetes deployment object creates a NEW replicaset under the hoods and starts deploying the containers there.** At the same time taking down the PODs in the old replica-set following a Rolling Update strategy.

|  |  |
| --- | --- |
|  | In upgrade - We see the old replicaset with 0 PODs and the new replicaset with 5 PODs. |

##### ROLLBACK

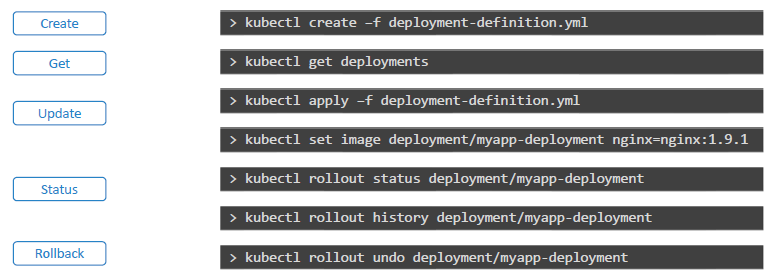


1. For instance, once we upgrade our application and something wrong with the new version of build we used to upgrade. Kubernetes deployments allow us to rollback to a previous revision.

|  |  |
| --- | --- |
| **kubectl rollout undo *<deployment\_name>*** | The deployment will then destroy the PODs in the new replicaset and bring the older ones up in the old replicaset. And the application is back to its older format. |

1. When we compare the output of the **kubectl get replicasets** command, before and after the rollback, we can able to notice this difference. Before the rollback the first replicaset had 0 PODs and the new replicaset had 5 PODs and this is reversed after the rollback is finished.

#### COMMANDS SUMMARY



##### ROLLOUT HISTORY

|  |
| --- |
| In this following command, there is no entry in the in the **CHANGE-CAUSE**  **kubectl rollout history deployment/myapp-deploymen**t |
| **kubectl create -f deployments.yaml --record**  The record option to record the CHANGE-CAUSE in the history |

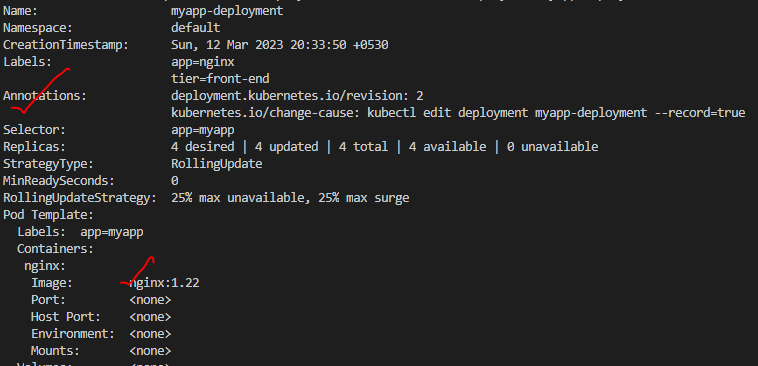
##### UPDATE

USING EDIT COMMAND

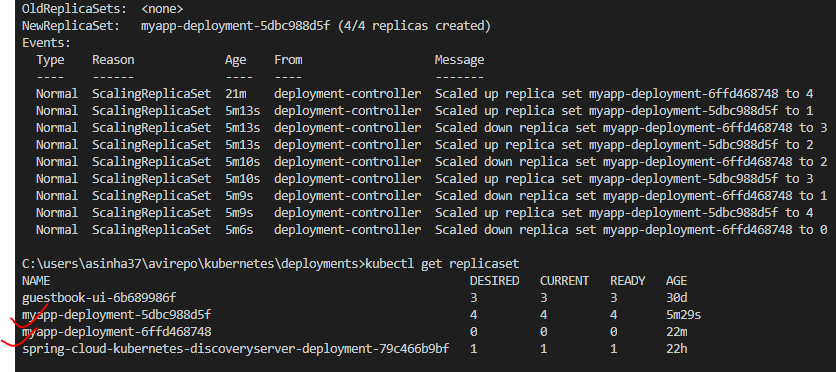
|  |  |
| --- | --- |
| **kubectl edit deployment myapp-deployment --record** | We can update the container by editing the YAML file. We will we updating the image name to older version. |

1. When we updated the image – this deployment happened with Rollback update strategy. The OLD pods(replicaset) go down and NEW pod (replicaset) scale up.

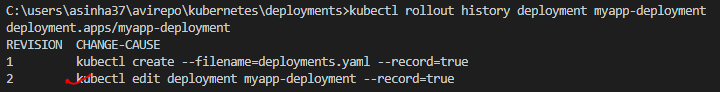
POD DETAILS



REPLICA SET



ROLLOUT HISTORY

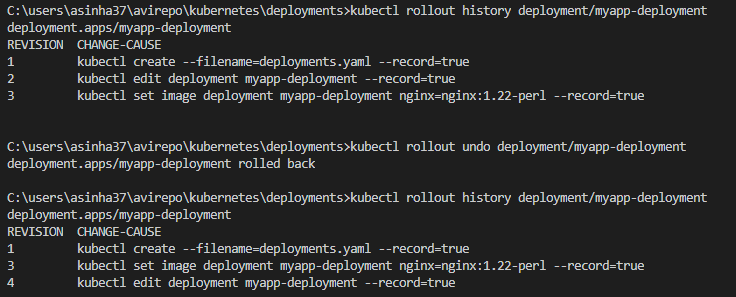


1. USING SET IMAGE COMMAND

|  |  |
| --- | --- |
| **kubectl set image deployment myapp-deployment nginx=nginx:1.22-perl** | * This command will update the POD to use **nginx:1.22-perl** image * <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#updating-a-deployment> |

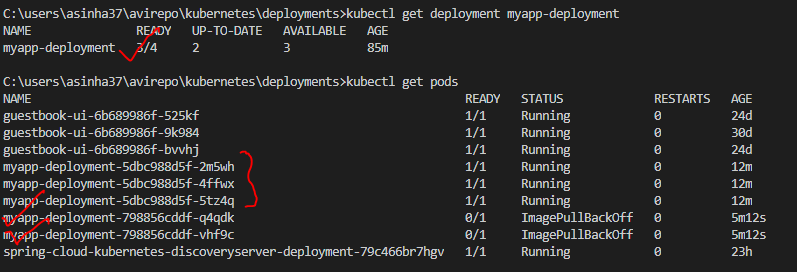
UNDO ROLLBACK

|  |  |
| --- | --- |
| **kubectl rollout undo deployment/myapp-deployment** | * The deployment has multiple revision (3 revisions). In order to rollback to previous version |



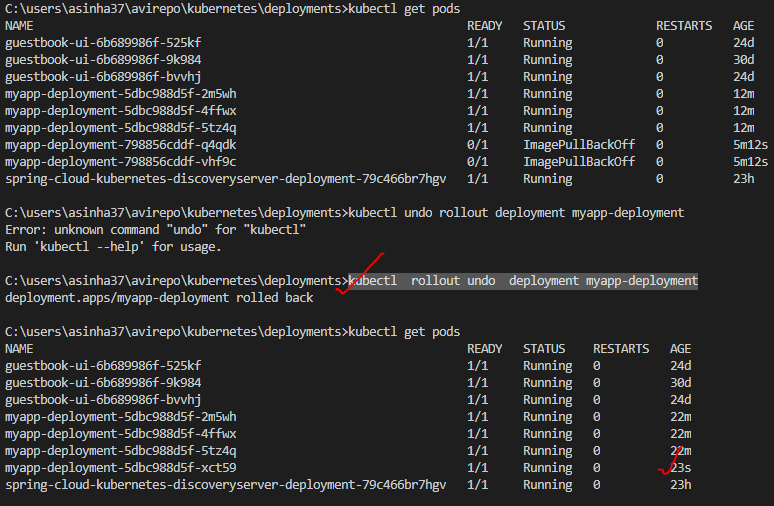
|  |  |
| --- | --- |
| Scenarios – Lets consider an example – if we update the image which does not exist (wrong image). |  |
|  | The rollout will be stuck as the image name is incorrect.Hence the deployment will fail |

* **The deployment will terminate one of the POD to create a new POD with the new image.** Note – As per rollback update deployment strategy - all the pods are not updated at once.
* Hence – due to this– even though the image is wrong, the application is not impacted and end user can able to access the application via 3 running PODs



Note – To fix the above issue – we can undo the rollout the deployment – Now this will create 1 more POD keeping other 3 POD as is :

**kubectl rollout undo deployment myapp-deployment**

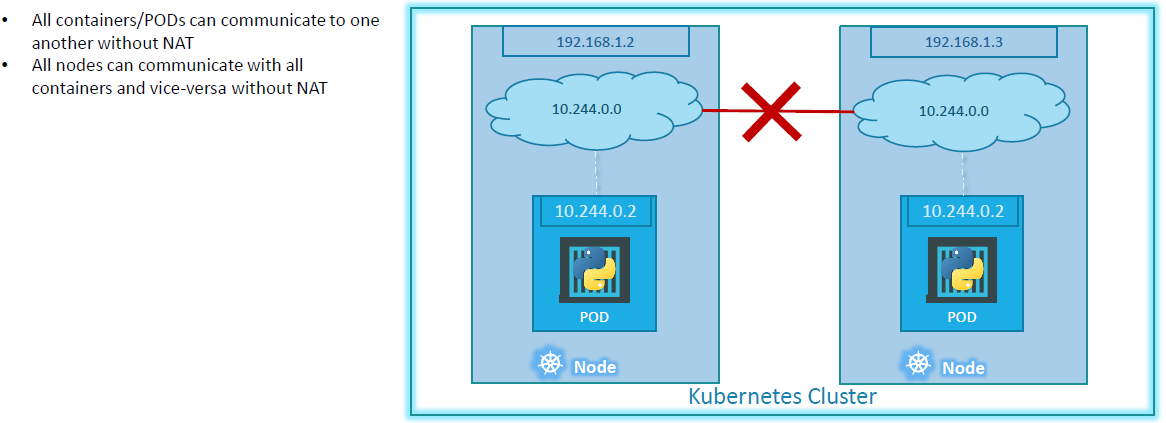


## NETWORKING

### SINGLE NODE

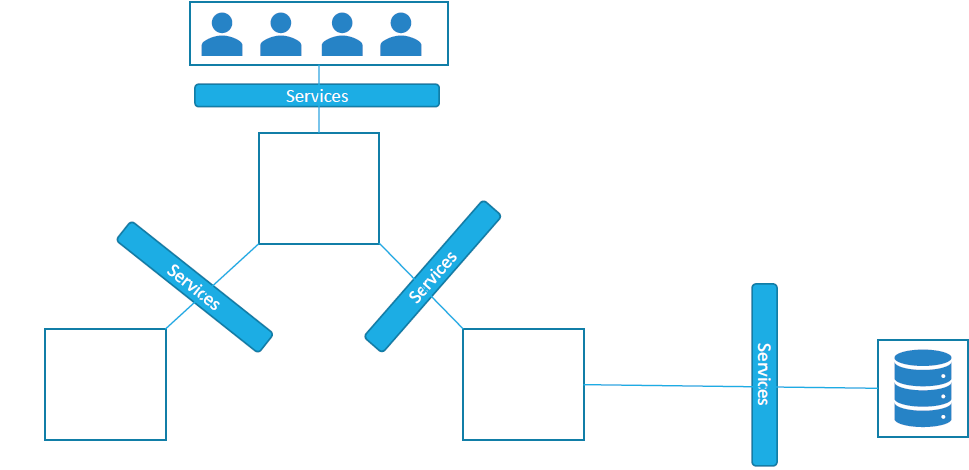
|  |  |
| --- | --- |
|  | * Starting with a single node kubernetes cluster. The node has an IP address, say it is 192.168.1.2. In this case, this is the IP address can be used to access the kubernetes node, SSH into it etc. * Let’s say - on the single node kubernetes cluster we have created a Single POD. As we know a POD hosts a container. Unlike in the docker world where an IP address is always assigned to a Docker CONTAINER, but * **In Kubernetes the IP address is assigned to a POD. Each POD in kubernetes gets its own internal IP Address**. * **For example -** In this case its in the range 10.244 series, then the IP assigned to the POD is 10.244.0.2. * When Kubernetes is initially configured it creates an internal private network with the address 10.244.0.0 and all PODs are attached to it. When we deploy multiple PODs, they all get a separate IP assigned. * Although- The PODs can communicate to each other through this IP. But accessing other PODs using this internal IP address MAY is not a good idea as its subject to change when PODs are recreated. |

### MULTIPLE NODES(CLUSTER NETWORKING)



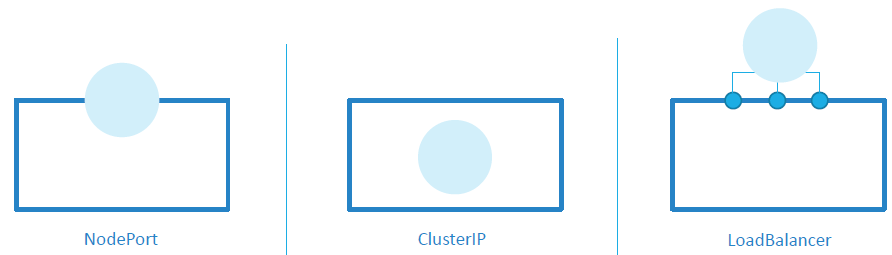
* In this case of two nodes running kubernetes and they have IP addresses 192.168.1.2 and 192.168.1.3 assigned to them. Note that they are not part of the same cluster yet. Each of them has a single POD deployed. As discussed in the previous slide these pods are attached to an internal network and they have their own IP addresses assigned. HOWEVER, if you look at the network addresses, you can see that they are the same. The two networks have an address 10.244.0.0 and the PODs deployed have the same address too.
* This is NOT going to work well when the nodes are part of the same cluster. The PODs have the same IP addresses assigned to them and that will lead to IP conflicts in the network. Now that’s ONE problem. When a kubernetes cluster is SETUP, kubernetes does NOT automatically setup any kind of networking to handle these issues. As a matter of fact, kubernetes expects US to setup networking to meet certain fundamental requirements. Some of these are that all the containers or PODs in a kubernetes cluster MUST be able to communicate with one another without having to configure NAT. All nodes must be able to communicate with containers and all containers must be able to communicate with the nodes in the cluster. Kubernetes expects US to setup a networking solution that meets these criteria.

## SERVICES



* Kubernetes Services enable communication between various components within and outside of the application. Kubernetes Services helps us connect applications together with other applications or users.
* For example, our application has groups of PODs running various sections, such as a group for serving front-end load to users, another group running back-end processes, and a third group connecting to an external data source. It is Services that enable connectivity between these groups of PODs.
* Services enable the front-end application to be made available to users, it helps communication between back-end and front-end PODs and helps in establishing connectivity to an external data source. Thus, services enable loose coupling between microservices in our application.

### SERVICE TYPES



|  |  |
| --- | --- |
| **NODEPORT SERVICE** | * NodePort where the service makes an internal POD accessible on a Port on the Node |
| **CLUSTER IP** | * Cluster IP service creates a virtual IP inside the cluster to enable communication between different services such as a set of front-end servers to a set of backend-servers. |
| **LOAD BALANCER** | * Load Balancer service it provisions a load balancer for our service in supported cloud providers. * A good example of that would be to distribute load across different web servers. We will now look at Each of these in a bit more detail along with some Demos. |

### USE CASE OF SERVICES

* As we know, PODs communicate with each other through internal networking. To understand external communication- Let say we deployed our POD having a web application running on it.

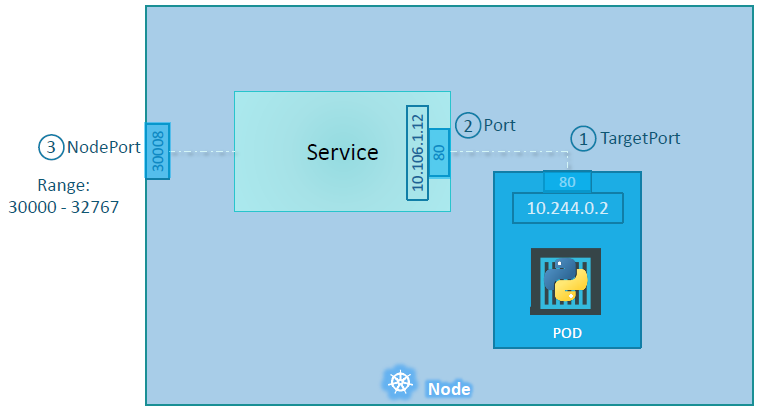
**Now how do WE as an external user access the web page?**

1. First of all, let’s look at the existing setup. The Kubernetes Node has an IP address and that is 192.168.1.2. My laptop is on the same network as well, so it has an IP address 192.168.1.10. The internal POD network is in the range 10.244.0.0 and the POD has an IP 10.244.0.2. Clearly, we cannot ping or access the POD at address 10.244.0.2 as it’s in a separate network.

**So, what are the options to see the webpage?**

* First, if we were to SSH into the kubernetes node at 192.168.1.2, from the node, we would be able to access the POD’s webpage by doing a curl or if the node has a GUI, we could fire up a browser and see the webpage in a browser following the address http://10.244.0.2. But this is from inside the kubernetes Node and that’s not what we really want. We want to be able to access the web server from my own laptop without having to SSH into the node and simply by accessing the IP of the kubernetes node. So, we need something in the middle to help us map requests to the node from our laptop through the node to the POD running the web container.
* That is where the kubernetes service comes into play. The kubernetes service is an object just like PODs. One of its use cases is to listen to a port on the Node and forward requests on that port to a port on the POD running the web application. **This type of service is known as a NodePort service because the service listens to a port on the Node and forwards requests to PODs.**

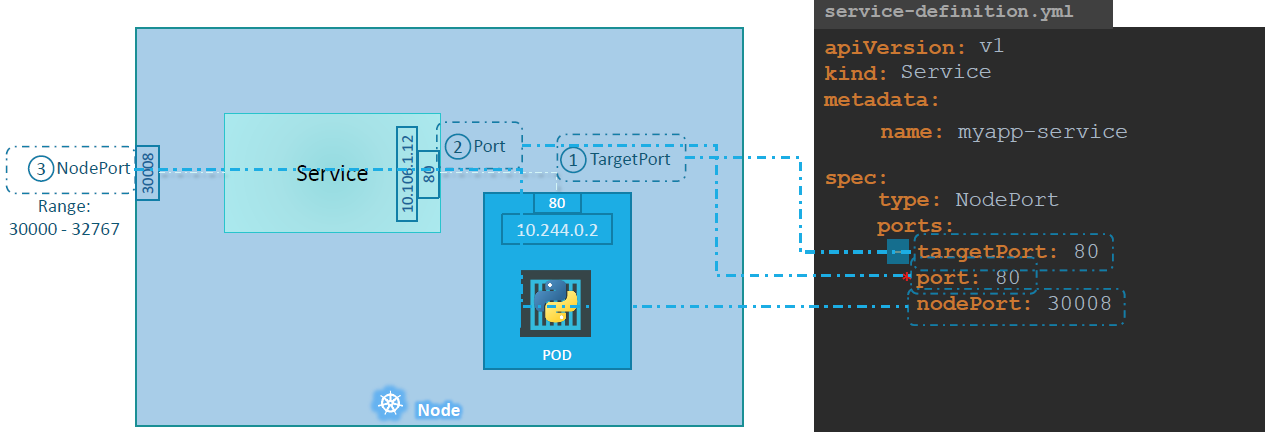
### NODEPORT SERVICE



In Node Port Service - There are 3 ports involved.

|  |  |
| --- | --- |
| **TARGET PORT** | * The port on the POD where the actual web server is running is port 80. * It is referred to as the **targetPort** because that is where the service forwards the requests to |
| **PORT**  **(SERVICE PORT)** | * The second port is the port on the service itself. * It is simply referred to as the port. (*Note- these terms are from the viewpoint of the service.*). * The service is in fact like a virtual server inside the node. Inside the cluster it has its own IP address. And that IP address is called the Cluster-IP of the service |
| **PORT OF THE NODE (NODE PORT)** | * The third one is the port on the Node itself which we use to access the web server externally called e NodePort(it is 30008). * **NodePorts can only be in a valid range which is from 30000 to 32767.** |

#### CREATING A NODEPORT SERVICE

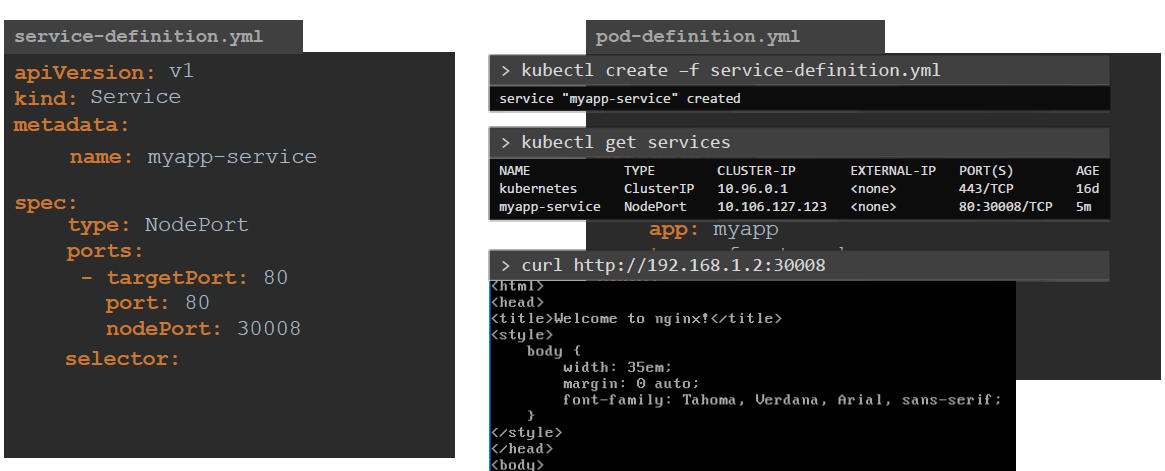


|  |  |
| --- | --- |
| apiVersion | v1 |
| Kind | Service |
| Spec/type | * In the spec section of a service, we have type and ports. The type refers to the type of service we are creating. It could be ***ClusterIP, NodePort, or LoadBalancer*** |
| Spec/Ports | In above example - The ports in the spec section. (ports is an array)   * The first type of port is the targetPort, which we will set to 80. * The next one is simply port, which is the port on the service object, and we will set that to 80 as well. * The third is NodePort which we will set to 30008 or any number in the valid range * Note - Out of these, the only mandatory field is port . If we don’t provide a targetPort it is assumed to be the same as port and if we don’t provide a NodePort a free port in the valid range between 30000 and 32767 is automatically allocated. |

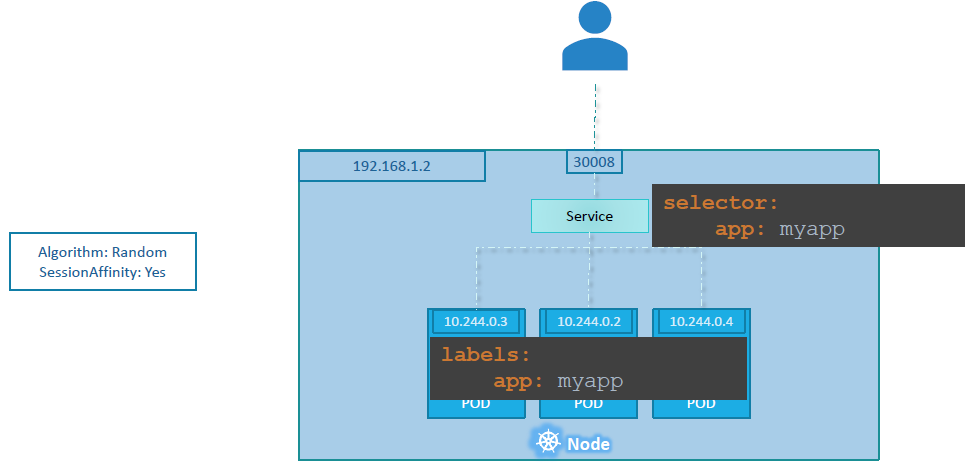
* **To connect the service with a POD we make use of use labels and selectors to link these together. We know that the POD was created with a label. We need to bring that label into this service definition file.**
* Under the selector provide a list of labels to identify the POD. For this refer to the pod-definition file used to create the POD. Pull the labels from the pod-definition file and place it under the selector section. This links the service to the pod.

|  |  |
| --- | --- |
| Kubectl get services | * Listing the service, their cluster-ip and the mapped ports. * The type is NodePort as we created and the port on the node automatically assigned is 32432 |

**EXAMPLE**

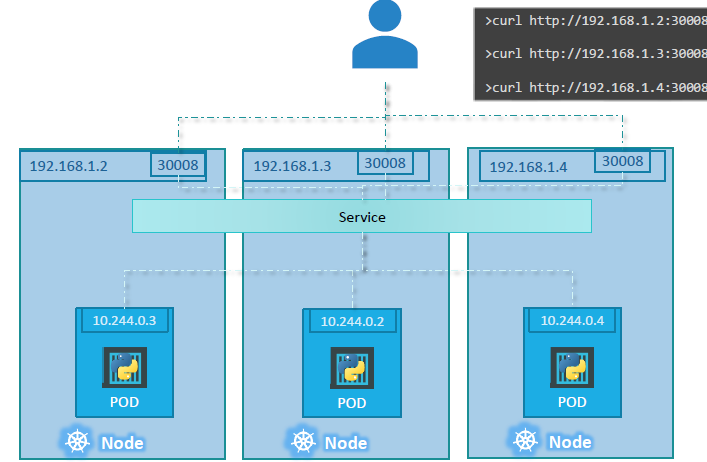


##### MULTIPLE PODS IN A NODE



* Usually ,in a production environment we have multiple instances of the web application running for high-availability and load balancing purposes. In this case we have multiple similar PODs running our web application.
* They all have the same labels with a **key app set to value myapp**. The same label is used as a selector during the creation of the service. So, when the service is created, it looks for matching PODs with the labels and finds 3 of them. The service then automatically selects all the 3 PODs as endpoints to forward the external requests coming from the user.
* The algorithm it uses to balance load is a random algorithm. Thus, the service acts as a built-in load balancer to distribute load across different PODs.

##### MULTIPLE PODS IN A CLUSTER (MULTIPLE NODES)

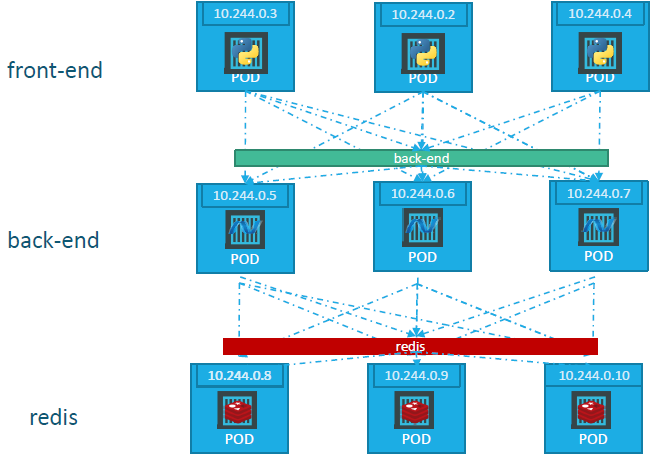


* When the PODs are distributed across multiple nodes. In this case we have the web application on PODs on separate nodes in the cluster. When we create a service, without us having to do ANY kind of additional configuration, **kubernetes creates a service that spans across all the nodes in the cluster and maps the target port to the SAME NodePort on all the nodes in the cluster.** This way we can access our application using the IP of any node in the cluster and using the same port number which in this case is 30008.

EXAMPLE

|  |  |
| --- | --- |
| Deployment.yaml   1. Run the deployment Yaml. This will start the “nginx” POD with a selector “**app:myapp**” | Service-defination.yaml |
| apiVersion: apps/v1  kind: Deployment  metadata:    name: myapp-deployment    labels:      tier: front-end      app: nginx  spec:    template:      metadata:       name: nginx-2       labels:        app: myapp      spec:        containers:          - name: nginx            image: nginx    replicas: 4    selector:      matchLabels:        app: myapp | apiVersion: v1  kind: Service  metadata:    name: myapp-service  spec:    type: NodePort    ports:      - port: 80        targetPort: 80        nodePort: 30002    selector:      app: myapp   1. **CREATING A SERVICE:**   Kubectl create -f service-defination.yaml   1. ACCESS THE POD USING NODE IP ADDRESS   In case of minikube below command will give the node IP addess  **minikube service <service\_name> --url** |

### CLUSTER-IP SERVICE



A full stack web application typically has different kinds of PODs hosting different parts of an application. For example,

1. PODs running a front-end web server,
2. PODs running a backend server,
3. Set of PODs running a key-value store like Redis,
4. Set of PODs running a persistent database like MySQL etc.

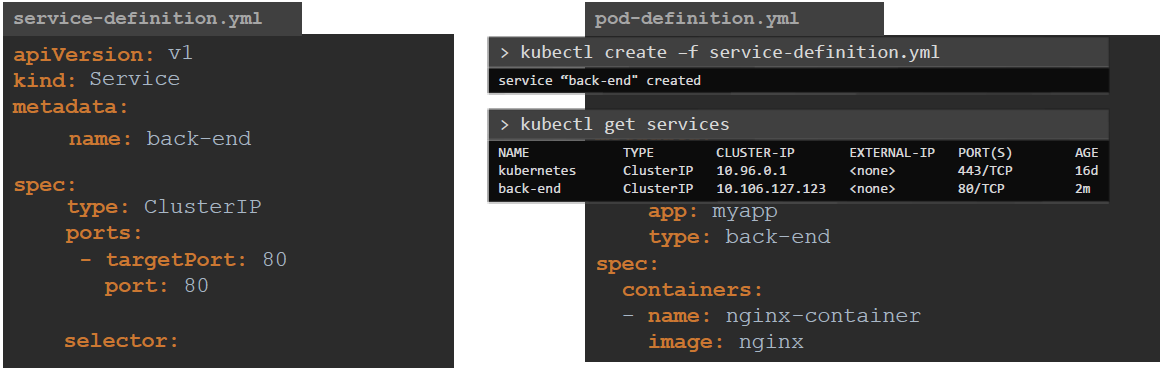
**QUESTION**

1. The web front-end servers need to connect to the backend-workers and the backend-workers need to connect to database as well as the redis services. **So what IS the right way to establish connectivity between these PODs**?
2. The PODs all have an IP address assigned to them as we can see on the screen. But these IPs are not static, these PODs can go down anytime, and new PODs are created all the time –and so we CANNOT rely on these IP addresses for internal communication within the application. Also, **what if the first front-end POD at 10.244.0.3 need to connect to a backend service? Which of the 3 would it go to and who makes that decision**?

**ANSWER**

* A Kubernetes service can help us group these PODs together and provide a single interface to access the PODs in a group.
* For example, 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, creating additional services for Redis and allow the backend PODs to access the redis system through this service. This enables us to 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 services. Each service gets an IP and name assigned to it inside the cluster and that is the name that should be used by other PODs to access the service. This type of service is known as **ClusterIP**.

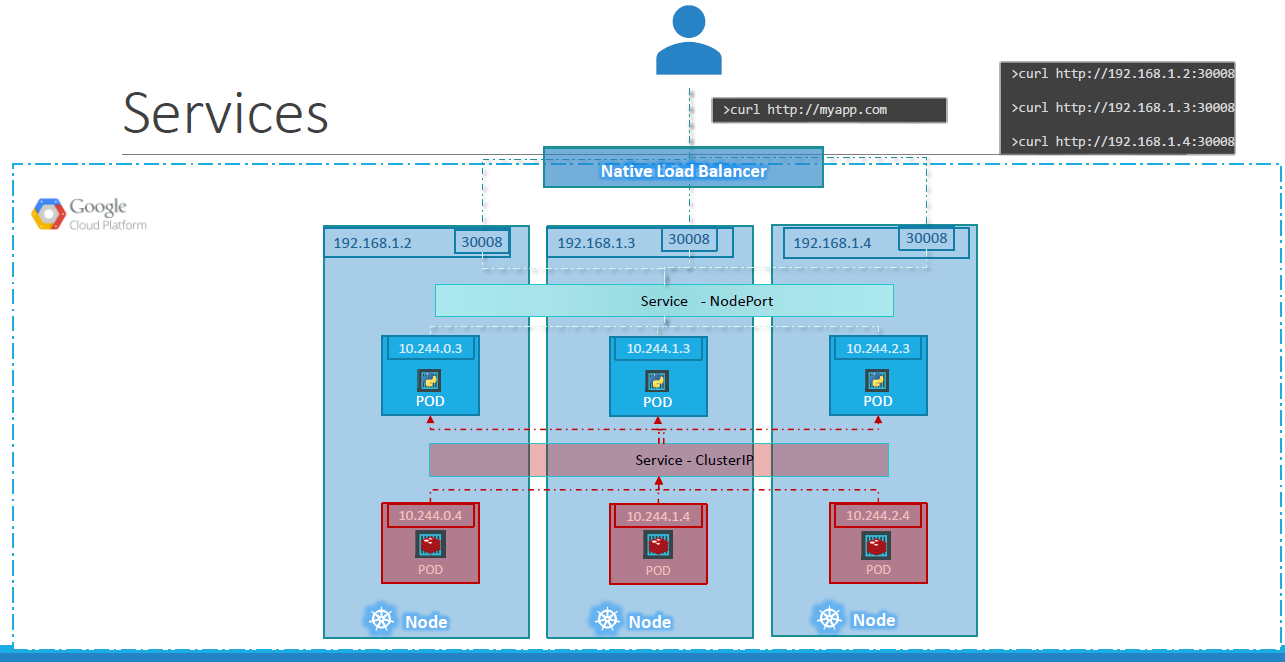
#### CREATING A CLUSTER-IP SERVICE



* Under Specification we have type and ports.

|  |  |
| --- | --- |
| type | The type is ClusterIP. In fact, ClusterIP is the default type, so even if we didn’t specify it, it will automatically assume it to be ClusterIP |
| ports | Under ports we have a targetPort and port. **The target port is the port were the back-end is exposed, which in this case is 80. And the port is where the service is exposed which is 80 as well** |
| selector | To link the service to a set of PODs, we use selector |

### LOADBALANCER SERVICE



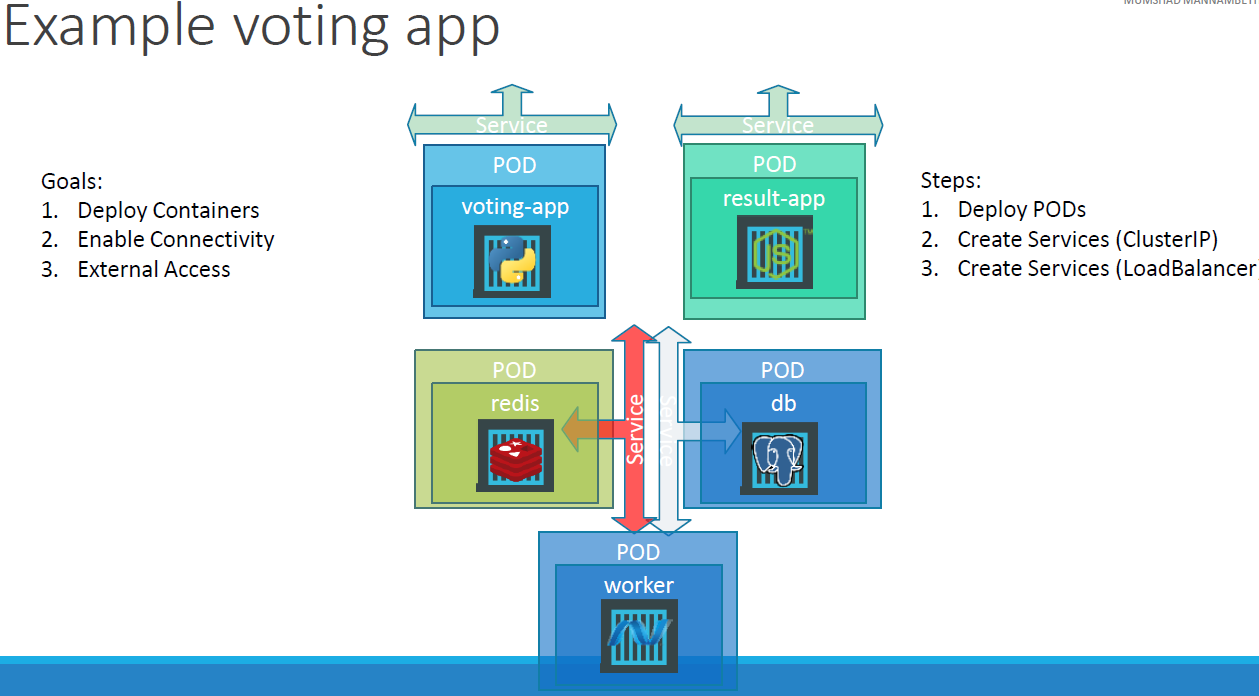
* Let’s consider an example - We have a 3 node cluster with IPs 192.168.1.2,3 and 4. Our application is two tier:
  + A database service and a front-end web service for users to access the application. The default service type –known as ClusterIP–makes a service, such as a redis or database service available internally within the kubernetes cluster for other applications to consume.
  + The next tier in application happens to be a python based web front-end. This application connects to the backend using Service created for the redis service. To expose the application to the end users, we create another service of type NodePort. Creating a service of type NodePort exposes the application on a high-end port of the Node and the users can access the application at any IP of my nodes with the port 30008.

**QUESTION**

Now, what IP do we will give the end users to access your application? We cannot give them all three and let them choose one of their own. What end users really want is a single URL to access the application. For this, we will be required to setup a separate Load Balancer VM in our environment. In this case we deploy a new VM for load balancer purposes and configure it to forward requests that come to it to any of the IPs of the Kubernetes nodes. We can then configure organizations DNS to point to this load balancer when a user hosts <http://myapp.com> .

Note - Kubernetes sets it up for us. Kubernetes has built-in integration with supported cloud platforms.

## SAMPLE APP (VOTING APP)



* This is a sample voting application which provides an interface for a user to vote and another interface to show the esults.
* The application consists of various components such as
  + The voting app, which is a web application developed in Python to provide the user with an interface to choose between two options a cat and a dog. When we select, the vote is stored in Redis.
  + This vote is then processed by the worker, which is an application written in dot net. The worker application takes the new vote and updates the persistent database, which is a PostgreSQL. In our case, the PostgreSQL simply has a table with a number of votes for each category cats and dogs. In this case, it increments the number of votes for cats as our vote was for cats.
  + Finally, the result of the vote is displayed in a web interface, which is another web application developed in Node.js. This resulting application rates the count of votes from the Postgres SQL database and displays it to the user.

Assumption - Let us assume that all images of applications are already built and are available on Docker Repository.

### GOAL

Our goal is to

1. Deploy these containers, these applications as containers on a Kubernetes cluster.
2. Enable connectivity between the containers so that the applications can access each other and the databases
3. Enable external access for the external facing applications which are the voting and the result app so that the users can access the web browser.

STEPS:

* We must first deploy these applications as a pod on our Kubernetes cluster.
* Once the pods are deployed, the next step is to enable connectivity between the services.
* The redis database is accessed by the voting app and the worker app. The voting app saves the vote to the Redis database, and the worker app reads the vote from the redis database.
* The PostgreSQL database is accessed by the Worker app to update it with the total count of votes, and it's also accessed by the result app to read the total count of votes to be displayed in the resulting web page in the browser.