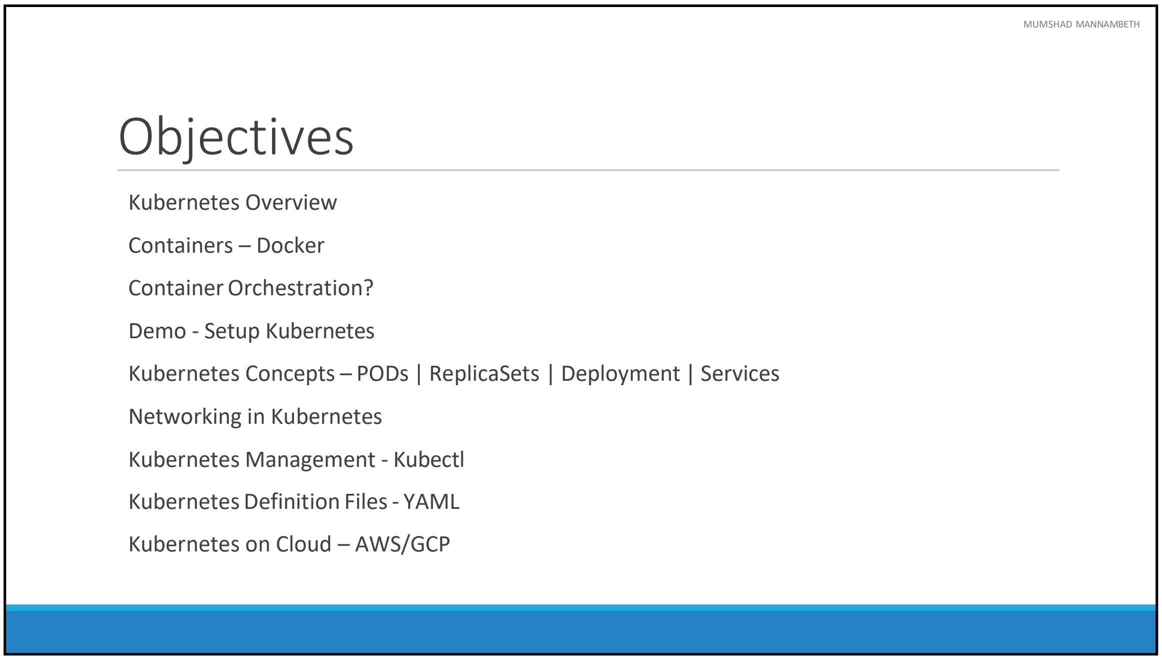

MUMSHAD MANNAMBETH

Objectives

Kubernetes Overview

Containers – Docker

Container Orchestration?

Demo - Setup Kubernetes

Kubernetes Concepts – PODs | ReplicaSets | Deployment | Services

Networking in Kubernetes

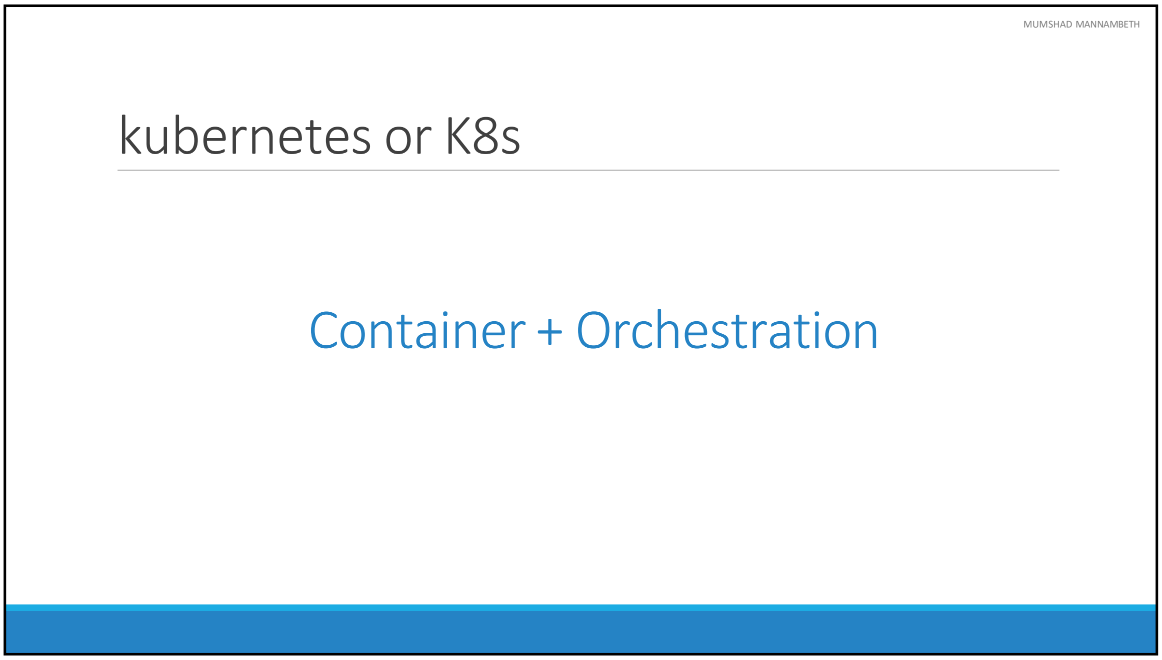
Kubernetes Management - Kubectl

Kubernetes Definition Files - YAML

Kubernetes on Cloud – AWS/GCP

Moving on the topics we will be covering. In this course we will go through the basics of Kubernetes, we will try to understand what containers are and what container orchestration is. We will see different ways of setting up and getting started with Kubernetes. We will go through various concepts such as PODs, ReplicaSets, Deployments and Services. We will understand the basics of Networking in kubernetes. We will also spend some time working with kubectl command line utility and developing kubernetes YAML definition files. And finally we will see how to deploy a microservices application on a public cloud platform like Google Cloud.

As always feel free to go through this course at your own pace. There may be sections in the course that you may be familiar with already, and so feel free to skip them. Let’s get started and I will see you in the first module.



MUMSHAD MANNAMBETH

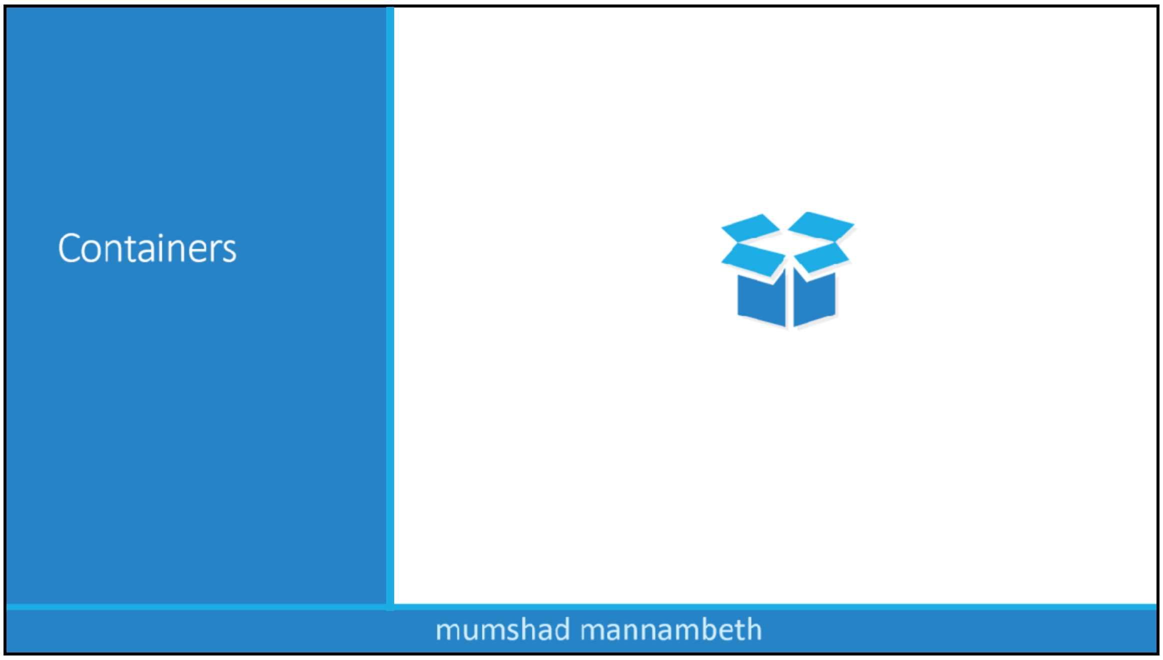
kubernetes or K8s

Container + Orchestration

In this lecture we will go through an overview of Kubernetes.

Kubernetes also known as K8s was built by Google based on their experience running containers in production. It is now an open-source project and is arguably one of the best and most popular container orchestration technologies out there. In this lecture we will try to understand Kubernetes at a high level.

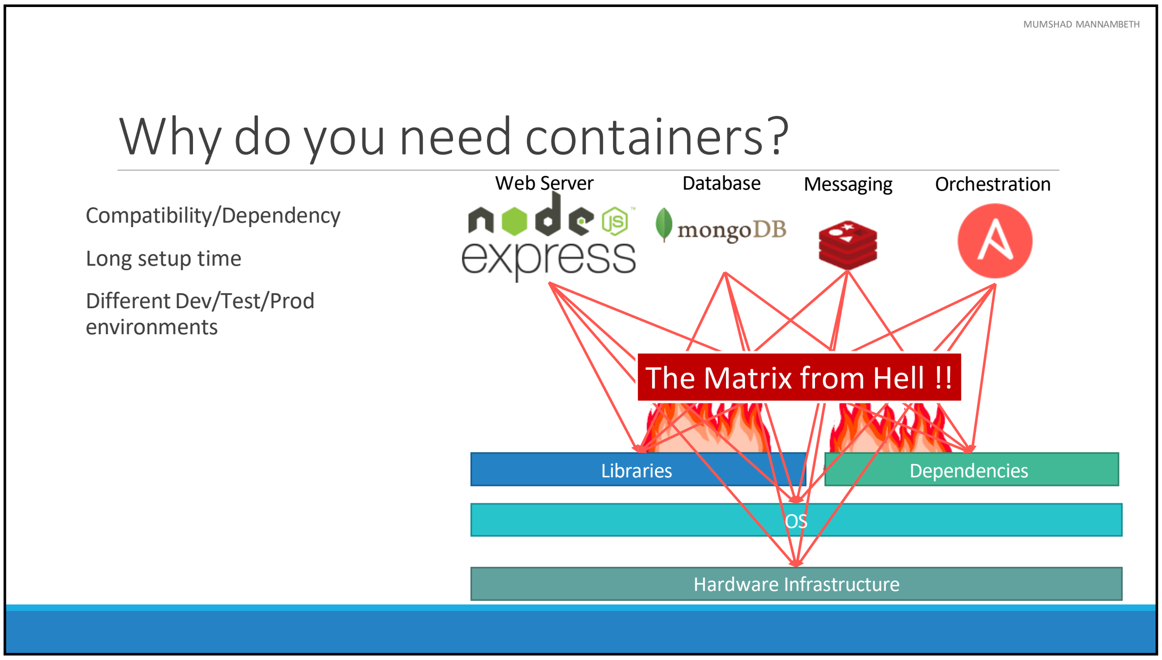
To understand Kubernetes, we must first understand two things – Container and Orchestration. Once we get familiarized with both of these terms we would be in a position to understand what kubernetes is capable of. We will start looking at each of these next.



Containers

mumshad mannambeth

We are now going to look at what containers are, specifically we will look at the most popular container technology out there – Docker. If you are familiar with Docker already, feel free to skip this lecture and move over to the next.


MUMSHAD MANNAMBETH

Why do you need containers?

- Compatibility/Dependency
- Long setup time
- Different Dev/Test/Prod environments

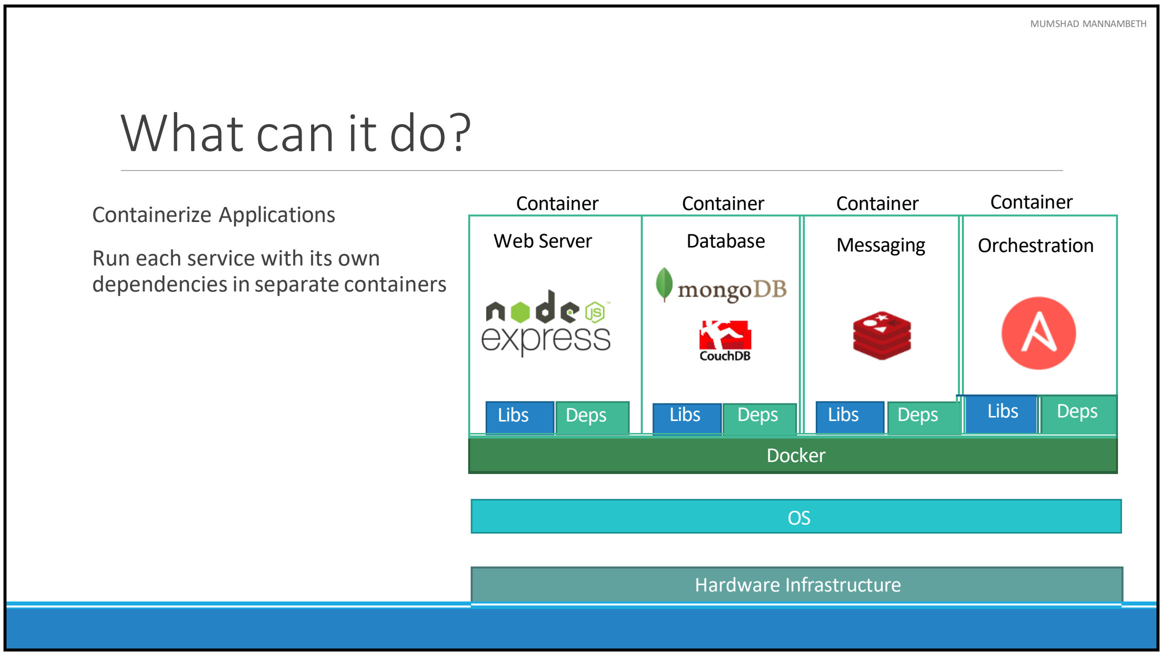
Let me start by sharing how I got introduced to Docker. In one of my previous projects, I had this requirement to setup an end-to-end stack including various different technologies like a Web Server using NodeJS and a database such as MongoDB/CouchDB, messaging system like Redis and an orchestration tool like Ansible. We had a lot of issues developing this application with all these different components. First, their compatibility with the underlying OS. We had to ensure that all these different services were compatible with the version of the OS we were planning to use. There have been times when certain version of these services were not compatible with the OS, and we have had to go back and look for another OS that was compatible with all of these different services.

Secondly, we had to check the compatibility between these services and the libraries and dependencies on the OS. We have had issues were one service requires one version of a dependent library whereas another service required another version.

The architecture of our application changed over time, we have had to upgrade to newer versions of these components, or change the database etc and everytime something changed we had 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.

Next, everytime we had a new developer on board, we found it really difficult to setup a new environment. The new developers had to follow a large set of instructions and run 100s of commands to finally setup their environments. They had to 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 gurantee 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.



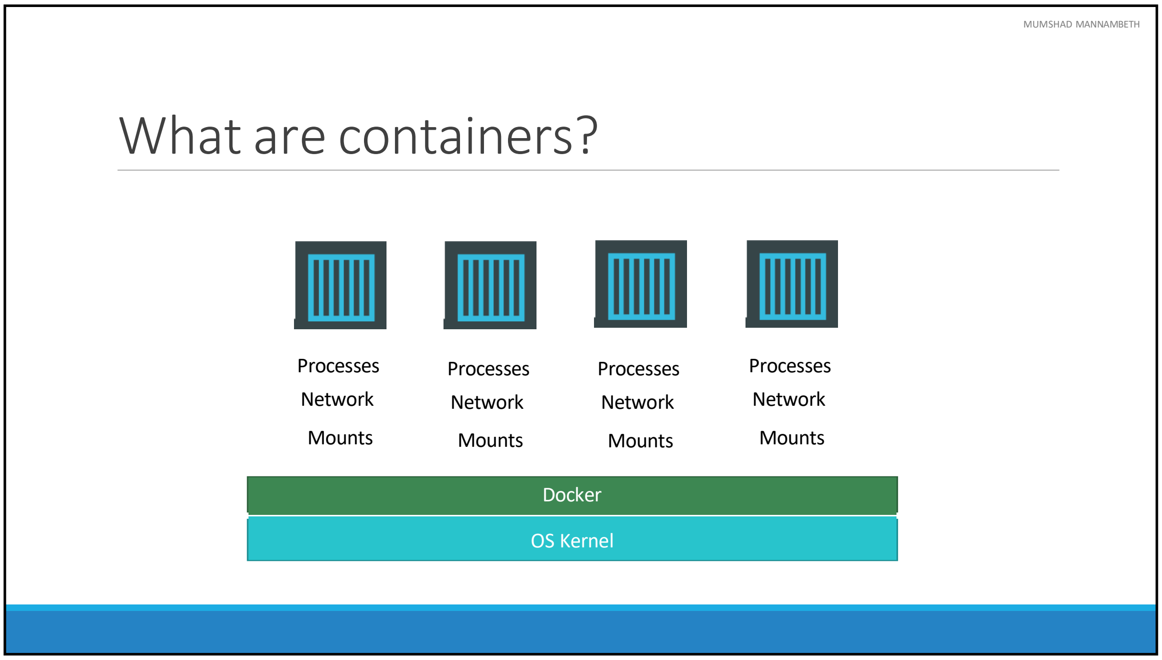
MUMSHAD MANNAMBETH

What can it do?

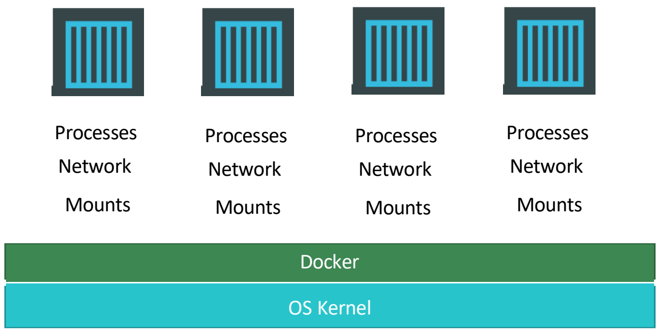
Containerize Applications

Run each service with its own dependencies in separate containers

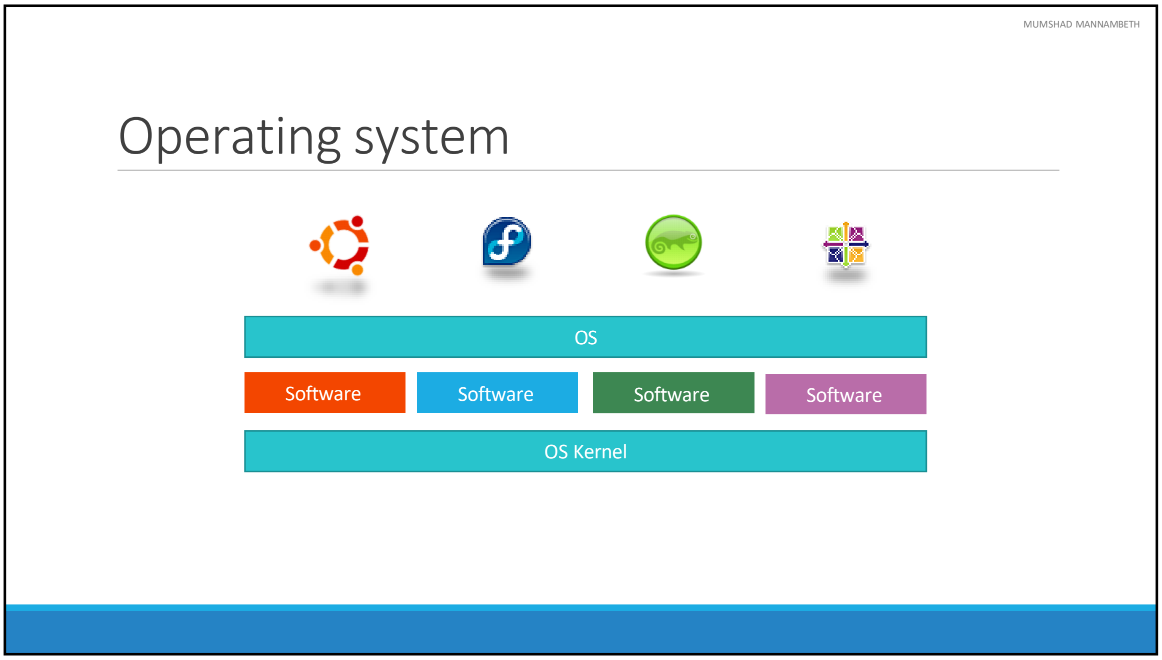
So I needed something that could help us with the compatibility issue. And something that will allow us to modify or change these components without affecting the other components and even modify the underlying operating systems as required. And that search landed me on Docker. With Docker I was able 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.


MUMSHAD MANNAMBETH

What are containers?



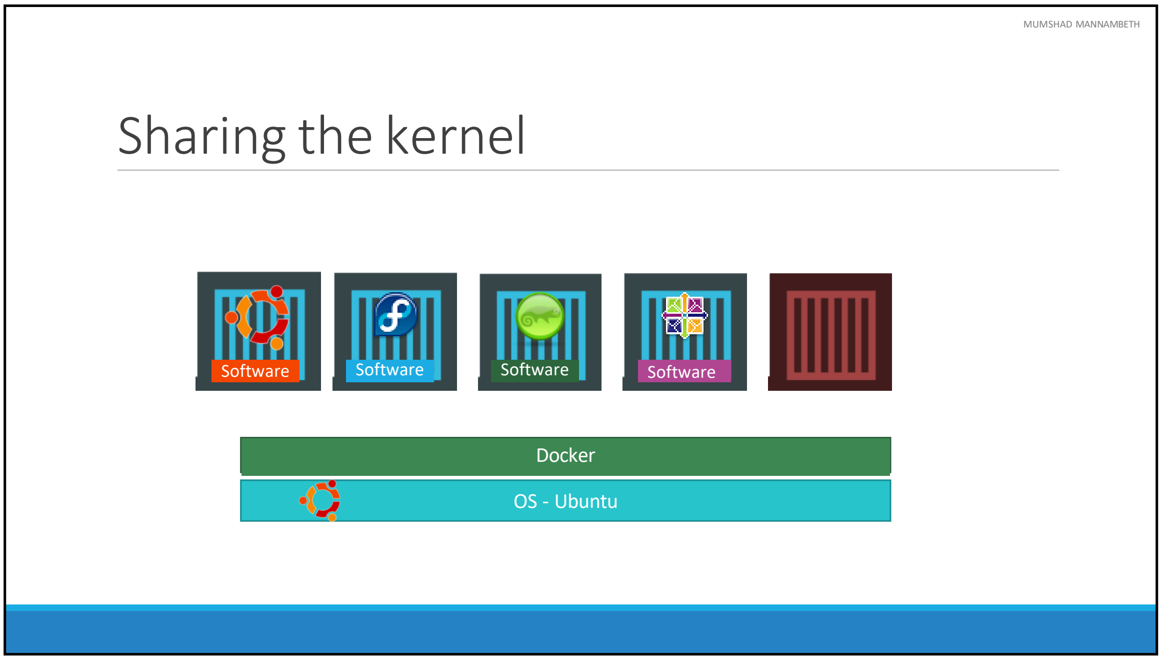
So what are containers? Containers are completely isolated environments, as in they can have their own processes or services, their own network interfaces, their own mounts, just like Virtual machines, except that they all share the same OS kernel. We will look at what that means in a bit. But its also 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 were Docker offers a high-level tool with several powerful functionalities making it really easy for end users like us.



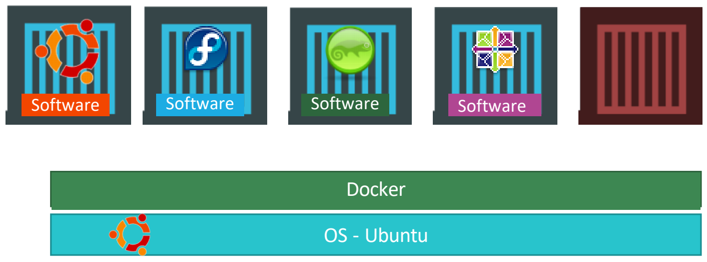
MUMSHAD MANNAMBETH

Operating system

To understand how Docker works let us revisit some basics concepts of Operating Systems first. If you look at operating systems like Ubuntu, Fedora, Suse or Centos – they all consist of two things. An OS Kernel and a set of software. The OS Kernel is responsible for interacting with the underlying hardware. While the OS kernel remains the same– which is Linux in this case, it's the software above it that make these Operating Systems different. This software may consist of a different User Interface, drivers, compilers, File managers, developer tools etc. SO you have a common Linux Kernel shared across all Oses and some custom softwares that differentiate Operating systems from each other.


MUMSHAD MANNAMBETH

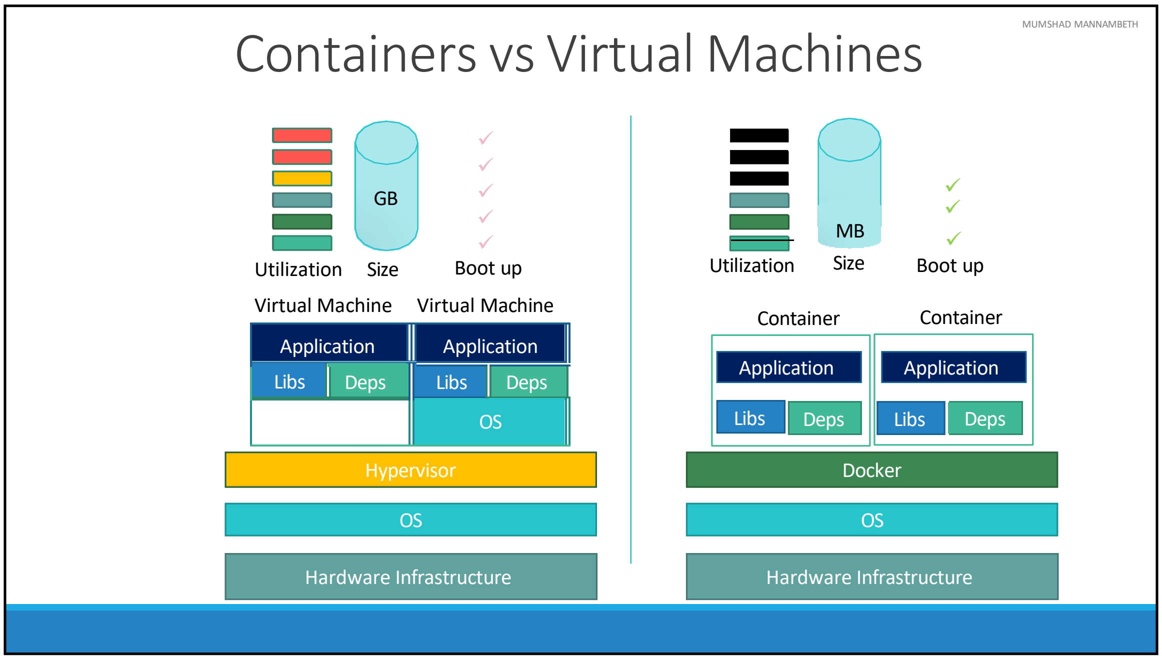
Sharing the kernel



We said earlier that Docker containers share the underlying kernel. What does that actually mean – sharing the kernel? Let's say we have a system with an Ubuntu OS with Docker installed on it. Docker can run any flavor of OS on top of it as long as they are all based on the same kernel – in this case Linux. If the underlying OS is Ubuntu, docker can run a container based on another distribution like debian, fedora, suse or centos. Each docker container only has the additional software, that we just talked about in the previous slide, that makes these operating systems different and docker utilizes the underlying kernel of the Docker host which works with all Oses above.

So what is an OS that do not share the same kernel as these? Windows ! And so you wont be able to run a windows based container on a Docker host with Linux OS on it. For that you would require docker on a windows server.

You might ask isn't that a disadvantage then? Not being able to run another kernel on the OS? The answer is No! Because unlike hypervisors, Docker is not meant to virtualize and run different Operating systems and kernels on the same hardware. The main purpose of Docker is to containerize applications and to ship them and run them.



So that brings us to the differences between virtual machines and containers. Something that we tend to do, especially those from a Virtualization.

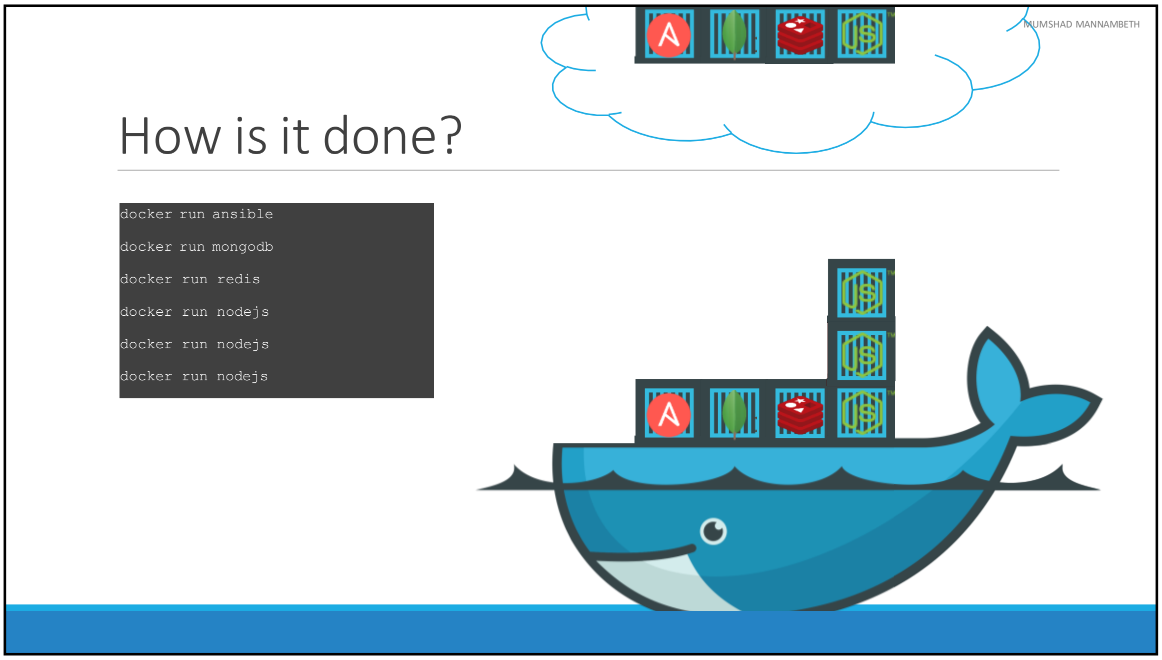
As you can see on the right, in case of Docker, we have the underlying hardware infrastructure, then the OS, and Docker installed on the OS. Docker then manages the containers that run with libraries and dependencies alone. In case of a Virtual Machine, we have the OS on the underlying hardware, then the Hypervisor like a ESX or virtualization of some kind and then the virtual machines. As you can see each virtual machine has its own OS inside it, then the dependencies and then the application.

This overhead causes higher utilization of underlying resources as there are multiple virtual operating systems and kernel running. The virtual machines also consume higher disk space as each VM is heavy and is usually in Giga Bytes in size, wereas docker containers are lightweight and are usually in Mega Bytes in size.

This allows docker containers to boot up faster, usually in a matter of seconds whereas VMs we know takes minutes to boot up as it needs to bootup the entire OS.

It is also important to note that, Docker has less isolation as more resources are shared between containers like the kernel etc. Whereas VMs have complete isolation from each other. Since VMs don’t rely on the underlying OS or kernel, you can run different types of OS such as linux based or windows based on the same hypervisor.

So these are some differences between the two.



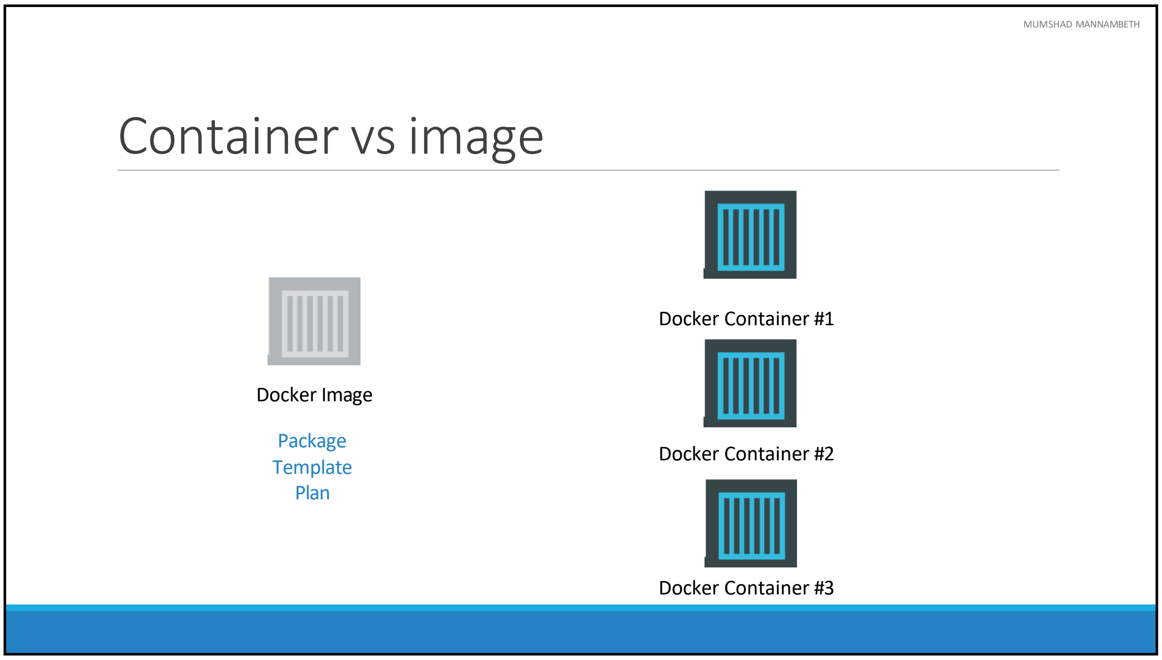
How is it done?

```
docker run ansible
docker run mongodb
docker run redis
docker run nodejs
docker run nodejs
docker run nodejs
```

MUMSHAD MANNAMBETH

SO how is it done? There are a lot of containerized versions of applications readily available as of today. So most organizations have their products containerized and available in a public docker registry called dockerhub/or docker store already. For example you can find images of most common operating systems, databases and other services and tools. Once you identify the images you need and you install Docker on your host..

bringing up an application stack, is as easy as running a docker run command with the name of the image. In this case running a docker run ansible command will run an instance of ansible on the docker host. Similarly run an instance of mongodb, redis and nodejs using the docker run command. And then when you run nodejs just point to the location of the code repository on the host. If we need to run multiple instances of the web service, simply add as many instances as you need, and configure a load balancer of some kind in the front. In case one of the instances was to fail, simply destroy that instance and launch a new instance. There are other solutions available for handling such cases, that we will look at later during this course.



MUMSHAD MANNAMBETH

Container vs image

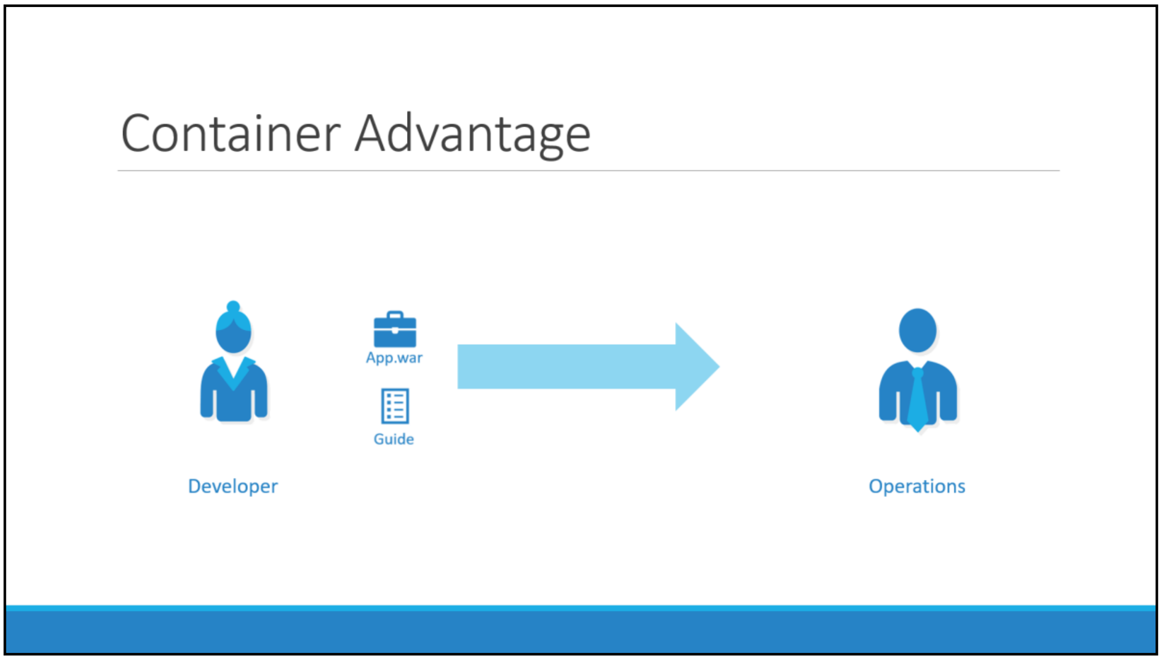
Docker Image
Package
Template
Plan
Docker Container #1
Docker Container #2
Docker Container #3

We have been talking about images and containers. Let’s understand the difference between the two.

An image is a package or a template, just like a VM template that you might have worked with in the virtualization world. 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

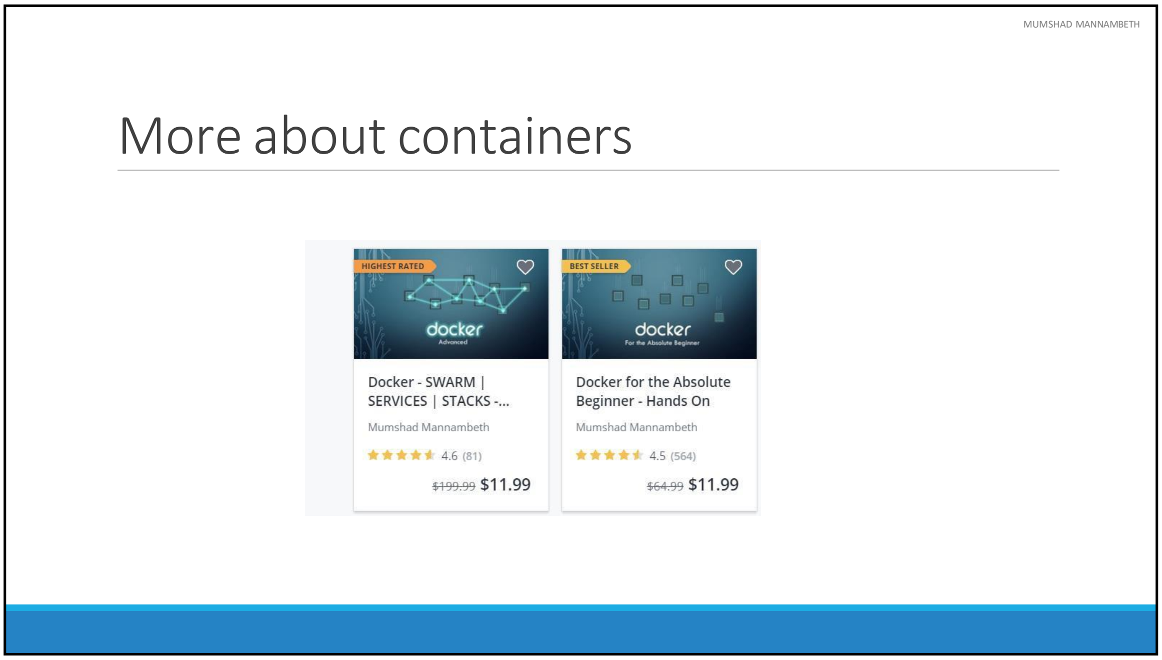
As we have seen before a lot of products have been dockerized already. In case you cannot find what you are looking for you could create an image yourself and push it to the Docker hub repository making it available for public.



Container Advantage

If you look at it, traditionally developers developed applications. Then they 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.



More about containers

Docker - SWARM | SERVICES | STACKS -...

Mumshad Mannambeth

4.6 (81)

~~$199.99~~ $11.99

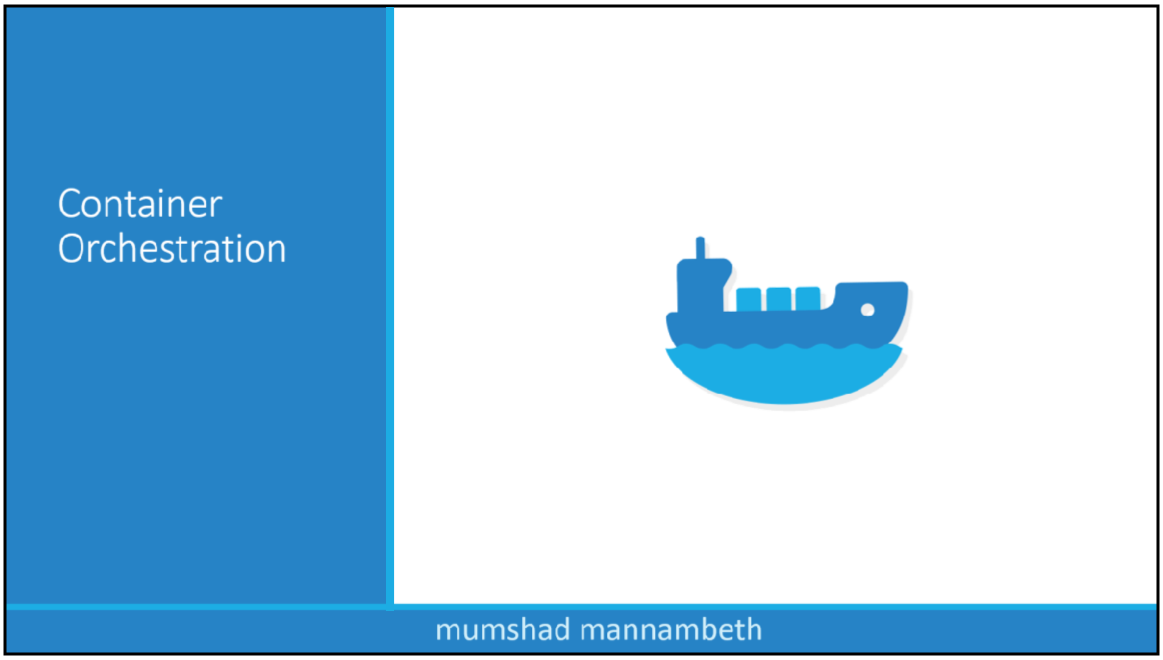
Docker for the Absolute Beginner - Hands On

Mumshad Mannambeth

4.5 (564)

~~$64.99~~ $11.99

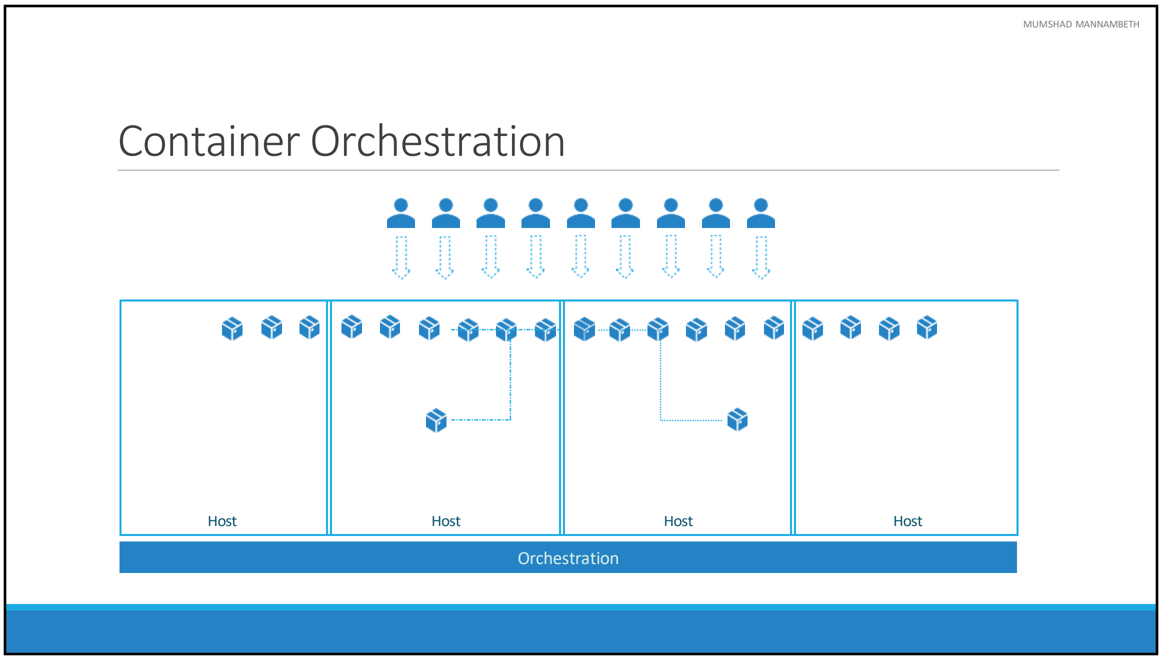
To learn more about containers, checkout my other courses - Docker for the Absolute Beginners and Docker Swarm were you can learn and practice docker commands and create docker files. That's the end of this lecture on Containers and Docker. See you in the next lecture.



Container Orchestration

mumshad mannambeth

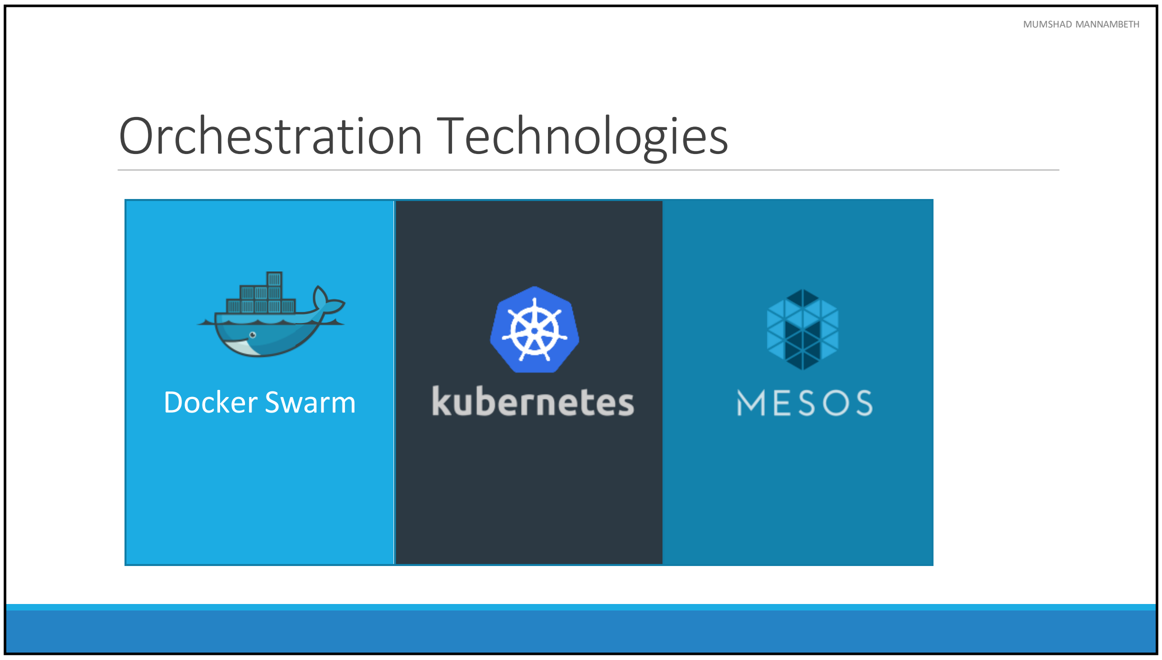
In this lecture we will talk about Container Orchestration.


MUMSHAD MANNAMBETH

Container Orchestration

So we learned about containers and we now have our application packaged into a docker container. But what next? How do you run it in production? What if your application relies on other containers such as database or messaging services or other backend services? What if the number of users increase and you need to scale your application? You would also like to scale down when the load decreases.

To enable these functionalities you 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.

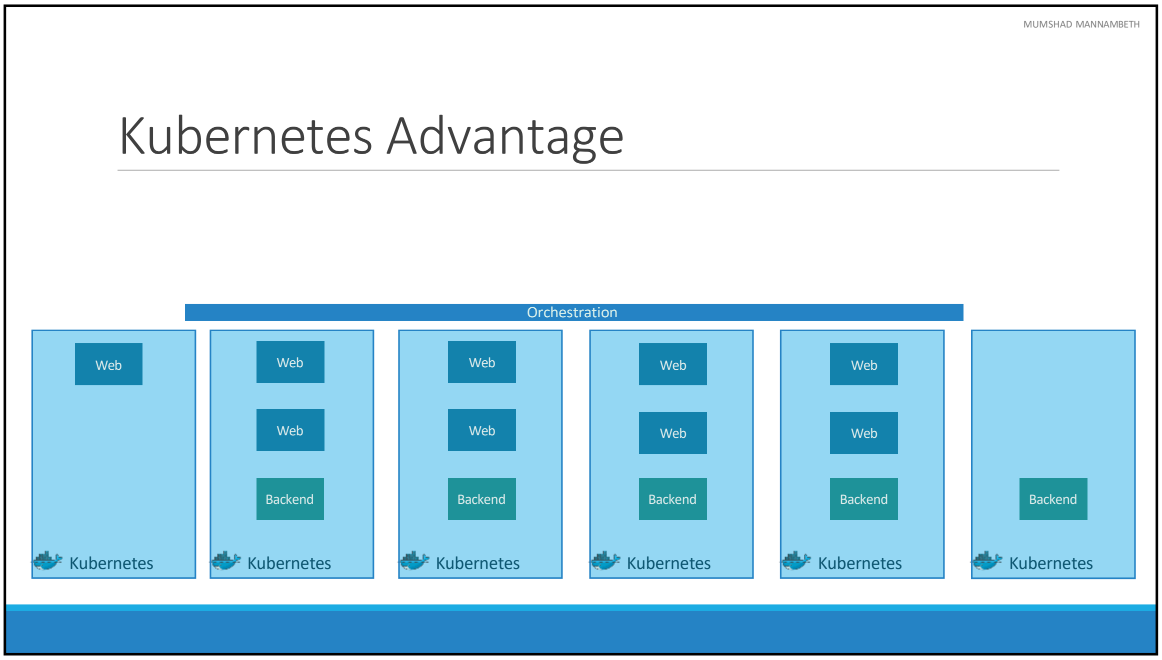


MUMSHAD MANNAMBETH

Orchestration Technologies

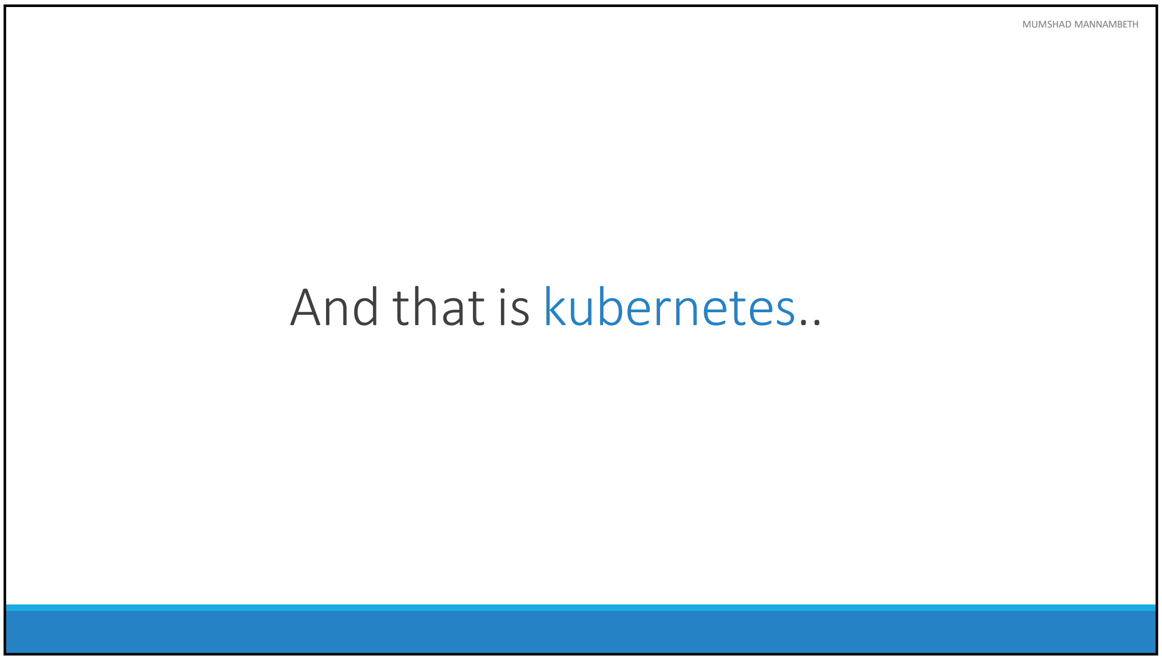
Docker Swarm
kubernetes
MESOS

Kubernetes is thus a container orchestration technology. There are multiple such technologies available today – Docker has its own tool called Docker Swarm. Kubernetes from Google and Mesos from Apache. While Docker Swarm is really easy to setup and get started, it lacks some of the advanced autoscaling features required for complex applications. Mesos on the other hand is quite difficult to setup and get started, but supports many advanced features. 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.


MUMSHAD MANNAMBETH

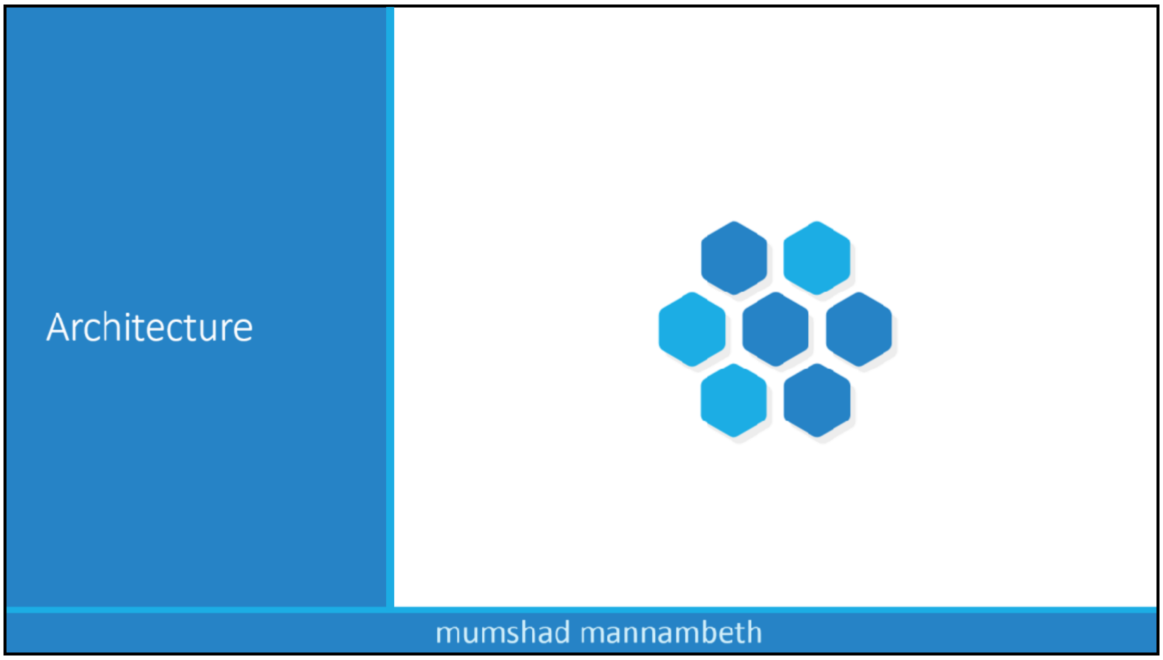
Kubernetes Advantage

There are various advantages of container orchestration. Your application is now highly available as hardware failures do not bring your application down because you have multiple instances of your 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 have the ability to 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 of these easily with a set of declarative object configuration files.


MUMSHAD MANNAMBETH

And that is kubernetes..

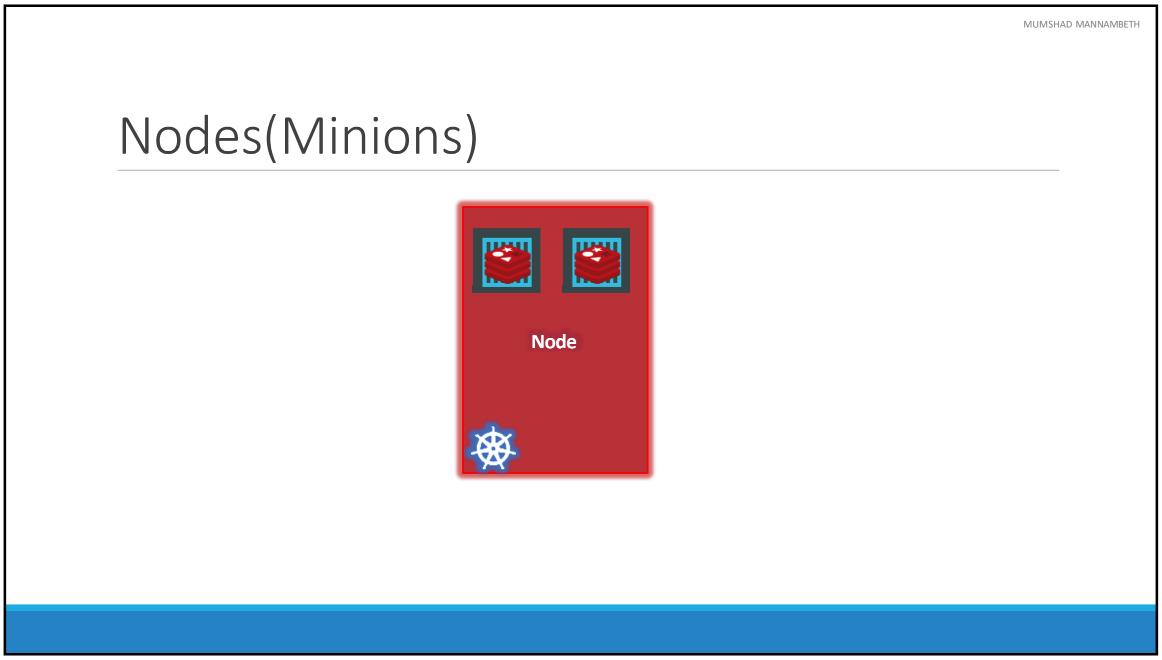
And THAT IS Kubernetes. It is a container Orchestration technology used to orchestrate the deployment and management of 100s and 1000s of containers in a clustered environment. Don’t worry if you didn’t get all of what was just said, in the upcoming lectures we will take a deeper look at the architecture and various concepts surrounding kubernetes. That is all for this lecture, thank you for listening and I will see you in the next lecture.



Architecture

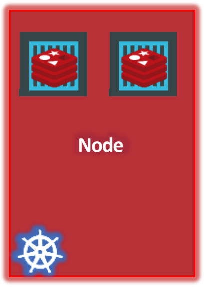
mumshad mannambeth

Before we head into setting up a kubernetes cluster, it is important to understand some of the basic concepts. This is to make sense of the terms that we will come across while setting up a kubernetes cluster.



MUMSHAD MANNAMBETH

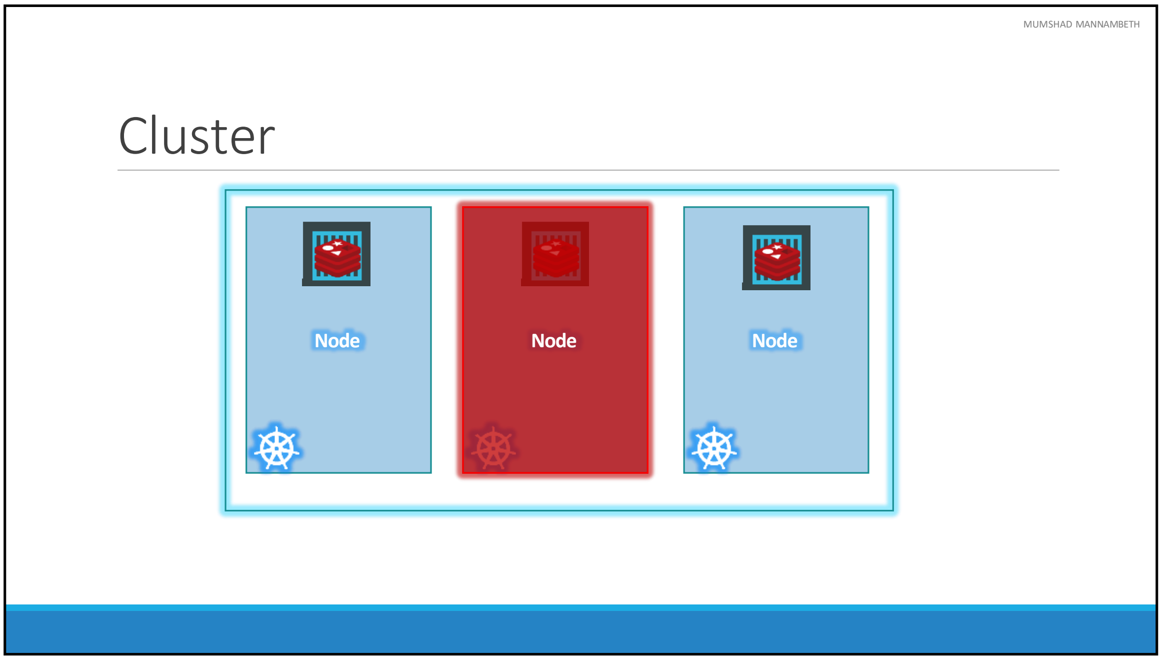
Nodes(Minions)



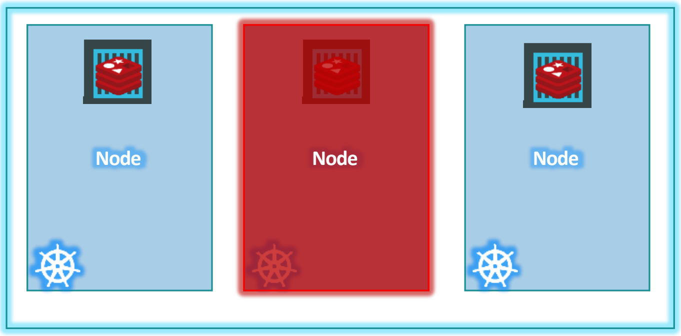
Let us start with Nodes. A node is a machine – physical or virtual – on which kubernetes is installed. A node is a worker machine and this is were containers will be launched by kubernetes.

It was also known as Minions in the past. So you might here these terms used inter changeably.

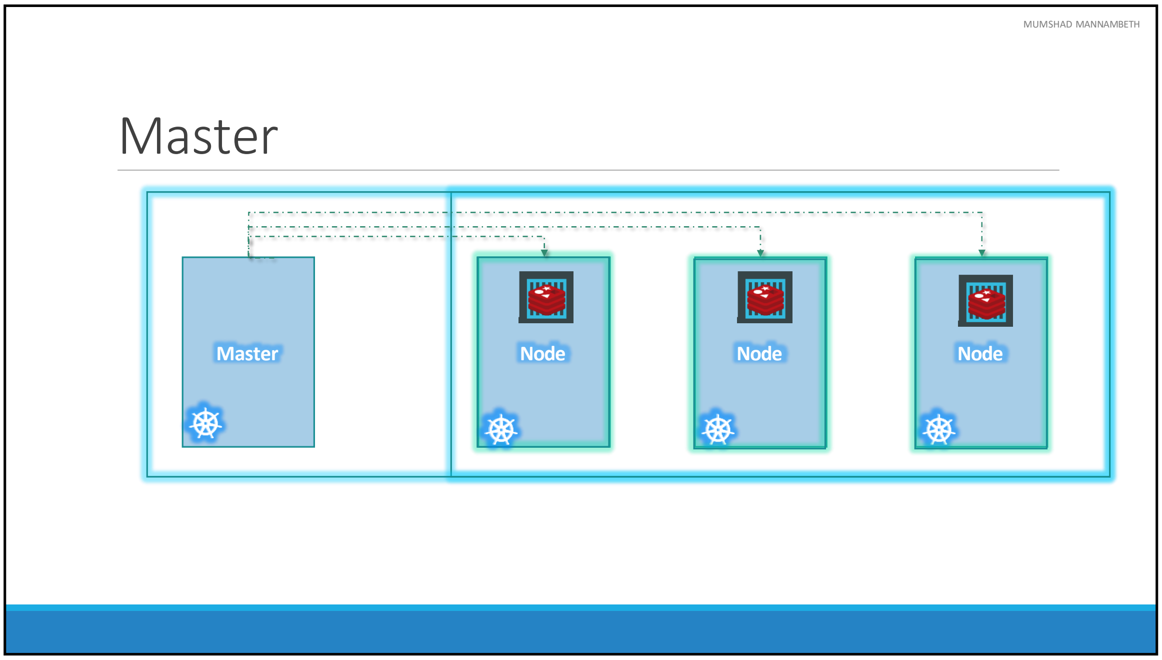
But what if the node on which our application is running fails? Well, obviously our application goes down. So you need to have more than one nodes.



Cluster

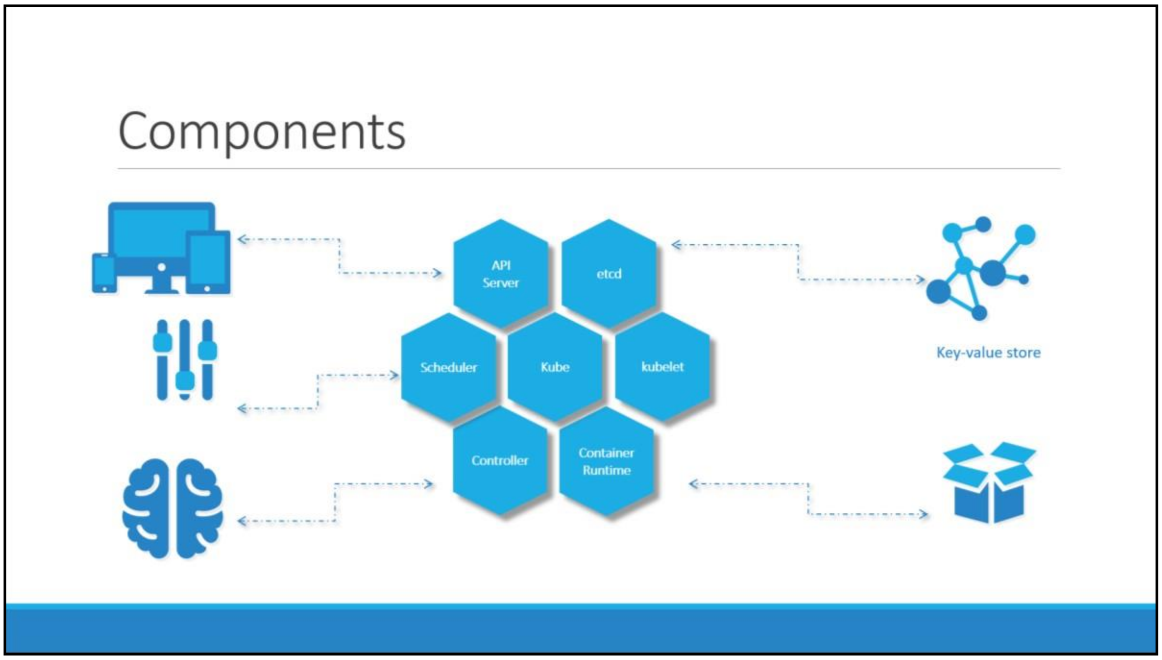


A cluster is a set of nodes grouped together. This way even if one node fails you have your application still accessible from the other nodes. Moreover having multiple nodes helps in sharing load as well.


MUMSHAD MANNAMBETH

Master

Now we have a cluster, but who is responsible for managing the cluster? Were is the information about the members of the cluster stored? How are the nodes monitored? When a node fails how do you move the workload of the failed node to another worker node? That's were the Master comes in. 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

When you install Kubernetes on a System, you are actually installing the following components. An API Server. An ETCD service. A kubelet service. A Container Runtime, Controllers and Schedulers.

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.

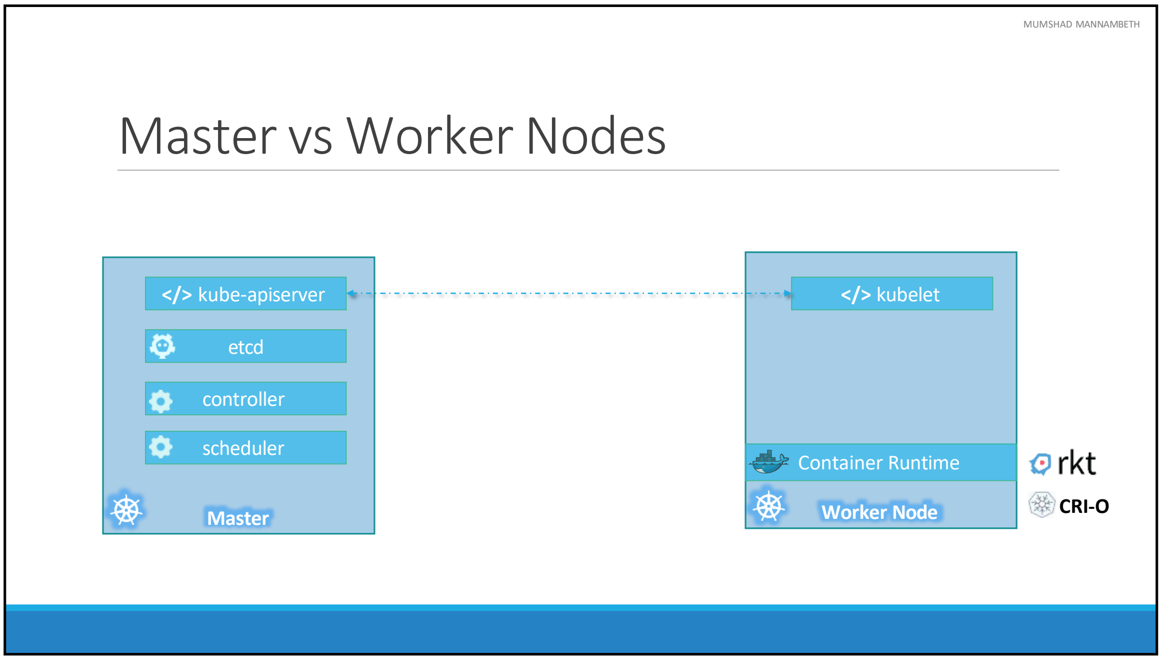
Next is the ETCD key store. ETCD is a distributed reliable key-value store used by kubernetes to store all data used to manage the cluster. Think of it this way, when you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster in a distributed manner. ETCD is responsible for implementing locks within the cluster to ensure there are no conflicts between the Masters.

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

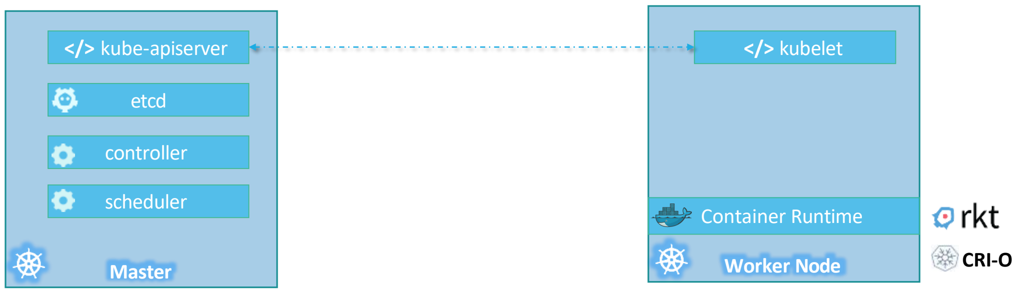
The controllers are the brain behind orchestration. They are responsible for noticing and responding when nodes, containers or endpoints goes down. The controllers makes decisions to bring up new containers in such cases.

The container runtime is the underlying software that is used to run containers. In our case it happens to be Docker.

And finally 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.


MUMSHAD MANNAMBETH

Master vs Worker Nodes



So far we saw two types of servers – Master and Worker and a set of components that make up Kubernetes. 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?

The worker node (or minion) as it is also known, is were the containers are hosted. For example Docker containers, and to run docker containers on a system, we need a container runtime installed. And that's were the container runtime falls. In this case it happens to be Docker. This doesn't HAVE to be docker, there are other container runtime alternatives available such as Rocket or CRIO. But throughout this course we are going to use Docker as our container runtime.

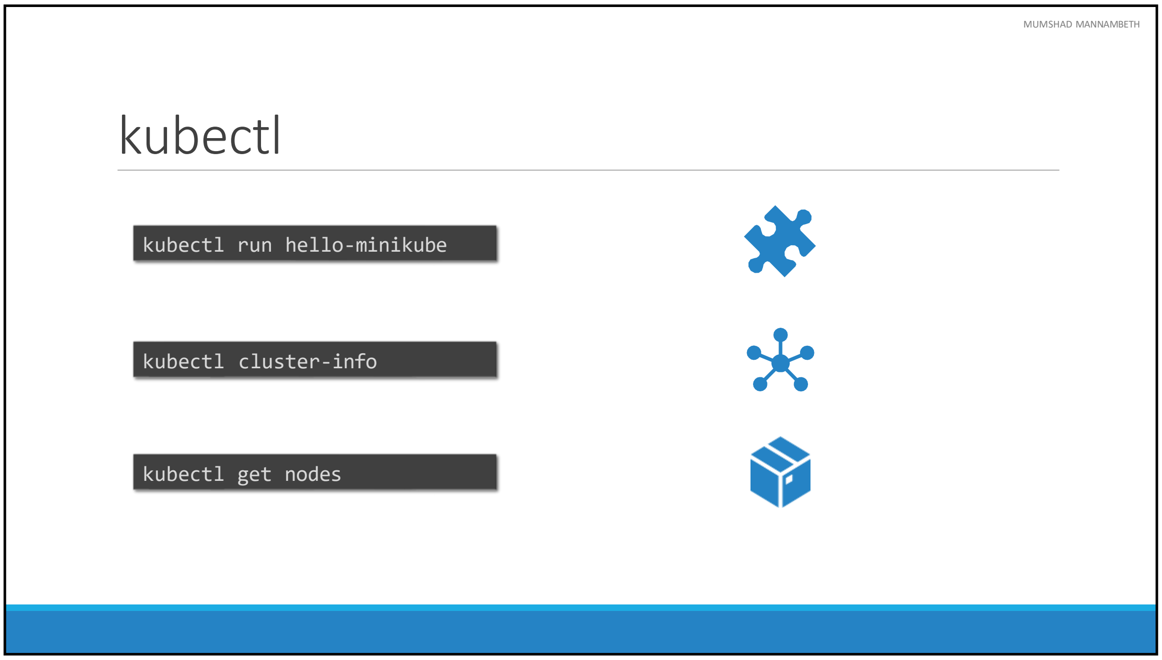
The master server has the kube-apiserver and that is what makes it a master.

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 is based on the popular etcd framework as we just discussed.

The master also has the controller manager and the scheduler.

There are other components as well, but we will stop there for now. The reason we went through this is to understand what components constitute the master and worker nodes. This will help us install and configure the right components on different systems when we setup our infrastructure.



MUMSHAD MANNAMBETH

kubectl

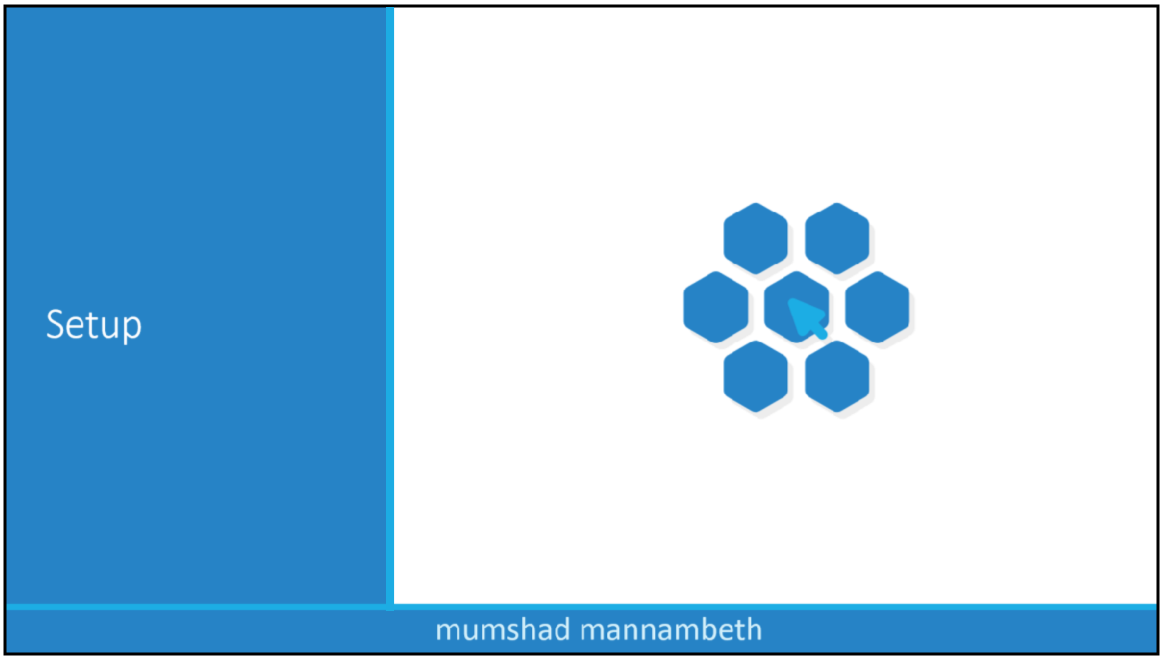
```
kubectl run hello-minikube
```

```
kubectl cluster-info
```

```
kubectl get nodes
```

And finally, we also need to learn a little bit about ONE of the command line utilities known as the kube command line tool or kubectl or kube control as it is also called. The kube control tool is used to deploy and manage applications on a kubernetes cluster, to get cluster information, get the status of nodes in the cluster and many other things.

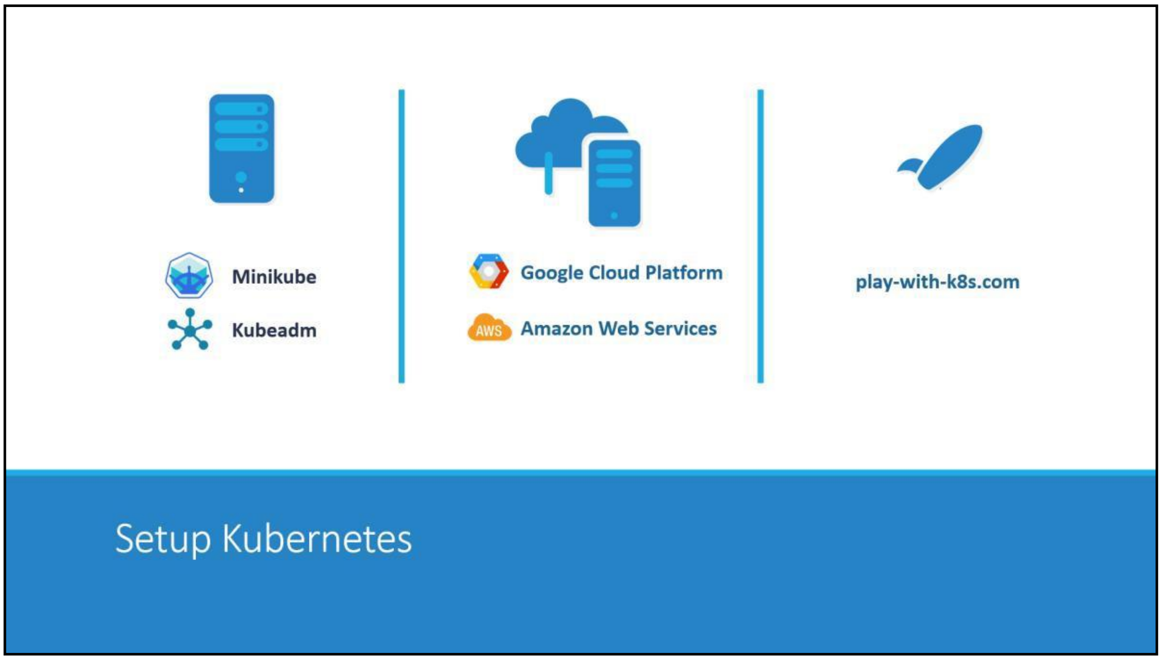
The kubectl run command is used to deploy an application on the cluster. The kubectl cluster-info command is used to view information about the cluster and the kubectl get pod command is used to list all the nodes part of the cluster. That’s all we need to know for now and we will keep learning more commands throughout this course. We will explore more commands with kubectl when we learn the associated concepts. For now just remember the run, cluster-info and get nodes commands and that will help us get through the first few labs.



Setup

mumshad mannambeth

In this lecture we will look at the various options available in building a Kubernetes cluster from scratch.


Minikube
Kubeadm
Google Cloud Platform
Amazon Web Services
play-with-k8s.com

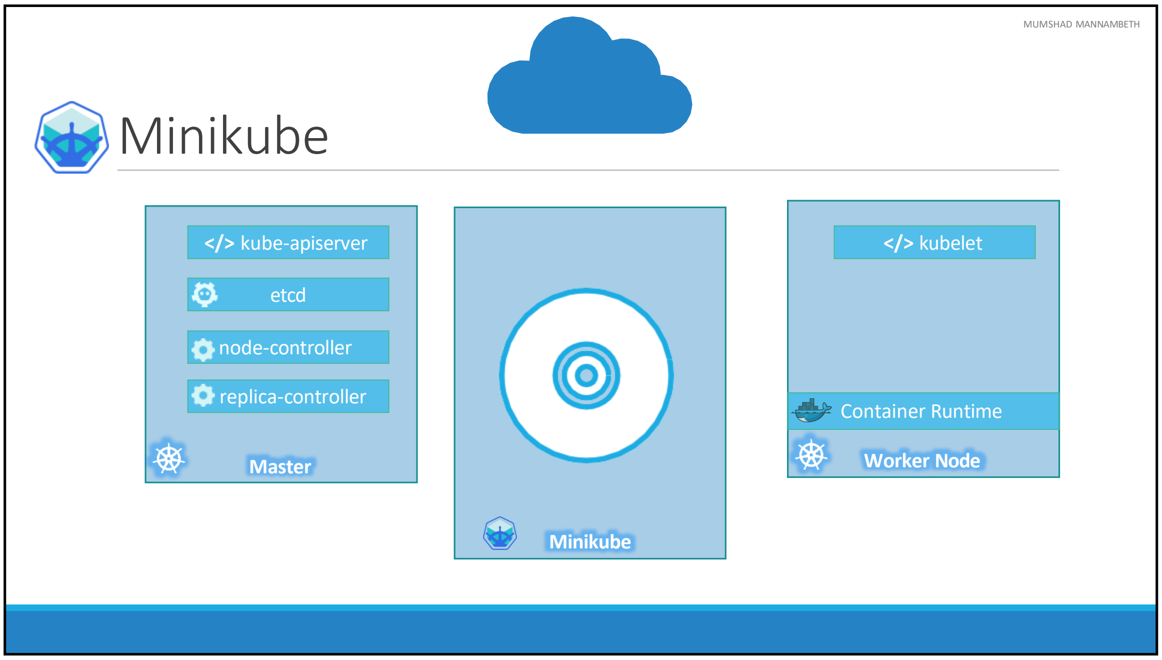
Setup Kubernetes

There are lots of ways to setup Kuberentes. We can setup it up ourselves locally on our laptops or virtual machines using solutions like Minikube and Kubeadmin. Minikube is a tool used to setup a single instance of Kubernetes in an All-in-one setup and kubeadmin is a tool used to configure kubernetes in a multi-node setup. We will look more into that in a bit.

There are also hosted solutions available for setting up kubernetes in a cloud environment such as GCP and AWS. We will also have some demos around those.

And finally if you don't have the resources or if you don't want to go through the hassle of setting it all up yourself, and you simply want to get your hands on a kubernetes cluster instantly to play with, checkout play-with-k8s.com . I also have a demo on this.

So feel free to chose the one that is right for you. You need not go through all the demos, pick the ones that best suite your needs based on your time and resources.



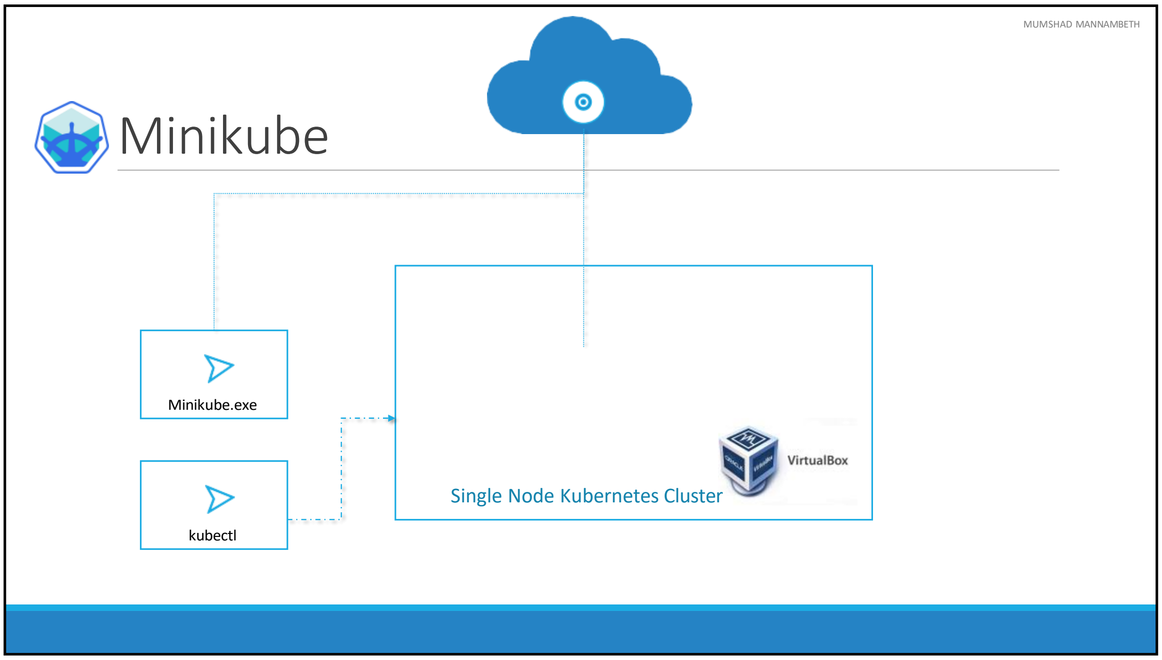
MUMSHAD MANNAMBETH

Minikube

We will start with Minikube which is the easiest way to get started with Kubernetes on a local system. If Minikube is not of interest to you, now would be a good time to skip this lecture. Before we head into the demo it's good to understand how it works. Earlier we talked about the different components of Kubernetes that make up a Master and worker nodes such as the api server, etcd key value store, controllers and scheduler on the master and kubelets and container runtime on the worker nodes. It would take a lot of time and effort to setup and install all of these various components on different systems individually by ourlselves.

Minikube bundles all of 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.

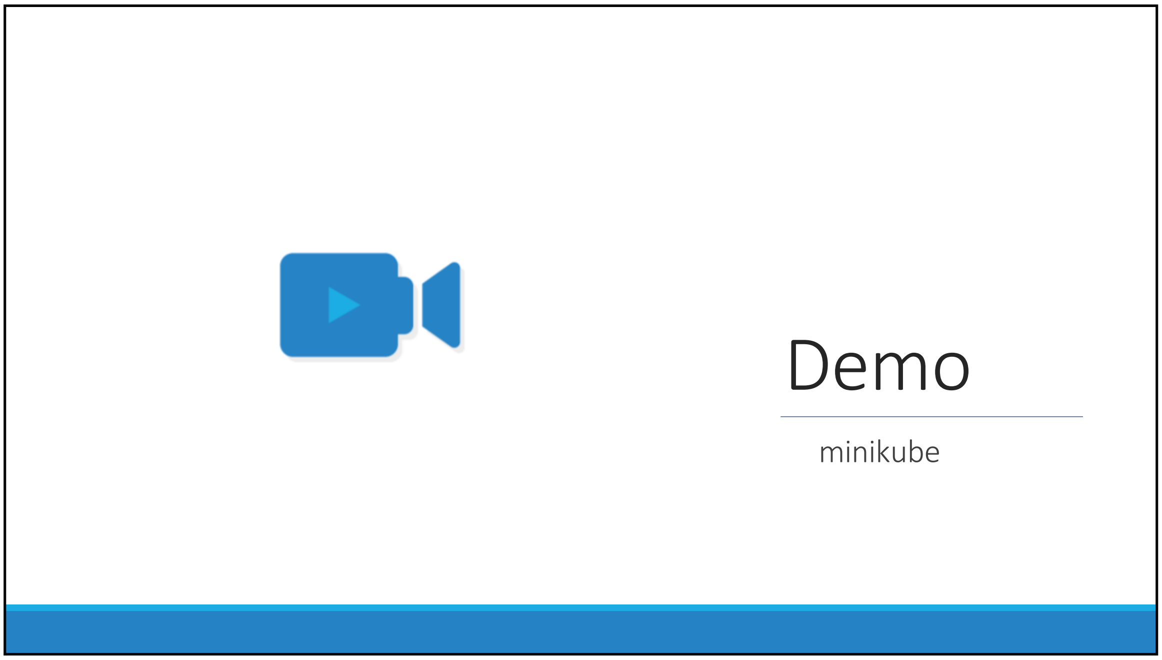

MUMSHAD MANNAMBETH

Minikube

Now you don’t HAVE to download it yourself. 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 Vmware fusion. So you must have a Hypervisor installed on your system. For windows you could use Virtualbox or Hyper-V and for Linux use Virtualbox or KVM.

And finally to interact with the kubernetes cluster, you must have the kubectl kubernetes command line tool also installed on your machine. So you need 3 things to get this working, you must have a hypervisor installed, kubectl installed and minikube executable installed on your system.

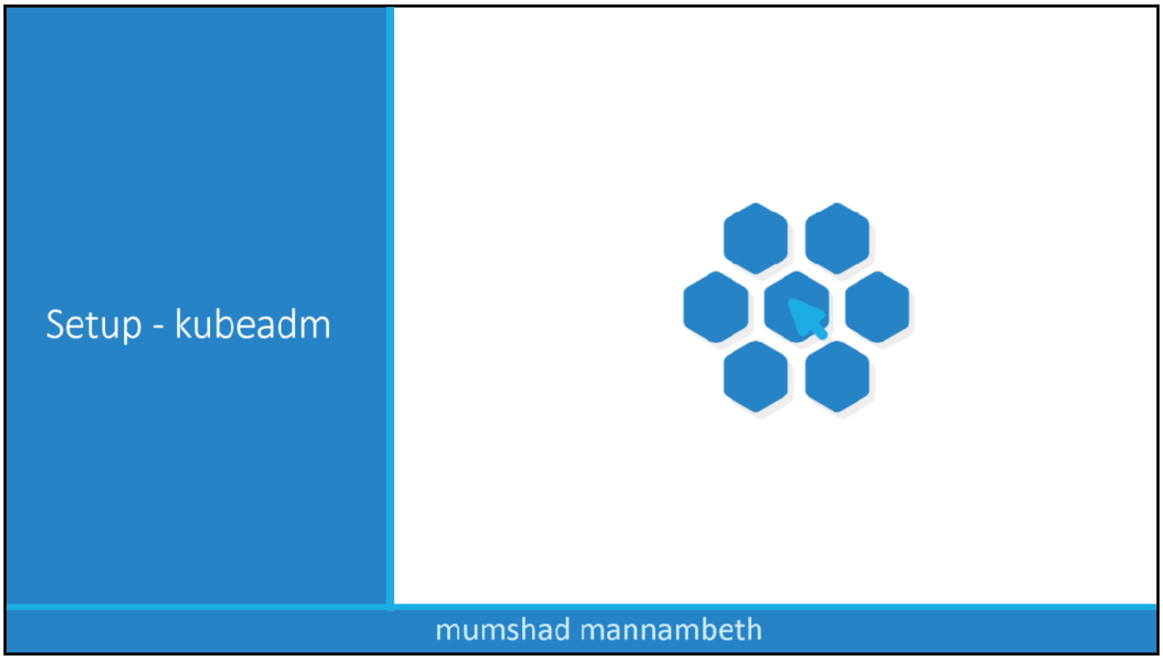
So let’s get into the demo



Demo

minikube

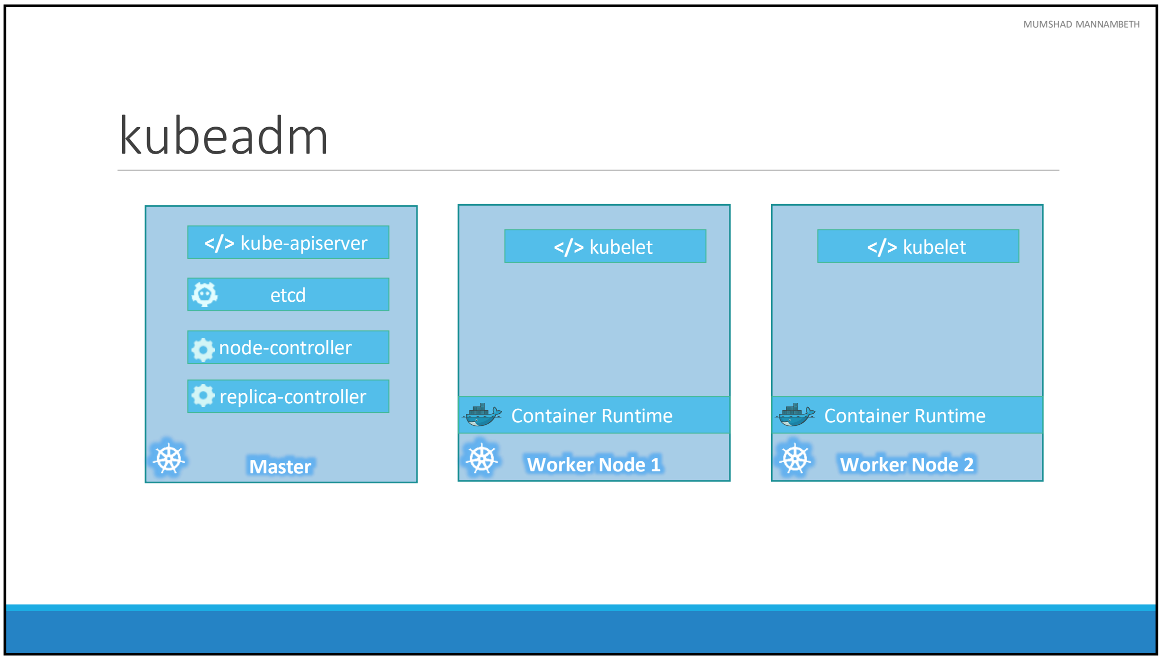
That’s it for this lecture, lets head over to the demo and see this in action.



Setup - kubeadm

mumshad mannambeth

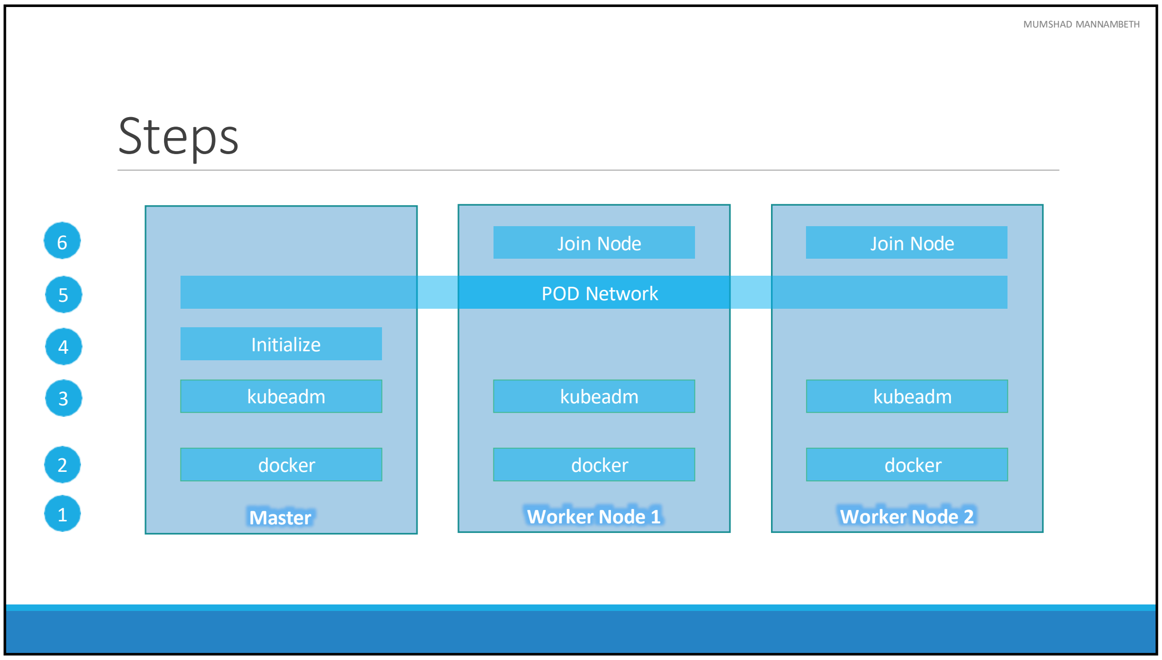
Hello and welcome to this lecture on setting up Kubernetes with kubeadm. In this lecture we will look at the kubeadm tool which can be used to bootstrap a kubernetes cluster.



MUMSHAD MANNAMBETH

kubeadm

With the minikube utility you could only setup a single node kubernetes cluster. The kubeadmin tool helps us setup a multi node cluster with master and workers on separate machines. Installing all of these various components individually on different nodes and modifying the configuration files to make it work is a tedious task. Kubeadmin tool helps us in doing all of that very easily.


MUMSHAD MANNAMBETH

Steps

Let’s go through the steps – First, you must have multiple systems or virtual machines created for configuring a cluster. We will see how to setup up your laptop to do just that if you are not familiar with it. Once the systems are created, designate one as master and others as worker nodes.

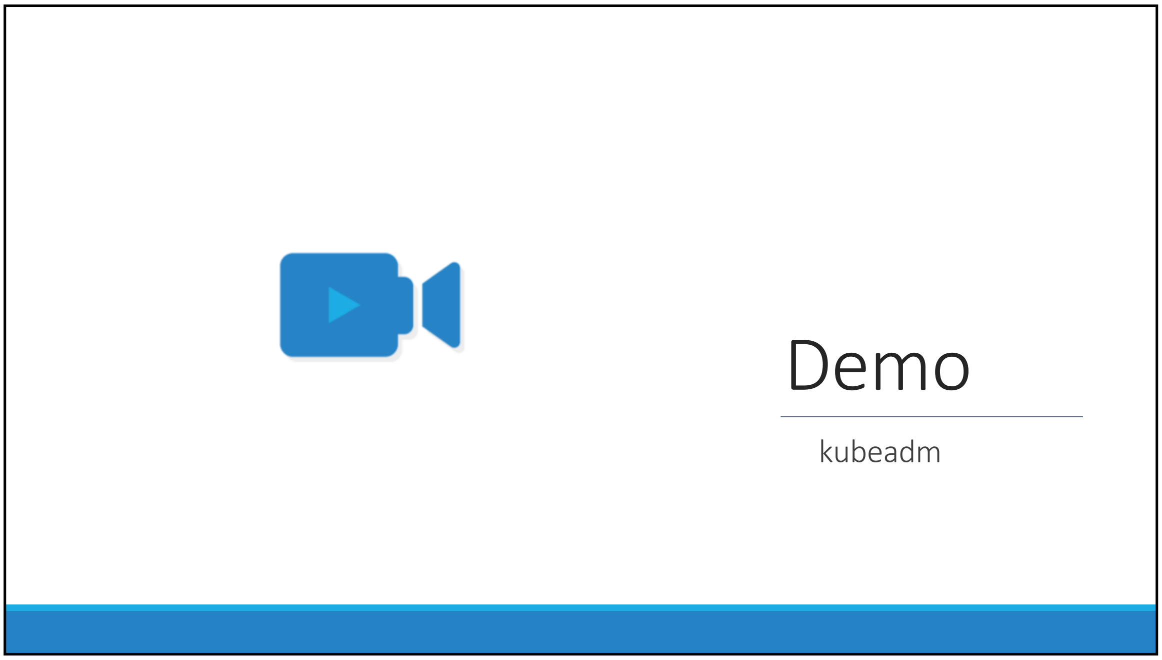
The next step is to install a container runtime on the hosts. We will be using Docker, so we must install Docker on all the nodes.

The next step is to install kubeadmin tool on all the nodes. The kubeadmin tool helps us bootstrap the kubernetes solution by installing and configuring all the required components in the right nodes.

The third step is to initialize the Master server. During this process all the required components are installed and configured on the master server. That way we can start the cluster level configurations from the master server.

Once the master is initialized and before joining the worker nodes to the master, we must ensure that the network pre-requisites are met. A normal network connectivity between the systems is not SUFFICIENT for this. Kubernetes requires a special network between the master and worker nodes which is called as a POD network. We will learn more about this network in the networking section later in this course. For now we will simply follow the instructions available to get this installed and setup in our environment.

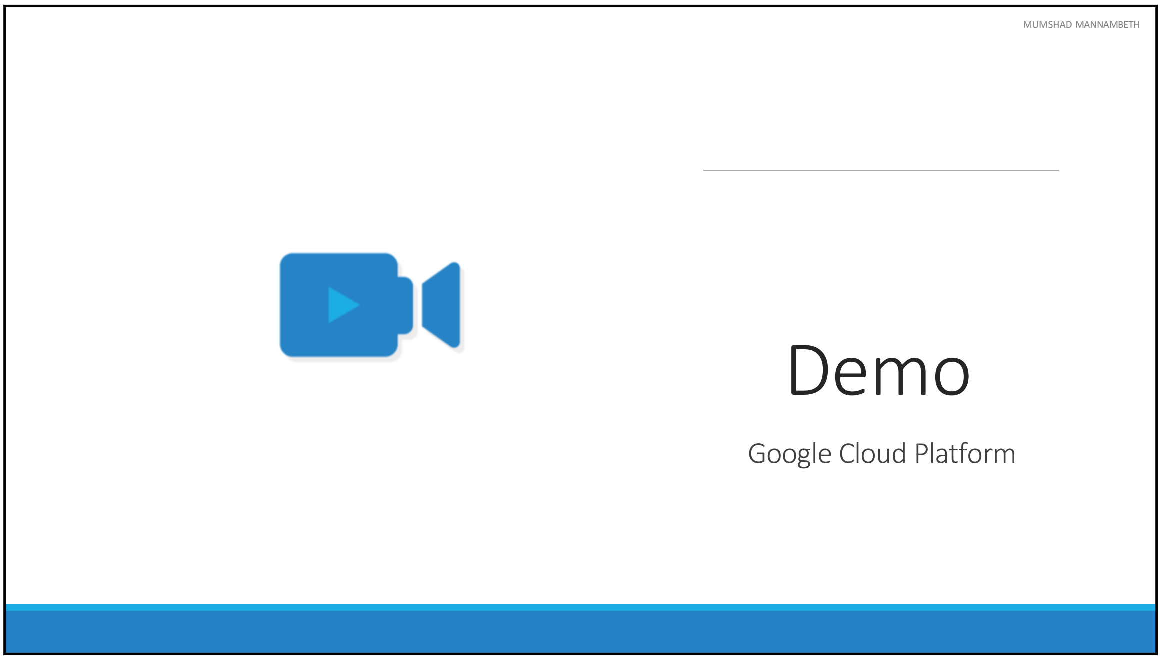
The last step is to join the worker nodes to the master node. We are then all set to launch our application in the kubernetes environment.



Demo

kubeadm

We will now see a demo of setting up kubernetes using the kubeadmin tool in our local environment.

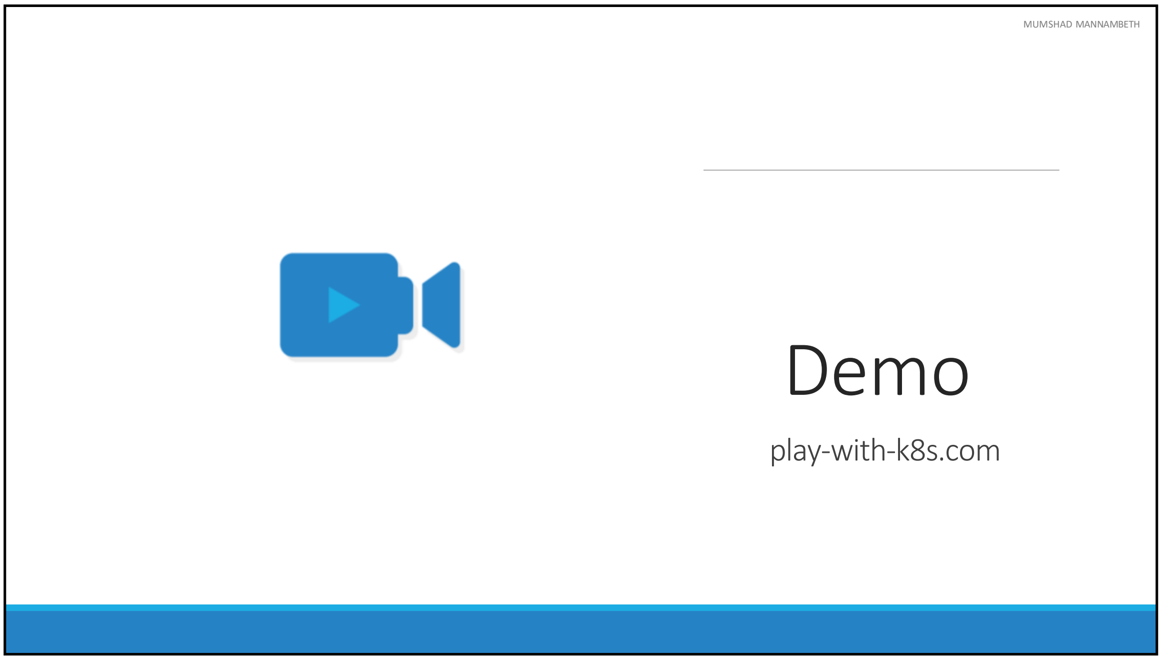


MUMSHAD MANNAMBETH

Demo

Google Cloud Platform

We will now see a demo of setting up kubernetes using the kubeadmin tool in our local environment.

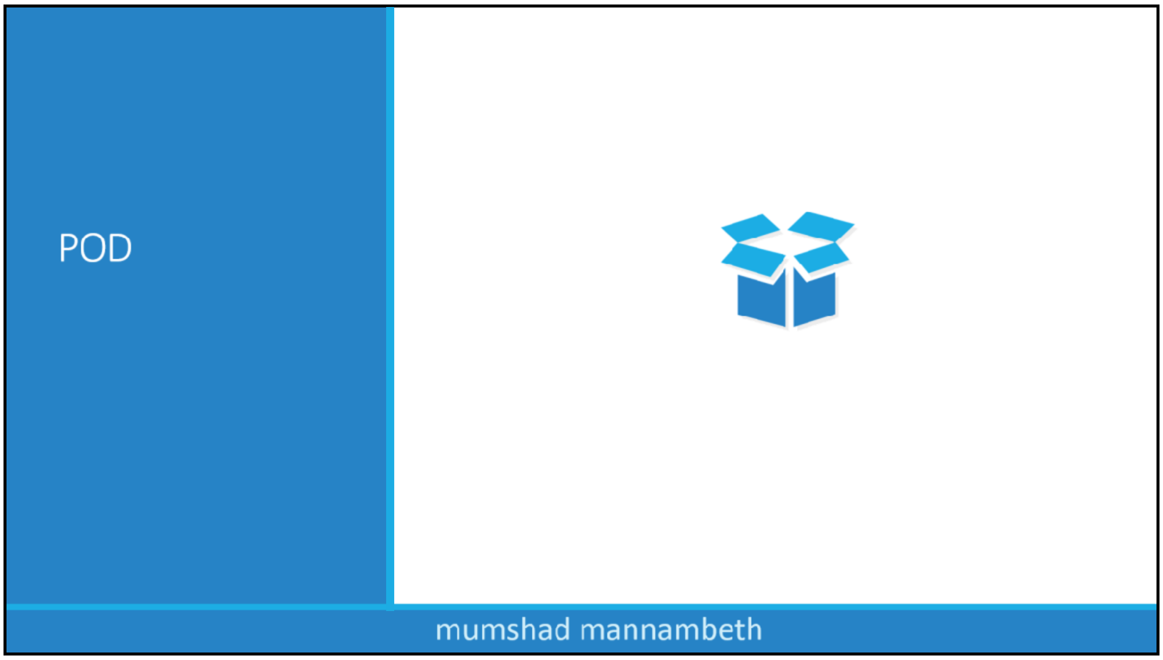


MUMSHAD MANNAMBETH

Demo

play-with-k8s.com

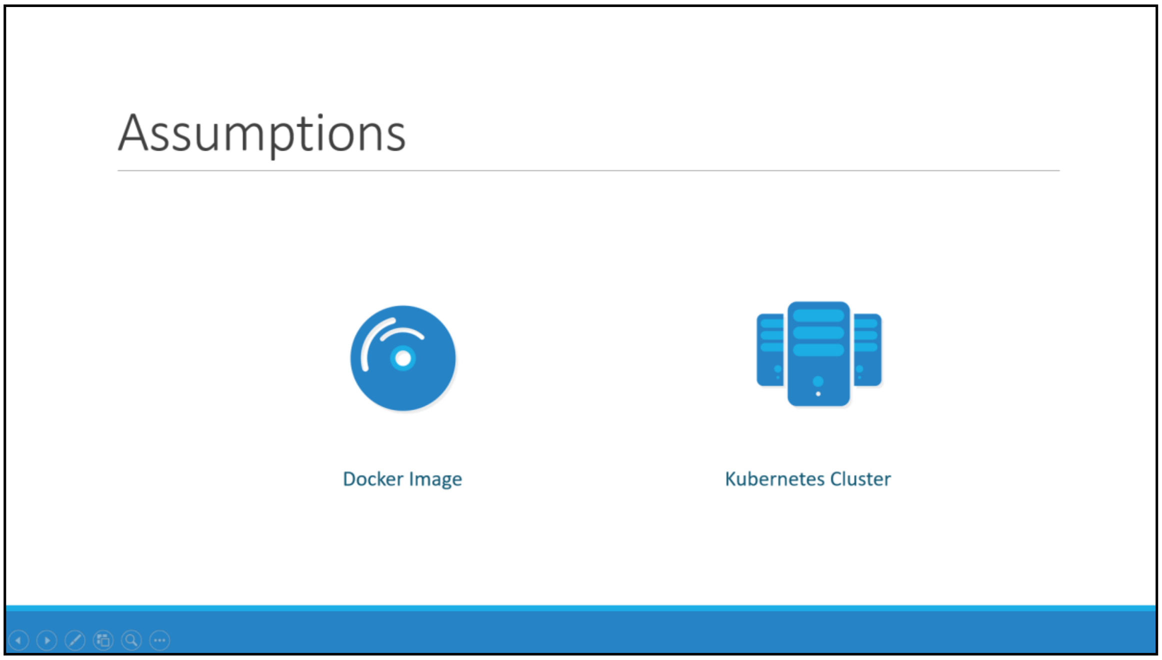
We will now see a demo of setting up kubernetes using the kubeadmin tool in our local environment.



POD

mumshad mannambeth

Hello and welcome to this lecture on Kubernetes PODs. In this lecture we will discuss about Kubernetes PODs.

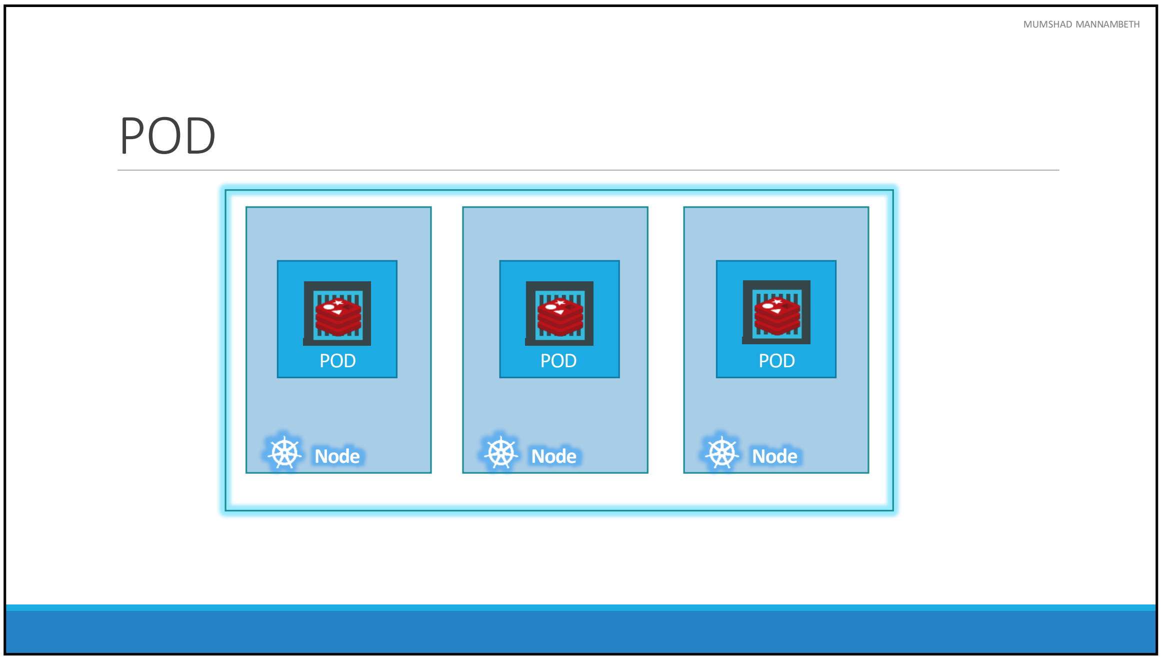


Assumptions

Docker Image

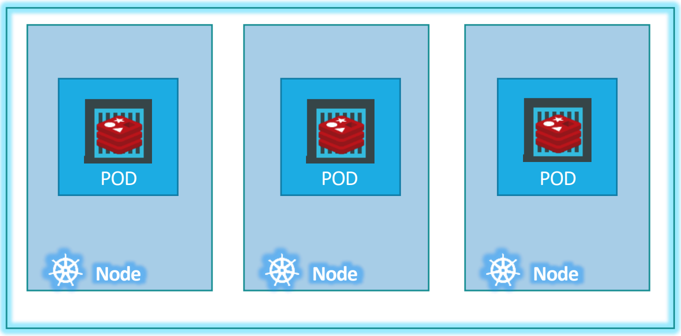
Kubernetes Cluster

Before we head into understanding PODs, we would like to assume that the following have been setup already. At this point, we assume that the application is already developed and built into Docker Images and it is availalble on a Docker repository like Docker hub, so 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.

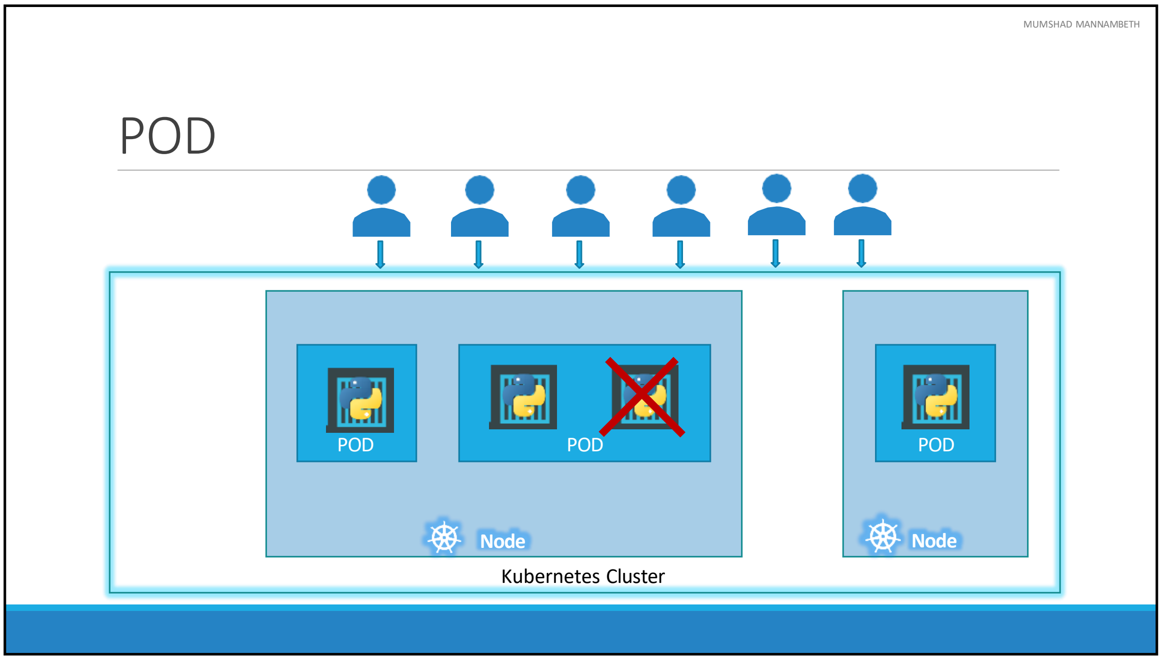


MUMSHAD MANNAMBETH

POD



As we discussed before, with kubernetes our ultimate 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 you can create in kubernetes.

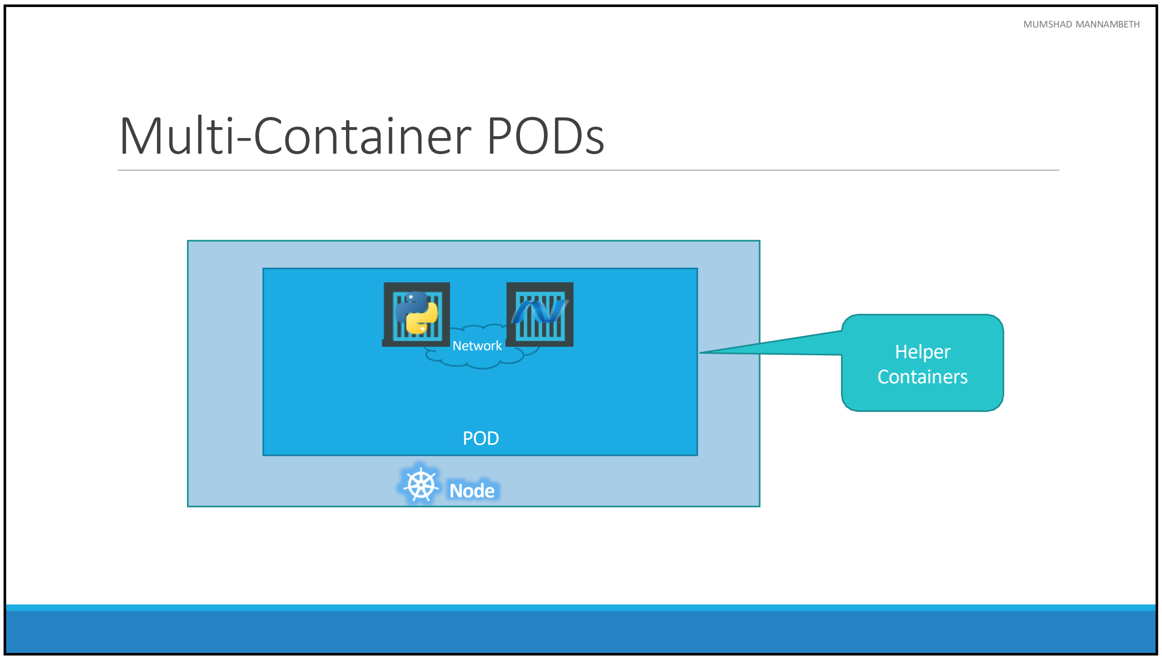


MUMSHAD MANNAMBETH

POD

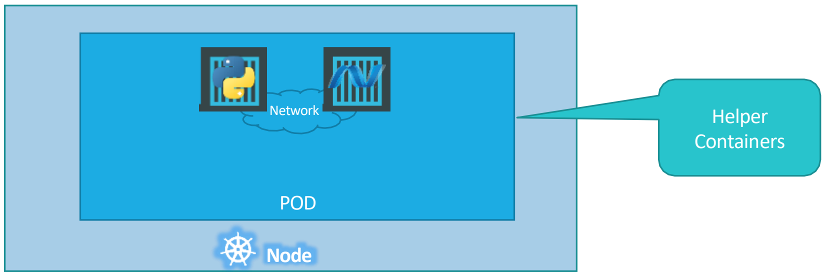
Here we see the simplest of simplest cases were you have a single node kubernetes cluster with a single instance of your application running in a single docker container encapsulated in a POD. What if the number of users accessing your application increase and you need to scale your application? You need to add additional instances of your web application to share the load. Now, were would you spin up additional instances? Do we bring up a new container instance within the same POD? No! We create a new POD altogether with a new instance of the same application. As you can see we now have two instances of our web application running on two separate PODs on the same kubernetes system or node.

What if the user base FURTHER increases and your current node has no sufficient capacity? Well THEN you 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, what I am trying to illustrate in this slide is that, PODs usually have a one-to-one relationship with containers running your application. To scale UP you create new PODs and to scale down you delete PODs. You do not add additional containers to an existing POD to scale your application. Also, if you are wondering how we implement all of this and how we achieve load balancing between containers etc, we will get into all of that in a later lecture. For now we are ONLY trying to understand the basic concepts.

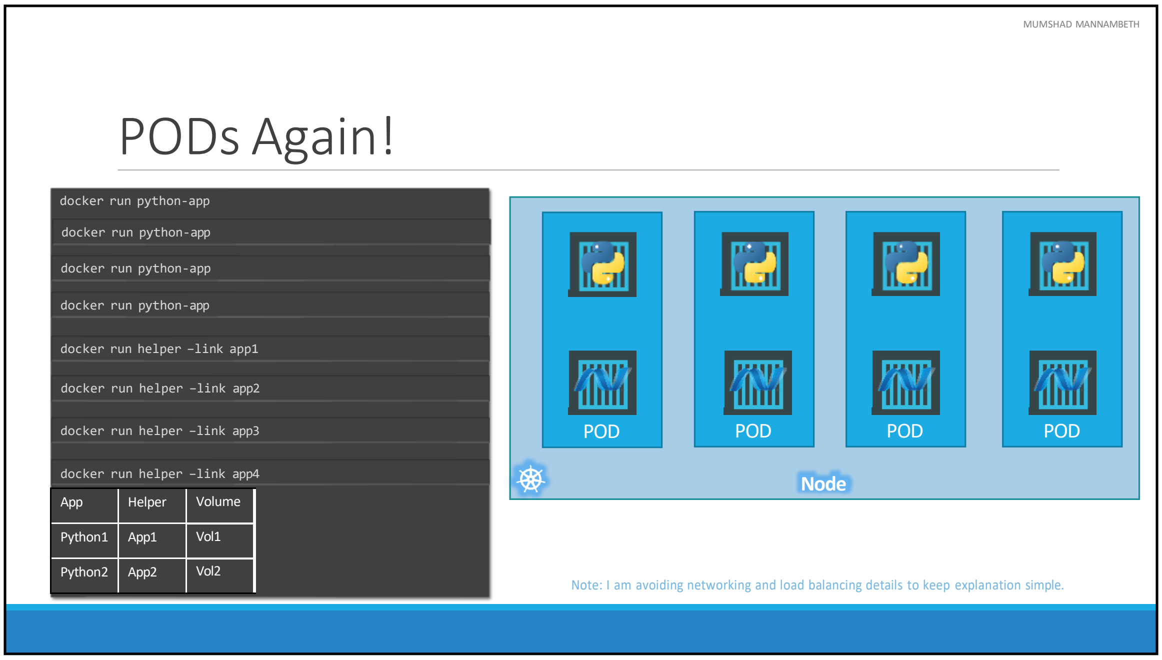


MUMSHAD MANNAMBETH

Multi-Container PODs



Now we just said that PODs usually have a one-to-one relationship with the containers, but, are we restricted to having a single container in a single POD? No! A single POD CAN have multiple containers, except for the fact that they are usually not multiple containers of the same kind. As we discussed in the previous slide, if our intention was to scale our application, then we would need to create additional PODs. But sometimes you might have a scenario were you have a helper container, that might be doing some kind of supporting task for our web application such as processing a user entered data, processing a file uploaded by the user etc. and you want these helper containers to live along side your application container. In that case, you CAN have both of these containers 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. Plus they can easily share the same storage space as well.



MUMSHAD MANNAMBETH

PODs Again!

```
docker run python-app
docker run python-app
docker run python-app
docker run python-app
docker run helper –link app1
docker run helper –link app2
docker run helper –link app3
docker run helper –link app4
```

App	Helper	Volume
Python1	App1	Vol1
Python2	App2	Vol2

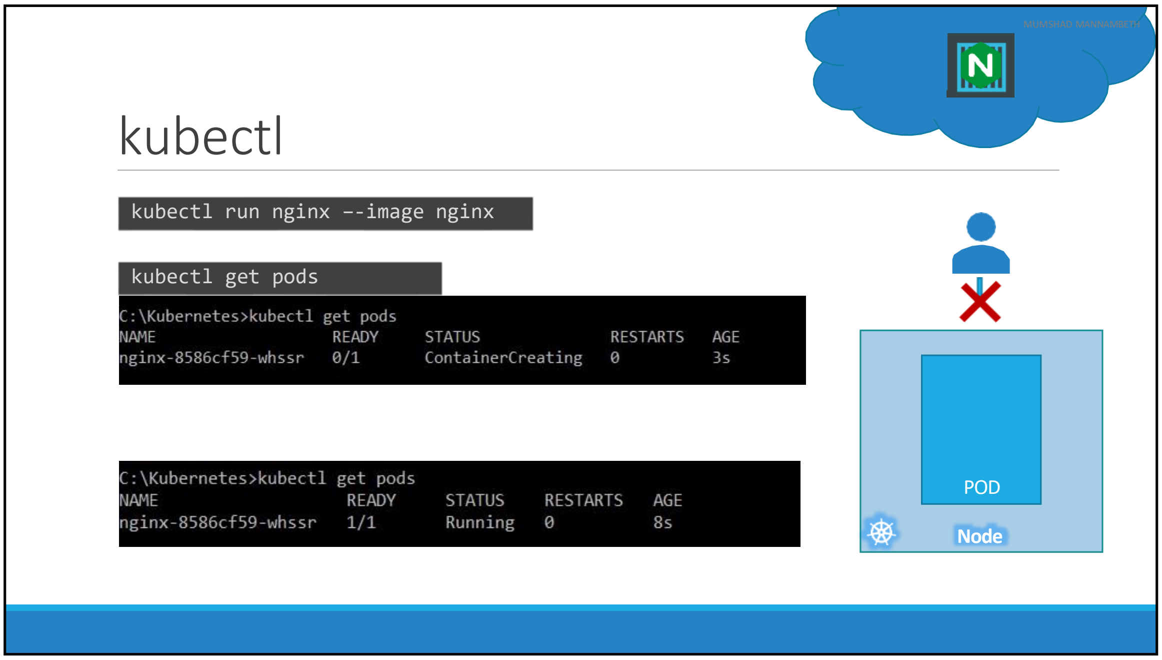
Note: I am avoiding networking and load balancing details to keep explanation simple.

If you still have doubts in this topic (I would understand if you did because I did the first time I learned these concepts), we could take another shot at understanding PODs from a different angle. Let’s, for a moment, keep kubernetes out of our discussion and talk about simple docker containers. Let’s assume we were developing a process or a script to deploy our application on a docker host. Then we would first simply deploy our application using a simple docker run python-app command and the application runs fine and our users are able to access it. When the load increases we deploy more instances of our application by running the docker run commands many more times. This works fine and we are all happy. Now, sometime in the future our application is further developed, undergoes architectural changes and grows and gets complex. We now have new helper containers that helps our web applications by processing or fetching data from elsewhere. These helper containers maintain a one-to-one relationship with our application container and thus, needs to communicate with the application containers directly and access data from those containers. For this we need to maintain a map of what app and helper containers are connected to each other, we would need to establish network connectivity between these containers ourselves using links and custom networks, we would need to create shareable volumes and share it among the containers and maintain a map of that as well. And most importantly we would need to monitor the state of the application container and when it dies, manually kill the helper container as well as its no longer required. When a new container is deployed we would need to deploy the new helper container as well.

With PODs, kubernetes does all of this for us automatically. We just need to define what containers a POD consists of and the containers in a POD by default will have access to the same storage, the same network namespace, and same fate as in they will be created together and destroyed together.

Even if our application didn’t happen to be so complex and we could live with a single container, kubernetes still requires you to create PODs. But this is good in the long run as your application is now equipped for architectural changes and scale in the future.

However, multi-pod containers are a rare use-case and we are going to stick to single container per POD in this course.



kubectl

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

```
kubectl get pods
```

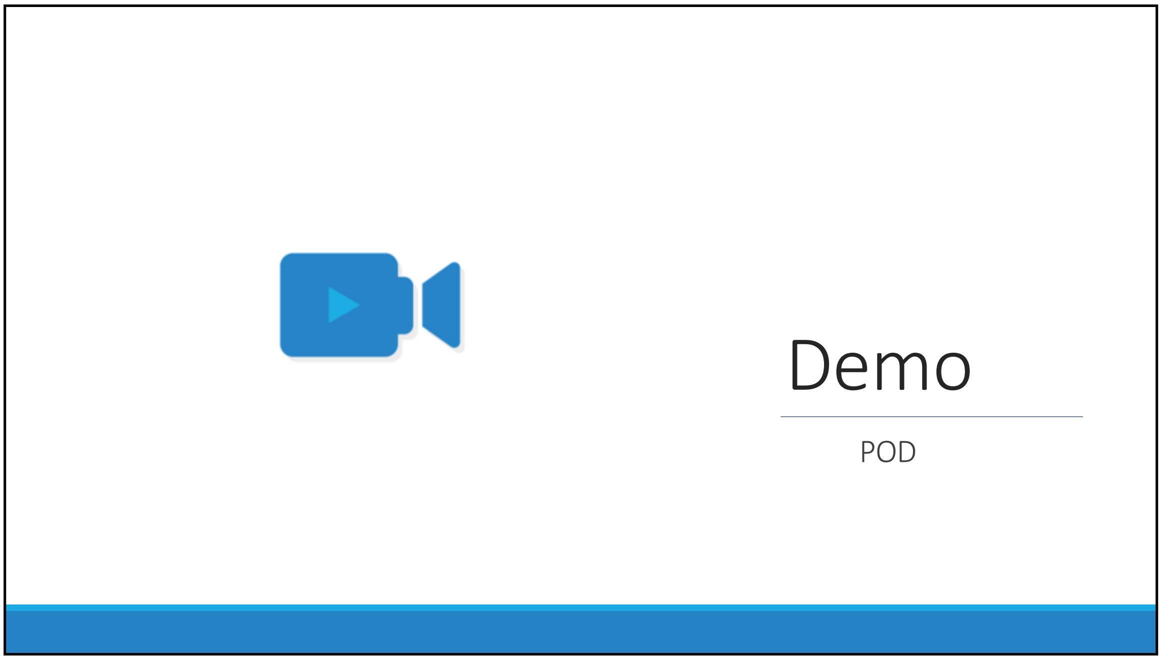
```
C:\Kubernetes>kubectl get pods
NAME                   READY     STATUS              RESTARTS   AGE
nginx-8586cf59-whssr   0/1       ContainerCreating   0          3s
```

```
C:\Kubernetes>kubectl get pods
NAME                   READY     STATUS    RESTARTS   AGE
nginx-8586cf59-whssr   1/1       Running   0          8s
```

Let us now look at how to deploy PODs. Earlier we learned about the kubectl run command. What this command really does is it deploys a docker container by creating a POD. So it first creates a POD automatically and deploys an instance of the nginx docker image. But were does it get the application image from? For that you need to specify the image name using the –-image parameter. The application image, in this case the nginx image, is downloaded from the docker hub repository. Docker hub as we discussed is a public repository were latest docker images of various applications are stored. You could configure kubernetes to pull the image from the public docker hub or a private repository within the organization.

Now that we have a POD created, how do we see the list of PODs available? The kubectl get PODs command helps us see the list of pods in our cluster. In this case we see the pod is in a ContainerCreating state and soon changes to a Running state when it is actually running.

Also remember that we haven't really talked about the concepts on how a user can access the nginx web server. And so in the current state we haven't made the web server accessible to external users. You can access it internally from the Node though. For now we will just see how to deploy a POD and in a later lecture once we learn about networking and services we will get to know how to make this service accessible to end users.



Demo

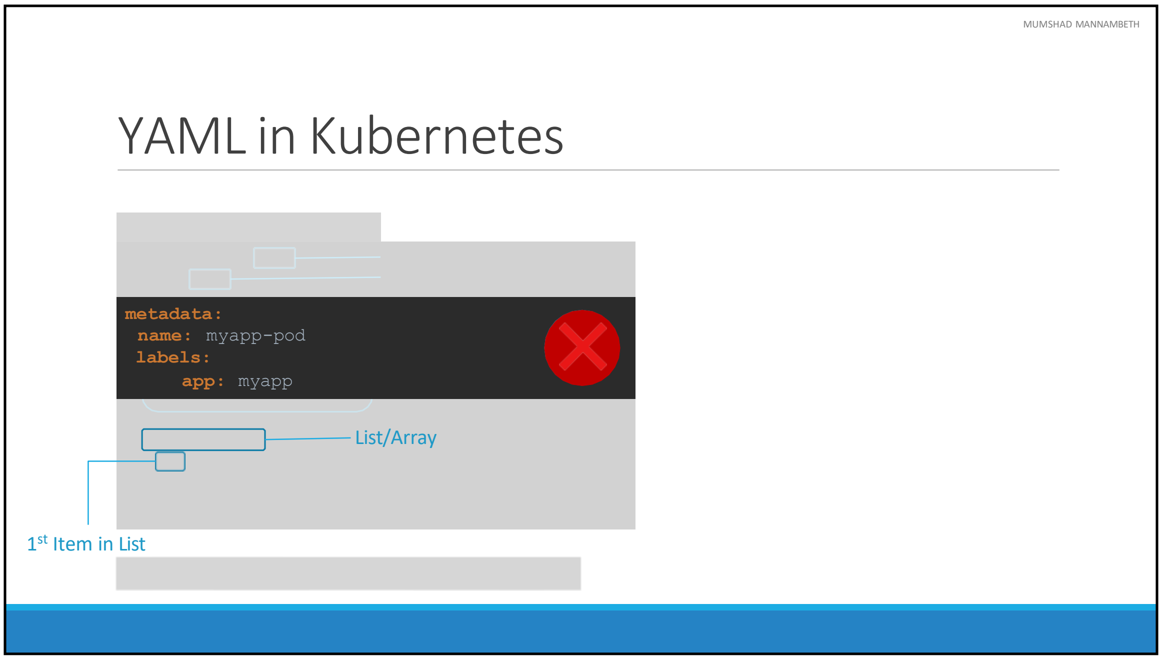
POD

That’s it for this lecture. We will now head over to a Demo and I will see you in the next lecture.

POD

With YAML

Hello and welcome to this lecture, In this lecture we will talk about creating a POD using a YAML based configuration file.



YAML in Kubernetes

```
metadata:
 name: myapp-pod
 labels:
     app: myapp
```

In the previous lecture we learned about YAML files in general. Now we will learn how to develop YAML files specifically for Kubernetes. Kubernetes uses YAML files as input for the creation of objects such as PODs, Replicas, Deployments, Services etc. All of these follow similar structure. A kubernetes definition file always contains 4 top level fields. The apiVersion, kind, metadata and spec. These are top level or root level properties. Think of them as siblings, children of the same parent. These are all REQUIRED fields, so you MUST have them in your configuration file.

Let us look at each one of them. The first one is the 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. For now since we are working on PODs, we will set the apiVersion as v1. Few other possible values for this field are apps/v1beta1, extensions/v1beta1 etc. We will see what these are for later in this course.

Next is the kind. The kind refers to the type of object we are trying to create, which in this case happens to be a POD. So we will set it as Pod. Some other possible values here could be ReplicaSet or Deployment or Service, which is what you see in the kind field in the table on the right.

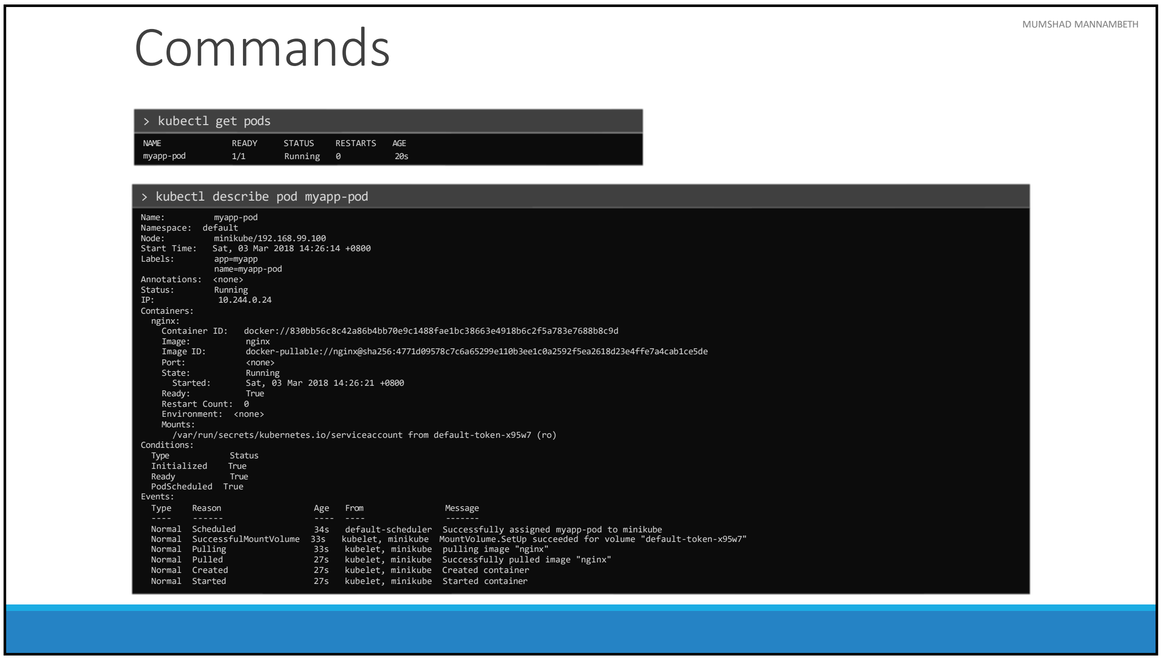
The next is metadata. The metadata is data about the object like its name, labels etc. As you can see unlike the first two were you specified a string value, this, is in the form of a dictionary. So everything under metadata is intended to the right a little bit and so names and labels are children of metadata. The number of spaces before the two properties name and labels doesn't matter, but they should be the same as they are siblings. In this case labels has more spaces on the left than name and so it is now a child of the name property instead of a sibling. Also the two properties must have MORE spaces than its parent, which is metadata, so that its intended to the right a little bit. In this case all 3 have the same number of spaces before them and so they are all siblings, which is not correct. Under metadata, the name is a string value – so you can name your POD myapp-pod - and the labels is a dictionary. So labels is a dictionary within the metadata dictionary. And it can have any key and value pairs as you wish. For now I have added a label app with the value myapp. Similarly you could add other labels as you see fit which will help you identify these objects at a later point in time. 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.

It's IMPORTANT to note that under metadata, you can only specify name or labels or anything else that kubernetes expects to be under metadata. You CANNOT add any other property as you wish under this. However, under labels you CAN have any kind of key or value pairs as you see fit. So its IMPORTANT to understand what each of these parameters expect.

So far we have only mentioned the type and name of the object we need to create which happens to be a POD with the name myapp-pod, but we haven't really specified the container or image we need in the pod. The last section in the configuration file is the specification which is written as spec. Depending on the object we are going to create, this is were we provide additional information to kubernetes pertaining to that object. This is going to be different for different objects, so its important to understand or refer to the documentation section to get the right format for each. Since we are only creating a pod with a single container in it, it is easy. 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 them as we learned in the lecture earlier. In this case though, we will only add a single item in the list, since we plan to have only a single container in the POD. The item in the list is a dictionary, so add a name and image property. The value for image is nginx.

Once the file is created, run the command kubectl create -f followed by the file name which is pod-definition.yml and kubernetes creates the pod.

So to summarize remember the 4 top level properties. apiVersion, kind, metadata and spec. Then start by adding values to those depending on the object you are creating.



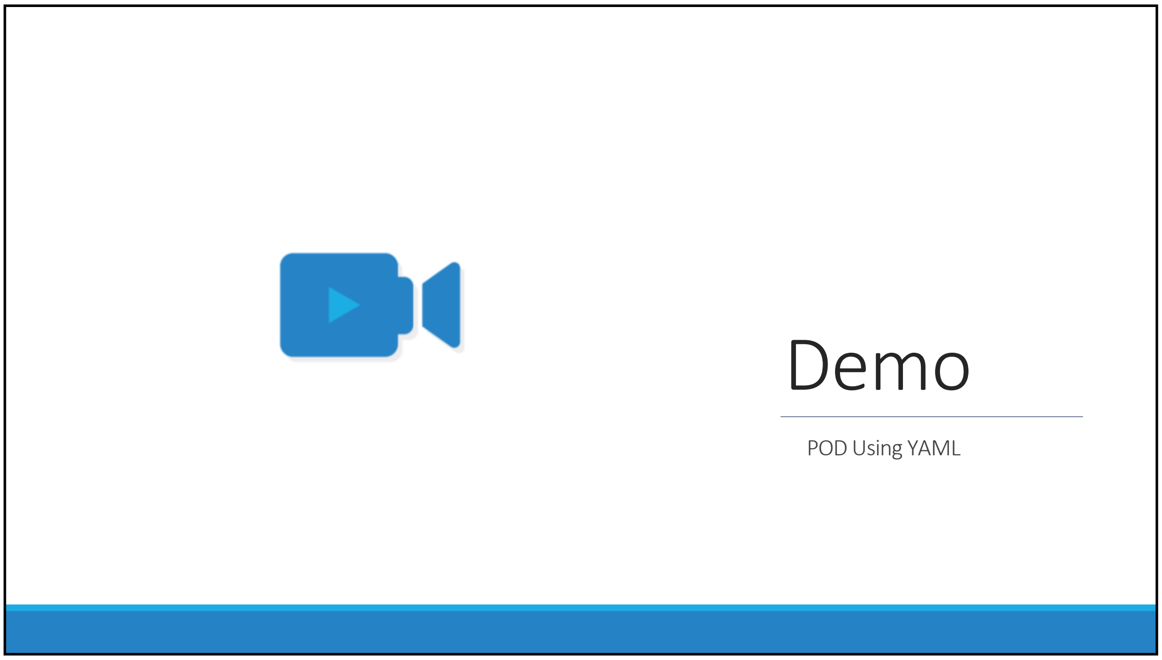
Commands

MUMSHAD MANNAMBETH

```
> kubectl get pods
NAME            READY     STATUS    RESTARTS   AGE
myapp-pod       1/1       Running   0           20s
```

```
> kubectl describe pod myapp-pod
Name:         myapp-pod
Namespace:  default
Node:         minikube/192.168.99.100
Start Time:   Sat, 03 Mar 2018 14:26:14 +0800
Labels:       app=myapp
              name=myapp-pod
Annotations:  <none>
Status:       Running
IP:            10.244.0.24
Containers:
  nginx:
    Container ID:   docker://830bb56c8c42a86b4bb70e9c1488fae1bc38663e4918b6c2f5a783e7688b8c9d
    Image:           nginx
    Image ID:        docker-pullable://nginx@sha256:4771d09578c7c6a65299e110b3ee1c0a2592f5ea2618d23e4ffe7a4cab1ce5de
    Port:            <none>
    State:           Running
      Started:       Sat, 03 Mar 2018 14:26:21 +0800
    Ready:           True
    Restart Count:  0
    Environment:  <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-x95w7 (ro)
Conditions:
  Type           Status
  Initialized    True
  Ready           True
  PodScheduled  True
Events:
  Type    Reason                  Age   From               Message
  ----    ------                  ----  ----               -------
  Normal  Scheduled               34s   default-scheduler  Successfully assigned myapp-pod to minikube
  Normal  SuccessfulMountVolume  33s   kubelet, minikube  MountVolume.SetUp succeeded for volume "default-token-x95w7"
  Normal  Pulling                 33s   kubelet, minikube  pulling image "nginx"
  Normal  Pulled                  27s   kubelet, minikube  Successfully pulled image "nginx"
  Normal  Created                 27s   kubelet, minikube  Created container
  Normal  Started                 27s   kubelet, minikube  Started container
```

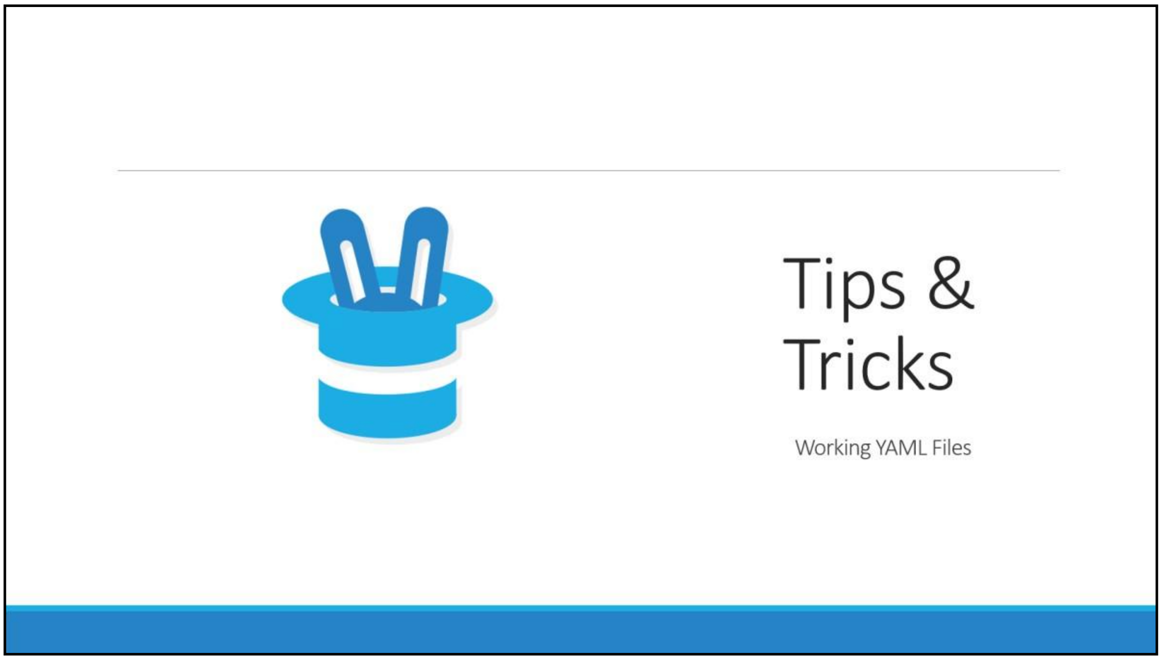
Once we create the pod, how do you see it? Use the kubectl get pods command to see a list of pods available. In this case its just one. To see detailed information about the pod run the kubectl describe pod command. This will tell you information about the POD, when it was created, what labels are assigned to it, what docker containers are part of it and the events associated with that POD.



Demo

POD Using YAML

That’s it for this lecture. We will now head over to a Demo and I will see you in the next lecture.



Tips & Tricks

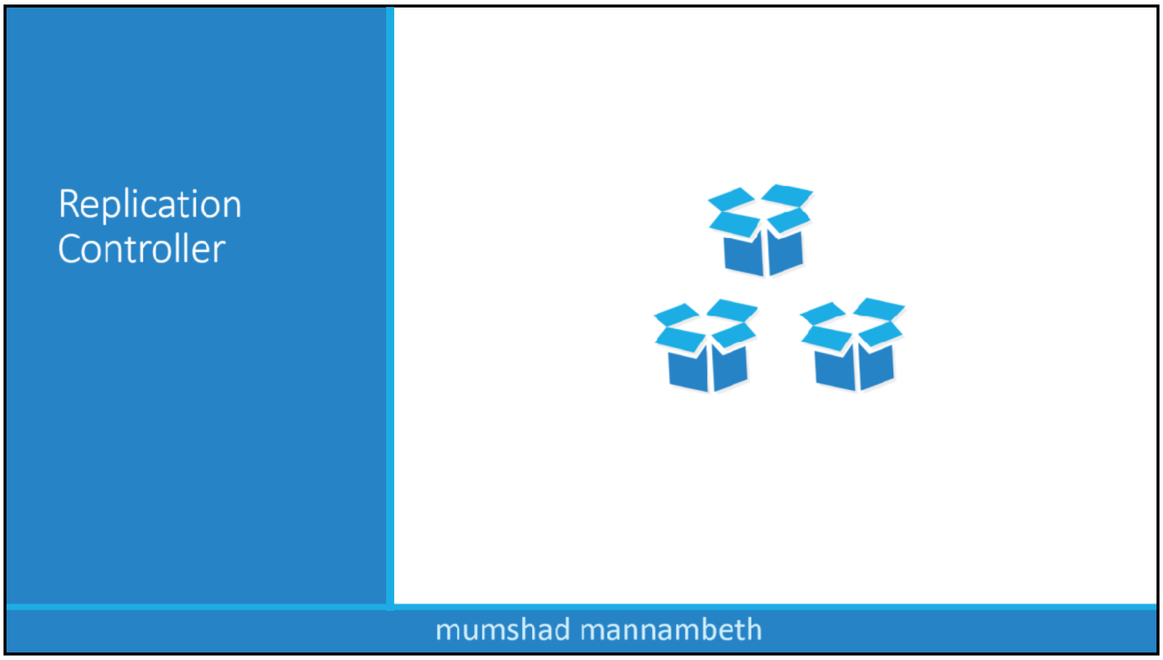
Working YAML Files

That’s it for this lecture. We will now head over to a Demo and I will see you in the next lecture.

Resources

Link to Versions and Groups - https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.9/#replicaset-v1-apps

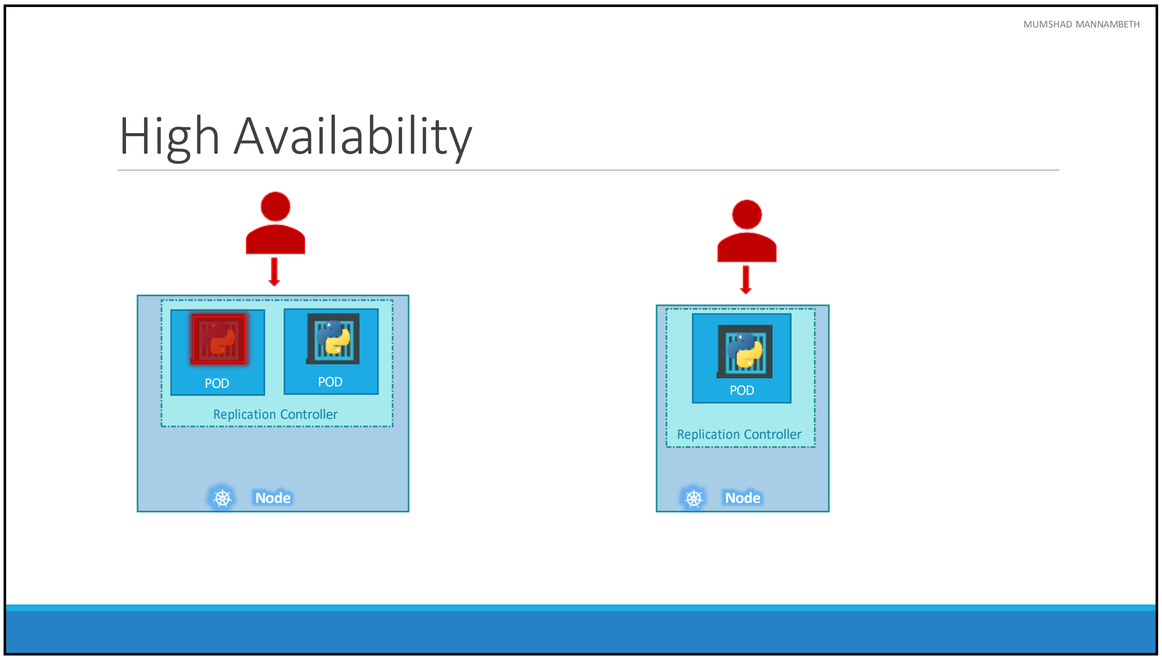
https://plugins.jetbrains.com/plugin/9354-kubernetes-and-openshift-resource-support



Replication Controller

mumshad mannambeth

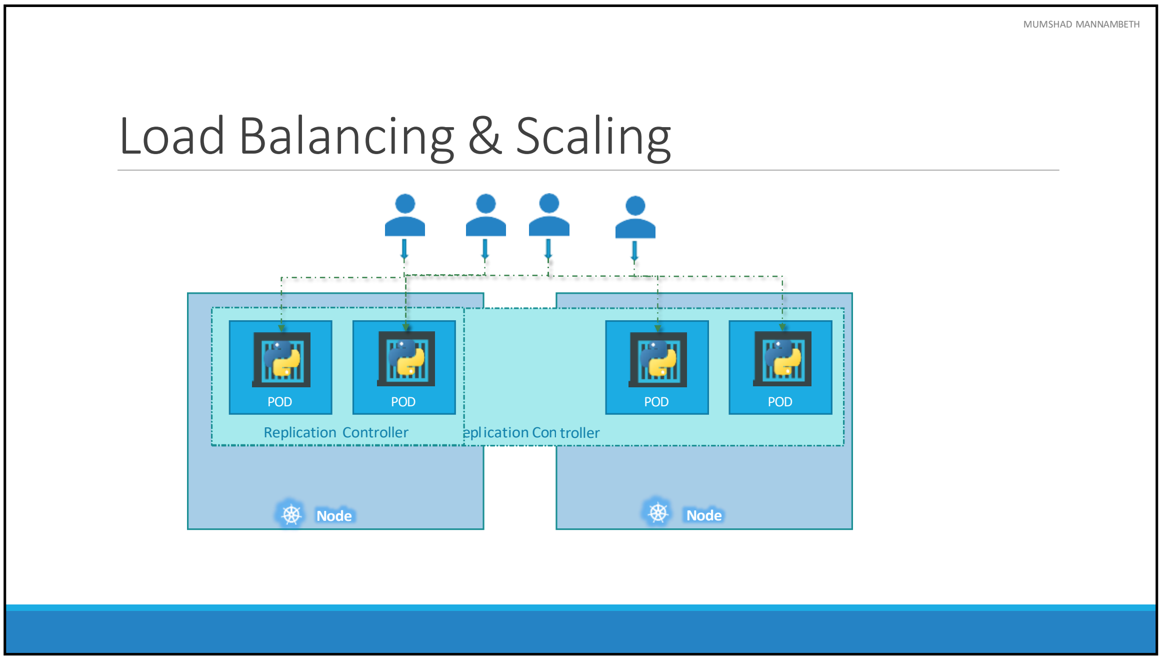
Hello and welcome to this lecture on Kubernetes Controllers. In this lecture we will discuss about Kubernetes Controllers. Controllers are the brain behind Kubernetes. They are processes that monitor kubernetes objects and respond accordingly. In this lecture we will discuss about one controller in particular. And that is the Replication Controller.


MUMSHAD MANNAMBETH

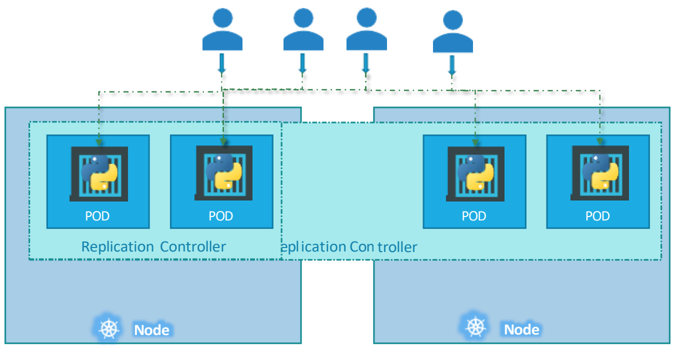
High Availability

So what is a replica and why do we need a replication controller? Let's go back to our first scenario were 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 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.

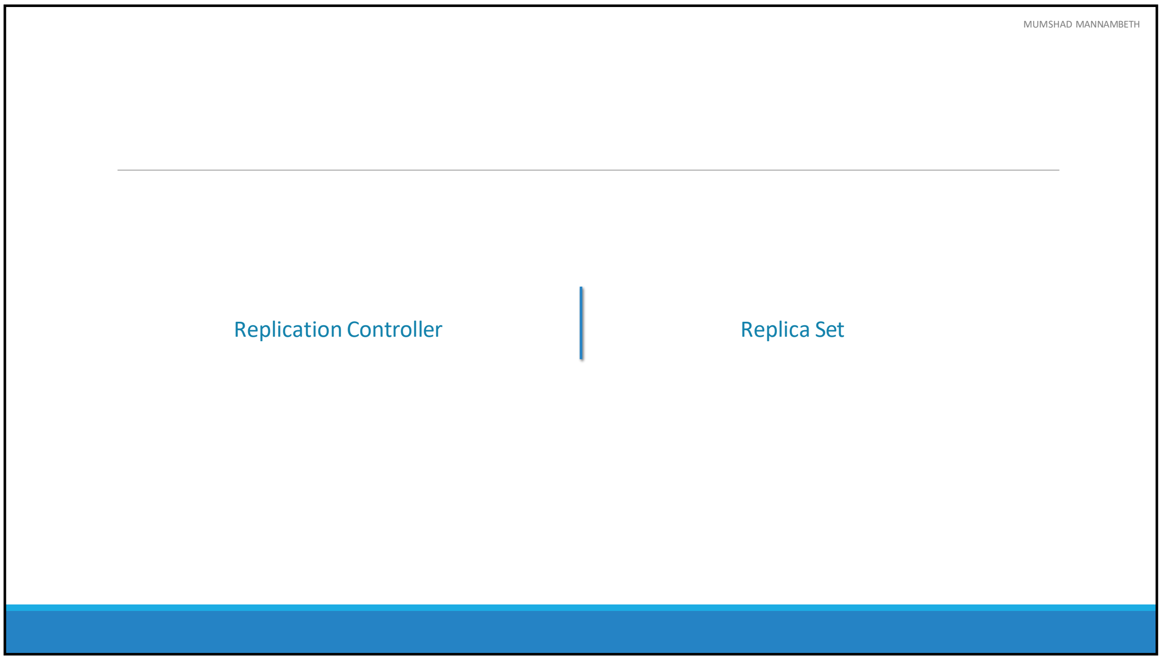
So does that mean you can't use a replication controller if you plan to have a single POD? No! Even if you have a single POD, the replication controller can help by automatically bringing up a new POD when the existing one fails. Thus the replication controller ensures that the specified number of PODs are running at all times. Even if it's just 1 or 100.


MUMSHAD MANNAMBETH

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 you can see, 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 scale our application when the demand increases.

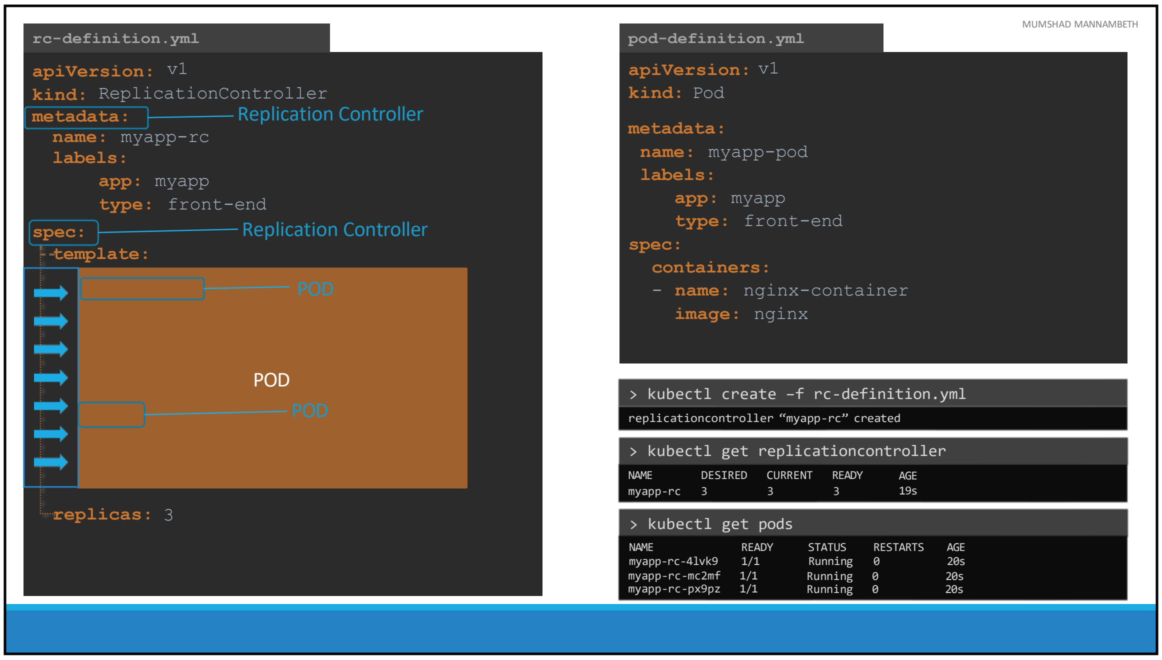

MUMSHAD MANNAMBETH

Replication Controller

Replica Set

It’s important to note that there are two similar terms. Replication Controller and Replica Set. 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. However, whatever we discussed in the previous few slides remain applicable to both these technologies. There are minor differences in the way each works and we will look at that in a bit.

As such we will try to stick to Replica Sets in all of our demos and implementations going forward.



```
rc-definition.yml
apiVersion: v1
kind: ReplicationController
metadata:
  name: myapp-rc
  labels:
      app: myapp
      type: front-end
spec:
 -template:

  replicas: 3
```

```
pod-definition.yml
apiVersion: v1
kind: Pod

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

```
> kubectl create –f rc-definition.yml
replicationcontroller “myapp-rc” created
```

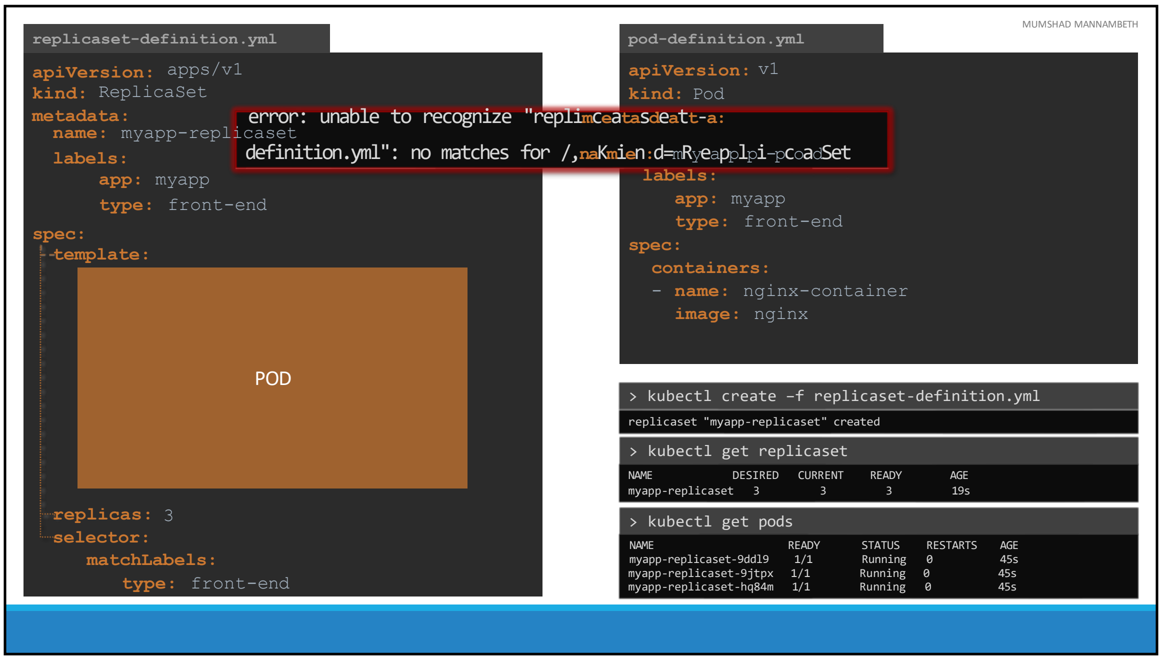
```
> kubectl get replicationcontroller
NAME       DESIRED   CURRENT   READY    AGE
myapp-rc   3         3         3        19s
```

```
> kubectl get pods
NAME              READY     STATUS    RESTARTS   AGE
myapp-rc-4lvk9    1/1       Running   0          20s
myapp-rc-mc2mf    1/1       Running   0          20s
myapp-rc-px9pz    1/1       Running   0          20s
```

Let us now look at how we create a replication controller. As with the previous lecture, we start by creating a replication controller definition file. We will name it rc-definition.yml. As with any kubernetes definition file, we will have 4 sections. The apiVersion, kind, metadata and spec. The apiVersion is specific to what we are creating. In this case replication controller is supported in kubernetes apiVersion v1. So we will write it as v1. The kind as we know is ReplicationController. Under metadata, we will add a name and we will call it myapp-rc. And we will also add a few labels, app and type and assign values to them. So far, it has been very similar to how we created a POD in the previous section. The next is the most crucial part of the definition file and that is the specification written as spec. For any kubernetes definition file, 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. But what POD? We create a template section under spec to provide a POD template to be used by the replication controller to create replicas. Now how do we DEFINE the POD template? It’s not that hard because, we have already done that in the previous exercise. Remember, we created a pod-definition file in the previous exercise. 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. Remember whatever we move must be UNDER the template section. Meaning, they should be intended to the right and have more spaces before them than the template line itself. 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.

Now, there is something still missing. We haven't mentioned how many replicas we need in the replication controller. For that, add another property to the spec called replicas and input the number of replicas you need under it. Remember that the template and replicas are direct children of the spec section. So they are siblings and must be on the same vertical line : having equal number of spaces before them.

Once the file is ready, run the kubectl create command and input the file using the –f parameter. The replication controller Is created. 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 the list of created replication controllers run the kubectl get replication controller command and you will see the replication controller listed. We can also see the desired number of replicas or pods, the current number of replicas and how many of them are ready. If you would like to see the pods that were created by the replication controller, run the kubectl get pods command and you will see 3 pods running. Note that all of them are starting with the name of the replication controller which is myapp-rc indicating that they are all created automatically by the replication controller.



replicaset-definition.yml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
      app: myapp
      type: front-end

spec:
 -template:

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

```
error: unable to recognize "replicaset-
definition.yml": no matches for /, Kind=ReplicaSet
```

pod-definition.yml

```yaml
apiVersion: v1
kind: Pod

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

```
> 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-9ddl9     1/1        Running    0            45s
myapp-replicaset-9jtpx    1/1        Running    0            45s
myapp-replicaset-hq84m    1/1        Running    0            45s
```

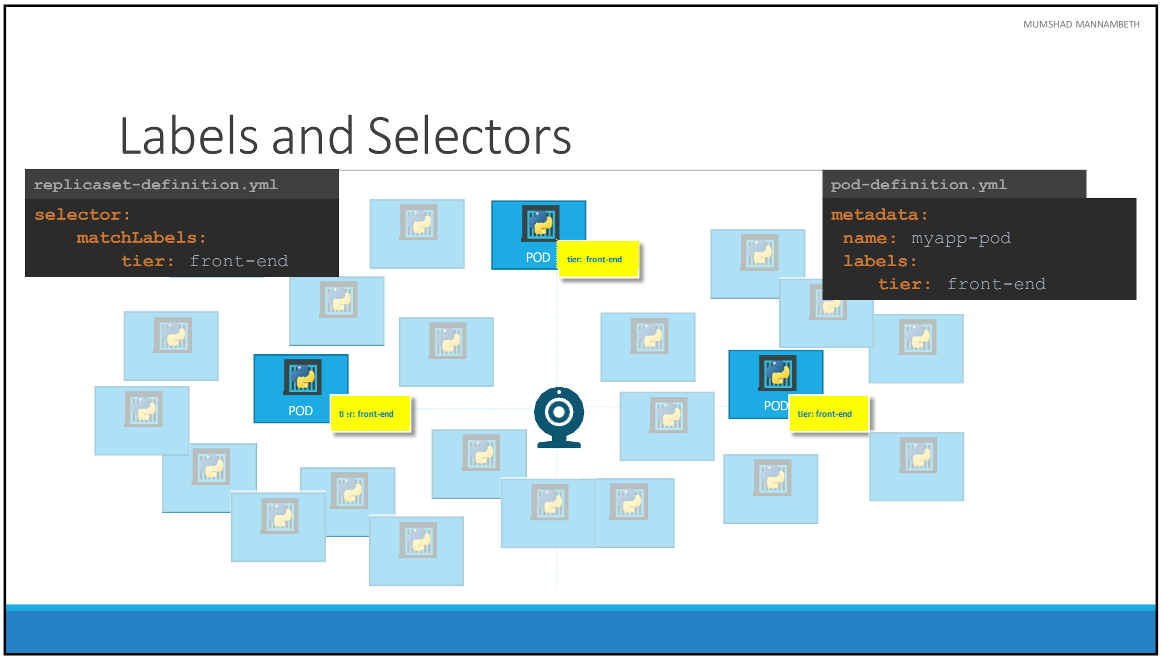
What we just saw was ReplicationController. Let us now look at ReplicaSet. It is very similar to replication controller. As usual, first we have apiVersion, kind, metadata and spec. The apiVersion though is a bit different. It is apps/v1 which is different from what we had before for replication controller. For replication controller it was simply v1. If you get this wrong, you are likely to get an error that looks like this. It would say no match for kind ReplicaSet, because the specified kubernetes api version has no support for ReplicaSet.

The kind would be ReplicaSet and we add name and labels in metadata.

The specification section looks very similar to replication controller. It has a template section were we provide pod-definition as before. So I am going to copy contents over from pod-definition file. And we have number of replicas set to 3. However, 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. But why would you have to specify what PODs fall under it, if you have provided the contents of the pod-definition file itself in the template? 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. I will elaborate this in the next slide.

But before we get into that, I would like to mention that 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. When you skip it, as we did in the previous slide, 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. And it has to 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. The replicaset selector also provides many other options for matching labels that were not available in a replication controller.

And as always to create a ReplicaSet run the kubectl create command providing the definition file as input and to see the created replicasets run the kubectl get replicaset command. To get list of pods, simply run the kubectl get pods command.


MUMSHAD MANNAMBETH

Labels and Selectors

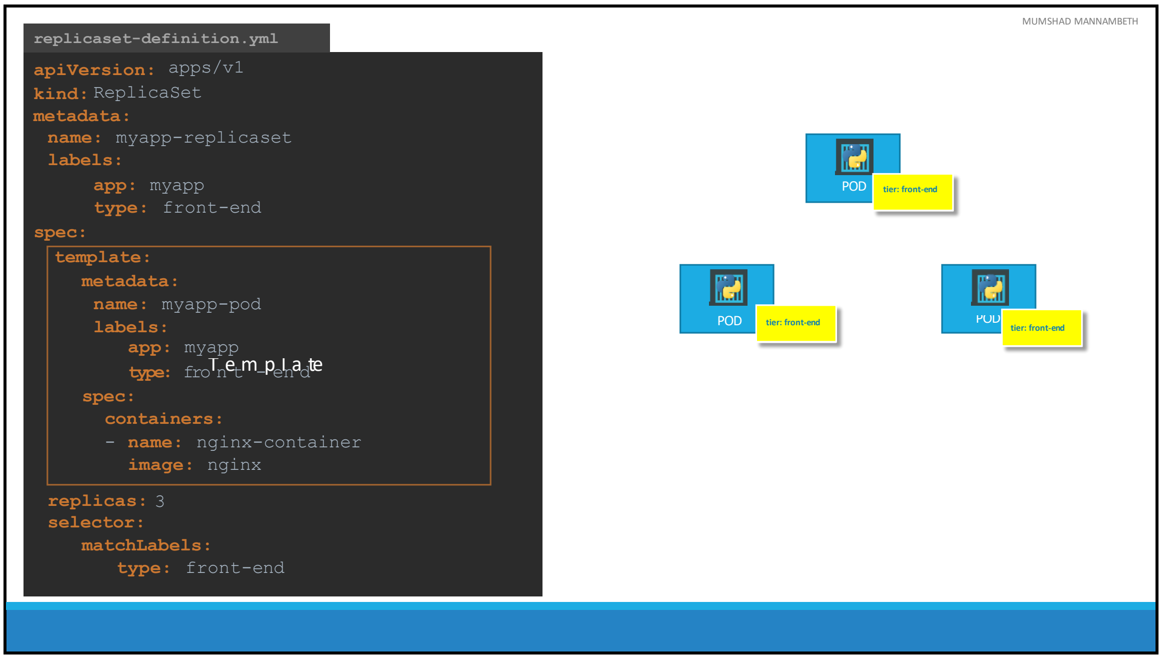
replicaset-definition.yml

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

pod-definition.yml

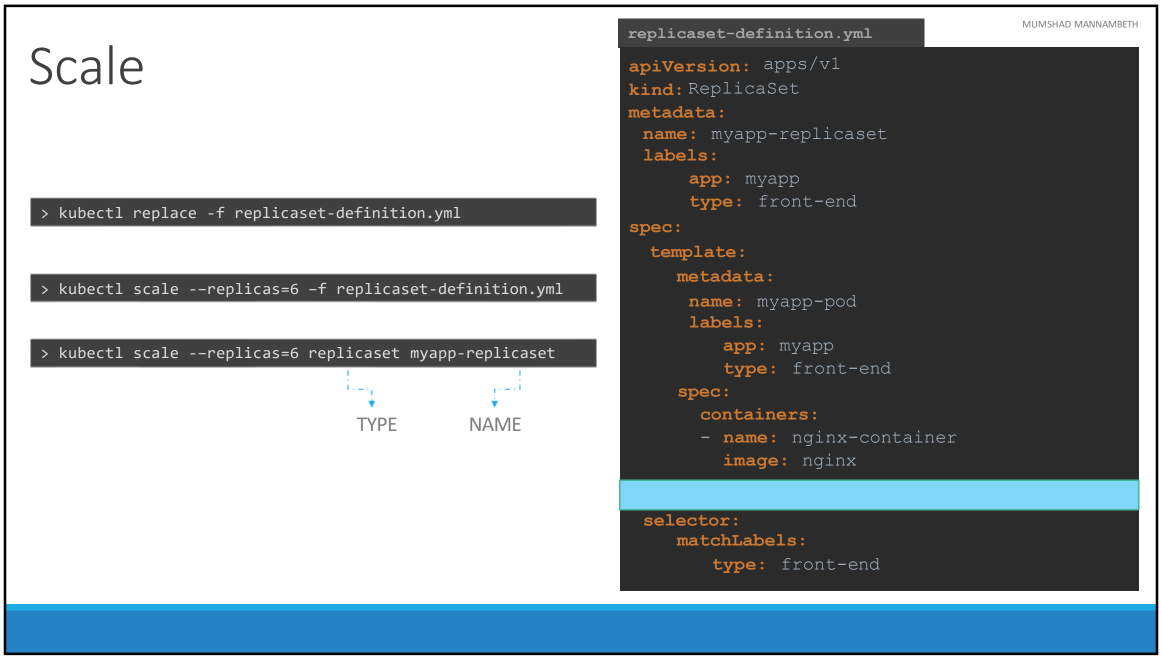
```
metadata:
  name: myapp-pod
  labels:
      tier: front-end
```

So what is the deal with Labels and Selectors? Why do we label our PODs and objects in kubernetes? Let us look at a simple scenario. Say we deployed 3 instances of our frontend web application as 3 PODs. We would like to create a replication controller or replica set to ensure that we have 3 active PODs at anytime. And YES that is one of the use cases of replica sets. You CAN use it to monitor existing pods, if you have them already created, as it IS in this example. In case they were not created, the replica set will create them for you. The role of the replicaset is to monitor the pods and if any of them were to fail, deploy new ones. The replica set is in FACT 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 were 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.



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

Now let me ask you a question along the same lines. 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 in the previous slide. Say for instance we have the same scenario as in the previous slide were 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.



Scale

MUMSHAD MANNAMBETH

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

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

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

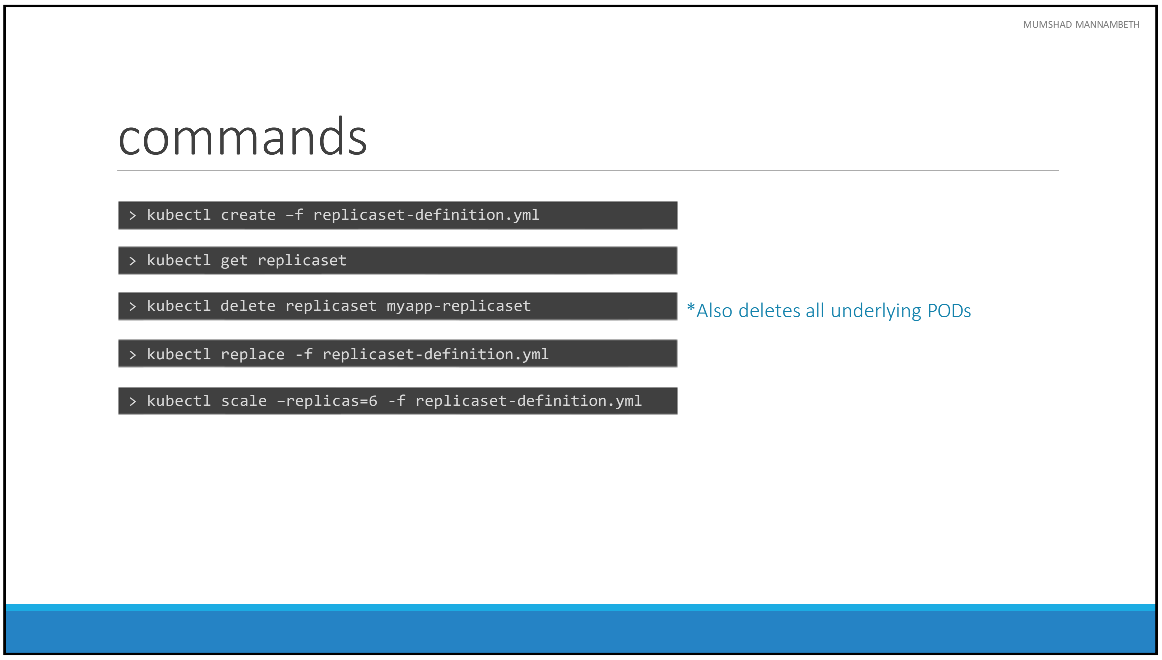
replicaset-definition.yml

```
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

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

Let’s look at how we scale the replicaset. Say we started with 3 replicas and in the future we decide to scale to 6. How do we update our replicaset to scale to 6 replicas. Well there are multiple ways to do it. The first, is to update the number of replicas in the definition file to 6. Then run the kubectl replace command specifying the same file using the –f parameter and that will update the replicaset to have 6 replicas.

The second way to do it is to run the kubectl scale command. Use the replicas parameter to provide the new number of replicas and specify the same file as input. 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 otherwords, 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.

There are also options available for automatically scaling the replicaset based on load, but that is an advanced topic and we will discuss it at a later time.


MUMSHAD MANNAMBETH

commands

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

```
> kubectl get replicaset
```

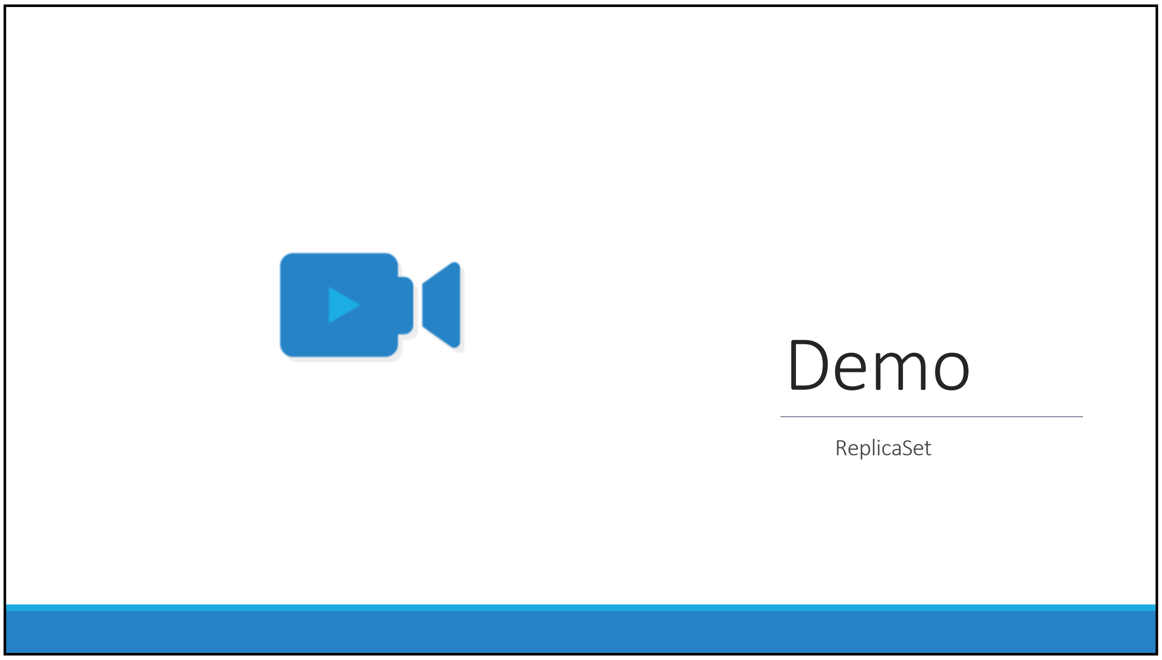
```
> kubectl delete replicaset myapp-replicaset
```

*Also deletes all underlying PODs

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

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

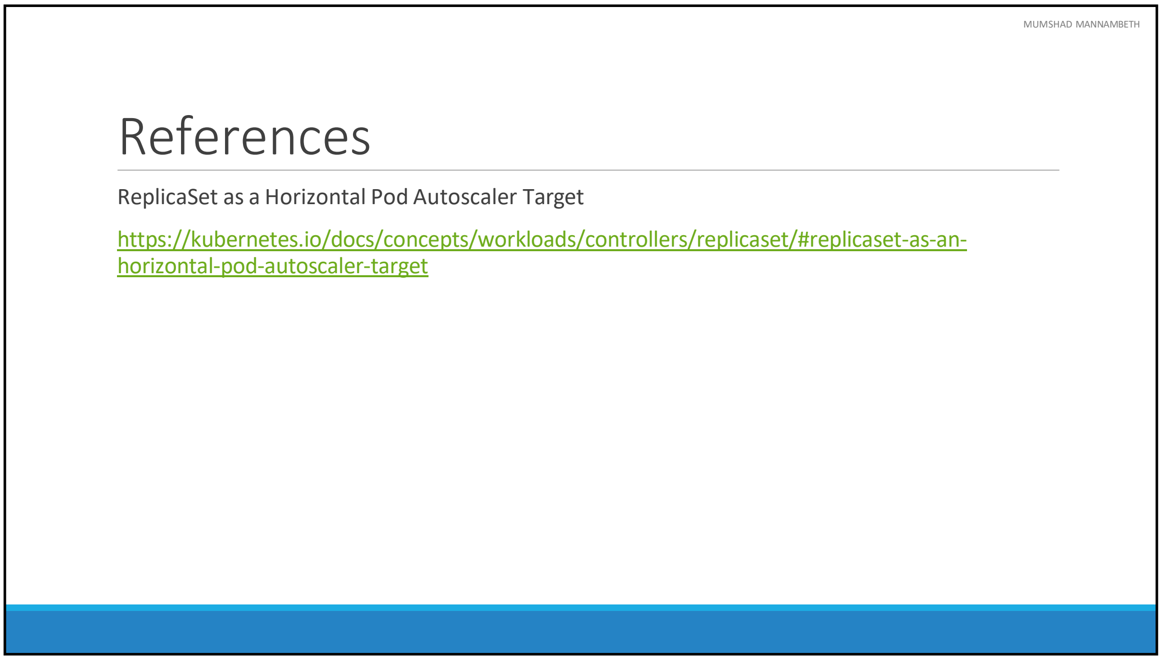
Let us now review the commands real quick. The kubectl create command, as we know, is used to create a replca set. You must provide the input file using the –f parameter. Use the kubectl get command to see list of replicasets created. Use the kubectl delete replicaset command followed by the name of the replica set to delete the replicaset. And then we have the kubectl replace command to replace or update replicaset and also the kubectl scale command to scale the replicas simply from the command line without having to modify the file.



Demo

ReplicaSet

That’s it for this lecture. We will now head over to a Demo and I will see you in the next lecture.

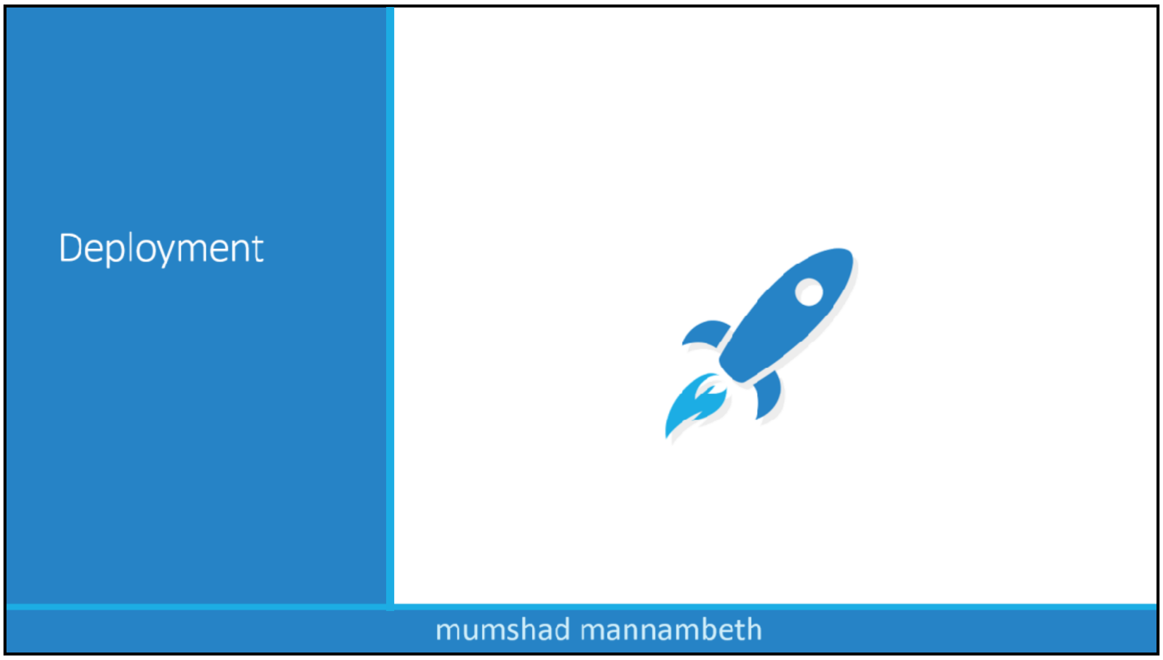


MUMSHAD MANNAMBETH

References

ReplicaSet as a Horizontal Pod Autoscaler Target

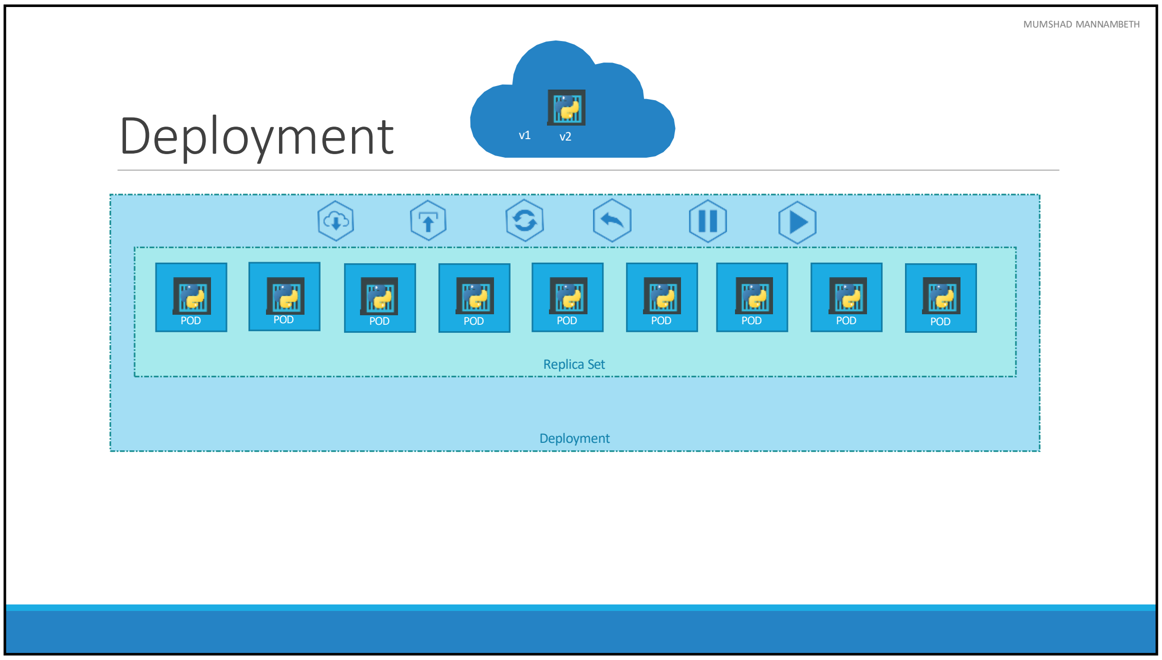
https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/#replicaset-as-an-horizontal-pod-autoscaler-target



Deployment

mumshad mannambeth

In this lecture we will discuss about Kubernetes Deployments.



For a minute, let us forget about PODs and replicasets and other kubernetes concepts and talk about how you might want to deploy your application in a production environment. Say for example you have a web server that needs to be deployed in a production environment. You need not ONE, but many such instances of the web server running for obvious reasons.

Secondly, when newer versions of application builds become available on the docker registry, you would like to UPGRADE your docker instances seamlessly.

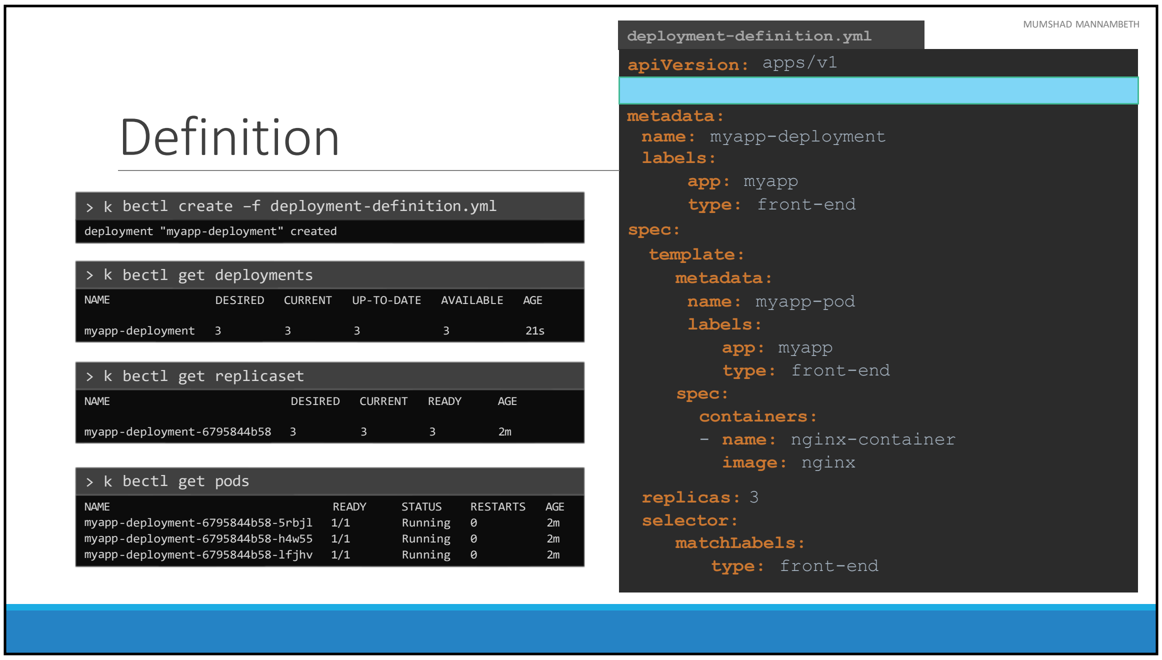
However, when you upgrade your instances, you do not want to upgrade all of them at once as we just did. This may impact users accessing our applications, so you may want to upgrade them one after the other. And that kind of upgrade is known as Rolling Updates.

Suppose one of the upgrades you performed resulted in an unexpected error and you are asked to undo the recent update. You would like to be able to rollBACK the changes that were recently carried out.

Finally, say for example you would like to make multiple changes to your environment such as upgrading the underlying WebServer versions, as well as scaling your environment and also modifying the resource allocations etc. You do not want to apply each change immediately after the command is run, instead you would like to apply a pause to your environment, make the changes and then resume so that all changes are rolled-out together.

All of these capabilities are available with the kubernetes Deployments.

So far in this course we discussed about PODs, which deploy single instances of our application such as the web application in this case. 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.


MUMSHAD MANNAMBETH

Definition

```
> k bectl create -f deployment-definition.yml
deployment "myapp-deployment" created
```

```
> k bectl get deployments
NAME              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE

myapp-deployment  3         3         3            3           21s
```

```
> k bectl get replicaset
NAME                         DESIRED   CURRENT   READY     AGE

myapp-deployment-6795844b58  3         3         3         2m
```

```
> k bectl get pods
NAME                                READY     STATUS    RESTARTS   AGE
myapp-deployment-6795844b58-5rbjl   1/1       Running   0          2m
myapp-deployment-6795844b58-h4w55   1/1       Running   0          2m
myapp-deployment-6795844b58-lfjhv   1/1       Running   0          2m
```

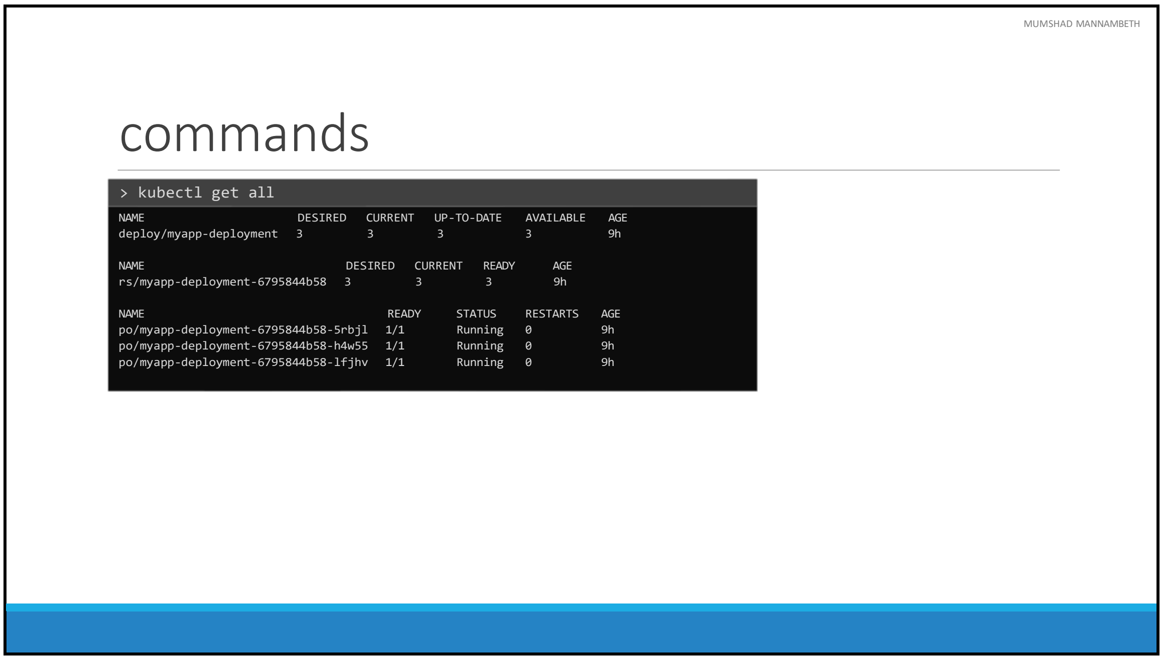
```
deployment-definition.yml
apiVersion: apps/v1

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
```

So how do we create a deployment. As with the previous components, we first create a deployment definition file. The contents of the deployment-definition file are exactly similar to the replicaset definition file, except for the kind, which is now going to be Deployment.

If we walk through the contents of the file it has an apiVersion which is apps/v1, metadata which has name and labels and a spec that has template, replicas and selector. The template has a POD definition inside it.

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

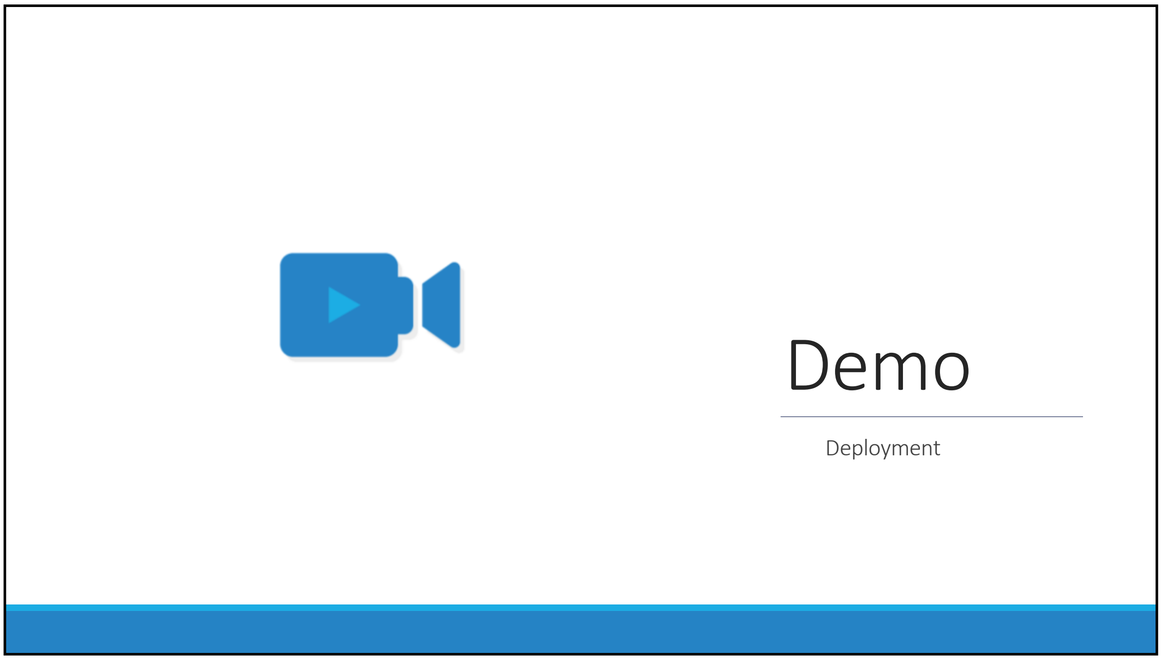
So far there hasn't been much of a difference between replicaset and deployments, except for the fact that deployments created a new kubernetes object called deployments. We will see how to take advantage of the deployment using the use cases we discussed in the previous slide in the upcoming lectures.


MUMSHAD MANNAMBETH

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
```

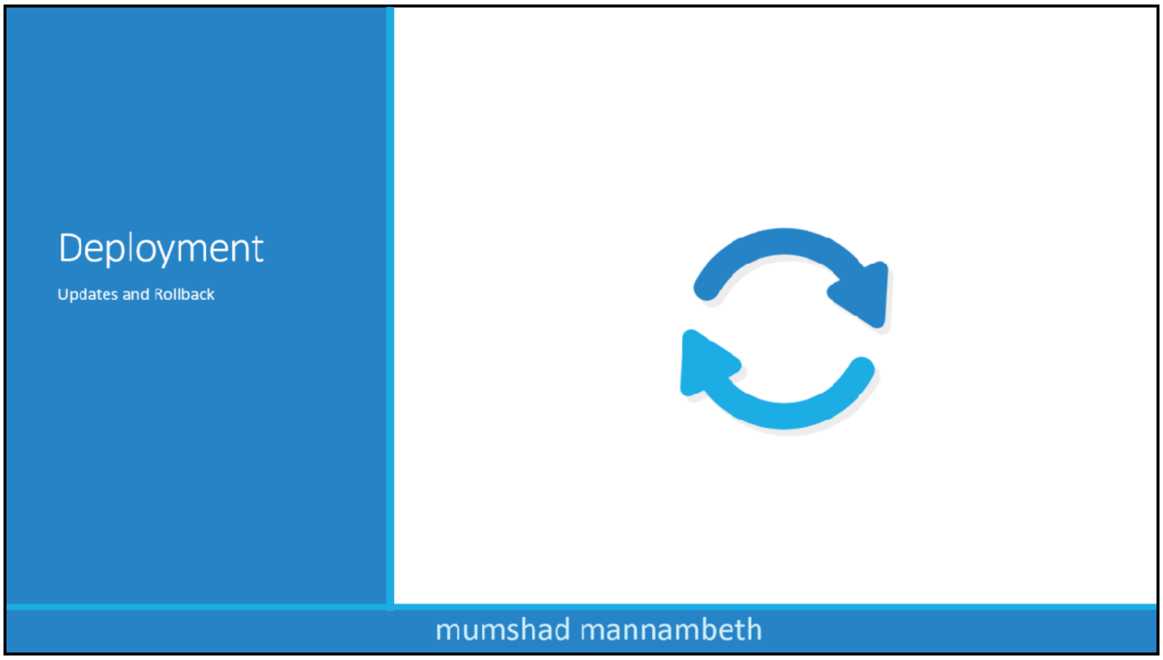
To see all the created objects at once run the kubectl get all command.



Demo

Deployment

That's it for this lecture. We will now head over to a Demo and I will see you in the next lecture.

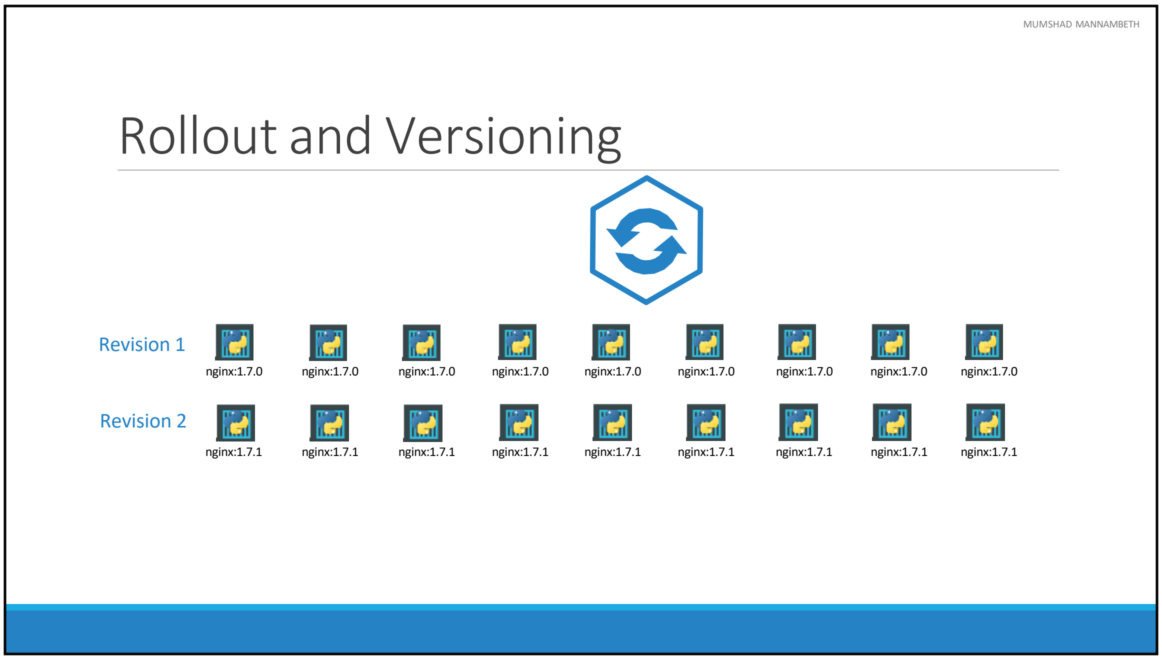


Deployment

Updates and Rollback

mumshad mannambeth

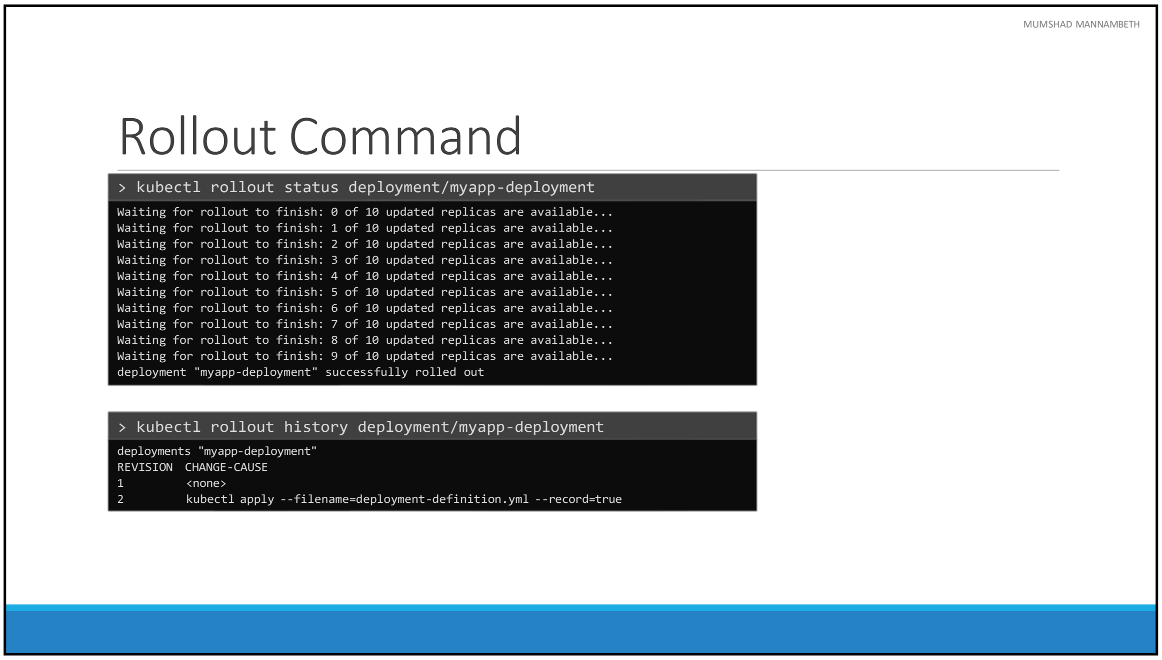
In this lecture we will talk about updates and rollbacks in a Deployment.



MUMSHAD MANNAMBETH

Rollout and Versioning

Before we look at how we upgrade our application, let's try to understand Rollouts and Versioning in a deployment. Whenever you 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 your application containers. When you first create a deployment, it triggers a rollout. A new rollout creates a new Deployment revision. Let's call it 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 named Revision 2. This helps us keep track of the changes made to our deployment and enables us to rollback to a previous version of deployment if necessary.


MUMSHAD MANNAMBETH

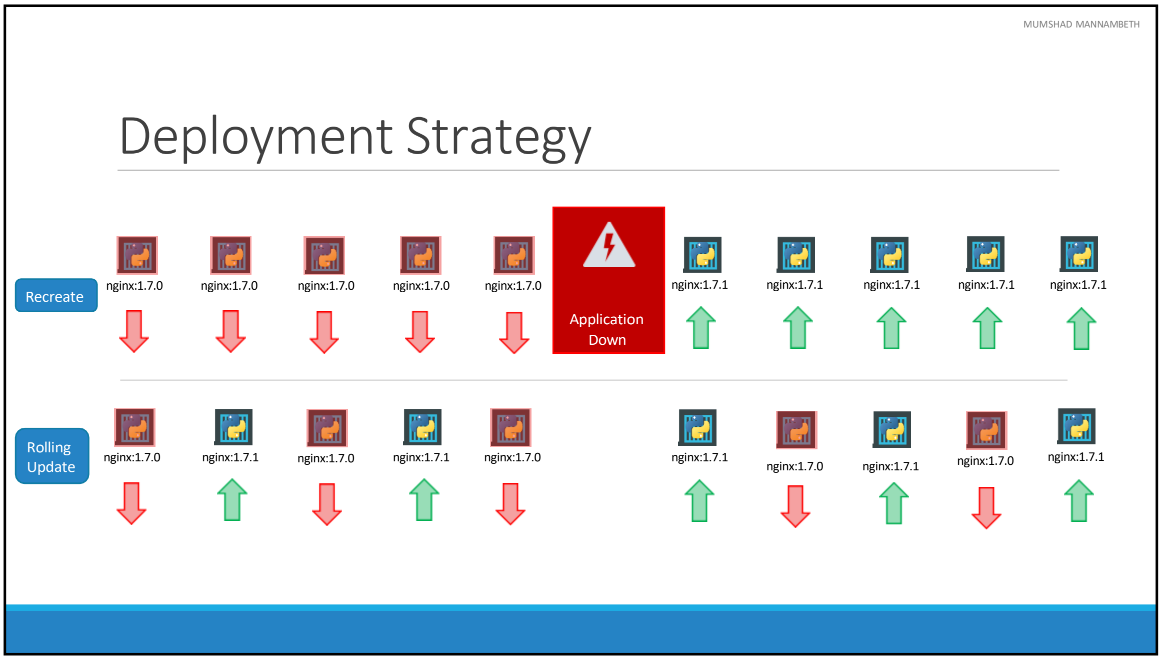
Rollout Command

```
> kubectl rollout status deployment/myapp-deployment
Waiting for rollout to finish: 0 of 10 updated replicas are available...
Waiting for rollout to finish: 1 of 10 updated replicas are available...
Waiting for rollout to finish: 2 of 10 updated replicas are available...
Waiting for rollout to finish: 3 of 10 updated replicas are available...
Waiting for rollout to finish: 4 of 10 updated replicas are available...
Waiting for rollout to finish: 5 of 10 updated replicas are available...
Waiting for rollout to finish: 6 of 10 updated replicas are available...
Waiting for rollout to finish: 7 of 10 updated replicas are available...
Waiting for rollout to finish: 8 of 10 updated replicas are available...
Waiting for rollout to finish: 9 of 10 updated replicas are available...
deployment "myapp-deployment" successfully rolled out
```

```
> kubectl rollout history deployment/myapp-deployment
deployments "myapp-deployment"
REVISION  CHANGE-CAUSE
1         <none>
2         kubectl apply --filename=deployment-definition.yml --record=true
```

You can see the status of your rollout by running the command: kubectl rollout status followed by the name of the deployment.

To see the revisions and history of rollout run the command kubectl rollout history followed by the deployment name and this will show you the revisions.



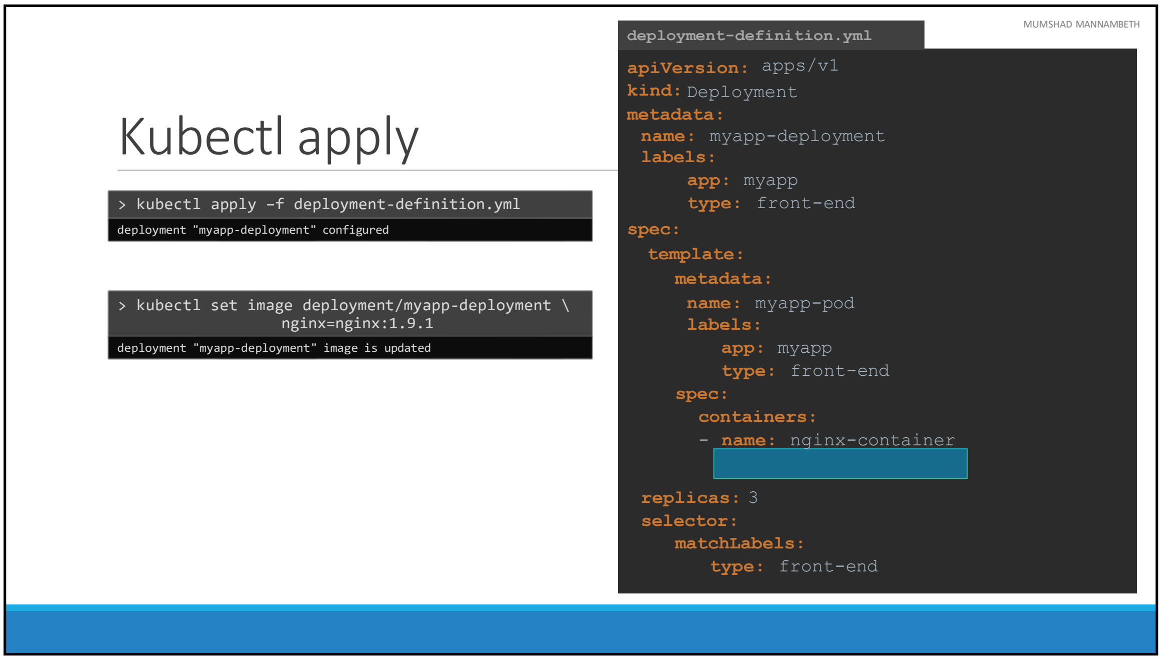
MUMSHAD MANNAMBETH

Deployment Strategy

There are two types of deployment strategies. Say for example you have 5 replicas of your 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. Meaning first, destroy the 5 running instances and then deploy 5 new instances of the new application version. The problem with this as you can imagine, 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, and thankfully this is NOT the default deployment strategy.

The second strategy is were 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.

Remember, if you do not specify a strategy while creating the deployment, it will assume it to be Rolling Update. In other words, RollingUpdate is the default Deployment Strategy.


MUMSHAD MANNAMBETH

Kubectl apply

```
> kubectl apply -f deployment-definition.yml
deployment "myapp-deployment" configured
```

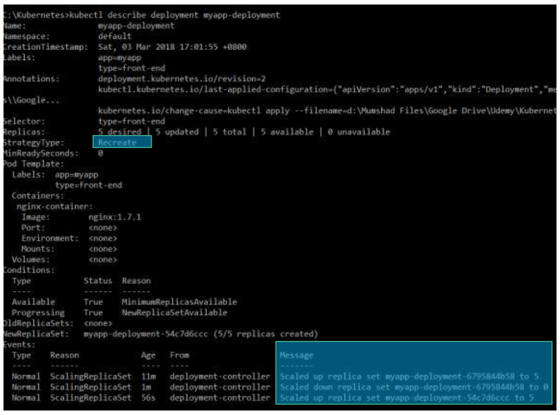
```
> kubectl set image deployment/myapp-deployment \
                        nginx=nginx:1.9.1
deployment "myapp-deployment" image is updated
```

deployment-definition.yml

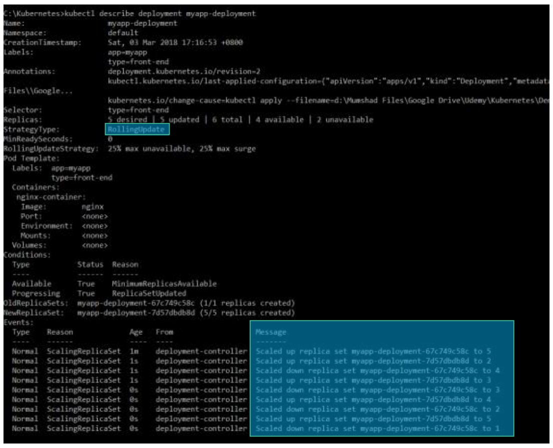
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
      app: myapp
      type: front-end
spec:
  template:
     metadata:
       name: myapp-pod
       labels:
           app: myapp
           type: front-end
     spec:
       containers:
       - name: nginx-container

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

So we talked about upgrades. How exactly DO you update your deployment? When I say update it could be different things such as updating your application version by updating the version of docker containers used, updating their labels or updating the number of replicas etc. Since we already have a deployment definition file it is easy for us to modify this file. Once we make the necessary changes, we run the kubectl apply command to apply the changes. A new rollout is triggered and a new revision of the deployment is created.

But there is ANOTHER way to do the same thing. You could use the kubectl set image command to update the image of your application. But remember, doing it this way will result in the deployment-definition file having a different configuration. So you must be careful when using the same definition file to make changes in the future.



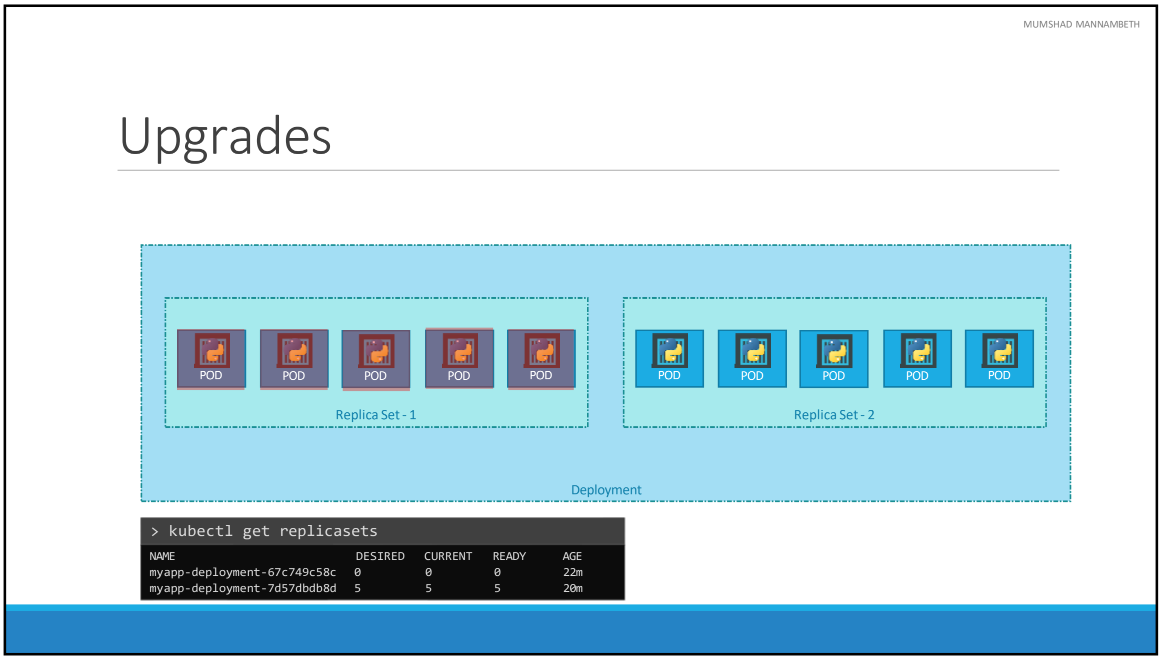
```
C:\Kubernetes>kubectl describe deployment myapp-deployment
Name:                   myapp-deployment
Namespace:              default
CreationTimestamp:      Sat, 03 Mar 2018 17:01:55 +0800
Labels:                 app=myapp
                        type=front-end
Annotations:            deployment.kubernetes.io/revision=2
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","me
s\\Google...
                        kubernetes.io/change-cause=kubectl apply --filename=d:\Mumshad Files\Google Drive\Udemy\Kubernet
Selector:               type=front-end
Replicas:               5 desired | 5 updated | 5 total | 5 available | 0 unavailable
StrategyType:           Recreate
MinReadySeconds:        0
Pod Template:
  Labels:  app=myapp
           type=front-end
  Containers:
   nginx-container:
    Image:        nginx:1.7.1
    Port:         <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets:  <none>
NewReplicaSet:   myapp-deployment-54c7d6ccc (5/5 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  11m   deployment-controller  Scaled up replica set myapp-deployment-6795844b58 to 5
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled down replica set myapp-deployment-6795844b58 to 0
  Normal  ScalingReplicaSet  56s   deployment-controller  Scaled up replica set myapp-deployment-54c7d6ccc to 5
```



```
C:\Kubernetes>kubectl describe deployment myapp-deployment
Name:                   myapp-deployment
Namespace:              default
CreationTimestamp:      Sat, 03 Mar 2018 17:16:53 +0800
Labels:                 app=myapp
                        type=front-end
Annotations:            deployment.kubernetes.io/revision=2
                        kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment","metadat
Files\\Google...
                        kubernetes.io/change-cause=kubectl apply --filename=d:\Mumshad Files\Google Drive\Udemy\Kubernetes\De
Selector:               type=front-end
Replicas:               5 desired | 5 updated | 6 total | 4 available | 2 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=myapp
           type=front-end
  Containers:
   nginx-container:
    Image:        nginx
    Port:         <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    ReplicaSetUpdated
OldReplicaSets:  myapp-deployment-67c749c58c (1/1 replicas created)
NewReplicaSet:   myapp-deployment-7d57dbdb8d (5/5 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  1m    deployment-controller  Scaled up replica set myapp-deployment-67c749c58c to 5
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 2
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 4
  Normal  ScalingReplicaSet  1s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 3
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 3
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 4
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 2
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled up replica set myapp-deployment-7d57dbdb8d to 5
  Normal  ScalingReplicaSet  0s    deployment-controller  Scaled down replica set myapp-deployment-67c749c58c to 1
```

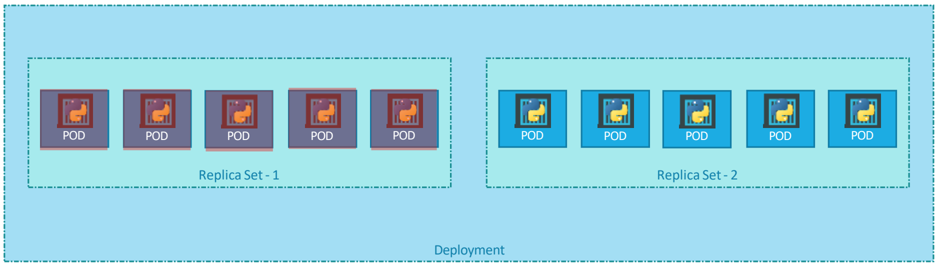
Recreate   RollingUpdate

The difference between the recreate and rollingupdate strategies can also be seen when you view the deployments in detail. Run the kubectl describe deployment command to see detailed information regarding the deployments. You will notice when the Recreate strategy was used the events indicate that the old replicaset was scaled down to 0 first and the new replica set scaled up to 5. However when the RollingUpdate 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.



MUMSHAD MANNAMBETH

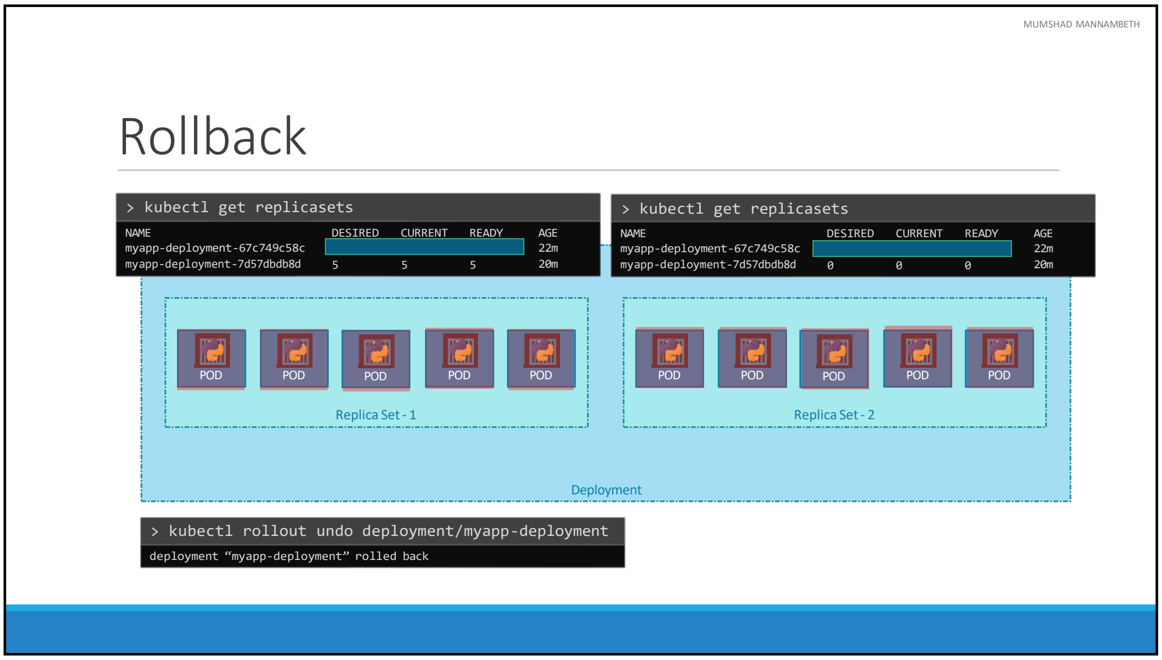
Upgrades



```
> kubectl get replicasets
NAME                       DESIRED   CURRENT   READY     AGE
myapp-deployment-67c749c58c   0         0         0         22m
myapp-deployment-7d57dbdb8d   5         5         5         20m
```

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 you upgrade your application as we saw in the previous slide, 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 RollingUpdate strategy.

This can be seen when you try to list the replicasets using the kubectl get replicasets command. Here we see the old replicaset with 0 PODs and the new replicaset with 5 PODs.



MUMSHAD MANNAMBETH

Rollback

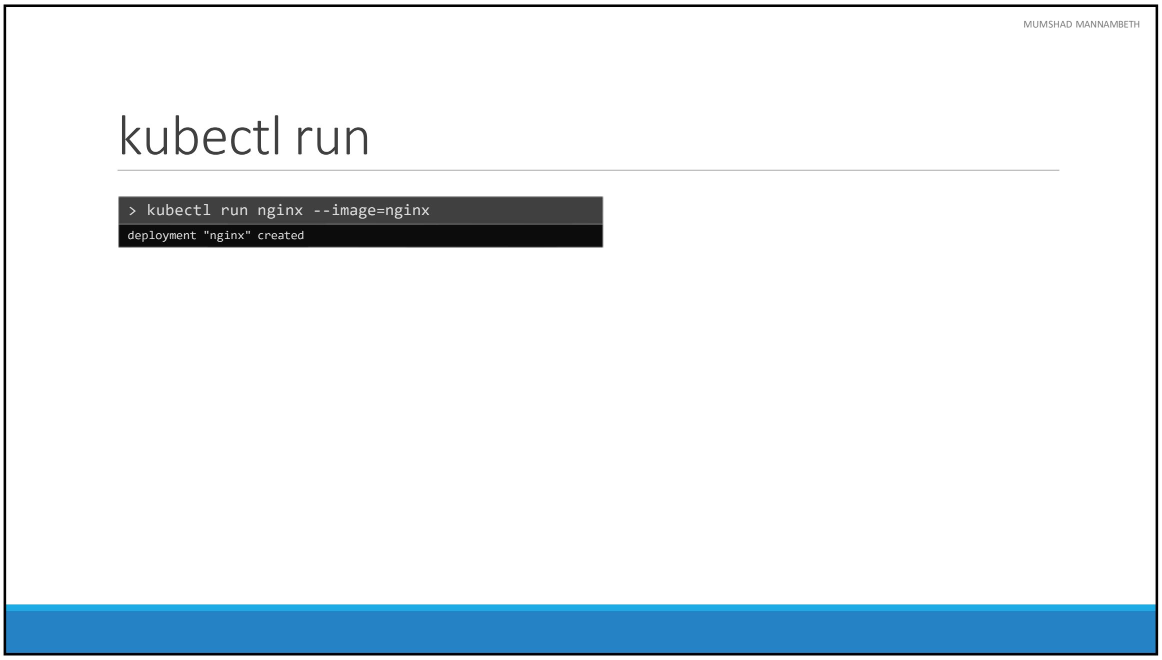
```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c                               22m
myapp-deployment-7d57dbdb8d   5         5         5       20m
```

```
> kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
myapp-deployment-67c749c58c                               22m
myapp-deployment-7d57dbdb8d   0         0         0       20m
```

```
> kubectl rollout undo deployment/myapp-deployment
deployment "myapp-deployment" rolled back
```

Say for instance once you upgrade your application, you realize something isn't very right. Something's wrong with the new version of build you used to upgrade. So you would like to rollback your update. Kubernetes deployments allow you to rollback to a previous revision. To undo a change run the command kubectl rollout undo followed by the name of the deployment. The deployment will then destroy the PODs in the new replicaset and bring the older ones up in the old replicaset. And your application is back to its older format.

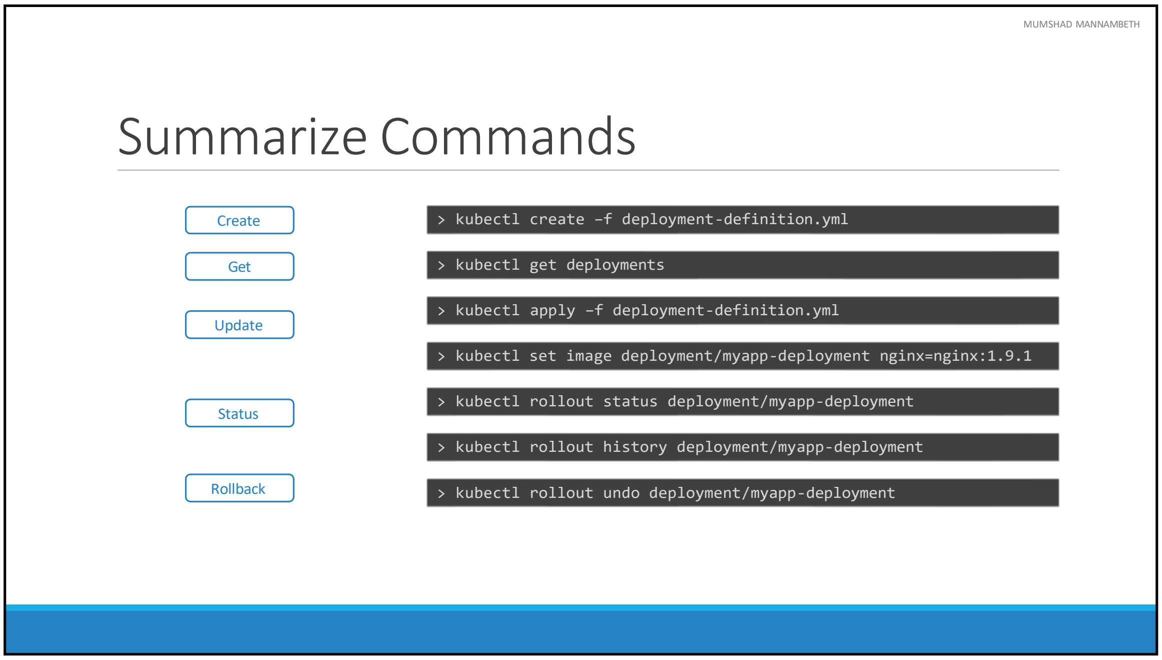
When you compare the output of the kubectl get replicasets command, before and after the rollback, you will be able to notice this difference. Before the rollback the first replicaset had 0 PODs and the new replicaset had 5 PODs and this is reversed after the rollback is finished.


MUMSHAD MANNAMBETH

kubectl run

```
> kubectl run nginx --image=nginx
deployment "nginx" created
```

And finally let's get back to one of the commands we ran initially when we learned about PODs for the first time. We used the kubectl run command to create a POD. This command infact creates a deployment and not just a POD. This is why the output of the command says Deployment nginx created. This is another way of creating a deployment by only specifying the image name and not using a definition file. A replicaset and pods are automatically created in the backend. Using a definition file is recommended though as you can save the file, check it into the code repository and modify it later as required.

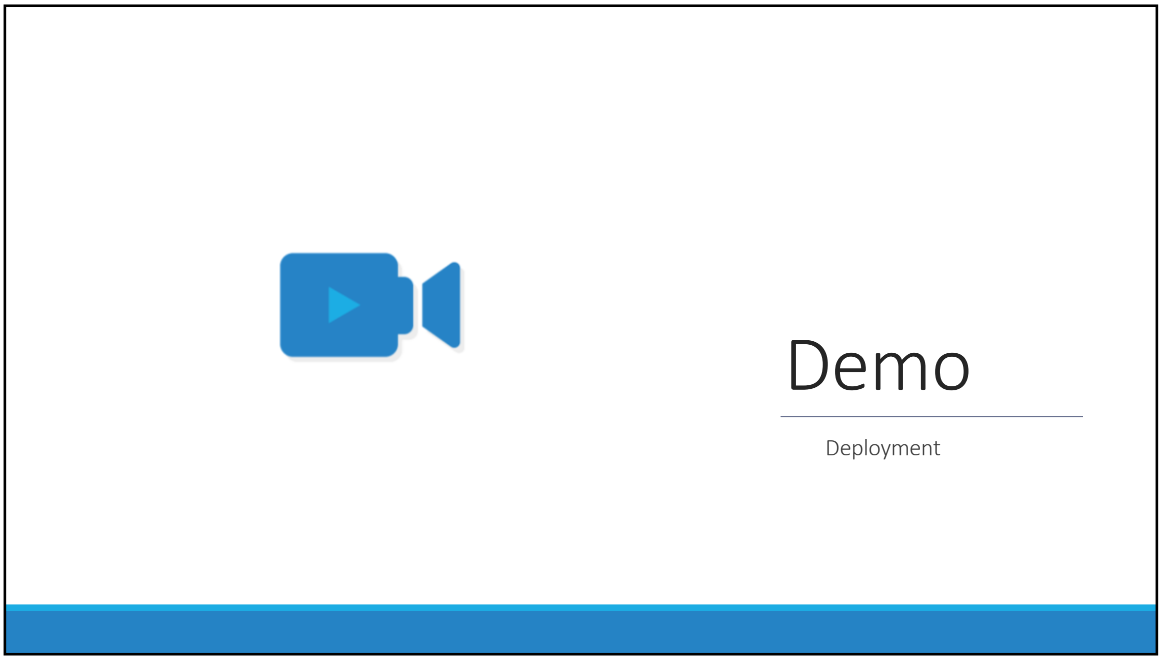

MUMSHAD MANNAMBETH

Summarize Commands

Create
Get
Update
Status
Rollback

```
> kubectl create –f deployment-definition.yml
> kubectl get deployments
> kubectl apply –f deployment-definition.yml
> kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
> kubectl rollout status deployment/myapp-deployment
> kubectl rollout history deployment/myapp-deployment
> kubectl rollout undo deployment/myapp-deployment
```

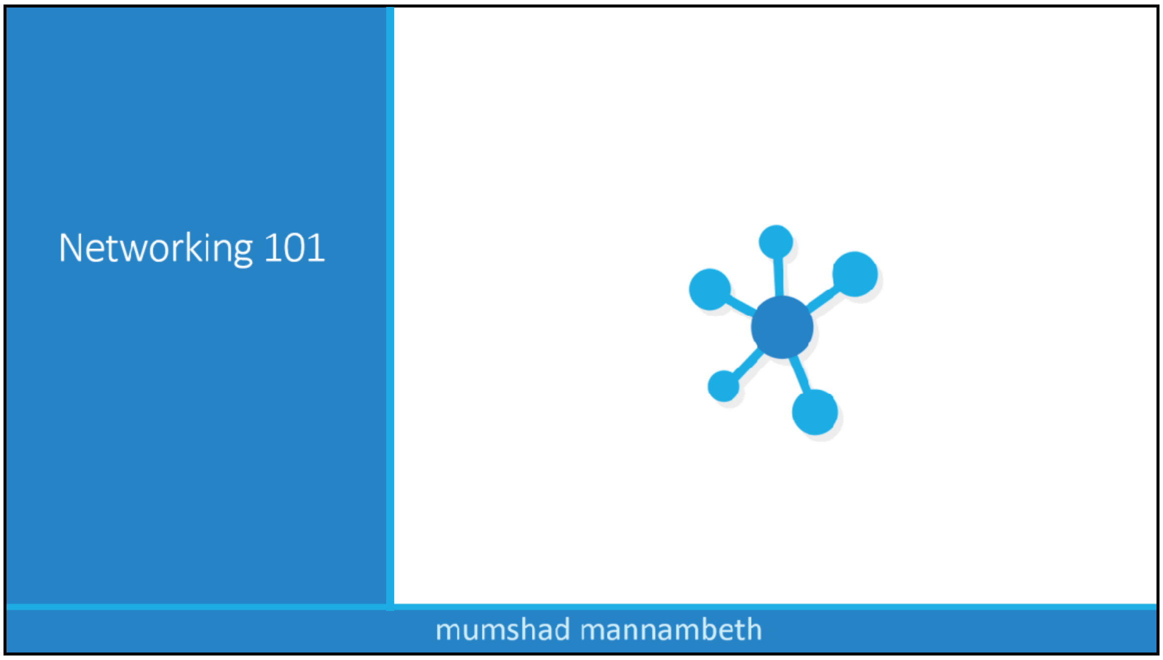
To summarize the commands real quick, use the kubectl create command to create the deployment, get deployments command to list the deployments, apply and set image commands to update the deployments, rollout status command to see the status of rollouts and rollout undo command to rollback a deployment operation.



Demo

Deployment

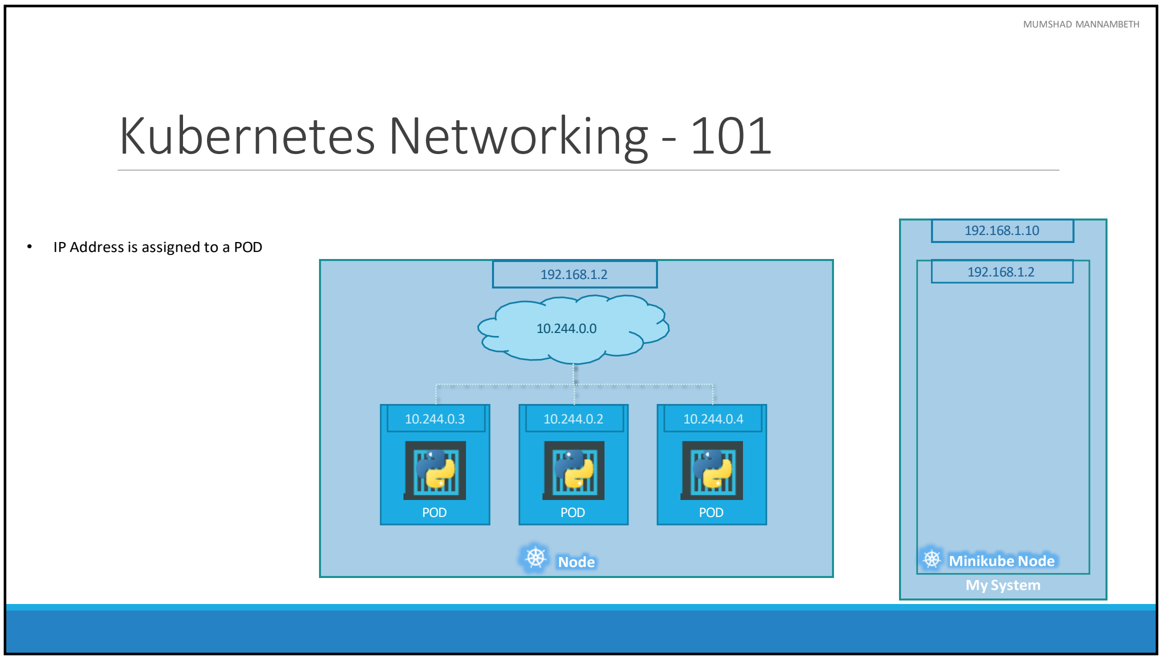
That’s it for this lecture. We will now head over to a Demo and I will see you in the next lecture.



Networking 101

mumshad mannambeth

In this lecture we will discuss about Networking in kubernetes.


MUMSHAD MANNAMBETH

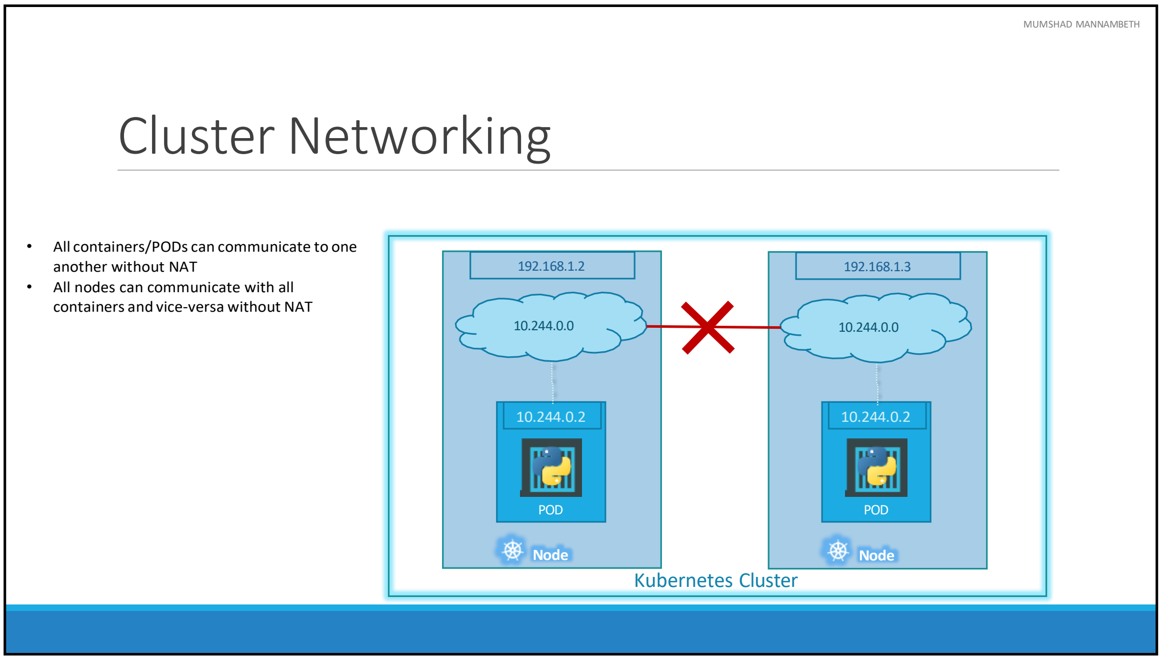
Kubernetes Networking - 101

- IP Address is assigned to a POD

Let us look at the very basics of networking in Kubernetes. We will start 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 we use to access the kubernetes node, SSH into it etc. On a side note, remember if you are using a MiniKube setup, then I am talking about the IP address of the minikube virtual machine inside your Hypervisor. Your laptop may be having a different IP like 192.168.1.10. So its important to understand how VMs are setup.

So on the single node kubernetes cluster we have created a Single POD. As you know a POD hosts a container. Unlike in the docker world were an IP address is always assigned to a Docker CONTAINER,
in Kubernetes the IP address is assigned to a POD. Each POD in kubernetes gets its own internal IP Address. In this case its in the range 10.244 series and the IP assigned to the POD is 10.244.0.2. So how is it getting this IP address? 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 you deploy multiple PODs, they all get a separate IP assigned. The PODs can communicate to each other through this IP. But accessing other PODs using this internal IP address MAY not be a good idea as its subject to change when PODs are recreated. We will see BETTER ways to establish

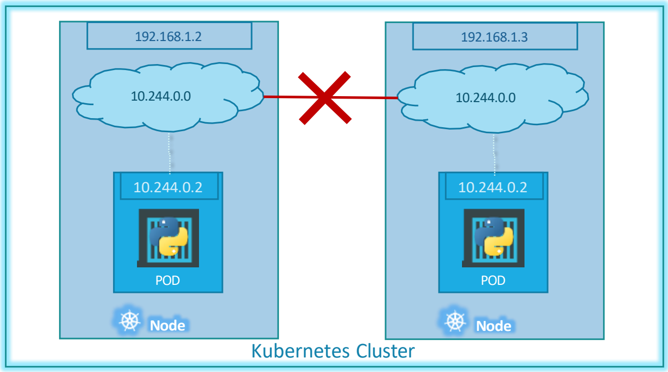
communication between PODs in a while. For now its important to understand how the internal networking works in kubernetes.



MUMSHAD MANNAMBETH

Cluster Networking

- All containers/PODs can communicate to one another without NAT
- All nodes can communicate with all containers and vice-versa without NAT

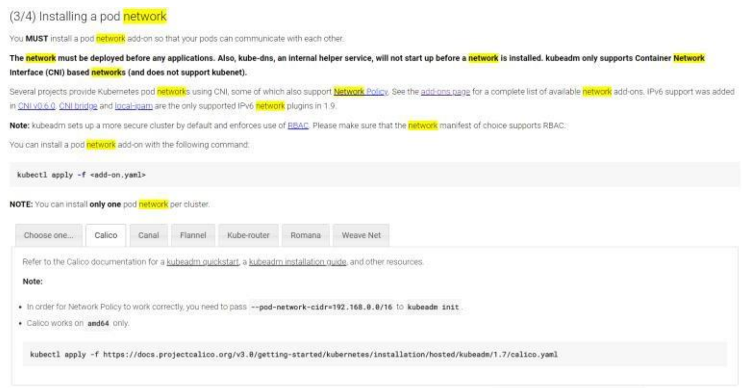


So it's all easy and simple to understand when it comes to networking on a single node. But how does it work when you have multiple nodes in a cluster? In this case we have two nodes running kubernetes and they have IP addresses 192.168.1.2 and 192.168.1.3 assigned to them. Note that they are not part 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.

Fortunately, we don’t have to set it up ALL on our own as there are multiple pre-built solutions available. Some of them are the cisco ACI networks, Cilium, Big Cloud Fabric, Flannel, Vmware NSX-t and Calico. Depending on the platform you are deploying your Kubernetes cluster on you may use any of these solutions. For example, if you were setting up a kubernetes cluster from scratch on your own systems, you may use any of these solutions like Calico, Flannel etc. If you were deploying on a Vmware environment NSX-T may be a good option. If you look at the play-with-k8s labs they use WeaveNet. In our demos in the course we used Calico. Depending on your environment and after evaluating the Pros and Cons of each of these, you may chose the right networking solution.

Cluster Networking Setup



(3/4) Installing a pod network

You **MUST** install a pod network add-on so that your pods can communicate with each other.

The network must be deployed before any applications. Also, kube-dns, an internal helper service, will not start up before a network is installed. kubeadm only supports Container Network Interface (CNI) based networks (and does not support kubenet).

Several projects provide Kubernetes pod networks using CNI, some of which also support Network Policy. See the add-ons page for a complete list of available network add-ons. IPv6 support was added in CNI v0.6.0. CNI bridge and local-ipam are the only supported IPv6 network plugins in 1.9.

Note: kubeadm sets up a more secure cluster by default and enforces use of RBAC. Please make sure that the network manifest of choice supports RBAC.

You can install a pod network add-on with the following command:

```
kubectl apply -f <add-on.yaml>
```

NOTE: You can install **only one** pod network per cluster.

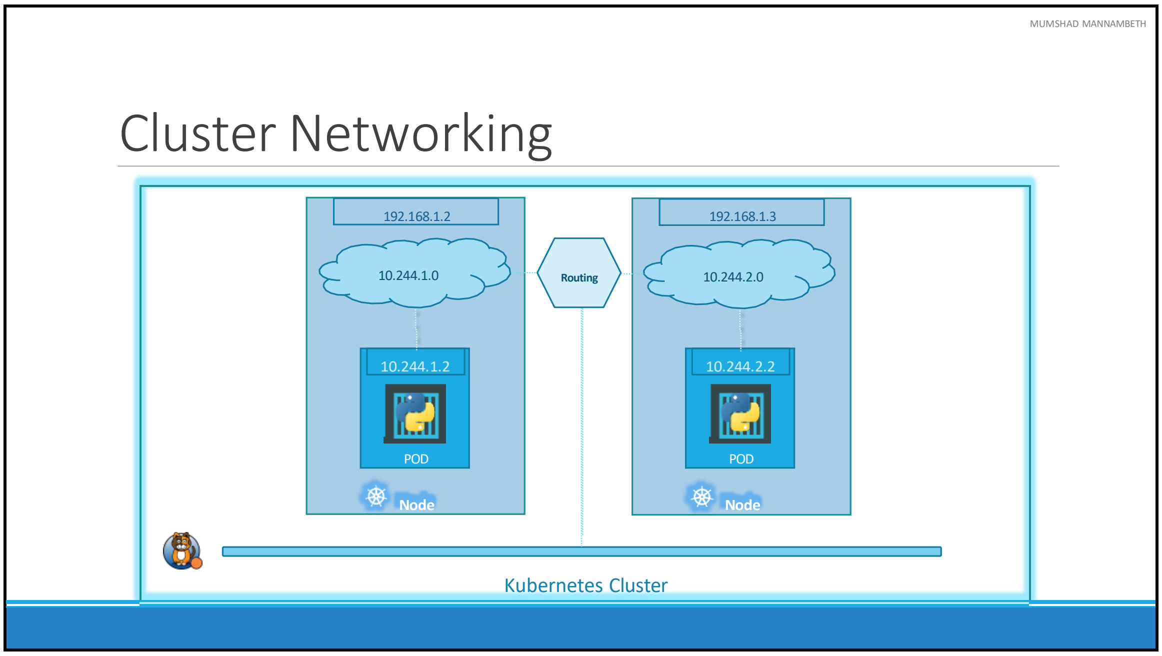
Refer to the Calico documentation for a kubeadm quickstart, a kubeadm installation guide, and other resources.

Note:

- In order for Network Policy to work correctly, you need to pass `--pod-network-cidr=192.168.0.0/16` to `kubeadm init`.
- Calico works on `amd64` only.

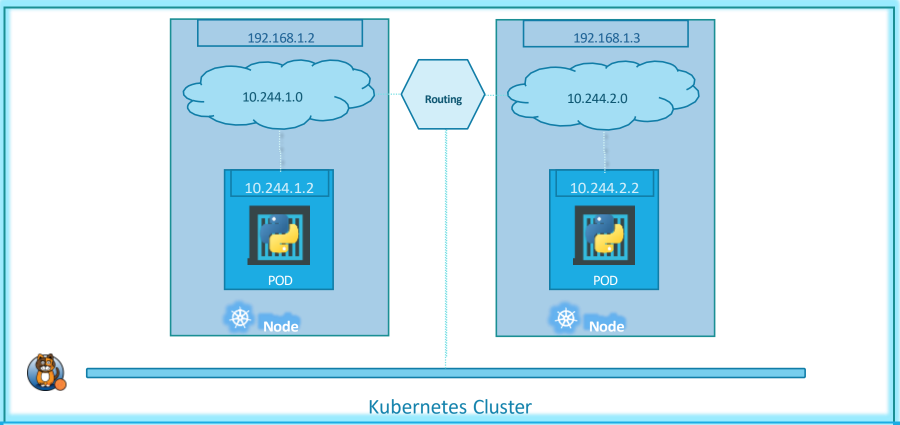
```
kubectl apply -f https://docs.projectcalico.org/v3.0/getting-started/kubernetes/installation/hosted/kubeadm/1.7/calico.yaml
```

If you go back and look at the demo were we setup a kubernetes cluster initially, we setup networking based on Calico. It only took us a few set of commands to get it setup. So its pretty easy.

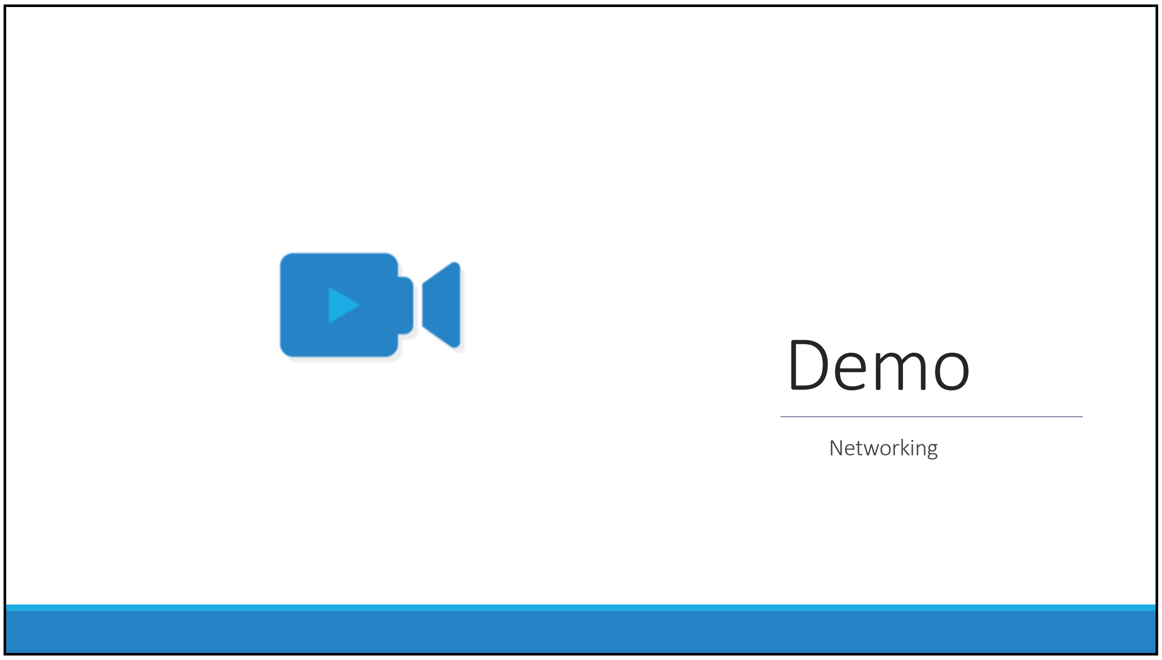


MUMSHAD MANNAMBETH

Cluster Networking



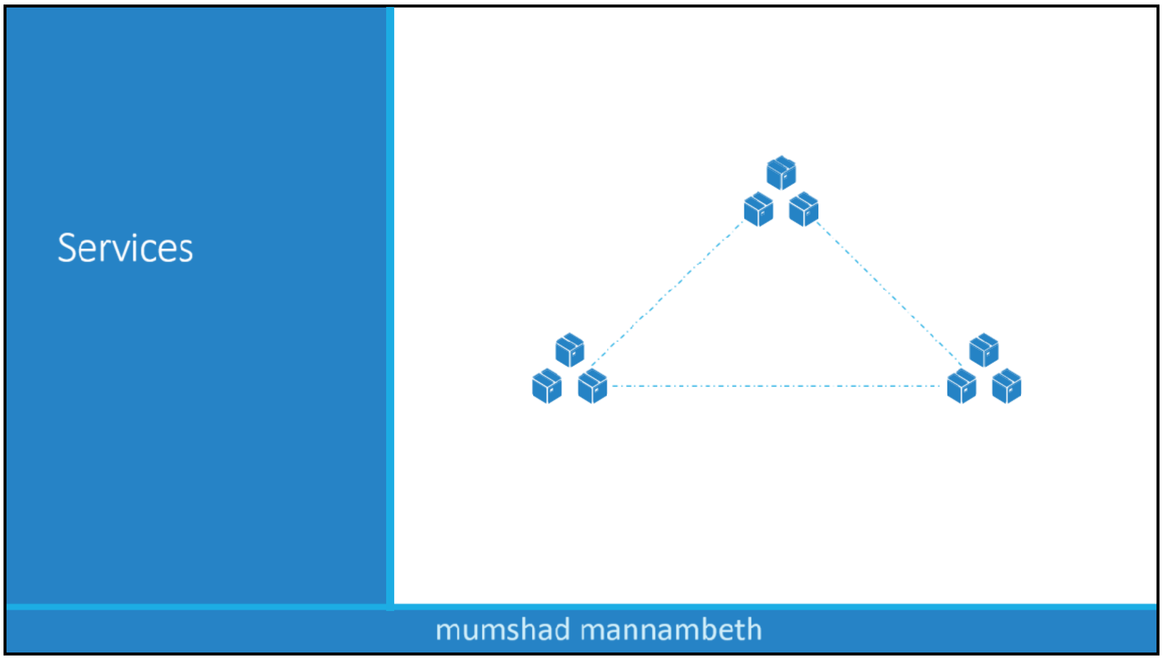
So back to our cluster, with the Calico networking setup, it now manages the networks and Ips in my nodes and assigns a different network address for each network in the nodes. This creates a virtual network of all PODs and nodes were they are all assigned a unique IP Address. And by using simple routing techniques the cluster networking enables communication between the different PODs or Nodes to meet the networking requirements of kubernetes. Thus all PODs can now communicate to each other using the assigned IP addresses.



Demo

Networking

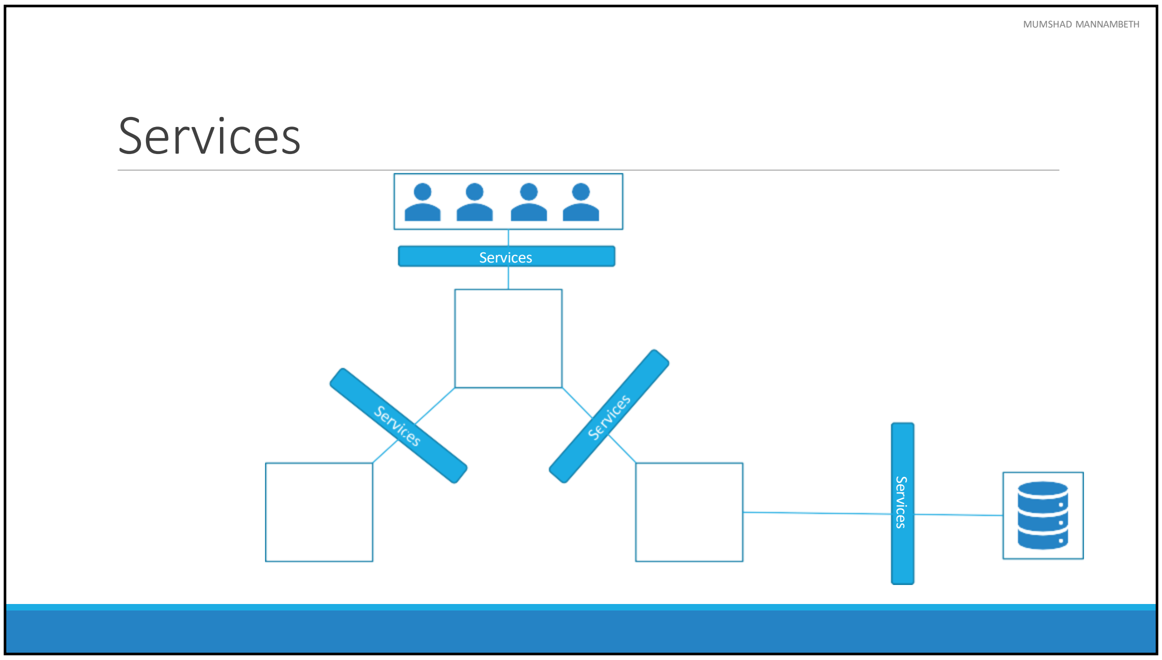
That's it for this lecture. We will now head over to a Demo and I will see you in the next lecture.



Services

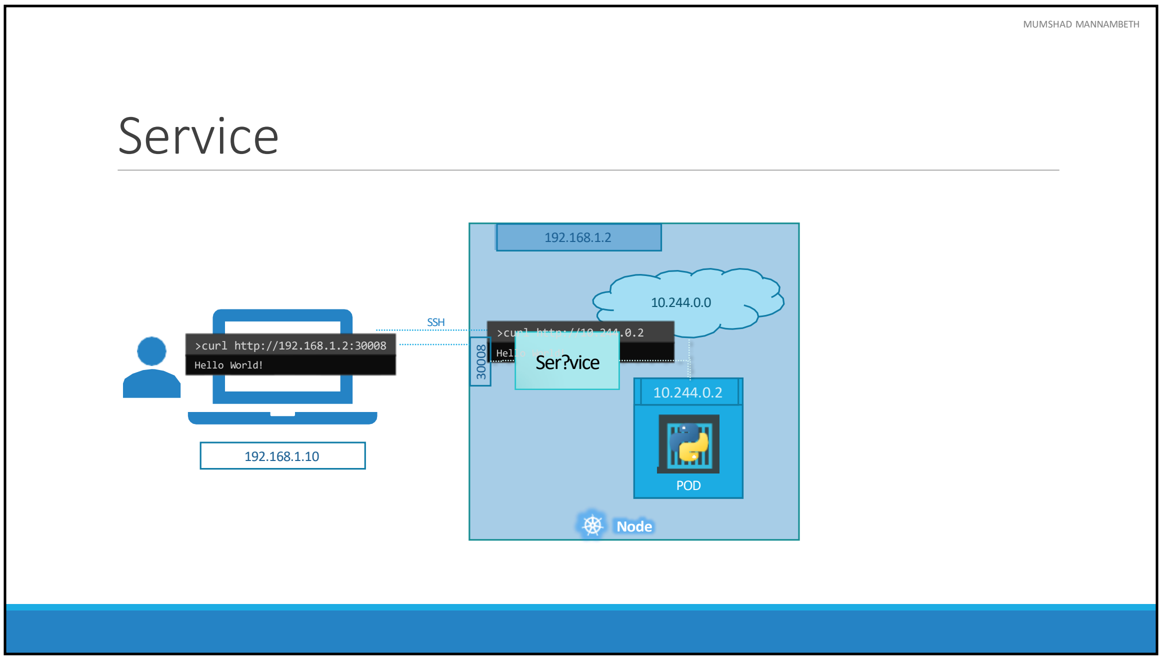
mumshad mannambeth

In this lecture we will discuss about Kubernetes Services.

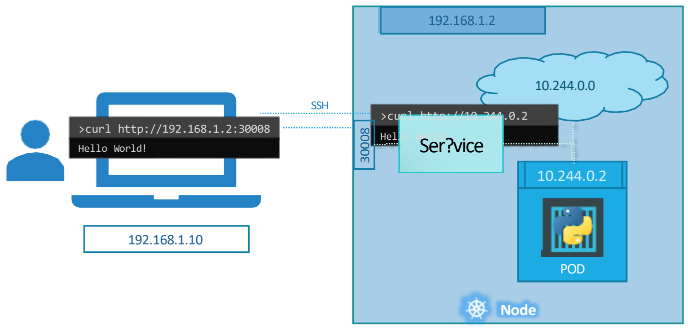

MUMSHAD MANNAMBETH

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



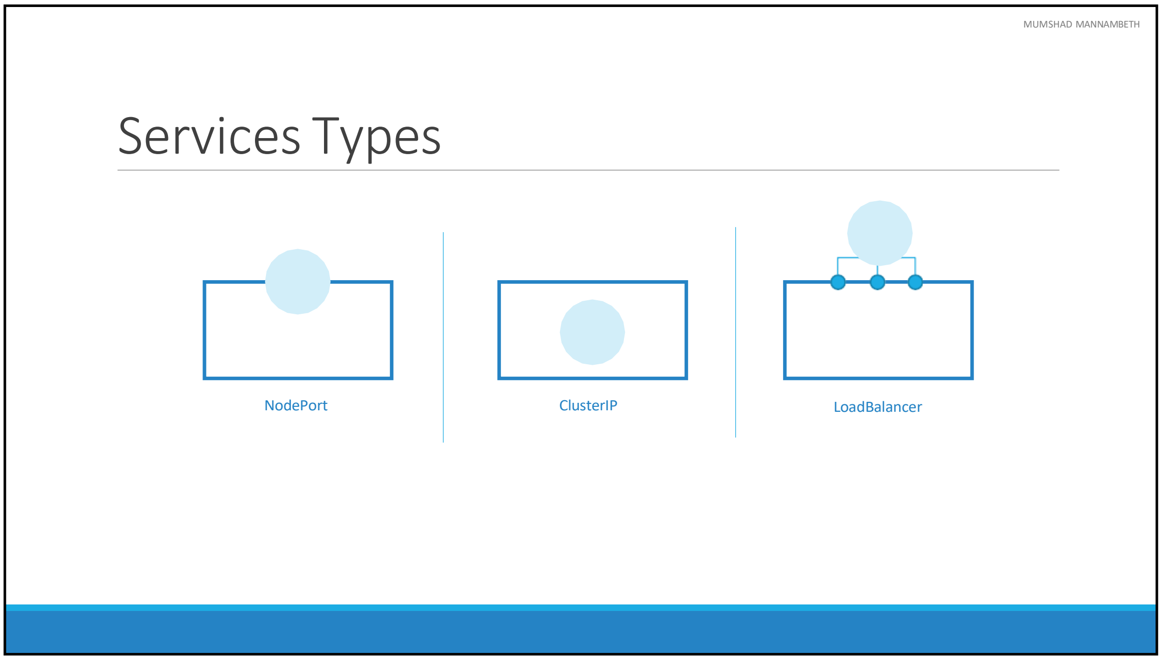
```
>curl http://192.168.1.2:30008
Hello World!
```

```
>curl http://10.244.0.2
Hel
```

Let’s take a look at one use case of Services. So far we talked about how PODs communicate with each other through internal networking. Let’s look at some other aspects of networking in this lecture. Let’s start with external communication. So we deployed our POD having a web application running on it. How do WE as an external user access the web page? First of all, lets 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, I cannot ping or access the POD at address 10.244.0.2 as its in a separate network. So what are the options to see the webpage?

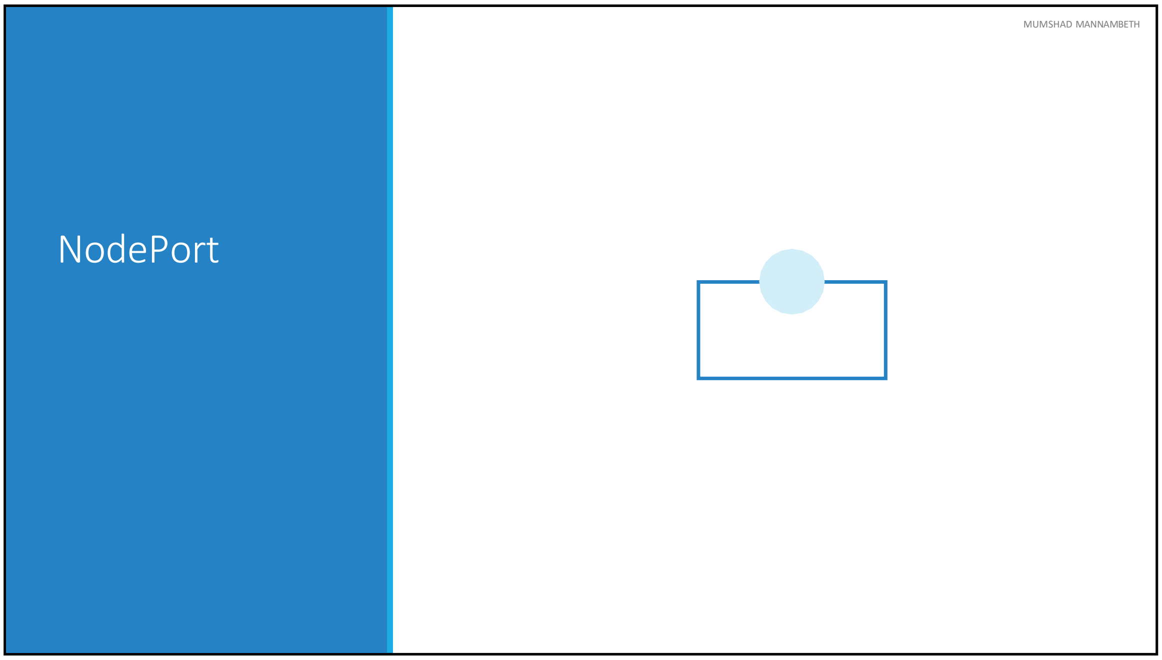
First, if we were to SSH into the kubernetes node at 192.168.1.2, from the node, we would be able to access the POD’s webpage by doing a curl or if the node has a GUI, we could fire up a browser and see the webpage in a browser following the address http://10.244.0.2. But this is from inside the kubernetes Node and that’s not what I really want. I 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, Replicaset or Deployments that we worked with before. One of its use case 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. There are other kinds of services available which we will now discuss.



Services Types

The first one is what we discussed already – NodePort were the service makes an internal POD accessible on a Port on the Node. The second is ClusterIP – and in this case the 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. The third type is a LoadBalancer, were 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.

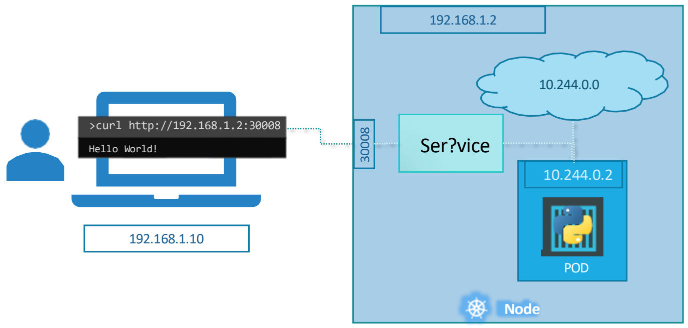


MUMSHAD MANNAMBETH

NodePort

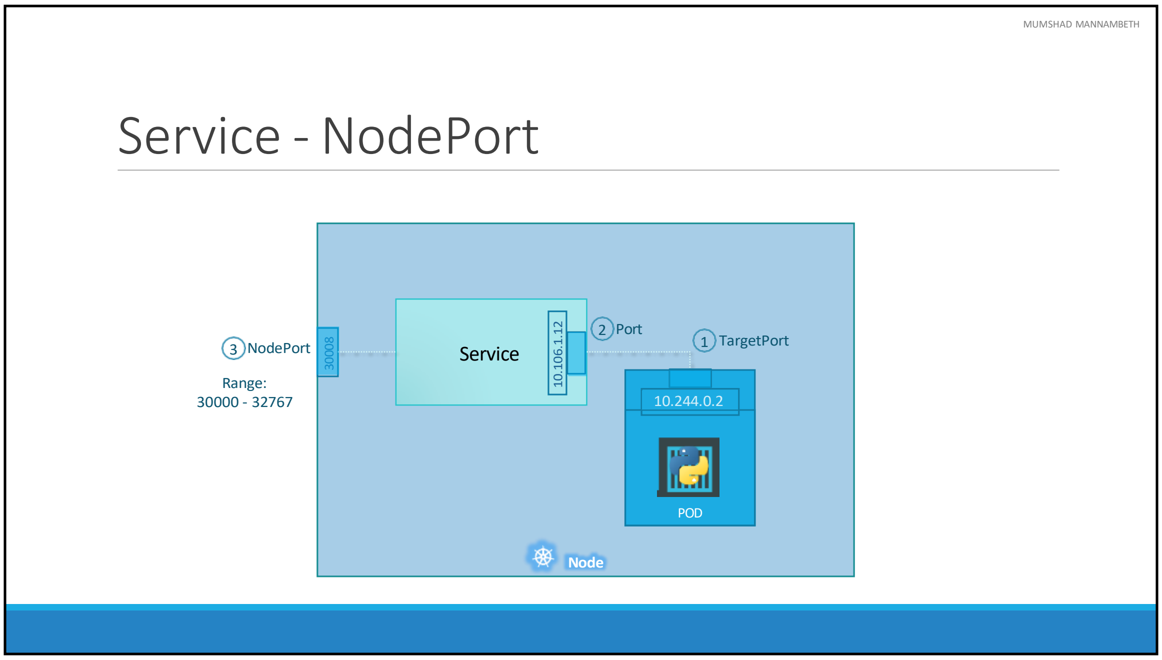
In this lecture we will discuss about the NodePort Kubernetes Service.

Service - NodePort

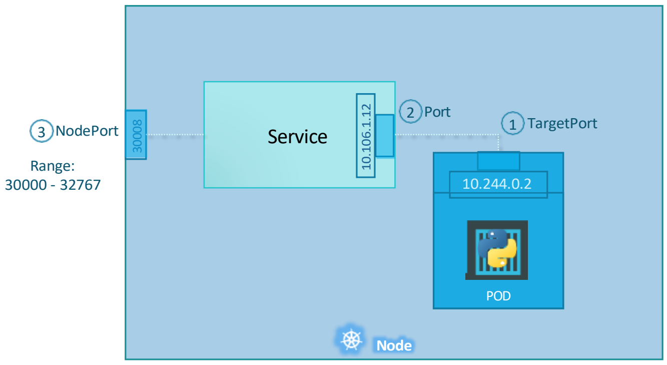


```
>curl http://192.168.1.2:30008
Hello World!
```

Getting back to NodePort, few slides back we discussed about external access to the application. We said that a Service can help us by mapping a port on the Node to a port on the POD.


MUMSHAD MANNAMBETH

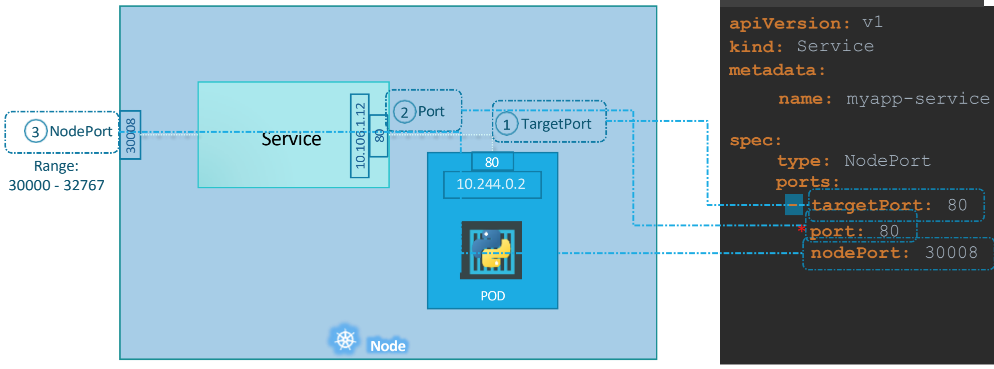
Service - NodePort



Let’s take a closer look at the Service. If you look at it, there are 3 ports involved. The port on the POD were the actual web server is running is port 80. And it is referred to as the targetPort, because that is were the service forwards the requests to. The second port is the port on the service itself. It is simply referred to as the port. Remember, 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. And finally we have the port on the Node itself which we use to access the web server externally. And that is known as the NodePort. As you can see it is 30008. That is because NodePorts can only be in a valid range which is from 30000 to 32767.

MUMSHAD MANNAMBETH

Service - NodePort



service-definition.yml

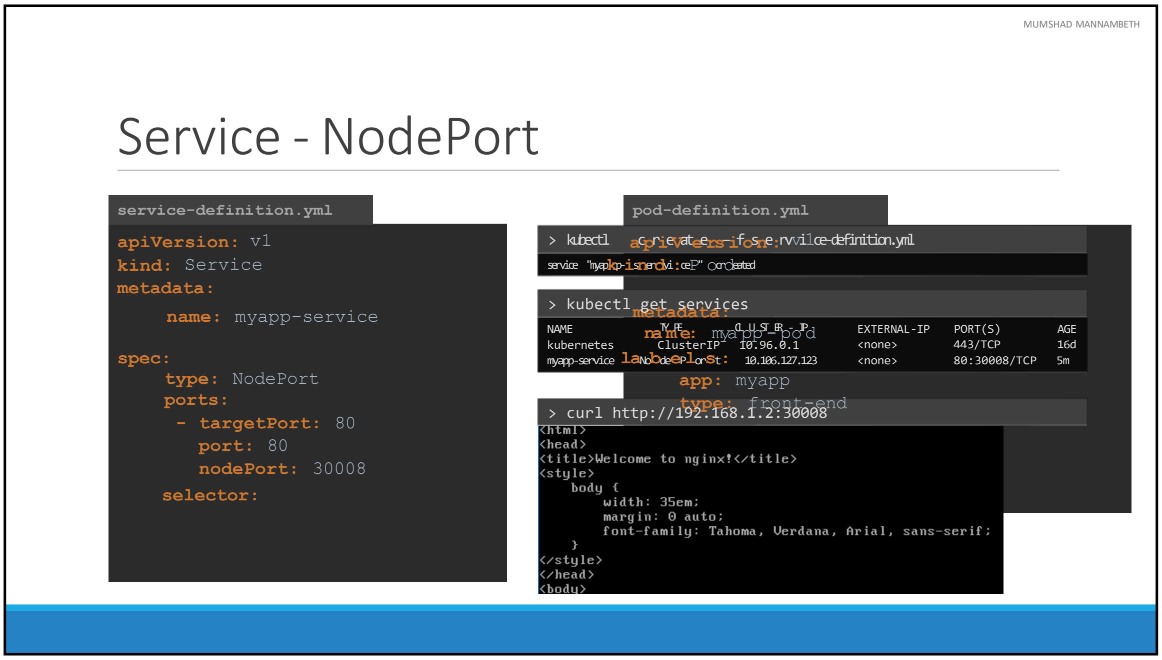
```yaml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

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

Let us now look at how to create the service. Just like how we created a Deployment, ReplicaSet or Pod, we will use a definition file to create a service.
The high level structure of the file remains the same. As before we have apiVersion, kind, metadata and spec sections. The apiVersion is going to be v1. The kind is ofcourse service. The metadata will have a name and that will be the name of the service. It can have labels, but we don't need that for now. Next we have spec. and as always this is the most crucial part of the file as this is were we will be defining the actual services and this is the part of a definition file that differs between different objects. In the spec section of a service we have type and ports. The type refers to the type of service we are creating. As discussed before it could be ClusterIP, NodePort, or LoadBalancer. In this case since we are creating a NodePort we will set it as NodePort. The next part of spec is ports. This is were we input information regarding what we discussed on the left side of this screen. 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. Remember that out of these, the only mandatory field is port . If you don't provide a targetPort it is assumed to be the same as port and if you don't provide a nodePort a free port in the valid range between 30000 and 32767 is automatically allocated. Also note that ports is an array.

So note the dash under the ports section that indicate the first element in the array. You can have multiple such port mappings within a single service.

So we have all the information in, but something is really missing. There is nothing here in the definition file that connects the service to the POD. We have simply specified the targetPort but we didn’t mention the targetPort on which POD. There could be 100s of other PODs with web services running on port 80. So how do we do that?

As we did with the replicasets previously and a technique that you will see very often in kubernetes, we will 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.



MUMSHAD MANNAMBETH

Service - NodePort

service-definition.yml

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

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

pod-definition.yml

```
apiVersion:
kind:
metadata:
  name: myapp-pod
  labels:
     app: myapp
     type: front-end
```

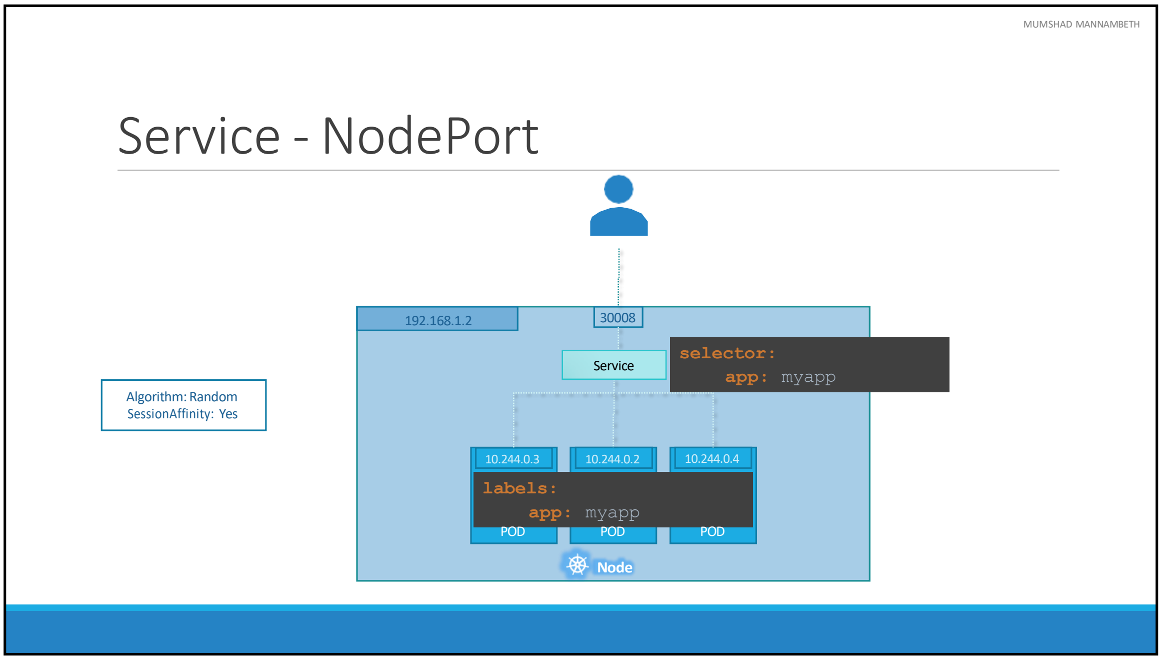
```
> kubectl create -f service-definition.yml
service "myapp-service" created
```

```
> 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>
```

So we have a new property in the spec section and that is selector. 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. Once done create the service using the kubectl create command and input the service-definition file and there you have the service created.

To see the created service, run the kubectl get services command that lists the services, their cluster-ip and the mapped ports. The type is NodePort as we created and the port on the node automatically assigned is 32432. We can now use this port to access the web service using curl or a web browser.



MUMSHAD MANNAMBETH

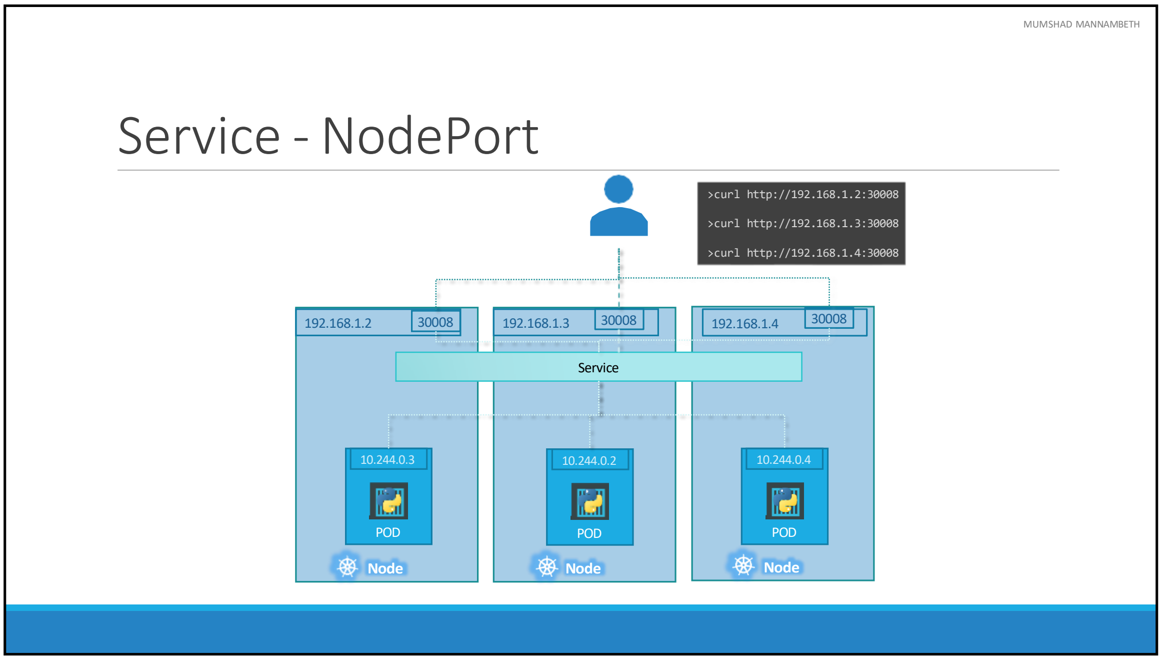
Service - NodePort

```
selector:
    app: myapp
```

```
labels:
    app: myapp
```

So far we talked about a service mapped to a single POD. But that’s not the case all the time, what do you do when you have multiple PODs? In a production environment you have multiple instances of your 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. You don’t have to do any additional configuration to make this happen. And if you are wondering what algorithm it uses to balance load, it uses a random algorithm. Thus the service acts as a built-in load balancer to distribute load across different PODs.



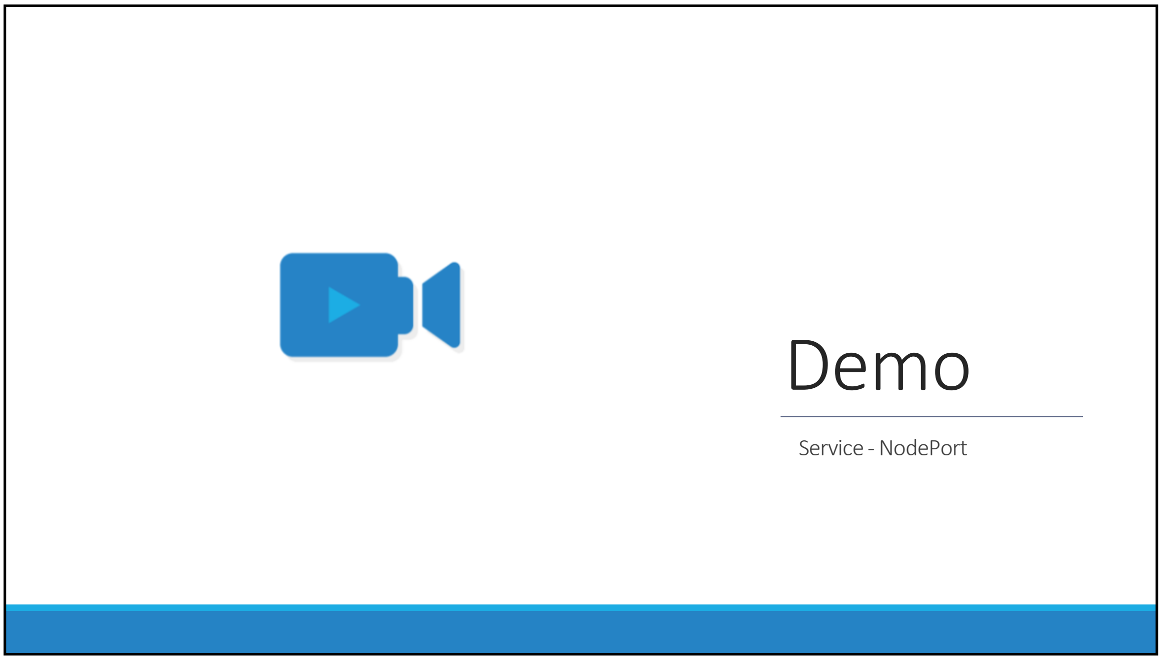
MUMSHAD MANNAMBETH

Service - NodePort

```
>curl http://192.168.1.2:30008
>curl http://192.168.1.3:30008
>curl http://192.168.1.4:30008
```

And finally, lets look at what happens 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 you can access your application using the IP of any node in the cluster and using the same port number which in this case is 30008.

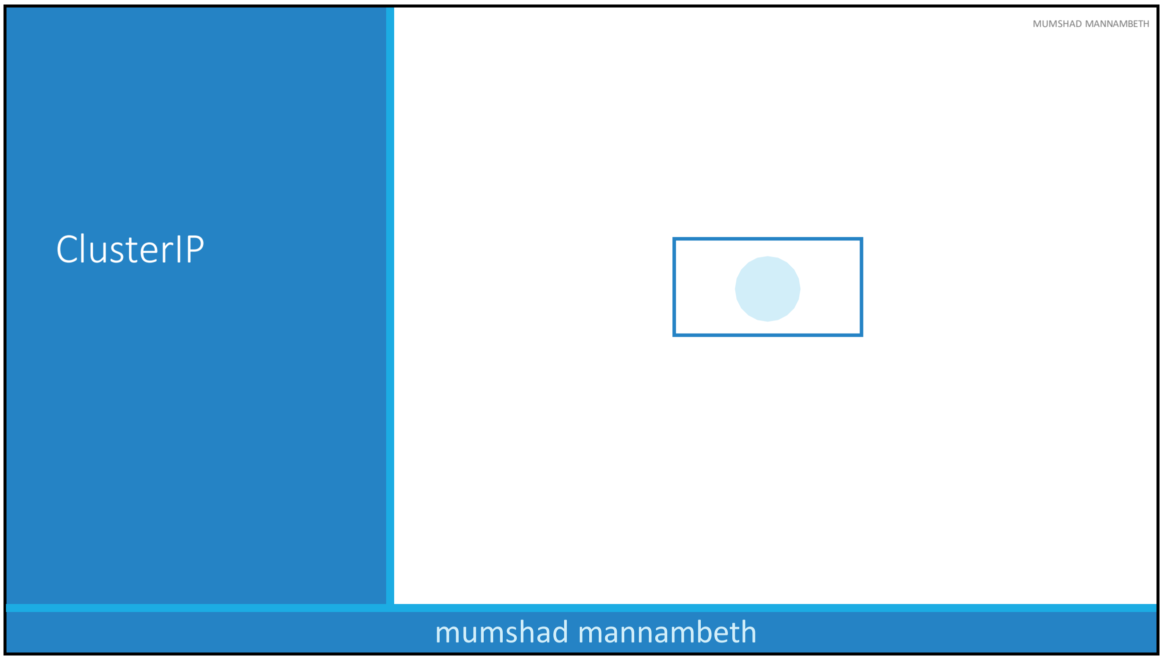
To summarize – in ANY case weather it be a single pod in a single node, multiple pods on a single node, multiple pods on multiple nodes, the service is created exactly the same without you having to do any additional steps during the service creation. When PODs are removed or added the service is automatically updated making it highly flexible and adaptive. Once created you won’t typically have to make any additional configuration changes.



Demo

Service - NodePort

That's it for this lecture, head over to the demo and I will see you in the next lecture.

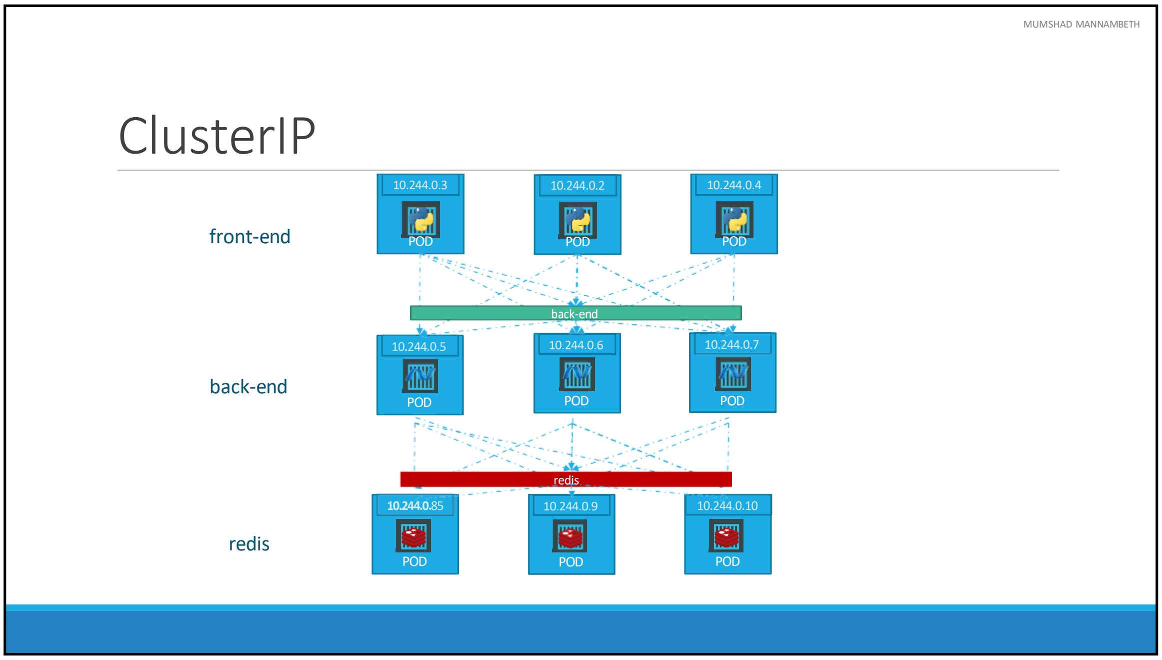


MUMSHAD MANNAMBETH

ClusterIP

mumshad mannambeth

In this lecture we will discuss about the Kubernetes Service - ClusterIP .

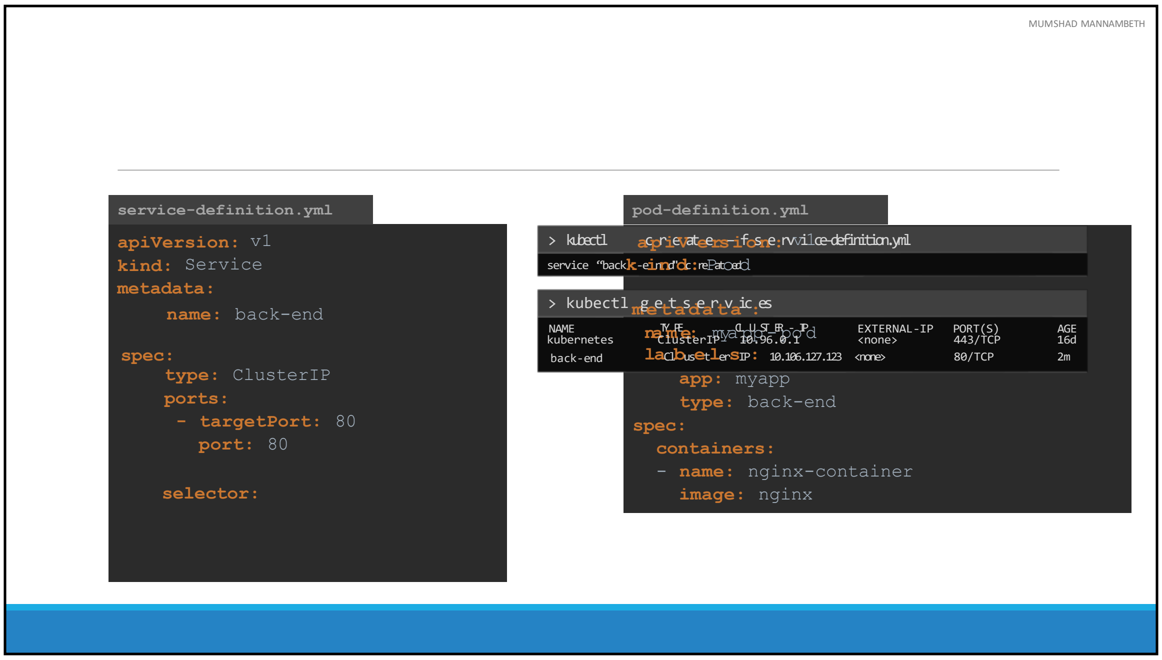

MUMSHAD MANNAMBETH

ClusterIP

A full stack web application typically has different kinds of PODs hosting different parts of an application. You may have a number of PODs running a front-end web server, another set of PODs running a backend server, a set of PODs running a key-value store like Redis, another set of PODs running a persistent database like MySQL etc. 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?

The PODs all have an IP address assigned to them as we can see on the screen. But these Ips as we know are not static, these PODs can go down anytime and new PODs are created all the time – and so you 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?

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, create 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.


MUMSHAD MANNAMBETH

service-definition.yml
```
apiVersion: v1
kind: Service
metadata:
     name: back-end

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

     selector:
```

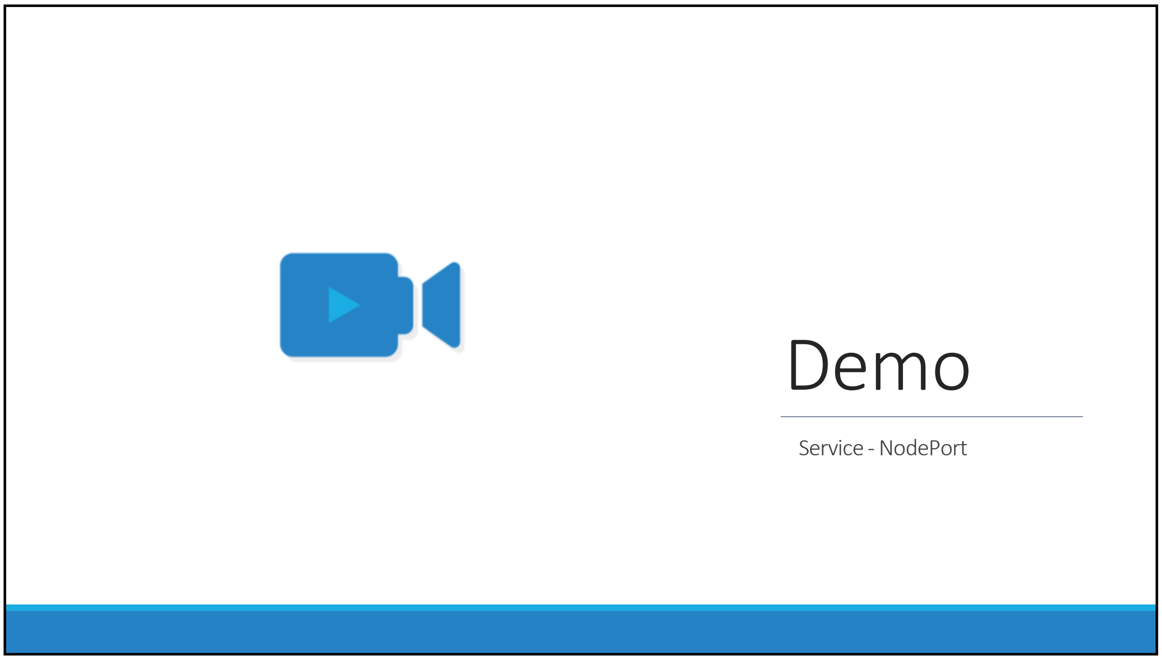
pod-definition.yml
```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
     app: myapp
     type: back-end
spec:
  containers:
  - name: nginx-container
     image: nginx
```

```
> 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
```

To create such a service, as always, use a definition file. In the service definition file , first use the default template which has apiVersion, kind, metadata and spec. The apiVersion is v1 , kind is Service and we will give a name to our service – we will call it back-end. Under Specification we have type and ports. The type is ClusterIP. In fact, ClusterIP is the default type, so even if you didn't specify it, it will automatically assume it to be ClusterIP. 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 were the service is exposed. Which is 80 as well. To link the service to a set of PODs, we use selector.

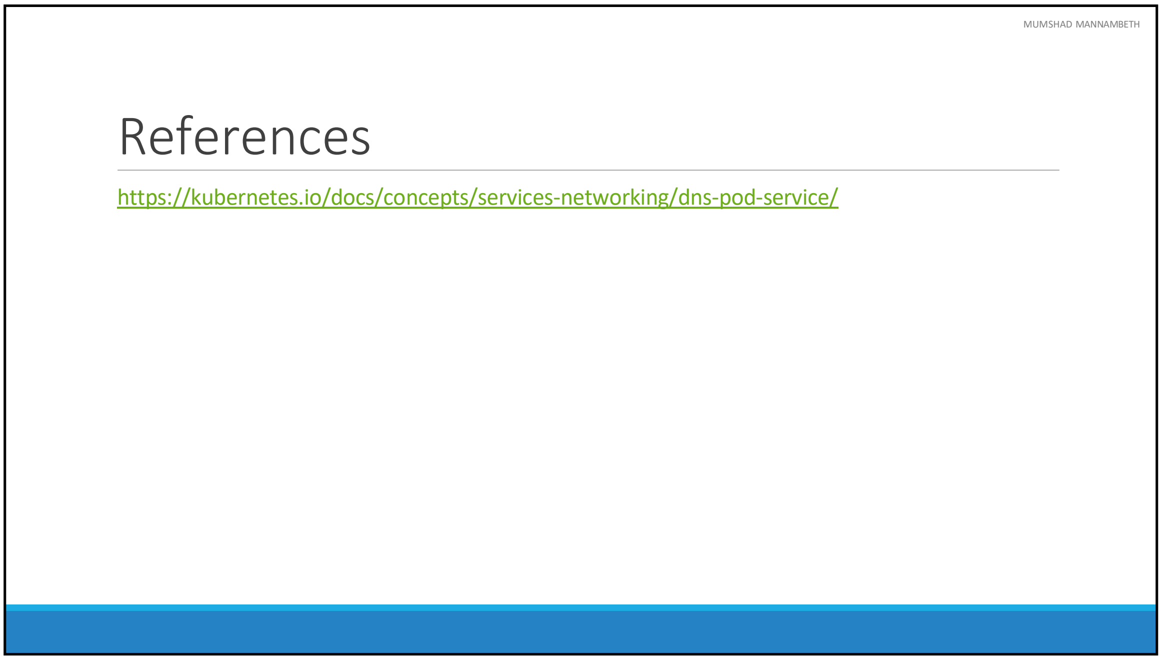
We will refer to the pod-definition file and copy the labels from it and move it under selector. And that should be it. We can now create the service using the kubectl create command and then check its status using the kubectl get services command. The service can be accessed by other PODs using the ClusterIP or the service name.



Demo

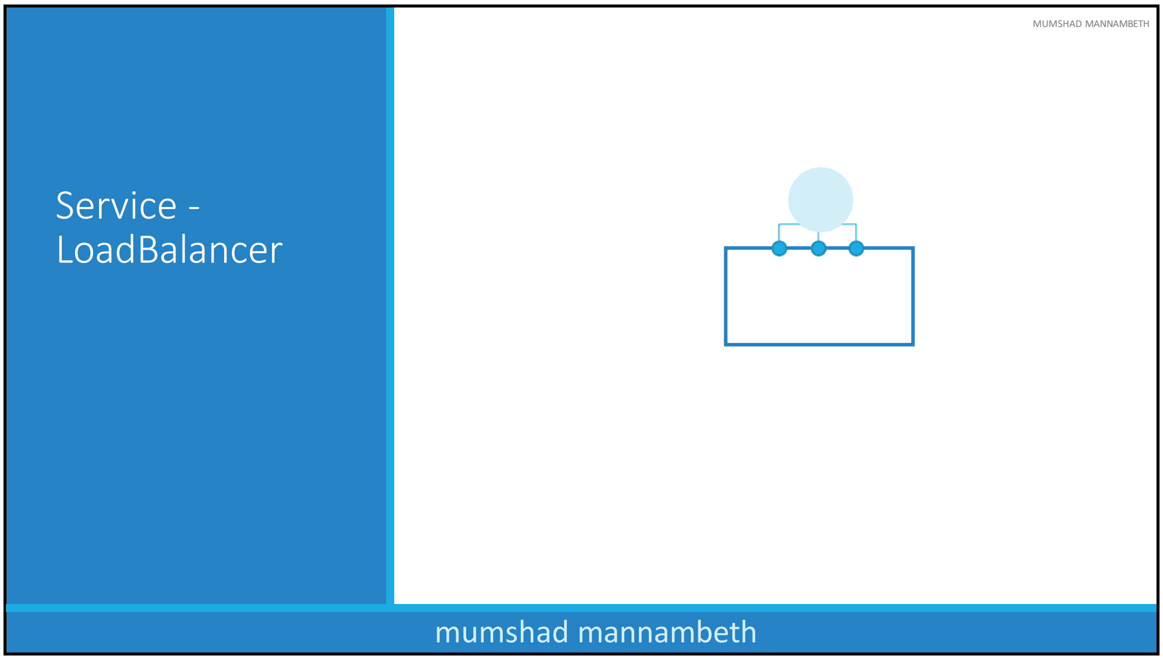
Service - NodePort

That’s it for this lecture, head over to the demo and I will see you in the next lecture.



References

https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

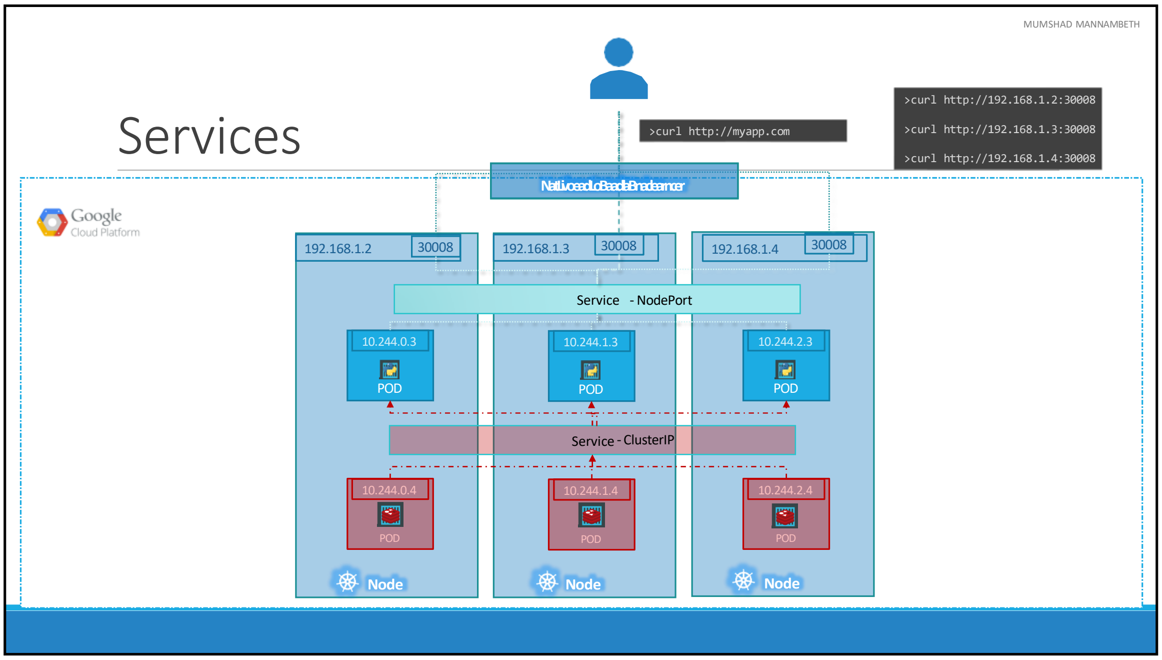


MUMSHAD MANNAMBETH

Service - LoadBalancer

mumshad mannambeth

In this lecture we will discuss about the third kind of Kubernetes Service - LoadBalancer.



MUMSHAD MANNAMBETH

Services

```
>curl http://myapp.com
```

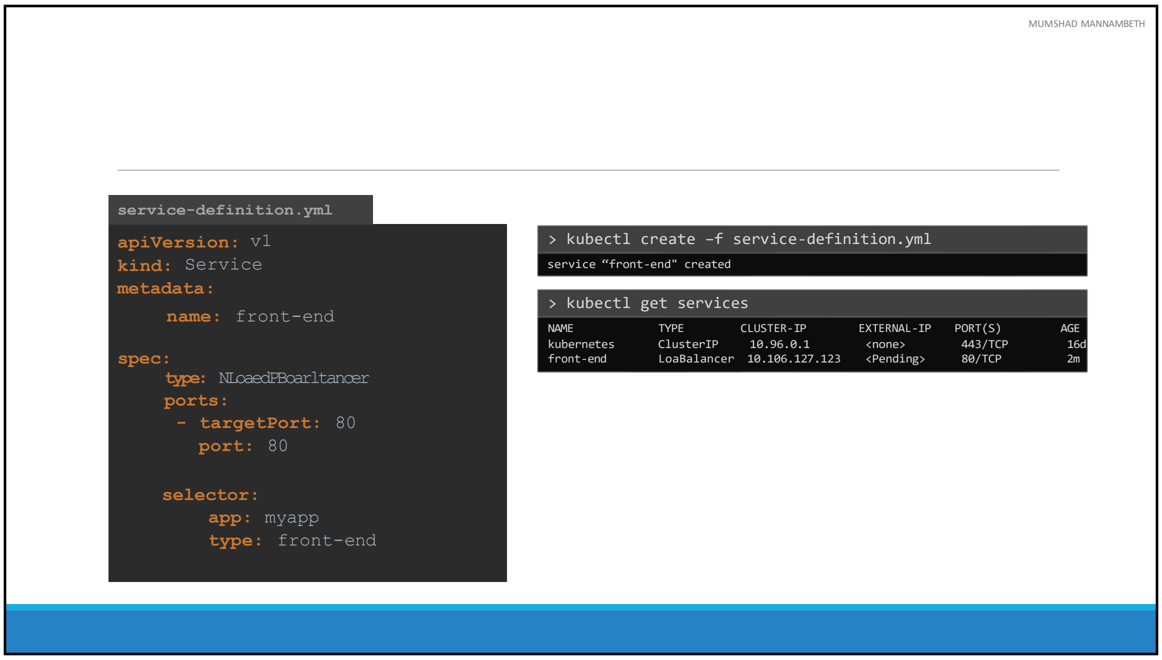
```
>curl http://192.168.1.2:30008

>curl http://192.168.1.3:30008

>curl http://192.168.1.4:30008
```

We will quickly recap what we learned about the two service types, so that we can work our way to the LoadBalancer type. We have a 3 node cluster with Ips 192.168.1.2,3 and 4. Our application is two tier, there is 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 my 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.

Now, what IP do you give your end users to access your application? You 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, you will be required to setup a separate Load Balancer VM in your environment. In this case I 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. I will then configure my organizations DNS to point to this load balancer when a user hosts http://myapp.com. Now setting up that load balancer by myself is a tedious task, and I might have to do that in my local or on-prem environment. However, if I happen to be on a supported CloudPlatform, like Google Cloud Platform, I could leverage the native load balancing functionalities of the cloud platform to set this up. Again you don't have to set that up manually, Kubernetes sets it up for you. Kubernetes has built-in integration with supported cloud platforms.



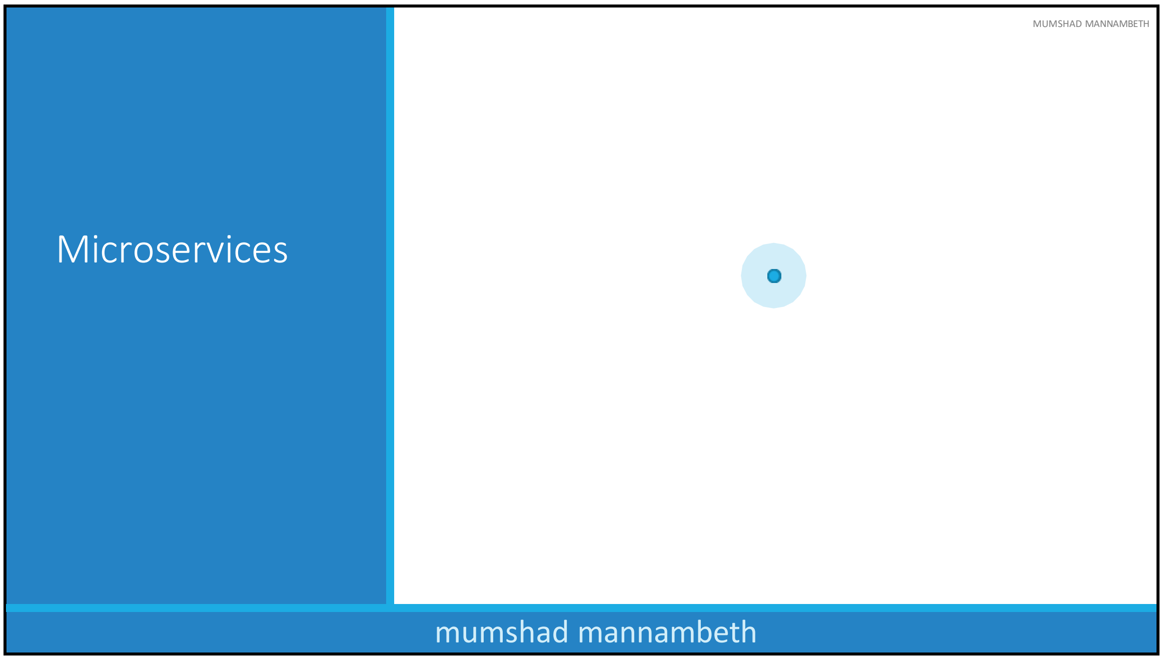
MUMSHAD MANNAMBETH

service-definition.yml

```
apiVersion: v1
kind: Service
metadata:
     name: front-end

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

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

```
> kubectl create –f service-definition.yml
service “front-end” created
```

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

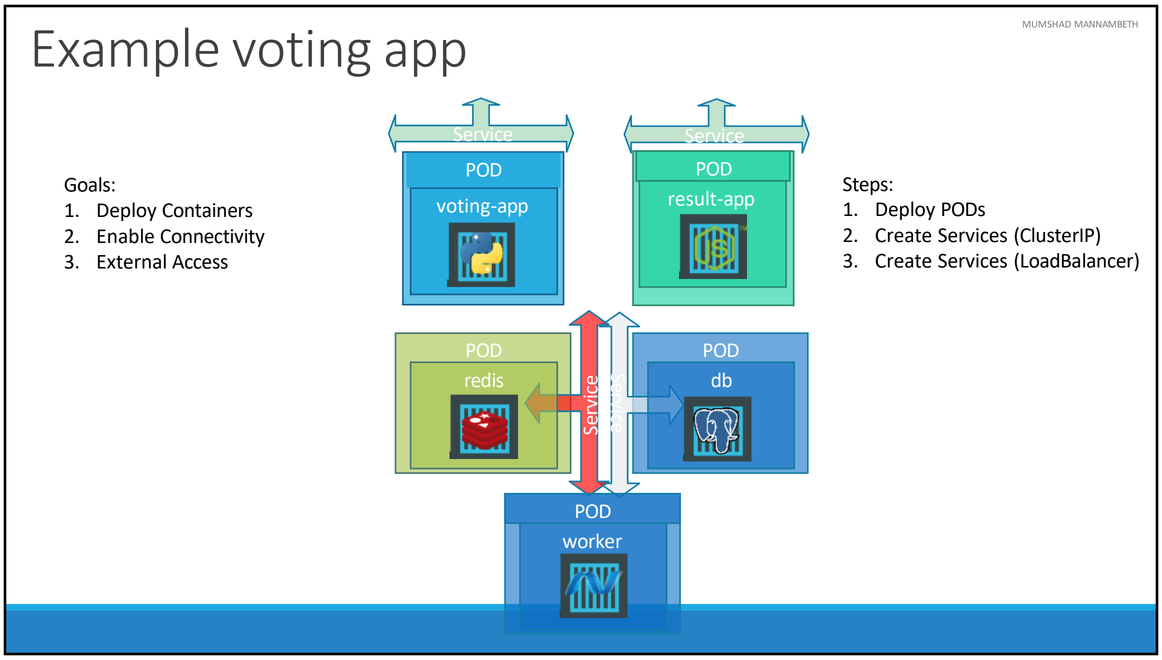


MUMSHAD MANNAMBETH

Microservices

mumshad mannambeth

In this lecture we will try and understand Microservices architecture using a simple web application. We will then try and deploy this web application on multiple different kubernetes platforms.



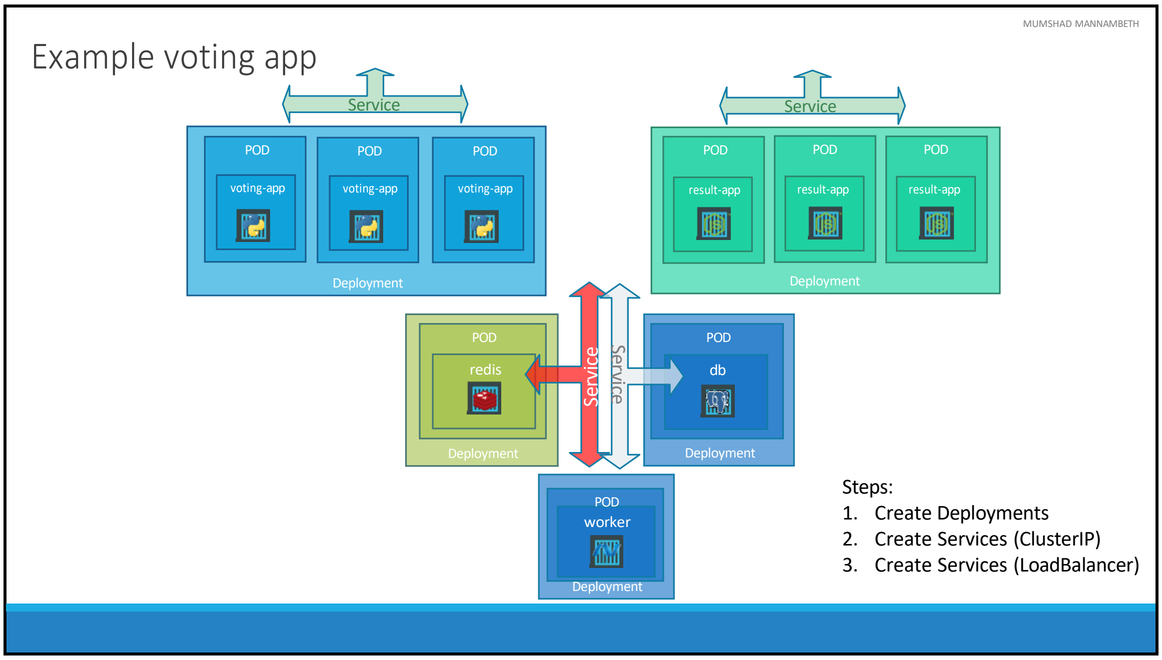
MUMSHAD MANNAMBETH

Example voting app

Goals:
1. Deploy Containers
2. Enable Connectivity
3. External Access

Steps:
1. Deploy PODs
2. Create Services (ClusterIP)
3. Create Services (LoadBalancer)

So this is what we saw in the last demo. We deployed PODs and services to keep it really simple. But this has its own challenges. Deploying PODs doesn't help us scale our application easily. Also if a POD was to fail it doesn't come back up or deploy a new POD automatically. We could deploy multiple PODs one after the other, but there are easier ways to scale using ReplicaSets and Deployments as we discussed in the lectures.

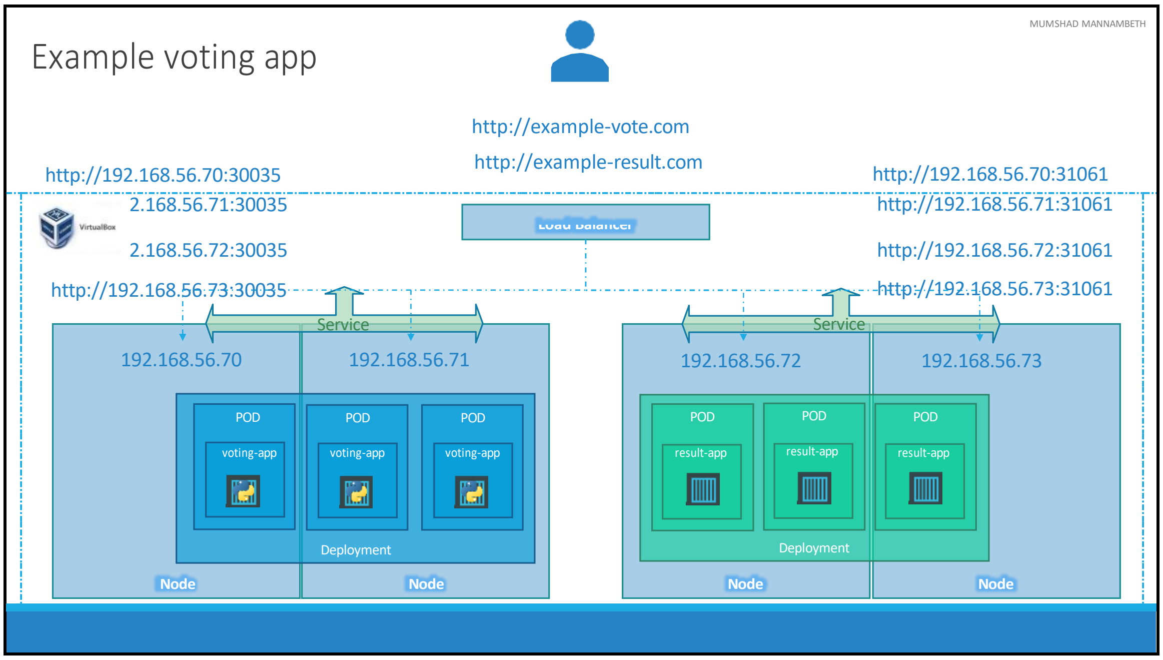

MUMSHAD MANNAMBETH

Example voting app

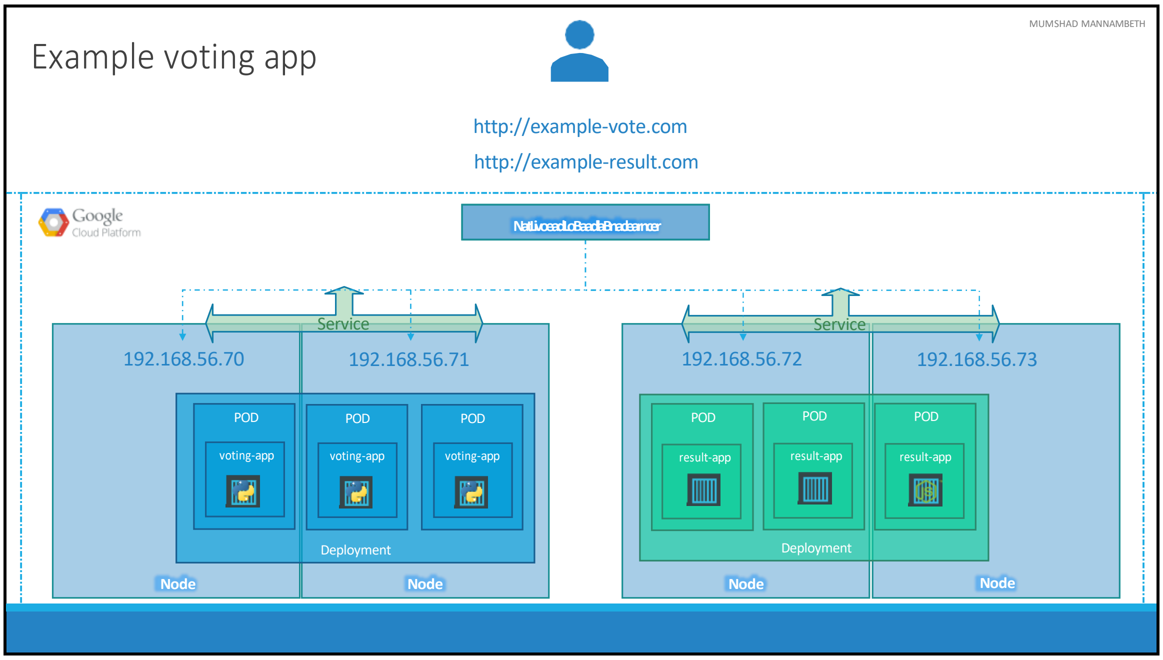
Steps:

1. Create Deployments
2. Create Services (ClusterIP)
3. Create Services (LoadBalancer)

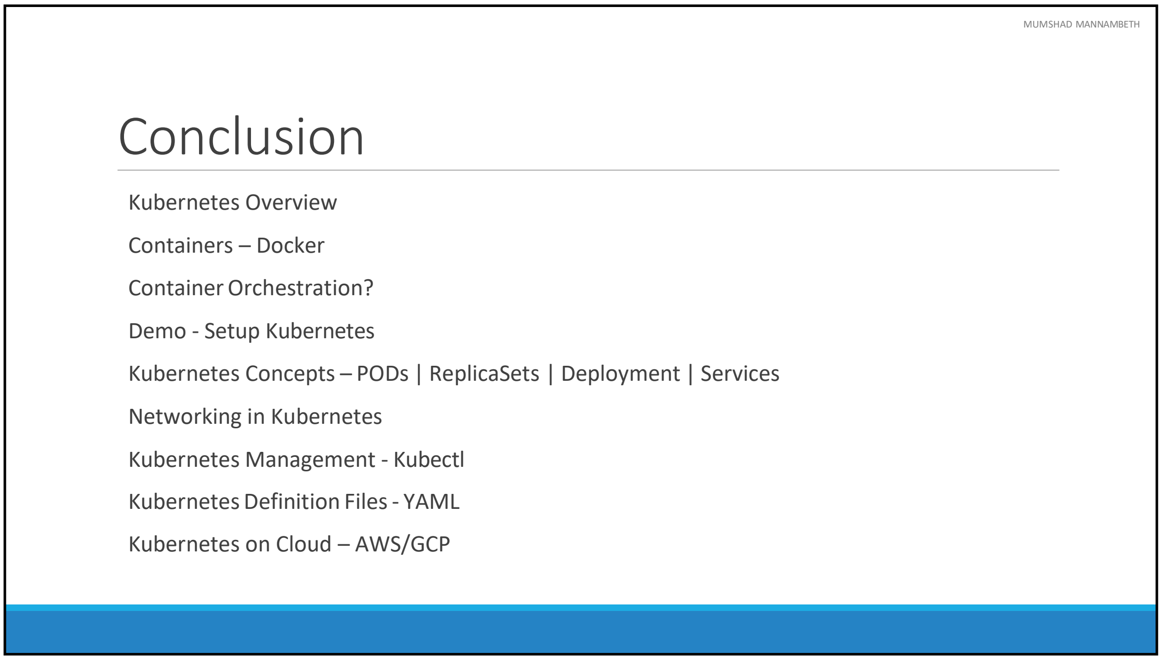
Let us now improvise our setup using Deployments. We chose deployments over ReplicaSets as Deployments automatically create replica sets as required and it can help us perform rolling updates and roll backs etc. Deployments are the way to go. So we add more PODs for the front end applications voting-app and result-app by creating a deployment with 3 replicas. We also encapsulate database and workers in deployments. Let’s take a look at that now.



Let us now focus on the front end applications – voting-app and result-app. We know that these PODs are hosted on different Nodes. The services with type NodePOrt help in receiving traffic on the ports on the Nodes and routing and load balancing them to appropriate PODs. But what IP address would you give your end users to access the applications. You could access any of these two applications using IP of any of the Nodes and the high end port the service is exposed on. That would be 4 IP and port combination for the voting-app and 4 IP and port combination for the result-app. But that’s not what the end users want. They need a single URL like example-vote.com or example-result.com. One way to achieve this in my current VirtualBox setup is to create a new VM for Load Balancer purpose and install and configure a suitable load balancer on it like HAProxy or NGINX etc. The load balancer would then re-route traffic to underlying nodes and then through to PODs to serve the users.



Now setting all of that external load balancing can be a tedious task. However, if I was on a supported cloud platform like Google Cloud, I could leverage the native load balancer to do that configuration for me. Kubernetes has support for getting that done for us automatically. All you need to do is set the Service Type for the front end services to LoadBalancer. Remember this only, works with supported cloud platforms. If you set the Type of service to LoadBalancer in an unsupported environment like VirtualBox, then it would simply be considered as if you set it to NodePOrt, were the services are exported on a high-end port on the Nodes. The same setup we had earlier. Let us take a look at how we can achieve this on Google Cloud Platform.


MUMSHAD MANNAMBETH

Conclusion

Kubernetes Overview

Containers – Docker

Container Orchestration?

Demo - Setup Kubernetes

Kubernetes Concepts – PODs | ReplicaSets | Deployment | Services

Networking in Kubernetes

Kubernetes Management - Kubectl

Kubernetes Definition Files - YAML

Kubernetes on Cloud – AWS/GCP

We are at the end of the Kubernetes for Beginners course. I hope I have covered enough topics to get you started with Kubernetes. We started with Containers and Docker and we looked what container orchestration is. We looked at various options available to setup kubernetes. We went through some of the concepts such as PODs, ReplicaSets, Deployments and Services. We also looked at Networking in Kubernetes. We also spent some time on working with kubectl commands and kubernetes definition files. I hope you got enough hands-on experience in developing kubernetes definition files. And finally we also saw how to deploy a sample microservices application on Kubernetes on Google Cloud Platform.

I will be adding additional topics to the course in the near future. So watch out for an announcement from me. In case you need me to cover any other topics, feel free to send me a message or post a question and I will try my best to add a new lecture on it.

We also have an assignment as part of this course. So if you have time, please go ahead and develop your solution and submit your assignment. Feel free to review other students assignments and share your views on them.

It was really nice having you for this course. I hope you gained enough knowledge and experience to get started with kubernetes at your work or otherwise and I wish you good luck in your Kubernetes Journey. See you in the next course, until then Good Bye!