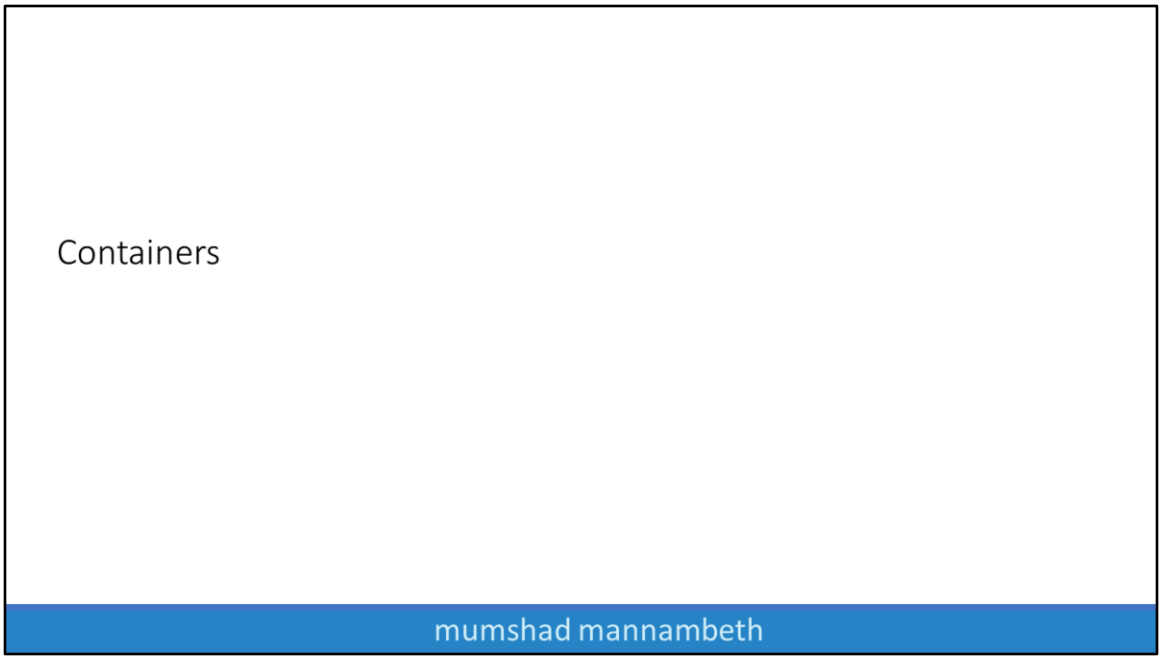
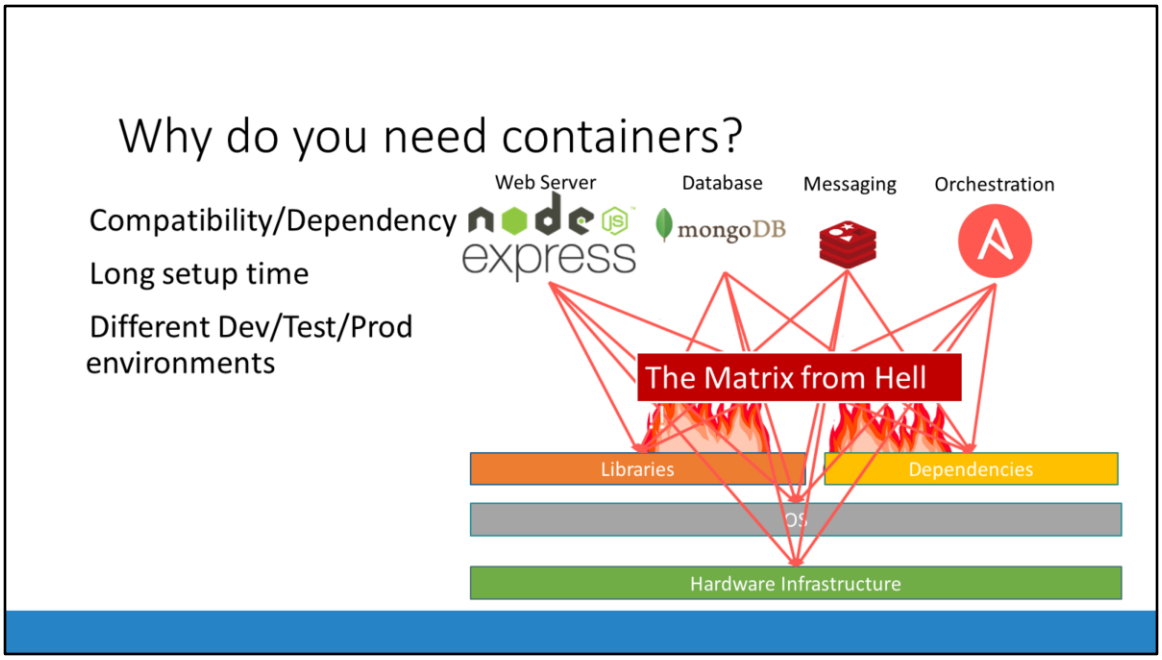

KodeKloud



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.



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.

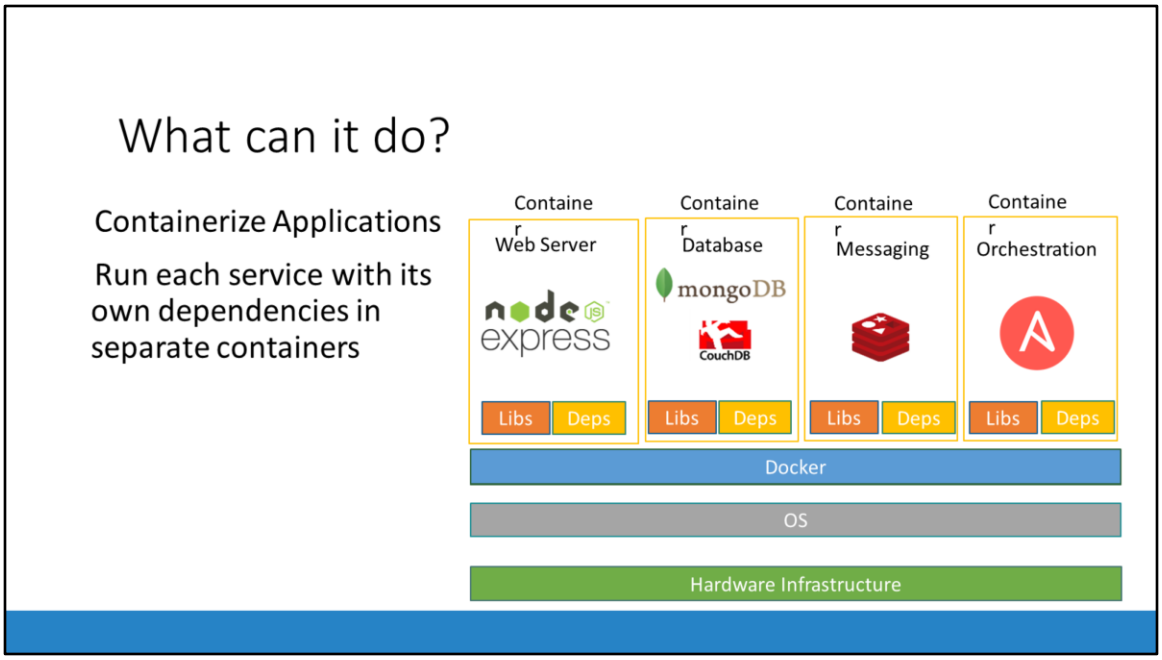
Secondly, we had to check the compatibility between these services and the libraries and dependencies on the OS. We have had issues were one service requires one version of a dependent library whereas another service required another version.

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. <click> This

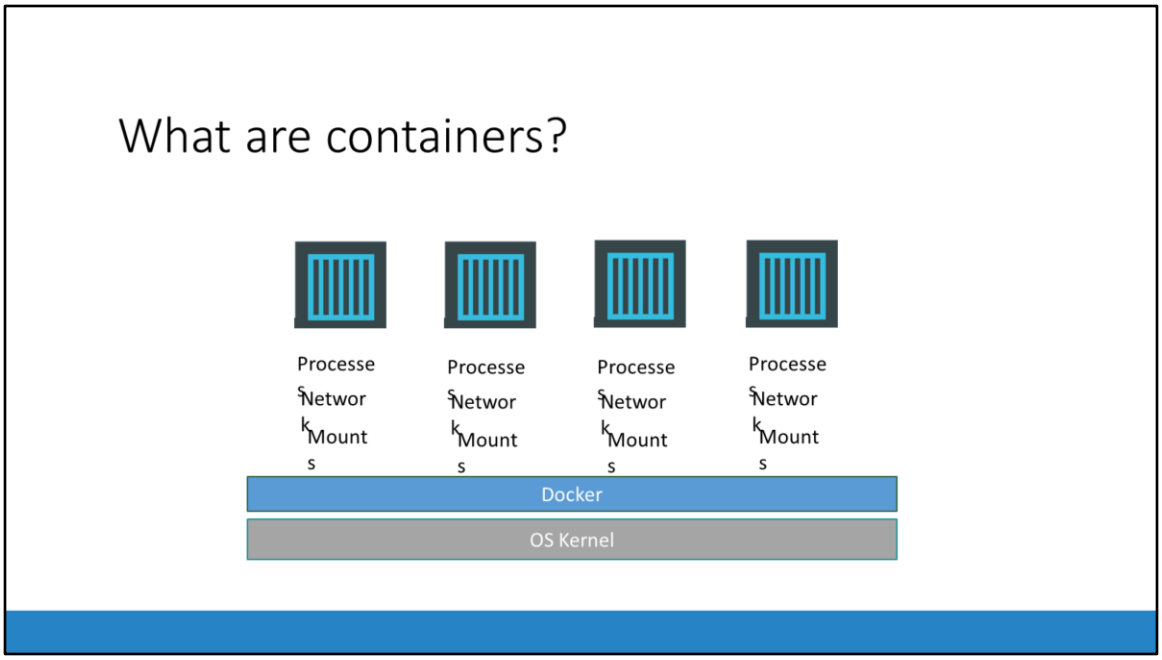
compatibility matrix issue is usually referred to as the matrix from hell.

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.

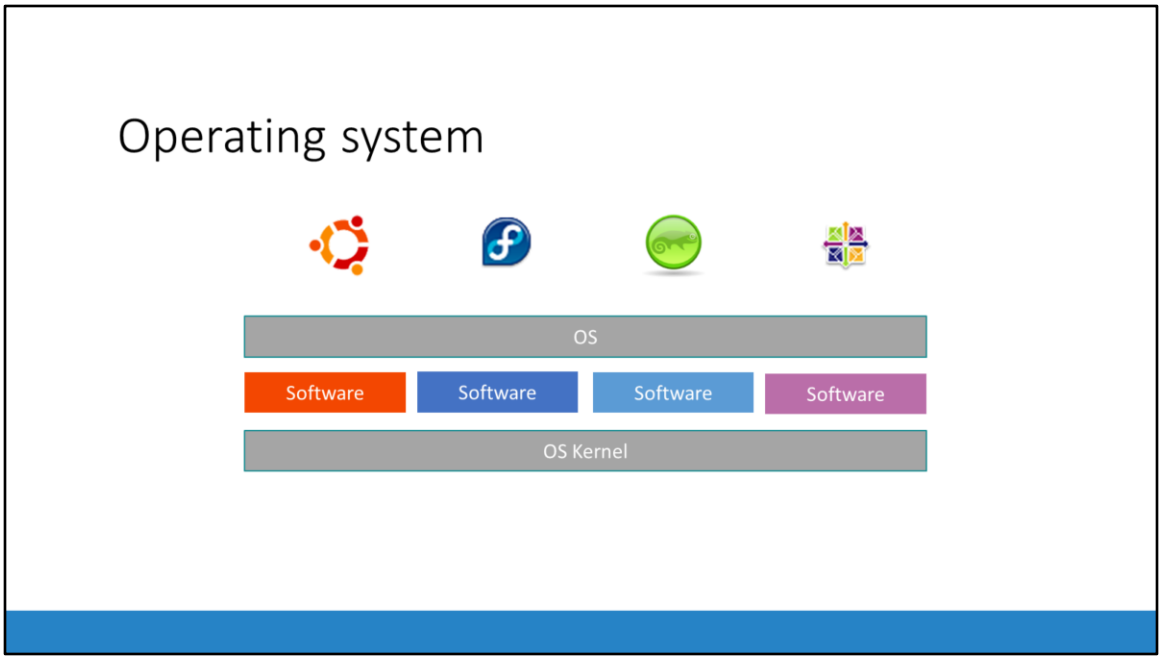
We also had different development test and production environments. One developer may be comfortable using one OS, and the others may be using another one and so we couldn't gurantee the application that we were building would run the same way in different environments. And So all of this made our life in developing, building and shipping the application really difficult.



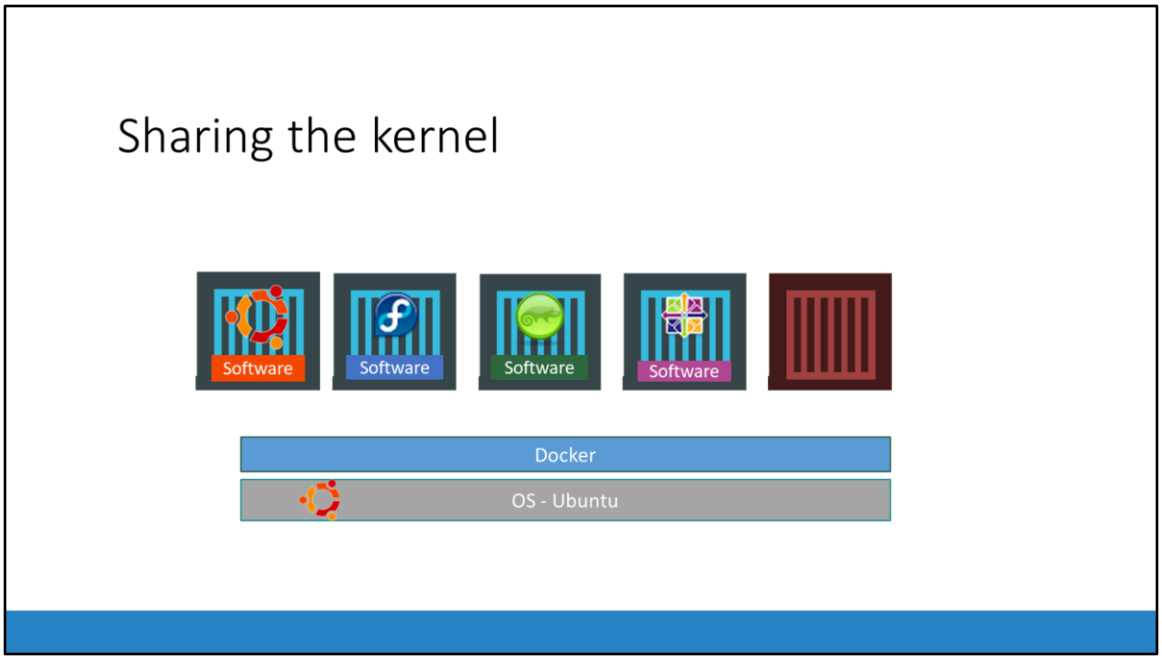
So I needed something that could help us with the compatibility issue. And something that will allow us to modify or change these components without affecting the other components and even modify the underlying operating systems as required. And that search landed me on Docker. <click> With Docker I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



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.



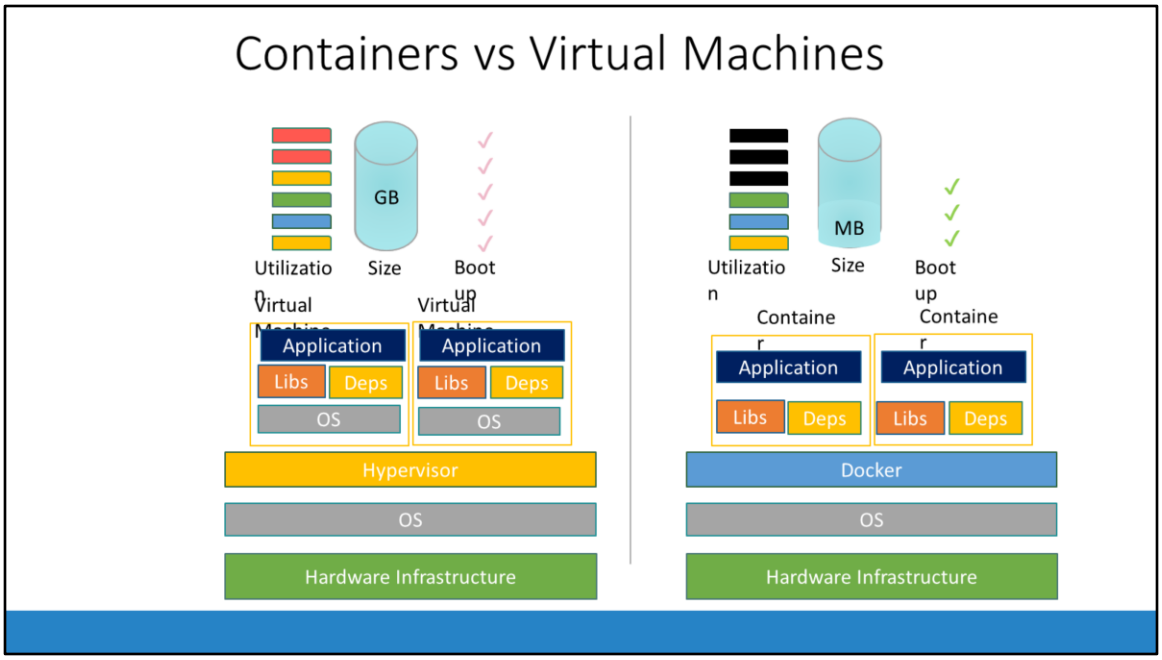
To understand how Docker works let us revisit some basics concepts of Operating Systems first. If you look at operating systems like Ubuntu, Fedora, Suse or Centos – they all consist of two things. An OS Kernel and a set of software. The OS Kernel is responsible for interacting with the underlying hardware. 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.



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. <click> 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.

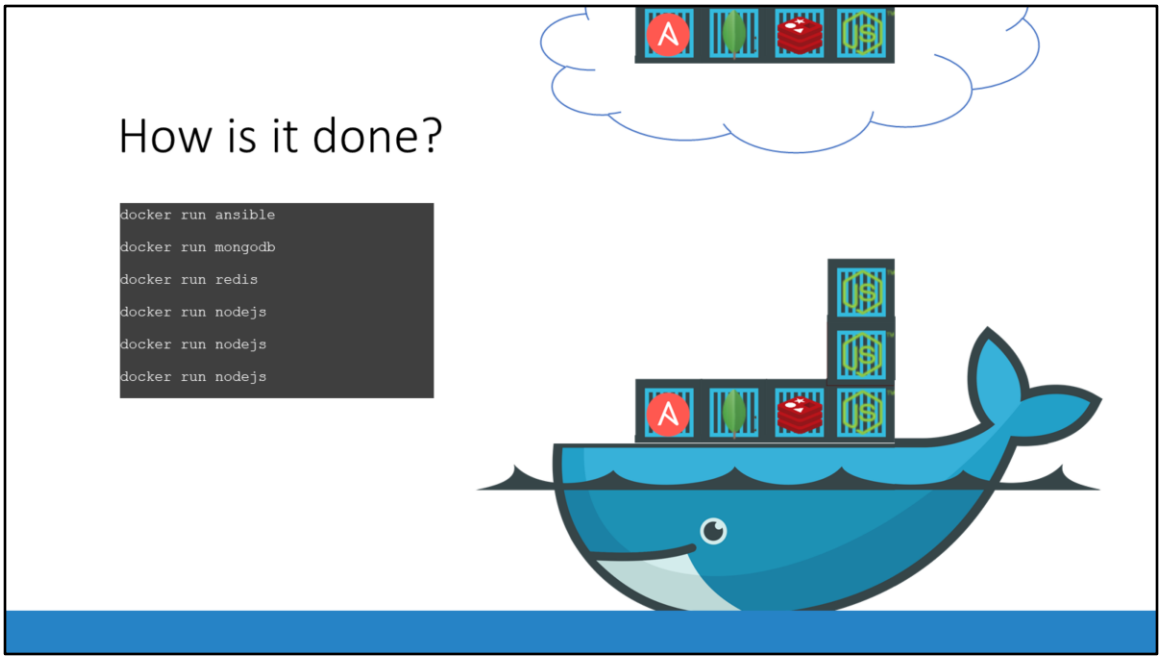
As you can see on the right, in case of Docker, we have the underlying hardware infrastructure, then the OS, and Docker installed on the OS. Docker then manages the containers that run with libraries and dependencies alone. In case of a Virtual Machine, we have the OS on the underlying hardware, then 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.

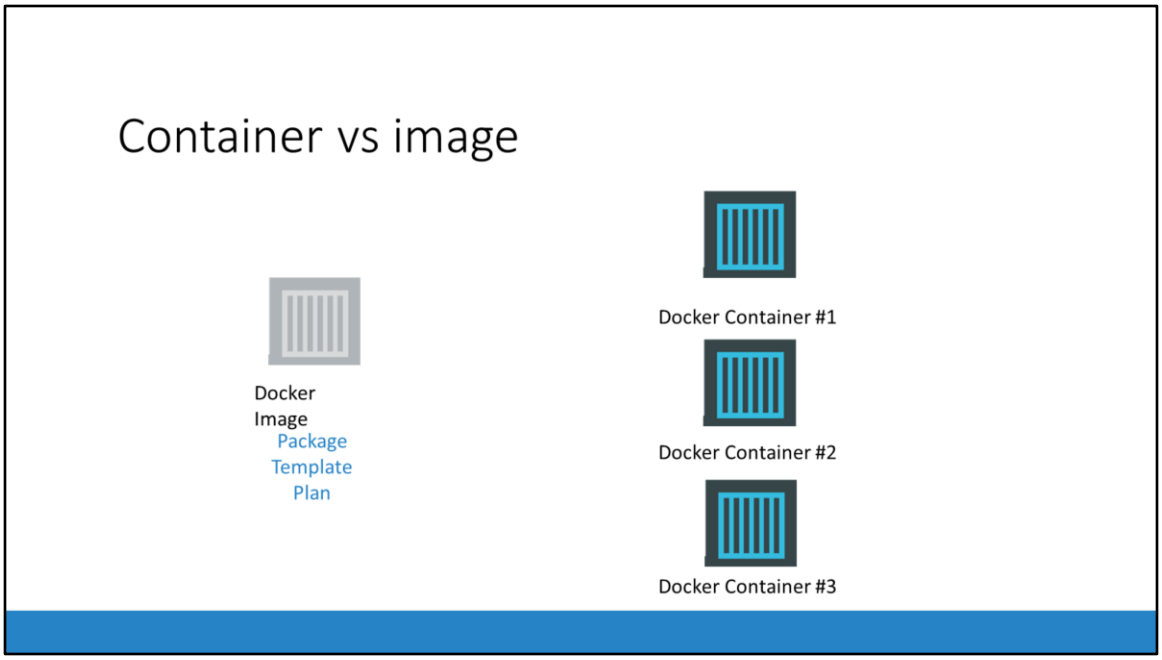
It is also important to note that, Docker has less isolation as more resources are shared between containers like the kernel etc. Whereas VMs have complete isolation from each other. Since VMs don't rely on the underlying OS or kernel, you can run different types of OS such as linux based or windows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

So these are some differences between the two.



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. <show dockerhub>. 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..

<click> 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. <click> 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.

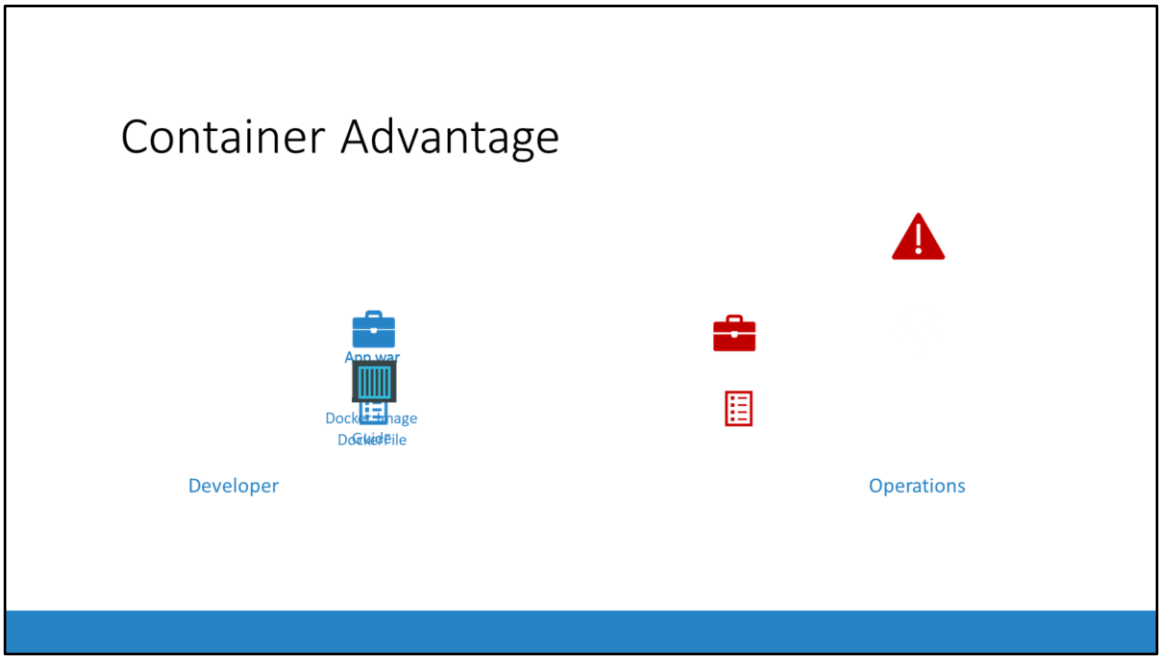


We have been talking about images and containers. Let's understand the difference between the two.

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

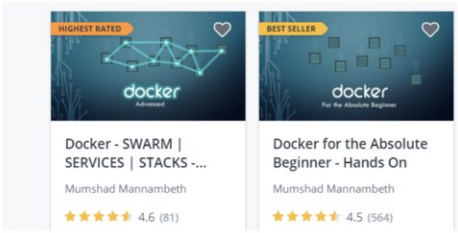
<show dockerhub> As we have seen before a lot of products have been dockerized already. In case you cannot find what you are looking for you could create an image yourself and push it to the Docker hub repository making it available for public.



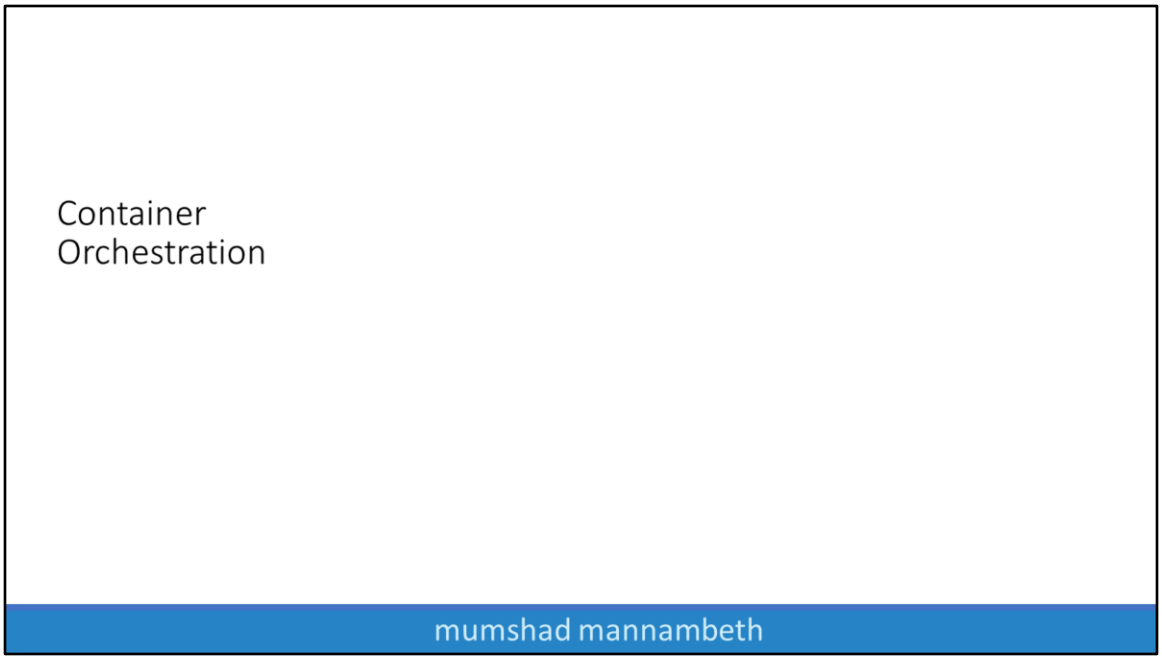
If you look at it, <click> traditionally developers developed applications. <click> Then they hand it over to Ops team to deploy and manage it in production environments. They do that by providing a set of instructions such as information about how the hosts must be setup, what pre-requisites are to 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. <click> When they hit an issue, they work with the developers to resolve it.

<click> 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. <click> The guide that the developers built previously to setup the infrastructure can now easily put together into a Dockerfile to <click> 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

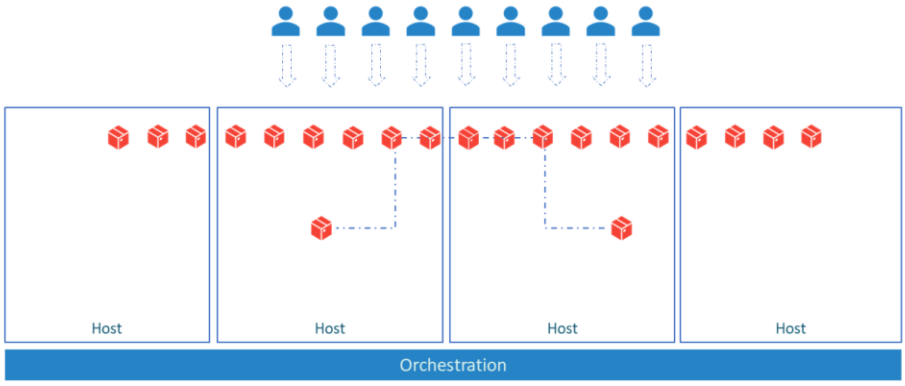


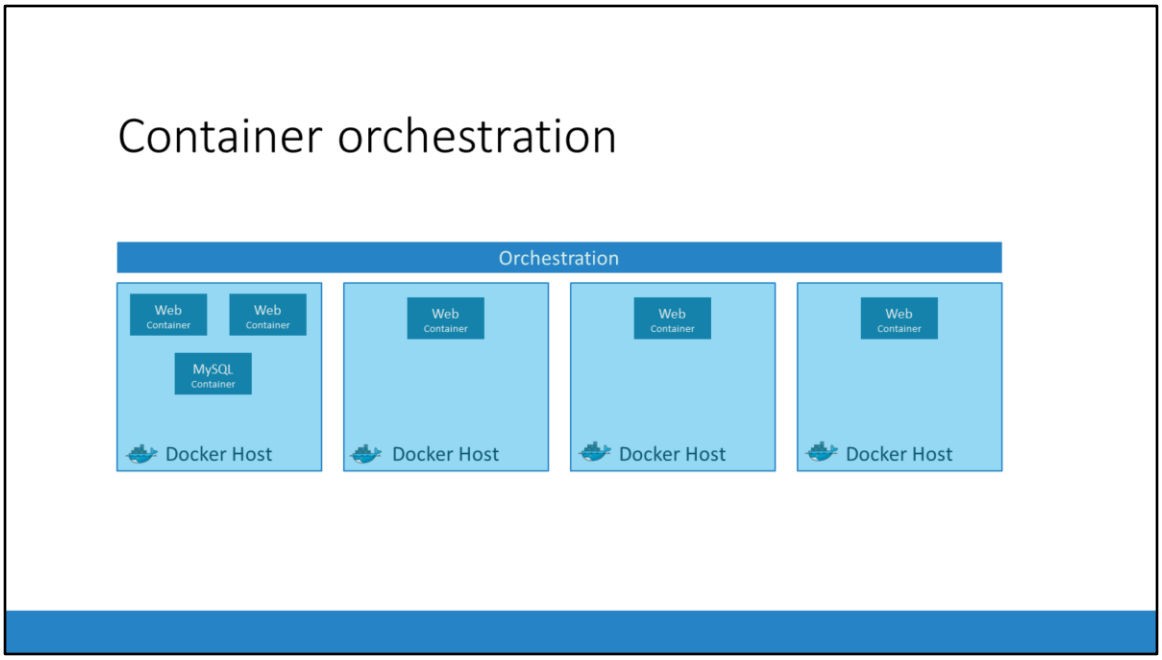
To learn more about containers, checkout my other courses - Docker for the Absolute Beginners and Docker Swarm were you can learn and practice docker commands and create docker files. That's the end of this lecture on Containers and Docker. See you in the next lecture.



In this lecture we will talk about Container Orchestration.

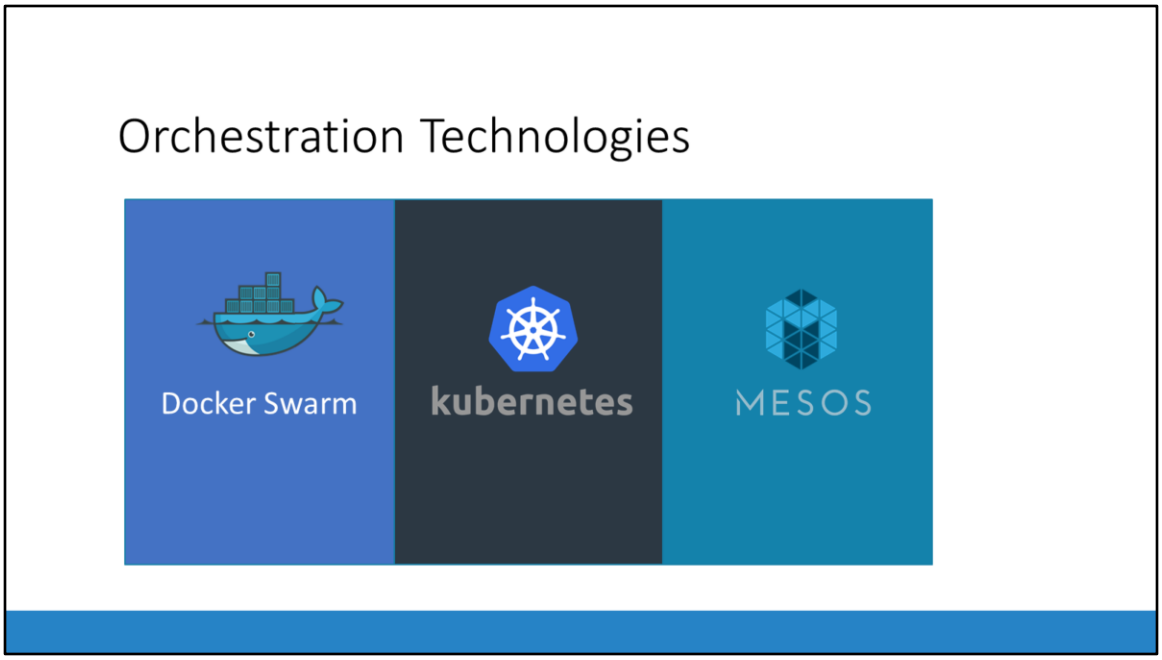
Container Orchestration



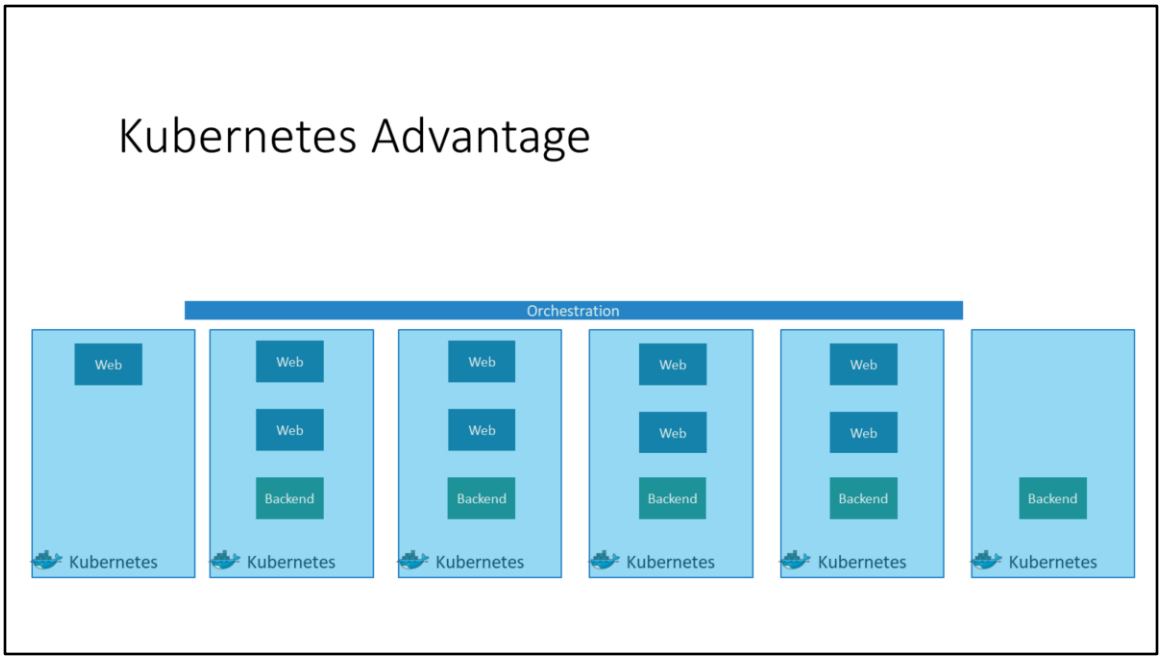


So we learned about containers and we now have our application packaged into a docker container. But what's next? How do you run it in production? What if your application relies on other containers such as databases or messaging services or other backend services? What if the number of users increase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

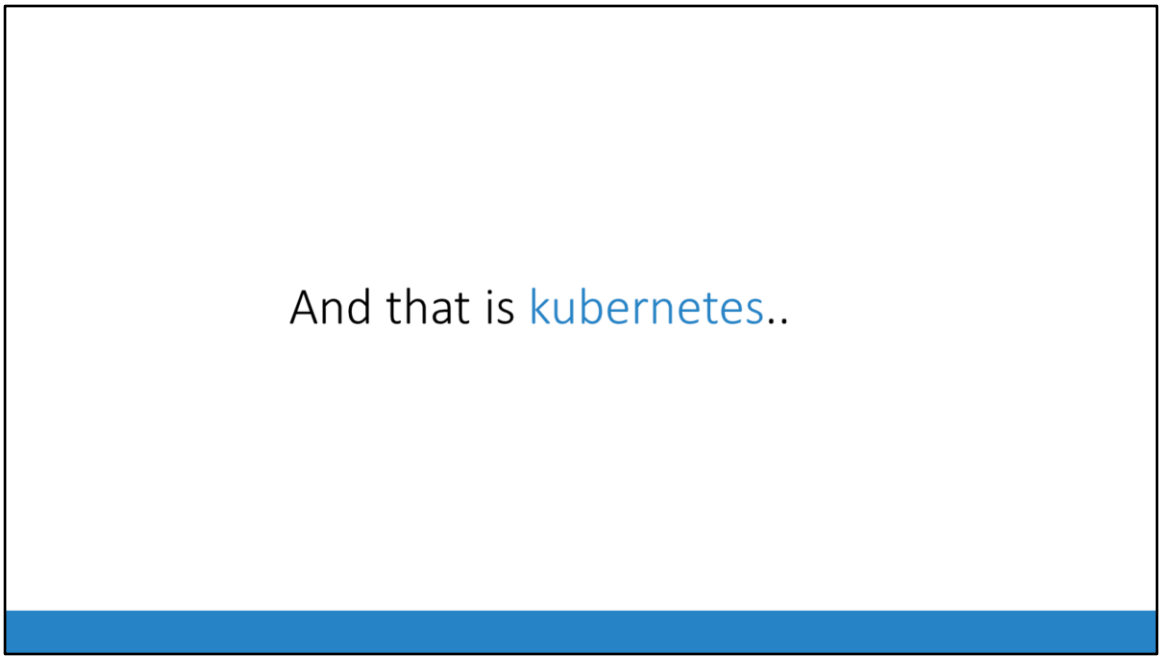
<click> To enable these functionalities you need an underlying platform with a set of resources and capabilities. The platform needs to orchestrate the connectivity between the containers and automatically scale up or down based on the load. This whole process of automatically deploying and managing containers is known as Container Orchestration.



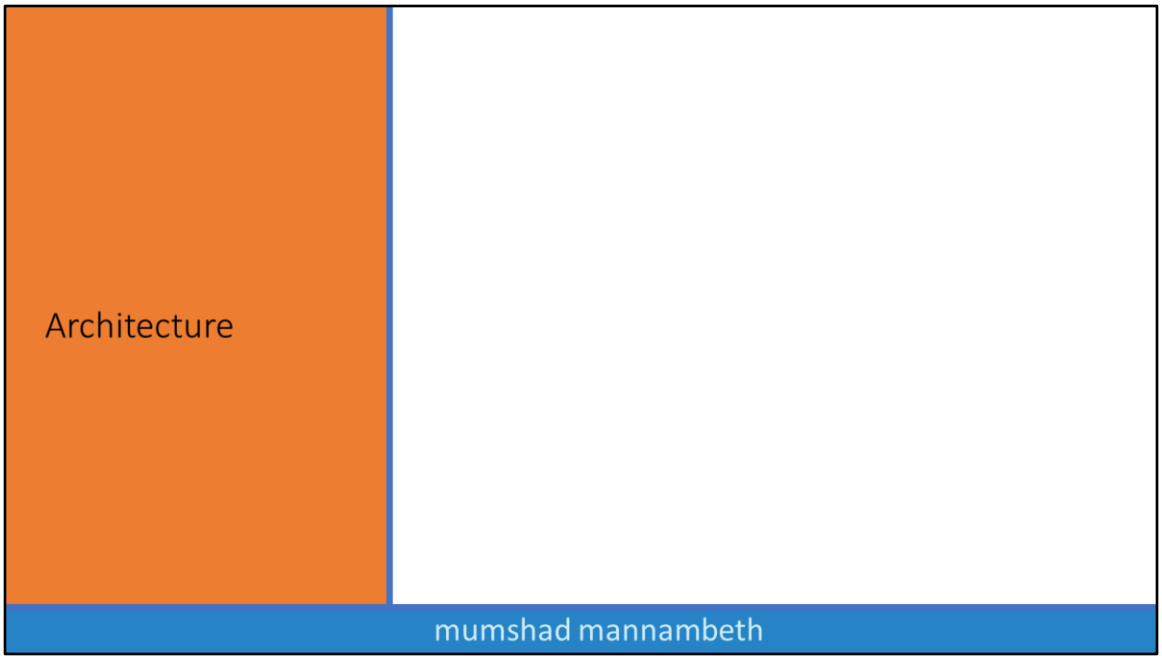
Kubernetes is thus a container orchestration technology. There are multiple such technologies available today – Docker has its own tool called Docker Swarm. Kubernetes from Google and Mesos from Apache. While Docker Swarm is really easy to setup and get started, it lacks some of the advanced autoscaling 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.



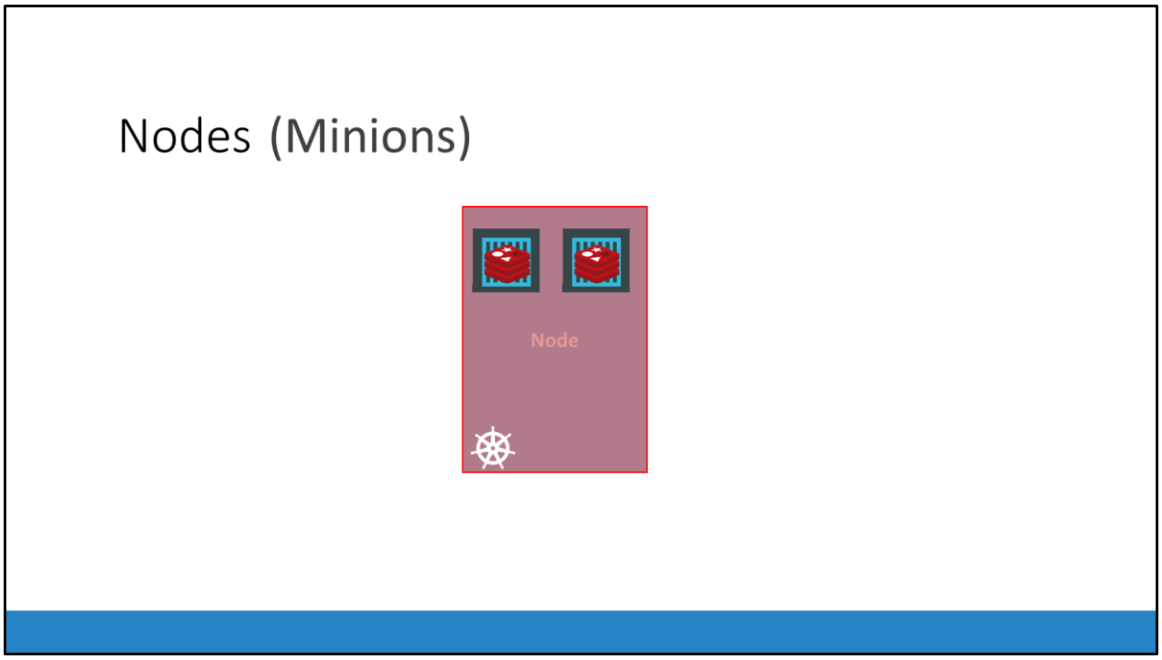
There are various advantages of container orchestration. Your application is now highly available as hardware failures do not bring your application down because you have multiple instances of your application running on different nodes. <click> The user traffic is load balanced across the various containers. <click> 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. <click> 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.



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.



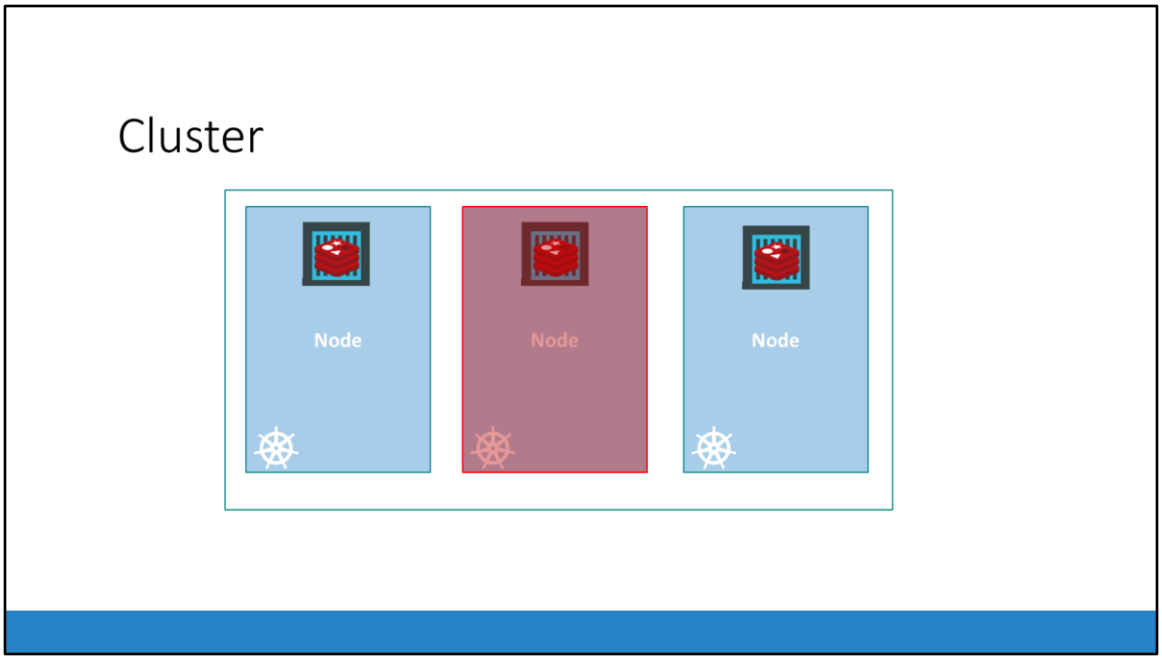
Before we head into setting up a kubernetes cluster, it is important to understand some of the basic concepts. This is to make sense of the terms that we will come across while setting up a kubernetes cluster.



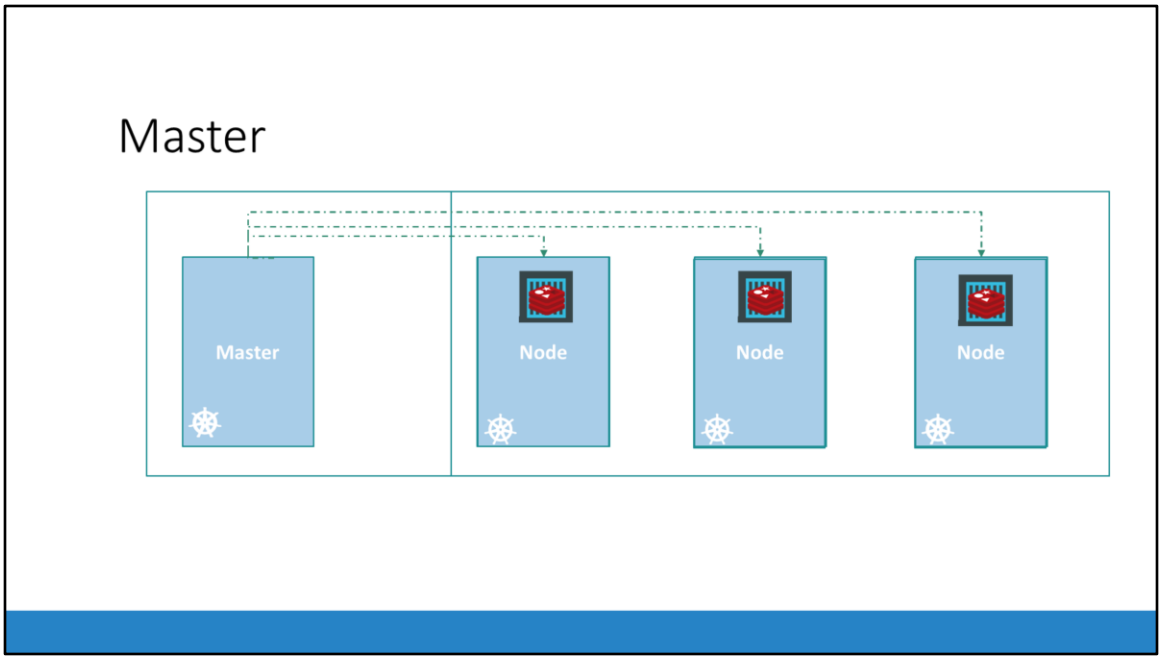
Let us start with Nodes. A node is a machine – physical or virtual – on which kubernetes is installed. A node is a worker machine and this is were containers will be launched by kubernetes.

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

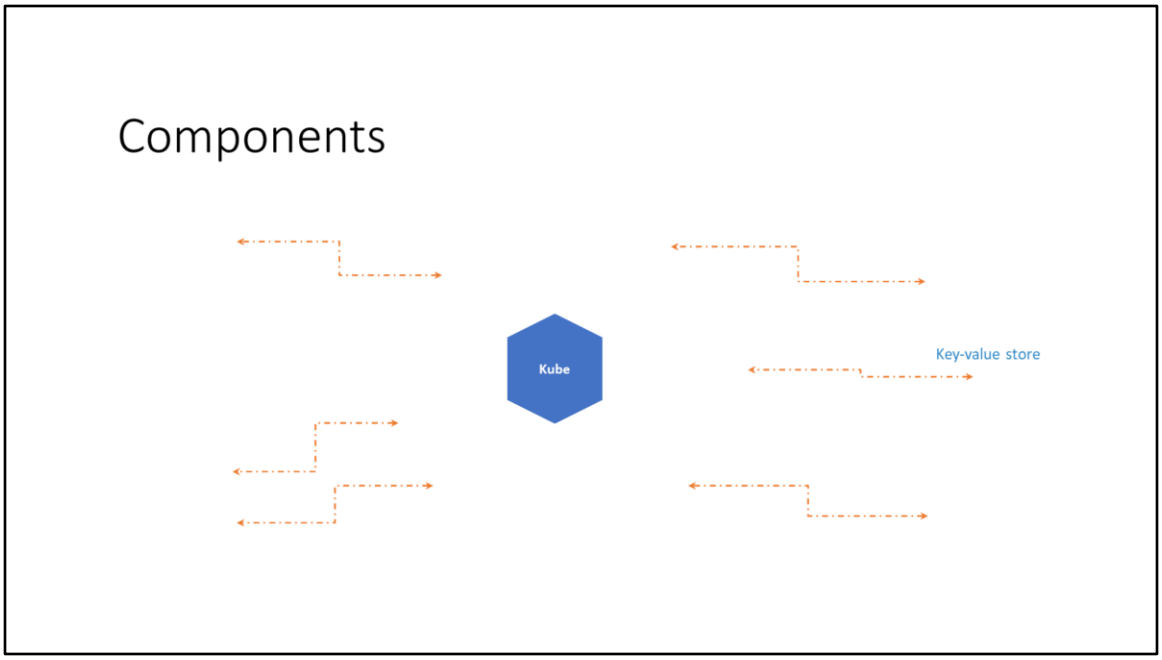
<click>
But what if the node on which our application is running fails? Well, obviously our application goes down. So you need to have more than one nodes. <click>



<click>
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.



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.



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.

<click> 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.

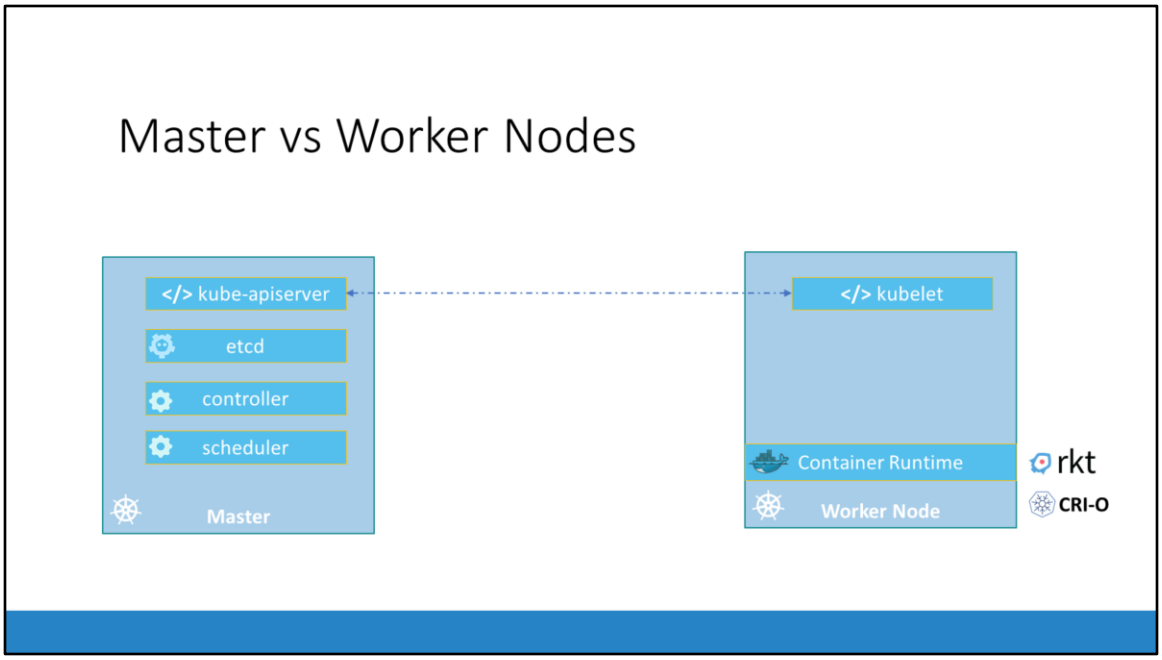
<click>
Next is the ETCD key store. ETCD is a distributed reliable key-value store used by kubernetes to store all data used to manage the cluster. Think of it this way, when you have multiple nodes and multiple masters in your cluster, etcd stores all that information on all the nodes in the cluster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

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

<click><click>
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.

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

<click><click>
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.



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. <click> 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.

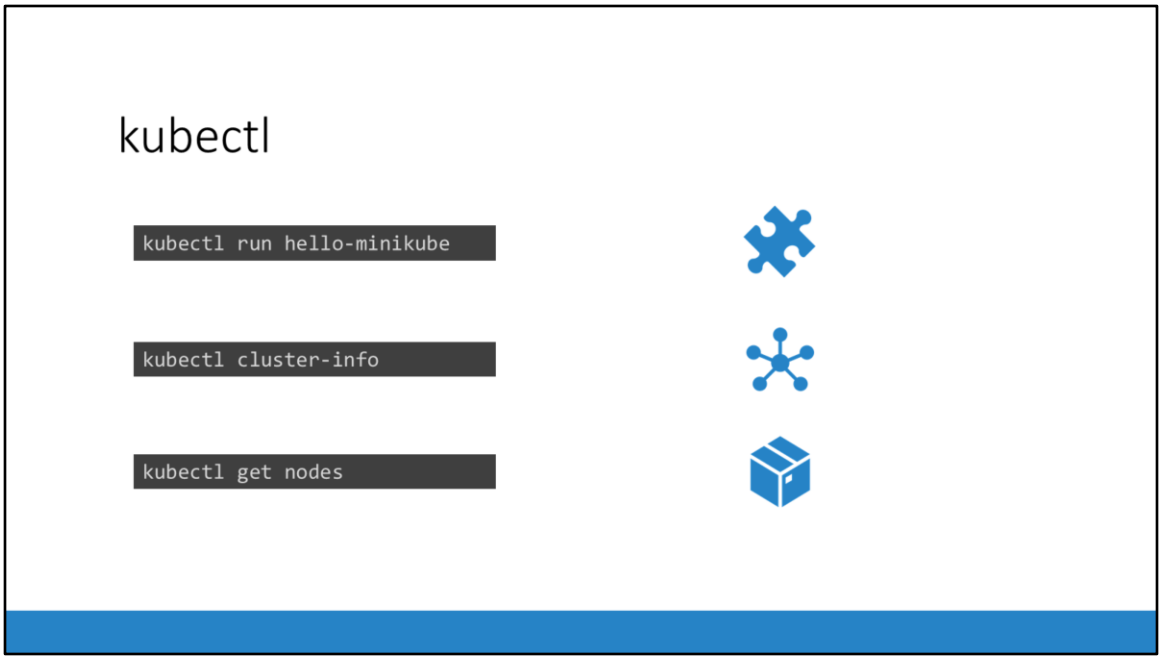
<click>
The master server has the kube-apiserver and that is what makes it a master.

<click>
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.

<click>
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.

<click>
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.



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.


KodeKloud