Contents

[KUBERNETES 2](#_Toc118731339)

[CONTAINERS 2](#_Toc118731340)

[WHY WE NEED CONTAINERS 2](#_Toc118731341)

[SOLUTION 3](#_Toc118731342)

[WHAT IS CONTAINER? 3](#_Toc118731343)

[HOW DOCKER WORKS? 3](#_Toc118731344)

[CONTAINER VERSUS VM 4](#_Toc118731345)

[CONTAINER VERSUS IMAGE 4](#_Toc118731346)

[CONTAINER ADVANTAGE 4](#_Toc118731347)

[CONTAINER ORCHESTRATION 5](#_Toc118731348)

[ORCHESTRATION TECHNOLOGIES 5](#_Toc118731349)

[ADVANTAGE OF KUBERNETES 5](#_Toc118731350)

[KUBERNETES ARCHITECTURE 6](#_Toc118731351)

[NODES 6](#_Toc118731352)

[CLUSTER 6](#_Toc118731353)

[MASTER 6](#_Toc118731354)

[COMPONENTS OF KUBERNETES 7](#_Toc118731355)

[MASTER VERSUS WORKER NODES 7](#_Toc118731356)

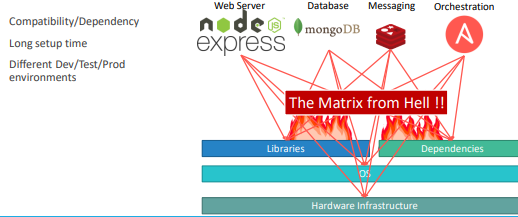
[KUBERNETES PODS 8](#_Toc118731357)

# KUBERNETES

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

## CONTAINERS

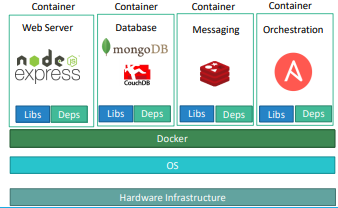
### WHY WE NEED CONTAINERS



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

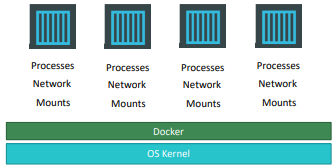
* Let say we have a requirement to setup an end-to-end stack including various technologies like a Web Server using NodeJS and a database such as MongoDB/CouchDB, messaging system like Redis and an orchestration tool like Ansible.
* We will have lot of issues developing this application with all these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

### SOLUTION



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

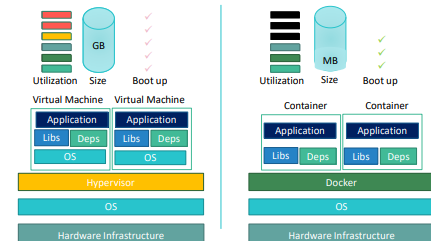
### WHAT IS CONTAINER?



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

### HOW DOCKER WORKS?

### CONTAINER VERSUS VM



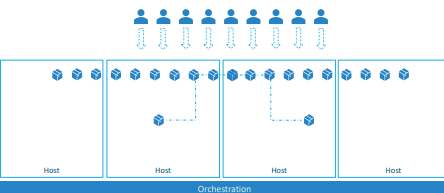
### CONTAINER VERSUS IMAGE

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

### CONTAINER ADVANTAGE

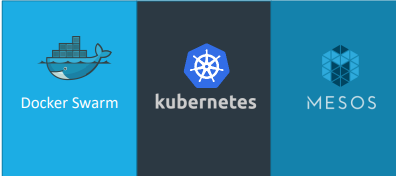
* For a traditionally developed application, developer hand it over to Ops team to deploy and manage it in production environments. They do that by providing a set of instructions such as information about how the hosts must be setup, what pre-requisites are to be installed on the host and how the dependencies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

## CONTAINER ORCHESTRATION



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

### ORCHESTRATION TECHNOLOGIES



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

1. DOCKER SWARM.

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

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

## ADVANTAGE OF KUBERNETES

There are various advantages of container orchestration.

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

## KUBERNETES ARCHITECTURE

### NODES

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

### CLUSTER

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

### MASTER



Master node is Kubernetes are

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

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

### COMPONENTS OF KUBERNETES



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

1. AN API SERVER.

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

1. AN ETCD SERVICE.

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

1. A KUBELET SERVICE.

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

1. A CONTAINER RUNTIME

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

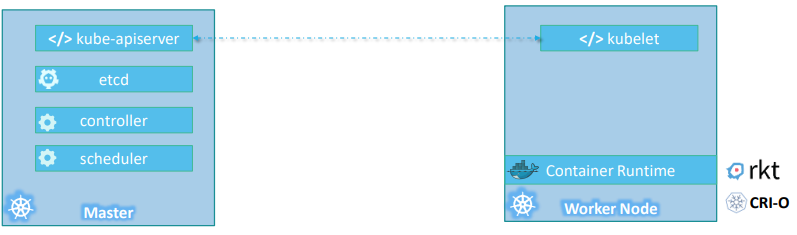
1. CONTROLLERS

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

1. SCHEDULERS.

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

### MASTER VERSUS WORKER NODES



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

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

**WORKER**

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

**MASTE**

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

## KUBERNETES PODS