**Virtual machines (VM)**

Vms are basically machines inside real machines. We realize it with the hypervisor. Hypervisor is just an application.

Virtualization is the process of creating a software-based, or "virtual" version of a computer, with dedicated amounts of CPU, memory, and storage that are "borrowed" from a physical host computer—such as your personal computer— and/or a remote server—such as a server in a cloud provider's datacenter. A virtual machine is a computer file, typically called an image, that behaves like an actual computer. It can run in a window as a separate computing environment, often to run a different operating system—or even to function as the user's entire computer experience—as is common on many people's work computers. The virtual machine is partitioned from the rest of the system, meaning that the software inside a VM can't interfere with the host computer's primary operating system.

Hypervisor virtualizes hardware components (cpu,ram,storage…) from the host machine.

A hypervisor, also known as a virtual machine monitor or VMM, is software that creates and runs virtual machines (VMs). A hypervisor allows one host computer to support multiple guest VMs by virtually sharing its resources, such as memory and processing.

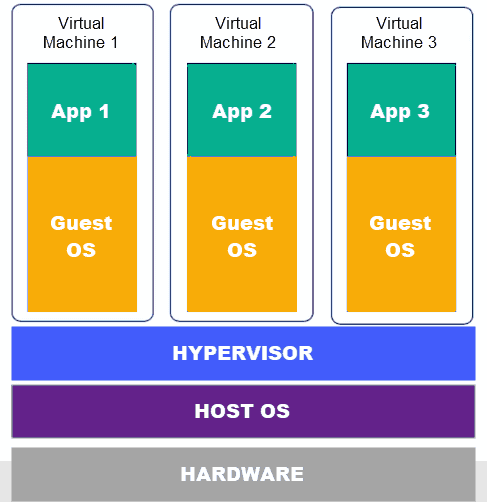
This type of hypervisor is called the type 2 hypervisor. Host os allows guest os’s to use his resources.

So the type 2 hypervisor asks the host’s os for the resources whereas the type 1 hypervisor has direct full control to the host’s hardware.

In type 2 hypervisor the os of the host shares its resources for vms meaning that if there are many vms it has to share its resources for all of them.

We install ISO files of the os that we want. An ISO file (often called an ISO image), is an archive file that contains an identical copy (or image) of data found on an optical disc, like a CD or DVD (so when we install an os we use cds here it is an image of the disk). They are often used for backing up optical discs, or for distributing large file sets that are intended to burned to an optical disc.

ISO image as a complete copy of everything stored on a physical optical disc like CD, DVD, or Blu-ray disc—including the file system itself. So we install iso images to have vms. Kali linux iso and etc.



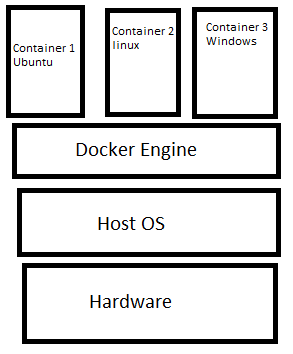
**Docker**

Docker replaces vms. Vms virtualize hardware whereas docker virtualizes os. In docker, you have a single OS (or kernel to be precise), and the resources are shared between the containers. Hence it is lightweight and boots in seconds.

In docker, all of the containers share the same os but it is appearing for each container as if they have their own os.

A container is a way to package application with all the necessary dependencies and configuration and this package is portable, meaning that it can run on any platform. Containers share the same kernel.

Containers live in a special repository called container repository.



Containers are made up of layers. We have linux image layer, application image layer and so on. Images are just binary numbers representing the exact copy of the software.

**Container vs Image**

Container is a running environment for an image. An image is a template for creating an environment of our choice. This could be a database, a web app and etc. An image is a snapshot. You can create multiple snapshots in versions then you can point to a version that you want at a particular time.

So we have an image and from this image we run a container. A container only lives as long as there is a process running inside.

**Pulling an image and running container**

Let’s pull an image from the docker hub and create a container from this image.

After downloading an image, we run a container from the image we pulled. 🡺 Docker run nginx:latest

Latest is the name of the tag. To look at the running containers 🡺 docker container ls or docker ps

To run a container in a detached mode so that we don’t just hang in the command line🡺

Docker run –d nginx:latest

To stop a contaner 🡪 docker stop containerId or containerName

To remove a container 🡺 docker rm containerId or containerName

To remove all the container from a single command🡺

docker rm –f $(docker ps -aq) – here –f is for forcing removal so that running containers are also removed. –a is to display all containers (running and stopped containers) and –q is for only displaying ids.

**Exposing Port**

We wanna go from the host to the container (we wanna access the container from our browser). In this case we have to expose the port. We use –p 🡺 docker run –d –p 8080:80 nginx:latest

So whenever we type “localhost:8080” we want that to be mapped to port 80 on the container.

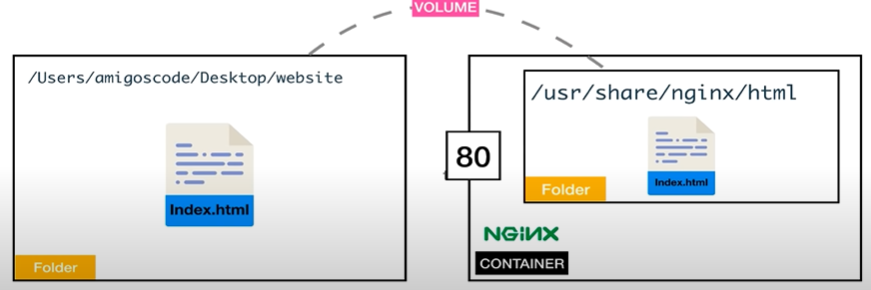
**Container names**

If we don’t specify a container name when we run it, a random name is given. If we want to have a custom container name 🡺

docker run –name containerName –d –p 8080:80 nginx:latest

**Docker Volumes**

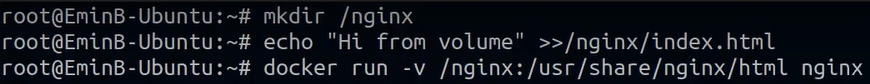
When we have data in a container, once we stop the container the data is gone. So in order to have some persistent data we can take it from the host file system. Volumes allow sharing of data between host and container and also between containers. Files & Folders. A folder in physical host file system is mounted into the virtual file system or docker.



The folder is in the host.



ro- read-only so that it can’t be changed.



We create a ngnix dir and we write “Hi from volume” string with echo and then we mount that file (index.html) into the container.

**Creating custom images**

To create a custom image, we create a Dockerfile. We never build an image from scratch. We always use an existing image as our base image.

From nginx:latest

RUN apt-get update

RUN apt-get install nginx -y

First our base image is nginx. Then update and then install nginx.

**Interactive mode (-it)**

We can get into a running container 🡺 Docker exec –it containerId bin/sh – docker execute in an interactive mode into this container that is running and when you get there execute /bin/sh. If it is alpine then bin/sh for the terminal.

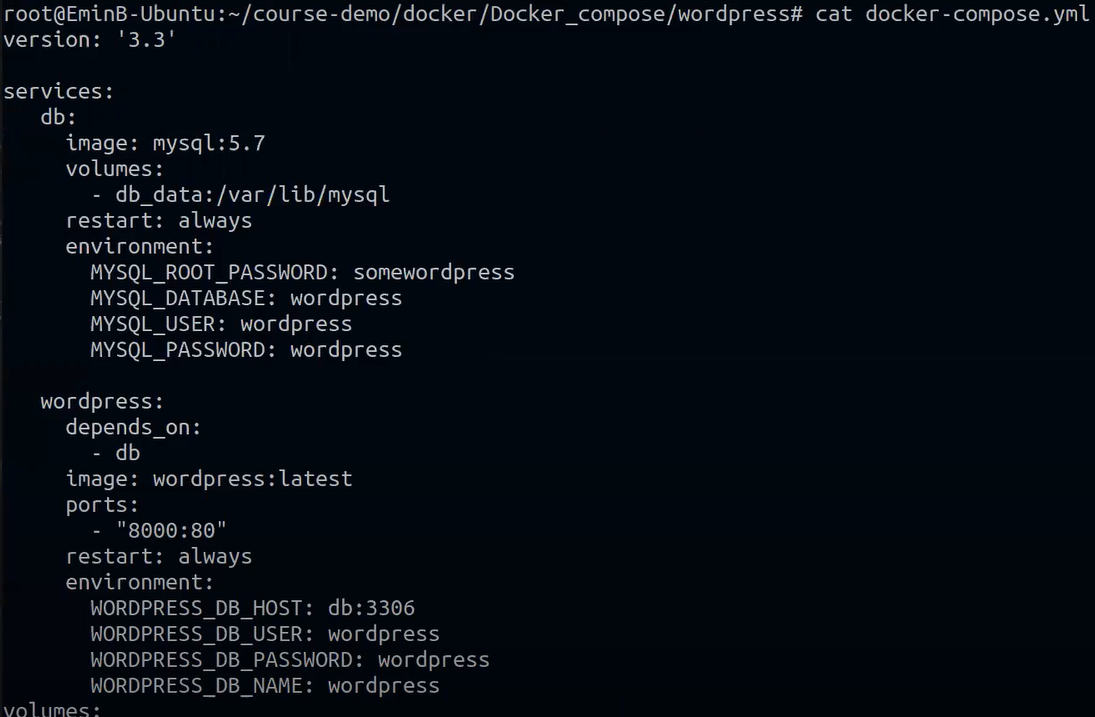
We use this so that we get inside a container and we can run commands inside this container.

**Docker start vs docker run**

Docker run creates a new container but “docker start” restarts already stopped container.

**Docker compose**

Docker compose is just a structured way to contain common docker commands. We just don’t want to have all the commands in one line. It makes editing easier, for example, if we want to change an enviromental variable or if one app requires multiple containers🡺



Services are basically our containers which have their own image.

**Kubernetes**

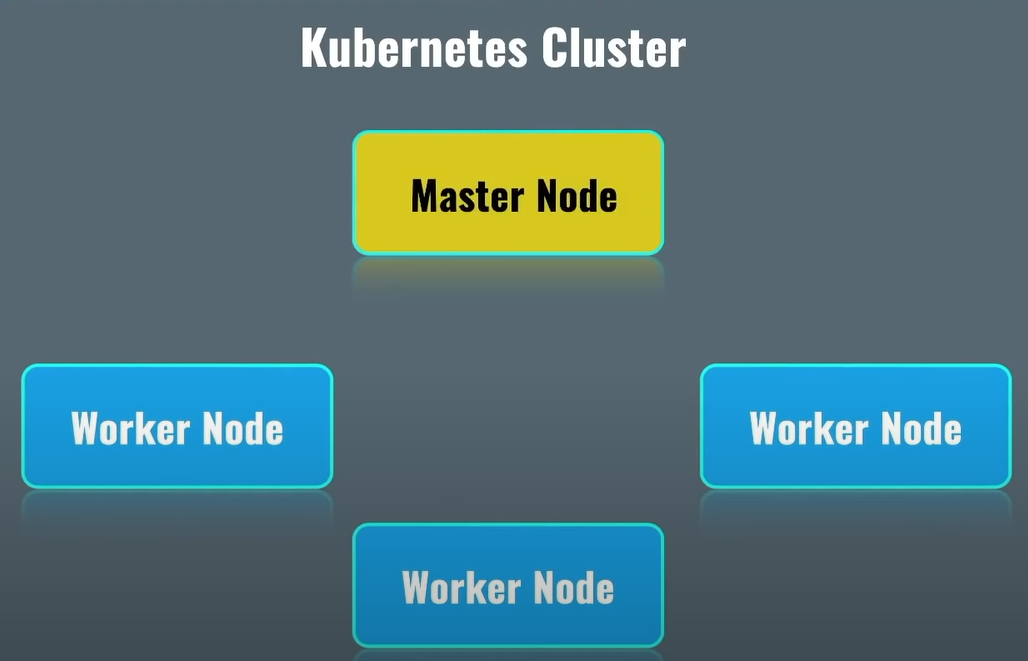
Kubernetes, also known as K8s, is an open source system for managing containerized applications across multiple hosts. It provides basic mechanisms for deployment, maintenance, and scaling of applications. Your app is never down.

The smallest unit of K8 is the pod. It is the abstraction over container. It creates a running environment for containers. We can run multiple apps inside one pod.

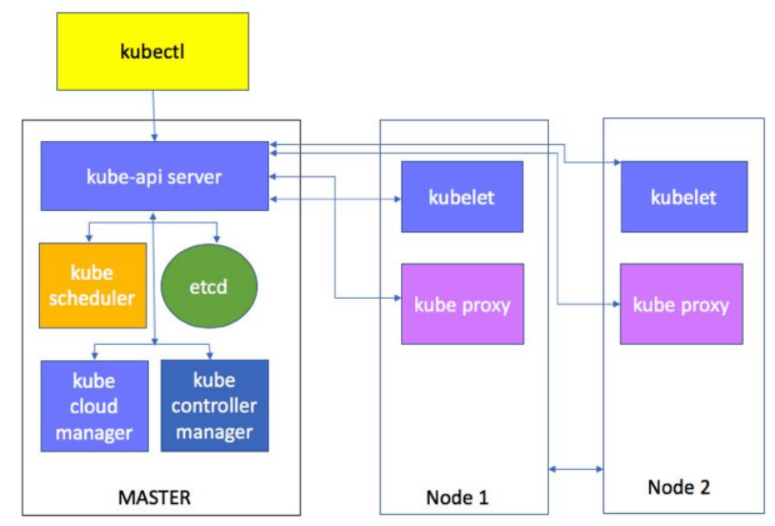
K8 offers a virtual network so each pod has its own IP address. So each pod can communicate with each other using their ips.

If a container inside a pod dies, a new one will be created and new ip address will be recreated for the pod which is inconvenient. Because of regular change of an ip address of the pod it is difficult to communicate with pods using their ips since they change.

Because of this another componenet is introduced, namely Service. Service is basically a permanent ip address that can be attached to each pod. The lifecycle of service and pod is not connected! So even if a pod dies the service and its ip address stay.



Master node or control plane manages worker nodes.



Kubectl (aka kube control) is a command line tool which allows us to connect to a specific kubernetes cluster and manage it remotely.

Every request is handled by Api server. So, for instance, if you wanted to get info from etcd key value base, then you would have to first send request to the api then, it would go to the etcd.

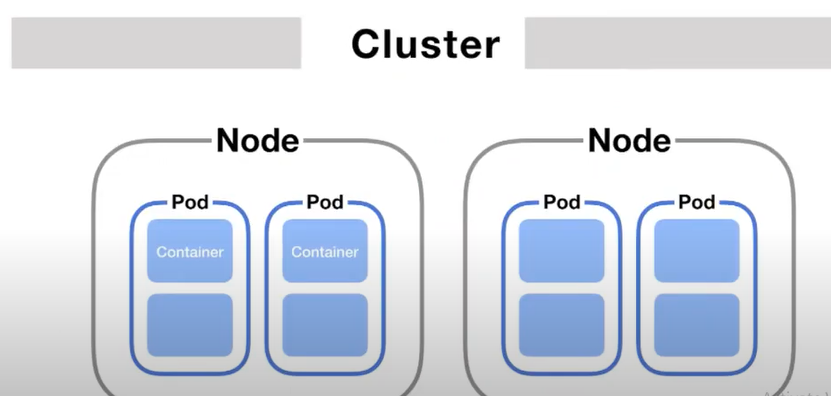
**minikube** is a tool that lets you run Kubernetes locally. minikube runs a single-node Kubernetes cluster on your personal computer (including Windows, macOS and Linux PCs) so that you can try out Kubernetes, or for daily development work.

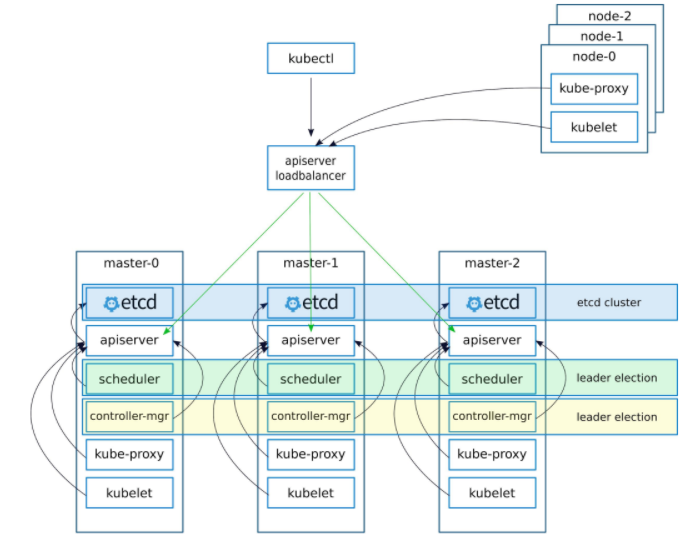
Kube-scheduler decides where a container has to rise. So it doesn’t raise anything but just decides. It decides where to run a container by analyzing nods’ resources (cpu and ram), if they have the most space then the container is decided to run in that node.

Kubelet sends information about node and containers, that are run under that node, to the etcd via kube-api-server. And when a container is decided to run on a node, that nodes’ kublet received command and sends this command to docker so that a container is realized.

Kube-proxy sends the received traffic to a particular container.

One master can die so that’s why it is better to have 3 master nodes. When there are more master nodes, one controller manager leader and one scheduler leader are selected.





**Common commands**

kubectl run podName --image imageName

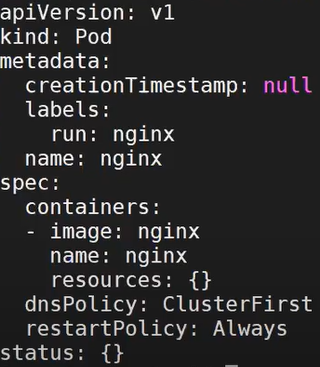
kubectl run nginx --image=nginx

**Applying common commands with one command**

To create a file that has all the specs to make our lives easier so that we don’t type commands, we use --dry-run=client –o yaml command to print the object that would be sent, without sending it🡺



If the command needs to run so that yaml file can be displayed, we use --dry-run option to imitate run command.



It is the manifest file of the pod. Then to apply this yaml file we use the following command 🡺

kubectl apply –f pod.yaml 🡪 -f is to specify filename

--dyr-run=client just runs the command without actually running it for real and when used together with “-o yaml”, then the all the specs to run the command are put in a yaml format and printed to the console.

-o format- output it in a specific format (json,yaml) 🡺 -o json

Dry run mode gives you the possibility of issuing a command without side effects for testing an actual command that you intend to run

-f option🡪 --filename=[]: that contains the configuration to apply,

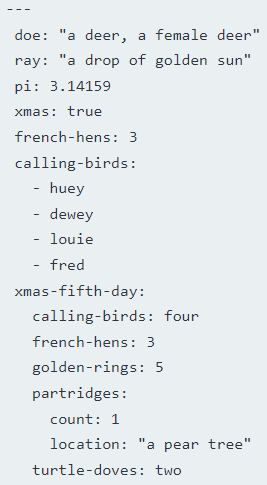
--force is true (force deletion).

**YAML FILE**

YAML is a data serialization language that is often used for writing configuration files. Depending on whom you ask, YAML stands for yet another markup language or YAML ain’t markup language (a recursive acronym), which emphasizes that YAML is for data, not documents.

It's often used as a format for configuration files, but its object serialization abilities make it a viable replacement for languages like JSON.

YAML uses Python-style indentation to indicate nesting. Tab characters are not allowed, so whitespaces are used instead. There are no usual format symbols, such as braces, square brackets, closing tags, or quotation marks. YAML files use a .yml or .yaml extension.



The file starts with three dashes. These dashes indicate the start of a new YAML document. Next, we see the construct that makes up most of a typical YAML document: a key-value pair.

**Kubectl logs**

kubec logs podName



This will print all the logs of the single container in that pod. If there are more containers in the pod, then the name of the container must be specified.

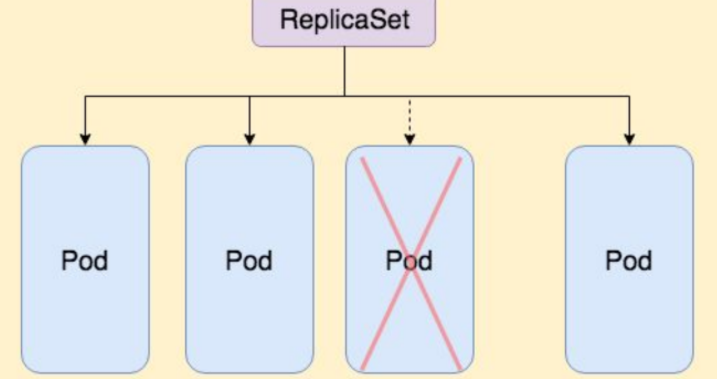
If we wanted to watch logs live then we use -f 🡺



**Raising deleted pods (Replicaset)**

If we delete a pod, pods are not raised again because the controller manager (main component) doesn’t manage any pods. Contoller manager manages replica sets. To actually automatically restart pods, we need replica sets.

Inside replica set we have pods. Pods in a replicaSet are all the same.



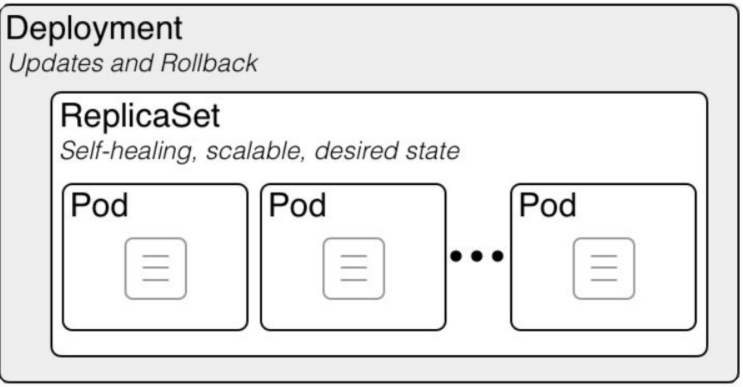
So we do like this 🡺 create a pod whose replicaSet is 3 and inside that pod raise this container/containers.

We can’t create replicaSet manually by simply using pod because they will not be able to be updated. We won’t be able to scale replicas, for instance (change the numbe of replicas). So we need something called deployment to be able to update.

**Deployment**

We create a deployment with a specific image and number of replicas for the replicaset.

If a pod dies inside the replicaSet then a new pod is started from the replicaSet.



So we basically create a deployment with a specific image and replicaSet number.



Now there will be 3 pods of the image nginx.

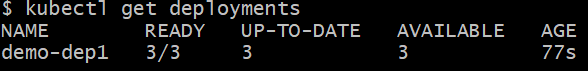
Now if we delete a pod, it will raise a new pod.

Basically, when we delete a pod, controller manager sees that the replica number is 3 but one pod died so now it is 2, so controller manager has to raise 1 more pod.

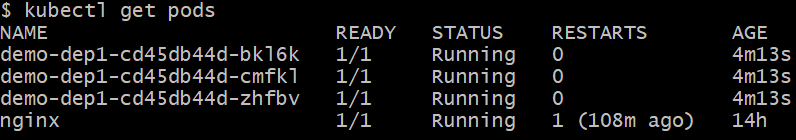
So controller manager always keeps however many pods we specified alive. So if we delete all 3 of the pods, it will just restart raise new 3 pods.

**Normally the maximum number of replicas is 3.**

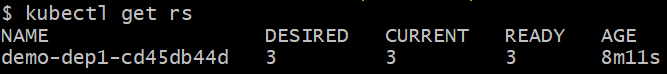
 or  to print all the deploys



Ready shows that there are 3 pods in this deployment so the number of replicas is 3.



The first part of the pod name is deployment name and the second part is the replicaSet 🡺



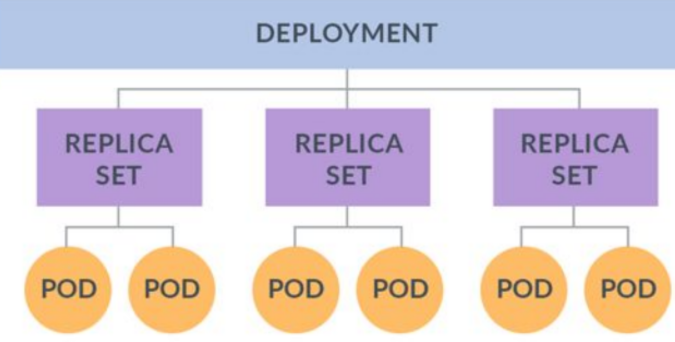
The third part is the unique hash to distinguish pods.

Now we can also scale replicaSet. 🡺



Now we increased the number of replicas.

This means in one deployment we can have multiple replicaSets.



Let’s create a manifest file of the deployment and use that file to create a specific deployment when needed.

 to get the manifest of the command in the yaml format. Then we can put everything from here to .yaml file 🡺



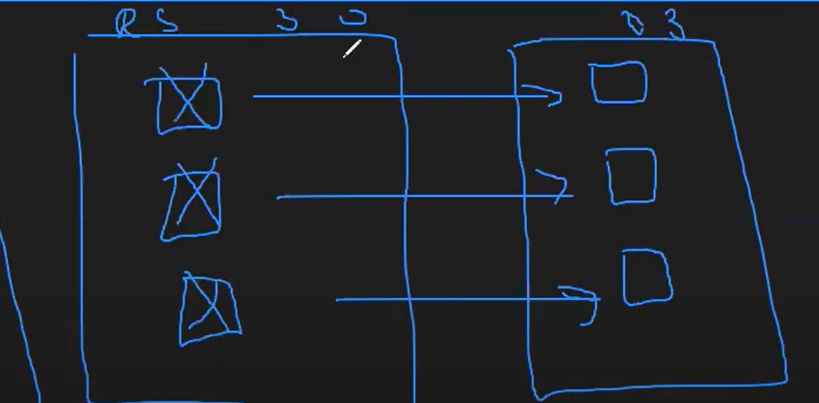


With deployment, we can set a new image for the pod meaning that we can update the deployment.



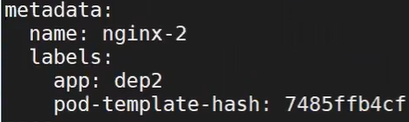
This will terminate the old rs and create a new replicaSet. The process goes like this 🡺

1. First a new replicaSet is created with no pods
2. Then a new pod is created in that replicaSet with the specified image.
3. When the pod is on ready status which means it is ready then one replica from the old replicaSet is terminated because controller manager tries to keep the number of replicas 3



The old replicaSet is not deleted for rollback purposes.

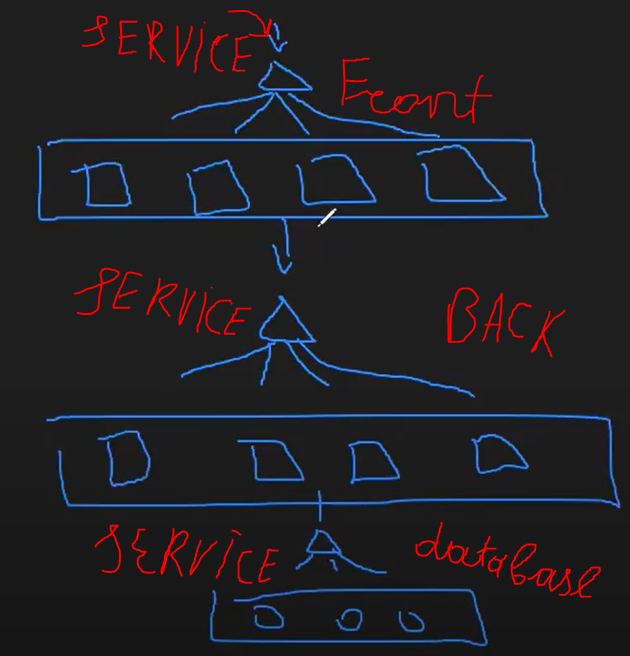
When a pod dies in a deployment, a new pod is raised for that deployment so that the number of pods is the same as the replicas number. We can have lots of deploys. So it figures out to which deploy it has to send this newly raised pod by the labels metadata of the pod 🡺

 it is the pods metadata when created. So it references dep2 with the specific hash so that it is not confused with another deployment. We are making sure that this newly raised pod is delivered where it needs to be delivered.

**Services**

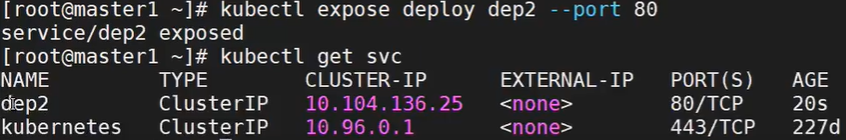
There are different types of services: ClusterIP, NodePort service

ClusterIp services work only inside kubernetes clusters.

As we know, pods ips are dynamic, meaning that the change. So we use services to communicate amon deployments🡺  


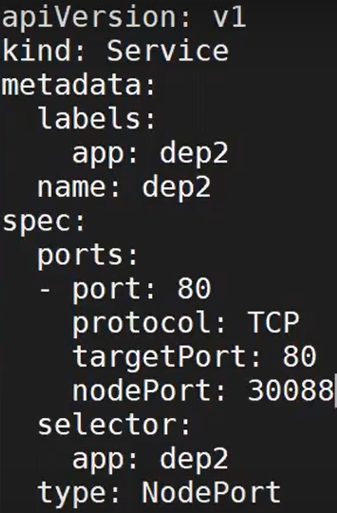
So here we have 3 deployments: front, back and some database. Front will send request to the database so it needs a service. BackEnd also needs another service to send a reqeust to the database.

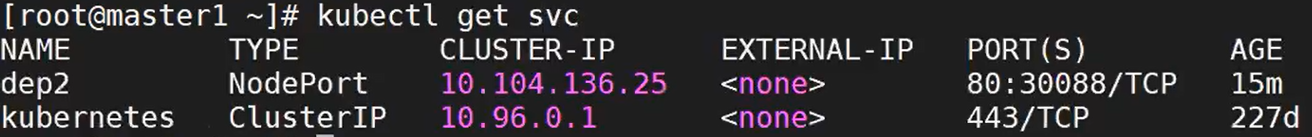
So we create a service by exposing something with a specific port🡺



kubernetes service is the api-server. These services type is ClusterIp.

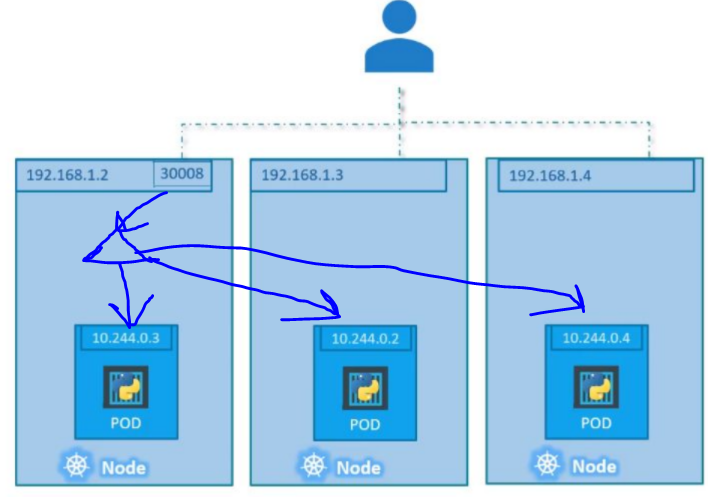
With **nodePort** service, we can send a request from a node. When we create a nodePort service, one port is specified for all the worker nodes to which if the request is sent, it forwards the request to the ClusterIP service which will then direct the request to the pod.





As we can see, if we send a request to the nodes’ 30088 port then it will be mapped to 80’s port of the ClusterIP service.

So nodeport inside itself has a ClusterIP svc. When we create a nodePort service, one port from 30\_000 range activated for all worker nodes and when it is sent to this port then it’ll forward the request to the ClusterIP service which will then direct the request to the pod.



That one svc inside the pod is the ClusterIP service and this entire thing is inside nodeport Service.

Now we send requests to worker nodes with the port 30008, for example, and after receiving this request from this port it will forward this request to the ClusterIP service which then forwards the request to the specific pod.