**Kubernetes**

Kubernetes main components :

- Pod: It adds a layer on top of the container layer so that we can interact with k8s and not docker for example. Each pod runs 1 container inside of it.

- Node: Physical server

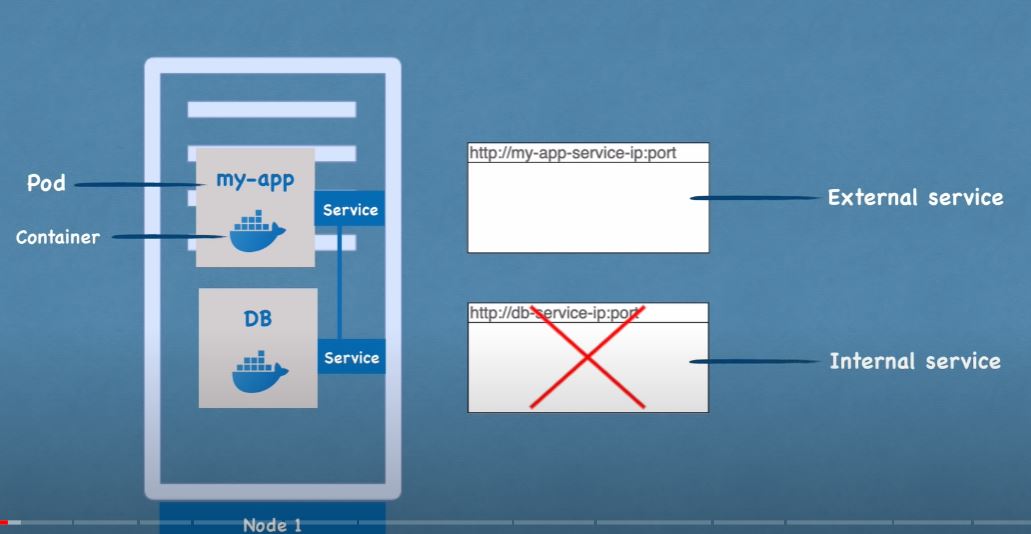
- Each pod has his own virtual network so each pod has his own IP address. If the pod has been re-created, a new IP address will be allocated for the pod which can sometimes be messy if the pod is a database container for example. To fix that problem, a component called service and ingress is used.

- Service and ingress: Permanent IP address that can be allocated to each pod. The good thing here is that the lifecycle of the pod and the service is not connected so if pod dies, the service and its IP address will stay.

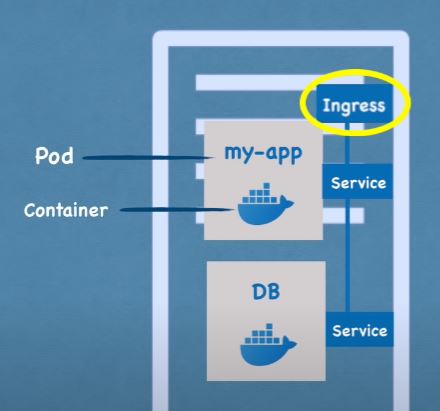
- Services:. To make our app inside of pods accessible to the outside, we must use external services. The service give us permanent IP address for our pod and act as a load balancer. We have 2 types of services :

External service: For our app container

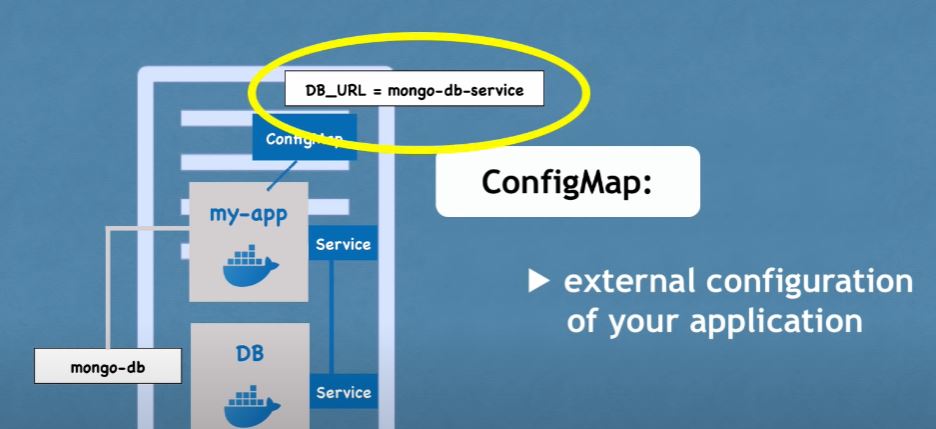
Internal service: For our database container



As we can see the external service is used for our app but if we look deeply into the URL, we are using the ip address of the node (not the service) but In real life we do not need it to look like that we need to use names so DNS. So to fix that issue, we will use something called ingress. So instead of service the request comes to ingress and then forward to service.



- Config map: It’s the external configuration of our application so it will configuration data like URL for the database… So, we will connect it to the pod. In this case, now if we change the name of the database URL for example, we do not need to rebuild the image… all we have to do is modify the config map component.



We can use data from config map inside of our pod app by using environment variables or as properties file.

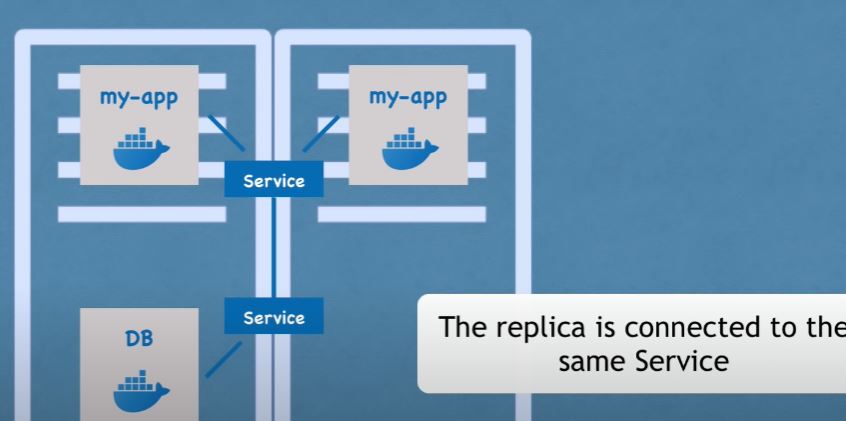
We can also use config map to change databases names or passwords… (But it is not good practice to put database passwords in the config map!) For this purpose, K8S has another service called secret.

- Secret: Used to store secret data

- Volumes: If our database pod is restarted, the data will be lost. That’s why we will use volumes. It attaches physical storage of machine to our pod files. The physical storage can be on our local machine or on a remote storage (outside of the K8S cluster)

Replicating pods :

If our pods die, we will lose all data. That’s why we will be replicating our pods on different nodes (servers). The replica will be connected to the same service.



The service has a permanent IP address and work as a load balancer so the service will catch the request and send it to the least busy pod. To create the replica, we will not create a second pod but we will write a blueprint for our pods and specify how many replicas of that pod. And this blueprint is called deployment.

- Deployment: Blueprint for the pods that will create and replicate. So in practice, we will not create pods but we will write deployments and there we will specify how many replicas we want.

N.B: Databases can NOT be replicated via deployment and the reason for that is that databases have a state which its data so they always need one central database to avoid data inconsistences. This mechanism is offered by another K8S component called stateful set.

- Stateful set: So, our database pods should be created using stateful sets and not deployment. And stateful set just like deployment will be taking care of replicating pods by making sure that the replicated databases are synchronized. Deploying stateful sets is more complicated than deploying deployment. That’s why a best practice is to host databases outside of the K8S cluster and just stateless (deployment) apps inside of the K8S cluster.

Explaining K8S architecture:

2 types of nodes that K8S operates on:

- master node

- slave node

**Worker machine :**

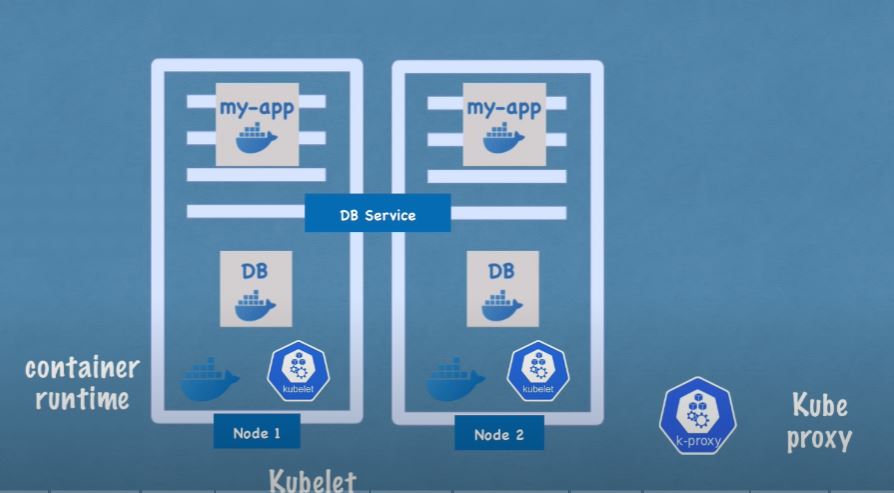
- Now we will consider we have 1 node (Worker machine) with 2 pods installed on it (application and database pod).

- 3 processes must be installed on every node that are used to schedule and manage those pods. Worker machine is a worker node that actually do the work.

a) The first process that needs to run it the **container runtime** (can be docker or other technologies as well)

b) but the process that schedule the pod and the container inside of it is **kubelet** that interacts with both the container and the node.

c) **Kube Proxy** also must be installed on each node which has logic and intelligent inside to forwards the request and make sur the communication also works in a performant way. It is responsible for the forwarding of request from service to pods. And to note that if the service receives a request, he can either send it to the actual pod or to its replica but normally he will send It to the actual pod.



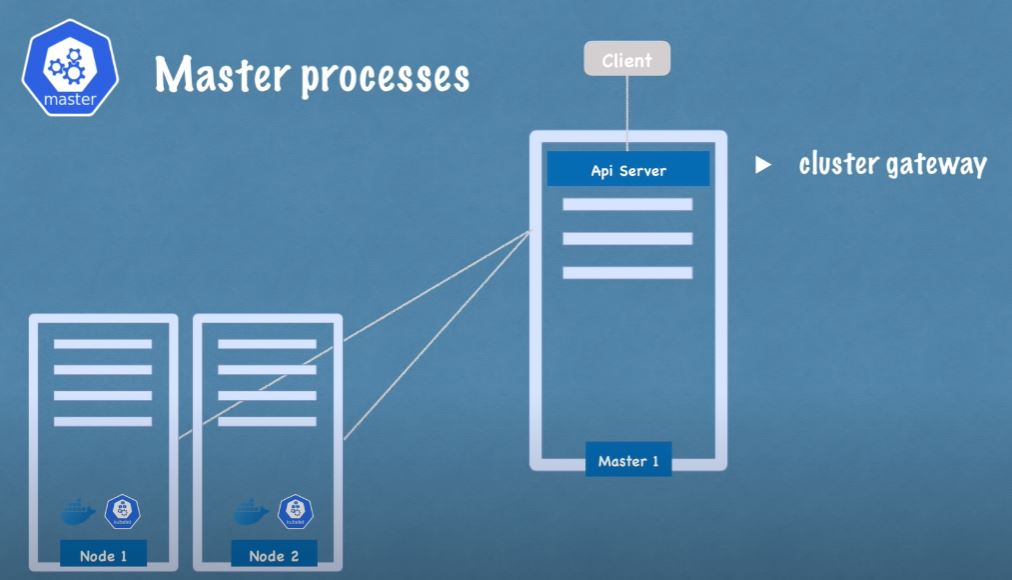
**How to schedule pod, monitor, re-schedule and restart pod, join a new node?**

All these theories are done by the master node.

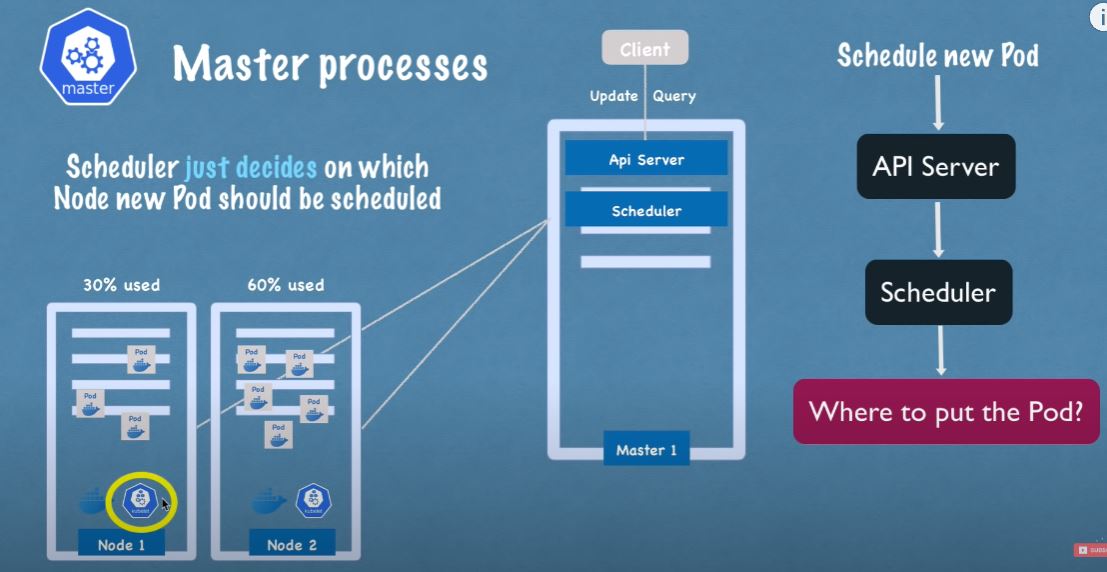
**Master node:**

There are 4 processes that run on every master node that control the cluster state and the worker nodes inside of the K8S cluster.

a) **API server:** Cluster gateway, used so that client can communicate with the master node. Plays the role of authenticator also, so only authenticated clients can have the access to access the cluster.

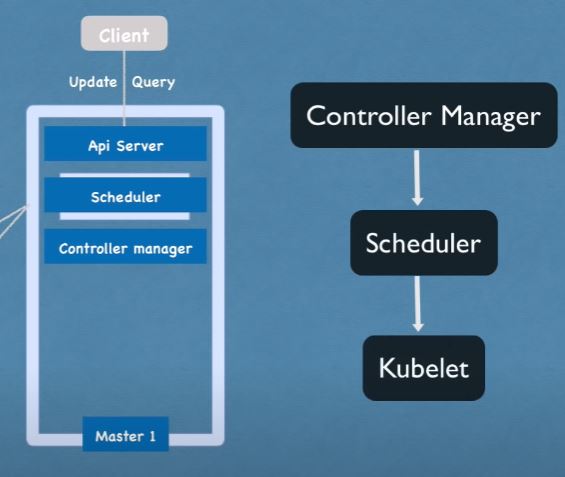
****

So it is the only entrypoint to the cluster.

b) **Scheduler:**

So the scheduler will only choose on which node the new pod will be scheduler but it is the role of the kubelet inside of each node that will schedule the pod on its node.

c) **Controller manager:** Used to monitor the pods on the nodes so if a pod on a worker node dies there must be a process that can re-schedule the pods and it’s the role of the controller manager so it detects state changing like crashing of pods for example.



d) **etcd:** It is a key value store for a cluster state. It is the cluster brain so all cluster changes get stored in the key value store. So, all the data responsible for the other components are stored in the etcd. So, the questions like (is the cluster healthy? what resources are available? Did the cluster state change?) All these informations are stored in this component.

**Replicating Master node:**



The API server on the two nodes (actual and replicated) will work as load balancer and the etcd data will be distributed on the 2 nodes.

Example of a Cluster Set-up :

Master nodes needs less resources than the worker nodes because the actual work is done on the worker nodes.

To add new Master/Node server:

1) Get new bare server.

2) Install all the master/worker node processes.

3) Add it to the K8S cluster.

Set local cluster with miniKube and kubectl :

**MiniKube:**

In fact, with miniKube we have 1 master process and 1 worker process and both run on same machine so it’s a test environment where we lack resources. Docker will be pre-installed on the node with miniKube. We need virtual box to let that happen so the node will run in the virtual box.

**Kubectl:**

Now that we have miniKube installed in our virtual box, we need kubectl so that we can create the components for the master and worker processes…

Run miniKube:

minikube start --vm-drive=none

Now here we are running minikube while configuring the hypervisor for miniKube, we used ‘none’ for our case we are running minikube inside of a VM.

Basic kubectl commands:

**See nodes:**

minikube kubectl -- get nodes

Here we will see our miniKube node. In fact, we are typing minikube before kubectl because kubectl is installed inside of the minikube module so in real world we will remove the minikube in the beginning of the command.

**See pods:**

minikube kubectl -- get pod

**See services:**

minikube kubectl -- get services

Here we can see the cluster IP address.

**Create deployment:**

In fact, in Kubernetes, pod is the smallest unit so we will not be creating one pod but we will be using the Deployment which is the abstraction over the pods.

minikube kubectl -- create deployment NAME --image=image [--dry-run] [options]

minikube kubectl -- create deployment nginx-depl --image=nginx

**See deployment and pod:**

minikube kubectl -- get deployment

minikube kubectl -- get pod

Now we will see that we have a deployment created and a pod also.

**Replicaset:**

Another layer managed by Kubernetes when creating the deployment is the replicaset and when can see it with the command:

minikube kubectl -- get replicaset

What we need to know is that we do not manage replicaset, in fact we write the complete blueprint with deployment with how much replicas and the replicaset will then be generated.

**Theory**

**Deployment** manages: **Replica** manages: **pod** manages: **container**

**Edit deployment:**

minikube kubectl -- edit deployment name\_of\_deployment

Here we will get an auto-generated config file with default values.

Once we edit the deployment, everything below that will be updated (Replicaset…)

**Logs:**

minikube kubectl -- logs pod\_name

**Get shell of pod:**

minikube kubectl -- exec -it pod\_name -- bin/bash

**Delete deployment:**

minikube kubectl -- delete deployment name\_of\_deployment

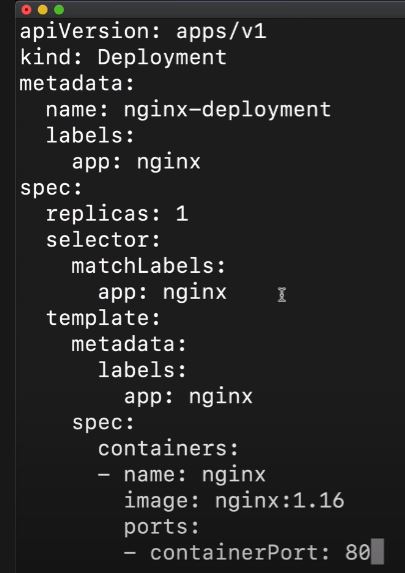
This command will delete the pod as well as the replicaset.

Automating the process of creating deployments:

In fact, we will be using a YAML config file to automate the process. We will use the following command to execute this config file:

minikube kubectl -- apply -f file\_name.yml

The config file will contain the following for example:



The ‘kind’ is the type so here we are saying it’s a deployment file. The ‘metadata’ contains the ‘name’ of the deployment and ‘labels’ will be discussed later. The first ‘spec’ that we see is for the deployment so we specify the replicas etc… The ‘template’ and everything below it is the blueprint for the pod.

Another example of a configuration file:

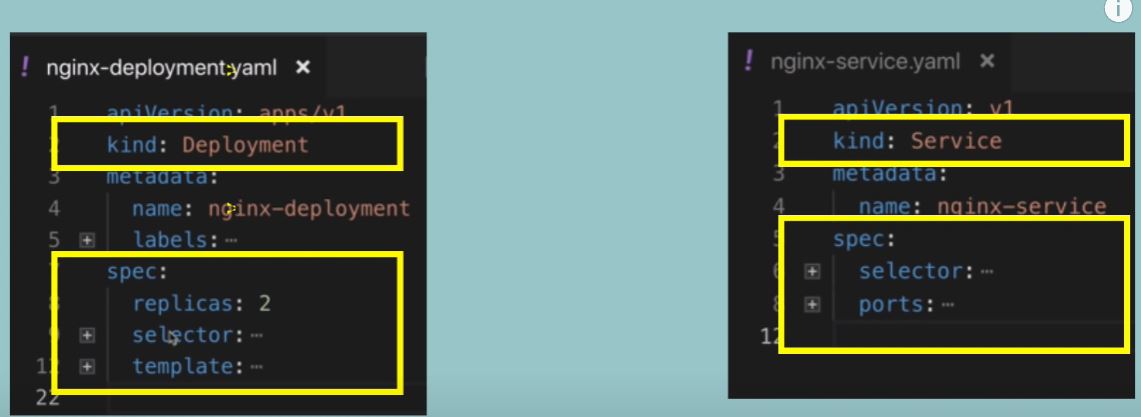
Each config file has 3 parts:

- Metadata

- Specification

- Status

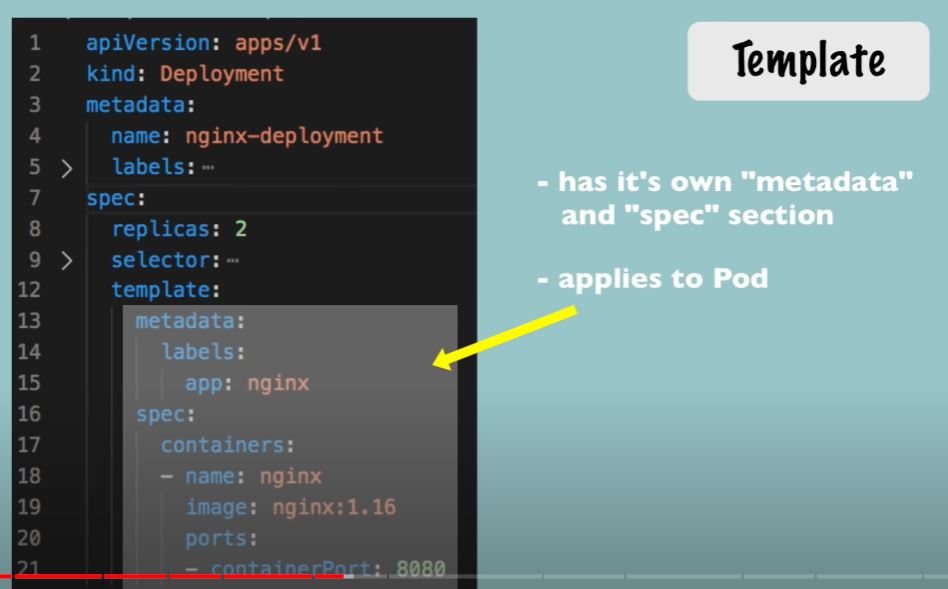
In this example we have 2 config files (one for deployment and other for service) and we are seeing 2 parts only (metadata and specification). Status will happen automatically once we execute the ‘apply’ command so it compares the actual state to the desired state in our configuration file. For example, if the actual state is 1 replica and we modified the config file and changed it to 2 replicas then executed the ‘apply’ command, the status is the part that will do this comparison and apply the changes. The question is how K8S can see the actual status? Well by using the **etcd** which is the brain and holds the status of the cluster.



**N.B :**

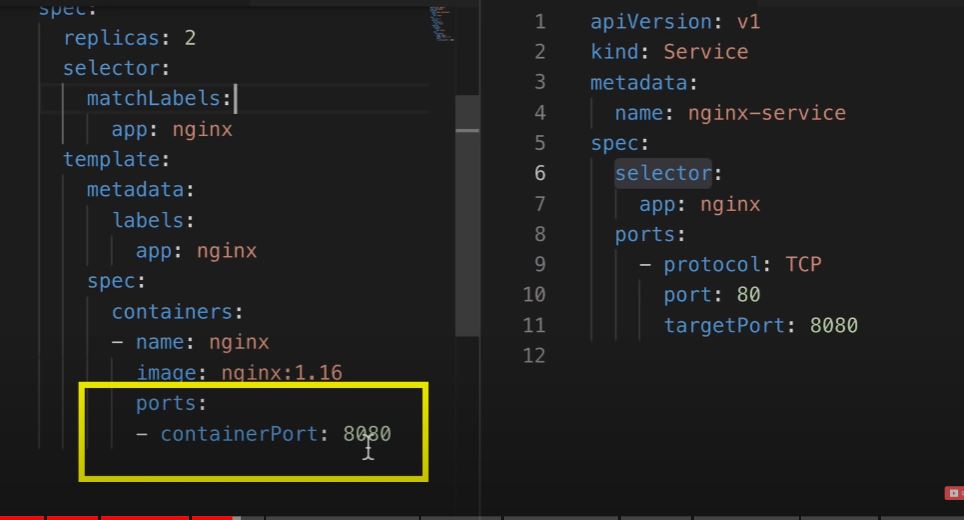
We can use an online yaml validator when writing configuration files which can help us with indentation human errors etc…

Here we can see that the ‘template’ contains the configuration of the pod. So we have a configuration file for the ‘deployment’ with its name and replicas etc… and inside of it we have another configuration file which is pod concerned.



Labels and selectors:

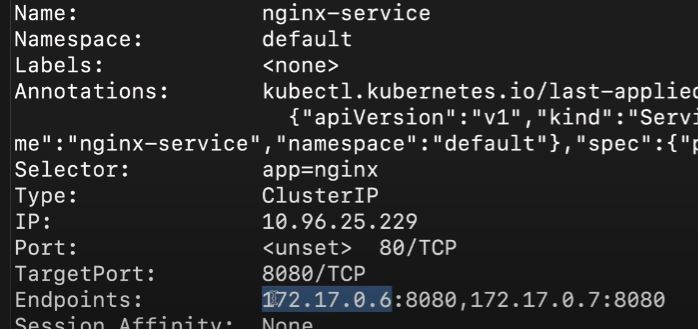
The connection between the deployment and services are done by using labels and selectors. The ‘metadata’ uses labels and the ‘spec’ uses selectors. In Deployment we make connection between the deployment and the pod. However, the service uses the label of the deployment label and not the pod label.



Another thing to understand is that the service has a dedicated port that targets the pod or container port. So when user will be making a request he will be actually transferring the request to the service port that redirects it to the pod port.

After applying both of the config files (Deployment and service), we can test and see if service is doing the redirection.

minikube kubectl -- describe service name\_of\_service



The endpoints are the ip addresses and ports of the pods that the service must forward the request to. Well, to know the ip addresses of our pods we can do the following:

minikube kubectl -- get pod -o wide