Kubernetes Notes

Iñaki Lakunza

June 1, 2024

Contents

1 Introduction

Kubernetes is an open source container orchestration framework, originally developed by Google.

It manages containers, be Docker containers or from some other technology.

This means that Kubernetes helps manage applications that are made up of a large number of containers, helping manage them in different environments, such as physical machines, virtual machines or cloud environments, or even hybrid deployment environments.

What problems does Kubernetes solve?

The containers offer the perfect host for small independent applications like microservices, so we need a way o managing a large number of containers across multiple environments.

What features do orchestration tools offer?

- High availability or no downtime.
- Scalability or high performance.
- Disaster recovery - backup and restore.

2 Kubernetes Components

- **Pod**: The basic component or the smallest unit of Kubernetes is a pod. It is an abstraction over a component.

  What pod does is to create a running environment, such as a docker container. It does this because Kubernetes wants to abstract away the container runtime or container technologies so that we can replace them if we want to. And also because we do not have to directly work with the Docker container, so **we only interact with the Kubernetes layer**.

  So we have an application Pod, which is our own application and, that will maybe use a database pod.

  A pot is usually meant to run one application container inside of it, even if it is possible to run more than one in one pod.

  So, in our Node we have until now two pods, one which contains the app and the other one containing the database.

  Each Pod gets its own IP address, NOT THE COTAINER, but the pod, and each Pod can communicate with each other using that IP address.

  It is important to take into account that **these IP addresses are EPHIMERAL, which means that they can die very easily**. So, this means that if we for example lose the database container because the container crahshed because the application chrashed inside or because the nodes, the server that we are running them on, run out of resources, the Pod will die.

  If this happens **a new Pod will be created in its place and it will get assigned a new IP address**. This is not convenient if we are communicating with the database using the IP address because now we would have to adjust it every time the Pod restarts.

Because of that, another component of Kubernetes called **Service** is used.

- **Service**: A Service is a STATIC IP address or permanent IP address that can be attached to each Pod.

  So our app Pod will have its own service and the database Pod will have its own service.

  And, **the life cycles of the Pod and the Service are not connected**, so even if a Pod dies and gets replaced, the Service will not change.

  We want our application to be accessible through a browser, and for this we would have to create an external Service.

  So, a **External Service** is a Service which opens the communication from external source.

  But obviously we do not want our database to be open to the public requests. So that is why we have to create something called **Internal Service**.

  A Internal Service is a type of a Service that we specify when creating one.

  When we are doing test, we want our address to be easy to work with, a one which has the port number of the service.

  But, for the final application we want to work with a secure protocol and the domain name (https://my-app.com), so, for that there is another component of Kubernetes called **Ingress**.

  So, instead of Service, the request first goes through Ingress and it does the forwarding then to the service.

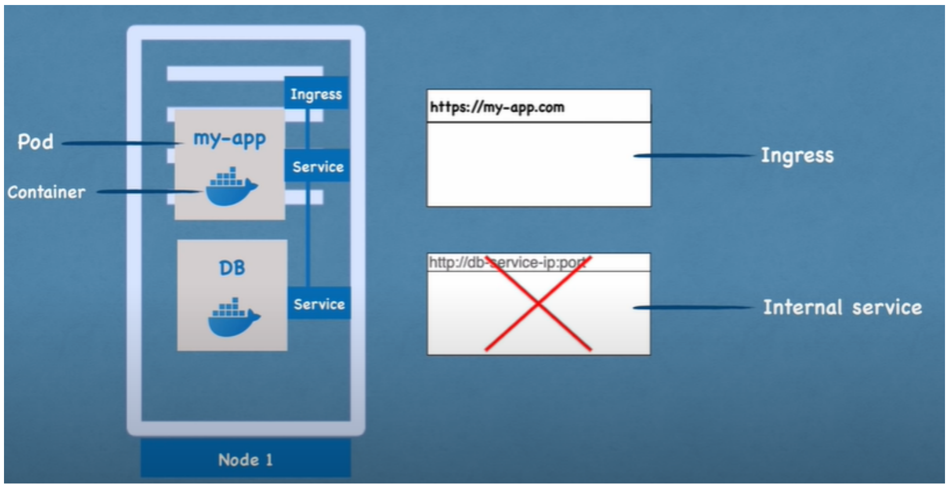


Figure 1:

- **ConfigMap and Secret**: Pods communicate with each other using a Service, so our application will have a database endpoint that it uses to communicate with the database. But where we configurate usually this database URL or endpoint is in application properties file or some kind of external environmental variable. But usually is done inside of the built image of the application.

So, for example, if the endpoint of the Service or Service Name changed, we would have to adjust that URL in the application, so we would have to re-build the application with a new version and we would have to push it to the repository, and now we would have to pull that new image to our Pod and restart everything.

So this is a bit tedious for a very small change like the database URL. For this purpose, Kubernetes has a component called **ConfigMap**.

What is does is our external configuration to our application. So the ConfigMap would usually contain configuration data like URLs of the database or some other services that we use.

And in Kubernetes we just connect it to the Pod, so that the Pod gets the data that the ConfigMap contains (refers to).

And now, if we change the name of the Service, the endpoint of the Service, we just have to adjust the ConfigMap and that's it, we do not have to build a new image and go through the whole cycle.

Part of the external configuration can also be the database username and password, which may also change in the application deployment process. But putting a password or other credentials in a ConfigMap, in a plain text format would be insecure, even though it is an external configuration. So, for this purpose, Kubernetes has another component called **Secret**.

Secret is just like a ConfigMap but the difference is that it is used to store secret data credentials, for example. And they are not stored in plain text format, they are stored in *base64 encoding* format.

So, the Secret would contain things like credentials. And just like the ConfigMap, we connect it to the Pod so that it will be able to see and work with the stored information in the Secret.

We can actually use the data from ConfigMap or Secret using **Environmental Variables**, or even as a properties file.

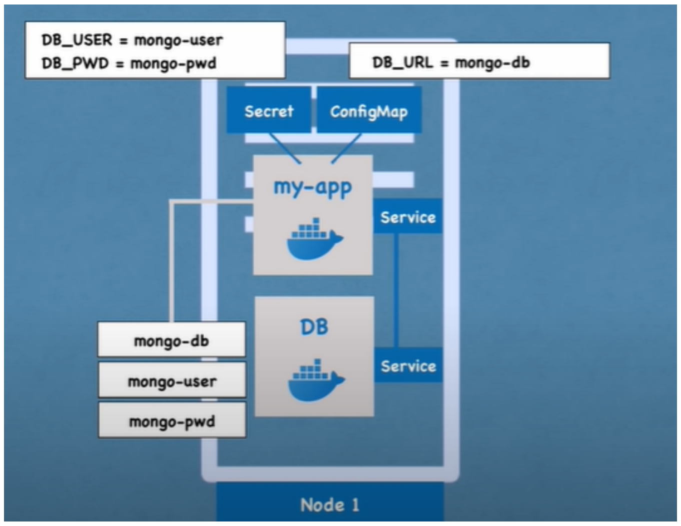


Figure 2:

- **Volumes**: Data Storage is a very important concept we will work with. In our example we have our Database Pod that our application uses and it has some data.

  If the database container or the Pod gets restarted the data would be gone. And we want our database data or log data to be persisted reliably in a long term. So, the way we can do this is by using **Volumes**.

  Volumes work by attaching a physical storage on a hard drive to our Pod, and this storage could either be on our local machine (so it would be on the same server node where the Pod is running), or it could also be in a remote storage (meaning that it would be outside the Kubernetes cluster, like a cloud storage or our own premise storage, which is not part of the Kubernetes cluster, so we just have an external reference on it).

  So now, by using the Volume we would be able to keep our data even if the Pod gets restarted.

  The Kubernetes cluster does not manage any data persistance, so that means that we, as a Kubernetes user, are responsible for backing up the data and replicating and managing it, and making sure that it is kept on a proper hardware, ...

- **Deployments and StatefulSets**: If we have our application restarted or it crashes it would mean that we would have a downtime where a user will not be able to reach the application.

  So, instead of relying of one application Pod and one database pod we will **replicate everything in multiple servers**.

  So we would have another node where a replica or clone of our application would run, which will also be connected to the Service.

  We have said that the Service is like a persistent IP address with a DNS name, so that we do not have to constantly adjust the endpoint when a Pod dies.

  A Service is also a load balancer, which means that the service will catch the request and forward it to the Pod that is less busy. SO the service has both of these functionalities.

  But, **in order to create a second replica of our application we would not create a second Pod, we would instead define a BLUEPRINT for our application Pod and specify how many replicas of that Pod we would like to run**.

  This component is known as **Deployment**, and in practice we would not be working with Pods, we would be creating Deployments, because there we can specify how many replicas we want, and we can also scale up or down the number of replicas or Pods we need.

  So, we have said that a Pod is a layer of abstraction on top of the containers and the Deployment is another abstraction on top of Pods, which makes it more convenient to interact with the Pods, to replicate them and configurate them.

  So in practice we would mostly work with Deployments and not with Pods.

  On the other hand, if our Database would die we could not work with it, so we would need another replica. **But we can't replica a database using a Deployment** because the database has a state which is its data, meaning that if we have clones or replicas of the database they would all need to access the same shared data storage. And there we would need some kind of mechanism that manages which Pods are currently writing to that storage or which Pods are reading from that storage in order to avoid data inconsistencies.

This feature is offered by another component called **StatefulSet**, which is specifically meant for applications like databases. So **we should work with StatefulSets when we work with databases, and not with Deployments**,

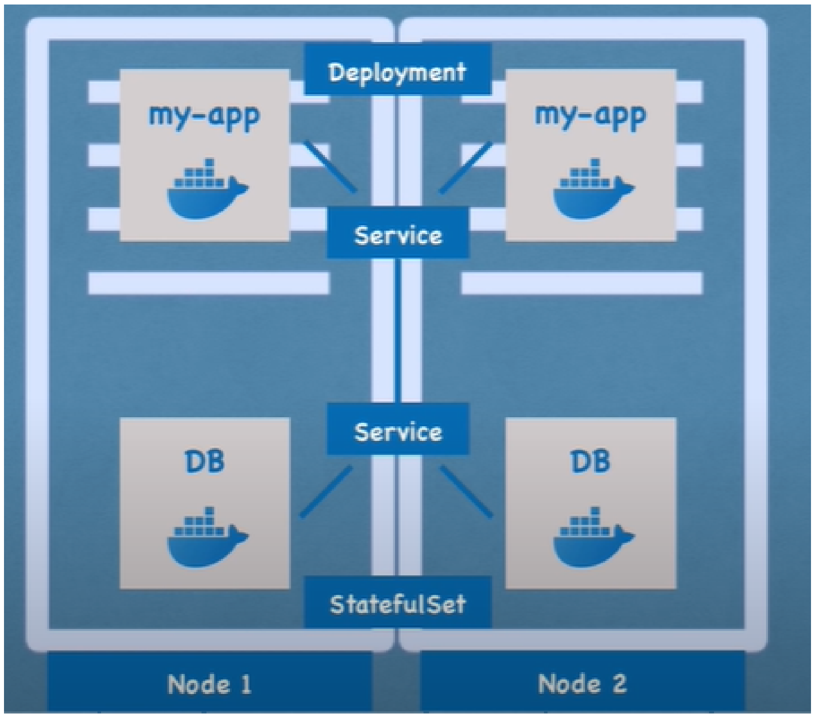


Figure 3:

But we have to take into account that Deploying StatefulSets is not easy. So, it **is common practice to host database applications outside of the Kubernetes cluster and just have the deployments or stateless applications that replicate and scale with no problem inside the Kubernetes cluster, and communicate with the external database**.

3 Kubernetes Architecture

We will again use the previous example, where in a node we have an application and a database.

We know that each node will have multiple applications Pods running on that node and the way Kubernetes does it is by using 3 processes that must be installed in every node that are used to schedule and manage those parts.

So nodes are the cluster servers that actually do that work, so that is why they are sometimes called **worker nodes**.

The first process that needs to run every node is the **Container Runtime**. Beacause application Pods have containers running inside Container Runtime needs to be installed on every node. But the process that actually schedules those Pods and the containers that are on it is named **Kubelet**, which is a process of Kubernetes itself unlike Container Runtime that has Interfece with both Container Runtime and the machine, the node itself.

At the end of the day **Kubelet is responsible for taking that configuration and running a Pod or starting a Pod with a container inside, and then assigning resources from the node to the container**. like the CPU RAM storage resources.

Usually a Kubernetes cluster is made up of multiple nodes which also must have Container Runtime and Kubelet services installed. And we can have hundreds of those worker nodes which will run other Pods and containers and replicas of the existing Pods, like in our example the application and the database Pods.

And the way that the communication between them works is using Services, that as we saw, are sort of a load balancer that catches the request directed to the Pod of the application and then forwards it to the respective Pod.

And the third process that is responsible for is for forwarding requests from Services to Pods. This is done by **Kube Proxy**. It also must be installed in every node.

Kube Proxy has actually an intelligent forwarding logic inside that makes sure the coommunication also works in a performant way with low overhead. For example, if our application replica is making a request to the database instead of the Service just randomly forwarding the request to any replica, it will actually forward it to the replica that is running **on the same node** as the Pod that initiated the request. Thus, avoiding the network overhead of sending the reques to another machine.

So, to summarize, **Kubelet and Kube Proxy must be installed on every Kubernetes worker node along with an independent Container Runtime, in order for the Kubernetes cluster to function properly**.

How do we interact with this cluster?

How do we decide which nodes a new application Pod or Database Pod should be scheduled, or if a replica Pod dies what process actually monitors it and then reschedules it or restarts it.

Or when we add another server how does it join the cluster to become another node and get Pods and other components created on it.

All these Managing Processes are done by MASTER NODES

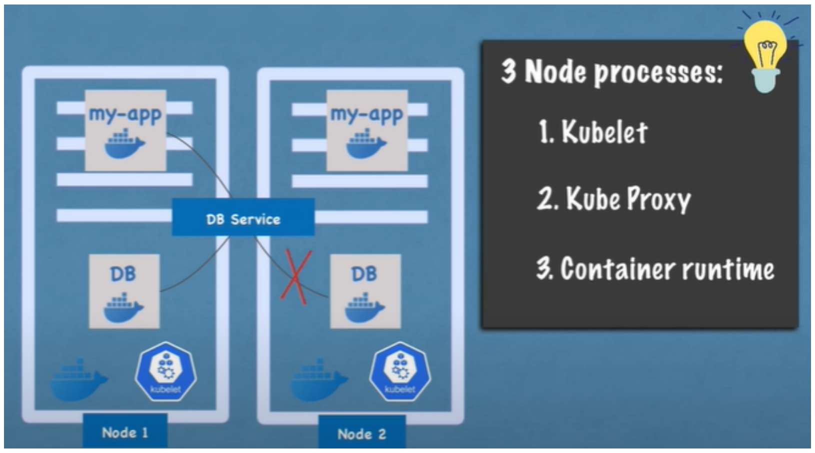


Figure 4:

3.1 Master Processes

Master Servers or Master Nodes have completely different processes running inside. They have 4 processes that run for controlling the cluster state and the worker nodes.

- The first service is **API Service**. When we want to deploy a new application in a Kubernetes cluster we interact with the API server using some client. It could be UI like the Kubernetes dashboard, could be a command line tool like Kubelet or a Kubernetes API.

  So **API Server** is like a cluster gateway, which gets the initial reques of any updates into the cluster or even the queries from the cluster and it also acts as a gatekeeper for authentication to make sure that only authenticated and authorized requests get through the cluster.

  That means whenever we want to schedule new Pods, deploy new applications, create a new service or any other components we have to talk to the API Server on the Master Node and then the API Server will have to validate the request and if everything is fine then it will forward our request to other processes in order to schedule the Pod or create the component we have requested.

  Also if we want to query the status of our deployment or the cluster health, ..., we make a request to the API Server and it will give us the response, which is good for security because we just have one entry point into the cluster.

- **Scheduler**: If we send an API server a request to schedule a new Pod, the API Server after it validates the request it will handle it over to the Scheduler in order to start that application Pod on one of the worker nodes.

  Instead of just randomly assigning to any node, the Scheduler has a intelligent way of deciding in which specific Pod/component it will be scheduled.

  So first it will look at the request and see how much resources the application that we want to schedule will need: how much CPU and RAM; and then it is going to go through the worker nodes and see the available resources for each one of them.

  And it will handle the new Pod on that node.

The Scheduler just decides on which node the new Pod will be scheduled. The process that actually does the scheduling is the Kubelet.

- **Controller Manager**: When pods die on any node there must be a way to detect that the nodes died and reschedule those Pods as soon as possible.

  So what the Controller Manager does is detect state changes like crashing of Pods, for instance.

  So, when Pods die, the Controller Manager detects them and tries to recover the cluster state as soon as possible, and for that it makes a request to the Scheduler to reschedule those dead Pods in the same cycle.

  Again, the scheduler makes the decisions based on the resources calculation and makes the reques to the corresponding Kubelets on those worker nodes to restart them.

- **ETCD**: The etcd is a key-value store of a cluster state. We can think of it as a cluster brain.

  Every change in the cluster, such as a new Pod scheduling or the death of a Pod for instance, will get saved or updated into this key-value store.

  All the other processes are based on the information given by the etcd.

  What is not stored in the etcd key-value store is the actual application data

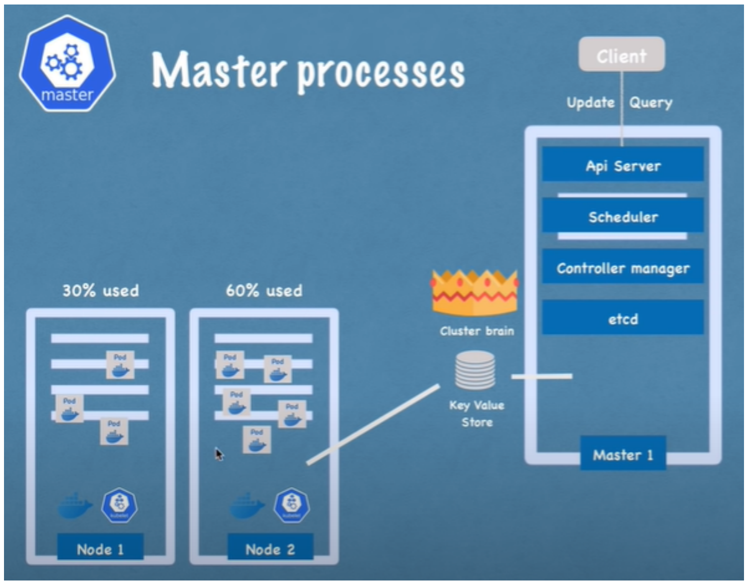


Figure 5:

In practice the Kubernetes cluster is usually made up of multiple Master nodes, where each master node runs its master processes.

The API Server is load balanced, it is distributed accros different master nodes.

Usually people uses 1 master node for 2 worker nodes, but it depends on the needs.

4 Minikube and Kubectl - Local Setup

Usually, in a Kubernetes world, when we are setting up a production cluster we will have multiple master nodes, at least 2 in a production setting and we would have multiple worker nodes.

And as we know master nodes and worker nodes have their own separate responsibilities.

If we want to test something rapidly in our local machine, but it is untractable to reproduce all the setting in just one local machine.

So that is why there exists **Minikube**. It is a 1 node cluster where the master processes and worker processes both run on 1 node. And this node will have a Docker Container Runtime pre-installed, so we will be able to run the containers or the Pods with the containers on this node.

And the way it is going to run on our laptop is through a virtual box or some other hypervisor.

So, basically, Minikube will create a virtual box on our laptop and the nodes will run there. **We can test Kubernetes this way in our local setup**.

So, once we have a mincluster on our local machine, we need some way to interact with the cluster (create components, configurate, ...), this is where we use **Kubectl**.

KUBECTL

Now that we have our virtual node on our local machine that represents a Minikube, we need a way to interact with the cluster, we need a way to create Pods and other Kubernetes components on the node.

Kubectl is a command line tool for Kubernetes cluster. One of the master node's processes is the API Server, which is the main entry point into the Kubernetes cluster. So, if we want to do anything with Kubernetes we have to talk to the API Server. We can do this using the UI, the API or the CLI, which is Kubectl.

Once Kubectl submits commands to the API Server the work processes on the Minikube node will make it happen.

Kubectl is not jsut for the Minikube cluster, if we have a cloud cluster or a hybrid cluster Kubectl is the tool used to interact with the setup.

Once everything has been installed we can easily create and start the cluster with the following line in a cmd terminal.

We also have to tell Minikube which hypervisor it should use. (In the tutorial she uses **hyperkit** as the hypervisor, but I will use Docker, so, the first line is the one from the tutorial and the second one the one the terminal is telling me to use and the third one the one that I have used)

```
minikube start --vm-driver=hyperkit
minikube start --driver=docker
minikube start --vm-driver=docker
```

Then, we can check the status of the nodes with the following line:

```
kubectl get nodes
```

We can also get the status:

```
minikube status
```

We can also check which version of Kubernetes we have with the following line:

```
kubectl version
```

From now on we will be interacting with the Minikube cluster useing the Kubectl command line tool. So, Minikube is basically for the start up and for deleting the cluster, but everything else is done through Kubectl.

5 Main Kubectl Commands

As we said, once we have the cluster set up we are just going to use Kubectl to work with it.

We can check the Pods and the Services with the following lines:

```
kubectl get pod
kubectl get services
```

We will use the **create** command to create components. We can use **kubectl create -h** to see the different options, and as we will see there is no Pod option.

This is because the Pod is the smallest unit, but **in practice we are not working with the Pods directly, we are working with an abstraction layer over the Pods which is called DEPLOYMENT**. So this is what we will be creating.

We have to give a name to the Deployment. And then there are different options, such as the **image** and **dry-run**.

The Pod needs to be created based on a certain image.

For the example, we will create a **nginx** Deployment (so it will download the image from DockerHub):

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

So now if we use the line **kubectl get deployment** we will get its information.

We will see that it is not ready. And we can also use **kubectl get pod** to get the Pod information. We will see that we will get one, with the name of our deployment and some random hash afterwards.

And we will see that it says *ContainerCreating* in its status, so it is not ready yet.

But once it has been created we will see how the status changes to *Running*.

Once we create the Deployment it will have all the information (blueprint) for creating the Pod.

We can also get information of the replica set using **kubectl get replicaset**

So, how the layers of abstraction work iw the following:

- The Deployment manages a ReplicaSet
- The ReplicaSet manages all the replicas of the Pod
- The Pod is an abstraction of the container

Everything below the Deployment is handled by Kubernetes.

If we want to edit something we will have to go through the Deployment, so, using the name we gave to the deployment previously:

```
kubectl edit deployment nginx-depl
```

And we will get a auto-generated configuration file with default values.
We can scroll down and for instance change the version of the image we are using.

And then, if we try **kubectl get pod** we will see that the previous Pod is being terminating and the new one is running. And same with the ReplicaSet.

Debugging Pods
We can log to some Pods (for the case of nginx not, but yes with Mongo for instance), for that we have to create the deployment and then log in using the associated Pod name, for instance:

```
kubectl get pod

(it will show the info)

kubectl logs _______
```

We can also use **kubectl exec**, it gets the terminal of the application container we are using (for instance with Mongo):

```
kubectl exec −it _____ −− bin/bash
```

And this way we will access the terminal.

And for exiting we can just use **exit**.

We will want to delete the Pods, so we will have to delete the deployment, using its name (**the name of the deployment, not the Pod**):

```
kubectl delete deployment ____
```

And we will see how the Pod and the replica will be deleted.

If we want to change a large number of configurations the easiest way to go is to create a default Deployment and then to create a yaml file specifying the custom configurations, then we can just use the following line:

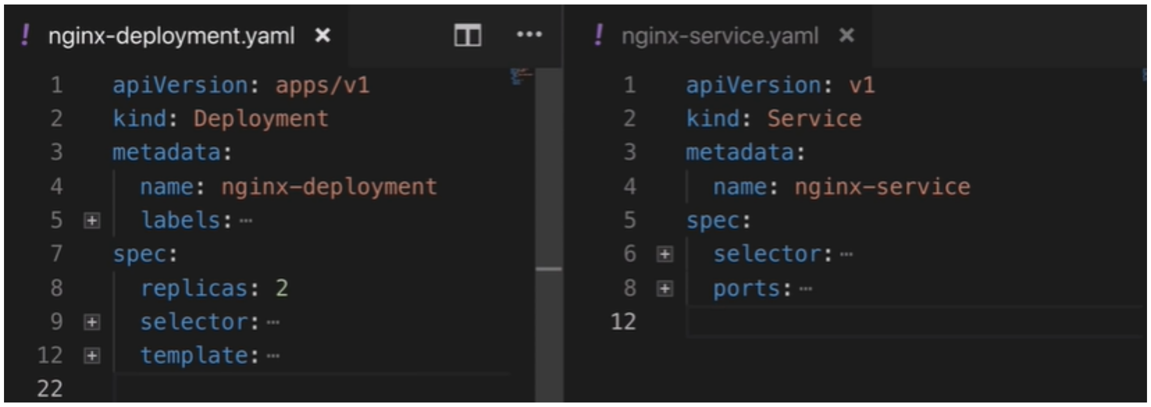
```
kubectl apply −f (config_file_name).yaml
```

The easiest way to do this is by creating a configuration file using as copy the default one and changing it there, for instance (there we would copy the default file):

```
touch nginx−deployment.yaml
vim nginx−deployment.yaml
```

6 Kubernetes YAML File Configuration

6.1 YAML file parts



nginx-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels: ⋯
spec:
  replicas: 2
  selector: ⋯
  template: ⋯
```

nginx-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector: ⋯
  ports: ⋯
```

Figure 6:

The above are examples of a Deployment and a Service configuration files.
Every configuration file in Kubernetes has 3 files:

- **Metadata**: The Metadata of the component we are creating, the name of the component and the Specification.

- **Specification:** In the Specification we put every kind of configuration that we want to apply for that component.

  apiVersion will specify the API version of each component, each specific component will have its own.

  And the **kind** will specify the kind of component we have.

  Inside the specification part the attributes will be specific for each component.

- **Status**: This part will be automatically generated and added by Kubernetes.

  Kubernetes will always compare what is the desired state and the actual state of the component. And if the desired and actual state of the component do not match then Kubernetes knows that there is something to be fixed, so it will try to fix it.

  This is the basis of the self-healing feature that Kubernetes provides.

  This information comes from the **etcd**, which stores the cluster data.

The data representation is very straightforward, and **it must be done with YAML, and we have to take care of the identation since YAML files are very strict with it**.
We can use a YAML online validator if we want to be sure it is well-written.

Usually, the YAML file is stored with the code.

6.2 Blueprint for Pods

As we saw, the Deployment manages the parts that are below them, so whenever we edit something in the Deployment it cascades down to the ReplicaSet, then the Pods and then the Containers.

And so, whenever we want to create some Pods we have to create a Deployment and then it will take care of the rest by itself.

All of this happens in the part **template** of the configuration file, which is kind of another configuration file, which also has its own metadata and specification.

So, the Pod will have its own configuration file inside the Deployment's configuration file.

6.3 Connecting Components

The way the connection is established is using **labels** and **selectors**.

The Metadata part contains the labels and the specification part contains the selectors.

In the Metadata we give components a key-value pair, it can be any key-value pair and that label sticks to that component.

And we tell the deployment to connect or to match all the labels with the key-value pair we have chosen to create that connection.

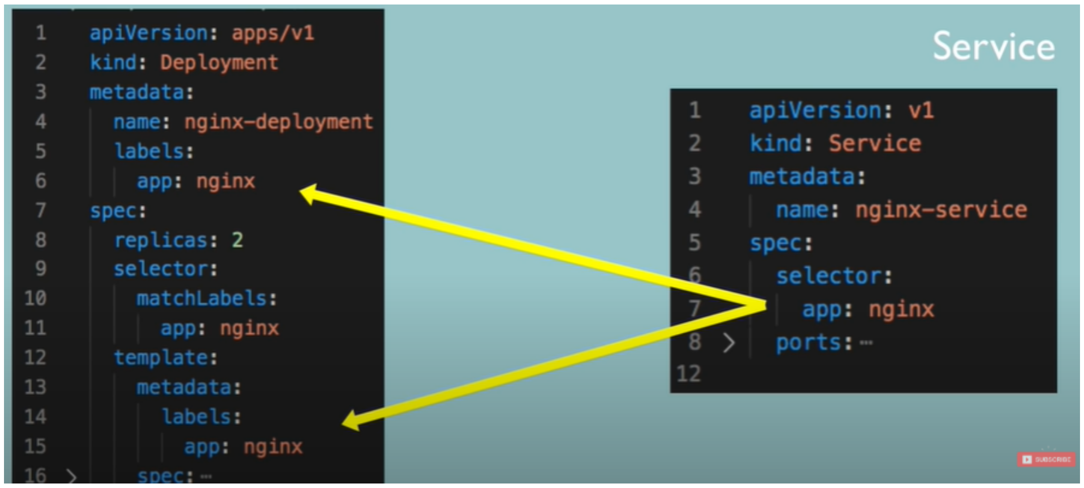
For instance:

```
(Component 1)
metadata:
    name: nginx—deployment
    labels:
        app: nginx

spec:
    replicas: 2
    selector:
    matchLabels:
        app: nginx
    template:
        metadata:
            labels:
                app: nginx


(Component 2)
    selector:
        matchLabels:
            app: nginx
```

The ports must also be considered in the Service and the Pod.


Service

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:…
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:…
```

Figure 7:

The Service has a port where the Service itself is accessible at, so if annother component sends a request to it, it must be accessed through that port.

But **the Service needs to know through which Pod it should forward the request, but also at which port is that Pod listening**. And that is the **target port, so this one should match the Container port**

So, we will create both Deployment and Service:

```
kubectl apply —f nginx—deployment.yaml
kubectl apply —f nginx—service.yaml

kubectl get pod
kubectl get service (we will see the default service that is always,
                    and ours, and the port will be seen)

kubectl describe service nginx—service (we will see all its info)
```

We will see that it has an attribute called **Endpoints**, so there we will see the address, but we have to look the Pod address too, and that does not appear when we use **kubectl get pod**, so we have to use the following:

```
kubectl get pod —o wide
```

So now more information will appear, and we have to check that they match.

If we want to configure the configuration file, we will first save the default one using the following line:

```
kubectl get deployment nginx—deployment —o yaml > nginx—deployment—result.yaml
```

So we can open our text editor and do the modifications there.

7 Kubernetes Namespaces

In a Kubernetes cluster we can organize resources in Namespaces, so we can have different Namespaces in a cluster.

A namespace is a virtual cluster inside a Kubernetes cluster.

When we create a cluster Kubernetes gives us Namespaces out of the box We can see these writting **kubectl get namespace**.

Here are the default ones explained:

- **kubernetes-dashboard**: Only with Minikube, we will not have this in a standard cluster.
- **kube-system**: It is not meant for our use, we should not create anything or should not modify anything there. It contains the system processes.
- **kube-public**: It contains the publicly accessible data, it contains a ConfigMap with cluster information.
- **kube-node-lease**: It contains information about the **heartbeats of the nodes**.

  Each node gets its own object that contains the information about that node's availability.
- **default**: Is the one we will be using to create the resources at the beginning if we have not created any new Namespace

If we want to create a new Namespace we have to use the following line:

```
kubectl create namespace (name)
```

Or we can also use a Namespace configuration file for creating the Namespace, which is a better way of creating them since we would have have a history in our repository of what resources we created in a cluster.

Why use Namespaces

If we only use the default Namespace provided by Kubernetes, and we create all the resources in that Namespace, if we have a complex application that has multiple Deployments which create replicas of many Pods, and we have resources like Services and ConfigMaps, ... Very soon our default Namespace will be filled with lots of components, and it will be really diffficult to have an overview of what it in there, specially if we have mutliple users creating stuff inside.

So, a better way is to **group reousrces into Namespaces**, for instance we can group the ones belonging to the Database, then we can group the ones belonging to the Monitoring, ...

It is also a good idea to **use Namespaces if we have multiple teams**.

Or we can also **differentiate the staging and development environments**.

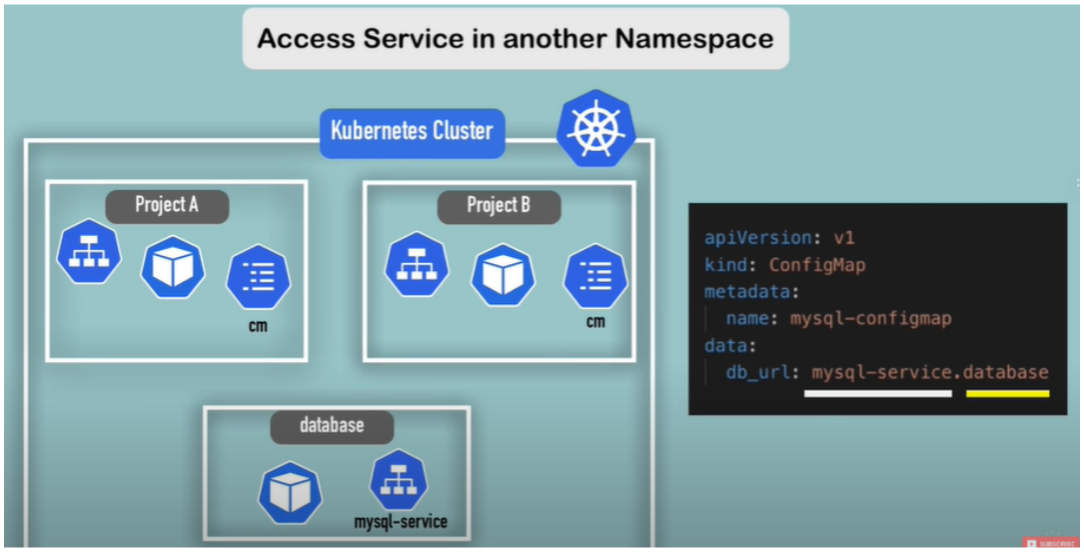
Or we can also have **different versions of the production**, the actual and the one that will be the future one.

Or we can also **limit the resources and access to namespaces when we are working with multiple teams**.

7.1 Things to considerate

- **We can't access most resources from another Namespace**. For instance, we can have different projects separated in Namespaces, and even if they need the same ConfigMap each one should have its own, so we would have to get a copy and put it on the other one too. So, **each Namespace must define its own CongiMap and Secret**.

  But, **we can share a Serive acrros Namespaces**.



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: mysql-configmap
data:
  db_url: mysql-service.database
```

Figure 8:

- **There are components which can't be created within a Namespace**, they live globally in the cluster so that we can not isolate them in a certain cluster: Volume, Persintent Volume and Node.

7.2 Creating Components in a Namespace

Until now we have not defined any Namespace when creating our components.

By default it will be created in the default namespace.

But, if we want to create it in a chosen namespace we have to use the following line:

```
kubectl apply -f mysql-configmap.yaml --namespace=my-namespace
```

Or we can also do it inside the configuration file, adjusting the **namespace** attribute in the metadata section.

This second option is better.

7.3 Change Active Namespace

Kubectl does not have a direct solution for this problem, but there is a tool called **Kube NS**, and we have to install it.

For that we will have to install **kubectx**, and then use **kubens**, which comes inside.

And for changing a namespace we will have to:

```
kubens (it will highlight the one which is active,
            and change it with the following line)
kubens my—namespace
```

8 Kubernetes Ingress

Let's image a simple Kubernetes cluster where we have a Pod of our application and its correspondins Service.

So, the first thing we need for a UI application is to be accessible through the browser so that external requests are able to reach our application.

One easy way to do this is through an external service where we can access the application using the HTTP protocoal, the IP address of the node (the port).

However this is only good for test cases, if we want to try something very fast. But this is not what the final product should look like.

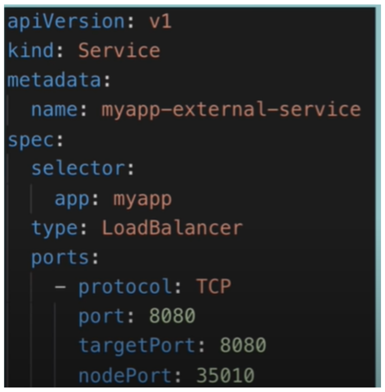
Instead of having the IP address the final product should have the domain name (not http://124... , but instead http://my-app.com).

The way to do this is by using the component known as **Ingress**. So we will have our app's Ingress and, instead of an external service, we would have an internal service. So we would not open our application through the API address and the Port.

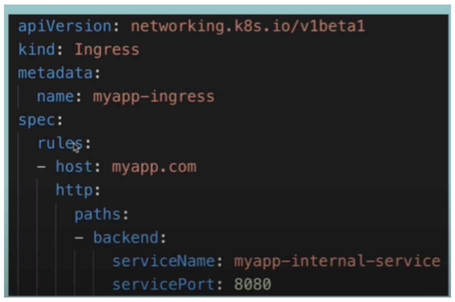
And now if the request comes from the browser it is going to first reach the Ingress and then the Ingress will redirect it to the internal service and eventually end up with the Pod.

8.1 Example YAML File: External Service

From the **kind** attribute we see that we have a Service, of **type** LoadBalancer (this means that we are opening it to the public by assigning an external IP address to the service).
We can also see the Port number which with the user can access the application at



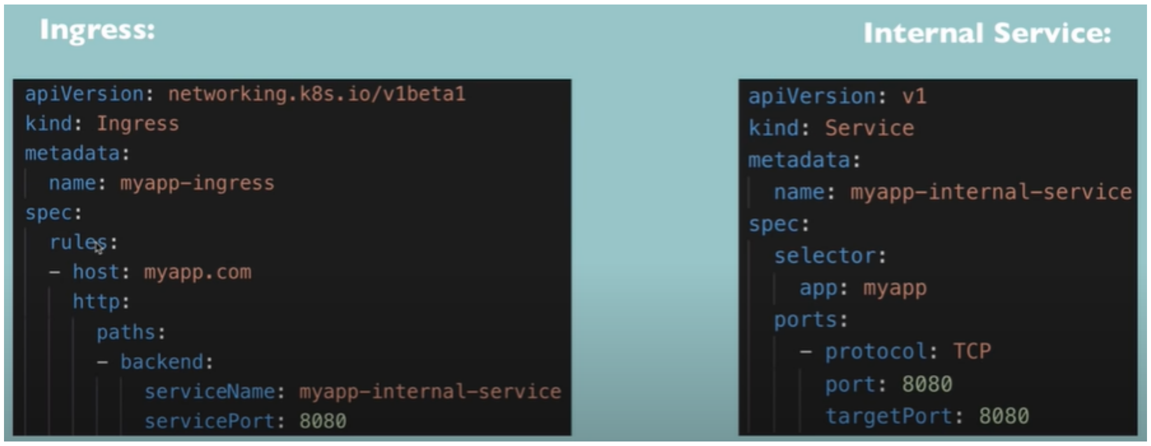
```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-external-service
spec:
  selector:
    app: myapp
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
      nodePort: 35010
```



```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.com
    http:
      paths:
      - backend:
          serviceName: myapp-internal-service
          servicePort: 8080
```

Now we see that we have an Ingress instead of a Service.
And in the Specification part of the configuration file, where all the configuration happens, we have the **rules** or **routing rules**.
This basically defines that the main address or all the requests to that host must be forwarded to an internal service.
Here we can see the host that the user will enter in the browser. And in the Ingress we define a mapping, so when the request to that host gets issued we will redirect it internally to the Service. The **paths** basically mean the URL path, so everything after the domain name.
The HTTP attribute does NOT correspond to the http protocol. It is a protocol that the incoming request gets forwarded to the internal service.

Here **backend** is the target where the incoming request will be redirected and the **serviceName**



```yaml
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: myapp-ingress
spec:
  rules:
  - host: myapp.com
    http:
      paths:
      - backend:
          serviceName: myapp-internal-service
          servicePort: 8080
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: myapp-internal-service
spec:
  selector:
    app: myapp
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 8080
```

Figure 9:

should correspond to the internal service name (the one which can be seen in the metadata of the internal service).
And the **servicePort** should be the internal service port.

As it can be seen, the only difference between the external and the internal service is that **in the internal service we do not have the third port, which is the nodePort**, and the type is a default type, not a load balancer, it is a clusterIP type.

Then, the **address which appear in the host must be a valid domain address, and we should map that domain anme to the IP address of the node that represents and entry point for our Kubernetes cluster**.

8.2 How to configure Ingress in our Cluster

We know that the information must first go through the Ingress, then through the Service and then will reach the Pod.
In order to create the pipeline, only creating the Ingress will NOT be enough for Ingress routing rules to work. What we need in addition is an implementation for Ingress, which is called **Ingress Controller**.

So the first step will be to install an Ingress Controller, which is basically another Pod or another set of Pods that run in our node in our Kubernetes cluster and **does evaluation and processing of Ingress rules**.

The function of the ingress controller is to define all the rules and so manage all the redirections.
So this will be the entry point in the cluster for all the requests to that domain or subdomain rules that we have configured.
And this will evaluate all the rules.

We have to decide which one of the many third-party implementations we should use (there is one from Kubernetes itsefl, called K8s Nginx Ingress Controller).

we have to consider the environment on which our cluster runs. If we are using some cloud service provider, we will have a cloud balancer that is specifically implemented by that cloud provider and external requests coming from the browser will first hit the load balancer and that load balancer will then redirect the request to the Ingress controller (we can configure it in different ways).

And the advantahe of using the cloud provider is that we do not have to implement a load balancer ourselves.

On the other hand, **if we are implementing our Kubernetes cluster in a real machine, not in the cloud, we will have to configure the entrypoint ourselves.**.

Generally speaking, either inside of the cluster of outside we will have to provide an entry point.

8.3 Install Ingress Controller in Minikube

The first thing is to enable the addons:

```
minikube addons enable ingress
```

This automatically starts the K8s Nginx implementation of Ingress Controller. So with just one command the Ingress Controller will be configured in our cluster.

And if we use the following line, we will see that there is an Ingress Controller:

```
kubectl get pod -n kube-system
```

So, one we have the Ingress Controller installed, now we can create an Ingress Rule that the controller can evaluate.

```
kubectl get all -n kubernetes-dashboard
```

Here we will see all the components that we have in the Kuberntes dashboard.

And since we already have the Internal Service dashboard and the Pod that is running, we can now create an Ingress Rule in order to access the Kubernetes dashboard using some host name.

So we will create an Ingress for the Kubernetes dashboard. We will have the usual metadata, and **the namespace is going to be the same namespace as the Service and the Pod**.

Then, in the specification we will define the rules. The first one is the host name. Then the http forwarding to the Internal service, with the backend, with the name and the port, these being already defined to work with the Service and the Pod.

Then we always have to create a forwarding adderss if something occurs. But we will first create the Ingress. For that we know that we have to use **kubectl apply -f dashboard-ingress-yaml** .

And then we can see the ingress just with **kubectl get ingress -n kubernetes-dashboard** .

We will see that the address will be empty because it takes a little bit of time to assign the address to the Ingress.

Then, we have to modify the file:

```
sudo vim /etc/hosts
```

And then add the IP address below the other lines, for instance:

```
192.168.64.5    dashboard.com
```

The in our local machine we can just go to the browser and write the address, and we will be redirected to that IP address.

8.4 Ingress Default Backend

If we write the following line:

```
kubectl describe ingress dashborad—ingress —n kubernetes—dashboard
```

In the line **Default Backend** we can see how we handle the request by default.
So, if we do not have this Service defined in our cluster then Kubernetes will try to forward it to the Service and will not find it, and we would get some default error response.

So, a good usage is to define a custom error message when a page is not found or when a request we can not handle comes in.

So, all we have to do is to create an Internal Service with the same name and port number, and also create a Pod or application that sends that custom error message response.

9 Helm - Package Manager of K8s

Helm is used as a Package Manager for Kubernetes, it is a convenient way for packaging collections of Kubernetes YAML files and distribute them in public and private repositories.

Let's say we have deployed our application in K8s cluster and we want to deploy ElasticStack additionally in our cluster so that our application uses it to collect its logs.

In order to deploy ElasticStack in our K8s cluster we would need a StatefulSet (for stateful applications like databases), we would need a ConfigMap with external configuration, a Secret, the K8s user with its respective permissions and we would also have to create a couple of Services.

If we were to create all of these files manually by searching for each one of them separately on internet it would bw a tedious job.

And this process would have to be repeated across multiple clusters, so they created all these files once and packaged them up and made it available so that other people could use them.

This bundle of YAML files is called **Helm Charts**.

We can try to find a Helm Chart which might be useful for us using the command line: **helm search ¡keyword¿** or in HelmHub.

9.1 Templating Engine

Let's image that we have an application that is made up of multiple microservices and we are deploying all of them in our K8s cluster.

The Deployment and Service of each of those microservices are pretty much the same with the only difference that the application name and version are different. Or the Docker image name and version tags are different.

Without Helm we would write separate YAML configuration files for each of these microservices, so we would have multiple deployment Service files where each one has its own application name and version defined.

But, since the only difference between those YAML files are just a couple of lines or values, **using Helm we can define a common Blueprint for all the microservices, and the values that are going to change can be replaced by placeholdres**. And that would be a templete file.

And, **then we can use additional YAML files, *values.yaml* to fill in these values for each microservice**.

9.2 Same Applications across different environments

We will usually have development, staging and production environments, so, instead of creating different files for each one of these, we can package them up to make our own application chart that will have all the necessary files that the particular application needs and then we can use them to redeploy the same application in a different K8s cluster environment, using only one command.

9.3 Helm Chart Structure

The directory structure would be the following:

```
mychart/
    Chart.yaml
    values.yaml
    charts/
    tremplates/
    ...
```

Chart.yaml is a file that contains all the meta information about the chart, such as the name, version, list of dependencies, ...

As we know **values.yaml** has all the values to fill the template.

The **charts** directory will have chart dependencies inside.

The **templates folder** is where the template files are stored.

So, when we execture the Helm install command to deploy those yaml file into K8s the template files will be filled with the values from **values.yaml** producing valid K8s manifest that can then be deployed into K8s.

In this folder we can also have other files like README of License file.

10 Kubernetes Volumens

We will see 3 different components:

- Persistent Volume
- Persintent Volume Claim
- Storage Class

Let's consider a case where we have a *mySQL* database which our application uses.
Data gets added and updated in the database. And we maybe create a new database with a new user and etc. , by default, when we restart the database Pod all those changes will be gone. Because **Kubernetes does not give us data persistence out of the box. It is something that we have to explicitly configure for each application that needs saving data between Pod restarts.**
So basically we need a storage that does not depend on the Pod life cycle, so it wil still be there when the Pod dies and a new one gets created, so the new Pod can pick up where the previous one left off. So it will read the existing data from that storage to get up-to-date data.

However, **we do not know in which node the new Pod restarts, so our storage must also be available on all nodes, not just on a specific one**.

So when the new Pod tries to read the existing data the up-to-date data is there on any node in the cluster.

Also **we need a highly available storage that will survive even if the whole cluster crushes**.

Apart from using Volumes when working with databases, we may have to use them when we have session files for instance.

10.1 Persistent Volume

We have to think of the Persistent Volume as a cluster resource like a RAM or a CPU that is used to store data.

The Persistent Volume gets created like the other Kubernetes components: using a YAML file where we can specify the **kind** (PersistentVolume) and in the specification section we have to define different parameters like how much storage should be created for the Volume.

But since the Persistent Volume is an abstract component it must take the storage from the actual physical storage, like the local hard drive, cloud-storage, ...

But we have to configure it with the outside storage, and K8s does not care about it, so what the Persistent Volume does is to give as an interface to the actual storage that we have to take care of, and we have to manage it ourselves.

So, by creating Persistent Volumes we can use actual physical storages, we have to define which storage backend we want to use to create that storage abstaction in the specification part.

We can see in the Kubernetes documentation which storages it supports.
And, we have to remember that **Persistent Volumes are NOT namespaced, meaning that they are accessible to the whole cluster**, unlike Pods, Services, ...

10.1.1 Local vs Remote Volume Types

The local volume types violate the 2nd and 3rd requirements for data persistence for databases:
Not being tied to 1 specific node but rather to each node equally, because we do not know where the new Pod will start.
Surviving in cluster crush scenarios.

This is why for database persistance we should almost always use remote storage.

10.1.2 K8s Administrator and K8s User

Persistent Volumes are resources that need to be there BEFORE, when the Pod that depends on it is created.

So, there are two main roles in K8s:

- **Administrator**: The one who sets up the cluster and maintains it, and also makes sure that the cluster has enough resources.
- **User**: The one who deploys the applications in the cluster, either directly or though a CI pipeline.

So the administrator is the one to configure the actual storage and to create Persistent Volume Components from these storage backends.

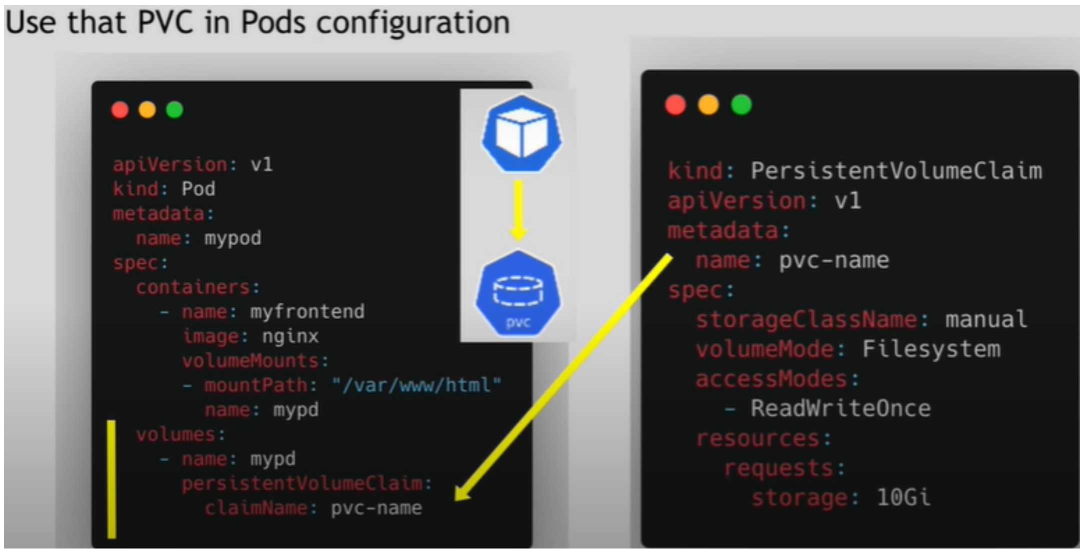
10.2 Persistent Volume Claim

Developers have to configure the application YAML file to use those Persistent Volume components, in other words, their application has to **CLAIM the Volume storage** and to do that we use **Persistent Volume Claims (PVC)**.

PVCs are also created with YAML configurations. The PVC claims a Volume with a certain storage size capacity which is define in the Persistent Volume Claim and needs some characteristics, such as the access type (RealOnly, ReadWriteOnce, ...).

And so, whatever Persistent Volume which satisfies the introduced criterial will be used for the application.

And we have to use the claim in our Pods configuration:



Use that PVC in Pods configuration

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: pvc-name
```

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: pvc-name
spec:
  storageClassName: manual
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Figure 10:

So, to use those levels of abstraction, the Pods request the Volume through the Persistent Volume Claim.

Then the Claim will go and try to find a Persistent Volume in the cluster which satisfies the claim and then the Volume will have storage backend that will create that storage resource from.

Claims must exist in the same namespace as the Pod using the claim, while Persistent Volumens are not namespaced.

10.3 Storage Class

Storage Class provisions persistent Volumes dynamically.
Whenever a PVC claims it the Storage Class creates or provisionates Volumes in a cluster automatically.

The Storage Class gets created using a YAML configuration file

11 Kubernetes StatefulSet

A **StatefulSet** is a K8s component that is used specifically for stateful applications.

An example of stateful applications are all databases, any application that stores data to keep track of its state. These are applications that track state by saving that information in some storage.

Stateless applications do not require information about previous states, and each interaction is independent.

Let's image a simple setup of a *node.js* application that is connected to MongoDB database.

When a request comes in to the node.js application it does not depend in any previous data to handle this incoming request. It can handle it based on the payload, in the request itself.

Now, a typical request like this will additionally need to update some data in the database or query the data. That is where *mongodb* comes in. So when *node.js* forwards that request to *mongodb*, *mongodb* will update the data based on its previous state, or query the data from its storage.

Because of this difference in stateful and stateless aplications, they are both deployed in different ways using different components in K8s.

Stateless applications are deployed using Deployment components, where the Deployment is an abstraction of Pods and allows us to replicate that application in a cluster.

While stateless aplications are deployed using Deployment, stateful applications in K8s are deployed using **StatefulSet components**.

Just like Deployments, the StatefulSet makes it possible to replicate the stateful app parts, running multiple replicas of it.

And we can also configure storage with both of them equally in the same way.

11.1 Deployment vs StatefulSet

Replicating stateful applications is more difficult and has a couple of requirements that stateless applications do not have.

Let's say that we have one *mySQL* database Pod that handles requests from a Java application, which is deployed uisng a Deployment component, and let's say that we scale the Java application to three parts so they can handle more client requests.

In parallel we want to scale the mySQL app so it can handle more Java requests as well.

Scaling the Java application is straightforward, their Pods will be identical and interchangeable, so we can scale it using the Deployment easily.

The deployment wil create the Pods in any order, they will get one Service that will load balance to any one of the replica Pods for any request, and also when we delete them they will get deleted in a random order or in the same time. And also when we scale them down one random replica will be chosen to be deleted. So we do not have any complications here.

On the other hand, the *mySQL* Pod replicas can not be created and deleted at the same time or in any random order and they can not be randomly addressed. And the reason is that **the replica Pods are not identical**, they each have their own additional identity on top of the common blueprint of the Pod they get created from.

11.2 Scaling Database Applications

When we start with a single *mySQL* Pod, it will be used for both reading and writing data, but when we add a second one it cannot act the same way, because if we allow two independent instances of *mySQL* to change the same data, we will end up with data inconsistency.

So instead there is a mechanism which decides that only one Pod is allowed to write or change the data which is shared, being read at the same time by multiple *mySQL* instances from the same data, but **we will just allow one Pod to update the data, it will be kwown as MASTER, and the rest will be WORKERS**.

On the other hand, these Pods do not have access to the same physical storage, even though they use the same data they are not using the same physical storage of the data. They each have their own replicas of the storage, and this means that each replica at any time myst have the same data as the other ones and in orde rto achieve that they **must continuously synchronize their data**, and since the Master is the only one allowed to change data and the slaves need to take care of their own data storage, the workers must know about each such change so they can update their data storage to be up to date for the next query requests.

There is a mechanism to allow this continuous data synchronization: **The Master changes data and all slaves update their own data storage to keep in sync to make sure that each Pod has the same state**.

If we add another worker node it will first clone the **previous** worker node data and then will start the continuous synchronization.

And the same when we downscale, it will start by the last one.

This means that we can have a temporary storage for a stateful application ad not persist the data at all, since the data gets replicated between the Pods. So theoretically it is possible to just rely on data replication between the Pod. But this would also mean that the whole data would be lost when all the Pods die.

And therefore it is still the best practice to use data persistence for stateful applications, because losing the data would be unacceptable.

So this means that we have to work with Volumes.

12 Kubernetes Services

In a K8s cluster each Pod gets its own internal IP address, but the Pods in K8s are ephemeral, meaning that they come and go very frequently, and when the Pod restarts or when the old one dies and the new one gets started in its place it gets a new IP address.

So it does not make sense to use Pod IP addresses because we would have to adjust it every time the Pod gets recreated.

With the **Service** we have a solution of a stable or static IP address that does not change even if the Pod dies. So it represents a persistent stable IP address.

A Service also provides a load balancing because when we have Pod replicas, the Service will get each request targeted to our application or database and then forward it to one of the Pods that we have.

So the clients can call a single stable IP address instead of calling each Pod individually.

12.1 ClusterIP Services

This is the default type of Service.

Let's image that we have a Pod with our microservice app deployed, and it also contains a side-car container to collect the logs of the microservice and send them to some destination database. (These 2 containers in the Pod probably will be running on different pots).

The Pod will also get an IP address from a range that is assigned to a node. If we have for example 3 working nodes in a cluster, each worker node will get a range of IP addresses which are internal in the cluster. SO the first one will get IP addresses from a range of 10.2.1.x onwards, the second one 10.2.2.x and the third one 10.2.3.x.

We will be able to check these values using the following command:

```
kubectl get pod —o wide
```

If we use more than one replicas, the containers inside will use obviously the same ports, but the Pod's own IP address will be different, as we have commented above.

One the request gets handed over to the Service at its address, then the Service will know to forward that request to one of those Pods that are registered as the Service endpoints.

12.2 Service Communication: Selector

The Service must know which Pods to forward the request to and which Port to forward it to.

The **Selector** will identify its member Pods (its endpoint Pods) using.

So, in the Service's configuration file, in the YAML file, we have to specify the **selector** attribute, and sinse we have to use key-value pairs defines as a list.

These key-value pairs are labels that Pods should have to match that Selector.

So, in the Pod configuration file, we assign the Pods certain **labels** in the **metadata** section. And these labels can be arbitrarily named.

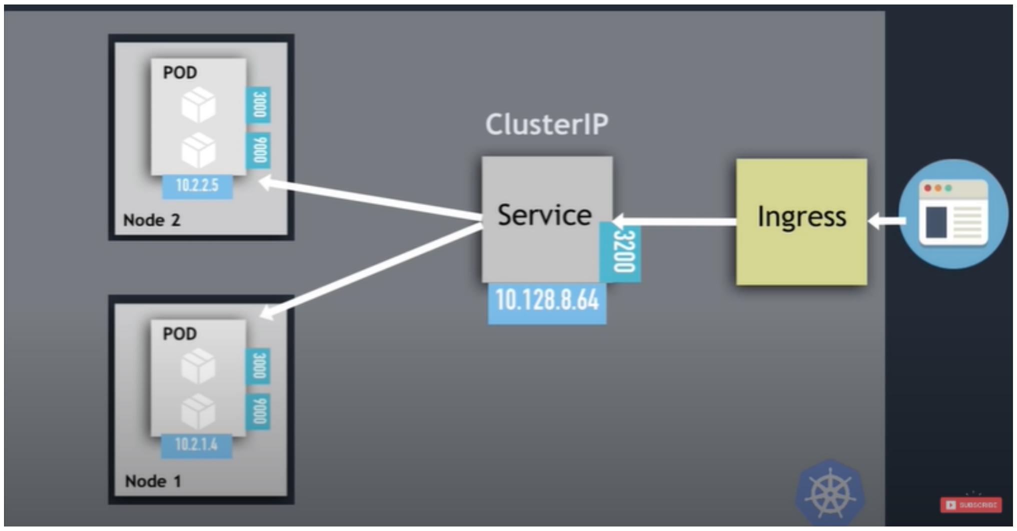
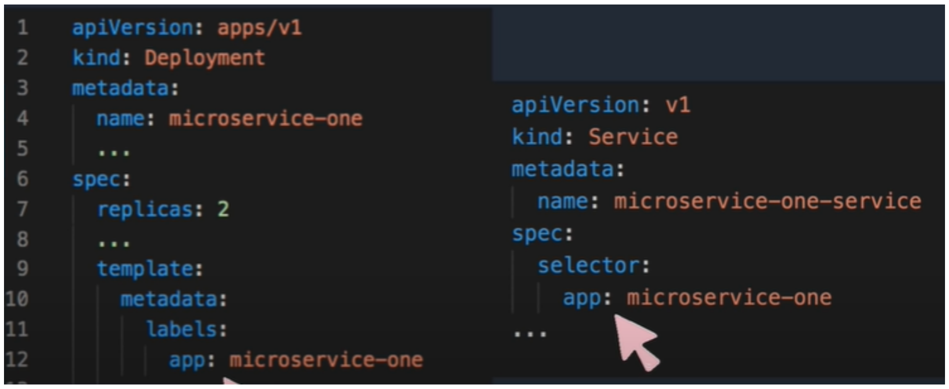


Figure 11:



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: microservice-one
  ...
spec:
  replicas: 2
  ...
  template:
    metadata:
      labels:
        app: microservice-one
```

```
apiVersion: v1
kind: Service
metadata:
  name: microservice-one-service
spec:
  selector:
    app: microservice-one
...
```

Figure 12:

This means that if we have a Deployement component that creates replicas of Pods, we have to choose the labels key-values and in the Service, in the **selector** attribute, match those key-value pairs.

And it should match ALL the selectors, not just one. This is how the Service will know which Pods belong to it.

On the other hand, if a Pod has multiple Pods open for different applications, the Service must know to which ports to forward the request. And this is defined in the **targetPort** attribute.

12.3 Headless Services

Each request to the Service is forwarded to one of the Pod replicas that are registered as Service endpoints.

But, if a client wants to communicate with one of the Pods directly and selectively. **This is necessary when we are deploying stateful applications in K8s**, like databases.

As we know, in such applications the Pod replicas are not identical, each one has its individual state and characteristic, since we will have a Master Pod and Worker/Slave Pods. And the Worker nodes will have different states.

For a client to connect all Pods individually it needs to figure out the IP address of each individual Pod. We could make an API call to the K8s API service and it will return its list of Pods and their IP addresses. But this type of communication is too tight to the K8s specific API and also is inneficient since we would have to get the whole list of Pods and their addresses every time we want to connect to one of the Pods.

But, as an alternative solution, **K8s allows clients to discover Pod IP addresses through DNS lookups**. When a client performs a DNS lookup for a Service, the DNS server returns a single IP address which belongs to the Service. And this is the Service's cluster IP address.
But, if we set the IP address of the Service to None, then when creating the Service the DNS server will return the Pods' IP addresses instead of the Service's IP address. And so now the client can do a simple DNS lookup to get the IP address of the Pods that are members of that Service, and the client can use that IP address to connect to the specific talk part it wants to, or all the Pods.

So, **the way we define a Headless Service in a service configuration file setting the clusterIP attribute to None.**

12.4 3 Service type attributes

When we define a Service configuration we can specify a type of Service, which can have 3 different values:

- **ClusterIP**: The default. Creates a Service that is accessible on a static port on each worker node in the cluster
- **NodePort**: This type of Service makes the external traffic accessible on a static port on each worker node. So, instead of from the Ingress, the browser request will come directly to the worker node at the port that the Service specification defines. And the port that NodePort service type exposes is defined in the **nodePort** attribute.

  Note that this values is predefine to be in the range 30000-32767. Anything outside of it will not be accessible.
- **LoadBalancer**: The Service becomes accessible externally through a cloud provider's load balancer functionality. So each cloud provider has its own native load balancer implementation. And that is created and used whenever we create a load balancer service type

So, the LoadBalance Service is an extension of the NodePort service, which at the same time is an extension of the ClusterIP Service

In a real K8s setup example we would probably not use NodePort for external connections, we would maybe use it to test some Service very quickly, but not for production use cases.