

CKA

Alejandro Campos

January, 2023

Contents

1 K8s Core Concepts	10
1.1 High Level Cluster Architecture	10
1.1.1 What is K8s Cluster?	10
1.1.2 Worker Nodes vs Master Node	10
1.1.3 Cluster etcd	10
1.1.4 Kube-scheduler	10
1.1.5 Controller Manager	10
1.1.6 Kube-apiserver	11
1.1.7 Todo es containerizable	11
1.1.8 Kubelet	11
1.1.9 Worker Nodes vs Master Node 2	12
1.1.10 High Level Cluster Architecture Picture	12
1.2 Etcd Cluster	12
1.2.1 What is key-value store?	13
1.2.2 Etcd in Local	14
1.2.3 Etcd in K8s	15
1.3 Kube API Server	17
1.3.1 Kube-apiserver in K8s	18
1.4 Kube Controller Manager	19
1.4.1 Node-Controller	20
1.4.2 Replication-Controller	20
1.4.3 Kube-Controller-Manager in K8s	20
1.5 Kube Scheduler	21
1.5.1 How does kube-scheduler work?	21
1.5.2 Kube-Scheduler in K8s	22
1.6 Kubelet	23
1.6.1 kubelet in K8s	23
1.7 Kube Proxy	24
1.7.1 kube-proxy in K8s	25
2 Core Resources	26
2.1 Pods	26
2.1.1 Pod, a single instance of an app	26
2.1.2 Multi-Container Pod	27
2.1.3 Helping containerization life cycle	27
2.1.4 How to easily deploy a basic Pod	29

2.1.5	How to define a Pod using YAML files?	29
2.1.6	ContainerPort, how the works exposure works with Pods	33
2.1.7	How to deploy a Pod from a YAML file using kubectl?	33
2.1.8	Creating a Pod from YAML from scratch	36
2.2	Images Used by Pods	36
2.2.1	Introduction to Container Registries	36
2.2.2	How can I specify registry for Docker?	37
2.2.3	How can I specify registry for K8s?	38
2.2.4	How is that I don't specify anyone and I can pull images?	39
2.2.5	Docker commands to interact with Registries	39
2.2.6	Pushing a local image to a public or private container registry	40
2.3	ReplicaSet	41
2.3.1	What is replica and why we need a Replication-Controller?	41
2.3.2	How to define a ReplicaSet?	42
2.3.3	How to create a ReplicaSet using kubectl?	43
2.3.4	What is the deal with labels and selectors?	43
2.3.5	How to scale up and down a ReplicaSet?	43
2.3.6	Tips in Creating and manipulating ReplicaSet	45
2.4	Deployments / StatefulSet	45
2.4.1	Introduction to Deployments	45
2.4.2	How to define a Deployment	45
2.4.3	How to create a Deployment using kubectl?	46
2.4.4	What is a StatefulSet?	46
2.4.5	How to create a StatefulSet using kubectl?	46
2.5	Services	46
2.5.1	Introduction to Services	47
2.5.2	NodePort Services	48
2.5.3	How to define a NodePort Service?	49
2.5.4	How to create a Service using kubectl?	50
2.5.5	One Service, multiple instance of the same Pod	50
2.5.6	Service ClusterIP	51
2.5.7	How to define a ClusterIP Service?	52
2.5.8	Service LoadBalancer	53
2.5.9	How to define a NodeBalancer Service?	54
2.5.10	K8s DNS: How to reach the services from other Pods?	54
2.5.11	CurlPod: How to reach the services from the Nodes?	55
2.5.12	Port Forward: How to reach the services from outside the cluster	55
2.6	Namespaces	57

2.6.1	Introduction to Namespaces	57
2.6.2	K8s automatically created Namespaces	58
2.6.3	Create own Namespaces	59
2.6.4	kubectl commands to manage Namespaces	59
2.6.5	Defining Namespace in K8s Objects	60
2.6.6	Namespace Policies	60
2.7	Imperative vs Declarative, apply command	61
2.7.1	Introduction	61
2.7.2	Working Imperative	62
2.7.3	Working Declarative	63
2.7.4	kubectl apply	64
3	Labels, Selector & Annotations	65
3.1	Introduction	65
3.2	Kubectl commnads with Labels	66
3.3	Annotations	66
4	Pod Scheduling	67
4.1	Manual Scheduling	67
4.2	Taints & Tolerations	68
4.2.1	Introduction	68
4.2.2	kubectl commands	69
4.2.3	Taint in Master Nodes	70
4.3	Node Selector	70
4.3.1	Introduction to Node Selectors	70
4.3.2	Pod Configuration	71
4.3.3	Node Configuration	71
4.4	Affinity and anti-Affinity	72
4.4.1	Introduction to Node Affinity	72
4.4.2	Node Affinity Types	74
4.4.3	Inter-pod affinity and anti-affinity	75
4.5	Resource Requirements & Limits	77
4.5.1	Introduction	77
4.5.2	Resources types	78
4.5.3	Requests & Limits	79
4.5.4	Ideal Scenario for CPU	80
4.5.5	Default Configuration	81
4.5.6	Changing Default Configuration, LimitRange Object	81
4.5.7	Restrict the total amount of resources	82

4.6	DaemonSets	82
4.6.1	Introduction to DaemonSets	82
4.6.2	Defition of DaemonSets	83
4.6.3	Creation and view of DaemonSets	83
4.6.4	How dos DaemonSets work?	84
4.6.5	Example of DaemonSet - Fluentd	84
4.7	Static Pods	86
4.8	Multiple Schedulers	87
4.8.1	Introduction	87
4.8.2	Defining schedulers	87
4.8.3	Creating new schedulers	87
4.8.4	Configuring Pods to consume a custom scheduler	88
4.8.5	Monitor the correct functioning of a custom scheduler	88
4.9	Scheduler Profiles	88
5	Logging and Monitoring	90
5.1	Monitoring Cluster Components	90
5.1.1	Basic K8s Metrics Server	90
5.1.2	kubectl top	91
5.2	Application Logs	91
5.2.1	Logs in docker	91
5.2.2	Logs in K8s	91
5.2.3	Logs in K8s	91
5.3	Logs with K8s-Ready EFK	91
5.3.1	Kubernetes Logging Structure	91
5.3.2	Necessity of another solution	92
5.3.3	EFK Solution	93
5.3.4	Fluentd (Logs Collector)	93
5.3.5	ElasticSearch + Kibana (Logging Endpoint)	94
5.3.6	EFK K8s Resources to Deploy	94
5.3.7	EFK: Kibana	98
6	Application Lifecycle Management	99
6.1	Rolling Updates & Rollback	99
6.2	Deployment Strategies	99
6.2.1	Recreate Strategy	99
6.2.2	Rolling Update	99
6.3	How to update a Deployment?	100
6.4	Deployment updates under the hoods	101

6.5	Rollback	101
6.6	Commands & Arguments in Pod definition	101
6.6.1	Introduction - Docker CMD & ENTRYPOINT	101
6.6.2	Commands & Arguments in a K8s Pod	102
6.7	Env Vars in Pods	103
6.7.1	Env Vars in Docker Containers	103
6.7.2	Env Vars in Pods Containers with key-values directly	103
6.7.3	Env Vars in Containers using ConfigMaps	103
6.7.4	Env Vars in Pods Containers using Secrets	103
6.8	ConfigMaps	104
6.8.1	ConfigMaps Creation	104
6.8.2	ConfigMaps Pod Injection	105
6.8.3	Pods update when ConfigMaps changes?	106
6.9	Secrets	106
6.9.1	Secret Creation	107
6.9.2	Secrets Pod Injection	107
6.9.3	View Secrets	109
6.9.4	Secrets Notes	110
6.10	Multi Container Pods	110
6.11	Init Containers	111
6.12	Self Healing Applications	112
7	Cluster Maintenance	112
7.1	Upgrades	112
7.1.1	What happens when a Node goes down?	112
7.1.2	Node drain	112
7.2	Cluster Upgrade Process	113
7.2.1	K8s Releases	113
7.2.2	Cluster Upgrade Process	113
7.3	Backup and Restore Methods	117
7.3.1	Resources	117
7.3.2	ETCD	117
8	Security	118
8.1	Security in K8s Introduction	118
8.2	Authentication and Authorization, who can access the cluster?	118
8.2.1	Static File Authentication	119
8.3	TLS Introduction (not for K8s)	119
8.3.1	Why do we need certificates?	119

8.4	Introduction to TLS on K8s	122
8.5	Certificate Creation for K8s	123
8.5.1	Generating CA Certificate	123
8.5.2	Generating Client Certificates	123
8.5.3	How to generate and configure all the K8s Certificates?	127
8.5.4	Certificate Details	127
8.6	Certificates API	129
8.6.1	Need of the API?	129
8.6.2	How does Certificate API works?	129
8.7	Kubeconfig	130
8.8	Kubeconfig File	130
8.9	kubectl config command	132
8.10	kubectx	132
8.11	API Groups	133
8.11.1	/api	134
8.11.2	/api	134
8.12	Authorization	135
8.12.1	Node Authorization	135
8.12.2	ABAC	135
8.12.3	RBAC	136
8.13	Webhook	136
8.13.1	Authorization Mode	136
8.13.2	RBAC	137
8.14	ServiceAccounts and Tokens	141
8.14.1	Introduction	141
8.14.2	Creation and Usage	141
8.14.3	Namespace ServiceAccount	143
8.14.4	Security inside Docker containers	143
8.15	Network Policies	144
8.15.1	Just ingress	145
8.15.2	Ingress and Egress	147
9	Storage	149
9.1	Docker Storage	149
9.1.1	Introduction	149
9.1.2	Where does Docker stores its persistent data?	149
9.1.3	Docker Image Layer Architecture	149
9.1.4	Volumes in Docker	151
9.1.5	Storage Drivers & Volume Drivers	153

9.2	K8s CSI (Container Storage Interface)	153
9.3	K8s Volumes & Persistent Volumes	154
9.3.1	Docker Volumes	154
9.3.2	K8s Volumes	155
9.3.3	K8s Persistent Volumes	156
9.4	K8s Persistent Volume Claims	157
9.4.1	Creating a Persistent Volume Claim	158
9.4.2	Deleting Persistent Volume Claims	159
9.5	Storage Class	160
9.5.1	Different Storage Classes	162
10	Networking Basis in Linux (not K8s)	163
10.1	Switching and routing in Linux	163
10.1.1	Switching	163
10.1.2	Routing	164
10.1.3	Private and Public Networks configuration	165
10.1.4	Using a Linux System as a router	166
10.2	DNS in Linux	167
10.2.1	Introduction to DNS	167
10.2.2	DNS server	167
10.2.3	FQDN	168
10.2.4	Creating own FQDN	170
10.2.5	Record Types (Tipos de Registros)	170
10.2.6	Nslookup or dig	170
10.3	Network Namespaces in Linux	171
10.3.1	Introduction to Namespaces in Linux	171
10.3.2	Namespaces for Networking	171
11	Networking in K8s	174
11.1	Docker Networking	174
11.2	CNI (Container Networking Interface)	174
11.3	Networking Cluster Nodes	174
11.3.1	Useful Commands	176
11.4	POD Networking Concepts	177
11.5	CNI in K8s	177
11.6	WeaveWorks Solution (based on CNI)	177
11.7	IPAM (CNI)	178
11.8	Service Networking	178
11.8.1	Introduction	178

11.8.2 How services works, kube-proxy	178
11.8.3 How kube-proxy set that rules?	178
11.8.4 Example of iptable rule application	179
11.9 DNS on K8s	179
11.9.1 How K8s Implements DNS in the Cluster	180
11.9.2 How Pods point to the CoreDNS Server?	181
11.10 Ingress	181
11.10.1 The need of Ingress	181
11.10.2 Introduction to Ingress	184
11.10.3 Ingress Controller & Ingress Resources	184
11.10.4 Ingress Controller	184
11.10.5 Ingress Resources	187
11.10.6 Lab X - Playing with Ingress 1	192
11.10.7 Lab X - Playing with Ingress 2	197
12 CheatSheet	199
12.1 K8s Best Practices	199
12.2 Get	199
12.3 Create Objets	199
12.3.1 Imperative	199
12.3.2 Declarative	201
12.4 Delete	201
12.5 Policies	201

1 K8s Core Concepts

1.1 High Level Cluster Architecture

1.1.1 What is K8s Cluster?

El objetivo de K8s es alojar aplicaciones en forma de contenedores; para que puedan desplegar fácilmente tantos contenedores de la aplicación como sean necesarios y permitir la comunicación entre los diferentes servicios de la aplicación.

Un Cluster de K8s consiste en un conjunto de nodos, que pueden ser físicos o virtuales, que pueden estar desplegados On Premises o en Cloud y que despliegan y gestionan aplicaciones en forma de contenedores.

1.1.2 Worker Nodes vs Master Node

Imaginemos un Cluster de K8s como una empresa de barcos, que tiene que albergar contenedores (físicos) en el mar. Para poder hacer eso, necesitamos barcos de carga que puedan albergar los contenedores, y eso en K8s serían los **Worker Nodes**. Pero estos barcos tienen que estar gestionados por algún cerebro, en nuestra analogía será el puerto.

El puerto se encarga de cargar los contenedores en los barcos, planificar como cargarlos para hacerlo eficientemente (sin desperdiciar espacio en los barcos), identificar barcos adecuados para cada tipo de contenedor, albergar información sobre los barcos, controlar y seguir los contenedores en los barcos, etc. Nuestro puerto en K8s es el **Master Node**.

El **Master Node** se encarga de gestionar todo el Cluster de K8s almacenando la información relativa a los diferentes nodos, planificando que contenedores van a parar a cada uno de ellos, monitorizando los nodos y los contenedores que en ellos corren, etc. Todo esto lo hace mediante una serie de componentes llamados **Control Plane**.

1.1.3 Cluster etcd

Hay muchísimos contenedores que cargar y descargan en muchísimos barcos, por ello es necesario tener almacenado en algún sitio un registro de: en qué barcos se cargan en qué contenedores, a qué hora, estado de los barcos, estado de los contenedores, etc. Para ello, K8s utiliza un almacén de pares clave-valor de alta disponibilidad **Cluster etcd**.

Cluster Etcd es una DB que almacena información en formato pares clave-valor.

1.1.4 Kube-scheduler

Dentro del puerto, tiene que haber un sistema de grúas que se encargue de gestionar que contenedores se cargan en qué barcos, determinar el barco adecuado en función del tamaño del contenedor, capacidad del barco, contenedores que ya lleva cargados, entre otros factores (como el destino del barco, el tipo de contenedores que puede llevar, etc.). De esto, se encarga un elemento dentro del **Master Node** llamado **kube-scheduler**.

Kube-scheduler identifica los nodos correctos para alojar cada uno de los contenedores, en función de los recursos necesarios para desplegar el contenedor, la capacidad disponible del nodo, cualquier política, restricción, regla de afinidad entre nodos, etc.

1.1.5 Controller Manager

También, en nuestro puerto, necesitamos un elemento que se encargue del control del estado de los barcos, de las rutas que siguen, los daños que sufren, su estado, si se destruyen... Y poder, en función de este

conocimiento, tener otros barcos disponibles donde reubicar los contenedores. Además, este nodo también se encargará de gestionar la comunicación entre los diferentes barcos.

De estas tareas, se encarga el **Controller Manager**, compuesto por dos controladores más específicos.

- **Node controller:** se encarga de añadir nuevos nodos en el cluster, gestionar los nodos que ya tiene y gestionar también situaciones de indisponibilidad de nodos, para poder reubicar contenedores.
- **Replication Controller:** se encarga de controlar las replicas de contenedores desplegadas en los nodos, garantizando siempre que el numero de replicas deseado se encuentre desplegado entre los diferentes nodos.

1.1.6 Kube-apiserver

Todo esto está genial, ¿pero como se comunican entre ellos los diferentes elementos de nuestro puerto? ¿Como le dice el **Replication Controller** al **kube-scheduler** que necesita cargar un contenedor X que se ha quedado sin barco? ¿Cómo a su vez el **kube-scheduler** le responde diciendo te lo he metido en este nodo y lo anota en el **Cluster etcd**? Pues todo eso se hace mediante **Kube-apiserver**.

Kube-apiserver es el responsable de la orquestación de todas las operaciones dentro del Cluster y además, exponer la **API de K8s** necesaria para que usuarios externos al Cluster pueden realizar sus operaciones sobre él, así como supervisar el estado del Cluster y realizar las modificaciones que se requieran.

1.1.7 Todo es containerizable

Aprovechando las geniales características que nos ofrecen los contenedores, **todos los elementos del Master Node son container-ready**, es decir, pueden desplegarse como servicios containerizados. La solución de red DNS, también puede levantarse como un servicio containerizado. Para ello, necesitamos un software que funcione como motor de construcción y ejecución de esos contenedores, que suele ser **Docker**, pero K8s también puede utilizar Rocket, o ContainerD. Por tanto, necesitamos **Docker** (o alternativa) instalado en cada uno de los Nodos del Cluster.s

1.1.8 Kubelet

Siguiendo con nuestro ejemplo de empresa naval, cada uno de nuestros barcos tiene un capitán, responsable de la gestión de actividades en los barcos y el que mantiene la conexión con el puerto. Este capitán envía información constante al puerto, indicando si pueden o no albergar nuevos contenedores, el estado en el que se encuentran sus contenedores, la capacidad actual a la que se encuentra el barco... Este capitán en K8s es el **kubelet**.

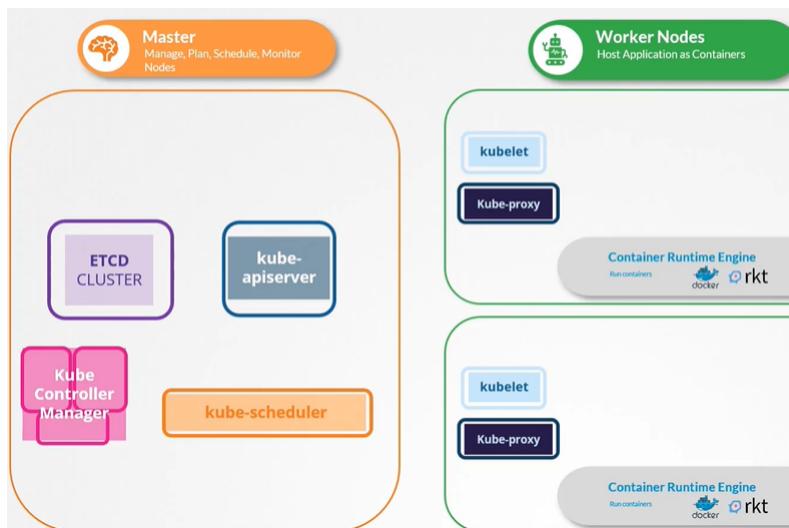
Kubelet es un agente que se ejecuta en cada **Worker** del Cluster, se encarga de recibir las instrucciones del **kube-apiserver** y desplegar o destruir contenedores en su barco según el **kube-apiserver** se lo indique. Además se encarga de reportar el estado de su nodo y de los contenedores que se están ejecutando en el nodo.

Pero estos **kubelets**, deben poder comunicarse también entre ellos, pues en un nodo puede estar ejecutándose en un contenedor el Front-End de una página web y en otro nodo alejado en otro contenedor el Back-end de la misma, y en otro la DB. ¿Como se comunican los diferentes elementos de una aplicación desplegada en diferentes nodos? Esto se hace mediante el elemento **kube-proxy**

El servicio **kube-proxy** se ejecuta también en cada **Worker** del Cluster, y se asegura de que existan las reglas a nivel de red necesarias para que los diferentes contenedores de los diferentes workers puedan comunicarse entre si.

1.1.9 Worker Nodes vs Master Node 2

Master Node	Worker Nodes
Cluster etcd Kube-scheduler Node controller Replication Controller Kube-apiserver	kubelets kube-proxy



1.1.10 High Level Cluster Architecture Picture

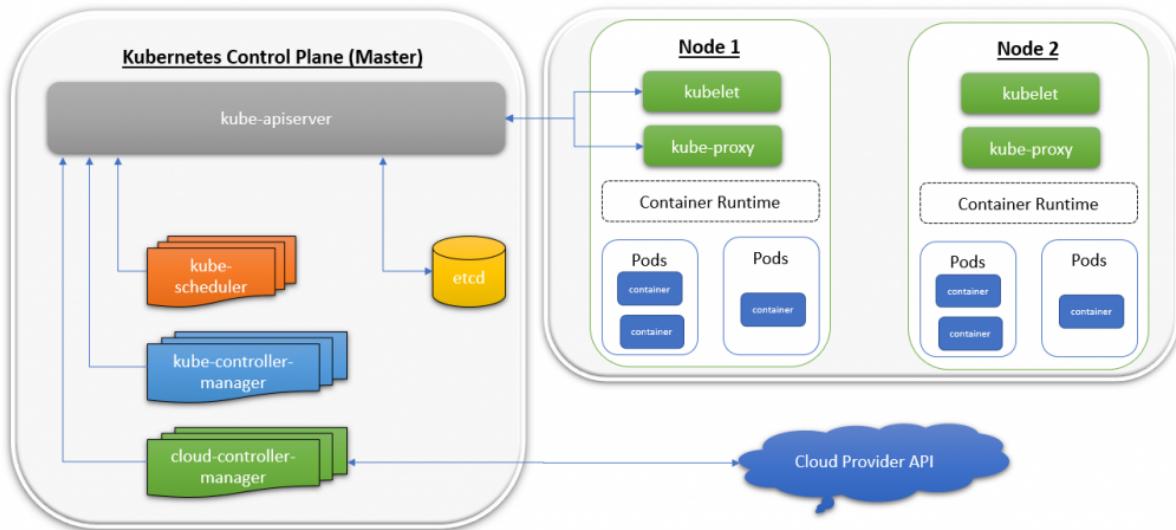


Figure 1: High Level K8s architecture

1.2 Etcd Cluster

Official K8s Doc

Etcd es una DB que almacena los datos en pares clave-valor y es simple, rápido y seguro.

1.2.1 What is key-value store?

1.2.1.1 DBS tabulares

Normalmente, las DBs han tenido un formato de tabla; como mysql o base de datos relacionales, que almacenan los datos en forma de filas y columnas.

name	age
Alex	25
Claudia	24
Mario	17

Aquí tenemos una imagen de una estructura de base de datos en tabla. Imaginemos que ahora queremos añadir otro campo, como por ejemplo salario. En las DBs en tabla, este campo debe añadirse para todas las entradas, pues se añadirá como columna y todos las filas la tendrán.

name	age	salary
Alex	25	45k
Claudia	24	31k
Mario	17	

Sin embargo, no todos estan trabajando, de forma que la celda de salario de la fila de Mario vacía. Y lo mismo iria pasando con otros campos que vayamos añadiendo

name	age	salary	cinturón
Alex	25	45k	
Claudia	24	31k	
Mario	17		verde

1.2.1.2 DBS pares clave-valor

Estas DBs almacenan la información en forma de ficheros o páginas. Así cada entrada tiene asociada un fichero con toda la información acerca de esa entrada. Estos ficheros pueden tener cualquier formato y estructura, aunque suelen utilizarse formatos como **JSON** o **YANO**. Los cambios en estos ficheros no afectan a los demás, Claudia y Alex pueden tener información sobre su salario en su fichero y no sobre el cinturón, mientras que Mario a la inversa.

Se pueden añadir detalles adicionales a cada uno de estos ficheros sin tener que actualizar los campos de los demás documentos.

```
alex.json
{
  "name": "Alex",
  "age": 25,
  "salary": 45k
}
```

```
claudia.json
```

```
{  
    "name": "Claudia",  
    "age": 24,  
    "salary": 31k  
}
```

```
mario.json
```

```
{  
    "name": "Mario",  
    "age": 17,  
    "cinturon": "verde"  
}
```

1.2.2 Etcd in Local

1.2.2.1 Install etcd:

We will follow: [Github etcd](#)

```
etcd_download.sh
```

```
ETCD_VER=v3.4.26
```

```
# choose either URL  
GOOGLE_URL=https://storage.googleapis.com/etcd  
GITHUB_URL=https://github.com/etcd-io/etcd/releases/download  
DOWNLOAD_URL=${GOOGLE_URL}  
  
rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz  
rm -rf /tmp/etcd-download-test && mkdir -p /tmp/etcd-download-test  
  
curl -L ${DOWNLOAD_URL}/${ETCD_VER}/etcd-${ETCD_VER}-linux-amd64.tar.gz \  
-o /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz  
tar xzvf /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz \  
-C /tmp/etcd-download-test --strip-components=1  
rm -f /tmp/etcd-${ETCD_VER}-linux-amd64.tar.gz
```

```
$ source etcd_download.sh
```

```
$ /tmp/etcd-download-test/etcd --version
```

```
$ /tmp/etcd-download-test/etcdctl version
```

Run etcd Service:

```
$ /tmp/etcd-download-test/etcd
```

Etcd service:

Cuando se ejecuta **etcd**, se arranca un servicio que escucha en el **puerto 2379 por defecto**. A continuación puede añadir cualquier cliente al servicio **etcd** para almacenar y recuperar información.

1.2.2.2 Etc controller client (etcdctl)

Por defecto, etcd service trae incorporado el cliente **etcdctl**, un cliente que funciona por linea de comandos y que se utiliza para almacenar y recuperar pares clave-valor.

Store key-value pairs: creando una entrada en la DB con la información introducida.

```
$ /tmp/etcd-download-test/etcdctl set key1 value1
```

Retrieve key-value pairs:

```
$ /tmp/etcd-download-test/etcdctl get key1
```

View more options:

```
$ /tmp/etcd-download-test/etcdctl
```

1.2.3 Etcd in K8s

Es la DB que almacena todos los datos relativos al cluster, sobre: nodos, PODs, configs, secrets, accounts, roles, etc. Toda la información que nos muestra el comando `kube control get` proviene del servidor etcd. Cada cambio que se produzca en el Cluster: incorporación de nuevos nodos, despliegues de Pods o ReplicaSets, etc. Se actualiza en el servidor etcd, y solo cuando se actualiza en este, se puede considerar que el cambio ha sido completado con éxito.

Dependiendo de como se configure el Cluster, etcd se despliega de forma diferente. A lo largo de esta sección veremos 2 tipos de despliegue de K8s: uno desplegado desde cero y otro utilizando la herramienta Qadium.

1.2.3.1 Installing etcd-cluster

Si has arrancado tu Cluster de K8s utilizando la herramienta `kubeadm`, tu **etcd-cluster** se generará automáticamente ya configurado. Pero si estás configurando el hardware del Cluster desde cero, entonces el **etcd-cluster** está disponible como binario en la página de K8s release.

- Descargar binario

```
$ wget -q --https-only \
"https://github.com/correos/etcd/releases/download/v3.3.9/etcd-v3.3.9-linux-amd64.tar.gz"
```

- Descomprimir

```
$ tar -xzvf archive.tar.gz -C /path/to/directory/to/extract
```

- Una vez descargado puede arrancarse como servicio

```
$ etcd.service
```

1.2.3.2 Configuring etcd-cluster

Hay muchas variables que parametrizan el **etcd.service** por ahora es importante que nos quedemos con las siguientes:

- dirección donde escucha **etcd**

```
--advertise-client-urls https://<IP>:2379
```

- Parametros para garantizar la seguridad de las comunicaciones con certificados **SSL/ TLS**

```
--cert-file=/etc/etcd/kubernetes.pem  
--key-file=/etc/etcd/kubernetes-key.pem  
--peer-cert-file=/etc/etcd/kubernetes.pem  
--peer-key-file=/etc/etcd/kubernetes-key.pem  
...
```

1.2.3.3 View etcd-cluster

Podemos controlar el cluster con el comando **kubeadm**, y en cuanto al **etcd**, **kubeadm** puede desplegar el servicio **etcd** como un Pod en el namespace del **Master Node == kube-system namespace**.

```
$ kubectl get pods -n kube-system  
==> etcd-master
```

Para listar todas las claves almacenadas por K8s:

```
$ kubectl exec etcd-master -n kube-system etcdctl get / --prefix -keys-only
```

En entornos de alta disponibilidad tendremos varios **master Nodes** dentro del cluster, teniendo también diversas instancias de **etcd** repartidas entre los **master Nodes**. Para que los diferentes **etcd** se reconozcan entre ellas debemos configurar en los **etcd.service** y configurar las diferentes instancias (por url) de los etcd services:

```
-- initial-cluster
```

Apart from that, you must also specify path to certificate files so that ETCDCTL can authenticate to the ETCD API Server. The certificate files are available in the etcd-master at the following path.

```
--cacert /etc/kubernetes/pki/etcd/ca.crt  
--cert /etc/kubernetes/pki/etcd/server.crt  
--key /etc/kubernetes/pki/etcd/server.key
```

So to modify that in one command (ETCDCTL API version and path to certificate files):

```
$ kubectl exec etcd-master -n kube-system -- sh -c "ETCDCTL_API=3 etcdctl \  
get / --prefix --keys-only --limit=10 \  
--cacert /etc/kubernetes/pki/etcd/ca.crt \  
--cert /etc/kubernetes/pki/etcd/server.crt \  
--key /etc/kubernetes/pki/etcd/server.key"
```

1.2.3.4 Etcd commands

API Version 2

```
etcdctl backup  
etcdctl cluster-health  
etcdctl mk  
etcdctl mkdir  
etcdctl set
```

API Version 2

```
etcdctl snapshot save  
etcdctl endpoint health  
etcdctl get  
etcdctl put
```

To set the right version of ETCDCTL API set the environment variable ETCDCTL_API command, so when API version is not set, it is assumed to be set to version 2:

```
$ export ETCDCTL_API=3
```

1.3 Kube API Server

Official K8s Doc

Principal componente de gestión de K8s. Cuando se ejecuta un comando **kubectl**, este llega al **kube-apiserver**, y este primero autentifica la solicitud y la valida. A continuación recupera la información del **etcd-cluster** y responde con la información.

Pongamos el ejemplo de la creación de un Pod:

1. **kubectl** create pod.yaml
2. **kube-apiserver** crea un Pod sin asignarlo a ningun nodo, consolida esta información en el **etcd-cluster** y reporta al usuario que ya lo ha creado
3. **kube-scheduler** monitoriza constantemente el **kube-apiserver** y se da cuenta de que se ha creado un nuevo Pod sin nodo asignado.
4. **kube-scheduler** busca el nodo adecuado para hostear el Pod y se lo comunica al **kube-apiserver**
5. **kube-apiserver** actualiza la información en el **etcd-cluster** y contacta con el **kubelet** del **worker** correspondiente.
6. El **kubelet** del **worker** seleccionado crea el Pod y contacta con el **container runtime** para que despliegue en el la imagen de la aplicación.
7. Una vez hecho, el **kubelet** actualiza el estado del Pod al **kube-apiserver** y este actualiza los datos en el **etcd-cluster**

Para todas las peticiones kubectl sobre el Cluster se sigue un patrón similar, el **kube-apiserver** gestiona todas las diferentes tareas que hay que realizar para llevar a cabo un cambio en el Cluster, como se puede ver en la Figura 1.

Note

El **kube-apiserver** es el responsable de autenticar y validar todas las solicitudes por **kubectl** y actualizar los datos en el **etcd-cluster**. De echo **kube-apiserver** es el único componente que puede interactuar directamente con el **etcd-cluster**. Los otros elementos como el **kube-scheduler**, el **kube-controller-manager** o los **kubelet** utilizan el **kube-apiserver** como intermediario para actualizar los datos en el **etcd-cluster**. Es una forma de evitar la inconsistencia de datos gestionando la concurrencia desde el **kube-apiserver**.

1.3.1 Kube-apiserver in K8s

1.3.1.1 Installing kube-apiserver

Si has arrancado tu Cluster de K8s utilizando la herramienta **kubeadm**, tu **kube-apiserver** se generara automaticamente ya configurado. Pero si estás configurando el hardware del Cluster desde cero, entonces el **kube-apiserver** está disponible como binario en la página de K8s release.

- Descargar binario

```
$ wget https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-apiserver
```

- Descomprimir

```
$ tar -xzvf archive.tar.gz -C /path/to/directory/to/extract
```

- Una vez descargado puede arrancarse como servicio

```
$ kube-apiserver.service
```

Se puede descargar y configurar para que se ejecute como un servicio en el **Master Node** de tu Cluster. El **kube-apiserver** se configura con muchos parametros, esto se explica ya que un Cluster de k8s son un monton de componentes diferentes que trabajan juntos, comunicandose entre ellos de muchas maneras diferentes, así que todos deben saber donde se encuentran los demás componentes. Además, existen diferentes modos de autenticación, autorización, cifrado, seguridad, etc. Es por eso que cada componente del cluster requiere tantos parametros de configuración.

1.3.1.2 Parametros importantes

Note

Como todos los componentes se comunican entre ellos, todos ellos tendran configurable una sección de parametros para garantizar la seguridad de las comunicaciones, con certificados **SSL/ TLS**

- Parametros para garantizar la seguridad de las comunicaciones entre los diferentes componentes, con certificados **SSL/ TLS**

```
--etcd-cafile=/var/lib/kubernetes/ca.pem  
--etcd-certfile=/var/lib/kubernetes/kubernetes.pem  
--etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem
```

- Cadena de conexión contra **etcd-servers**

```
--etcd-servers=https://127.0.0.1:2379
```

1.3.1.3 View kube-apiserver

Con kubeadm

- Si lo hemos levantado con `kubeadm` tool el **kube-apiserver** estará desplegado como un Pod dentro del namespace del **Master Node == kube-system namespace**:

```
$ kubectl get pods -n kube-system
==> kube-apiserver-master
```

- También lo podremos ver accediendo a su yaml dentro del directorio `/etc/kubernetes/manifests/`:

```
$ cat /etc/kubernetes/manifests/kube-apiserver.yaml
```

Sin kubeadm

- Si no hemos utilizado `kubeadm` para levantar el Cluster de K8s, podemos ver las opciones accediendo a la definición del servicio en el sistema `/etc/systemd/system/kube-apiserver.service`:

```
$ cat /etc/systemd/system/kube-apiserver.service
```

- También encontrarlo en la lista de tareas en ejecución de tu sistema si tienes el cluster levantado:

```
$ ps -aux | grep kube-apiserver
```

1.4 Kube Controller Manager

Official K8s Doc

El **kube-controller-manager** gestiona diferentes controladores en K8s. Un controlador en K8s es como una oficina o departamento dentro del puerto que tiene su propio conjunto de responsabilidades, las cuales deben estar constantemente pendientes del estado de los **worker** y tomando las acciones necesarias para remediar la situación.

En definición K8s, un controlador es un proceso que supervisa constantemente el estado de varios componentes en el sistema y trabaja para llevar todo el sistema al estado de funcionamiento deseado.

Los diferentes controladores que gestiona el **Kube Controller Manager** son:

- Node-Controller
- Replication-Controller
- Namespace-Controller
- Deployment-Controller
- Job-Controller
- PV-Protection-Controller
- Service-Account-Controller
- ...

1.4.1 Node-Controller

El **Node-controller** se encarga de supervisar el estado de los nodos y tomar las acciones necesarias para mantener los Pods corriendo. Esto lo hace comunicandose con el **kube-apiserver**, preguntando por el estado de todos nodos cada 5 segundos. Si un nodo deja de responder durante 40 segundos este se marca como **unreachable** por el **node-controller**.

Una vez un nodo ha sido marcado como **unreachable**, le da 5 minutos para recuperarse, si no lo hace, elimina los Pods asignados a ese worker y los aprovisiona en otros nodos saludables.

1.4.2 Replication-Controller

Se encarga de supervisar el estado de los diferentes **ReplicaSets**, comprobando en cada momento que haya desplegados en el cluster el numero de Pods disponibles indicados por el ReplicaSet. Si un Pod muere, crea otro.

1.4.3 Kube-Controller-Manager in K8s

¿Como podemos ver todos estos controladores y donde estan ubicados en el Cluster? Todos ellos estan empaquetados bajo un solo proceso, **Kube-Controller-Manager**. De forma que si nos descargamos e instalamos **Kube-Controller-Manager**, estaremos descargando e instalando con el todos los controladores necesarios para el correcto funcionamiento de K8s.

1.4.3.1 Install Kube-Controller-Manager

Si has arrancado tu Cluster de K8s utilizando la herramienta `kubeadm`, tu **kube-controller-manager** se generara automaticamente ya configurado. Pero si estás configurando el hardware del Cluster desde cero, entonces el **kube-controller-manager** está disponible como binario en la página de K8s release.

- Descargar binario

```
$ wget https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-controller-man
```

- Descomprimir

```
$ tar -xzvf archive.tar.gz -C /path/to/directory/to/extract
```

- Una vez descargado puede arrancarse como servicio

```
$ kube-controller-manager.service
```

1.4.3.2 Configure Kube-Controller-Manager

Como con todos los elementos, tiene una larga lista de parametros configurables. Por aquí dejo algunos relevantes (no voy a nombrar más los de certificados, que este, como todos los elemntos, tambien necesita, sección 1.3.1.2).

```
--node-monitor-period=5s  
--node-monitor-grace-period=40s  
--pod-eviction-timeout=5m0s  
--controllers
```

En la sección de controllers pueden especificarse los controladores que queremos tener habilitados, por defecto lo están todos.

En caso de que alguno de los controladores no parezca funcionar o no exista, este sería un buen punto de partida donde mirar.

1.4.3.3 View Kube-Controller-Manager

Con kubeadm

- Como siempre, si levantamos nuestro Cluster con `kubeadm`, el **kube-controller-manager** se levanta como un Pod en el namespace del **Master Node == kube-system namespace**:

```
$ kubectl get pods -n kube-system  
=> kube-controller-manager-master
```

- También puedes acceder a su manifest dentro de la carpeta que `kubeadm` utiliza para desplegar estos Pods:

```
cat /etc/kubernetes/manifests/kube-controller-manager.yaml
```

Sin kubeadm

- Como siempre también, puedes inspeccionar las opciones viendo el servicio en si ejecutándose por el sistema:

```
cat /etc/systemd/system/kube-controller-manager.service
```

- O filtrando su nombre en todos los servicios de ejecución del sistema:

```
$ ps -aux | grep kube-controller-manager
```

1.5 Kube Scheduler

Official K8s Doc

K8s Scheduler es el encargado de programar los Pods en los nodos, de decidir que Pod va en cada nodo. Pero no crea los pods en los nodos, ya que esto es tarea del kubelet. El **kube-scheduler** simplemente decide en qué worker debe crearse cada pod.

1.5.1 How does kube-scheduler work?

Cuando hay muchos nodos y muchos Pods diferentes, hay que asegurarse de que el Pod en concreto puede levantarse en otro nodo concreto, dependiendo de algunas características específicas del Pod, pero sobretodo basándose en los requisitos de recursos del Pod y capacidad disponible del nodo.

Para ello, el **kube-scheduler** crea 2 etapas de filtros:

1. Descarta aquellos Nodos en los cuales el Pod no puede desplegar, por características definidas de ambos que veremos más adelante.
2. Descarta aquellos nodos con menos capacidad disponible de los recursos que requiere el Pod para desplegar (required).

3. Dentro de los nodos con suficiente capacidad como para levantar el Pod busca aquel con mayor capacidad sobrante una vez desplegado el Pod con sus requirements de recursos. Aquel nodo al que le sobre más capacidad es el que finalmente levanta el Pod.

1.5.2 Kube-Scheduler in K8s

1.5.2.1 Install Kube-Scheduler

Si has arrancado tu Cluster de K8s utilizando la herramienta `kubeadm`, tu **kube-scheduler** se generara automaticamente ya configurado. Pero si estás configurando el hardware del Cluster desde cero, entonces el **kube-scheduler** está disponible como binario en la página de K8s release.

- Descargar binario

```
$ wget https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-scheduler
```

- Descomprimir

```
$ tar -xzvf archive.tar.gz -C /path/to/directory/to/extract
```

- Una vez descargado puede arrancarse como servicio

```
$ kube-scheduler.service
```

1.5.2.2 Kube-Scheduler options

kubeadm

- Como siempre, si levantamos nuestro Cluster con `kubeadm`, el **kube-scheduler** se levanta como un Pod en el namespace del **Master Node == kube-system namespace**:

```
$ kubectl get pods -n kube-system  
==> kube-scheduler-master
```

- También puedes acceder a su manifest dentro de la carpeta que `kubeadm` utiliza para desplegar estos pods:

```
cat /etc/kubernetes/manifests/kube-scheduler.yaml
```

kubeadm

- Como siempre también, puedes inspeccionar las opciones viendo el servicio en si ejecutandose por el sistema:

```
cat /etc/systemd/system/kube-scheduler.service
```

- O filtrando su nombre en todos los servicios de ejecución del sistema:

```
$ ps -aux | grep kube-scheduler
```

1.6 Kubelet

[Official K8s Doc](#)

Es el capitán del barco (Worker Node), es quien carga los contenedores en los barcos (está mamadísimo) según las instrucciones del **kube-scheduler**, está comunicándose constantemente con el **kube-apiserver** y con el resto de **kubelets** enviando informes constantes sobre el estado del barco y los contenedores que este alberga, además es quien registra el nodo en un cluster de K8s.

Cuando le llega una orden directa del **kube-scheduler** (pasando siempre por el **kube-apiserver**) de levantar un pod, el **kubelet** es el encargado de contactar con el motor de containerización para que extraiga la imagen necesaria y ejecute una instancia dentro de un pod.

El **kubelet** constantemente monitoriza el estado de los pods y los contenedores que este contiene, enviando informes constantes al **kube-apiserver**

1.6.1 kubelet in K8s

1.6.1.1 Install kubelet

El kubelet está disponible como binario en la página de K8s release.

Si has arrancado tu Cluster de K8s utilizando la herramienta **kubeadm**, tu **kubelet** se generará automáticamente ya configurado. Pero si estás configurando el hardware del Cluster desde cero, entonces el **kubelet** está disponible como binario en la página de K8s release.

- Descargar binario

```
$ wget https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kubelet
```

- Descomprimir

```
$ tar -xzvf archive.tar.gz -C /path/to/directory/to/extract
```

- Una vez descargado puede arrancarse como servicio

```
$ kubelet.service
```

Warning

Si hemos levantado nuestro cluster de K8s con **kubeadm**, este NO levanta automáticamente kubelets.
SIEMPRE debes instalar el kubelet en tus Worker Nodes

Una vez descargado podemos correrlo como un servicio:

```
$ kubelet.service
```

1.6.1.2 kubelet options

- Puedes inspeccionar las opciones viendo el servicio en si ejecutándose por el sistema:

```
cat /etc/systemd/system/kubelet.service
```

- O filtrando su nombre en todos los servicios de ejecución del sistema:

```
$ ps -aux | grep kubelet
```

1.7 Kube Proxy

Official K8s Doc

Dentro de un cluster de K8s, todos los pods pueden conectarse con todos los demás pods. Esto es posible gracias al despliegue de un red de pods en el cluster. Esta red, es una **red virtual interna**, que se extiende por todos los nodos del cluster para tener todos los pods conectados. Gracias a esta red, todos los nodos son capaces de comunicarse entre ellos.

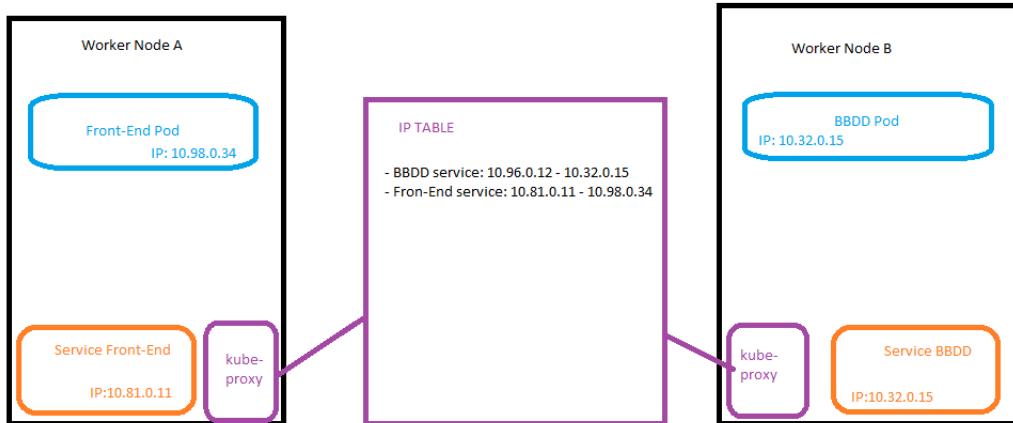
Un ejemplo de necesidad de esta red es una aplicación web sencilla que funciona con un front-end y una DB. El Pod que corre el front-end está desplegado en un Nodo A, en una punta del cluster, mientras que el Pod que corre la DB está desplegado en un Nodo B, en la otra punta del cluster. ¿Como se comunican entre ellos? ¿Cuando el front-end necesita información de la DB para mostrar?

Si no tuvieramos ni idea de K8s, podríamos decir que asociando a nuestro front-end la IP del Pod de la DB, pero esto sería un error. Porque no podemos asegurar que el IP del Pod de DB sea siempre la misma, ya que en caso de reinicio la IP del Pod cambiará.

La mejor forma de que el Pod de front-end se comunique con la DB es a través de los **Services**. Los **Servicios** en K8s nos exponen las aplicaciones mediante el nombre del servicio o su IP y las hacen accesibles dentro del Cluster, desde cualquier lugar o parte dentro del mismo.

Así pues, el Front-end podrá llegar a la DB utilizando el nombre del **Service**. El **Service** también tiene una IP asignada. Pero entonces... ¿Puede el servicio desplegarse como un Pod y añadirse a esta red interna? La respuesta es NO, pues el **Service** no es una cosa real, no ejecuta ningún contenedor como los Pods, no tiene un proceso de escucha activa, es un elemento virtual que solo vive en la memoria de K8s.

Estos **Services** deben ser accesibles desde cualquier nodo del Cluster, ¿cómo conseguimos eso? Aquí es donde entra el **Kube-proxy**, un proceso que se ejecuta en cada nodo del Cluster de K8s, su trabajo es buscar nuevos servicios, y cada vez que encuentra un nuevo servicio, crea las reglas necesarias en cada **Worker Node** para poder reenviar tráfico de los nuevos servicios a los Pods. Una de las formas de hacerlo es mediante **IPtables rules**. En este caso, **Kube-proxy** crea una regla **iptable** en cada nodo del cluster para reenviar el tráfico desde la IP del service hacia la ip del pod.



1.7.1 kube-proxy in K8s

1.7.1.1 Install kube-proxy

Si has arrancado tu Cluster de K8s utilizando la herramienta `kubeadm`, tu **kube-proxy** se generara automaticamente ya configurado. Pero si estás configurando el hardware del Cluster desde cero, entonces el **kube-proxy** está disponible como binario en la página de K8s release.

- Download binary

```
$ wget https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-proxy
```

- Unpackage it

```
$ tar -xzvf archive.tar.gz -C /path/to/directory/to/extract
```

- Once downloaded and unpackaged it can be run as a service

```
$ kube-proxy.service
```

1.7.1.2 kube-proxy options

With kubeadm

Note

If K8s Cluster is setup with `kubeadm`, **kube-proxy** it deploys as **DaemonSet**. That means that a **kube-proxy** Pod will be deployed on every Node of the K8s Cluster. Without `kubeadm` it should be done it by hand.

- If K8s Cluster is setup with `kubeadm`, **kube-proxy** runs as Pod in every Node of K8s Cluster as **DaemonSet**

```
$ kubectl get daemonset -n kube-system  
==> kube-proxy
```

- También puedes acceder a su manifest dentro de la carpeta que `kubeadm` utiliza para desplegar estos pods:

```
cat /etc/kubernetes/manifests/kube-proxy.yaml
```

Sin kubeadm

- Como siempre también, puedes inspeccionar las opciones viendo el servicio en si ejecutandose por el sistema:

```
cat /etc/systemd/system/kube-proxy.service
```

- O filtrando su nombre en todos los servicios de ejecución del sistema:

```
$ ps -aux | grep kube-proxy
```

2 Core Resources

2.1 Pods

[Official K8s Doc](#)

Note

In order to start explaining pods, we assume the following:

- The application is already developed and built into Docker image.
- The image is available on a Docker repository (for example Docker Hub), so K8s can pull it down.
- Also assume the K8s Cluster is already setup, configured and correctly working. It could be configured as a single-node setup (as in your PC local cluster) or a multi-node setup (like ICP).

2.1.1 Pod, a single instance of an app

As we discuss before, with K8s our ultimate aim is to deploy our application in the form of containers on a set of machines that are configured as worker Nodes inside the K8s Cluster. However **K8s does not deploy containers directly on the worker Nodes, the container are encapsulated into a K8s Object known as Pods.**

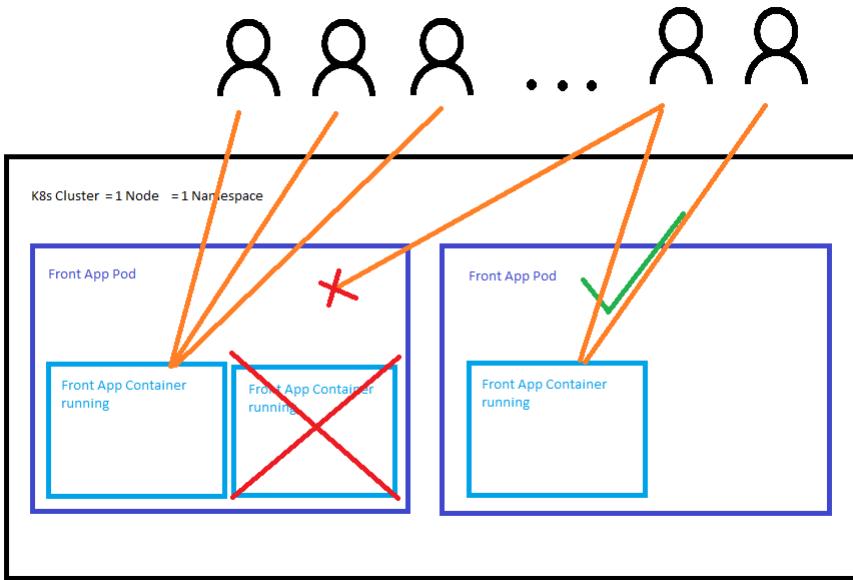
A Pod is a single instance of an application, a Pod is the smallest object you can create in K8s. Let's see an example:

EXAMPLE:

Imagine that you have the simplest K8s cluster, with only one Node, and one Pod running an application container.

1. What happens if the number of users starts to increase?

- **Bad answer:** start another container inside the Pod
- **Good answer:** create a new Pod with a new instance of the same application



What happens if the Node has not enough capacity?

It can be created new pods with a new instance of the same application on another Nodes inside the Cluster. So we should add new Nodes to expand the cluster's physical capacity.

Note

What is important to remark, is that is a best practice to have a one-to-one relationship between containers and running applications. To scale up, you create new pods, not new containers in the Pods. And to scale down, just remove those pods.

2.1.2 Multi-Container Pod

We said it is a best practise to have one-to-one relationship between Pods and application. But sometimes, the applications need to run two or more containers in the same Pod, a helper container which might be doing some kind of supporting task for our application (for example to run a healthcheck solution to expose the state of the app). And you want these helper containers to run alongside application main process (the main container). For that cases, multi-container Pods can be created, because Pods can run multiple containers.

But you must know, that the scalation will be for the two containers, you cannot scale up the main application without scaling up the side-car container too. And if the main container has some problem and it dies, both container dies, since they are part of the same pod.

Note

The two containers can also communicate with each other directly by referring to each other as **localhost**, since they share the same network space. Plus, they can easily share the same storage space as well.

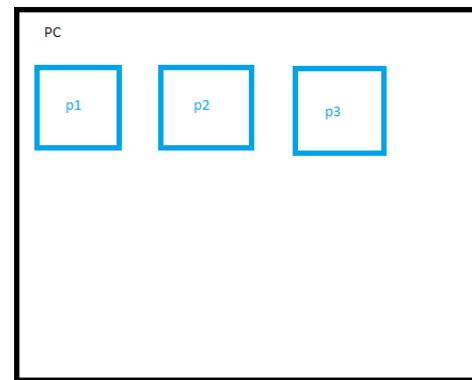
2.1.3 Helping containerization life cycle

Imagine you don't have K8s, you just have docker, and you want to containerize and run your application. You should use the docker run command to do it.

```
$ docker run python-app
```

If the load increases we deploy more instances of our application:

```
docker run --name p1 python-app  
docker run --name p2 python-app  
docker run --name p3 python-app
```

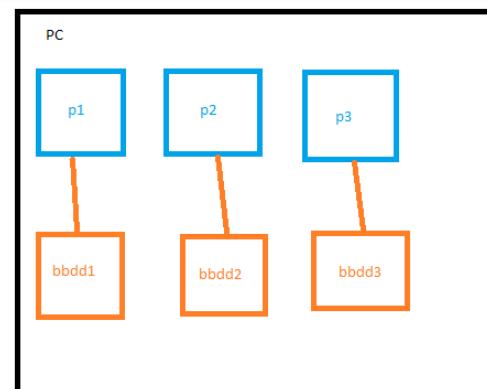


If our architecture grows and gets complex, for example our application uses a DB. We now have to run a new DB container linked to our web application. They must have a one-to-one relationship.

```
docker run --name p1 python-app  
docker run --name p2 python-app  
docker run --name p3 python-app
```



```
docker run --name bbdd1 mysql -link p1  
docker run --name bbdd2 mysql -link p2  
docker run --name bbdd3 mysql -link p3
```



To maintain this infrastructure in docker, it is needed to establish the Network Connectivity and configuration to enable these communications, we would need to establish persistent storage to share it among the containers, we would need to maintain a communication map and storage map. And the most important, we would need to monitor the state of the application container and when it dies, kill the helper container as well as it is no longer required. And when a new container of the app is deployed, a new container of DB should be deployed as well.

Note

Maybe the DB example is not quite good, because it is better to run DB on another Pod, but imagine it is another service less detachable, for example healthcheck, or a service to do something for the main application.

With Pods, K8s makes all these tasks for us automatically, we just need to define what containers a Pod consists off and the containers in a Pod by default will have access to the same Network and Storage. Also they will be created together and destroyed together.

Even the application is simple and it doesn't need to run more than one container K8s still requires to create pods. However multi-container Pods are rare, so we will focus in single containers per Pod in this course.

2.1.4 How to easily deploy a basic Pod

```
$ kubectl run nginx-pod --image nginx
```

It deploys a Docker container by creating a Pod in the current namespace. So in this case, it creates at first a Pod automatically (called nginx-pod) and then it deploys an instance of the NGINX Docker image. But where does it get the application image from? If nothing is specified it downloads it from the Docker Hub repository. You could configure K8s to pull the image from another public Docker repository or a private repository.

```
$ kubectl get po
```

It is used to see pods running in the namespace, you can check K8s DO180 pdf document where it is very good explained.

But remember that we haven't made the web server accessible to external users, you only can access internally (from the same Node / PC). This will be explained in Networking and Services lectures.

2.1.5 How to define a Pod using YAML files?

K8s uses YAML files as inputs for the creation of objects, such as: Pods, ReplicaSets, Deployments, StatefulSets, PV's, Secrets, Services, Ingress, etc. All of them follow a similar structure.

Note

A K8s object definition ALWAYS must have four top level fields:

- apiVersion
- kind
- metadata
- spec

2.1.5.1 apiVersion

The version of the K8s API you are using to create the objects. Depending on what we are trying to create we must use the right API version. For now, as we are working on pods, we will set the apiVersion as **v1**. Another possible values are:

- **v1**: for Pods and Services.
- **apps/V1beta**: for ReplicaSet and Deployment.
- **extensions/V1beta**

2.1.5.2 kind

Refers to the type of object we will create with the template.

- Pod
- Service

- Ingress
- Deployment
- Statefulset

2.1.5.3 metadata

It is data about the object, like its name, labels, etc. Metadata only accepts name, labels or anything else that K8s expects to be under metadata. You cannot add any other property as you wish under this.

Name

It is a string, and it is the name the Pos will receive when creating.

Labels

Labels is a dictionary within the metadata dictionary, it can have any key-value pairs as desired. In our example we just define one (name: myapp), but similarly you could add other labels as you see, which it will help you identify these objects at a later point in time.

Say for example there are hundreds of pods running on a front-end application and hundreds of pods running a backend application or DB. It will be difficult for you to group these pods one they are deployed. If you label them now as front-end, backend or database including a label in the labels dictionary it will be able to filter the pods based on this label at a later point in time.

2.1.5.4 spec (specifications)

We already haven't specified the image to run into the pod, neither the version, init-containers, envVars, volumeMounts, ports, liveness and readiness probes. This is what we should do in spec section. It will be different (very different) depending on the object, so it's important to refer to the documentation section to get the right format for each.

Since we are only creating a Pod, with a single container in it, it is easy.

NOTE

To refer different indentations, parent / children elements, we will use point nomenclature.

EXAMPLES:

- **.spec:** refer to the first group, spec section.
- **..containers:** refer to the second group, all containers sections.
- **...env:** refer to the third group, all env sections.
- **..containers[0].env:** refer to elem containers[0] env section

To get specific data from a YAML file:

```
$ kubectl get <object_type> <object_name> \
-o jsonpath='{..containers[0]{.name}{"\t"}{.image}{"\n"}' \
|sort|column -t
```

In a range:

```
$ kubectl get RESOURCE_TYPE RESOURCE_NAME \
-o jsonpath='{range ..containers[*]}{.name}{"\t"}{.image}{"\n"}' \
|sort|column -t
```

WARNING

Spec it is a dictionary, but it can contain different arrays or list inside it.

- ..initContainers
- ..containers
- ...env
- ...volumeMounts
- ...ports
- ..volumes

containers it is an array or list, because the pods can run multiple containers within them. The same happens with env, env inside a container or initContainer is a list. It is the reason why appears "-", it marks an element in the list, every element in a list is a dictionary.

EXAMPLE

common-template.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
      ports:
        - containerPort: 8080
      env:
        - name: MAIN_PATH
          value: "/apps"
        - name: http_proxy
          value: "None"
        - name: USER
          value: "Default"
        - name: ENVIRONMENT
          value: "tst"
        ...
      
```

```
pod-template.yaml

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  initContainers:
  - image: certificates...
    name: cas-....
    resources:
      requests:
        cpu: xxx
        memory: yyy
      limits:
        cpu: zzz
        memory: www
  containers:
  - name: nginx-container
    image: nginx
    ports:
    - containerPort: 8080
    env:
      - name: MAIN_PATH
        value: "/apps"
      - name: http_proxy
        value: "None"
      - name: USER
        value: "Default"
      - name: ENVIRONMENT
        value: "tst"
      ...
    resources:
      requests:
        cpu: xxx
        memory: yyy
      limits:
        cpu: zzz
        memory: www
    volumeMounts:
      ...
  readinessProbe:
    ...
  livenessProbe:
    ...
  ports:
    ...
volumes:
- ...
```

NOTE

The lists are ALWAYS ordered alphabetically, this is because when K8s process the YAML file and creates the object, it will be ordered this way. Actually you can order the lists as you want, but once processed in the K8s Object, they will be ordered alphabetically.

Note

The number of spaces at the beginning of the lines doesn't matter. But children always should have more spaces than their parents and they should be the same at the same label, as they are siblings. If it is not respected, our YAML will be wrong.

2.1.6 ContainerPort, how the works exposure works with Pods

The `containerPort` field is declarative, meaning it tells Kubernetes which port the container inside the pod is expected to be listening on. However, it does not configure the container or application itself to listen on that port.

The `containerPort` field in a Kubernetes pod definition serves primarily as metadata for describing the network configuration of a container. It helps document which port the application inside the container is listening on, but it does not actually configure or expose the port.

What is happening behind the scenes is that Pod network is attached to the image network, so the Pod will be exposing the same ports than the image is running the services.

2.1.7 How to deploy a Pod from a YAML file using kubectl?

```
$ kubectl apply -f pod-template.yaml [-f pod-template.json]
```

```
$ kubectl create -f pod-template.yaml [-f pod-template.json]
```

2.1.7.1 How to check Pod if there are services running

Check the "ephemeral" Pod IP, remember that is not a good practice to access applications directly from Pod IP, but it is just a test:

```
$ kubectl get po -o wide
```

Then curl the ip and the port:

```
$ curl <pod_ip>:<container_port>
```

2.1.7.2 kubectl basic commands

What is resource?

<resource> can be replaced by the resource kind we want to see. Object examples and their shortcuts:

- **pod:** po
- **deployment:** deploy
- **replicaset:** rs
- **statefulset:** sts
- **job:** job
- **service:** svc
- **ingress:** ingress
- **configmap:** cm
- **secret:** secret
- **persistent-volume:** pv
- **persistent-volume-claim:** pvc
- **virtualservice:** virtualservice
- **gateway:** gw

NOTE

To know which alias can be used for each K8s resource, use the following command

```
$ kubectl create <object> --help  
# EX: kubectl create deployment --help
```

- To see all resources in the current namespace

```
$ kubectl get all
```

- To see <resource> in loop in the current namespace

```
$ kubectl get <resource> -w
```

- To see <resource> with one specific value from a label

```
$ kubectl get <resource> -l <label_key>:<label_value>
```

```
$ kubectl get <resource> -l name=mysql -l app=php-quote
```

- See <resource> from specific namespace

```
$ kubectl get <resource> -n <specific_namespace>
```

- See <resource> from all-namespaces

```
$ kubectl get <resource> --all-namespaces [-A]
```

- See <resource_name> <resource> YAML [or JSON] file

```
$ kubectl get <resource> <resource_name> -o yaml [-o json]
```

- Describe <resource> definition and current state

```
$ kubectl describe <resource> <resource_name>
```

- See <resource_name> <resource> filtered by JSON

```
$ kc get po -o jsonpath='{..containers[0].image}'
```

```
$ kubectl get <resource> <resource_name> \
-o jsonpath='{range ..containers[*]}{.name}{"\t"}{.ports}{"\n"}' \
| sort|column -t
```

- Get Networking IP's: Obtener los recursos de un tipo con información sobre sus tablas de enrutamiento:

```
$ kubectl get <resource> -o wide
```

- Describe <resource> definition and current state filtering for values

```
$ kubectl describe <resource> <resource_name> | grep -i "STATE" -A 5 -B 5
```

JQ

JQ is a linux binary that prints in a beautiful way jsons. To use it:

```
$ kubectl get po -o jsonpath='{..containers[0]}' | jq '..'
```

2.1.8 Creating a Pod from YAML from scratch

```
pod-example.yaml

apiVersion: v1
kind: Pod
metadata:
  name: nginx-front
  labels:
    name: nginx-front
    tier: frontend
spec:
  containers:
    - image: nginx:latest
      name: nginx-main
      ports:
        - containerPort: 8080
```

2.2 Images Used by Pods

2.2.1 Introduction to Container Registries

An **container registry or image registry** (also called an **image repository**) is a centralized platform or service that stores, manages, and distributes container images. These images are essential for deploying and running containerized applications using technologies like Docker or Kubernetes. Below is a breakdown of the key concepts: **Container Images**: A container image is a lightweight, standalone package that includes everything needed to run a piece of software, such as the application code, runtime, libraries, and system tools. These images are used by container runtimes (like Docker or Kubernetes) to create and run containers.

Registry/Repository: While the terms **registry** and **repository** are sometimes used interchangeably, they have distinct meanings:

- A **repository** is a collection of related container images, often grouped by versions (tags). For example, a repository might contain different versions of a web application (e.g., v1.0, v2.0).
- A **registry** is a service that hosts repositories and provides APIs for uploading, managing, and retrieving container images. It acts as a centralized hub where developers can store and share their container images.

Key Functions of a Container Registry:

- **Storage**: Stores multiple versions of container images, allowing for version control and management.
- **Distribution**: Provides access to images, allowing teams or automated systems to pull the images for use in different environments (development, testing, production).
- **Security**: Often includes features like access control, image scanning for vulnerabilities, and signing images to ensure their integrity.
- **Organization**: Helps organize and manage images by namespaces, repositories, and tags.

Examples of Popular Container Registries:

- **Docker Hub**: A public registry managed by Docker where anyone can publish or access images.

- **Google Container Registry (GCR)** and **Amazon Elastic Container Registry (ECR)**: Cloud-specific registries provided by Google Cloud and AWS, respectively, for managing container images in cloud-based applications.
- **Harbor**: An open-source, on-premise container registry that offers enhanced security and performance.

In short, an **container registry** is a key part of the container ecosystem, enabling teams to efficiently manage, share, and deploy containerized applications.

2.2.2 How can I specify registry for Docker?

In docker there are two ways of specify the registry:

- **Using the docker login command:**

```
$ docker login <registry_fqdn>:<registry_port>
```

```
$ docker login <registry_fqdn>:<registry_port> -u <username> -p <password>
```

NOTE

The port is not mandatory, there are many public registries which accepts the query without port.

- **Editing docker configuration** in `/etc/docker/daemon.json`

```
{
  "insecure-registries" : [
    "<registry_fqdn>:<registry_port>"
  ],
  "registry-mirrors": [
    "<registry_fqdn>:<registry_port>"
  ]
}
```

NOTE

A mirror in the context of Docker registries is essentially a duplicate of a registry that serves the same images as the original. Mirroring is useful for:

- **Load Balancing**: Distributing image pulls across multiple servers to reduce the load on a single registry.
- **Improved Availability**: If one registry goes down, clients can still pull images from the mirror.
- **Geographic Proximity**: Providing a closer registry for users in different regions can reduce latency and improve pull speeds.
- **Caching**: Mirrored registries can cache images that are frequently pulled, which can be beneficial in large environments with multiple users or CI/CD pipelines.

When you specify a mirror in your Docker configuration, Docker will check the mirror for images first before falling back to the primary registry.

NOTE 2

If your **registry is secured** and uses TLS (Transport Layer Security), **you do not need to specify anything** on your `/etc/docker/daemon.json`. You can have a config like this:

```
{  
    "insecure-registries": [],  
    "registry-mirrors": [],  
    "log-driver": "json-file",  
    "storage-driver": "overlay2"  
}
```

Instead of adding the configuration to `/etc/docker/daemon.json`, you should ensure that **Docker is configured to trust the TLS certificates** presented by your registry.

Take into account that the registry should be correctly configured to serve over HTTPS with the TLS certificate properly installed.

If you are using a **self-signed certificate or a certificate from a private CA**, you need to add the CA certificate to Docker's trusted certificates:

```
/etc/docker/certs.d/<registry_fqdn>:<registry_port>/ca.crt
```

2.2.3 How can I specify registry for K8s?

Kubernetes does **not automatically inherit any custom Docker registry configuration** (such as `docker login` or Docker's `/etc/docker/daemon.json`). This is because Kubernetes uses its own container runtime (which could be Docker, containerd, or another runtime).

The most explicit way to ensure Kubernetes pulls images from the correct registry is to **fully specify the registry in the image field of your Pod** or Deployment manifests. But what if you need authentication on the registry?

If you're using a private registry that requires authentication, you can configure Kubernetes to use specific credentials via image pull secrets. This way, every time the Pod pulls an image from the registry, it will authenticate correctly.

1. Create a Secret:

```
$ kubectl create secret docker-registry myregistrykey \  
--docker-server=https://pearl.harbor:8443 \  
--docker-username=myusername \  
--docker-password=mypassword \  
--docker-email=myemail@example.com
```

2. Attach the Secret to Your Pod or Deployment: You can specify the secret in the `imagePullSecrets` section of your manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mycontainer
      image: pearl.harbor:8443/myproject/myimage:latest
    imagePullSecrets:
      - name: myregistrykey
```

2.2.4 How is that I don't specify anyone and I can pull images?

This is because if you don't specify any registry in Docker or K8s, for both the **default registry** is **Docker Hub (docker.io)**.

2.2.5 Docker commands to interact with Registries

Search for all images with specific name on your configured registries:

```
$ docker search <image_name>
```

To download images from specific registry:

```
$ docker pull registry[:<port>]/<image_name>:<image_tag>
```

Save a Docker image into a tar file:

```
$ docker save -o my_image.tar <image_name_or_id>
```

Load Docker image from tar file:

```
$ docker load -i my_image.tar
```

Save the changes made inside a running or stopped container into a new Docker image:

```
$ docker commit <container_id_or_name> <new_image_name:tag>
```

NOTE

You can add a commit message with the -m option to describe the changes made in the container:

```
$ docker commit -m "Installed mtr" <container_id_or_name> <new_image_name:tag>
```

You can also specify the author of the commit using the -a option:

```
$ docker commit -a "titocampis" <container_id_or_name> <new_image_name:tag>
```

WARNING

While the `docker commit` command allows you to save changes from a container to a new image, it is not recommended for production or long-term projects. This command captures the current state of a container's filesystem, but it lacks transparency, reproducibility, and version control—key practices in containerized environments.

Why This Is Not a Best Practice:

- **Lack of Documentation:** The changes made inside a container are not easily traceable, which can lead to confusion about how the image was created.
- **No Version Control:** There's no way to track incremental changes, making it difficult to maintain or update the image over time.
- **Reproducibility:** `docker commit` does not capture the configuration (like environment variables, exposed ports, or volumes). This can lead to inconsistencies between different environments or when others try to reproduce your setup.

Best Practice: Use a Dockerfile Instead The proper way to create Docker images is by using a **Dockerfile**, which allows you to:

- **Explicitly define** all the steps needed to build the image.
- **Version and track** changes in source control (e.g., Git), ensuring consistent builds.
- **Easily reproduce** the environment across different systems and teams.

By following best practices and using a **Dockerfile**, you ensure a **maintainable, reliable, and scalable** approach to containerized application development.

2.2.6 Pushing a local image to a public or private container registry

Pushing a local image to a public or private container registry involves a few key steps, whether you're working with Docker or any containerized platform like Kubernetes. I'll walk you through the general process, assuming you're using Docker, but this can be easily adapted for other container engines.

1. Build the Docker Image Locally (if not already built):

```
$ docker build -t <image_name>:<image_tag>
```

2. Tag the image for the Registry: You need to tag your image with the repository URL or registry path before pushing it. The format is:

```
$ docker tag <local_image_name>:<image_tag> <registry-url>/<repository-name>/<image-name>:<tag>
```

3. Authenticate to the Registry:

```
$ docker login <registry_fqdn>:<registry_port>
```

4. Push the Image to the Registry::

```
$ docker push <registry-url>/<repository-name>/<image-name>:<tag>
```

2.3 ReplicaSet

[Official K8s Doc](#)

2.3.1 What is replica and why we need a Replication-Controller?

ReplicaSet is a K8s controller, commonly called **Replication-Controller**. Controllers are the brain behind K8s, they are the processes that monitor K8s Objects and respond accordingly.

Imagine the basic scenario, one app running in one pod. What happens if the app crashed and the Pod fails? Users will no longer be able to access our application. To prevent users from losing access to our application, we would like to have more than one application instance or Pod running at the same time, or some object that controls the state of the instance and restore it. So this is the work of Replication Controller.

Replication Controller helps us run multiple instances of a single Pod in the K8s Cluster, regardless the Node. Besides, the Replication Controller is in charge of automatically bringing up new pods when the existing ones fail. Its work is to have always a number of **replicas** or instances of pods running healthy, regardless of whether there are 1 or 100. Another reason to have a Replication Controller is to share load across the Pod instances and to scale up easily.

Note

There are two similar terms, which both have the same purpose, but they are not the same: **ReplicaSet** and **Replication Controller**.

- **Replication Controller:** is the older technology that is being replaced by ReplicaSet.
- **ReplicaSet:** new recommended way to set up replication

However, whatever we discussed in the previous few slides remain applicable to both these technologies. We will see differences in other sections, now we are going to focus on ReplicaSets.

2.3.2 How to define a ReplicaSet?

```
replicaSet-template.yaml

apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  tier: front-end
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: front-end
  template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        tier: front-end
    spec:
      containers:
        - image: nginx
          name: nginx-container
```

- **apiVersion:** apps/v1
- **.spec.template:** it is where we **define** the **Pod instance** that we want to replicate
- There are **2 metadata**:
 - One for ReplicaSet metadata (name, labels, etc.) (..labels)
 - Another for Pod metadata (name, labels, etc.) (....labels)
- There are **2 spec**:
 - One for ReplicaSet specifications (.spec)
 - Another for Pod instance specifications (...spec)
- **.spec.replicas:** it is where we define the number of running replicas desired.
- **.spec.selector:** it helps the ReplicaSets to identify what pods fall under it, this is **because ReplicaSet can manage Pods created before the creation of the ReplicaSet itself that match labels specified in the selector!!**
- **.spec.selector.matchLabels:** it indicates to selector which labels of the Pods can take into consideration to manage those who have them.

Note

In ReplicaSet **.spec.selector** is **mandatory**, if it is not specified the object will be wrong.

2.3.3 How to create a ReplicaSet using kubectl?

```
$ kubectl create -f replicaSet-template.yaml
```

```
$ kubectl create -f replicaSet-template.yaml
```

```
$ kubectl get rs
```

Note

In Replication Controller there are the following difference:

- **apiVersion:** v1
- **kind:** ReplicationController
- **.spec.selector** is not mandatory but it can be specified as well. If not specified the value will be get it from **.spec.metadata.labels**.

2.3.4 What is the deal with labels and selectors?

Why do we label our Pods and objects in K8s? Let's look a simple scenario: we run our front-end application as **3 pods**, we would like to create our **ReplicaSet** to ensure there are 3 active Pods at any time. If there are pods created with the **.metadata.labels** specified in **.spec.selector.matchLabels** the **ReplicaSet** will monitor the already created running pods. In case they are not created, the **ReplicaSet** will create them for you, because the role of the replicaset is to monitor Pods and if any of them were to fail, deploy new ones. It is in fact a process that monitors the pods.

How does the **ReplicaSet** know the pods to monitor? They can be hundreds, thousands of Pods running in the Cluster running different applications. This is why labeling Pods during creation comes in handy, so this labels can be provided as a filter for **ReplicaSet** in the **.spec.selector.matchLabels** in the same way we defined in the **.metadata.labels** on the Pod. This way, the ReplicaSet knows which pods to monitor.

The same concept as labels and selectors is used in many other places throughout K8s.

EXAMPLE

Imagine you have 3 pods running in your Cluster without a Controller, all with the **.metadata.labels** (**tier:front-end**). If one of them fails, it won't be restored. However, if we create a ReplicaSet after pods creation, with **.spec.replicas = 3** and the **.spec.selector.matchLabels** (**tier:front-end**), the ReplicaSet won't deploy a new instance of Pod, as 3 of them with matching labels are already created.

Nevertheless, we must still define a **.spec.template section**, because in case the pods were to fail in the future, the ReplicaSet must know how to create a new one, so **.spec.template section** is required.

2.3.5 How to scale up and down a ReplicaSet?

There are multiple ways to do it, we are going to explain 3:

1. Edit the ReplicaSet definition file (YAML) with the replicas desired and apply its configuration using a kubectl command.

```
$ vim replicaSet.yaml
```

```
$ kubectl apply -f replicaSet.yaml
```

2. Edit the ReplicaSet object with the replicas desired with kubectl edit.

```
$ kubectl edit <replicaset-object-name>
```

3. Use the kubectl scale command. Using the definition file or the K8s object, it doesn't matter.

```
$ kubectl scale --replicas=<new_number_of_replicas> -f replicaSet.yaml
```

```
$ kubectl scale --replicas=<new_number_of_replicas> replicaset <rs-name>
```

WARNING

```
$ kubectl scale --replicas=3 -f replicaSet.yaml
```

Notice that use the command above only modifies the replicas in the object generated from this file, it doesn't modificate the number of replicas in the definition file.

2.3.6 Tips in Creating and manipulating ReplicaSet

```
$ kubectl scale --replicas=2 rs new-replica-set
```

2.4 Deployments / StatefulSet

[K8s Official Deployment Doc](#) [K8s Official StatefulSet Doc](#)

2.4.1 Introduction to Deployments

To understand Deployment we are going to see an example. Imagine a **Production Environment**, where a Web App must have deployed multiple instance of its Web App Pods. In addition, whenever newer versions of application bills become available on the Docker Registry, you would like to upgrade your Docker instances seamlessly. However, you don't want to upgrade all the instances at once, so this may impact users accessing our application, so you might want to upgrade them one after the other, and that kind of upgrade is known as **rolling updates**.

What's more, when you update the image to the latest version and it fails, you would like to be able to roll back the changes that were recently carried out.

Finally it's supposed you would like to make multiple changes to your environment, such as: upgrading the underlying web server versions, scaling your environment, modifying the resource allocations, etc. You don't want to create each change immediately once the command is run, instead you would like to create a pause in your environment, make the changes, and then resume so that all the changes are rolled out together.

All these functionalities are provided by K8s Deployment or StatefulSet. Apps are encapsulated in Pods, multiple such pods are deployed using ReplicaSets. And then comes Deployment which is higher in the hierarchy. **Deployment** object provides us with the capability to upgrade the underlying instances seamlessly using **rolling updates**, undo changes, pause, resume changes as required.

2.4.2 How to define a Deployment

The contents of the **Deployment** definition file are exactly similar to the **ReplicaSet** definition file, except from **kind: Deployment**.

```
deploy-template.yaml

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

2.4.3 How to create a Deployment using kubectl?

```
$ kubectl create -f deploy-template.yaml
```

NOTE

When a **Deployment** is created in K8s Cluster, it automatically creates a **ReplicaSet** in the same namespace as Deployment. Ultimately, the **ReplicaSet** creates Pods.

Actually, there are no more differences in the creation between **Replicaset** and **Deployments**. The differences are related to the function of the Deployment object itself, as we discussed before.

2.4.4 What is a StatefulSet?

2.4.5 How to create a StatefulSet using kubectl?

```
$ kubectl create -f deploy-template.yaml
```

NOTE

When a **Deployment** is created in K8s Cluster, it automatically creates a **ReplicaSet** in the same namespace as Deployment. Ultimately, the **ReplicaSet** creates Pods.

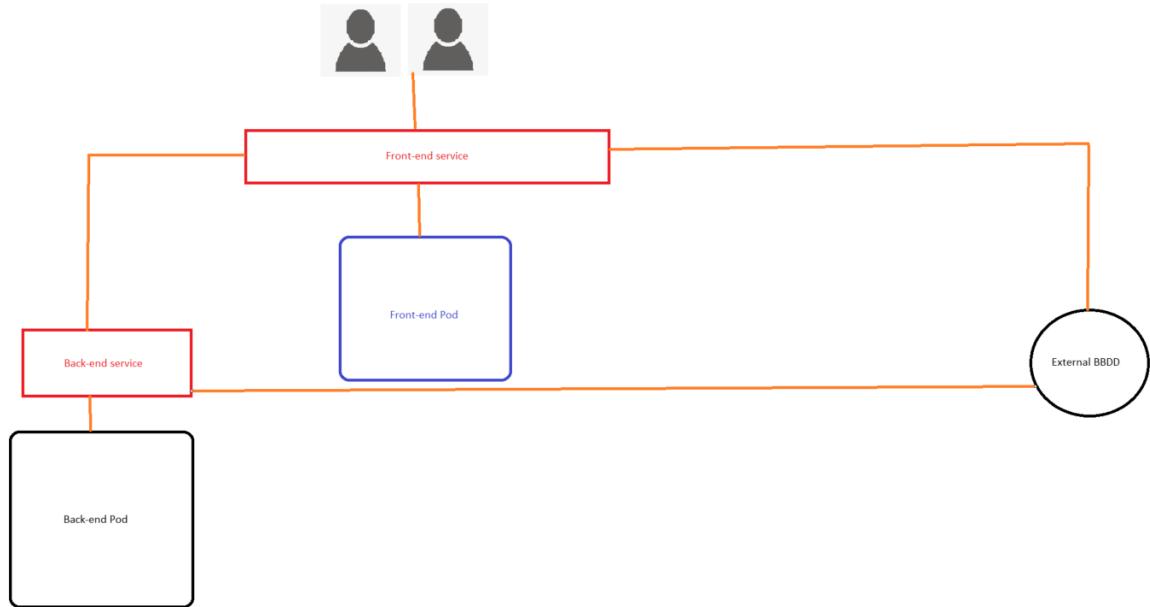
2.5 Services

[Official K8s Doc](#)

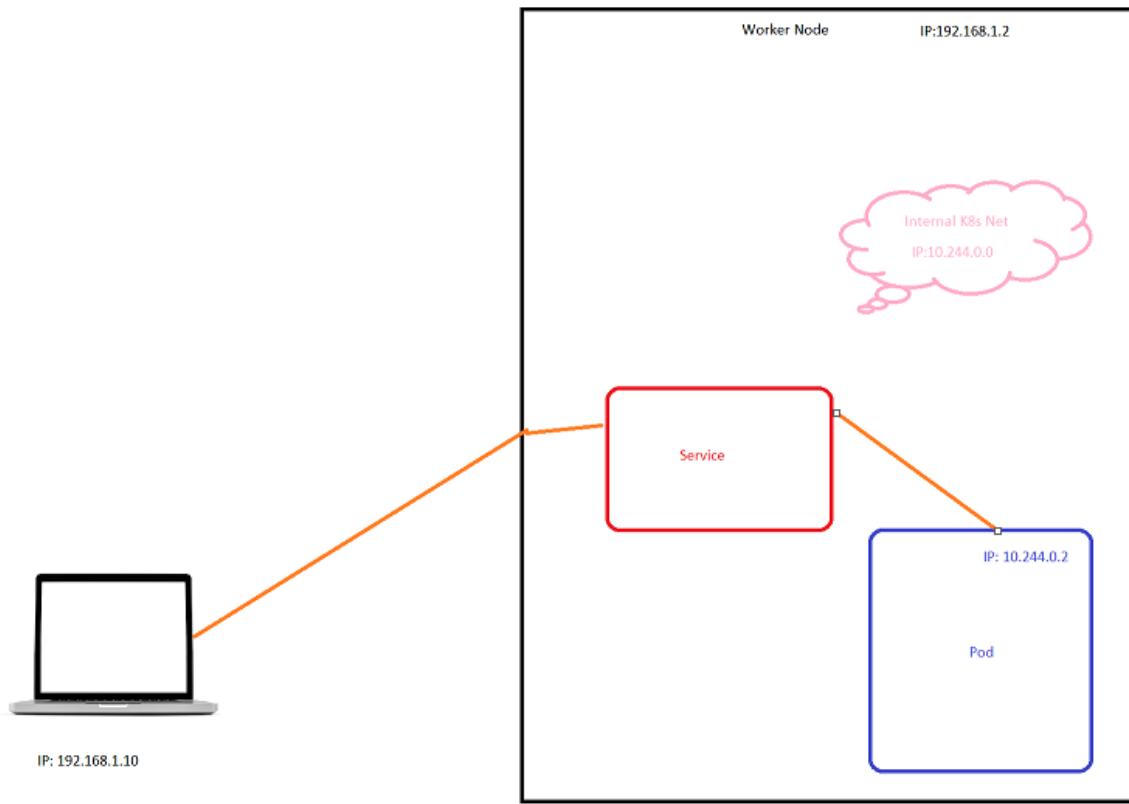
2.5.1 Introduction to Services

K8s Services enable communication between various components within and outside of the application. K8s Services help us connect applications together with other applications or users.

Imagine an application splitted in 3 microservices: front-end, back-end and external DB. Are their services which enable connectivity between these groups of Pods.



But in a higher level, how the users can access to applications running in Pods in a Worker Node? We cannot access from outside the cluster to the Pod IP directly, because it is constantly changing and it just exists under K8s Network. Inside the cluster we can connect `curl http://10.244.0.2`. But from outside? Now is when enters Service.



Service is another K8s object like Pods, or ReplicaSets which forward requests to Pods.

There are three types of **Services**:

- **Node Port Service:** it makes an internal port accesible on a port on the Node.
- **ClusterIP Service:** it creates a Virtual IP inside the cluster to enable communication between different applications (such as a set of front-end servers to a set of backend servers)
- **LoadBalancer:** it provides a Load Balancer to our application, for example to distribute the load across the different web servers in your front-end tier.

2.5.2 NodePort Services

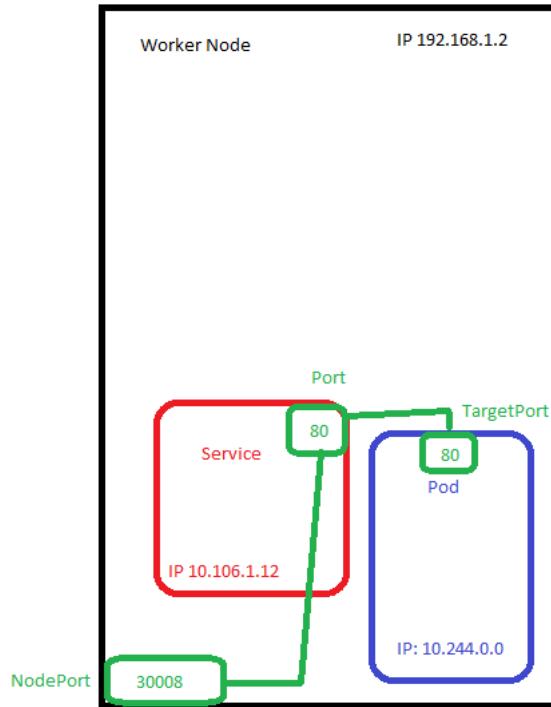
NodePort Services listens on a port on the Node and forward requests on that port to a port on the Pod running the application. It makes a mapping of ports, mapping a port on the nodo to a port on the Pod. The Service is like a Virtual Server inside the Node. Inside the Cluster it has its own IP adress, called the ClusterIP of the Service.

There are 3 ports involved from the viewpoint of the Service:

- **TargetPort:** Pod port where the app is running (80), that where the service forwars their request to.
- **Port:** Service port where the app is running (80)
- **NodePort:** Node port used to access the app externally (30008)

WARNING

NodePorts can only be in a valid range which by default is from 30.000 to 32.767. As well, as Pods can run in any Node, **NodePort** are assigned to all the nodes at the same time, so you will be able to reach the application on each **Worker Node** on the **NodePort** assigned.



2.5.3 How to define a NodePort Service?

```
service-template.yaml

apiVersion: v1
kind: Service
metadata:
  name: myapp-service
  labels:
    tier: front-end
spec:
  type: NodePort
  selector:
    tier: front-end
  ports:
    - nodePort: 30008
      port: 80
      targetPort: 80
```

As we discuss before, in the spec section we have only two fields:

- **type:** NodePort, ClusterIP, LoadBalancer
- **ports:** list of port maps always agrouping (targetPort, Port and nodePort). You can have multiple such port mappings within a single service.

NOTE

If targetPort is not specified it is assumed to be the same as port, and if nodePort is not specified a free port on the Worker Node in the valid range between 30.000 and 32.767 is automatically assigned.

So, we have all the information in, but something is really missing. There is nothing in the definition file that connects the service to the Pod. We have simply specified the target port but we didn't mention the targetPort on which Pod. There could be hundreds of other Pods with applications running on port 80. So how do we do that?

As we did with the ReplicaSet, it is used a technique that we will see very often in K8s, **labels and selectors!** Selectors and labels are used to link Services with Pods. But now **without matchLabels**

2.5.4 How to create a Service using kubectl?

```
$ kubectl create -f service-template.yaml
```

```
$ kubectl get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
myapp-service	NodePort	10.106.127.123	<none>	80:30008	5m

PORT(S): 80:30008/TCP

PORT(S): port:nodePort/TCP

You can access to the application from a terminal connecting with the **Worker Node IP** and **port**, because the Server will be listening on it.

```
$ curl 192.168.1.2:30008
```

2.5.5 One Service, multiple instance of the same Pod

So far, we have talked about a service mapped to a single Pod, but that's not the case all time. What do we do when we have multiple Pods? In a production environment it is common to have multiple instances of the same Pod, just to ensure High Availability if one replica fails and load balancing. In this case we have multiple Pods running our application, with all the same labels with a key-value app=myapp. This same label is created in the **.spec.selector** section during the creation of the Service.

So when the Service is created, it looks for a matching Pod with the labels specified in **.spec.selector**. The **Service** automatically selects all the Pods as endpoints to forward the external request.

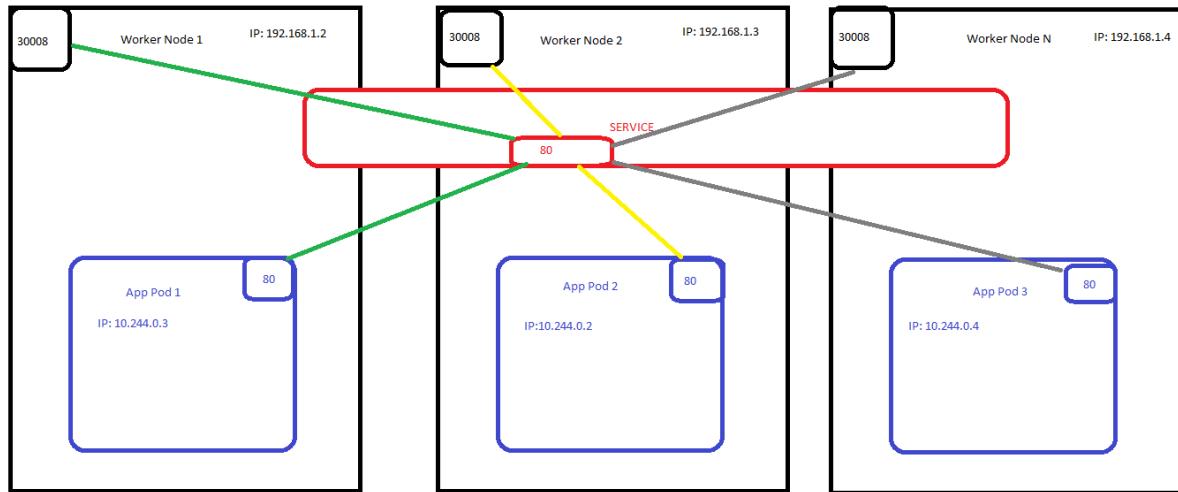
NOTE

To load the balance between the instances, **Service** by default has the following configuration:

- **Balance Policy:** Random
- **SessionAffinity:** yes

But what happens when the Pods are distributed across multiple Nodes (common case). In this case when we create a service, without us having to do any additional configuration (than labels), K8s automatically

creates a Service that spans across all the Nodes in the cluster and maps the target port to the same Node port on all the Nodes in the Cluster.



This way, you can access your application using the IP of any Node in the cluster and using the same port number:

```
$ curl 192.168.1.2:30004
```

```
$ curl 192.168.1.3:30004
```

```
$ curl 192.168.1.4:30004
```

To summarize, in any case: whether it be a single Pod on a single Node, or multiple Pods on a single Node, or multiple Pods on Multiple Nodes. **The Service is created exactly the same** without you having to do any additional steps during the Service creation. **When Pods are removed or added with the correct labels the Service is automatically updated** making it highly flexible and adaptive. Once created, you don't have to make any additional configuration changes.

2.5.6 Service ClusterIP

Fullstack applications usually have different kinds of pods hosting different parts of an application. You may have a set of Pod instances of front-end a set of Pod instances of back-end and a set of Pod instances of for example redis (key-value store) and another set of pods running for example a persistent DB like MySQL. Front-end servers needs to communicate with the back-end servers, and to the database and also to redis service, etc. So what is the best way to establish the connection between these services or tiers of my application?

Pods all have an IP address assigned to them, but these IPs, as we know, are not static, these Pods can go down anytime and new Pods are created all the time, so you cannot rely on these IP addresses for internal communication between the application.

ClusterIP Services allow different applications or microservices of applications connect to each other inside the cluster, with no need to go outside. This service creates a kind of interface to group all the Pods of a microservice for the other Pods to access this service. In our example, the front-end Service works as unique interface to connect with front-end, same with backend, and same with redis. This enables us to easily and effectively deploy a microservice based application on K8s Cluster. **Every layer can now**

scale or move as required without impacting communication.

A ClusterIP service is the **default service type** in Kubernetes, and it exposes the service **only within the cluster**. It assigns a virtual IP (known as the ClusterIP) that is **accessible from within the cluster pods and other services**.

The ClusterIP service is only accessible inside the cluster (i.e., from within the pods or other nodes that are part of the cluster network). The **worker node itself is not part of the Kubernetes pod network**, and its traffic is not routed from the worker nodes directly, only from within the pods or other Kubernetes-managed processes.

2.5.7 How to define a ClusterIP Service?

```
cluster-ip-service.yaml

apiVersion: v1
kind: Service
metadata:
  name: front-end
  labels:
    name: myapp
spec:
  type: ClusterIP
  ports:
  - port: 80
    targetPort: 80
```

As in the **NodePort** type:

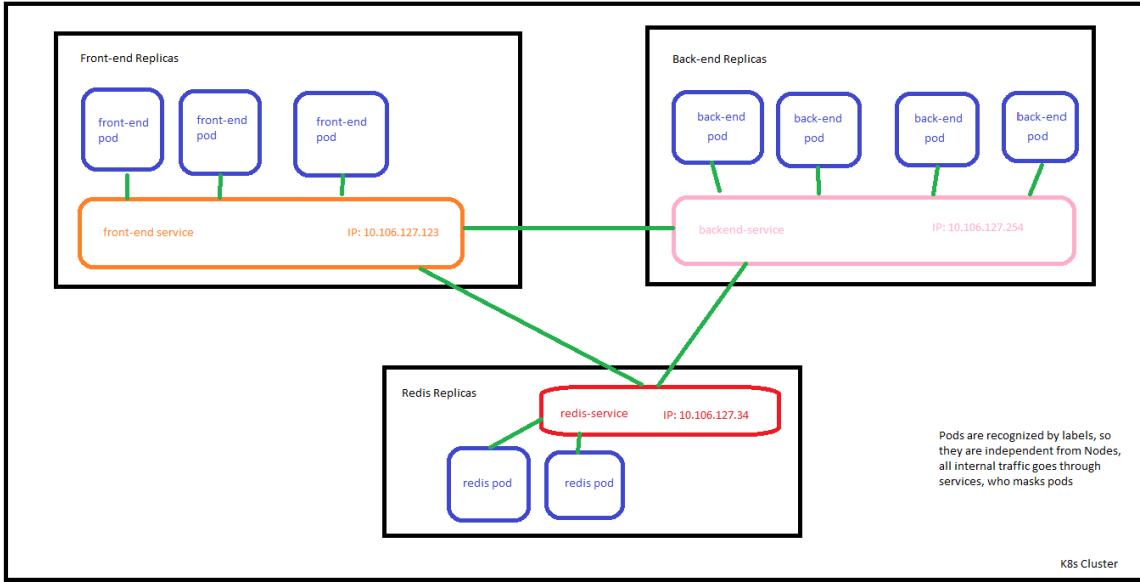
- **targetPort:** port exposed by the Pod
- **port:** port exposed by the service
- **selector:** to link the service to a set of Pods, copying the labels from the front-end Pod definition.

NOTE

The default Service type is **ClusterIP** so if we do not define Service type, by default it would be **ClusterIP**

```
$ kc get svc
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
front-end	ClusterIP	10.106.127.123	<none>	80:30008	5m

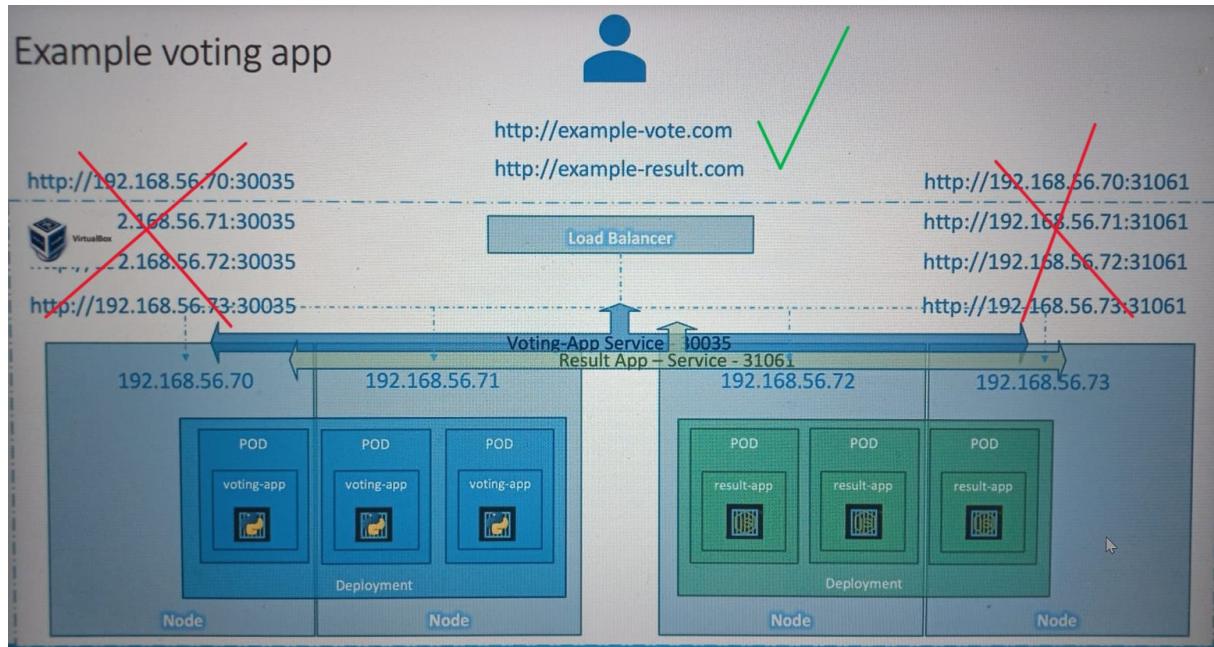


2.5.8 Service LoadBalancer

Imagine the case of a web application to vote and see results, you have both services with their associated Pods with the apps running. There are many replicas of each Pod running on different Nodes, and the Service is a NodePort service. As we know a NodePort services expose the app in a specific port of the Worker Nodes, so... What URL users should use to access the voting web, or the result web? They can use 4 IP's with its port to vote and 4 IP's with its port for the results. This is not what users want, they want single urls (like www.results.com).

So, **LoadBalancer Service** create a new VM for load balancer configuring in this VM a suitable load balancer as HAProxy or NGINX. Then configure the Load Balancers (as NGINX) to route traffic to the underlying Nodes and expose a single endpoint.

If we are on a Supported Cloud Platform as Google, AWS, Azure... It could leverage the native load balancer off that cloud platform. K8s has support for integrating with the native load balancers of certain cloud providers, configuring that for us. So all we need to do is setup our service type as LoadBalancer, letting the rest as NodePort service.



2.5.9 How to define a NodeBalancer Service?

node-balancer-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  labels:
    name: nginx
spec:
  type: LoadBalancer
  ports:
  - nodePort: 30008
    port: 80
    targetPort: 80

```

WARNING

LoadBalancer Services only work with Supported Cloud Platforms, as Azure, AWS, Google Cloud...

2.5.10 K8s DNS: How to reach the services from other Pods?

To reach a ClusterIP service from other pods within the same Kubernetes cluster, you can use the service's DNS name or its ClusterIP address. The Kubernetes service discovery and networking setup make it easy for pods to communicate with each other using these methods.

Kubernetes automatically sets up DNS for services within the cluster. Each service gets a DNS name based on its name and the namespace it is running in. DNS Name Format:

```
<service-name>.<namespace>.svc.cluster.local
```

Kubernetes will automatically resolve this DNS name to the ClusterIP and route the traffic to the service, which in turn forwards it to the appropriate pod.

NOTE

If the pod accessing the service is in the same namespace, you can use the `service_name` directly:

```
$ curl <service_name>:<service_port>
```

You can also reach the service by directly using its ClusterIP. However, this requires knowing the IP address assigned to the service and it is not a best practice. It's better to rely on DNS instead of the IP address, as the IP can change if the service is deleted and recreated, whereas the DNS name is stable.

```
$ curl <service_clusterIP>:<service_port>
```

2.5.11 CurlPod: How to reach the services from the Nodes?

If the Service is `NodePort` type, you will be able to reach it directly by:

```
$ curl localhost:<service_port>
```

But, if the Service is of another type, they will be only accessible inside the cluster (i.e., from within the pods or other nodes that are part of the cluster network). The **worker node itself is not part of the Kubernetes pod network**, and its traffic is not routed from the worker nodes directly, only from within the pods or other Kubernetes-managed processes.

The best approach then will be to run a **Pod** inside the namespace of the **Service** and use the curl tool:

```
$ kubectl run curlpod --image=alpine/curl -it --rm --restart=Never \
--command -- curl http://<service_name>:<service_port>
```

2.5.12 Port Forward: How to reach the services from outside the cluster

2.5.12.1 Introduction

Port-forwarding in Kubernetes (K8s) is a mechanism that allows you to access a specific resource (like a service or a pod) inside a Kubernetes cluster from your local machine. Typically, resources inside the cluster are isolated, and they can't be directly accessed from outside the cluster. However, by using port-forwarding, you can temporarily expose a local port to connect to a port of a service or a pod inside the cluster, which is especially useful for debugging, testing, or accessing internal applications.

Port-forwarding works by forwarding traffic from a local port on your machine to a port on a resource (like a service or pod) inside the Kubernetes cluster. This allows you to communicate with applications running inside the cluster without needing to expose them externally.

2.5.12.2 Use Cases

- **Access** internal services (e.g., databases or web services) without exposing them externally.
- **Debug** and **test** applications by directly accessing their endpoints.
- For local development, interact with in-cluster resources as if they were running locally.

2.5.12.3 kubectl port-forward

Services in Kubernetes manage access to a group of pods. By port-forwarding to a service, you can reach any pod behind the service (whichever the service forwards traffic to). Here's how to port-forward a service to your local machine.

```
$ kubectl port-forward service <service_name> <local_port>:<service_port>
```

Sometimes, you may want to port-forward directly to a specific pod. This can be useful if you want to directly access an individual pod (not through a service), or if you're testing or debugging a particular instance of your application.

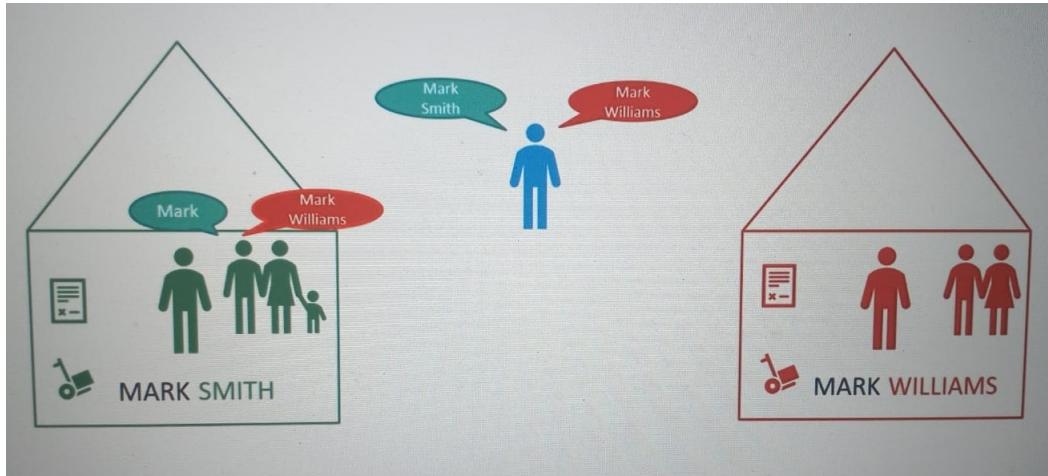
```
$ kubectl port-forward pod <pod_name> <local_port>:<image_port>
```

2.6 Namespaces

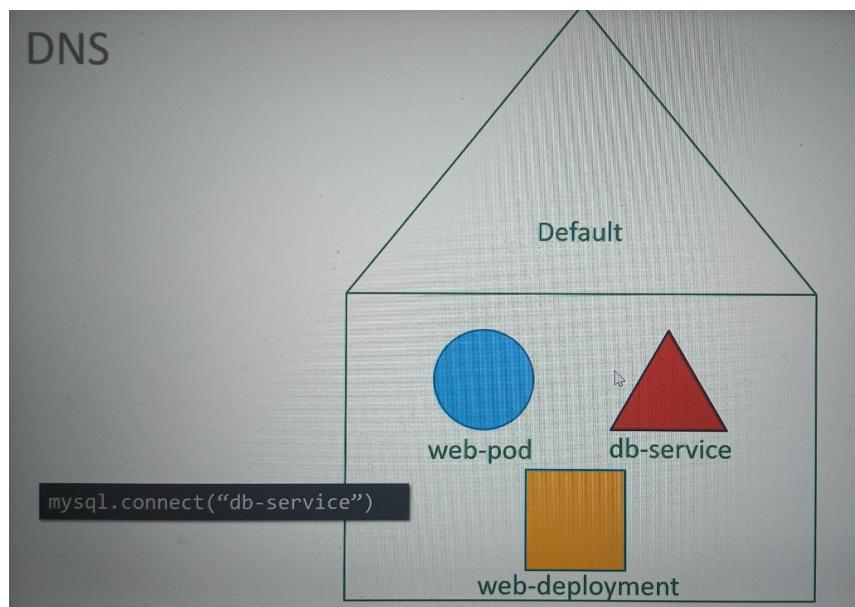
Official K8s Doc

2.6.1 Introduction to Namespaces

Let's start this section with an analogy. There are 2 boys called Mark, as they have the same name, people in general call them for their last name. But, in his houses, his family call them for their names, but out of the house, they should be called by the full-name (name and last name). In this analogy, houses are namespaces.

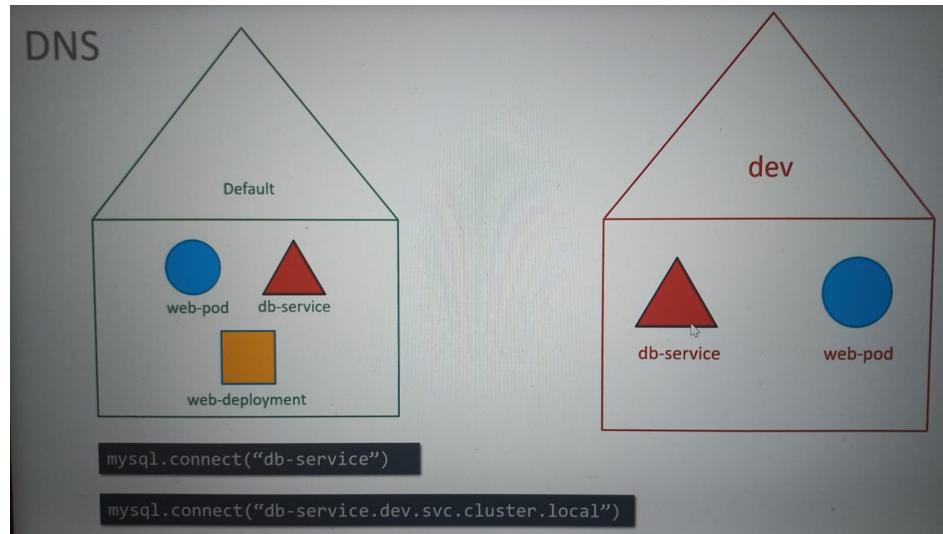


So, K8s resources within a **Namespace** can refer to each other simply by their names. In the following example the web app Pod can reach the db-service simply using the hostname db-service.



If required, the web app pod can reach a service in another Namespace as well, but for this, you must **append the name of the Namespace to the name of the service**. Using the format:

```
servicename.namespace.svc.cluster.local
```



You can use this format because when a Service is created, a DNS entry is added automatically in this format. Looking closely at the DNS name of the service:

- **.cluster.local:** default domain name of the K8s cluster
- **.svc:** sub domain for service
- **.dev:** sub domain for namespace
- **db-service:** sub domain for the object named itself



2.6.2 K8s automatically created Namespaces

So far, all the creation we have been done in K8s with kubectl has been done in a house. Yes! We have been deceived. This **Namespace** is **Default**. and it is created automatically by K8s when the cluster is first setup.

K8s creates a set of Pods and Services for its internal purpose, such as those required by the Networking solution, DNS Server, etc. To isolate them from the user and to prevent accident deletions or modifications of these objects K8s creates them under another **Namespace** created at Cluster startup named **kube-system**.

A third **Namespace** created by K8s automatically is called **kube-public**, this is where resources that should be made available to all users are created.

If your environment is small or you are playing around with a small cluster, you shouldn't really have to worry about **Namespaces**. You can continue working in the **default Namespace**. However, as in when you grow and use a K8s cluster for enterprise or production purposes, you may want to consider the use of **Namespaces**.

2.6.3 Create own Namespaces

Imagine that you want to isolate dev environment resources from production resources, you can create a different Namespace for each of them. That way, if you are working in the dev environment, you don't accidentally modify production objects.

2.6.3.1 Difficult way to create a Namespace

Like any other object in K8s, it is necessary to use a **Namespace** definition file to create a new Namespace:

```
namespace-template.yaml  
apiVersion: v1  
kind: Namespace  
metadata:  
  name: test-namespace
```

Now run the following command and you will have your new Namespace ready!

```
$ kubectl create -f namespace-template.yaml
```

2.6.3.2 Easy way to create a Namespace

```
$ kubectl create namespace tst-namespace
```

2.6.4 kubectl commands to manage Namespaces

Looking for resources

```
$ kubectl get po -n <namespace_name>
```

Get a resource for all namespaces

```
$ kubectl get <resource> --all-namespaces [-A]
```

Creating resources

```
$ kubectl create -f pod-def.yaml -n <namespace_name>
```

See all namespaces in our K8s Cluster

```
$ kubectl get ns (namespaces)
```

WARNING

If you do not pass namespace to kubectl commands, it would be set as the current namespace. If you do not switch it once entered the cluster, it is default

Switch the current Namespace

```
$ kubectl config set-context --current --namespace <desired_namespace>
```

NOTE

Context are used to manage multiple Clusters and multiple environments from the same management system. It is a totally separate topic to discuss and requires its own lecture.

2.6.5 Defining Namespace in K8s Objects

The definition of Namespace to create objects on it must be done in .metadata.namespace.

EXAMPLES:

```
pod-namespaced.yaml

apiVersion: v1
kind: Pod
metadata:
  name: myapp
  namespace: dev
  labels:
    app: myapp
    tier: front-end
spec:
  containers:
    - ...
```

```
service-namespaced.yaml

apiVersion: v1
kind: Service
metadata:
  name: myapp-service
  namespace: dev
spec:
  ports:
    - nodePort: 30080
      port: 8080
      protocol: TCP
      targetPort: 8080
  selector:
    app: myapp
    tier: front-end
  type: NodePort
```

2.6.6 Namespace Policies

[Official K8s Doc](#)

Each Namespace can have its own set of policies that define who can do what. You can also assign quota of resources to each of these Namespaces (like we do in ICP). To limit resources in a Namespace, we should create a **Resource Quota**

Define it

```

resource-quota.yaml

apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: dev
spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi

```

This allows a maximum of 10 Pods, or 4 request CPU, or 5Gi memory request...

Create it in the namespace dev

```
kubectl create -f resource-quota.yaml -n dev
```

2.7 Imperative vs Declarative, apply command

2.7.1 Introduction

As we have seen, there are different ways of creating and managing K8s Objects. We created objects directly by running commands as well as using object configuration files. In the **Infraestructure as a Code World**, there are different approaches in managing the infrastructure, classified into **Imperative** or **Declarative** approaches.

Imperative approaches are if you are co-pilot in a car and you say to the driver all the actions he must take, turn to the left, and no to the right, no straight, ect. **Declarative** is to say, this is the final destination, and the driver figures out the path.

In **Imperative** approaches we say step by step what to do, and at what time. In **Declarative** we just make a list of requirements, and everything that's needed to be done to get this infraestructure in place is done by the system or the software. It is no necessary to provide step-by-step instructions. Orchestration tools like Nasible, Puppet, Chef or Terraform fall into this category.

Infrastructure as Code

Imperative

1. Provision a VM by the name ‘web-server’
2. Install NGINX Software on it
3. Edit configuration file to use port ‘8080’
4. Edit configuration file to web path ‘/var/www/nginx’
5. Load web pages to ‘/var/www/nginx’ from GIT Repo - X
6. Start NGINX server

Declarative

```
VM Name: web-server
Package: nginx:1.18
Port: 8080
Path: /var/www/nginx
Code: GIT Repo - X
```

What happens in imperative way if the VM is already installed, if you want to change the configuration for port 8080, or if you want to upgrade the NGINX version. You have to add more steps to do such things. However, in a declarative way, it is as simple as changing the version, port or what you need in the declaration file, system should be intelligent enough what has already been done and apply the necessary changes only.

The way we have to run declarative instructions in K8s is with the `kubectl apply` command, for creating, updating or deleting an object. The apply command will look at the existing configuration and figure out what changes need to be made to the system.

2.7.2 Working Imperative

Imagine you have created a K8s Object:

```
$ kubectl create -f deployment_template.yaml
```

But now, once it is running, you want to modify something on it, for example the image of the Pod container because there is a new version available. So you can use the `edit` command and do it:

```
$ kubectl edit deployment <object-name>
```

When this command is run, an interactive terminal is open with a YAML definition file similar to the one you used to create the object but with some additional fields, such as the status fields, etc. **This is not the file you used to create the object, this is a similar Pod definition file within K8s memory.** If you make changes to this file, save and quit, those changes will be applied to the live object, but never to the definition object YAML file.

This is not a good way to work for production environments, so if in the future, a teammate decides to make a change to this object unaware that a change was made using the `kubectl edit` command and deploys a new version the previous change is lost. So it is hardly recommended to use `kubectl edit` only for tests or changes that you are not going to rely on the object configuration file in the future.

The best way to do it is always declarative:

```
$ kubectl get deploy <deployment_name> -o yaml > deployment_template.yaml
```

```
$ vim deployment_template.yaml
```

```
$ kubectl apply -f deployment_template.yaml
```

NOTE

There is a way to record changes in both file and object in an imperative way, because you say directly to K8s to replace the object:

```
$ kubectl replace -f deployment_template.yaml
```

If the replace command finds that the object you want to replace does not exist it fails. Like if you run the `kubectl create` command and the object already exists, it will fail.

WARNING

You need to pay attention on `kubectl edit` command, because it is very easy to fail, as there are a lot of fields which are **unmutable**. In case of failure we hardly recommend:

- Follow the procedure of:

```
$ kubectl get deploy <deployment_name> -o yaml > deployment_template.yaml
```

```
$ vim deployment_template.yaml
```

```
$ kubectl apply -f deployment_template.yaml
```

- If the error persists, then replace the hardway:

```
$ kubectl replace -f deployment_template.yaml --force
```

2.7.3 Working Declarative

First, edit the Object declaration YAML file:

```
$ vim deployment-template.yaml
```

Once we have made the required changes, we apply them:

```
$ kubectl apply -f deployment-template.yaml
```

This way, going forward, the changes made are recorded and can be tracked as part of the change review process. The `kubectl apply` command is intelligent enough to create an object if it doesn't already exist.

If you pass a path as argument it will create or modify all objects related to the YAML files inside the path.

```
$ kubectl apply -f /path/to/yamls/
```

For more information

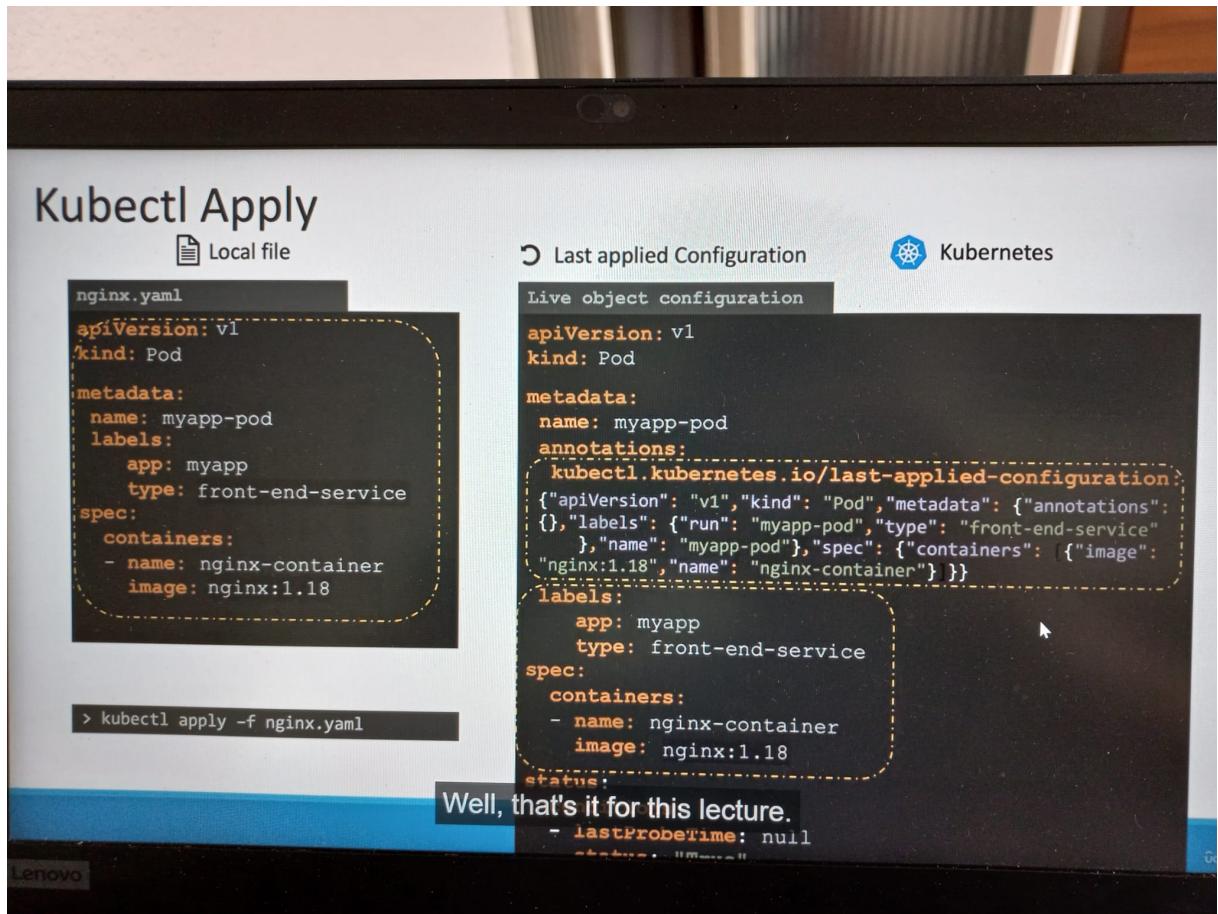
```
$ kubectl apply --help
```

2.7.4 kubectl apply

Official K8s Doc

It takes into consideration local configuration file or files to update, create, remove or do some task with K8s Objects. It compares the file and the last applied configuration before making a decision on what changes are to be made.

Also, when we use `kubectl apply` command our local YAML object configuration file is converted to a JSON format and it is then stored as the last applied configuration. Going forward, for any updates to the object, all the three files are compared to identify what changes are to be made on the live object. This new JSON file is stored on the live object configuration YAML file itself, as an annotation named last applied configuration. But it is only done when we use the apply command.



3 Labels, Selector & Annotations

Official K8s Doc

3.1 Introduction

Labels & Selectors are K8s standard method to group things together. Imagine you have a lot of animals of different species and the K8s Objects and Users want to be able to filter them based on different criteria:

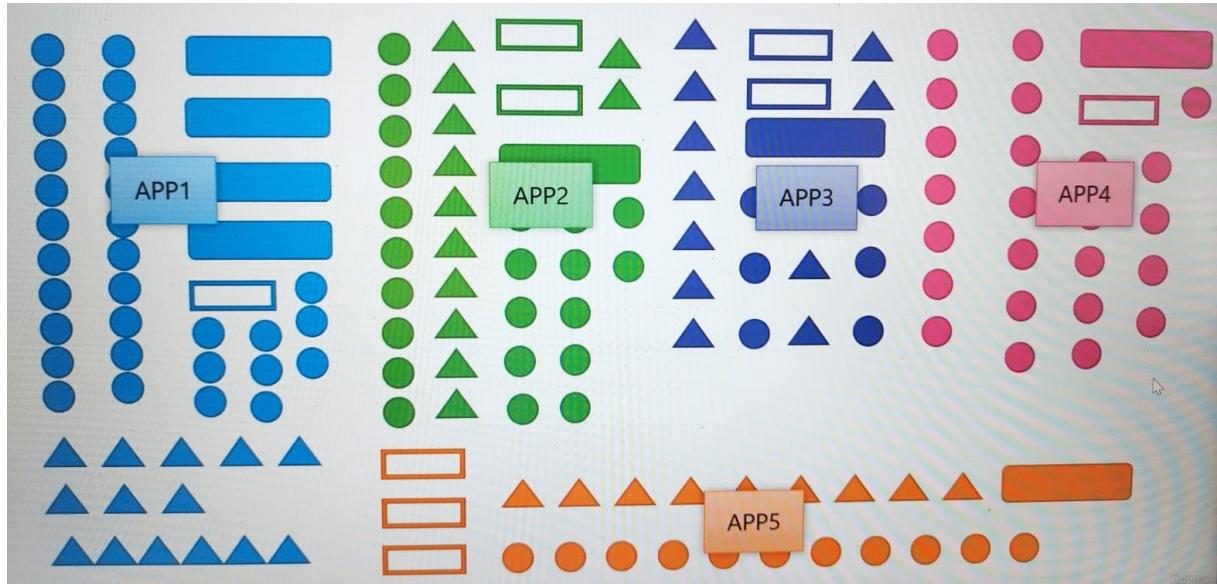
- Based on their type: mammals, fish, reptiles, birds, arthropods, amphibians...
- Based on if they live in the water or not
- Based on their climatic region
- Based on their feeding: omnivores, vegetarian, carnivore, scavenger...

We can use any classification we want, the point is we use them to filter a set of Objects based on a defined criteria.

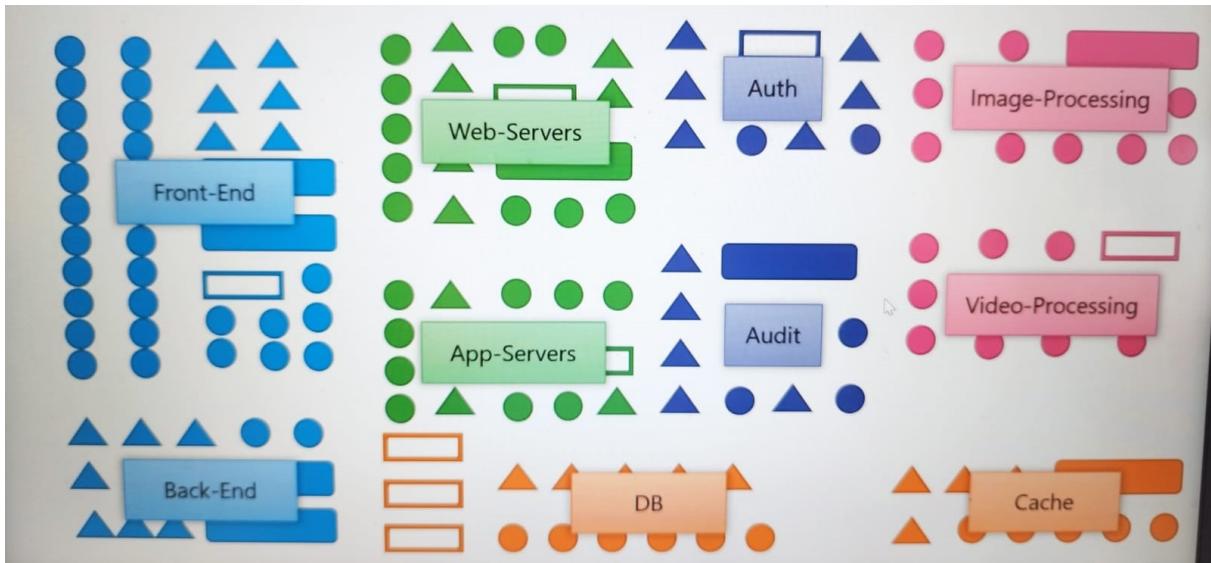
So in K8s, it is a very best practice to define Labels anywhere, so in the future you can filter and find the objects desired in every situation.

So in a K8s cluster, we will have hundreds of thousands K8s Objects, Pods, Deployments, ReplicaSets, Services, Ingress, PV's, ConfigMaps... And you will need to filter and to group all this Objects in different sets, for example to determine all the objects involved in an app, or by their functionality, you never know when you will need to filter.

Filtering by app:



Filtering by functionality:



3.2 Kubectl commands with Labels

To get Objects filtered by labels:

```
$ kubectl get <object> --selector [-l] key=value
```

It is possible to filter by different labels:

```
$ kubectl get <object> -l key1=value1 -l key2=value2 ... -l keyN=valueN
```

3.3 Annotations

[Official K8s Doc](#)

Annotations are used to record other details for informative purpose. For example: tool details, name, version, build info, owner contact, etc. You can use Kubernetes annotations to attach arbitrary non-identifying metadata to objects. Clients such as tools and libraries can retrieve this metadata. You can use either labels or annotations to attach metadata to Kubernetes objects. Labels can be used to select objects and to find collections of objects that satisfy certain conditions. In contrast, annotations are not used to identify and select objects.

4 Pod Scheduling

4.1 Manual Scheduling

In this section we will see how manually scheduling a pod on a Node, what do you do when you do not have a kube-scheduler in the cluster? So you should schedule the pods yourself.

Let's start with Pod definition. Every Pod has a field called **nodeName** (`.spec.nodeName`), which by default it is not set. You do not typically specify this field when you create the pod manifest file, K8s adds it automatically.

The kube-scheduler goes through all the pods and looks for those that do not have this property set, those are the candidates for scheduling. It then identifies the right Node for the Pod by running the scheduling algorithm. Once identified, it schedules the Pod on the Node by setting the **nodeName** (`.spec.nodeName`) property to the name of the Node.

So if there is no kube-scheduler to monitor and schedule nodes, what happens? The Pods will be always in a pending state, until you manually assign pods to Nodes yourself. And the easiest way to schedule a pod without kube-scheduler is setting the **nodeName** field to the name of the node while creating the Pod. And then the Pod will be assigned to the specified Node.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
  nodeName: node01
```

You only can specify the **nodeName** in creation time, what if the pod is already created and you want to assign the Pod to another Node? K8s will allow you to modify the **nodeName** assigned to a Pod by another way: creating a **Binding** Object and send a POST request to the pod's binding API. Thus mimicking what the actual kube-scheduler does. In the **Binding** Object you specify the `.target.node` as the name of the Node.

NOTE:

Also you can delete and create a new Pod with the same name. If the Pod is not assigned to any Port, it will have the Node to `<none>` but anyway you cannot reassign it to another node, you have to delete it and create it again. Or use the command replace!

NOTE:

With the command `kubectl replace -f` you can replace an object created by injecting them the specified in a YAML or JSON file, and then K8s will apply the changes. But it is a best practice to at first get the object, edit it and try to apply it.

In case of Node, we said that is not possible to modify them, but if you use the option `--force` the object will be automatically deleted and recreated with the specified changes, it does not matter what they were.

```
controlplane ~ ➔ kc replace -f nginx.yaml
The Pod "nginx" is invalid: spec: Forbidden: pod updates may not change fields other than `spec.containers[*].image`, `spec.initContainers[*].image`, `spec.activeDeadlineSeconds`, `spec.tolerations` (only additions to existing tolerations) or `spec.terminationGracePeriodSeconds` (allow it to be set to 1 if it was previously negative)
  core.PodSpec{
```

```
controlplane ~ ✘ kc replace --force -f nginx.yaml
pod "nginx" deleted
pod/nginx replaced
```

```
apiVersion: v1
kind: Binding
metadata:
  name: nginx
target:
  apiVersion: v1
  kind: Node
  name: node02
```

Then we have to send a **POST** request to the Pod's binding API with the data defined in the Binding Object **YAML** file buut in **JSON** format. So we must convert the YAML file into the equivalent JSON.

To get the Cluster Nodes:

```
$ kubectl get nodes
```

4.2 Taints & Tolerations

[Official K8s Doc](#)

4.2.1 Introduction

In this lecture we will talk about the **Pod-Node** relationship and how you can restrict what Pods are placed on what Nodes.

To understand Taints & Tolerations easily, we are going to see a vegetables garden example. Imagine we have a vegetable garden, where we have vegetables and insects. If we do not want the flies to get on the plant, we spray the plant with a repellent spray (**a taint**) we know the fly is intolerant, so the **taint** would make the fly goes away. However, there are a lots of insects that are tolerant to this smell, and the taint does not really affect them, so they end up in our plant.

So in this example, there are two things that decide if an insect can get a plant:

- **Taint** used on a plant
- The insect **Tolerance** to that particular Taint

Taking back to K8s, the Nodes are the plants on the vegetables garden (K8s Cluster) and the Pods are the insects. Taints and Tolerations are used to set restrictions on what Pods can be scheduled on a Node.

So imagine the following situation, we have 3 Nodes: Node 1, Node 2 and Node 3. Also we have 4 Pods: A, B, C and D. So if nothing is specified, K8s will distribute the load for example:

- **Node 1:** A
- **Node 2:** B
- **Node 3:** C & D

But now imagine that **Node A** is so important to the stability of the Cluster, so we have to prevent Pods to run in the Node A, how? **Placing a taint on the Node A**, let's call it **blue**, so app=blue. **By**

default, Pods have no toleration which means unless specified otherwise, any Pod can tolerate any Taint. So this is the solution of our problem!!!

But you can imagine our next requirement, because humanity is insatiable, we would want that certain Pods can run on a Node and not others. That's the reason why we need **Tolerations**. Imagine that we want that in **Node 2** we want that Pods A and B runs but C and D not. So we have to apply a **Taint to Node 2**, for example tier=front-end, and then we have to add the **Toleration** tier=front-end to the Pods A and B and not to C and D.

As we said, the kube-scheduler works this way 1.5.1.

4.2.2 kubectl commands

To taint a Node

```
$ kubectl taint nodes <node-name> key=value:taint-effect
```

Taints effectas can be:

- **NoSchedule**: Pods that do not tolerate the taint will not be scheduled on the node.
- **PreferNoSchedule**: Kubernetes will try to avoid scheduling Pods that do not tolerate the taint on the node if there are other supported nodes.
- **NoExecute**: Pods that do not tolerate the taint will be evicted from the node if they are already running, and they will be rescheduled into another nodes if it is possible.

[Official Doc](#) | `kubectl taint --help`

EXAMPLE:

```
$ kubectl taint nodes node01 app=blue:NoSchedule
```

The **taint-effect** defines what would happen to the Pods if they do not tolerate the taint. There are three taint effects:

- **NoSchedule**: the Pods do not be scheduled on the Node, actually what we have been discussing
- **PreferNoSchedule**: the system will try to avoid placing a Pod without Toleration to that taint on the Node, but that is not guaranteed
- **NoExecute**: new Pods will not be scheduled on the Node, and existing Pods on the Node, if there are any, will be evicted if they not tolerate that Taint (killed and rescheduled on another Node)

To add Toleration to a Pod | [Official Doc](#)

Define in it YAML file the section **.spec.tolerations** which will be a list:

```

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
  tolerations:
    - effect: "NoSchedule" | "PreferNoSchedule" | "NoExecute"
      key: "key"
      operator: "Equal" | "Exists"
      value: "value"

```

WARNING

It is mandatory to encode all the values specified in tolerations in double codes ("")!

NOTE 1

Remember that **Taints & Tolerations** do not tell Pods to go to a particular Node, because a Pod has a Taint it does not mean that it should go to the Nodes with the Taint, it is not the purpose. Instead, it tells the Nodes to only accept Pods with certain tolerations.

If your requirement is to restrict a Pod to certain Nodes, it is provided by another concept called as **NodeAffinity**.

Operator could be:

- **Equal:** Pod accepts to deploy in nodes with specific Taint value specified.
- **Exists:** Pod accepts to deploy in nodes with that Taint specified.

4.2.3 Taint in Master Nodes

So far we have only been referring to the **Worker Nodes**, but we also have **Master Nodes** in the Cluster, which is technically just another Node that has all the capabilities of hosting a Pod plus it runs all the management software. However, **kube-scheduler never schedule Pods on the Master Nodes**. Why is that?

When the K8s Cluster is first set up, a **Taint** is set on the **Master Node / Master Nodes** automatically, that prevent any Pods from being scheduling on these Nodes. You can revert this configuration as required, but it is recommended to not do it, not deploy application workloads on a **Master Service**.

```
$ kubectl get node kubemaster | grep -i taint
```

4.3 Node Selector

[Official K8s Doc](#)

4.3.1 Introduction to Node Selectors

Let's start with a simple example. We have 3 Nodes in the Cluster, for which two are smaller Nodes with lower hardware resources, and one of them is a larger Node configured with higher resources. You have

also different kinds of workloads running in your Cluster, you would like to dedicate the data processing workloads that require higher horsepower to the larger Node. However, in the current default setup, any pods can go to any nodes indifferently.

So to solve this, we can set a limitation on the Pods, so that they only run on particular Nodes and there are two ways of doing this: **nodeSelector** and

4.3.2 Pod Configuration

nodeSelector is the simplest way, we only need to add at Pod definition the specification **.spec.nodeSelector**:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
  nodeSelector:
    size: large
```

But wait a minute, where did the size large come from? How does K8s know which is the largest node? Deep down you know the answer, it is a label assigned to a Node. The kube-scheduler uses these labels assigned to Nodes to match and identify the right Node to place the pods on.

4.3.3 Node Configuration

[Official K8s Documentation](#)

To configure labels in Nodes:

```
$ kubectl label node <node-name> <label-key>=<label-value>
```

To see labels in Nodes:

```
$ kubectl get node --show-labels
```

EXAMPLE: To configure labels in Nodes:

```
$ kubectl label nodes node01 size=large

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
  nodeSelector:
    size: large
```

4.4 Affinity and anti-Affinity

[Official K8s Doc](#)

4.4.1 Introduction to Node Affinity

Continuing with the decision of what Pods goes to what Nodes. What happens with complex configuration? For example, if we would decide to put a Pod on a large or medium node, or put a Pod in every Node that are not small. You cannot achieve this using Node Selectors, you have to use **Node Affinity**

The primary purpose of **Node Affinity** is to ensure that Pods are hosted on particular Nodes. It provides us with advances capabilities to limit Pod placement on specific Nodes. But great power comes great complexity. Looks what we have to define to do the same as 4.2.2: set than a Pod can only deploy in large Nodes:

```
pod-only-largeNodes.yaml

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: size
                operator: In
                values:
                  - large
```

Let's analize it a little bit closer, under .spec we have .spec.affinity, and then .spec.affinity.nodeAffinity under that. Then we have a property that seems like a sentence (it does no not need definition). The we have nodeSelectorTerms and that is an array where we have to specify if the specified next must be fullfilled or not. The last key-value pairs define the key, the operator because we want that key in Nodes and the value. Easy peasy.

If you want your pod be deployed on large and medium Pods, we only need to add the following value:

```

pod-large-mid-Nodes.yaml

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: size
                operator: In
                values:
                  - large
                  - medium

```

If you want your Pod only to not be deployed in small Nodes, we can define the Node Affinity:

```

pod-notSmallNodes.yaml

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
              - key: size
                operator: NotIn
                value:
                  - small

```

If we want that Pods only deploy in Nodes with size label defined:

```

pod-notSmallNodes.yaml

apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
          - matchExpressions:
            - key: size
              operator: Exists

```

There are a number of other operators as well, check de official documentation for specific details: [Official K8s Documentation](#). But the most used are:

- **In:** have the label and with the value.
- **NotIn:** have the label but with different value.
- **Exists:** just have the label.
- **DoesNotExist:** just not have the label.

4.4.2 Node Affinity Types

So when the Pods are created, these rules are considered and the Pods are placed onto the right Nodes, but what happens when **nodeAffinity** could not match a Node with a given expression? And what happens when the pods are running and the labels on a Node are changed? All these are specified in the **.spec.affinity.nodeAffinity....** This long sentence specifies what happens with the Affinity.

There are two states in the life-cycle of a Pod when considering **Node Affinity**:

- **DuringScheduling:** state when the Pod does not exist and it is created for the first time
- **DuringExecution:** state when the Pod has been running and change is made in the environment that affects **NodeAffinity** (such as a change in the label of a Node)

There are three options to determine the severity of the affinity:

- **Required:** it is absolutely necessary to have a Node with the specified labels, so if the label is not find, the Pod won't be deployed.
- **Preferred:** it is preferred to have a Node with the specified labels, and if the label is not found but kube-scheduler do not find available nodes with the label, the rule will be simply ignored.
- **Ignored:** pods will continue to run and any changes in Node Affinity will not impact them once they are scheduled.

The most commonly used are:

- requiredDuringSchedulingIgnoredDuringExecution

- preferredDuringSchedulingIgnoredDuringExecution
- requiredDuringSchedulingRequiredDuringExecution

	DuringScheduling	DuringExecution
Type 1	Required	Ignored
Type 2	Preferred	Ignored
Type 3	Required	Required

4.4.3 Inter-pod affinity and anti-affinity

Inter-pod affinity and anti-affinity allow you to constrain which nodes your Pods can be scheduled on based on the labels of **Pods** already running on that node, instead of the node labels.

Inter-pod affinity and anti-affinity rules take the form "this Pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more Pods that meet rule Y". You express these rules (Y) as label selectors with an optional associated list of namespaces. Pods are namespaced objects in Kubernetes, so Pod labels also implicitly have namespaces. Any label selectors for Pod labels should specify the namespaces in which Kubernetes should look for those labels.

You express the topology domain (X) using a `topologyKey`, which is the key for the node label that the system uses to denote the domain.

NOTE

Pod anti-affinity requires nodes to be consistently labeled, in other words, every node in the cluster must have an appropriate label matching `topologyKey`. If some or all nodes are missing the specified `topologyKey` label, it can lead to unintended behavior.

Types of inter-pod affinity and anti-affinity:

- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`

To use inter-pod affinity, use the `affinity.podAffinity` field in the Pod spec. For inter-pod anti-affinity, use the `affinity.podAntiAffinity` field in the Pod spec.

In the following example, we define one Pod affinity rule and one Pod anti-affinity rule. The Pod affinity rule uses the "hard" `requiredDuringSchedulingIgnoredDuringExecution`, while the anti-affinity rule uses the "soft" `preferredDuringSchedulingIgnoredDuringExecution`.

The affinity rule specifies that the scheduler is allowed to place the example Pod on a node only if it is labeled with `topology.kubernetes.io/zone` and there are Pods running with the label `security=S1`. As well it will try to avoid deploy pods on nodes with other pods running and label `security=S2`.

```

apiVersion: v1
kind: Pod
metadata:
  name: with-pod-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: security
                operator: In
                values:
                  - S1
            topologyKey: topology.kubernetes.io/zone
      podAntiAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 100
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                  - key: security
                    operator: In
                    values:
                      - S2
            topologyKey: topology.kubernetes.io/zone
  containers:
    - name: with-pod-affinity
      image: registry.k8s.io/pause:2.0

```

Kubernetes includes an optional `matchLabelKeys` or `mismatchLabelKeys` field for Pod affinity or anti-affinity. The field specifies keys for the labels that should match with the incoming Pod's labels, when satisfying the Pod (anti)affinity. It sets the existing pods that will be taken into Pod (anti)affinity calculation. And the oposite for `mismatchLabelKeys`, they will take into account all pods except the ones with the labels specified.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: application-server
...
spec:
  template:
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values:
                      - database
            topologyKey: topology.kubernetes.io/zone
            # Only Pods from a given rollout are taken into consideration when calculating pod affinity
            # If you update the Deployment, the replacement Pods follow their own affinity rules
            # (if there are any defined in the new Pod template)
            matchLabelKeys:
              - pod-template-hash

```

4.5 Resource Requirements & Limits

[Official K8s Documentation](#)

WARNING

Resources (required and limits) are **always** defined by **container**, **NOT BY POD**

4.5.1 Introduction

Let's imagine a Cluster with 3 Worker Nodes available, each Node has a set of CPU and memory resources available. Every Pod requires a set of resources to run, and when a Pod is running on a Node, it consumes the resources available on that Node. As we discuss before, it is the **kube-scheduler** that decides which Node a Pod goes to, taking into account the amount of resources required by a Pod and those available on the Nodes. It identifies the best Node to place a Pod on in function of the resting resources if the Pod runs on it (explained in section 1.5.1).

So if we have a Pod which needs 1 core CPU and 2Gi of Memory, without no Affinity Policies, Taints, Node Selector, etc. It will deploy randomly in one of the 3 nodes.

If we get another, it would be allocated in another Node, because for kube-scheduler it would be better, since it would have more capacity once the Pod is running on it.

It would be done until all Nodes capacity is full, or since a Pod cannot run in any Node, because they have no enough resources, so the kube-scheduler holds back scheduling the Pod, so you could see it in a Pending state. If you look at the events you would see there is an insufficient CPU / Memory.

4.5.2 Resources types

4.5.2.1 CPU

Limits and requests for CPU resources are measured in CPU units. In Kubernetes, 1 CPU unit is equivalent to 1 physical CPU core, or 1 virtual core, depending on whether the node is a physical host or a virtual machine running inside a physical machine.

- **Floats:** 1.0, 0.5, 1.75, 0.005
- **Millicpu (in double quotes):** "1000m", 500m, 1750m, 5m

4.5.2.2 Memory

Memory refers to RAM. It can be defined as:

- **Plain Integers:**
- **Power-of-ten equivalents (with double quote):**
 - 1m=1e-3
 - 1.5k=1.5e3
 - 0.75M=0.75e6
 - 1G=1e9
- **Power-of-two equivalents (with double quote):**
 - 1Ki= 1×2^{10} (1 Kibibyte)
 - 1.5Mi= 1.5×2^{20} (1.5 Mebibytes)
 - 2.7Gi= 2.7×2^{30} (2.7 Gibibytes)
 - 0.8Ti= 0.8×2^{40} (0.8 Tebibytes)

NOTE

Just to remember, and it applies to all measures:

- Kilobyte = 10^3
- Megabyte = 10^6
- Kibibyte = 2^{10}
- Mebibyte = 2^{20}

Múltiplos de bytes			
Sistema Internacional (decimal)		ISO/IEC 80000-13 (binario)	
Múltiplo (símbolo)	SI	Múltiplo (símbolo)	ISO/IEC
kilobyte (kB)	10^3	kibibyte (KiB)	2^{10}
megabyte (MB)	10^6	mebibyte (MiB)	2^{20}
gigabyte (GB)	10^9	gibibyte (GiB)	2^{30}
terabyte (TB)	10^{12}	tebibyte (TiB)	2^{40}
petabyte (PB)	10^{15}	pebibyte (PiB)	2^{50}
exabyte (EB)	10^{18}	exbibyte (EiB)	2^{60}
zettabyte (ZB)	10^{21}	zebibyte (ZiB)	2^{70}
yottabyte (YB)	10^{24}	yobibyte (YiB)	2^{80}

Véase también: [nibble](#) • [byte](#) • [sistema octal](#)

4.5.3 Requests & Limits

WARNING

Resources (required and limits) are **always** defined by **container**, NOT BY POD. So each container in a Pod have its resources definition (requests and limits).

4.5.3.1 Requests

The amount of CPU and Memory **required** for a **Container** to run, so the kube-scheduler will never allocate a Pod in a Node with less resources than the **sum of required resources of all the containers in Pod**.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
      resources:
        requests:
          cpu: 1.0
          memory: "2Gi"
```

4.5.3.2 Limit

Says a Pod are allocated into a Node based on the Required Resources. The Pod container starts consuming 1 CPU on a Node, actually it can go up and consume as much resources as it requires and that suffocates the native processes on the Node or other containers of resources.

That's the reason we can set a limit for the resource usage on Pods. The limit has different behaviour depending of wheter is CPU or Memory:

- **CPU:** The CPU limit cannot be exceeded, so if the container needs more CPU, it wouldn't have it, it would work slower.
- **Memory:** A container can use more memory resources than its limit, but if it is detected the Pod will be automatically **Terminated** with an **OOM (Out of Memory)**. This is because once memory is used by a Pod, the only way to release it is killing the Pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  containers:
    - image: nginx:latest
      name: nginx-container
      resources:
        requests:
          cpu: 1.0
          memory: "2Gi"
        limits:
          cpu: "1750m"
          memory: "2.75Gi"
```

NOTE

We can define only limits or only requests in the resources of a container. But you would have to assume the consequences, but K8s allow it.

4.5.4 Ideal Scenario for CPU

So far it seems completely good for our environment, but don not you think it can be so inefficient sometimes? Imagine you have a Node with 2 Pods running:

1. **Pod 1:** request 1CPU, limit 3CPU
2. **Pod 1:** request 2CPU, limit 4CPU

So at one time, Pod 1 are consuming 3CPU (because it cannot consume more), but it needs more CPU at this moment to work properly. On the other hand, Pod 2 are consuming only 0.5CPU, because it does not need more resources at that moment. Is not this an inefficiency?

The answer is yes, it is, that's why the **Ideal Scenario** is:

- Define requests
- Do not define limits

This way any Pod can consume as many CPU cycle as available. So it is the most ideal setup, of course there are cases where you absolutely may want to limit a Pod of resources and in that case you have to set limits. For example the labs used for practise this course. All the labs are hosted as containers on a K8s Cluster, and of course, since it were made accessible to public, the containers have limits, to prevent users misusing the infrastructure to for example Bitcoin mining or other resource consuming activities.

Another advice is to set requests always, because if not, Pods can be allocated in Nodes without enough resources to run it, and if the containers have no limits it can be chaos.

4.5.5 Default Configuration

By default K8s does not have a CPU or a memory request or limit set, so this means that any Pod can consume as much resources as required on any Node, suffocating other Pods or processes that are running on the Node.

4.5.6 Changing Default Configuration, LimitRange Object

[Official K8s Doc](#)

How can we ensure that every Pod has some default set? This is possible with **LimitRange** Object. By default, containers run with unbounded compute resources on a Kubernetes cluster. Using Kubernetes resource quotas, administrators (also termed cluster operators) can restrict consumption and creation of cluster resources (such as CPU time, memory, and persistent storage) within a specified namespace. Within a namespace, a Pod can consume as much CPU and memory as is allowed by the ResourceQuotas that apply to that namespace. As a cluster operator, or as a namespace-level administrator, you might also be concerned about making sure that a single object cannot monopolize all available resources within a namespace..

```
apiVersion: v1
kind: LimitRange
metadata:
  name cpu-resources-constraint
spec:
  limits:
    - default:
        cpu: "500m"
      defaultRequest:
        cpu: "500m"
      max:
        cpu: 1
      min:
        cpu: "100m"
    type: Container
```

- **default:** defines default limit (if not defined)
- **defaultRequest:** defines default request (if not defined)
- **max:** refers to the maximum CPU that can be defined as request
- **minimum:** refers to the minimum CPU that can be defined as request

For memory it is the same:

```

apiVersion: v1
kind: LimitRange
metadata:
  name cpu-resources-constraint
spec:
  limits:
    - default:
        memory: "500m"
      defaultRequest:
        memory: "500m"
    max:
      memory: 1
    min:
      memory: "100m"
  type: Container

```

WARNING 1

If you define limits on the container definition LimitRange object will be overwritten by this limits.

WARNING 2

This Objects only affects to newer Pods that are created, so does not affect existing Pods

4.5.7 Restrict the total amount of resources

Is there any way to restrict the total amount of resources that can be consumed by applications in a K8s Cluster? The answer is not by cluster, but yes by Namespace, with **ResourceQuota** Objects, like we discuss in section 2.6.6.

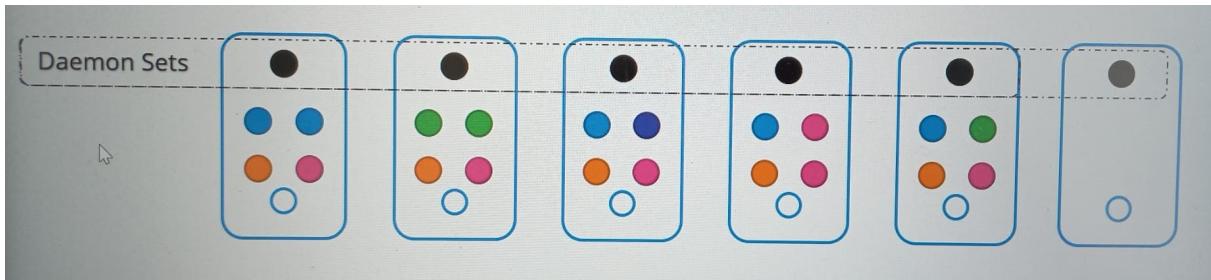
[Official K8s Doc](#)

4.6 DaemonSets

[Official K8s Doc](#)

4.6.1 Introduction to DaemonSets

So far we have deployed different replicas of Pods on different Worker Nodes on Cluster. with the help of ReplicaSet and Deployments we make sure multiple replicas of an instance of Pod are made available across different Worker Nodes. **DaemonSets** are like ReplicaSets as it helps you deploy multiple instances of Pods, but it runs one copy of your Pod on each Worker Node in your cluster. Whenever a new Node is added to the Cluster, DaemonSet add a replica of the Pod to that Node, and when a Node is removed the Pod is automatically removed.



DaemonSet ensures that one copy of a Pod is always present in all Nodes in the Cluster. So for what can it be used? Say you would like to deploy a monitoring agent or a log collector on each of your Nodes in the Cluster. DaemonSet could deploy your monitoring or log agent in each Node of the Cluster.

Another good example of use of DaemonSet is the **kube-proxy** component, it must be run on each Worker Node on the Cluster, so it is configured as a DaemonSet.

4.6.2 Definition of DaemonSets

It is very similar to ReplicaSet definition:

```
daemonSet-definition.yaml

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
spec:
  selector:
    matchLabels:
      app: monitoring-agent
  template:
    metadata:
      name: monitoring-agent
      labels:
        app: monitoring-agent
    spec:
      containers:
        - image: monitoring-agent
          name: monitoring-agent
```

4.6.3 Creation and view of DaemonSets

Creation

```
$ kubectl create -f daemonset-definition.yaml
```

View

```
$ kubectl get ds
```

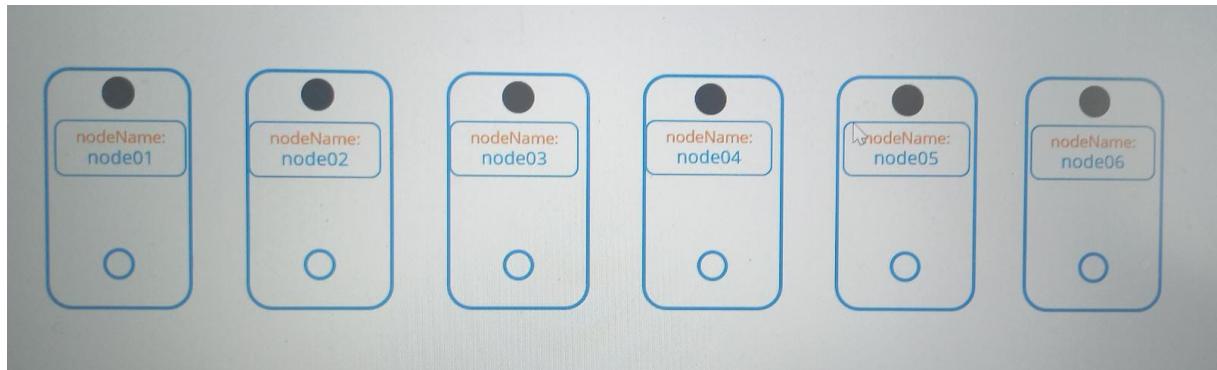
WARNING

This command does not work with daemonSets, I don't know why, it only accepts certain objects

```
$ kubectl create daemonset --help
```

4.6.4 How do DaemonSets work?

Before v1.12, it worked adding the **NodeSelector** of each Node of the Cluster to every Pod created by DaemonSet.



From K8s v1.12 onwards, the DaemonSet uses the default scheduler and **NodeAffinity** rules that we have learned to schedule Pods on Nodes.

4.6.5 Example of DaemonSet - Fluentd

To build one single log structure in K8s for a couple of apps, or even a dozen, you cannot use basic logging structure, because unfortunately, basic logging doesn't answer these questions. Data is temporary; any crash or rerun will erase all the previous records.

To address these problems, you need to set it to the node-logging level. Since we can interact with every single object within the node, we can try to make a vertical unique logging system.

Furthermore, you can implement cluster-level logging by including a node-level logging agent on each node. A logging agent is a dedicated tool that exposes or pushes logs to a backend. Usually, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.

Because the logging agent must run on every node, it's common to implement it as either a DaemonSet replica: **Logs Collector: Fluentd**. Fluentd is an ideal solution as a unified logging layer. You just have to open and download the type of logger you need for the project, it collects logs both from user applications and cluster components such as kube-apiserver and kube-scheduler. So... It needs to run as **DaemonSet**.

WARNING

Fluentd is just the log retriever, it won't expose anything, so `curl pod_ip` won't work.

Example of fluentd daemonset, from [link](#):

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
  version: v1
  kubernetes.io/cluster-service: "true"
spec:
  selector:
  matchLabels:
    k8s-app: fluentd-logging
  template:
    metadata:
      labels:
        k8s-app: fluentd-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      serviceAccount: fluentd # serviceAccountName accepted as well
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd
        image: fluent/fluentd-kubernetes-daemonset:v1.3-debian-elasticsearch
        env:
          - name: FLUENT_ES_HOST
            value: "elasticsearch.logging"
          - name: FLUENT_ES_PORT
            value: "9200"
          - name: FLUENT_ES_SCHEME
            value: "http"
          - name: FLUENT_UID
            value: "0"
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
      volumeMounts:
      - name: varlog
        mountPath: /var/log
      - name: varlibdockercontainers
        mountPath: /var/lib/docker/containers
        readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers

```

4.7 Static Pods

In a really weird scenario, where we are supposing there are no:

- kube-apiserver
- kube-scheduler
- etcd

So the kubelet is alone on the sea, it is able to create pods by its own in a very specific way, this pods are called **Static Pods**. You won't be able to create Deployments, ReplicaSets or other objects. You only will be able to create Pods this way, because the other objects needs some control-plane objects like replication-controller.

If the connection to the rest of the K8s come back, the kubelet will create a mirror of all the **Static Pods** on the kube-apiserver, so if you do `kubectl get po`: you will see as well the **Static Pods**. But you won't be able to edit or delete them using `kubectl`, you must modify or delete them directly from the static file inside kubelet configuration.

`kubeadm` tool makes use of **Static Pods** to deploy the control-plane components as Pods on a Node. It starts by installing `kubelet` on all the master Nodes, then create **Static Pod** definition files for the various control-plane components, such as: kube-apiserver, kube-controller, etcd, etc. Then, it places the definition files inside the designated manifest folder and the the kubelets automatically takes care of deploying them as **Static Pods** on the Cluster. This way it avoids to download binaries, configure and run services or worry about the services crashing, because it will be restarted by the kubelet.

The most common place to store the manifest of these static pods and the one used by `kubeadm` is: `/etc/kubernetes/manifests`. But it could be any directory on the Node, you just need to configure it properly on the `kubelet/config.yaml`, which use to be on: `/var/lib/kubelet/config.yaml`

NOTE

If you are not sure of where is this path, you can do:

1. Check kubelet is running on the node:

```
$ systemctl status kubelet
```

2. Check service configuration:

```
$ ps aux | grep kubelet
```

3. Check the configuration and you will find the `kubelet/config.yaml`

```
controlplane ~ ✘ ps aux | grep kubelet
root      4166  0.0  0.1 1481148 260028 ?        Ssl  11:23  0:58 kube-apiserver --advertise-address=192.22.165.6 --allow-privileged=true --authorization-mode=Node,RBAC --client-ca-file=/etc/kubernetes/pki/ca.crt --enable-admission-plugins=NodeRestriction --enable-bootstrap-token-auth=true --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt --etcd-certfile=/etc/kubernetes/pki/etcd/cert.pem --etcd-keyfile=/etc/kubernetes/pki/etcd/client.key --etcd-servers=https://127.0.0.1:2379 --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key --requestheader-allowed-names=front-proxy-client --requestheader-client-ca-file=/etc/kubernetes/pki/front-proxy-ca.crt --requestheader-extra-headers-prefix=X-Remote-Extra --requestheader-group-headers=X-Remote-Group --requestheader-username-headers=X-Remote-User --secure-port=6443 --service-account-issuer=https://Kubernetes.default.svc.cluster.local --service-account-key-file=/etc/kubernetes/pki/sa.pub --service-account-signing-key-file=/etc/kubernetes/pki/sa.key --service-cluster-ip-range=10.96.0.0/12 --tls-cert-file=/etc/kubernetes/pki/apiserver.crt --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
root      4727  0.0  0.0 4433172 183804 ?        Ssl  11:23  0:24 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf --config=/var/lib/kubelet/config.yaml --container-runtime=unix:///var/run/containerd/containerd.sock --pod-infra-container-image=registry.k8s.io/pause:3.9
root      15531  0.0  0.0   6928  2272 pts/3    S+   11:43  0:00 grep -color=auto kubelet
```

So you can check inside the config file where the kubelet reads for the manifests:

```
$ cat /var/lib/kubelet/config.yaml | grep -i static
```

So when you do `kubectl get pods -n kube-system` you will see all the control-plane services running as pods.

NOTE

If you pay attention, is that the reason why if you remove some of the control-plane pods, it is automatically recreated but it does not have any deployment, statefulset, daemonset or replicaset controlling them.

Well, take care of **kube-proxy**, because it is the exception, it is a **daemonSet**.

4.8 Multiple Schedulers

[Official K8s Doc](#)

4.8.1 Introduction

So far, we have seen how one single scheduler works in a K8s Cluster, it has an algorithm that distribute Pods across Nodes, taking into account Taint & Tolerations, NodeSelector, NodeAffinity, Resources, etc. But what if you need more? If you have an application that requires some additional checks? **K8s is highly extensible**, you can **write your own K8s scheduler program**, package it and deploy it as the default scheduler or as an additional scheduler in the K8s Cluster. That way some applications can use the default K8s scheduler and others can use your custom schedulers (you can configure more than one). A K8s Cluster accepts multiple schedulers at a time.

4.8.2 Defining schedulers

When creating a Pod (or a Pod Object Manager). You can instruct K8s to have the Pod scheduled by a specific scheduler, so the different schedulers must have different names, in order we can identify them as separate schedulers. The default scheduler is named default scheduler XD, and this name is configured in the **kube-scheduler** configuration file that looks like this:

```
scheduler-config.yaml
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
- schedulerName: default-scheduler
leaderElection:
  leaderElect: true
  resourceNamespace: kube-system
  resourceName: lock-object-my-scheduler
```

For other schedulers, you can create a similar file, changing the **schedulerName** value.

The **leaderElection** is configured because if there are multiple copies of an scheduler running on different Nodes only one can be active at a time. So this option helps using a leader who will lead the scheduling activities

4.8.3 Creating new schedulers

There are two ways to create new schedulers:

4.8.3.1 Bad way: as a service in the system

To create a new scheduler you must use the same **kube-scheduler** binary mentioned in Section 1.5.2 but setting the configuration with the YAML file that we have created... You can imagine it is terrible.

4.8.3.2 Good way: as a Pod in K8s kube-system Namespace

```
my-custom-scheduler.yaml

apiVersion: v1
kind: Pod
metadata:
  name: my-custom-scheduler
  namespace: kube-system
spec:
  containers:
    - command:
      - kube-scheduler
      - --address=127.0.0.1
      - --kubeconfig=/etc/kubernetes/scheduler.conf
      - --config=/path/to/my-scheduler.yaml
      image: k8s.gcr.io/kube-scheduler-amd64:v1.11
      name: kube-scheduler
```

4.8.4 Configuring Pods to consume a custom scheduler

```
pod-consuming-custom-sch.yaml

apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - image: nginx
      name: nginx
  schedulerName: my-custom-scheduler
```

4.8.5 Monitor the correct functioning of a custom scheduler

```
$ kubectl get events -o wide
```

```
$ kubectl logs my-custom-scheduler -n kube-system
```

4.9 Scheduler Profiles

Scheduling a Pod is not as easy as it seems. Default **kube-scheduler** follow the process:

- Put the Pod to create into the **scheduling queue**

- On this queue, Pods are sorted based on the **priority** defined directly on the Pod definition (if there is).

NOTE

You can check on the official documentation how to create **PriorityClass** and assign it to a Pod.

- Once the sorted phase, the Pod enter into the **Filtering Phase**, where Nodes that cannot run the pod because of taints, nodeAffinity, not enough resources, etc. are filtered out.
- **Scoring Phase:** not filtered nodes are scored based on their free resources and presence of the image locally.
- **Binding Phase:** the pod is finally bound to the node with the highest score.

All these actions are achieved with certain plugins:

- PrioritySort
- NodeResourceFit
- NodeName
- NodeUnschedulable
- NodeResourcesFit
- ImageLocality

So you can define different scheduler profiles to disable and enable different pluggins:

```
apiVersion: kubescheduler.config.k8s.io/v1
kind: KubeSchedulerConfiguration
profiles:
  - plugins:
      score:
        disabled:
          - name: PodTopologySpread
        enabled:
          - name: MyCustomPluginA
            weight: 2
          - name: MyCustomPluginB
            weight: 1
```

For more information, check the [Official K8s Documentation](#)

5 Logging and Monitoring

In this section we will see how to monitor K8s components as well as the applications hosted in the Cluster. How to view and manage the logs of the Cluster components as well as the application logs.

5.1 Monitoring Cluster Components

How do we monitor resource consumtion in K8s, or more important, what would you like to monitor? I would like to know node-level metrics, such as:

- The number of nodes in the cluster
- How many of them are healthy
- Performance Node metrics like CPU, Memory, Network Disk Utilization, etc.

Such as Pod level metrics:

- Number of replicas of a Pod
- The metric consumtion this POds and the state

So we need a solution that monitor all these metrics, store them and provides analytics around this data. Unfortunately, K8s does not come with a full-featured built-in monitoring solution. However, there are a number of open-source solutions available such as:

- Prometheus
- Elastic Stack
- Metrics Server
- Datadog (private)
- Dynatrace (private)

Any of the solutions retrieves data of every K8s Node and Pod. All this data are stored in a memory. But to do that, K8s must pass this data in anyway. **Kubelet** has a subcomponent called **Container Advisor (CAdvisor)**, this subcomponent is responsible for retreiving performance metrics from pods and exposing them through the kubelet API to make the metrics available for the metrics solution.

5.1.1 Basic K8s Metrics Server

If you are using Minikube:

```
$ minikube addons enable metrics-server
```

If you use another software, you can deploy the Metrics Servers from cloning the file from git:

```
$ git clone https://github.com/kubernetes-incubator/metrics-server.git  
$ cd metrics-server
```

```
$ kubectl create -f .
```

It deploys some Pods, Services, Roles, etc. To enable the Metrics Server.

5.1.2 kubectl top

kubectl top node

```
$ kubectl top node
```

This command provides the CPU and Memory consumption of each Node in your K8s Cluster.

kubectl top pod

```
$ kubectl top pod
```

This command provides the performance metrics of Pods in a K8s Cluster.

NOTE

This values are real, they are not related to request or limits, they are the real consumption of the Pod when running with all its containers.

5.2 Application Logs

5.2.1 Logs in docker

```
$ docker logs <container_name_or_id> [-f]
```

5.2.2 Logs in K8s

```
$ kubectl logs <pod_name> [-c <container_name>] [-f]
```

5.2.3 Logs in K8s

```
$ kubectl logs <pod_name> --all-containers
```

5.3 Logs with K8s-Ready EFK

5.3.1 Kubernetes Logging Structure

There are three different levels for logging in Kubernetes: basic I/O logging, node-level logging, and cluster-level logging.

First, we have the basic I/O logic, k8s support these basic data streams and retrieve them as logs using `kubectl logs <pod_name>`, where the data is collected and you can easily explore it.

Data from each pod transacts to a single logs storage file (a JSON file) that allows us to work with reruns and sudden deaths but tends to reach max capacity too quickly. Node-level logging only supports `stdout/stderr` types, and you still need to have separate logging settings for each node.

Note

In Kubernetes (K8s), logrotate is a utility used to manage and rotate log files to prevent them from consuming excessive disk space. Kubernetes generates a lot of logs, especially from containers, and without proper log management, these logs can fill up the disk, causing issues like node instability or service crashes.

Container logs are stored as files in JSON format under `/var/log/containers/` or `/var/lib/docker/containers/` (for Docker runtime).

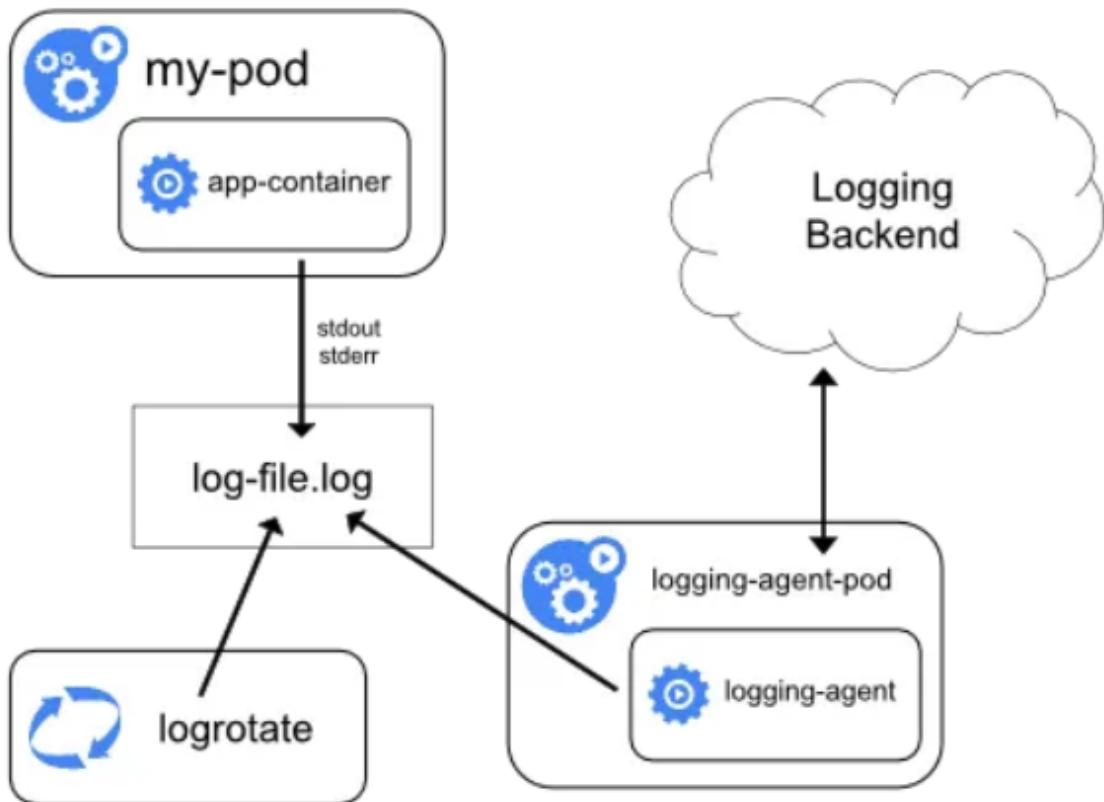
In Kubernetes, log rotation is usually configured on the host machine (node) where containers run on `/etc/logrotate.d/` or `/etc/logrotate.conf`

5.3.2 Necessity of another solution

But what if you need to build one single log structure for a couple, dozen or hundreds of apps? And what if a pod with an application crashes? Unfortunately, basic K8s logging doesn't answer these questions. Data is temporary; any crash or rerun will erase all the previous records, making impossible to have a resilient logging structure.

To address these problems, you need to set it to the **node-logging level**. Since we can interact with every single object within the node, we can try to make a vertical unique logging system.

Furthermore, you can implement cluster-level logging by including a node-level logging agent on each node. A logging agent is a dedicated tool that exposes or pushes logs to a backend. Usually, the logging agent is a container that has access to a directory with log files from all of the application containers on that node.



As the logging agent must run on every node, it's common to implement it as either a DaemonSet replica, a manifest pod, or a dedicated native process on the node.

5.3.3 EFK Solution

Now that we covered the basics of logging, let's explore EFK solution (ElasticSearch, Fluentd and Kibana). This solution is used by many important companies like: GitHub, Netflix, and Amazon, ITnow XD. It connects Fluentd, ES, and Kibana (as visualization tools) to make an exact namespace with a few services and pods.

5.3.4 Fluentd (Logs Collector)

Fluentd is an ideal solution as a unified logging layer. You just have to open and download the type of logger you need for the project. Fluentd has ready configurations and templates (snippets) to provide a simple setup and customization process when integrating Fluentd with other systems (forwarding the logs in the proper way), such as ElasticSearch (ES), Apache Kafka or Dynatrace.

Fluentd collects logs both from user applications and cluster components such as kube-apiserver and kube-scheduler. The main advantage of this approach is that data isn't stored in the JSON file, so it is saved with no exclusions, as well it has timestamps to be able to sort it properly.

There are two different concrete stacks for logging: ELK and EFK. The first is with Elastic domain product, Logstash. However, this tool, which has a lightning connection with ES, doesn't support k8s directly. This is where the Fluentd open-source project comes in handy.

5.3.5 ElasticSearch + Kibana (Logging Endpoint)

ElasticSearch (ES) can perform many tasks, all of them centered around searching. ES, developed and provided by Elastic company, is a rapid-fire queryset executor that has impressive data processing and transferring capabilities.

We have chosen Elasticsearch to store, process and extend the logging, monitoring and alerting capabilities. But here I let other available and as well good options:

- Prometheus
- Apache Kafka
- Dynatrace

5.3.6 EFK K8s Resources to Deploy

5.3.6.1 EFK: Elasticsearch

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: elasticsearch
spec:
  selector:
    matchLabels:
      component: elasticsearch
  template:
    metadata:
      labels:
        component: elasticsearch
  spec:
    containers:
      - name: elasticsearch
        image: docker.elastic.co/elasticsearch/elasticsearch:7.4.1
        env:
          - name: discovery.type
            value: single-node
        ports:
          - containerPort: 9200
            name: http
            protocol: TCP
        resources:
          limits:
            cpu: 500m
            memory: 4Gi
          requests:
            cpu: 500m
            memory: 4Gi
```

```

apiVersion: v1
kind: Service
metadata:
  name: elasticsearch
  labels:
    service: elasticsearch
spec:
  type: NodePort
  selector:
    component: elasticsearch
  ports:
    - port: 9200
      targetPort: 9200

```

5.3.6.2 EFK: Fluentd

In order to **Fluentd** is able to retrieve logs from all the K8s Cluster, we need to create some **Role-based access control** resources.

ServiceAccount: Fluentd runs as a pod in the Kubernetes cluster and requires a ServiceAccount to authenticate itself when accessing the Kubernetes API to retrieve the necessary log data. So it associates the Pod/fluentd with the specified permissions.

ClusterRole: defines what resources and actions Fluentd is allowed to interact with across the entire Kubernetes cluster: Get, List, and Watch Pods and Namespaces.

ClusterRoleBinding: links the ClusterRole (which defines the permissions) to the ServiceAccount (which identifies the Fluentd pod).

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: fluentd
  namespace: kube-system

```

```

apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: fluentd
  namespace: kube-system
rules:
- apiGroups:
  - ""
  resources:
  - pods
  - namespaces
  verbs:
  - get
  - list
  - watch

```

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: fluentd
roleRef:
  kind: ClusterRole
  name: fluentd
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: fluentd
  namespace: kube-system
```

```

Fluentd_daemonSet.yaml

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
  version: v1
  kubernetes.io/cluster-service: "true"
spec:
  selector:
  matchLabels:
    k8s-app: fluentd-logging
  template:
    metadata:
      labels:
        k8s-app: fluentd-logging
        version: v1
        kubernetes.io/cluster-service: "true"
    spec:
      serviceAccount: fluentd
      serviceAccountName: fluentd
      tolerations:
        - key: node-role.kubernetes.io/master
          effect: NoSchedule
      containers:
        - name: fluentd
          image: fluent/fluentd-kubernetes-daemonset:v1.3-debian-elasticsearch
          env:
            - name: FLUENT_ES_HOST
              value: "elasticsearch.logging"
            - name: FLUENT_ES_PORT
              value: "9200"
            - name: FLUENT_ES_SCHEME
              value: "http"
            - name: FLUENT_UID
              value: "0"
          resources:
            limits:
              memory: 200Mi
            requests:
              cpu: 100m
              memory: 200Mi
          volumeMounts:
            - name: varlog
              mountPath: /var/log
            - name: varlibdockercontainers
              mountPath: /var/lib/docker/containers
              readOnly: true
      terminationGracePeriodSeconds: 30
  volumes:
    - name: varlog
      hostPath:
        path: /var/log
    - name: varlibdockercontainers
      hostPath:
        path: /var/lib/docker/containers

```

5.3.7 EFK: Kibana

Finally, we will use Kibana to make a visual representation of the logs. Create a kibana.yml file with the following lines:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kibana
spec:
  selector:
    matchLabels:
      run: kibana
    template:
      metadata:
        labels:
          run: kibana
  spec:
    containers:
      - name: kibana
        image: docker.elastic.co/kibana/kibana:7.4.1
        env:
          - name: ELASTICSEARCH_URL
            value: http://elasticsearch:9200
          - name: XPACK_SECURITY_ENABLED
            value: "true"
        ports:
          - containerPort: 5601
            name: http
            protocol: TCP
```

```
apiVersion: v1
kind: Service
metadata:
  name: kibana
  labels:
    service: kibana
spec:
  type: NodePort
  selector:
    run: kibana
  ports:
    - port: 5601
  targetPort: 5601
```

6 Application Lifecycle Management

6.1 Rolling Updates & Rollback

Before we look at how we upgrade our application, let's try to understand **rollouts** and **versioning** in a deployment. When you first create a deployment, it triggers a rollout. A new rollout creates a new deployment revision. In the future, when the app is upgraded, the container version is upgraded to a new one, a new rollout is triggered and a new deployment revision is created. This helps us keep track of the changes made to our deployment and enables us to rollback to a previous version of the deployment if necessary.

See status of a rollout:

```
$ kubectl rollout status <deploy_name>
```

See revisions and history of a rollout:

```
$ kubectl rollout history <deploy_name>
```

6.2 Deployment Strategies

6.2.1 Recreate Strategy

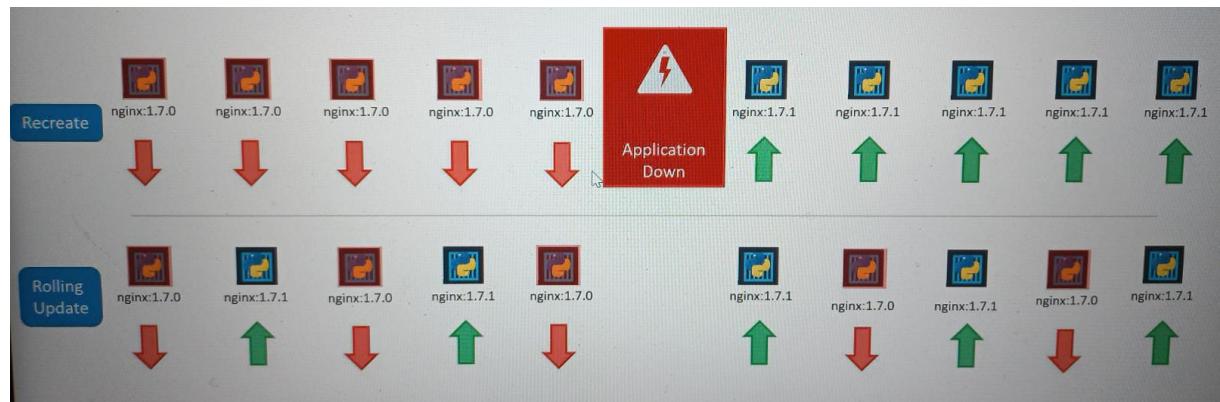
Destroy all replicas and then create new ones with the new version of the applications. So at first delete 5 instances and then deploy 5 new ones. The problem with this, as you can imagine, is that during the period after the older versions are down and before any newer version is up, the application is down and inaccessible to users

6.2.2 Rolling Update

We do not destroy all of them at once, instead we take down the older version and bring up a newer version one by one. This way, the application never goes down and the upgrade is seamless.

NOTE

If you not specify the strategy when you create the Deployment, it will assume it to be **rolling update**, the **default Deployment strategy**.



```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
      maxSurge: 1
      #maxUnavailable: 25%
      #maxSurge: 25%

```

NOTE

Kubernetes doesn't count terminating Pods when calculating the number of availableReplicas, which must be between replicas - maxUnavailable and replicas + maxSurge.

6.3 How to update a Deployment?

When we talk about update it can be of so many things: container version, app version, labels, number of replicas, etc. Since we already have a Deployment definition YAML file, it is easy for us to modify this file. Once we make the necessary changes, we can use:

```
$ kubectl apply -f <deployment-file>.yaml
```

So a new rollout is triggered and a new revision of the deployment is created.

WARNING

If you use `kubectl delete` and then `kubectl create` you are deleting all the replicas so the application will be down at this time and inaccessible to users.

But there is another way to do the same thing:

```
$ kubectl set image deploy <deploy-name> <container-name>=<image-name>:<tag>
```

```
$ kubectl edit deploy <deploy-name>
```

WARNING

But remember, doing this will result in the Deployment definition YAML file having a different configuration, so you must be careful if you use the same definition file in the future.

To see the strategy of a Deployment:

```
$ kubectl describe deploy <deploy-name> | grep -i StrategyType
```

6.4 Deployment updates under the hoods

When a deployment is created, it creates a new ReplicaSet to create and control the Pods, so if an update is made in the Deployment, it creates a new ReplicaSet and starts deploying the Pods and containers at the same time it is taking down the Pods in the old ReplicaSet following a rolling update strategy.

6.5 Rollback

```
$ kubectl rollout undo deploy <deploy-name>
```

6.6 Commands & Arguments in Pod definition

6.6.1 Introduction - Docker CMD & ENTRYPOINT

CMD

If you want to run a container from an Ubuntu image, when you run the command `docker run ubuntu` it runs an instance of ubuntu image but immediately dies, if you list the running containers you wouldn't see it running. If you list all the containers you will see that the container is in exited state. But why is that?

Unlike virtual machines, containers are not meant to host an OS (Operating System), **containers are meant to run a specific task or process**, such as to host an instance of a Web Server, Application Server, Database or simply carry out some computation or analysis. Once the task is complete, the container exits, **so the container only lives as long as the process inside it is alive**, if the web service inside the container is stopped, or crashes, the container exits. So who defines what process is run within the container? If you look inside popular Docker images like NGINX, you will see that it is specified in its Dockerfile in CMD instruction. So if we look in Ubuntu image, it uses bash as the CMD, and bash is not a real process.

So to specify the command to run inside the container you can do it:

- Defining CMD in Dockerfile:

Dockerfile

```
CMD ["init-exec.sh"]
```

- Overwriting the CMD via terminal:

```
$ docker run image init-exec.sh
```

ENTRYPOINT

The ENTRYPOINT instruction is like CMD instruction, as in, you can specify the program that will be run when the container starts, and whatever you specify on the CMD or un running commands will be appended to ENTRYPOINT. The default ENTRYPOINT is /bin/bash -c.

NOTE

It is not so recommended, but it is possible to overwrite the ENTRYPOINT of an image via commands:

```
$ docker run --entrypoint <entrypoint> <image>:<tag>
```

6.6.2 Commands & Arguments in a K8s Pod

The CMD and ENTRYPOINT can be overwritten by K8s in the Pod definition using `args` and `command`:

- `args`: overwrite CMD
- `command`: overwrite ENTRYPOINT

```
...
spec:
  containers:
    - image: ubuntu
      name: ubuntu-container
      command: ["sleep"]
      args:  ["10"]
...
```

NOTE

Command and args can be defined as well:

```
spec
  ...
  containers:
    - image: ...
      command:
        - "sleep"
        - "1200"
      args:
        - "c"
        - "while true; do echo hello; done"
```

6.7 Env Vars in Pods

6.7.1 Env Vars in Docker Containers

```
$ docker run -e APP_COLOR=pink nginx
```

6.7.2 Env Vars in Pods Containers with key-values directly

```
spec:  
  containers:  
    - image: ...  
      env:  
        - name: APP_COLOR  
          value: pink
```

6.7.3 Env Vars in Containers using ConfigMaps

```
spec:  
  containers:  
    - image: ...  
      env:  
        - name: APP_COLOR  
          valueFrom:  
            configMapKeyRef:  
              name: <cm_name>      # The ConfigMap this value comes from.  
              key: <key>           # The key to fetch.
```

```
apiVersion: v1  
kind: Pod  
spec:  
  containers:  
    - image: ...  
      envFrom:  
        - configMapRef:  
          name: <cm_name>      # The ConfigMap this value comes from.
```

We are going to explain that in more detail in following sections.

6.7.4 Env Vars in Pods Containers using Secrets

```
spec:  
  containers:  
    - image: ...  
      env:  
        - name: APP_COLOR  
          valueFrom:  
            secretKeyRef:
```

We are going to explain that in more detail in following sections.

6.8 ConfigMaps

The way to separate environment variables configuration from Pods definitions to ConfigMaps. They are used to pass configuration data in the form of key value pairs in K8s but out of Pod definition. When the Pod is created we should use the configMapRef to create the env vars inside the container into the Pod. So there are two phases involved in configuring ConfigMaps:

1. ConfigMap Creation
2. ConfigMap injection into Pods

6.8.1 ConfigMaps Creation

Imperative Creation

```
$ kubectl create cm --help
```

```
$ kubectl create cm <name> --from-literal=APP_COLOR=blue \
--from-literal=APP_LETER=mayus \
--from-literal=...
```

```
$ kubectl create cm <name> --from-file=/path/to/file
```

file.txt

```
APP_COLOR=blue
APP_LETER=mayus
...
```

Declarative Creation

```
cm.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_COLOR: blue
  APP_LETER: mayus
```

```
$ kubectl create -f cm.yaml
```

6.8.2 ConfigMaps Pod Injection

NOTE

It is common to have so many ConfigMaps for different Pods (front-end, backend, Database, etc.) for different purposes and configurations, so it is important to name them properly as you will be using these names later while associating it with Pods.

As Single Env Var

```
apiVersion: v1
kind: Pod
spec:
  containers:
    - image: ...
      env:
        - name: APP_COLOR
          valueFrom:
            configMapKeyRef:
              name: <cm_name>      # The ConfigMap this value comes from.
              key: <key>           # The key to fetch.
```

All variables from ConfigMap

```
apiVersion: v1
kind: Pod
spec:
  containers:
    - image: ...
      envFrom:
        - configMapRef:
            name: <cm_name>      # The ConfigMap this value comes from.
```

As File into a Volume

```
apiVersion: v1
kind: Pod
spec:
  containers:
    - name: <container_name>
      image: ...
      volumeMounts:
        - mountPath: <mount_path>
          name: app-config-volume
  volumes:
    - name: app-config-volume
      configMap:
        name: <cm_name>      # The ConfigMap this value comes from.
```

NOTE

If you don't configure the ConfigMap as volume, and you just add it to the `env` list, the variable is directly set as variable on the container accessible from shell.

6.8.3 Pods update when ConfigMaps changes?

In Kubernetes, Pods don't automatically restart when the ConfigMap they use is updated. This is because the ConfigMap is mounted as a volume or injected as environment variables, and changes to the ConfigMap itself don't trigger an update event for Pods.

If you want automatic updates of Pods when a ConfigMap changes without a rolling update, you can use the k8s-sidecar container to monitor ConfigMaps and trigger the reload of the application within the Pod.

You would add a sidecar container that watches for changes in the ConfigMap and reloads the application or updates the configuration in real-time.

Here, stakater/reloader or another reloader container can monitor the changes and reload the app inside the pod.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  ...
spec:
  ...
  template:
    ...
      spec:
        containers:
          - name: my-app
            image: my-app-image
            volumeMounts:
              - name: config-volume
                mountPath: /etc/config
            env:
              - name: CONFIG_PATH
                value: "/etc/config/my-config"
              - name: configmap-reloader
                image: stakater/reloader:latest
                env:
                  - name: WATCH_NAMESPACE
                    valueFrom:
                      fieldRef:
                        fieldPath: metadata.namespace
                  - name: CONFIGMAP_NAME
                    value: "my-configmap"
                  - name: RESOURCE_TYPE
                    value: "deployment"
            volumes:
              - name: config-volume
            configMap:
              name: my-configmap
```

6.9 Secrets

Store confidential data into ConfigMaps is not a good idea, like passwords or keys, because the value of this objects are clear. There is another type of object similar to ConfigMaps but with data encoded, it is called **Secret**. As in ConfigMaps there are two steps involved in the creation of a Secret: creation and pod injection.

6.9.1 Secret Creation

Imperative

```
$ kubectl create secret generic <secret-name> --from-literal=<key>=<value> \
--from-literal=<key>-<value>
```

```
$ kubectl create secret generic <secret-name> --from-file=<path/to/file>
```

Declarative

```
secret.yaml

apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Host: bXlzcWw=
  DB_User: cm9vdA==
  DB_Password: cGFzd3Jk
```

```
secret.yaml

apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  ssh-privatekey: |
    UG91cmluZzY1RW1vdGljb24lU2N1YmE=
```

WARNING

Note that the values are encoded (in base64), it is important than when you create Secrets in declarative way, so you should previously encode the values in base64:

```
$ echo -n 'mysql' | base64
```

6.9.2 Secrets Pod Injection

As a .file inside Container

```
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
```

```

apiVersion: v1
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: dotfile-secret
  containers:
    - name: dotfile-test-container
      image: registry.k8s.io/busybox
      command:
        - ls
        - "-l"
        - "/etc/secret-volume"
      volumeMounts:
        - name: secret-volume
          readOnly: true
          mountPath: "/etc/secret-volume"

```

As Single Env Vars

```

apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_Password: bXlzcWw=

```

```

apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
  labels:
    name: secret-pod
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      env:
        - name: DB_Pawd
          valueFrom:
            secretKeyRef:
              name: app-secret
              key: DB_Password
...

```

All env vars from Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
data:
  DB_User: root
  DB_Password: bXlzcWw=
  DB_connection_string: mysql.connection.string
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
  labels:
    name: secret-pod
spec:
  containers:
    - name: simple-webapp-color
      image: simple-webapp-color
      envFrom:
        - secretRef:
            name: db-secret
      ...
      ...
      volumeMount:
        ...
  volumes:
    - name: app-secret-volume
      secret:
        secretName: app-secret
      ...
      volumeMount:
        ...
  volumes:
    - name: app-secret-volume
      secret:
        secretName: app-secret
```

6.9.3 View Secrets

```
$ kubectl get secrets
```

WARNING

Note that the values are encoded (in base64), so to retrieve them you have to decode them in base64.

```
$ echo -n 'bXlzcWw=' | base64 -d
```

6.9.4 Secrets Notes

- Secrets are encoded not encrypted, meaning anyone can look up the file that you created for secrets or get the secret object and then decode it in base64 as we did. So it is recommended to not put secret definition inside the git code repositories, because it can be easily retrieved.
- Remember that some that can access to Deployments or Pods in a Namespace can also access to Secrets. So you should consider to configure a role-based access control to restrict access (RBAC).
- Secrets are not encoded in etcd, so none of the data is encrypted or coded in etcd by default. So you can check this document about encrypting secret data at rest: [Encrypting Secret Data at Rest](#)
- There are other better ways of handling sensitive data like passwords in Kubernetes, such as using tools like:
 - AWS Provider
 - Azure Provider
 - HashiCorp Vault
 - Helm Secrets
 - GCP Provider
 - ...

There are different type of secrets, for more information check the [Official Doc](#)

6.10 Multi Container Pods

The idea of decoupling a large monolithic application into sub-components known as **microservices** enables us to develop and deploy a set of independent, small and reusable code. This architecture helps us to scale every service as required, as opposed to modifying the entire application..

But in some case, you will need some instances or service run together to reuse code and scale up and down together, with the same access to Volumes, Secrets, Network... For that purpose you can define multiple containers in a Pod, but this is not usual.

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp
  labels:
    name: simple-webapp
spec:
  containers:
    - name: simple-webapp
      image: simple-webapp
      ports:
        - containerPort: 8080
    - name: log-agent
      image: log-agent
    - name: istio-proxy
      image: istio-envoy
```

You can retrieve logs for a specific container:

```
$ kubectl logs <pod_name> -c <container_name>
```

You can access via shell the specific container:

```
$ kubectl exec -it <pod_name> -c <container_name> -- <command_to_exec>
```

6.11 Init Containers

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. But at times you may want to run a process that runs to completion in a container. For example a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only one time when the pod is first created. Or a process that waits for an external service or database to be up before the actual application starts. That's where initContainers comes in.

An initContainer is configured in a pod like all other containers, except that it is specified inside a initContainers section, like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c',
      'git clone <some-repository-that-will-be-used-by-application> ; done;']
```

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts.

You can configure multiple such initContainers as well, like how we did for multi-containers pod. In that case each init container is run one at a time in sequential order. If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds.

You can retrieve logs for a specific container:

```
$ kubectl logs <pod_name> -c <container_name>
```

NOTE

As well, you can access via shell the specific container, but you should do it when the initcontainer is still running (to debug you can add as command some sleep):

```
$ kubectl exec -it <pod_name> -c <container_name> -- <command_to_exec>
```

6.12 Self Healing Applications

Kubernetes supports self-healing applications through ReplicaSets and Replication Controllers. The replication controller helps in ensuring that a POD is re-created automatically when the application within the POD crashes. It helps in ensuring enough replicas of the application are running at all times.

Kubernetes provides additional support to check the health of applications running within PODs and take necessary actions through Liveness and Readiness Probes. However these are not required for the CKA exam and as such they are not covered here. These are topics for the Certified Kubernetes Application Developers (CKAD) exam and are covered in the CKAD course.

7 Cluster Maintenance

7.1 Upgrades

Scenarios where you might have to take down nodes as part of your cluster, say for maintenance purposes, like upgrading a base software or applying patches, security patches, etc.

7.1.1 What happens when a Node goes down?

If the Node comes back online immediately, then the kubelet process starts and the Pods come back online.

But if the Node does not come back for more than 5 minutes, then the Pods running on that Node are terminated and K8s consider them as dead. And if the Pods are part of a replicaSet then they will be recreated into another Node.

This time of 5 minutes is called **pod-eviction-timeout** and it is configured in the **kube-controller-manager** by default as 5 minutes.

If the Node comes back after pod-eviction-timeout it comes up blank, without any pod scheduled on it.

7.1.2 Node drain

As you can imagine, if you know that a Node will be unavailable for some time, there is a safer way to do it without danger, purposefully drain the node of all the Workloads running on it:

At first, we should tell K8s to not schedule more Pods on the Node, because we are going to take actions on it:

```
$ kubectl cordon <node-name>
```

Then, we should move all the Pods running on the node into another Nodes in the Cluster. This means that Pods are terminated from the node they were running and recreated into another Node.

```
$ kubectl drain <node-name>
```

When we have finished with the maintenance actions, we should make the Node schedulable again:

```
$ kubectl uncordon <node-name>
```

WARNING

A Node cannot be drained if there are Pods running without a ReplicaSet or ReplicationController. It is logic, because without these objects the Pod will be lost forever.

But we can force it:

```
$ kubectl drain --force <node_name>
```

As well, nodes running **daemonSets** cannot be normally drained, so you can use force as well to remove the pods or **--ignore-daemonsets** to do not drain them:

```
$ kubectl drain --ignore-daemonsets <node_name>
```

7.2 Cluster Upgrade Process

7.2.1 K8s Releases

When we install a K8s Cluster, we install a specific version of K8s, we can see the version running on the nodes:

```
$ kc get nodes
```

Let's see for example **v1.11.3**, it consists of three parts (MAJOR, MINOR & PATCH). All the releases we can see here are stable releases. Also we can also see alpha and beta releases called: **v1.x.x-alpha** or **v1.x.x-beta** if it is in a more mature stage.

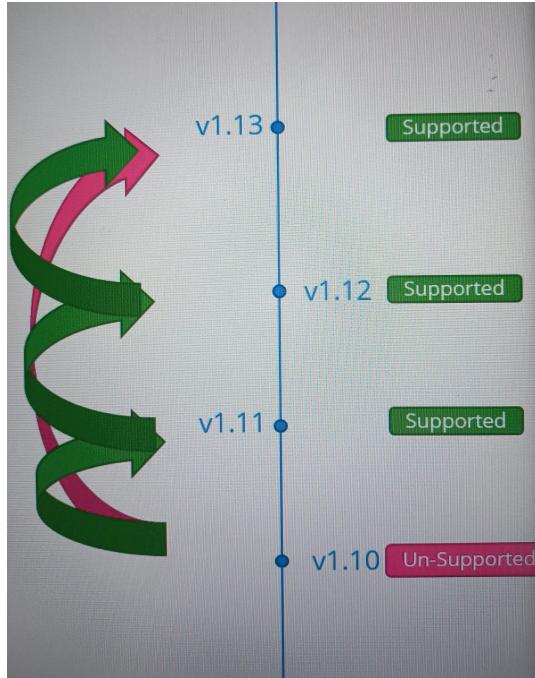
You can check K8s versions: <https://github.com/kubernetes/kubernetes>

7.2.2 Cluster Upgrade Process

The different components can be at different release versions, since the kube-apiserver is the primary component in the control plane, and this is the components that all other components talk to, **none of the other components can be at a version higher than the kube-apiserver**. The **controller-manager** and **scheduler** can be at most one version lower. And **kubelet** and **kubeproxy** can be at most 2 versions lower. The kubectl version could be at most 1 superior or 1 lower.



K8s Support The images supported by K8s are always the last three MINOR version release. But it is not recommended to upgrade 3 versions at time, it is better to upgrade step by step, by 1 version in 1 version.



7.2.2.1 kubeadm Upgrade Plan

Imagine we have a cluster with Master and Worker Nodes, with applications running in production, hosting Pods, serving users. At first they are upgraded the Master Nodes version and then the Worker Nodes versions. When Master Nodes goes down does not mean your Worker Nodes and applications on the Cluster are impacted. All workloads hosted on the Worker Nodes continue to serve users as normal, but all the management functions are down, so you cannot access the cluster using kubectl or other K8s API, so you cannot deploy new applications or modify existing ones. But deployments and ReplicaSets are already running, so if a Pod fails, it is automatically deployed.

Once the Master Nodes are Upgraded, now it is time to upgrade Worker Nodes, there are different strategies to update Worker Nodes, one is update all of them at once, but this is not recommended, because all Pods are down and users are no longer able to access the applications. Once the update is completed, new Pods are scheduled and users can resume access.

The second strategy, the one recommended is to update one Node as a time, so when we update one node, the Pods that were running on it are automatically re-scheduled into another Nodes. So when a Node is upgraded, then the next one is updated and so on until the end.

```
$ kubeadm upgrade plan
```

With this command you can see the version of all components in the K8s Cluster. Also you can see the version which they can be updated. Finally it says that once you have upgrade the K8s components, you have to upgrade manually the kubelet on each Node.

7.2.2.2 Steps to upgrade K8s version from 1.11 to 1.13

1. Update the system to have the latest versions of the binaries and services:

```
$ sudo apt-get update
```

2. Check the `kubeadm` version you want is available for your system:

```
$ sudo apt-cache madison kubeadm
```

3. Upgrade the `kubeadm` version to version 1.12

```
$ sudo apt-get upgrade -y kubeadm=1.12.0-00
```

4. Upgrade the Cluster

```
$ kubeadm upgrade plan
```

```
$ kubeadm upgrade apply v1.12.0 -y
```

5. Get Nodes to see the Nodes version, it will still be in 1.11, because this command shows the version of the kubelets.

```
$ kubectl get nodes
```

6. Go to the Master Nodes and update the system:

```
$ sudo apt-get update
```

7. Check on the master Node that the version you want to install is available:

```
$ sudo apt-cache madison kubelet
```

8. Upgrade kubelet version:

```
$ sudo apt-get upgrade -y kubelet=1.12.0-00
```

```
$ systemctl restart kubelet
```

9. Cordon the first Worker Node we want to upgrade:

```
$ kubectl cordon node-1
```

10. Drain Pods from this Worker Node:

```
$ kubectl drain node-1
```

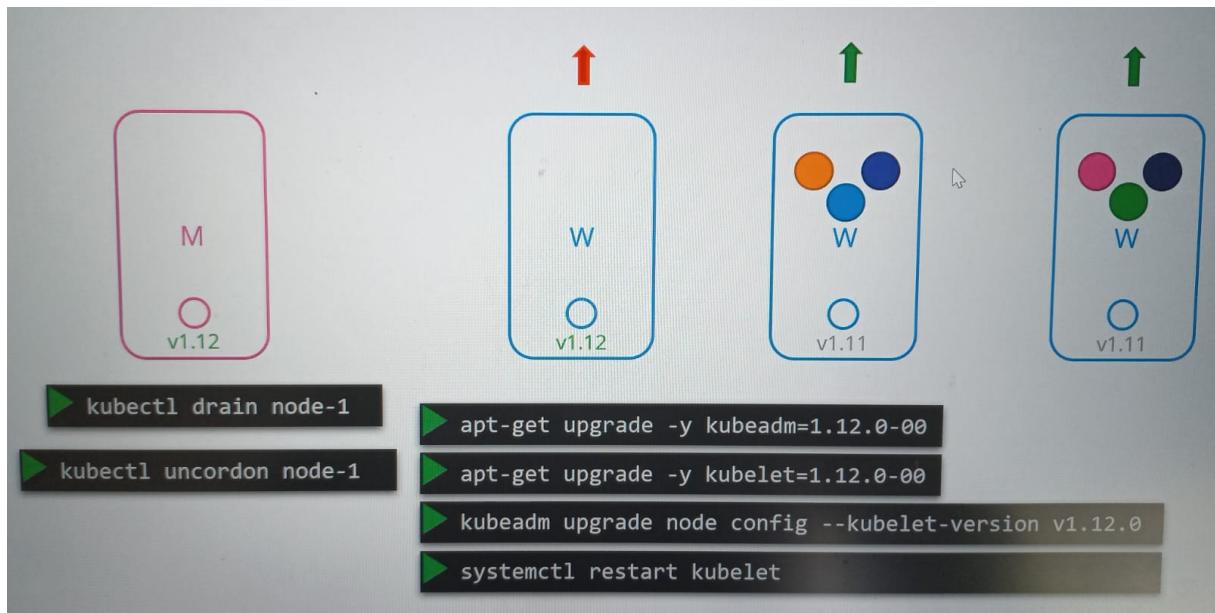
11. Go to the Worker Node and upgrade the `kubeadm` version

```
$ sudo apt-get upgrade -y kubeadm=1.12.0-00
```

12. Now install and download the kubelet version:

```
$ apt install kubelet=1.12.0-00 -y  
  
$ sudo apt-get upgrade -y kubelet=1.12.0-00  
  
$ kubeadm upgrade node config --kubelet-version v1.12.0  
  
$ systemctl restart kubelet  
  
$ kubectl uncordon node-1
```

13. Do the same for the rest of the Worker Nodes



WARNING

If kubeadm or kubelet are not installed in the Node, at first it is needed to install them:

```
$ sudo apt install kubelet=1.27.0-00
```

NOTE

If you are a cluster admin, and you need to access different nodes, you can use SSH to connect to them via their internal ip:

```
$ kubectl get nodes -o wide
```

NOTE

To know easily the IP of a Pod or the Node where it is running, you can use the following command:

```
$ kubectl get po -o wide
```

7.3 Backup and Restore Methods

7.3.1 Resources

In resources, the best way to ensure backups is based on 2 concepts:

- Use always `kubectl apply` instead of `kubectl create` or `kubectl run`. If you need the template, you can always do:

```
$ kubectl create deploy my-deployment --image=nginx --replicas=3 \
--dry-run="client" -o yaml > deployment.yaml
```

```
$ kubectl apply -f deployment.yaml
```

- Always store the resources files into a git repository, maintaining them always updated. And enabling you to recover backups if needed.

With this configuration, even if you lose your entire cluster, you will be able to recover the applications using the templates (YAML), however you will lose all your data.

You can retrieve logs for a specific container:

```
$ kubectl logs <pod_name> -c <container_name>
```

NOTE

If Pods has been created in imperative way, the best approach is then:

- Recover all the resources created on the cluster:

```
$ kubectl get all -A -o yaml > all_resources.yaml
```

- Store it into a git repo.

7.3.2 ETCD

Etcd which is running on all Master Nodes enables the option of take and recover snapshots. For more information check [The Official Documentation](#)

8 Security

Security in K8s cluster is related to how access to the K8s Cluster is managed (authentication) and how are the actions controlled inside it.

8.1 Security in K8s Introduction

The first security that we should take into account inside a K8s Cluster is the security of the Nodes itself. The best practices say:

- Route access disabled
- PasswordAuthentication disabled
- Just SSH key authentication enabled

But, our focus on this section is more on the K8s related security. As we have seen **kube-apiserver** is at the center of all operations within K8s, we interact with it using `kubectl` or by accessing the API directly. With that, you can almost perform any operation inside the cluster. So this should be the first line of defense, controlling the access to **kube-apiserver** itself.

Who can access? K8s has different methods of authentication:

- Username and passwords stored in static files.
- Username and tokens stored in static files.
- Certificates.
- External Authentication providers like LDAP.
- Service Accounts (for machines and K8s resources).

What they can do? K8s has different methods of control that, but the most used is **RBAC Authorization (Role-Based Access Control)**, where users are associated to groups with specific permissions.

All the communication inside the cluster can be secured using TLS encryption.

However, by default, all pods can access all other pods within the cluster, but you can restrict them using **network policies**

8.2 Authentication and Authorization, who can access the cluster?

To a K8s there are a lot of entities that want to access for different purposes:

- **Administrators:** to administrate the cluster.
- **DevOps:** to check all the objects they deploy are running properly.
- **Developers:** to check the applications are running properly.
- **End Users:** which are going to use the application. Actually it is not managed by the K8s cluster, it is managed by the application itself.
- **External Tools:** which connects to the cluster to retrieve data or trigger actions.

K8s does not manage user accounts natively, it relies on an external source like a file with user details or certificates or a third party identity service like LDA to manage the authentication. So, you cannot directly create users into a K8s Cluster.

The only thing related to Authentication and Authorization that K8s can manage is **ServiceAccounts**. All user Authentication and Authorization is managed by the API server (kube-apiserver). So, how does the **kube-apiserver** authenticates? There are different methods:

- List of usernames and passwords on an static file.
- List of usernames and tokens on an static file.
- Certificates.
- LDAP.

Let's start with **Static Files**.

8.2.1 Static File Authentication

A list of users and their passwords or tokens can be created in a CSV file and use it as the source for user information. The file should have 4 columns: password or token, username, userid, group.

It should be configured on the **kube-apiserver** service or pod (depending how are you running your cluster) with:

```
--basic-auth-file=user-details.csv
```

Or if we want tokens:

```
--token-auth-file=user-details.csv
```

To authenticate using static files, you should:

```
$ curl -vk https://<master-node-ip>:6443/api/v1/pods -u "user1:password123"
```

To authenticate using static files, you should:

```
$ curl -vk https://<master-node-ip>:6443/api/v1/pods \
--header "Authorization: Bearer fafhflafhsash..."
```

WARNING

This is not the recommended authentication and authorization mechanism, as it is not much secure.

8.3 TLS Introduction (not for K8s)

8.3.1 Why do we need certificates?

Certificates are used to guarantee trust between 2 parties during a transaction. For example: when a user is trying to access a web server, TLS certificates ensure that the communication between the user and the server is encrypted and they are both who they say they are (there is no man in the middle).

The data between communications are encrypted using asymmetric keys, so only the receiver of the data can decrypt the data, even the sender cannot decrypt the message once it has been sent. A good example to understand asymmetric encryption is to think like:

- **Public Key:** Lock that anyone can try to open.

- **Private Key:** The only key which can open the lock, never should be shared.

All the times you're requested for connection you are sharing your public key, so it is really public XD. Here is the "key" XD: public and private key are related each other, the sender encrypts the message using his own private key and the public key of the server. And then the server is able to decrypt it using his private key, all thanks to the relation between pubkey and privkey.

So the man in the middle is dead, because he does not have any of the private keys and he does not know the relation between public keys and private keys. In summary, he cannot decrypt the message.

So the only chance the man in the middle has, is to recreate the exact website and tell you: hey, I'm your bank, send me your credentials! So the website has its own public and private keys and it starts the negotiation with you. So if you answer, you are dead.

But what if you could know in advance if the pubkey of the server is a legitimate key from your real bank server? So that's what certificates exactly do! When you are connecting through **https**, your bank account is sending you a **Certificate** instead of the pubkey. This certificate (among other things) has the pubkey inside. Inside the certificate there is information as:

- CN (Common Name): domain it is certifying.
- Identity: individual, organization, or device
- Location: country, state, locality.
- PubKey.
- Signature.
- Signature Algorithm.
- Validity Period.

But, anyone can create its own certificate, so for that, the browsers check the certificates are signed by a **(CA) Certificate Authority** which recognizes that the certificate is legitimate and it is used to certify the domain it is saying. CA's can sign and validate the certificates, the most popular are: DigiCert, Symantec, Comodo, GlobalSign.

How all of them works?

1. You generate a **CSR: Certificate Signing Request** with your domain.

```
$ openssl req -new -key my-bank.key -out my-bank.csr \
-subj "C=US/ST=CA/O=MyOrg, Inc./CN=mydomain.com"
```

2. The CSR should be sent to the **CA** for signing.
3. The CA verify your details and once it checks out, they sign the certificate and send it back to you.
4. You now have a certificate signed by a CA which the Browsers trust.

Note

How Browsers know that the CA which signed the certificate is trustable? Because CA's generate the certificate using it's own pair keys, so the public keys of all the legitimate CAs are built into all the browsers. Then the browsers use the public key of the CA to validate that the certificate is actually signed by a legitimate CA.

As well, you can use **certbot (let's encrypt)** (like a CA), but more quickier and easier to do, take a look on [CertBot Official Documentation](#).

Once your browser and you trust in the server, you can send your sensitive data which will be encrypted using the asymmetric key encryption method, and only and actually just the receiver can decrypt.

Note

In some cases, servers can also request a client certificate to authenticate the user or device connecting to them, verifying they are who they say they are. You can not send them, then the server will decide what to share or not share with you.

However, your operating system or device (Windows, macOS, Linux, or even a smartphone) manages the whole process for you behind the scenes; your operating system comes with a built-in set of trusted root certificates from trusted certificate authorities (CAs) like DigiCert, Let's Encrypt, etc. These CAs issue certificates to servers (like the one hosting ChatGPT).

Certificates including public keys are stored always with this two extensions:

- **.pem**: server.pem
- **.crt**: server.crt

The private key needed to decrypt messages are stored always with this extension **.key**: server.key

Note

As the key pairs are related, you can encrypt using both of them. So if you encrypt with one of them, you only can decrypt with the other.

But take care, the common use is to encrypt with your public key and the public key of the receiver to just the receiver can decrypt using its private key. But if you encrypt using your privatekey, you message will be opened to the world because it can be decrypted with your pubkey.

All the infrastructure regarding this is called **PKI (Public Key Infrastructure)**, which contains:

- Certificates
- Certificate Signing Request
- CA'S
- PubKey and PrivKey
- ...

So with all of this we have 3 types of certificates based on where they are:

- **Server Certificates**: the ones configured on the servers side.
- **Root Certificate**: configured on the CA servers.
- **Client Certificates**: the ones configured on the clients side.

So in order to know properly which one is server, client... I recommend to name them properly:

- server.pem / server.crt / server.key
- client.pem / client.crt / client.key

8.4 Introduction to TLS on K8s

A K8s Cluster consists of a set of Master and Worker nodes, of course with constant communication between them. So, all the communications inside a K8s Cluster must be secured, so should be encrypted. For example an administrator using `kubectl` to interact with cluster must establish secure TLS connection as well as communications between all the components within a K8s Cluster.

To accomplish that, all the clients and servers in a K8s Cluster should have their client certificate and privkey and their server certificate and privkey, to ensure they are who they say they are.

So let's identify the different servers and clients we have within a K8s Cluster.

Kube-apiserver exposes an HTTPS service that other components as well as external users can connect to manage the K8s Cluster. So it is a **server**, so it must have its server certificate and privkey. But it is a client as well, because it needs to communicate with etcd and kubelet, so it can use just the same pair of cert-key for server and client or it can use two pair cert-key; one for server and other for client:

- apiserver.crt
- apiserver.key

Etcd stores all information about the cluster, so it requires as well its certificate and privkey:

- etcdserver.crt
- etcdserver.key

Kubelet exposes an HTTPS API endpoint which the kube-apiserver talks to interact with the Worker Nodes, so:

- kubelet.crt
- kubelet.key

Kube-scheduler is a client of the kube-apiserver, telling which pods need to be deployed on which node, so:

- scheduler.crt
- shceduler.key

User accessing kubectl is a client, so we need as well our client certificate and privkey to authenticate to the kube-apiserver:

- admin.crt
- admin.key

As well **kube-controller-manager** and **kube-proxy** are kube-apiserver clients, so they need their certificates and privkeys:

- controller-manager.crt
- controller-manager.key
- kube-proxy.crt
- kube-proxy.key

K8s at least one CA (Certificate Authority) for your cluster, to sign all these certificates. In fact, you can have more than one. The CA's have their own pair of cert-key:

- CA.crt
- CA.key

8.5 Certificate Creation for K8s

For generating certificates there are different tools available such as:

- `openssl`
- `easyrsa`
- `cfssl`
- ...

We are going to use the `openssl` tool to generate the certificates.

8.5.1 Generating CA Certificate

1. Create the **CA** private key:

```
$ openssl genrsa -out ca.key 2048
```

2. Request a CSR (Certificate Signing Request) for the **CA** along with the key we've just created:

```
$ openssl req -new ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr
```

Note

A CSR (Certificate Signing Request) will be like a normal certificate, with same information, but without the signature. When we create a CSR, we should specify the CN, which will be the name of the component the certificate is for.

3. Self-sign the certificate:

```
$ openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt
```

Note

As this certificate will be for the CA itself, it will be self-signed. For the rest of the generated certificates, they are going to be signed with the CA certificate-privkey pairs.

8.5.2 Generating Client Certificates

8.5.2.1 Admin User Certificate

1. Create the Admin User private key:

```
$ openssl genrsa -out admin.key 2048
```

2. Request a CSR for the Admin User along with the key we've just created:

```
$ openssl req -new admin.key -subj "/CN=kube-admin/O=system:masters" \
-out admin.csr
```

Note

The system:masters label into the CN is added to differentiate admin permissions from general permissions. To make it work, a group named system:masters should exist on K8s with administrative privileges.

3. Sign the certificate with the CA certificate and CA privkey, to make the certificate valid within the Cluster:

```
$ openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out admin.crt
```

This `admin.crt` will be the Certificate that admin users are going to use to authenticate into K8s Cluster.

8.5.2.2 How to use Admin User certificate

Not used way

```
$ curl https://kube-apiserver:6443/api/v1/pods \
  --key admin.key --cert admin.crt \
  --cacert ca.crt
```

Used way - Kubeconfig file

```

./kube/config

apiVersion: v1
clusters:
- cluster:
    certificate-authority: TST_ca.crt
    server: https://<HA_TST_Proxy>:6443
    name: TST_cluster
- cluster:
    certificate-authority: PRD_ca.crt
    server: https://<HA_PRD_Proxy>:6443
    name: PRD_cluster
- cluster:
    insecure-skip-tls-verify: true
    server: https://<RANCHER_URL>/k8s/clusters/c-u7i98
    name: STA_Rancher_Cluster
- cluster:
    certificate-authority-data: "LS0tLS1CRUdJTiBDR..." # base64 encoded
    server: https://<RANCHER_URL>/k8s/clusters/c-kt3ik
    name: PRD_Rancher_Cluster

contexts:
- context:
    cluster: TST_cluster
    namespace: default
    user: kubernetes-admin
    name: TST_cluster
- context:
    cluster: PRD_cluster
    namespace: default
    user: kubernetes-admin
    name: PRD_cluster
...
current-context: TST_cluster
kind: Config
users:
- name: kubernetes-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key

```

NOTE

If the server field in your kubeconfig file is not pointing to any of your Kubernetes master nodes directly, it is very likely pointing to a **load balancer** that is distributing traffic across the control plane (master nodes). This is the best practice setup in **high-availability (HA)** Kubernetes clusters. They should be an **NGNIX** or **HAProxy**

- Multiple master nodes are deployed for redundancy and fault tolerance.
- If the kubeconfig is configured to point to a single master node, the failure of that node would cause the API server to be unreachable.
- By configuring the server field to point to a load balancer, you ensure that requests from `kubectl` (or any other Kubernetes client) are distributed across the available master nodes. If one master node fails, the load balancer will redirect traffic to the others.

WARNING

For all this certificates to work, and validate each other, all the clients and servers need a copy of the CA's root certificate.

8.5.2.3 Other User/Server Certificates

Like the ones for Kube-apiserver, Kube-scheduler Kubelet, Proxy, Kube-controller, etc. You would need to follow the procedure explaining just before with the **Admin certificate** but changing the naming of the certificates.

Etcd-server: you need to specify on the configuration the location of the certificate and private key:

```
--key-file=/path/to/certs/etcd-server.key  
--cert-file=/path/to/certs/etcd-server.crt  
--client-cert-auth=true  
--peer-cert-file=/path/to/certs/etcd-peer1.crt  
--peer-client-cert-auth=true  
--peer-key-file=/etc/kubernetes/pki/etcd/peer.key  
--peer-trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt  
--trusted-ca-file=/etc/kubernetes/pki/etcd/ca.crt
```

Kube-apiserver: the key at this point is to configure not just one DNS, a lot of them because it is the component called by all the rest of k8s, so at first, generate a file called `openssl-apiserver.conf`

```
openssl-apiserver.conf  
  
[req]  
req_extensions = v3_req  
distinguished_name = req_distinguished_name  
[ v3_req ]  
basicConstraints = CA:false  
keyUsage = nonRepudiation,  
subjectAltName = @alt_names  
[ alt_names ]  
DNS.1 = kubernetes  
DNS.2 = kubernetes.default  
DNS.3 = kubernetes.default.svc  
DNS.4 = kubernetes.default.svc.cluster.local  
IP.1 = 10.96.0.1  
IP.2 = 172.17.0.87
```

Then use it when creating the **CSR**:

```
$ openssl req -new apiserver.key -subj "/CN=kube-apiserver" \  
-out apiserver.csr -config openssl-apiserver.conf
```

On the configuration file of the kube-apiserver:

```
--etcd-cafile=/var/lib/kubernetes/ca.pem/lib/kubernetes/ca.pem
--etcd-certfile=/var/lib/kubernetes/ca.pem/lib/kubernetes/apiserver-etcd-client.crt
--etcd-keyfile=/var/lib/kubernetes/ca.pem/lib/kubernetes/apiserver-etcd-client.key
--kubelet-certificate-authority=/var/lib/kubernetes/ca.pem
--kubelet-client-certificate=/var/lib/kubernetes/apiserver-kubelet-client.key
--kubelet-https=true
--service-account-key-file=/var/lib/kubernetes/service-account.pem
--client-ca-file=/var/lib/kubernetes/ca.pem
--tls-cert-file=/var/lib/kubernetes/apiserver.crt
--tls-private-key-file=/var/lib/kubernetes/apiserver.key
```

Kubelet server: as we know, it is running on each node, and it is the resource which the **apiserver** talks to monitor the node as well as send information regarding what pods to schedule on this node. So you need a certificate privatekey pairs for each kubelet, which means on each node in the cluster. This certificate is going to be named with the label of the **node**: node01, node02, node03, etc.

On the kubelet configuration file:

```
kind: KubeletConfiguration
apiVersion: kubelet.config.k8s.io/v1beta1
authentication:
  x509:
    clientCaFile: "/var/lib/kubernetes/ca.pem"
authorization:
  mode: Webhook
clusterDomain: "cluster.local"
clusterDNS:
  - "10.32.0.10"
podCIDR: "${POD_CIDR}"
resolvConf: "/run/systemd/resolve/resolv.conf"
runtimeRequestTimeout: "15m"
tlsCertFile: "/var/lib/kubelet/kubelet-node01.crt"
tlsPrivateKeyFile: "/var/lib/kubelet/kubelet-node01.key"
```

On the other hand, kubelet perform as a client too, so it needs certificate to authenticate into the kube-apiserver, so the clients should be named mandatory: **system:node:node0X**. To manage their permissions the nodes should be added to a group called **system**.

8.5.3 How to generate and configure all the K8s Certificates?

You need to know that if you create the **K8s Cluster from scratch** (without using `kubeadm`), all the certificates needs to be generated, properly stored and configured by yourself. But, if you rely on **kubeadm**, it takes care of automatically generating and configuring the cluster for you. As well as deploy all the components as native services (Pods) in the nodes in the hard way, it configure them properly to catch the generated certificates it has previously generated.

8.5.4 Certificate Details

8.5.4.1 Where can I find the certificates?

At first, to know which certificate belong to which resource, it is not enough trying to guess it based on the name convention. The good way of doing is: **checking the configuration files of the resources**. For example for the **kube-apiserver**, you can find the template config in:

```
/etc/kubernetes/manifests/kube-apiserver.yaml
```

The good way to check it:

```
$ kubectl describe po <resource> | grep -i "cert"
```

8.5.4.2 How can I see all the data inside the certificate?

```
$ openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
```

Subject:CN The Common Name (CN) is the primary identifier within this field, it usually represents the hostname or domain name of the server.

Subject:SAN The Subject Alternative Name (SAN) field is an extension of the certificate that allows multiple identifiers (domains, IP addresses, email addresses, etc.) to be included in a single certificate. They are used when a certificate needs to secure multiple hostnames.

Validity refers to the time frame during which the certificate is considered valid and trusted by clients. It is defined by two dates:

- Not Before (Start Date)
- Not After (Expiration Date)

Issuer: entity responsible for signing and issuing the digital certificate, so the issuer is typically the Certificate Authority (CA).

8.5.4.3 Check certificate issues?

When you think there are certificates issues, you will start checking logs of the nodes:

- If the components are Pods/DaemonSets:

```
$ kubectl logs etcd-master
```

- If the components are services:

```
$ journalctl -u etcd.service -l
```

WARNING

If the **kube-apiserver** is down regarding a certificate issue, the kubectl command won't work, so you will need to do:

```
$ docker ps
```

```
$ docker logs <container>
```

8.6 Certificates API

8.6.1 Need of the API?

As far as we have seen, does not matter if it is done by **kubeadm** tool or from scratch. The first thing kubeadm or we (in case of scratch) needs to do is create the CA in one of the master nodes. Then whenever a component or user want to create a new certificates, needs to use the CA of this master node and sign the certificate. So one of the admins should access the master node and sign the CSR with the CA. But as your infrastructure and your teams growth, a better way of managing CSR's is needed. As well as too manage / rotate certificates when they expire.

8.6.2 How does Certificate API works?

That's why K8s has a built-in certificates API that can do this for you. So you just need to send your CSR directly to K8s API.

So now when the administrator receives the CSR, instead of logging into the master node and sign the request with the CA, he should create a K8s resource: **CertificateSigningRequest**, that way, all the CSR's can be seen by the administrators of the cluster, and the request can be **reviewed and approved** using **kubectl commands**. Once the request is signed, the certificate can be extracted and shared with the user.

So know the procedure is:

1. The user creates his private key and the CSR and sends it to the administrator:

```
$ openssl genrsa -out alex.key 2048  
  
$ openssl req -new -key alex.key -subj "/CN=alex" -out alex.csr
```

2. The administrator takes the key and creates **CertificateSigningRequest** resource in K8s:

```
apiVersion: certificates.k8s.io/v1  
kind: CertificateSigningRequest  
metadata:  
  name: alex  
spec:  
  expirationSeconds: 600  
  groups:  
    - system:authenticated  
  signerName: kubernetes.io/kub-apiserver-client  
  usages:  
    - digital signature  
    - key encipherment  
    - server auth  
  request: # cat alex.csr | base64  
    IYHGFkakdhks3d&fdasf  
    kfj1kjsu8ygfafdfjdfj  
    ...
```

WARNING

Inside of the resource **CertificateSigningRequest** the CSR should be introduced into the **request** field but encoded in base64.

```
$ cat alex.csr | base64
```

Once the object is created, the certificate signing request can be seen by administrators:

```
$ kubectl get csr <CertificateSigningRequest_name>
```

And then approve it or deny it:

```
$ kubectl certificate approve <CertificateSigningRequest_name>
```

```
$ kubectl certificate deny <CertificateSigningRequest_name>
```

The certificate can be extracted from the csr decoding it. It is always under `.status.certificate`:

```
$ kubectl get csr alex -o yaml | grep -i "certificate:"
```

```
$ echo "oisyATDK09..." | base64 -d
```

8.7 Kubeconfig

8.8 Kubeconfig File

In order to authenticate into a K8s Cluster, we can do it in 2 not very useful ways:

1. Pinging the API with the certificates:

```
$ curl https://<loadbalancer_url>:6443/api/v1/pods \
-- key admin.key --cert admin.crt --cacert ca.crt
```

2. Using the `kubeconfig` with the flags of the certificates:

```
$ kubectl get po \
-- server <loadbalancer_url>:6443
--client-key admin.key
--client-certificate admin.crt
--certificate-authority ca.crt
```

But the most spreaded way of doing is to have a **kubeconfig file**. The default location for this file which you don't have to specify when doing `kubectl` is:

```
~/.kube/config
```

If you want to use another `kubeconfig` file or another location, you will need to specify it like this:

```
$ kubectl get pods --kubeconfig /path/to/your/custom/kubeconfig
```

The `kubeconfig` has 3 sections:

- **clusters:** different K8s Clusters you have access to. Because it is common to have different clusters, one for sandbox, other for test, other for dev, other for prod, etc.
- **users:** user accounts for different cluster accesses. They will have different permissions on the different clusters.
- **contexts:** marry users with clusters together, defining which user match which cluster. For example you can create a context called: `admin@production`.

```
/.kube/config

apiVersion: v1
clusters:
- cluster:
  certificate-authority: TST_ca.crt
  server: https://<HA_TST_Proxy>:6443
  name: TST_cluster
- cluster:
  certificate-authority: PRD_ca.crt
  server: https://<HA_PRD_Proxy>:6443
  name: PRD_cluster
- cluster:
  insecure-skip-tls-verify: true
  server: https://<RANCHER_URL>/k8s/clusters/c-u7i98
  name: STA_Rancher_Cluster
- cluster:
  certificate-authority-data: "LS0tLS1CRUdJTiBDR..." # base64 encoded
  server: https://<RANCHER_URL>/k8s/clusters/c-kt3ik
  name: PRD_Rancher_Cluster

contexts:
- context:
  cluster: TST_cluster
  namespace: default
  user: kubernetes-admin
  name: kubernetes-admin@TST
- context:
  cluster: PRD_cluster
  namespace: default
  user: kubernetes-admin
  name: kubernetes-admin@PRD
  ...

current-context: TST_cluster
kind: Config
users:
- name: kubernetes-admin
  user:
    client-certificate: admin.crt
    client-key: admin.key
```

To set your current context, you can specify the line:

```
current-context: <context-name>
```

WARNING

In the same way as for CSR the data should be encoded in base64, the same for **certificate-authority-data**. It should be **encoded** in base64.

8.9 kubectl config command

Help:

```
$ kubectl config -h
```

To view the kubeconfig from kubectl:

```
$ kubectl config view
```

To get quickly all the context available in your **kubeconfig** file:

```
$ kubectl config get-contexts [--kubeconfig=</path/to/custom/kubeconfig>]
```

To change the context: To get quickly all the context available in your **kubeconfig** file:

```
$ kubectl config use-context <context-name>
```

To change the **namespace** context: To get quickly all the context available in your **kubeconfig** file:

```
$ kubectl config set-context <context-name> [--current] --namespace=<namespace>
```

8.10 kubectx

8.10.0.1 Introduction

While this is excellent for hands-on practice, in a real “live” kubernetes cluster implemented for production, there could be a possibility of often switching between a large number of namespaces and clusters. That’s why the community has developed the following magic tool: [kubectx](#)

8.10.0.2 kubectx

List contexts:

```
$ kubectx
```

Check current context:

```
$ kubectx -c
```

Switch context:

```
$ kubectx <new_context_name>
```

To switch back to previous context:

```
$ kubectx -
```

8.10.0.3 kubens

kubens is a tool which comes included with kubectx.

To switch to a new namespace:

```
$ kubens <new_namespace>
```

Back to the previous namespace:

```
$ kubens -
```

8.11 API Groups

When we talk about K8s API, we are talking about the endpoint exposed by **kube-apiserver**.

If we want to check the version of the cluster:

```
$ curl https://master-node-01:6443/version -k  
--key admin.key  
--cert admin.crt  
--cacert ca.crt
```

```
$ curl https://master-node-02:6443/version -k  
--key admin.key  
--cert admin.crt  
--cacert ca.crt
```

```
$ curl https://load_balancer:6443/version -k  
--key admin.key  
--cert admin.crt  
--cacert ca.crt
```

But how many endpoint does it have?

- **/metrics**: to monitor the health of the clister.
- **/healthz**: to monitor the health of the clister.
- **/version**: check cluster version.
- **/api**: core group.
- **/apis**: named group.
- **/logs**: integrating third parties logs applications.

To check all the available endpoints make the query without specifying path:

```
$ curl https://load_balancer:6443 -k
```

8.11.1 /api

Core K8s API, with all the resources in the stable version of K8s.

8.11.1.1 /v1

- /api/v1/pods
- /api/v1/namespaces
- /api/v1/events
- /api/v1/endpoints
- /api/v1/nodes
- /api/v1/rc
- /api/v1/bindings
- /api/v1/PVC
- /api/v1/PV
- /api/v1/configmaps
- /api/v1/services
- /api/v1/secrets

8.11.2 /api

All the newer features, beta.

Each resource in this section has 6 actions:

- list
- get
- create
- delete
- update
- watch

/apps

- /apps/v1/replicasets
- /apps/v1/deployments
- /apps/v1/statefulsets
- /apps/v1/daemonsets

/extensions /networking.k8s.io

- /networking.k8s.io/v1/networkingpolicies

/storage.k8s.io /authentication.k8s.io /certificates.k8s.io

For curl you need to always specify the key, certificate and CA certificate like this:

```
$ curl https://load_balancer:6443/version -k  
--key admin.key  
--cert admin.crt  
--cacert ca.crt
```

But it can be really annoying, so the best option is to use:

```
$ kubectl proxy
```

kubectl proxy launches a proxy service locally on port 8001, using the endpoint, credentials and certificates from your kubeconfig file. That way, you don't have to specify all this data in the curl command.

Then you can access this service:

```
$ curl http://localhost:8001 -k
```

8.12 Authorization

Authentication is how someone can gain access to a cluster, but authorization is, once they are on the cluster, what can they do? Authorization is needed because you would like to have different "roles" on your cluster for the different user accessing the cluster:

- **Administrators:** get, create, update, delete,... Everything as well at cluster level.
- **DevOps:** get, create, update, delete,... Just at resources level. Restricted to some namespaces.
- **Robots:** like Jenkins, Bamboo, etc. Just at resources level.

8.12.1 Node Authorization

As we discussed before, the different kubelets on the different nodes need permissions to check things and share them with the kube-apiserver, so they need to have some special access on the cluster. This access is given by the **Node Authorizer**. We discussed that kubelets should be part of the **system:nodes** group and have a name prefixed with system node. So any request coming from a user with the name system node and part of the system:nodes group is authorized by the **Node Authorizer**.

8.12.2 ABAC

Attribute Based Authorization Control is where you associate a user or a group of users with a set of permissions. So as an example we can say that a user can:

- View Pods
- Create Pods
- Delete Pods

You can do creating the following policy file:

```
{
  "kind": "Policy",
  "spec": {
    "user": "dev-user",
    "namespace": "*",
    "resource": "pods",
    "apiGroup": "*"
  }
}
```

So you can pass this into the API server to apply it. But as you can imagine, this is not a good way of operate, because any time we want to change some policies, we are going to be modifying these policies manually and restart the kube-apiserver.

8.12.3 RBAC

Role-Based Access Control makes the management much easier. Instead of directly associate a user or a group with a set of permissions (policies), we define **roles**, for example developers. And we assing to this role the required permissions for developers. Then we just need to associate all the developers to that role.

When a change needs to be made to the user's access we simply modify the role and it will reflects on all developers (all users assigned to developer role) immediately. So it provides a more standard approach to managing access within the K8s Cluster.

8.13 Webhook

You can find third party tools (like for example Open Policy Agent or Rancher) which help with admissions control and authorization.

8.13.1 Authorization Mode

Your K8s Cluster will use the Authorization Mode that you configure on the kube-apiserver:

- AlwaysAllow
- AlwaysDeny
- Node
- RBAC
- ABAC
- Webhook

You can check that on the kubeapiserver configuration, if nothing specified will be by default AlwaysAllow:

```
--authorization-mode=AlwaysAllow
```

More than one can be specified, and then the users will be authorized in the same order it has been specified, so every time a method denies the request it goes to the next one in the chain, and as soon as a module approves the request no more check are done and the user is granted permissions.

```
--authorization-mode=Node,RBAC
```

8.13.2 RBAC

8.13.2.1 Roles and RoleBindings

We have explained what is RBAC in the previous lecture. Getting down to business, how do we create roles? As you are imagine, YES! Creatin a K8s object like this:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: developer
  namespace: <namespace_name>
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list", "get", "create", "update", "delete"]
- apiGroups: [""]
  resources: ["ConfigMaps"]
  verbs: ["list", "get", "create"]
```

apiGroups: as we have seen before, this field identifies the group of resources within Kubernetes.

- "" (default): Refers to the core API group, which includes resources like pods, services, namespaces, nodes, etc. If you're working with core resources like ConfigMaps, Secrets, or Pods, you would use "" for apiGroup.
- **apps:** This is the group that contains resources like deployments, statefulsets, and daemonsets.
- **batch:** Refers to jobs and cron jobs. Resources like Job and CronJob belong to this group.
- **networking.k8s.io:** For networking-related resources like NetworkPolicy and Ingress.
- **policy:** Contains policies like PodDisruptionBudget.
- **storage.k8s.io:** For storage-related resources like StorageClass, VolumeAttachment, and CSIDriver.

NOTE

Difference between list and get?

- **list:** Fetches a list of all resources of a certain type. When you want an overview of all resources (e.g., all pods or configmaps)

```
$ kubectl get po -n kube-system
```

- **get:** Fetches a specific resource by name. When you want details or **describe** a single resource (e.g., one pod or one configmap)

```
$ kubectl get/describe po my-pod -n kube-system [-o yaml]
```

One important thing to know is that **get** and **list** are distinct verbs in Kubernetes RBAC, and **get** does not implicitly grant the ability to list resources.

If a user only has get permission: they can retrieve details of specific resources (e.g., kubectl get pod my-pod), but they cannot retrieve a list of resources (e.g., kubectl get pods).

The next step is to link the user to that role, for this purpose we have the resource: **RoleBinding**:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: alex-developer-binding
  namespace: <namespace_name>
subjects:
- kind: User # Or ServiceAccount or Group
  name: alex # The user or service account you are granting access to
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: developer # Refers to the ClusterRole above
  apiGroup: rbac.authorization.k8s.io

```

NOTE

Roles and RoleBinding are **namespace-scoped**, or what is the same: **sensitive to namespaces**. So if you do not specify a namespace in the Role, K8s will default to using the **namespace where the Role object itself is created**. In other words, the Role will only apply to the namespace where you create it.

To give access to multiple namespaces, you would need to create a Role in each of the namespaces where you want to apply the permissions. You cannot assign a Role to multiple namespaces directly. Instead, you will need to:

- Create a **Role** in each of the desired namespaces (for example, in 3 namespaces).
- Bind the Role to the user or service account in each namespace using a **RoleBinding**.

If you want to give access to all namespaces, instead of using a Role, you should use a **ClusterRole** and a **ClusterRoleBinding**, without specifying any namespace for example:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: developer
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["list", "get", "create", "update", "delete"]
- apiGroups: []
  resources: ["ConfigMaps"]
  verbs: ["list", "get", "create"]

```

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: alex-developer-binding
subjects:
- kind: User # Or ServiceAccount or Group
  name: alex # The user or service account you are granting access to
  apiGroup: rbac.authorization.k8s.io
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:bootstrappers:kubeadm:default-node-token
roleRef:
  kind: Role
  name: developer # Refers to the ClusterRole above
  apiGroup: rbac.authorization.k8s.io

```

NOTE 2

You can allow just permissions to **specific named resources** within a resource type, rather than granting access to all resources of that type.

In other words, while the resources field specifies what types of resources (e.g., pods, configmaps) the Role can manage, the resourceNames field allows you to narrow it down to only specific instances of those resources based on **name exact match**.

```

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: multi-resource-access
  namespace: my-namespace
rules:
- apiGroups: []
  resources: ["pods"]
  verbs: ["get", "list"]
  resourceNames: ["my-pod", "another-pod"] # Access limited to these two pods
- apiGroups: []
  resources: ["configmaps"]
  verbs: ["get", "list"]
  resourceNames: ["my-configmap", "another-configmap"]
  # Access limited to these two configmaps

```

NOTE 3

In Kubernetes **RBAC** system, **groups** are defined and managed **externally** (e.g., through an identity provider like **LDAP**, **OIDC**, or another authentication system).

Kubernetes itself does not have built-in functionality to **create or manage group membership**. However, you can reference groups in **RoleBindings** and **ClusterRoleBindings** by specifying the group name in the RBAC configuration.

To check the roles and rolebindings:

```
$ kubectl get/describe roles
```

```
$ kubectl get/describe rolebindings
```

What if you like a simple user want to check if you have access to a particular resource in the cluster?

```
$ kubectl auth can-i create deployments [--namespace=]
```

```
$ kubectl auth can-i delete nodes -A
```

Just if you are admin, you can:

```
$ kubectl auth can-i delete pods --as maria [--namespace=]
```

8.13.2.2 ClusterRoles and ClusterRoleBindings

We have discussed about Roles and RoleBindings in the previous section, and we discussed that are **namespaced**, meaning they are created within namespaces (if you don't specify namespace they will be created on the current namespace). But what about accesses to list or get nodes? Nodes are not namespaced, they are **cluster-wide** and they cannot be associated to any particular namespace. There are more **cluster-wide** resources:

- Nodes
- PV's
- ClusterRoles
- ClusterRoleBindings
- CertificateSigningRequests
- Namespaces
- ...

To know all the **non-namespaced** resources:

```
$ kubectl api-resources --namespaced=false
```

To know all the **namespaced** resources:

```
$ kubectl api-resources --namespaced=true
```

A ClusterRole is exactly the same as a Role, just that it can provide access to **cluster-wide** resources. The same for ClusterRoleBinding.

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: developer
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list", "get", "create", "update", "delete"]
- apiGroups: [""]
  resources: ["ConfigMaps"]
  verbs: ["list", "get", "create"]

```

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: alex-developer-binding
subjects:
- kind: User # Or ServiceAccount or Group
  name: alex # The user or service account you are granting access to
  apiGroup: rbac.authorization.k8s.io
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:bootstrappers:kubeadm:default-node-token
roleRef:
  kind: Role
  name: developer # Refers to the ClusterRole above
  apiGroup: rbac.authorization.k8s.io

```

NOTE

As well as give access to all cluster-wide resources, it gives as well permissions to see resources on **ALL** namespaces on the cluster.

To check permissions in **not namespaced resources**:

```
$ kubectl auth can-i get nodes -A
```

8.14 ServiceAccounts and Tokens

8.14.1 Introduction

There are two types of accounts in K8s:

- **User Accounts:** used by humans, can be for an administrator, a devops, etc.
- **Service Accounts:** used by machines, like an application to interact with the K8s Cluster (for example Prometheus, Fluentd, Bamboo, Jenkins).

8.14.2 Creation and Usage

To create a service account:

```
$ kubectl create serviceaccount <serviceaccount_name>
```

But, how can we make use of this **ServiceAccount** in order to internal or external machines authenticate into the cluster? The answer is with tokens! After **ServiceAccount** creation, we should as well create a **token** associated to it using the **TokenRequest API**:

```
$ kubectl create token <serviceaccount_name> [--duration 10m] # By default 1m
```

If you want to directly attach the token to a Pod in order to use them from the Pod running application:

```
$ kubectl create token <serviceaccount_name> [--duration 10m] \
--bound-object-kind Pod --bound-object-name mypod
```

If you want to bound the token into a Secret:

```
$ kubectl create token <serviceaccount_name> [--duration 10m] \
--bound-object-kind Secret --bound-object-name mysecret
```

The generated token will be a bearer token, which means is simply a cryptographic token (usually a JWT or opaque token) that proves the client's identity. In K8s case, it is a JWT (JSON Web Token) token, which is a compact, URL-safe way to represent claims transferred between two parties. It's often used in modern web applications and APIs for authentication and authorization purposes. JWT tokens are part of the OAuth 2.0 framework.

A JWT token contains claims (pieces of information like user ID, validity time, roles, or permissions) in a JSON format, encoded and signed digitally. It's composed of three parts separated by dots ("."):

- **Header:** Contains metadata about the token, such as the type of token and the algorithm used for signing.
- **Payload:** Contains the claims. These claims represent the data being transferred (e.g., user information, permissions, validity, etc.).
- **Signature:** Ensures that the token hasn't been tampered with and verifies the sender's authenticity.

Each part is base64-encoded, and the sections are joined together with periods ".", forming a compact token that looks like this:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQssw5c
```

We can decode the token in two ways:

- From command line:

```
$ jq -R 'split(".") | select(length > 0) | .[0],.[1] | @base64d \
| fromjson' <<< euyyrugjgfcuagAguydaftuyfas.fnalfnakg3gj.fsaldj...
```

- Using the webpage: <https://jwt.io/>. Where as well you will find more information about JWT tokens.

The tokens generated by the TokenRequest API will be:

- Audience Bound

- Time Bound
- Object Bound

NOTE

You can use the token to authenticate via curl:

```
$ curl https://<loadbalancer_ip>:6443/api \
-H "Authorization: Bearer <your_token_in_clear>"

$ curl https://<loadbalancer_ip>:6443/api \
-H "Authorization: Bearer $(cat /path/to/tokenfile)"
```

Once the **ServiceAccount** and its **token** are created you can create **Roles** for it and bind this roles using **RoleBindings**.

WARNING

If you want to create a token associated to a service account with non expiring token you can do it creating the following secret object, which as well will create a secret storing the token:

```
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: mysecretname
  annotations:
    kubernetes.io/service-acount.name: <serviceaccount_name>
```

You should only create a service account token Secret object if you can't use the **TokenRequest API** to obtain a token, and the security exposure of persisting a non-expiring token credentials in a readable API Object is acceptable to you.

8.14.3 Namespace ServiceAccount

For every namespace in K8s, a **ServiceAccount** named **default** is automatically created, so each namespace has its own default ServiceAccount. Whenever a Pod is created into the namespace, the default ServiceAccount is associated and its token is automatically mounted to that pod in an special way (projected volume from the kube-api-access). You will find the token inside:

```
/var/run/secrets/kubernetes.io/serviceaccount
```

This default ServiceAccount is quite restricted, it only has permissions to run basic K8s API queries. If you want to use a different ServiceAccount you can specify it on the Pod definition file. As well, the token created for the ServiceAccount will have a restricted lifetime.

8.14.4 Security inside Docker containers

When you run a docker container, you have the option to define a set of **security standards** such as:

- **--user=titocampis**: to define the user inside the container.

- `--cap-add MAC_ADMIN`: to define Linux capabilities.

This can be configured in K8s as well using **securityContext**, consider that:

- If **securityContext** is defined at Pod level will apply to all containers inside the Pod.
- If **securityContext** is defined at container level it will override the **securityContext** defined at Pod level.

How to define **securityContext** at **Pod level**: c

```
apiVersion: v1
kind: Pod
metadata:
  name: my_pod
spec:
  securityContext:
    runAsUser: "titocampis" # Capabilities are not supported at Pod lvl
    ...
  containers:
    - name: ...
```

How to define **securityContext** at **container level**: c

```
apiVersion: v1
kind: Pod
metadata:
  name: my_pod
  ...
  containers:
    - name: ...
      spec:
        securityContext:
          runAsUser: "titocampis"
          capabilities:
            add: ["MAC_ADMIN"]
```

8.15 Network Policies

Egress and ingress traffic refer to the flow of data in and out of a network or device:

- **Ingress Traffic**: This is data coming into a network or a device from an external source, such as incoming requests or data from the internet to your internal servers, systems, or devices.
- **Egress Traffic**: This is data leaving a network or a device, going from inside the network to an external destination, like when your servers or devices send data to the internet or an external server.

In K8s we have a cluster with a set of Nodes hosting a set of pods and services. Inside the cluster, each node has an IP address, as well as does each pod and service. On requisite for K8s is that Pods always need to be able to communicate with each other without having to configure any additional settings like routes. That's why K8s is configured by default with **All Allow rule**, which **allows any traffic from any pod to another pod** inside the cluster.

But as you can imagine, it can be a problem, because imagine that your customers don't want his database to be accessible by other pods of other applications running on the cluster. To do that, you will have

to implement a **Network Policy** to allow traffic to the DB server only from Pods comming from this application.

NetworkPolicy is another namespaced object of K8s. They can be linked to one or more pods, and you can define rules to control the traffic to this Pod, it is like a kind of "**capsule**".

8.15.1 Just ingress

For example, you can define a rule to allow ingress trafic from the `app1_pod` on port 3306. If you define a rule the rest of the egress or ingress trafic will be disabled except of this rule. In order to link NetworkPolicy to Pod you use as well as in ReplicaSets **labels and selectors**:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector:
        matchLabels:
          name: app1_pod
  ports:
  - protocol: TCP
    port: 3306
```

NOTE

If you define **policyTypes** just as Ingress or Egress, **all traffic of the one defined will be blocked** except the rule, the other will become unaltered.

You have a label to enable traffic comming from just a namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              name: app1
      ports:
        - protocol: TCP
          port: 3306
```

You have a label no enable traffic comming from a range of IP's:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
  ingress:
    - from:
        - ipBlock:
            cidr: 192.168.5.10/32
      ports:
        - protocol: TCP
          port: 3306
```

We can, as well, combine them as you wish:

- One rule to allow just resources from namespace `app1` **and** just from `app1_pod`:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: app1
    podSelector:
      matchLabels:
        name: app1_pod
  ports:
  - protocol: TCP
    port: 3306
```

- One rule to allow just resources from namespace `app1` **or** allow ingress traffic from ip's:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          name: app1
    - ipBlock:
        cidr: 192.168.5.10/32
  ports:
  - protocol: TCP
    port: 3306
```

8.15.2 Ingress and Egress

Egress rules can be created alone as well as Ingress rules, but they can be created together, let's see an example:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: db-policy
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
    - Ingress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              name: app1
      ports:
        - protocol: TCP
          port: 3306
  egress:
    - to:
        - ipBlock:
            cidr: 192.168.5.10/32
      ports:
        - protocol: TCP
          port: 3306
```

9 Storage

9.1 Docker Storage

9.1.1 Introduction

To understand the Storage in container orchestration tool like K8s, it is important to first understand how storage works with containers, for example in Docker containers. It makes so much easier to understand how it works in K8s.

When it comes to storage in Docker, there are two concepts you must know about:

- Docker Storage Drivers and File Systems
- Volume Drivers

9.1.2 Where does Docker stores its persistent data?

Let's start with how Docker stores data on the local file system. When you install Docker on a system it creates this folder structure at `/var/lib/docker/`.

```
| - /var/lib/docker
|   | - aufs/
|   | - containers/
|   | - image/
|   | - ...
|   | - volumes/
```

This is where Docker stores all this data by default. Data as:

- **image/**: files with the data related to images stored on the Docker Host
- **containers/**: files with the data related to containers running on the Docker Host
- **volumes/**: files with the data related to any volumes created by the docker containers created

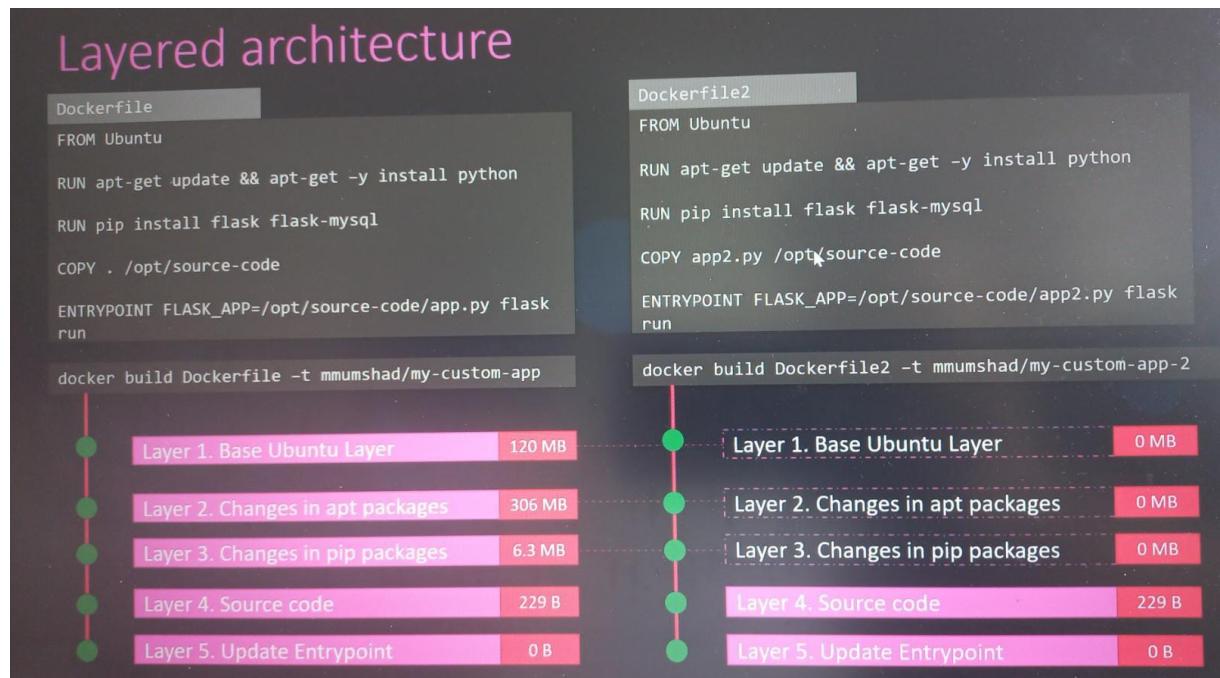
9.1.3 Docker Image Layer Architecture

But how does Docker exactly store the files of an image and containers? We need to understand **Docker's layered architecture**. When Docker builds images it follows a **layered architecture**, each line of instruction in the Dockerfile creates a new layer in the Docker image with just the changes from the previous layer.



Every layer **only stores the changes from the previous layer** it is reflected in the size as well.

To understand the advantages of this layered architecture let's consider a second Dockerfile, which is different but in some parts is common. Using the same image, the same python and flask dependencies, but uses a different source code to create a different application image. When you run the `docker build` command to build a new image for this application, since the first three layers of both the applications are the same, **docker is not going to build the first three layers**, instead it reuses the first 3 layers from the cache (generated with the build of the first Dockerfile). So only creates the last two layers. That is how Docker easily and efficiently saves disc space, and saves us a lot of time during builds and updates.



NOTE

Once the image is **completely build**, you cannot modify the contents of the layers, so they are **Read Only** and you can only modify them by initiation a new build.

When you run a container based on this images with `docker run` command, it adds a new writeable layer on the TOP of the image layer, this layer is created to store data created by the container such as:

- Logs files written by the applications
- Any temporary files generated by the container
- Just any file modified by the user on that container, or by external accessing

The life of this layer though is only as long as the container is alive. So when the container is destroyed, this layer and all of the changes stored on it are also destroyed. Remember that the same image layer is shared by all containers created using this image.

So imagine that we create a container, and in the runtime environment we want to create a file called `temp.txt`. It will be created in the container layer, but it won't take effect over Image Layers, so if you run a container with the same image, this file will not be created. Furthermore, if you for example modify the source code application that was created inside the Image Layer, it will only be modified in the Container Layer, so if you run another container with the same image the modification will not take effect. This is because when you modify some file in the Image Layer, Docker automatically creates a copy of that file in the Container Layer (it is invisible for the user but it happens), so you can play with this file as you want but never modify the image ones.

As we said, when a containers is destroyed, all the data stored in the Container Layer is automatically removed, the change in the app code, or the files we have created, all disappears. That's why we wanted to Persist the Data.

9.1.4 Volumes in Docker

Imagine we want to create a DB container, so we obviously want to preserve the data created in the Container Layer, as long as it is a DB, and users, applications and other servers will be constantly modifying its content. So it has to be resilient to restarts, and independent to the containers created. If the container restarts it must have the data it has when he died, and all the containers running the DB image must have the same data, for data consistency. For that purpose, **Docker has Persistent Volumes**.

9.1.4.1 Volume Mounting

The first step is to **mount a Docker Volume**, to tell Docker to creates a folder:

`/var/lib/docker/volumes/data_volume:`

```
$ docker volume create data_volume
```

Then have to pass it as an argument when you run the container:

```
$ docker run -v data_volume:/var/lib/mysql mysql:11.0.0
```

So this way, we create a Symbolic Link between docker container path and our local path, so every change made by the database container will be written in the local folder. Even if the container is destroyed the data is still here.

NOTE

But what would happen if you use the run command without having created a Docker volume previously?

```
$ docker run -v data_volume_2:/var/lib/mysql mysql1.0.0
```

Docker is so intelligent and will create before running the container the Persistent Volume, and then, create the link between the local folder and the container.

To see all the Persistent Volumes created by Docker:

```
$ docker volume ls
```

Or:

```
$ ls -la /var/lib/docker/volumes/
```

To inspect specific volume:

```
$ docker volume inspect volume_name
```

9.1.4.2 Bind Mounting

But what happen if we have our data already at another location? For example in our local folder, inside the application directory, under /data and we want to share this data as a volume with the container (maybe to do some tests), and not in the default /var/lib/docker/volumes/ folder. In this case we will use the docker run command with -v without creating a Volume, but in this case we must to provide the complete or relative path to the folder we would like to mount:

```
$ docker run -v /users/titocampis/alexwork/appmysql/data:/var/lib/mysql mysql1.0.0
```

```
$ docker run -v data:/var/lib/mysql mysql1.0.0
```

9.1.4.3 Volume Mounting vs Bind Mounting

Volume Mount: It mounts a Volume from the volumes directory.

Binding Mount: It mounts a Volume from any other location in the Docker Host.

9.1.4.4 The correct way: run --mount

NOTE

In the new versions of Docker, the correct way to mount a volume (Bind or not) inside a container is using the option `--mount`

```
$ docker run \
--mount type=bind,source=/data/mysql,target=/var/lib/mysql \
mysql:1.0.0

$ docker run \
--mount type=mount,source=mydata,target=/var/lib/mysql \
mysql:1.0.0
```

9.1.5 Storage Drivers & Volume Drivers

9.1.5.1 Storage Drivers

But who is in charge to these operations? The **Storage Drivers**.

- Maintaining the layered architecture
- Creating a writable layer
- Moving files across layers to enable copy and write
- etc.

Docker uses Storage Drivers to enable layered architecture, some of the common storage drivers are: AUFS, ZFS, BTRFS, Device Mapper, Overlay, Overlay2. The decision of which use depends on the underlying SO (Ubuntu, IOS, Windows, etc.).

9.1.5.2 Volume Drivers

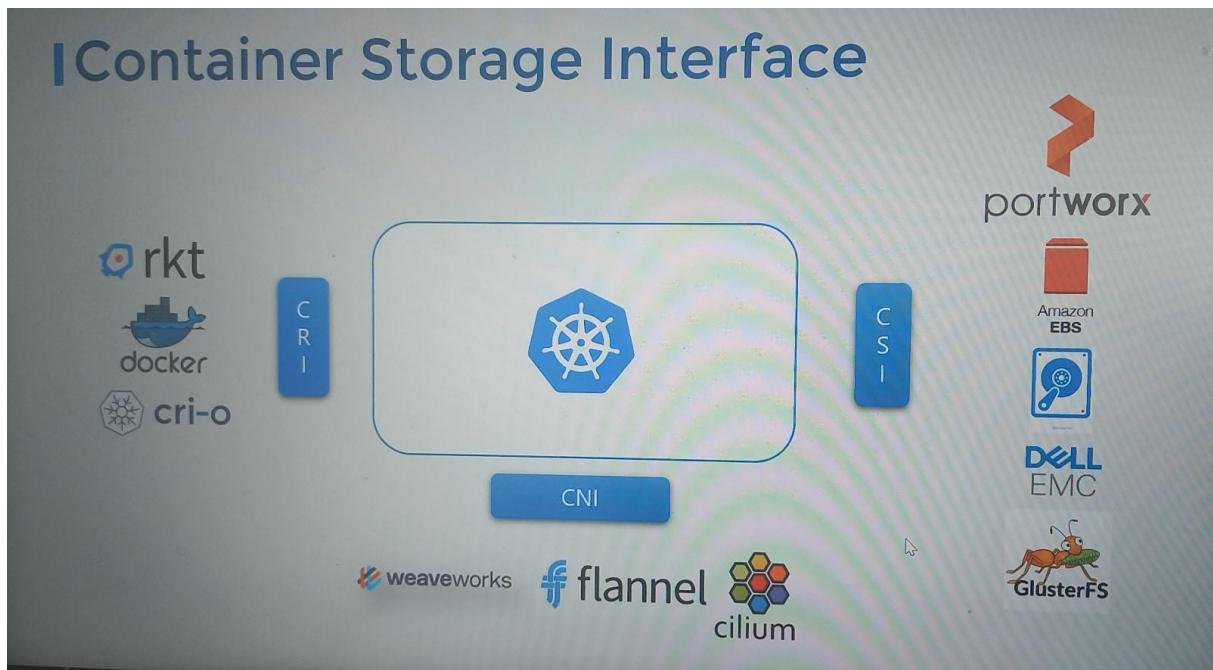
PENDING

9.2 K8s CSI (Container Storage Interface)

In the past K8s used Docker alone as the container runtime engine, and all the code to work with Docker was embedded within the K8s Cluster source code. But with other container runtimes coming in as: Rocket and CRI-O, it was important to open up and extend support to work with different container runtimes and no be dependent on the K8s source code. And that's why **Container Runtime Interface (CRI)** came to be. It is a standard that defines how an orchestration solution like K8s would communicate with container runtimes like Docker, so in the future if a new container runtime interface is developed, they can simply follow the CRI standards and that new container runtime would work with K8s.

It is similar to the extension of CNI (Container Networking Interface), to extend the support for different Networking Solutions. Now any Networking vendors could simply develop their plugin based on the CNI standards and make their solution work with K8s.

Container Storage Interface



So as you can imagine, CSI (Container Storage Interface) was developed to support multiple storage solutions, with CSI you can now write your own drivers for your own storage to work with K8s. Port Works, Amazon EBS, Azure Disk, etc. Everyone's got their own CSI drivers.

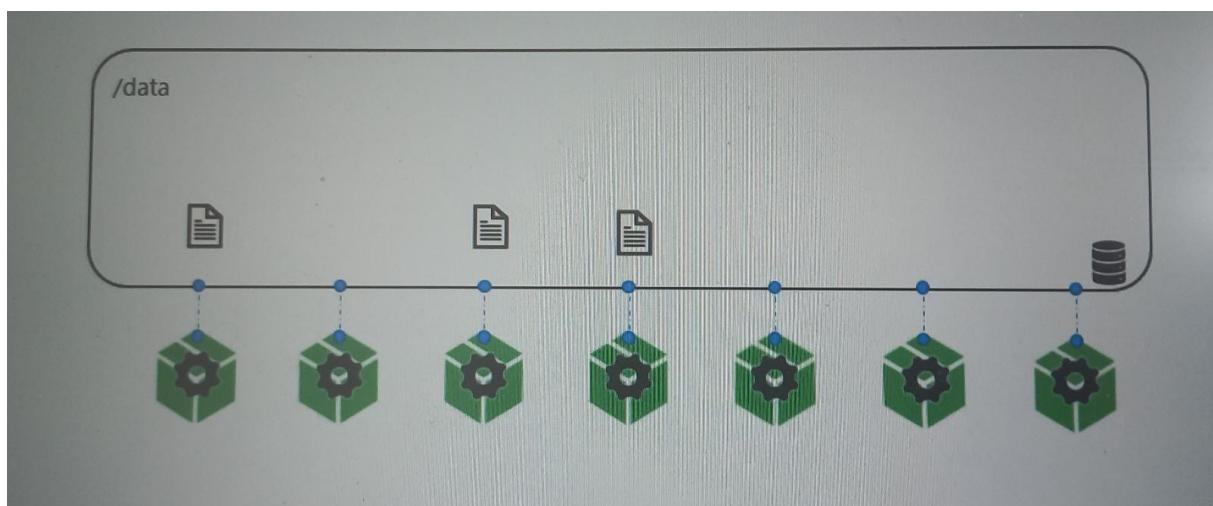
PENDING, you also have to describe a little bit these rules to build your solution and suits K8s

9.3 K8s Volumes & Persistent Volumes

9.3.1 Docker Volumes

Docker containers are meant to be transient in nature, which means they are meant to last only for a short period of time. They are designed to process, receive, modify and do things with this data but destroy all data once finished, the data is destroyed along with the container. To persist data processed by the containers we attach a volume to the container when they are created, as we saw in section 9.1.4.

So the data processed by containers are now stored in this volume, retaining it permanently.



9.3.2 K8s Volumes

How does it works in K8s world? Just as in Docker, the Pods created on K8s are transient in nature, when a pod is created it manages data and when it is destroyed, all its data is destroyed as well. **In order to persist data, we attach Volumes to the Pods**, so data generated by the Pod is now stored in the Volume, and even after the Pod is deleted, the data remains.

9.3.2.1 Creating a K8s Volume

Imagine we have a Pod that runs an image inside a container which has a command that generates a random number and stores it under `/opt/number.out`. If the Pod is generated without a volume, the data will be stored inside the Pod, but when the Pod dies, the data will be removed.

`.spec.volumes`

In order to retain the number generated by the Pod, we create a **volume** inside the Pod in `.spec.volumes`. There are multiple configurations, but for now, we are going to configure it to use a directory on the host. This way all the files stored in the directory on the Pod will be stored in the directory `data` on the Node where the Pod is running.

`.spec.containers.volumeMounts`

But that's not all, we have to mount this volume inside the container in order to container can use it.

```
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
    - image: alpine
      name: alpine
      command: ["/bin/bash","-c"]
      args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
      ...
      volumeMounts:
        - mountPath: /opt
          name: data-volume
  volumes:
    - name: data-volume
      hostPath:
        path: /data
        type: Directory
```

Now yes, the random number will be written in the `/opt/number.out` container directory and in the Node directory `/data`. When the Pod dies, the random number file still lives on the Node.

But you can imagine this is not suitable, because if the Pod restarts and now is allocated in another Node, the data will be lost!! Because it is only on the last Node. For that the solution involves 2 actors:

Use external shared storage solutions to share storage between nodes: K8s supports several types of different shared storage solutions as:

- NFS
- GlusterFS
- Flocker
- Ceph

- AWS
- Google Persistent

9.3.2.2 Example of use external shared storage solution

If we are using the AWS shared storage solution:

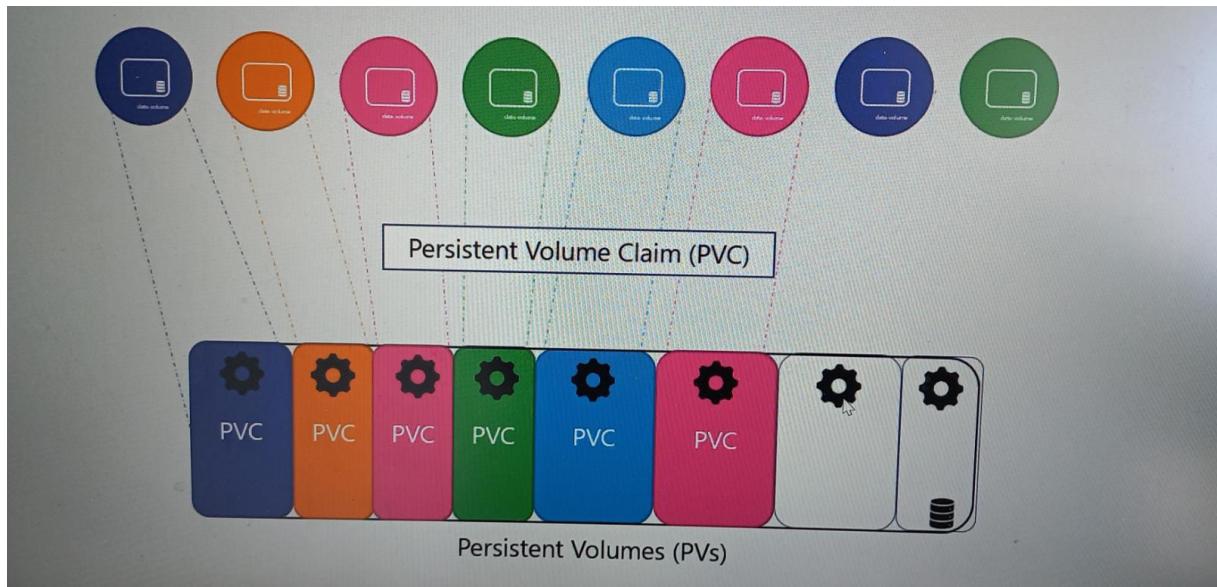
```
...
volumes:
- name: data-volume
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

9.3.3 K8s Persistent Volumes

[Official K8s Doc](#)

It is a way of manage the storage more centrally, you would like it to be configured in a way that an administrator can create a large pool of storage and then have users carve out pieces from it as required. That is where **Persistent Volumes** can help.

A Persistent Volume is a cluster-wide pool of storage volumes configured by an administrator to be used by users deploying applications on the cluster. The users can now select storage from this pool using **Persistent Volume Claims**



9.3.3.1 Creating a Persistent Volume

```
pv-template.yaml

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-log
spec:
  capacity:
    storage: 100Mi
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /tmp/data
```

accessMode

Defines how a volume should be mounted on the hosts, read-only, read-write, etc. The supported values are:

- **ReadOnlyOnce**: the PV will be read-write by a single node, it only can be read from resources inside the same node.
- **ReadOnlyMany**: the PV will be read-only by resources in many nodes.
- **ReadWriteMany**: the PV will be the volume read-write by many nodes.

WARNING

hostPath option must not be used in Production Environments, because it means you are associating the storage of the current Node where you are creating the Persistent Volume.

To avoid that, you can replace with one of the supported storage solutions as we saw in the previous lecture, for example:

```
pv-template.yaml

...
  capacity:
    storage: 1Gi
  awsElasticBlockStore:
    volumeID: <volume-id>
    fsType: ext4
```

```
$ kubectl create -f pv-template.yaml
```

```
$ kubectl get pv
```

9.4 K8s Persistent Volume Claims

[Official K8s Doc](#)

As we saw in the previous lecture, **Persistent Volume Claims** are the K8s objects that makes **Persistent Volumes** available on the Nodes. PV's and PVC's are two separate objects in the K8s Namespace. An administrator creates a set of PV's, and a user creates PVC's to use the storage.

WARNING

PV's are not attached to any Namespace, they are across all Namespaces, but it can have permissions of view depending on the role. It is recommended to only have read-write access to this objects by Cluster Administrators.

Once the PVC's are created, K8s binds (vincula) the PV's to claims based on the request and properties set on the volume. Every PVC must be bounded to a single PV, during the binding process, K8s tries to find a PV that matches the following requests of the PVC's:

- has sufficient capacity
- has same access modes
- has same volume modes
- has same Storage Class

However, if there are multiple possible matches for a single claim, and you would like to specifically use a particular PV, you could still use **labels and selectors** to bind to the right PV's.

WARNING

Remember the relationship between PV's and PVC's is one to one, so there cannot be more than one PVC bound to a PV. If the PVC requires less storage than the PV storage, no other PVC's can use the remaining capacity in the volume. If there are no available Volumes that match the PVC, it will remain in a Pending State until new volumes are made available on the Cluster, and when that happens, the PVC is automatically bounded.

9.4.1 Creating a Persistent Volume Claim

```
pvc-template.yaml

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 500Mi
```

```
$ kubectl get pvc
```

NOTE

If we pay attention on what we see when get pvc, we can see the PV attached to that pvc.

WARNING

If the **accessModes** do not match, the **PVC** will not be binded to the **PV**.

9.4.2 Deleting Persistent Volume Claims

Just use the `kubectl delete pvc <name>` command. But what happens with the underlying PV? You can choose what is to happen to the PV. By default it is set to retain, meaning the PV will remain until it is manually deleted by the administrator and it is not available for reuse by any other claims anymore, it is so romantic.

Also, you can change the default configuration:

```
pv-template.yaml

apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-vol1
spec:
  accessModes:
    - ReadWriteOnce
  capacity:
    storage: 1Gi
  hostPath:
    path: /tmp/data
  persistentVolumeReclaimPolicy: Recycle
# Default persistentVolumeReclaimPolicy: Retain
# persistentVolumeReclaimPolicy: Delete
```

If you use **Delete** option the PV will be deleted as soon as the PVC is deleted, freeing up storage on the end storage device. If you use **Recycle** the data in the data volume will be scrubbed before making it available to other claims.

9.4.2.1 Associating PVC's to Pods to containers can consume them

Once you create a PVC use it in a POD definition file by specifying the PVC Claim name under `persistentVolumeClaim` section in the `volumes` section like this:

```

apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
    - image: alpine
      name: alpine
      command: ["/bin/bash", "-c"]
      args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
      ...
  volumeMounts:
    - mountPath: /opt
      name: data-volume
  volumes:
    - name: data-volume
      persistentVolumeClaim:
        claimName: myclaim

```

WARNING

It is not necessary to create a PV object to share storage bare between a Pod and the Node. It's enough to create volume in the specs of the Pod an a volumeMount in the container, and you will do the same it does docker, in your system.

But this only applies to bare shared storage between a Node an a Pod, if you want to persist the storage between all Nodes it is more complex, so you have to create PV's and PVC's.

WARNING

If there is storage left over in the PV, the PVC will get it all.

WARNING

The PVC must be in the same Namespace as the Pod, if it is not like that, it won't work.

WARNING

If you try to delete a PVC used by a Pod, it won't be definitely deleted until the Pod dies. It will remain in Terminating state.

What happens if we modify, remove, or add files to the shared directory?

Wow, if you add files in the local directory they are added automatically to the pod:

Y lo mismo sucede al revés!!! WOOOOOW!!! Quedan automágicamente conectados para siempre. Amigos para siempre do you want to be my friend...

9.5 Storage Class

In the previous lectures, we discussed about how to create PV's and then create PVC's to claim that storage, and then use the PVC's in the Pod definition files as volumes. The problem here is before the PV's are created disk storage must be created on the Cloud Platform (Google, Azure, IBM, AWS, etc.). So every time an application requires storage it is needed to manually provision the disk and after create

manually a PV using the definition file using the same name as the disk you have already created. This is called static provisioning volumes.

It will be amazing if the volume is provisioned automatically when the application requires it, and that is where **StorageClass** come in. With **StorageClass** you can define a provisioner like google storage which automatically provision storage on the cloud and attach that to Pods when the claim is made. It is called dynamic provisioning on Pods. You do that by creating an Object **StorageClass**.

```
sc-template.yaml

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: object-storage
provisioner: kubernetes.io/gce.pd
```

We use the provisioner `kubernetes.io/gce.pd` to create a Volume on GCP. There are other any provisioners as well, as AWS EBS, Azure File, Azure Disk, Portworx, etc. With each of these provisioners, you can pass in additional parameters such as the type of disk to provision, the replication type, etc. These parameters are very specific to the provisioner that you are using.

If we use **StorageClass** we no longer need the PV definition, because the PV and any associated storage is going to be associated automatically when the StorageClasses created. But we have to let the PVC know about the StorageClass, so:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: claim-log-1
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: <name-of-storage-class>
  resources:
    requests:
      storage: 50Mi
```

With this definition, the **StorageClass** associated to the PVC uses the defined provisioner to provision a new disk with the required size.

WARNING

The `storageClassName` works similarly to `accessMode`, if the PVC has one defined, it cannot be bound to a PV that has another one defined or which does not have any defined. PV and PVC `accessMode` must match.

If the PVC has no `accessMode` it cannot be assigned to a PV defined with an `accessMode`.

WARNING

If the Storage Class makes use of `VolumeBindingMode` set to `WaitForFirstConsumer`, it will delay the binding and provisioning of a PersistentVolume until a Pod using the PersistentVolumeClaim is created.

9.5.1 Different Storage Classes

One thing that is common is to use different **StorageClasses** depending on the package that the enterprise or applications purchase. For example:

I Storage Class

```
sc-definition.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: silver
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
  replication-type: none
```

```
sc-gold-definition.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: gold
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  replication-type: none
```

```
sc-platinum-definition.yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: platinum
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  replication-type: regional-pd
```

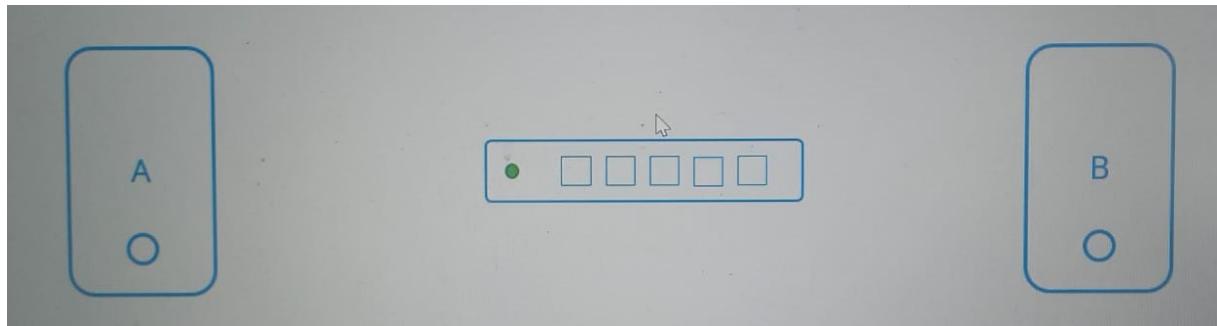
Silver SC Gold SC Platinum SC

10 Networking Basis in Linux (not K8s)

10.1 Switching and routing in Linux

10.1.1 Switching

What is a network? We have 2 Linux Machines, A and B, it can be: laptops, desktops, VMs on the Cloud, wherever. How does system A reach system B? So we connect them to a **switch** and it creates a network containing the two systems.



To connect the 2 Linux machines to a switch, we need an interface on each host (physical or virtual depending on the Host). And to see the interfaces from the Host, we use the following command:

- View and managing link-layer (MAC-level) settings.

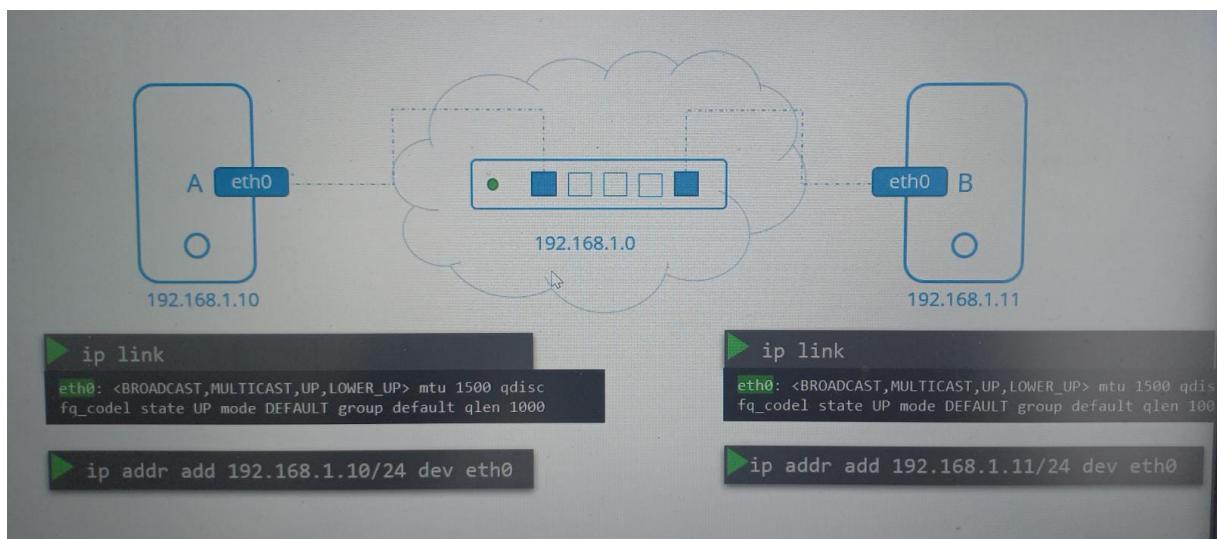
```
$ ip link
```

- Check network-layer (IP-level) details along with link-layer information.

```
$ ip a
```

We only want the information of **eth0** interface, which it will be used to connect the machine to the switch.

Let's assume it's a Network with the address 192.168.1.0, so we then assign the system with IP addresses on the same Network:



To add an IP address (192.168.1.10/24) to the network interface eth0

```
$ ip addr add 192.168.1.10/24 dev eth0
```

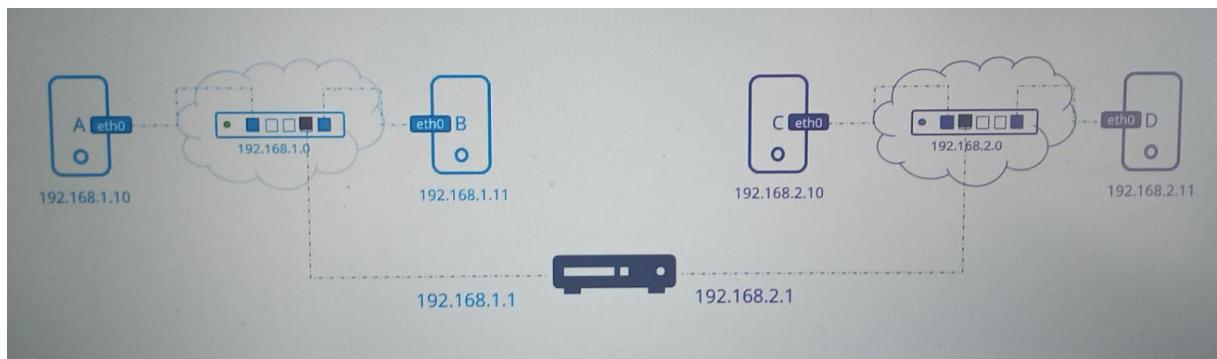
Once the links are up and the IP addresses are assigned, the computers can now communicate with each other through the switch. The switch can only enable communication within a Network, which means it can receive packets from a Host on the Network and deliver it to other systems within the same Network.

10.1.2 Routing

Imagine we have another 2 machines in another Network at address 192.168.2.0 and system address according to this address. How does a system in one Network reach a system in another Network?

Using **Routers**, a **Router** is an intelligent device which can connect two networks together. It is like another server with a lot of **Network Ports**. Since it connects to the two separate Networks it gets 2 IPs assigned, one on each Network, for example:

- For the first Network, it receives the IP: 192.168.1.1
- For the second Network, it receives the IP: 192.168.2.1



Imagine that system A wants to communicate with system C. How does the system A know where the Router is configured on the Network to send the packages through? The router is just another device set on the Network with an associated IP. This is why we configure the systems with a **gateway (route)**. If the Network was a room, the **gateway** would be the door to the external world (other Networks). The systems need to know where that door is.

See existing routing configuration on a system: it displays the kernel's routing table

```
$ route
```

If nothing is configured, system A cannot go outside its Network, so it cannot communicate with system C or D, only with D.

To configure a gateway in system A:

```
$ ip route add 192.168.2.0/24 via 192.168.1.1
```

Specifying that to reach 192.168.2.0/24 Network, it should pass through 192.168.1.1 (the router).

As you can imagine, we have to do the same for system B, to reach system C and system D. And the same for systems C and D to reach systems A and B. Configuring a route on the Network and configure routes on the systems.

Now suppose these systems need access to the internet, for example to google at 172.217.194.0. You only need to connect the router to the internet and add a new route to the systems, to pass through router to go to Google (172.217.194.0). But there are so many different sites on different Networks on the internet, so instead of adding a routing table entry for the same router's IP address, for each of those Networks, we can simply say for any Network that you do not know where to route, use this router as the default gateway. And that's the key.

```
$ ip route add default via 192.168.2.1
```

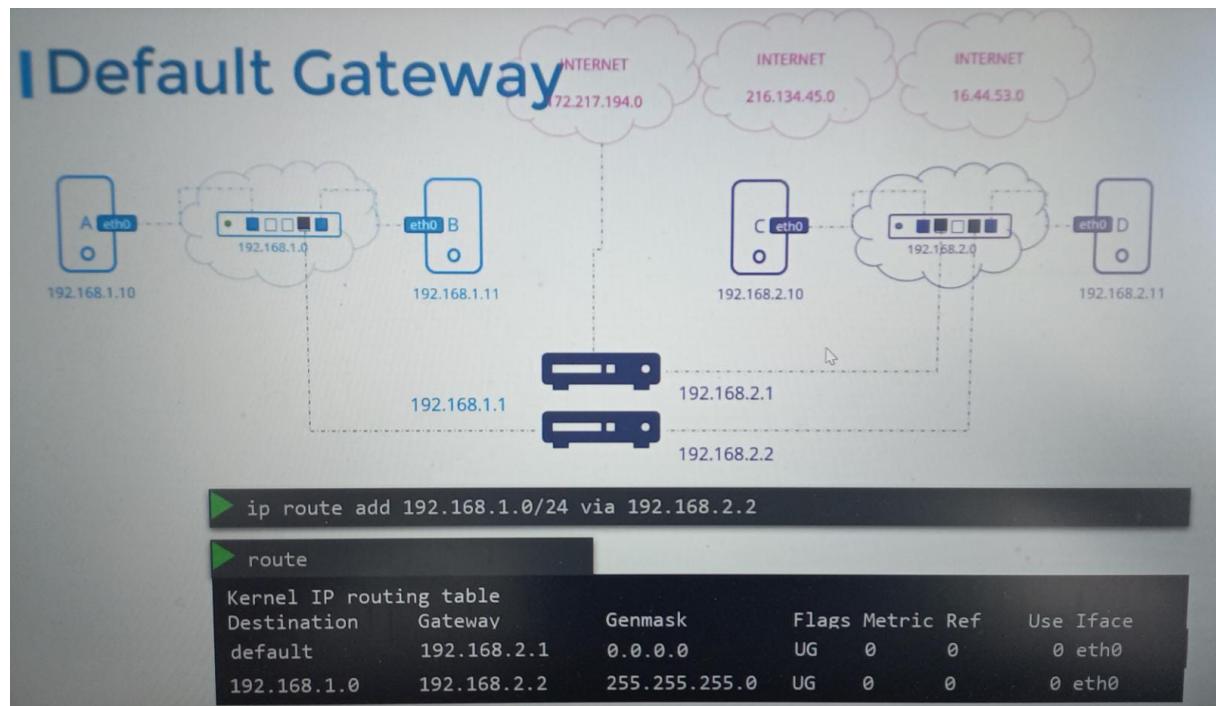
NOTE

Instead of the word "default" we can use 0.0.0.0, which means any IP destination.

This way, any request to an IP outside of your existing Network (not defined in route table) goes to this particular router.

10.1.3 Private and Public Networks configuration

So in a simple model like this, all you need is a single routing table entry with the default gateway set to the router's IP address. But if we want to manage connections with another private Network at the same time than Public Network (as internet), it gets complicated. We should have 2 routers with 2 route rules in the configured system tables, to use different routers to send the traffic depending on the Network systems want to reach.



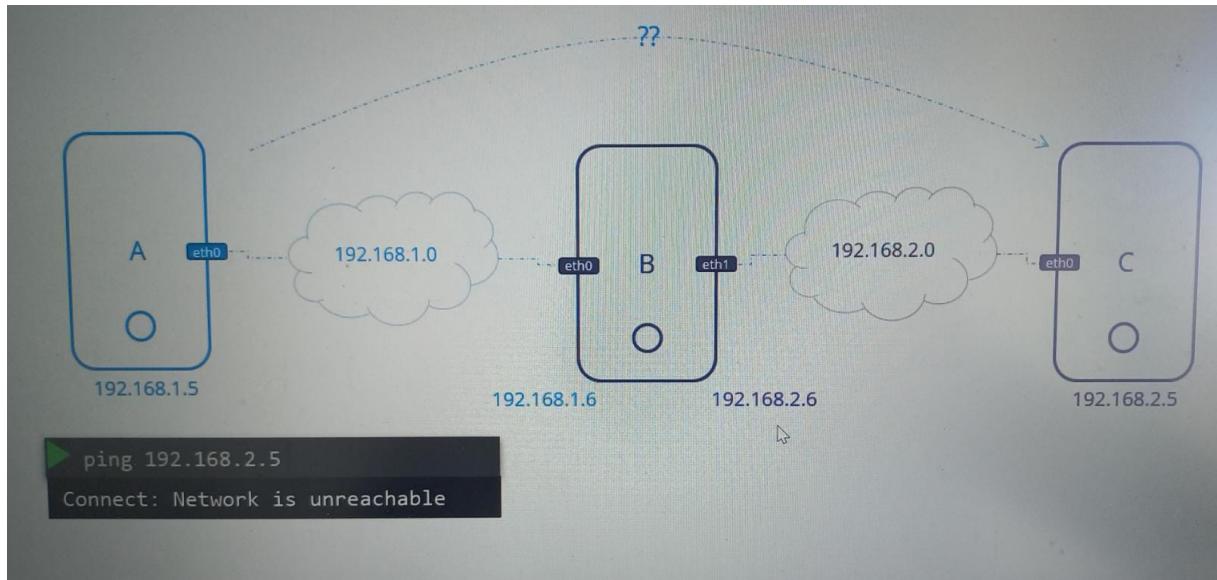
- One rule to connect to internal private Network on router 1

```
$ ip route add 192.168.1.0/24 via 192.168.2.2
```

- Another rule to connect to all other Networks (including internet)

```
$ ip route add default via 192.168.2.1
```

10.1.4 Using a Linux System as a router



As you know, Host A has no idea of how to reach Host C, because it is into another Network. So we have to say Node A to reach Host C through Node B. So we do that adding a routing rule in the table of Host A:

```
$ ip route add 192.168.2.0/24 via 192.168.1.6
```

When the Host C wants to response Host A will have the same issue, so we have to do the same on Host C:

```
$ ip route add 192.168.1.0/24 via 192.168.2.6
```

If we try to ping now, we no longer get the Network unreachable error message, that means our routing entries are right, but we still do not get any response back. This is because by default, in linux, packets are not forwarded from one interface to the next. For example: packets received on Eth0 of Node B are not forwarded to elsewhere through Eth1, for obviously security reasons. But in this case, that we know they are both secure Networks, and we want to use Node B as router, we can allow Host B to forward packets from one Network to the other. It is set in a file on the system config:

```
$ cat /proc/sys/net/ipv4/ip_forward
```

- **0:** No forward
- **1:** Forward

```
$ echo 1 > /proc/sys/net/ipv4/ip_forward
```

WARNING

This way of configuration is not resilient to reboot, so if the system is reboot the configuration will be lost. To persist the configuration you should configure that in /etc/sysctl.conf adding:

```
$ sudo vim /etc/sysctl.conf
#####
...
net.ipv4.ip_forward=1
...
```

10.2 DNS in Linux

10.2.1 Introduction to DNS

We have 2 Hosts in the same Network, Host A and Host B. And they have been assigned a Network with IP: 192.168.1.0. So they are 192.168.1.10 and 192.168.1.11. So as we saw in the last section 10.1. As they are in the same Network, they can reach each other using their IP's. But it can be so tediously for Hosts to know all the IP's, so instead of having to remember the IP address of system B, Host A can give it a name, for example db, that when you ping db it works like if it was using IP.

Basically we want to tell Host A that system B at IP address at 192.165.1.11 has a name db. So when db name is used, for Host A must be the same as 192.165.1.11. It can be done by adding an entry in the file /etc/hosts:

```
$ echo "192.168.1.11      db" >> /etc/hosts
```

Or vim and add at the end of the file the relationship.

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

So now Host A translates db as 192.165.1.11, this is called **name resolution**, associate a hostname to an IP. Whatever we put in the /etc/hosts is the source of true for the Host. But that may not be the truth, maybe Host B is called host-2, but Host A does not care, it will follow what is in the /etc/hosts file.

You can make a test, associate `www.google.com` hostname to IP of Host B and ping, the ping will be answered by Host B. You can have any names as you want for any servers as you want in the /etc/hosts file. Commands look into this file:

- ping
- ssh
- curl
- nc
- nmap
- mtr
- ...

10.2.2 DNS server

In a system with few systems, you can get away with the entries of /etc/hosts file, and this is how it was done in the past. However, when the environment grows and these files got filled with many entries you can imagine this is unmanageable. If one of the Hosts changes its IP, you are done, gj wp. This is

the reason why it was decided to **move all this configuration into a single server**, who will manage all this centrally, called the **DNS server**. And then we point all hosts to look up that server if they need to resolve a hostname to an IP address instead of its own `/etc/hosts` files.

How do we do that? How do we point our host to a DNS server? Imagine our DNS server has the IP 192.168.1.100, every host has a DNS resolution configuration file at `/etc/resolv.conf`. So you have to add an entry into it specifying the address of the DNS server.

```
$ echo "nameserver      192.168.1.100" >> /etc/resolv.conf
```

Or vim and add it at the end of the file.

```
$ sudo vim /etc/resolv.conf
```

Once this is configured into a Host, everytime it comes up across a hostname that it does not know about, it looks it up from the DNS server. With this configuration, if the IP of any of the host was to change, you only need to change it in one file in one Host.

WARNING

You can still have entries in your `/etc/hosts` file, but it is not hardly recommended because it will override the configuration in `/etc/resolv.conf` nameserver. Only use it when you are sure of what you are doing.

But this is default configuration, this order can be changed in the file `/etc/nsswitch.conf`

```
$ cat /etc/nsswitch.conf
```

OUTPUT: hosts: files dns

To change this, you should modify the content on the field to invert it.

What happens if the hostname is not found in `/etc/hosts` neither in `/etc/resolv.conf`, so it will fail. Unless you configure in the file `/etc/hosts` of the **DNS server** to forward all unknown hostnames to the public name server on the internet (ex: 8.8.8.8).

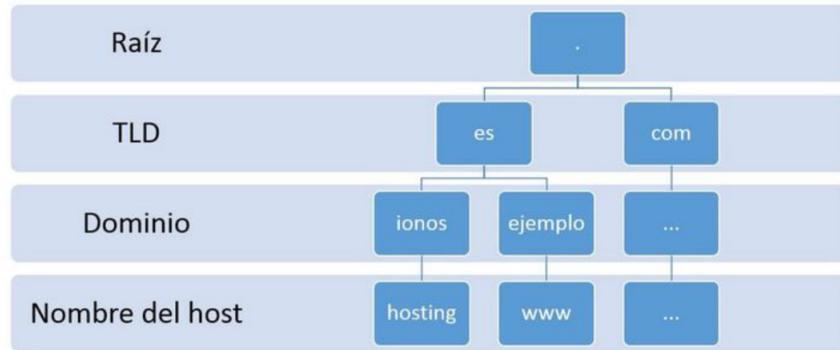
```
$ echo "Forward All to 8.8.8.8" >> /etc/hosts
```

10.2.3 FQDN

So far, we are naming with names like db, host-1, host-2... But the real services are called: www.google.com, www.github.com, etc. They are using **FQDN's**.

FQDN (Fully Qualified Domain Name) or **Domain Name**, which are a complete and unique direction needed to have presence on the internet. It is composed by the name of the host and the domain, and it is used to request to specific hosts on the internet using the name resolution instead of their IP.

Let's see a FQDN structure in detail:



1. **TLD (Top Level Domains):** they represent the intent or the geographical situation of the website:

- **.com:** for commercial or general purpose
- **.net:** for network
- **.edu:** for educational organizations
- **.es / .cat / .fr:** for geographical situation
- **.org:** for non-profit organizations

2. **Domain:** it is the hostname assigned to Google

3. **Name of the host:** is a subdomain, which helps in further grouping things together under the Domain, for example:

- **maps.google.com:** the maps of google
- **drive.google.com:** the drive of google
- **www.google.com:** google search
- **mail.google.com:** google email

But how does this magical works in all Hosts in the world, for example the PC we are using. This is because when your system tries to reach any of these **FQDN's** your system first check at your `/etc/hosts` or **internal DNS server**, and it normally does not know who apps or google is, so it forwards your request to **your internet provider DNS server (you can change this on your machine configuration)**. Then the DNS Server looks at your request and forwards it to the correct host.

Orange DNS (France)

- Primary DNS: 8.8.8.8
- Secondary DNS: 8.8.4.4

Google Public DNS

- Primary DNS: 8.8.8.8
- Secondary DNS: 8.8.4.4

Cloudflare DNS

- Primary DNS: 1.1.1.1
- Secondary DNS: 1.0.0.1

OpenDNS

- Primary DNS: 208.67.222.222
- Secondary DNS: 208.67.220.220

Quad9 DNS

- Primary DNS: 9.9.9.9
- Secondary DNS: 149.112.112.112

Comodo Secure DNS

- Primary DNS: 8.26.56.26
- Secondary DNS: 8.20.247.20

But as this is so tedious, your **local DNS server (router)** may choose to cache this IP for a period of time, typically few seconds up to few minutes. That way, it does not have to go through the whole process again each time.

10.2.4 Creating own FQDN

Imagine we want to expose a service to the internet with different sub-services, like a pay service, a consult service, a book service, a message service, etc. And do you want to access them on the internet by FQDN names and not IP's, so common. You must have an **internal DNS server**, and you must configure the hostname-IP relationship in its `/etc/hosts` file.

So when you want to access your service from inside the Network from any host, the host at first will look in its local configuration file `/etc/hosts`, then if nothing finds it will look in the **DNS server** configured in the `/etc/resolv.conf`, which will tell you that `pay.myapp.com` is the IP 192.68....

But we can one step further, inside my private network, maybe I want to use the service without the FQDN, because it is very long, maybe I want to use it by web, it is easier. Is this possible? The answer is yes, you must make an entry into your host's `/etc/resolv.conf`:

```
search mycompany.com
```

Next time you try to ping web, you will see it actually tries `web.mycompany.com`. As if you try pay, it will try `pay.company.com`.

10.2.5 Record Types (Tipos de Registros)

How are the records stored in the DNS server?

- **hostname to IP:** A records
- **hostnames to IPv6:** AAAA records
- **Mapping one name to another name:** CNAME records. For example, you may have multiple aliases for the same application (`maps.google.com = car.google.com, walk.google.com, cycle.google.com, etc.`) and that's why a CNAME record is used, name to name mapping.

10.2.6 Nslookup or dig

One of the best tools for **DNS name resolution** is **nslookup**. Instead of ping, you can use the binary nslookup to query a hostname from a DNS server, but remember that nslookup does not consider the entries in the local `/etc/hosts` file. Nslookup only queries the DNS server. The same happens with the binary **dig**, another useful tool to test DNS name resolution

10.3 Network Namespaces in Linux

10.3.1 Introduction to Namespaces in Linux

Network Namespaces in Linux are used by containers like Docker to implement Network isolation, as we know **containers are separated from their underlying host using Namespaces**, but what are Namespaces? You can imagine Host is a house, and the Namespaces are the different rooms you assign to each of your children, to have certain privacy and independence, so the kids can only see their own room, they cannot see what happens in other room or in the rest of the house (they are isolated). However, as a parent, you have visibility into all the rooms in the house as well as other areas of the house. if you wish, you can establish connectivity between two rooms in the house.

When you create a container you want to make sure that it is isolated, that it does not see any other processes on the host or any other containers, so we create a special room for it on our host using a Namespace, so the container only sees the processes run by it and thinks that it is on its own host XD. Nevertheless, the underlying host has visibility into all of the processes including those running inside containers. This can be seen listing system processes.

- If you list the system processes on a **running container**, you will see only the processes executed by the container and the container process itself

Inside Docker Container

```
$ ps aux
```

- If you list the **system processes** on the underlying host you will see all the other processes running in the host (a lot)

On the host directly

```
$ ps aux
```

10.3.2 Namespaces for Networking

When it comes to networking, our host has its own interfaces that connect to the LAN (Local Area Network), as well as the routing and ARP tables with information about resources of the Network. We want to seal all of those details from the container, so when the container is created, we create a Network Namespace for it, that way it has no visibility to any network-related information on the host. Within its Namespace, the container can have its own virtual interfaces, routing and ARP tables.

If you want to create a new Network Namespace on a Linux host:

```
$ ip netns add <net_namespace_name>
```

To list the Network Namespaces:

```
$ ip netns
```

If you want to execute a command inside a Network Interface:

```
$ ip netns exec <net_namespace_name> <command>
```

For example to check the isolation of the network namespace:

- To see the ARP table inside the network namespace:

```
$ ip netns exec <net_namespace_name> arp
```

- To check the network interfaces inside the network namespace:

```
$ ip netns exec <net_namespace_name> ip link
```

- To check the routing network inside the network namespace:

```
$ ip netns exec <net_namespace_name> route
```

To see the ARP table:

```
$ arp
```

NOTE

The ARP (Address Resolution Protocol) table is a crucial component of network communication, specifically within local networks. It functions as a mapping between IP addresses (logical addresses used in Layer 3 of the OSI model) and MAC addresses (physical hardware addresses used in Layer 2).

When devices on a network communicate, data is transmitted using MAC addresses at the hardware level. However, applications and services use IP addresses. The ARP table helps bridge this gap by maintaining a lookup table that translates an IP address into the corresponding MAC address of a device within the same local network.

So as they are networking isolated, they have no network connectivity, no networks on their own. The first think we should do is establish connectivity between namespaces using a **virtual internet pair (virtual ethernet veth)**: So create the veth:

```
$ ip link add veth-<net_namespace_name1> type veth peer  
      name veth-<net_namespace_name2>
```

Associate it to the network namespaces:

```
$ ip link set veth-<net_namespace_name1> netns <net_namespace_name1>
```

```
$ ip link set veth-<net_namespace_name2> netns <net_namespace_name2>
```

Assign IP address to each namespace

```
$ ip -n <net_namespace_name1> addr add <IP> dev veth-<net_namespace_name1>
```

```
$ ip -n <net_namespace_name2> addr add <IP> dev veth-<net_namespace_name2>
```

Bring up the interfaces:

```
$ ip -n <net_namespace_name1> set veth-<net_namespace_name1> up
```

```
$ ip -n <net_namespace_name2> set veth-<net_namespace_name2> up
```

The namespace will be able to see each other.

But what you would do when we have more than 2 namespaces and we have a lot of namespaces? We will need to create a **Virtual Network** inside the host. To create a Virtual Network:

- At first you need to create a **Virtual Switch**, which for our host will be just another interface.

```
$ ip link add v-net-0 type bridge
```

```
$ ip link set v-net-0 up
```

- Then connect the namespaces to the Virtual Network Switch.

```
$ ip link add veth-<net_namespace_name1> type veth peer  
name veth-<net_namespace_name1>-bridge
```

11 Networking in K8s

11.1 Docker Networking

Same as in the last section, this section is very complex and it only helps to understand how docker manage the network, for more information check the video assoiated.

11.2 CNI (Container Networking Interface)

As we have seen in the previous lecture, isolate container Network is so hard, and this task is carried out by every software of containerization like: docker, rkt, mesos, K8s. All of them **associate containers to a Bridge Network (Namespace)** using the Bridges and IP / Routing isolation concepts we saw in the previous lecture.

So, someone smart, thought to externalize the functionality, and create a binary called **bridge** which creates an isolated Namespaces automatically with only one command. This binary is used by K8s, docker, mesos, etc. When a container starts, the binary is executed to create a Bridge Network (Namespace) where the container will run.

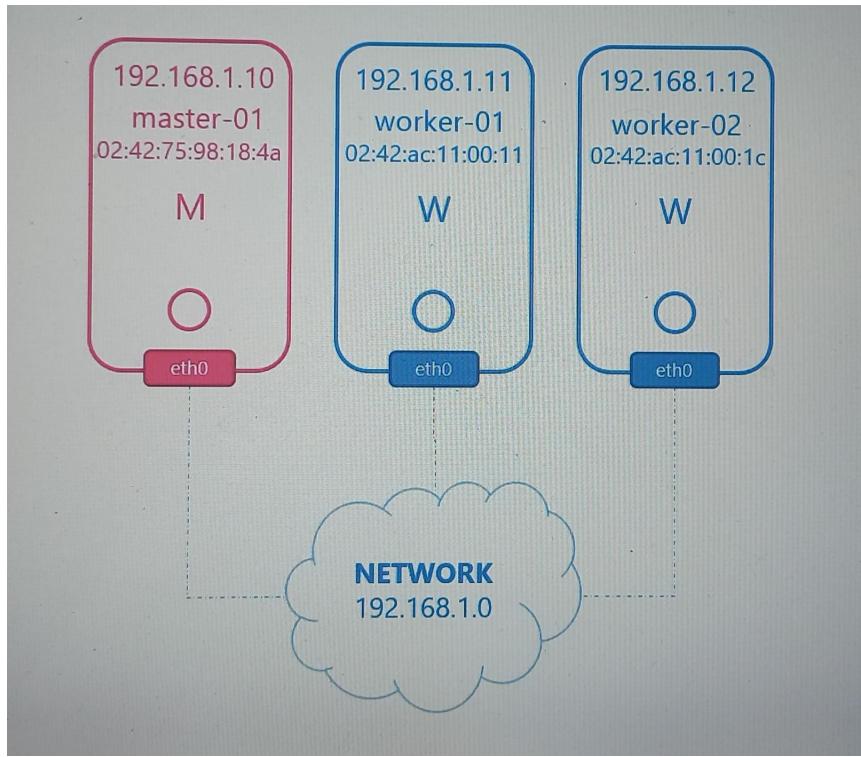
But this program is not closed, it has some configurations for the user to configure its own custom networks. That's where CNI (Container Network Interfaces) comes in. CNI is a set of standards that define how programs have to be defined to solve networking challenges in a container runtime environments. The programs are referred to as plugins, Bridge program is a plugin for CNI.

CNI defines a set of responsibilities for container runtimes and pluggins. All the container manage systems can use these CNI's to define their networks, except Docker. **Docker does not implement CNI**, Docker has its own set of standards known as CNM which stands for container network model which is another standard similar to CNI but with some differences.

So when K8s run a docker container it runs it with **non-Network** and then invokes the configured CNI plugins who take care of the rest of the configuration

11.3 Networking Cluster Nodes

The K8s Cluster consists of Master and Worker Nodes, each node must have at least one interface connected to a Network and each interface must have an address configured. The hosts must have a unique host name set, as well as a unique MAC adress.



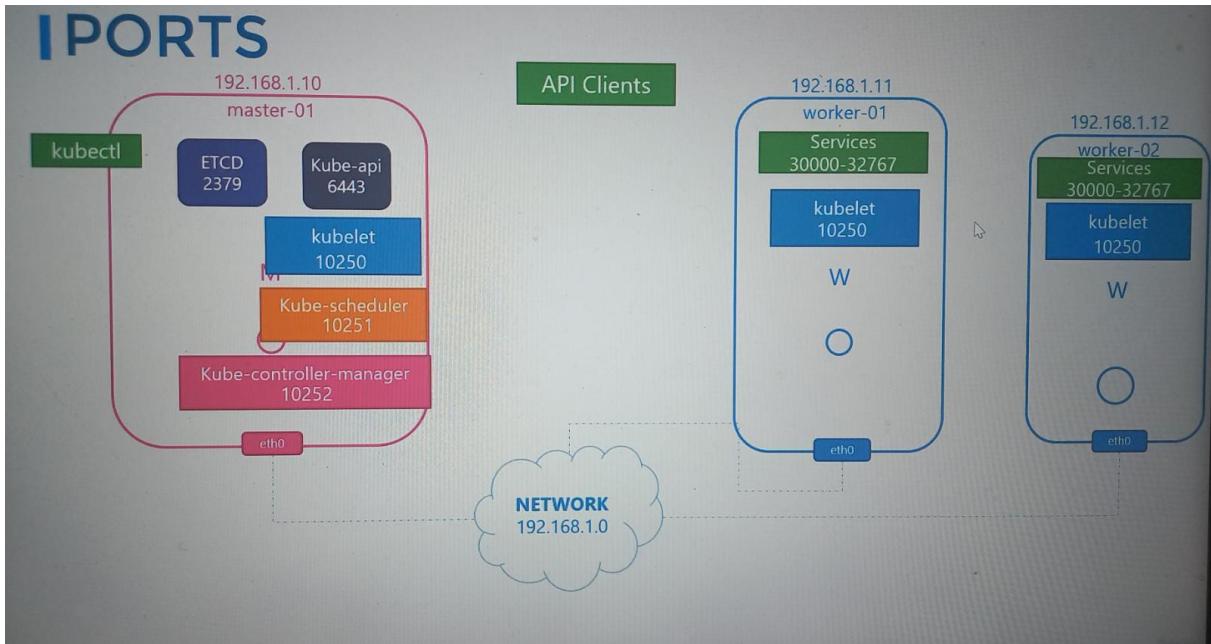
Also, the Nodes must have the following ports opened:

Worker Nodes

- **10250:** kubelet
- **30.000 - 32.767:** services

Master Node

- **6443:** Kube-apiserver
- **10250:** kubelet
- **10251:** Kube-scheduler
- **10252:** Kube-controller-manager
- **2379:** etcd-cluster



NOTE

Yes, a kubelet can be present on the Master Node. And if you have multiple Master Nodes all these ports need to be open those as well, and also need a **port 2380** to connect both **etcd-clients**

11.3.1 Useful Commands

```
$ ip link
```

```
$ ip addr
```

```
$ ip addr add 192.168.1.10/24 dev eth0
```

```
$ ip route
```

```
$ ip route add 192.168.1.0/24 via 192.168.2.1
```

```
$ ip route add default via 192.168.2.1
```

```
$ cat /proc/sys/net/ipv4/ip_forward
```

```
$ arp
```

NOTE

The ARP (Address Resolution Protocol) table is a crucial component of network communication, specifically within local networks. It functions as a mapping between IP addresses (logical addresses used in Layer 3 of the OSI model) and MAC addresses (physical hardware addresses used in Layer 2).

When devices on a network communicate, data is transmitted using MAC addresses at the hardware level. However, applications and services use IP addresses. The ARP table helps bridge this gap by maintaining a lookup table that translates an IP address into the corresponding MAC address of a device within the same local network.

```
$ netstat -plnt
```

PENDING

11.4 POD Networking Concepts

As we have seen, Master and Worker Nodes are configured to share the same Network, so they can reach each other. We also made sure the firewall and network security groups are configured correctly to allow for the K8s control plane components to reach each other. Supposing we have also set up all the K8s control plane components such as the kube-apiserver, etcd servers, kubelets, etc. Before applications can run, there is something that we must address. We have talked about the Network that connects the Nodes, but there is also another layer of networking that is crucial to the clusters functioning, and that is the networking at the Pod layer. Our K8s cluster is soon going to have a large number of Pods and services running on it, how are the Pods addressed? How do they communicate each other? Internally inside the cluster and externally outside the cluster?

These are challenges that K8s expects you to solve, as of today, K8s do not come with a built-in solution for this. It expects you to implement a Networking Solution that solves this challenges. Nevertheless, K8s have laid out clearly the requirements for Pod Networking:

- Every Pod have an IP Address ...

More information in the video section.

PENDING It is so complicated, continue when you have time.

A nivel general me quedo con el concepto de que en cada nodo se monta una red privada, rollo bridge a la cual se conectan todos los pods que en el nodo corren. Cada pod toma una IP de esa red privada y mediante esa red se comunican todos los pods que se ejecuten en un nodo. Para que los pods puedan comunicarse con pods de otros nodos, se establece un router a nivel de la red privada del cluster, que es otra red diferente a la red de los nodos, pero que se indica a cada nodo que reenvien las peticiones hacia esa red si no encuentran el host, y a partir de ahí, esta encamina hacia el resto de Nodos.

11.5 CNI in K8s

CNI defines the responsibilities of container runtime.. Continua el rollazo

PENDING

11.6 WeaveWorks Solution (based on CNI)

PENDING

11.7 IPAM (CNI)

PENDING

11.8 Service Networking

11.8.1 Introduction

IN previous lectures we have talked about Pods Networking, how bridge networks are created within each node and how pods get a namespace created for them, and how interfaces are attached to those namespaces, and how Pods get an IP address assigned to them within the subnet assigned for that Node.

We also saw through routes, we can get the Pods on different Nodes to talk to each other forming a large Virtual Network where all Pods can reach each other. But it is so rare to configure Pods to communicate directly with each other (todo este rollazo para nada, dios mio). **Normally, when you want a Pod to access apps hosted on another Pod you would always use a Service object.**

So, if a Pod wants to consume an app, it only have to use the IP assigned to its Service or the Service name. It does not matter if it a Pod from the same Node, because when a Service is created it is accessible from all Pods on the Cluster. Because a Service is hosted across the cluster, not bound to a specific Node.

WARNING

All the Services are accessible by each resource inside the cluster by its name or its IP, but it is not accessible from outside the Cluster, because its IP belongs only to the Virtual Internal Network of K8s

The more basic Service configuration is of type **ClusterIP**. **NodePort** does exactly the same as **ClusterIP** Service but in addition, it exposes the application on a port on all Nodes in the Cluster. This way, external users can use the service connecting directly with nodes IP's.

11.8.2 How services works, kube-proxy

So our focus must be more on Services than in Pods, how are these Services getting these IP addresses? And how are they made available across all the Nodes in the Cluster? How is the service made available to external users through a port on each Node? The responsible is **kube-proxy** element, which is running into every Node on the Cluster.

Kube-proxy element watches the changes in the cluster through kube-apiserver and every time a new service is to be created, kube-proxy gets into action. Unlike Pods, Services are not created on each Node or assigned to a Node, **Services are Cluster-wide concept**. They exist across all the Nodes in the Cluster. As a matter of fact, they do not exist at all. Besides, there are no processes, bridges, namespaces or interfaces for a Service, it is just a virtual object.

So how do they get an IP address? And how are we able to access the application on a Pod through a Service? **When we create a service object in K8s it is assigned an IP address from a predefined range.** The **kube-proxy** components running on each node gets that IP address and creates forwarding rules on each Node saying: "Any traffic coming to this IP should go to the IP of the Pod". Once that is in place, whenever a pod tries to reach the IP of the Service it is forwarded to the Pod's IP address, which is accessible from any Node in the Cluster (as we have discussed in previous lectures). But remember, it is not just an IP, it is an IP and a Port combination, whenever Services are created or deleted **kube-proxy** component of each Node creates or delete these rules.

11.8.3 How kube-proxy set that rules?

kube-proxy supports different ways to set networking rules:

- **userspace**: kube-proxy listens on a port
- **ipvs rules**
- **iptables (default)**: as we saw in section 10.1

The kube-proxy can be set using the proxy mode option, default is iptables:

```
$ kube-proxy --proxy-mode [userspace | iptables | ipvs] ...
```

11.8.4 Example of iptable rule application

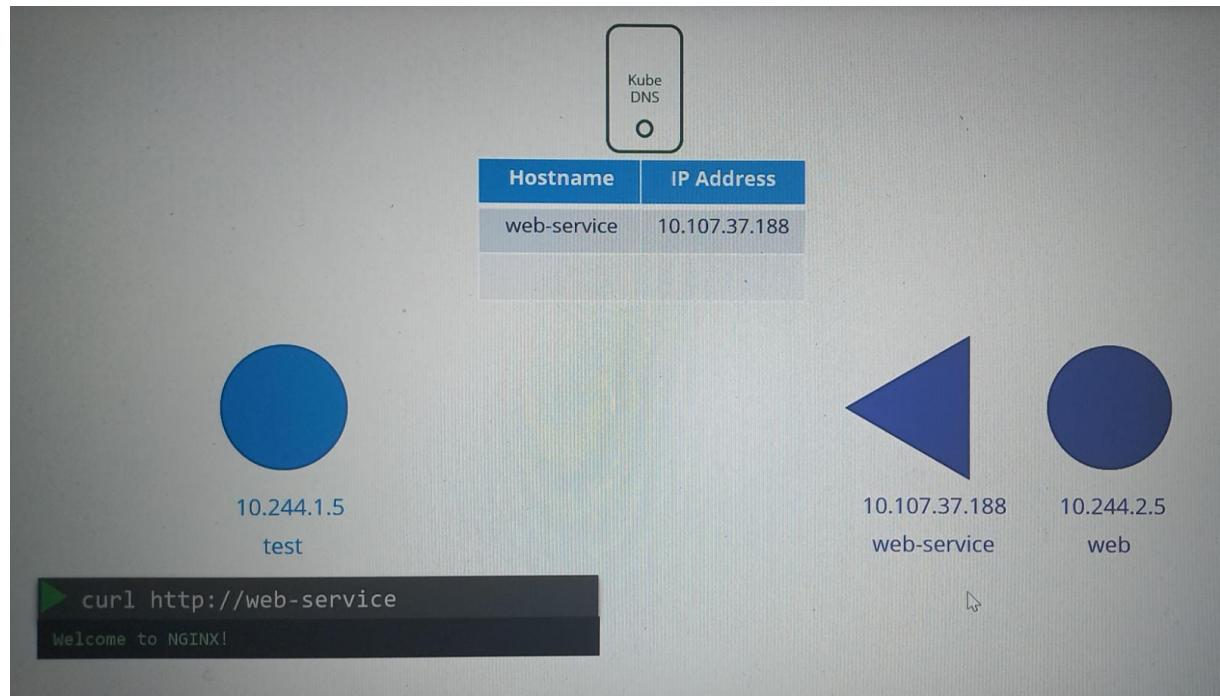
PENDING NTAP redirection

11.9 DNS on K8s

So imagine we have 3 Nodes in our K8s cluster with Pods and services running. Each node has a nodename and IP address assigned to it. The Node names and IP addresses of the Cluster are probably registered in a DNS server in your organization, and how access them are not of concern in this lecture, in this lecture we will discuss the DNS resolution within the cluster, between different components such as Pods and Services.

K8s deploys a **built-in DNS server** by default when you setup a cluster, if you run k8s from scratch, you have to do it by yourself. We focus purely on pods and services within the cluster, we are not interested on Nodes FQDN.

Let's see this example:



Looking at the Pods IP's you should notice that it would be running on different Nodes, but this is not important, because all Pods and Services can reach each other using their IP addresses. So to make the webserver Pod accessible to other Pods we use a Service.

Whenever a Service is created, the **K8s DNS Server** creates a record for the Service, mapping the Service name with its IP address. So within the Cluster, any Pod can reach now the newly created

Service using its service name. Remember that the Service name can change if the Pod who is calling it is in the same Namespace or in another Namespace

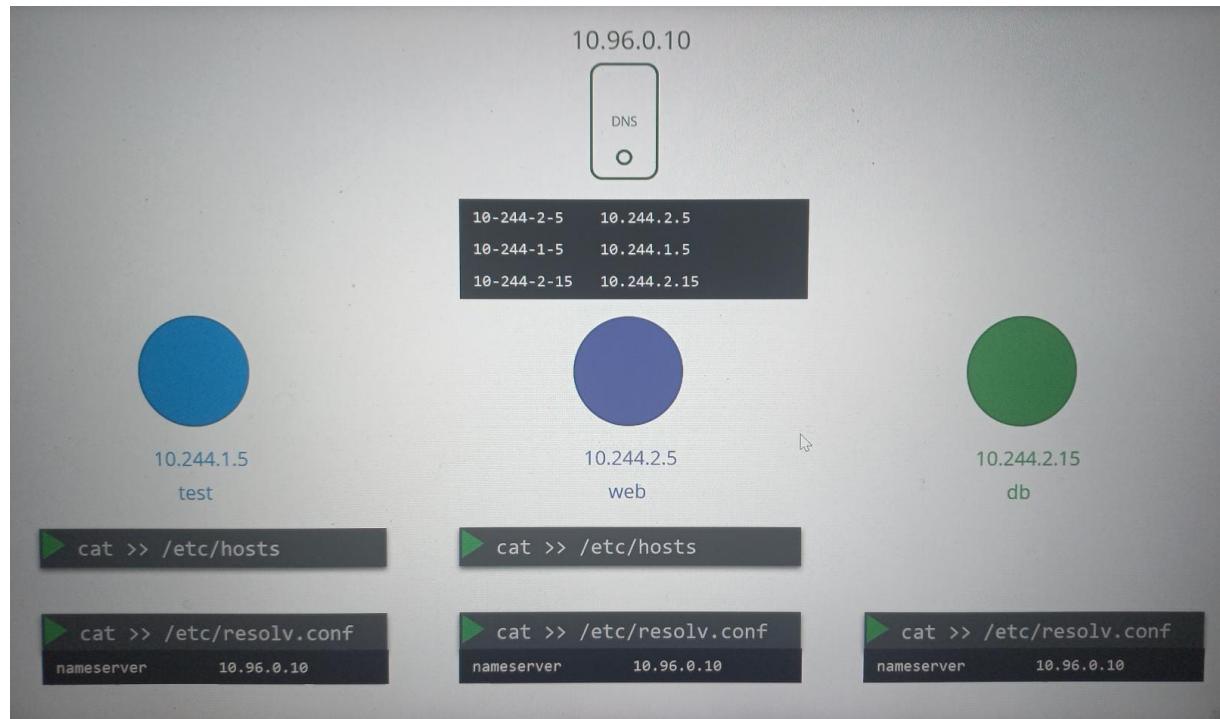
- **Pod in the same Namespace:** service-name
- **Pod outside the Service Namespaces:** service-name.namespace

For each Namespace the **K8s DNS Server** creates a Sub domain, all the Services are group together into another Sub Domain called **svc**, so you can reach your application using **service-name.namespace-name.svc**. Additionally, all resources in a Cluster are group together into a route domain for the cluster **cluster.local**, so you can also reach your application using the URL which is the FQDN **service-name.namespace-name.svc.cluster.local**. So as you see, **Services** are the suitable objects to consume apps within the Cluster.

Records for Pods are not created by default, but we can enable it explicitly. But once a Pod is created, **K8s DNS Server** creates a record for the Pod, exactly equal to the IP of the Pod but with dashes (10.3.4.0 : 10-3-4-0), **Namespace** is the same and type is pod. So to access directly a pod you can use: **pod-ip-dashes.namespace-name.pod.cluster.local**

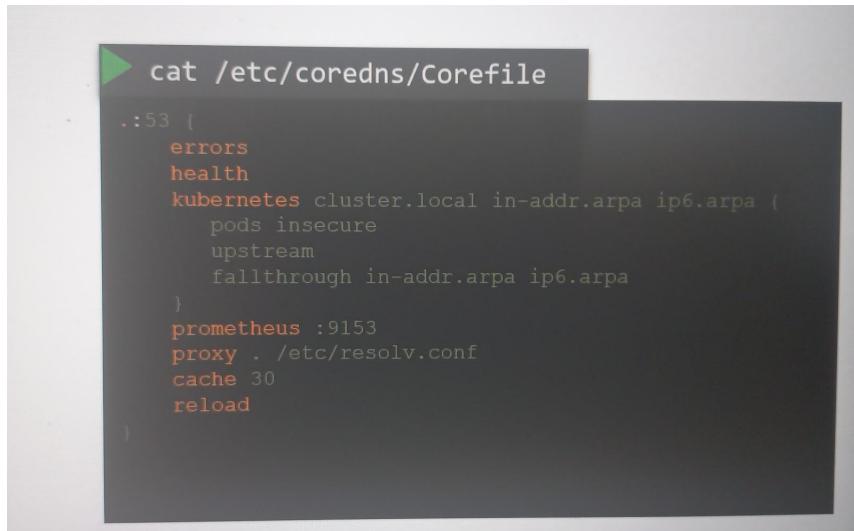
11.9.1 How K8s Implements DNS in the Cluster

In this lecture we will see how K8s makes possible to reach a Service or a Pod from another Pod. So if you have 2 or 3 Pods, you only have to configure the `/etc/hosts` file to add the mapping. But as in a Cluster can be thousands of hundreds of Pods it is not feasible. Instead we move these entries into a **Central K8s DNS Server**. We have to add this **DNS Server** to the Pods configuration file `/etc/resolv.conf` specifying that the nameserver is at the IP address of the **Central K8s DNS Server**. Every time a new Pod or Service is created we add a record in the DNS server for that Pod so the other Pods can access the new Pod or Service with their FQDN, and the new Pod can also resolve other Pods in the Cluster.



K8s implement DNS in the same way, it deploys a **DNS Server** within the Cluster, it must be CoreDNS. But how is CoreDNS setup in the Cluster. It is deployed as a Pod in the **kube-system Namespace**, actually it is created as two Pods for redundancy as part of a ReplicaSet and a Deployment. This Pod

runs **CoreDNs executable** the same executable we should run if we create our cluster from scratch. It requires a configuration file named **Corefile** located on `/etc/coredns/Corefile`



```
▶ cat /etc/coredns/Corefile
.:53 {
    errors
    health
    kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure
        upstream
        fallthrough in-addr.arpa ip6.arpa
    }
    prometheus :9153
    proxy . /etc/resolv.conf
    cache 30
    reload
}
```

This file is charged into a **ConfigMap** in the K8s **kube-admin** Namespace

11.9.2 How Pods point to the CoreDNS Server?

What address do the Pods use to reach the **DNS Server**?

PENDING NTAP redirection

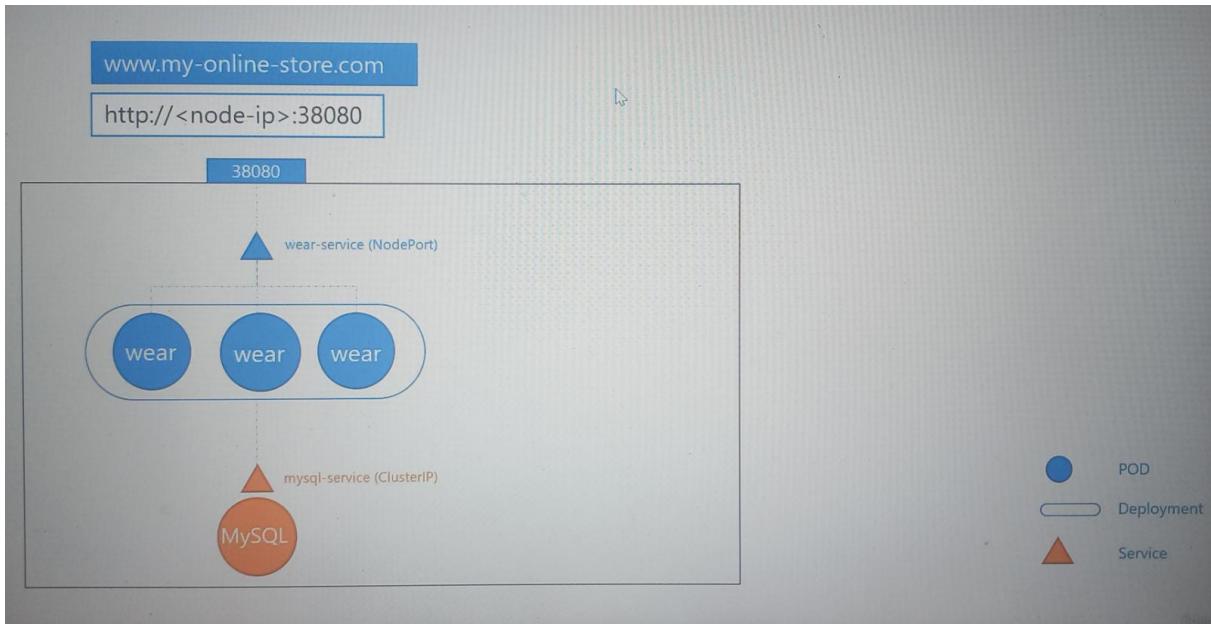
11.10 Ingress

[Official K8s Doc](#)

11.10.1 The need of Ingress

Imagine you are deploying an application on K8s for a company that has an online store selling products, your application would be available at `my-online-store.com`. You build your application into a docker image, and run it in your K8s Cluster in a Pod, into a Deployment. Your application also needs a DB, so you deploy a MYSQL DB as a Pod and create a service of type ClusterIP, called `mysql-service` to make it accessible by our application.

So the application is now working, and it connects with the `mysql-service` properly, but how can we make the application accessible to the outside world? At first you have to create another service, to homogenize the application under a stable IP inside the cluster, with load balancing. It does not matter, but imagine that you create your Service NodePort type with `nodePort=38080`, so you make your application available on an IP within the cluster and on a static Pod in every Node of the Cluster.



So actually, **users can access to the application using the url `http://IP_OF_ONE_RANDOM_NODE:38080`.** This incredibly works, and users are able to acces the application. But as you can imagine it is not very suitable.

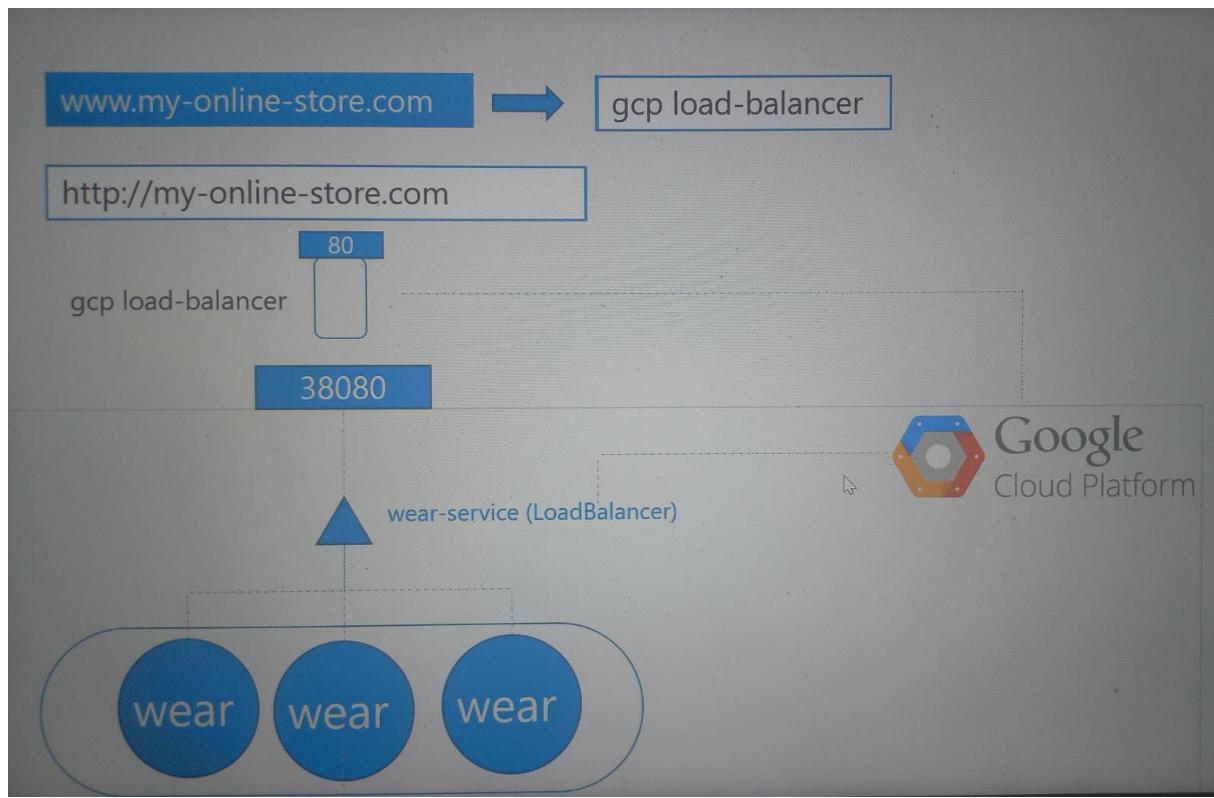
Imagine that the traffic increases, so it increase the number of replicas to handle the additional traffic. That's not a problem because the Service take care of splitting traffic between the Pods. However if you have deployed a production-grade application before, there are many more things involves in addition to simply splitting the traffic between the pods.

For example, we do not want users to have to type in the IP adress every time, so you to solve this, you can configure your DNS Server to point to the IP of the Nodes, so the user can now access the application using the URL `http://my-online-store.com:38080`.

We do not want to that the user have to remember port number either, however service nodePorts can only allocate high numbered ports. So you have to add some piece between the user and the service who maps the port 80 to port 38080 on your Nodes. You then point your DNS to this server and users can now access the application by simply visiting `my-online-store.com`

Imagine that you deploy your application in for example Google Cloud. You could instead of creating a Service of type NodePort you can create it as type LoadBalancer. When you do that, K8s will do the same that i has to do for a NodePort, which is creating a virtual ip for the Service accessible within all the Cluster and provision a high port for the service in all Cluster Nodes. But in addition to that, K8s also sends a request to Google Cloud Platform to provision a network load balancer for the Service. On receiving th request, Google Cloud Platform will then automatically deploy a load balancer configured to route traffic to the Service ports on all nodes.

The load balancer has an external IP that can be provided to users to access the application. In this case we can set the **DNS Server** to point to this IP and users access the application using the URL `my-online-store.com`

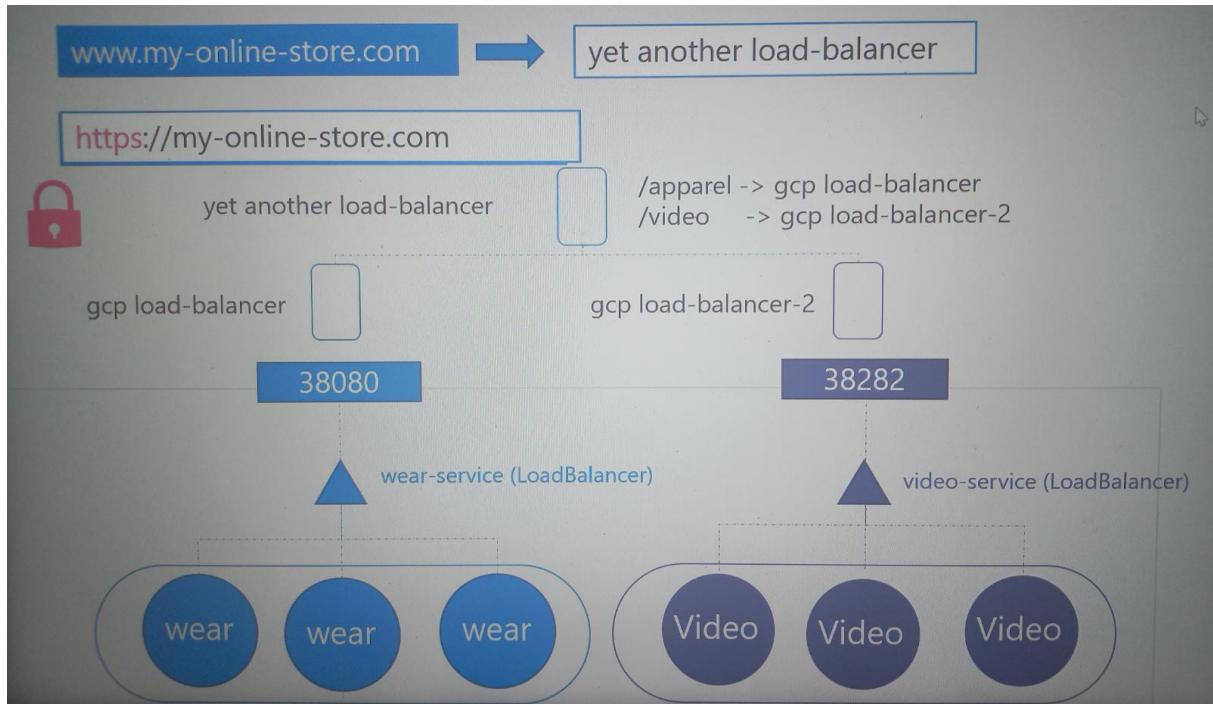


Your company grows and you now have new services for your customers, for example, a video streaming service. Now you want your users to be able to access your new video streaming service by going to `my-online-store.com/watch` and you want your old application be accessible now at `online-store.com/wear`. But they are actually two different microservices or directly applications. So you deploy your new application as a separate Pod, Deployment and Service (called video-service of type LoadBalancer) and K8s provision support 38282 for this Service and also provision a Network LoadBalancer on the Cloud. The new LoadBalancer has a new IP and remember that you must pay for each of these load balancers and having many such loadBalancers can inversely affect your Cloud Bill.

So how do you direct traffic of these LoadBalancers based on the URL that the users type in. You need yet another proxy or LoadBalancer that can redirect traffic based on URL to the different Services. And every time you introduce a new Service you have to reconfigure the LoadBalancer. In addition, you should enable SSL for your applications so your users can access your application using https. Where do you configure that? It can be done at different levels:

- Application Level
- Load Balancer Level
- Proxy Server Level

But you want to configure that in one place with minimal maintenance.



So when the Cluster scales it becomes unmanageable, it would be nice if K8s manage all of that within the K8s Cluster and have all that configuration in just another configuration file that lives along with the rest of your application Deployment and Services files. And this exists and they are Ingress.

11.10.2 Introduction to Ingress

Ingress helps users to access your application using a single externally accessible URL that you can configure to route traffic to different services within your Cluster based on the URL path. At the same time, implement SSL security as well. Think of **Ingress** as a layer 7 LoadBalancer built in the K8s Cluster that can be configured using native K8s primitives just like any other Object.

Even with **Ingress** you still need to expose it to make it accessible outside the Cluster, so you still have to either publish it as a NodePort or with a Cloud-native LoadBalancer. But that is just a one-time configuration. Going forward, you are going to perform all your load balancing of SSL and URL-based routing configuration on the **Ingress-Controller**.

11.10.3 Ingress Controller & Ingress Resources

But what is an Ingress? Where it is? How can you see it, and how can you configure it? Without Ingress all its works would be done with a proxy-reverse or a LoadBalancer like NGINX, HAProxy, etc. It will be deployed in the K8s Cluster and configure them to route traffic to another services, with user routes, SSL certificates, endpoint services, etc.

Ingress are implemented by K8s in a very similar way, the **product deployed** by K8s is called **Ingress Controller** (is a product like NGINX), and the **set of rules** you configure are called **Ingress Resources**. **Ingress Resources** are created using definition YAML files.

11.10.4 Ingress Controller

K8s Clusters does not come with an Ingress Controller by default, so you only create Ingress Resources and you expect it works, it won't. So you must deploy one.

There are a number of solutions available for **Ingress Controller**:

- **GCE:** Google Controller layer 7 HTTP Load Balancer
- **NGINX**
- **Istio**
- **Haproxy**
- **Traefic**

Currently, **GCE** and **NGINX** are currently being supported and maintained by the K8s project, and in this lecture we will use NGINX as an example. This Ingress Controller are not just another Load Balance or NGINX server, the Load Balancer components are just part of it. Ingress Controllers have additional intelligence to monitor the K8s Cluster for new definitions of **Ingress Resources** and configure the **NGINX Server accordingly**. A **NGINX Controller** is deployed just as another deployment in K8s.

To run **properly** an **Ingress Controller** in your K8s Clsuter you should **create** the following resources:

- **Deployment** with an Ingress Controller Image to run into containers
- **ConfigMap** to configure the NGINX
- **Service** type NodePort to expose it
- **ServiceAccount** with the right permission to access all of these objects

11.10.4.1 Deployment

```
nginx-ingress-controller.yaml

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx-ingress-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      name: nginx-ingress
  template:
    metadata:
      labels:
        name: nginx-ingress
    spec:
      containers:
        - image: quay.io/kubernetes-ingress-controller/nginx-ingress-controller:0.21.0
          name: nginx-ingress-controller
        args:
          - /nginx-ingress-controller
          - --configmaps=$(POD_NAMESPACE)/nginx-configuration
        env:
          - name: POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
      ports:
        - containerPort: 80
          name: http
        - containerPort: 443
          name: https
```

Within the image, the NGINX program is stored at location `/nginx-ingress-controller` so you must pass that as the command to start the NGINX controller service. Also, NGINX has more configurations like:

- Path to store the logs
- Keep alive threshold
- SSL settings
- Session timeout

Also NGINX server need the env configuration to run properly and to specify the ports used by the Ingress Controller, which happens to be 80 for http connections and 443 for secure (https) connections.

11.10.4.2 ConfigMap

In order to decouple this configuration data from the NGINX Controller Image we must create a ConfigMap Object:

```
nginx-configuration.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-configuration
```

You can let it blank if you want, it would apply the configuration by default, but it would make easy for you to modify a configuration setting in the future.

11.10.4.3 Service

We then need a Service to expose the Ingress Controller to the external world, so we create a Service of type NodePort with the nginx-ingress label selector to link the service to the Deployment:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-ingress
spec:
  type: NodePort
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
      name: http
    - port: 443
      targetPort: 443
      protocol: TCP
      name: https
  selector:
    name: nginx-ingress
```

11.10.4.4 ServiceAccount

As we said before, the Ingress Controllers have additional intelligence built into them to monitor the K8s Cluster for Ingress Resources and configure the underlying Ingress Controller when something is changed. But for the Ingress Controller to do this, it requires a Service Account with the right set of permissions: Roles, ClusterRoles and RoleBindings

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: nginx-ingress-serviceaccount
```

11.10.5 Ingress Resources

[Official K8s Doc](#)

An Ingress Resource is a set of rules and configurations applied on the **Ingress Controller**, you can configure rules to say simply forward all incoming traffic to a single application or route traffic to different applications based on URL, so if the user goes to my-online-store.com/wear then route to one app, or if the user visits the my-online-store.com/watch then route to another app. Or you can route user based on the domain name itself, for example wear.my-online-store.com and watch.my-online-store.com

The Ingress Resource Objects are created with a YAML definition file, this one for example routes all the incoming traffic in the Cluster to the **wear-service Service on port 80**:

```
ingress-wear.yaml

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  backend:
    serviceName: wear-service
    servicePort: 80
```

Then create it:

```
$ kubectl create -f ingress-wear.yaml
```

```
$ kubectl get ingress
```

11.10.5.1 Rules

You can use rules when you want to route traffic based on different conditions. For example, you can create one route for traffic originating from each domain or hostname.

That means when users reach your cluster using:

- domain name my-online-store.com, you can handle that traffic using rule 1.
- domain name wear.my-online-store.com you can handle that traffic using rule 2.
- domain name watch.my-online-store.com you can handle that traffic using rule 3.
- use rule 4 to handle everything else.

And as we discussed before, you could **get different domain names** to reach the Cluster by **adding multiple DNS entries all pointing to the same Ingress Controller Service** on the K8s Cluster.

So within each rule, you can define different paths and backend Services which serves in this hostnames.

So let's define rules:

```

ingress-wear.yaml

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  rules:
    - http:
        paths:
          - path: /wear
            backend:
              serviceName: wear-service
              servicePort: 80
          - path: /watch
            backend:
              serviceName: watch-service
              servicePort: 80
      backend:
        serviceName: wear-service
        servicePort: 80

```

NOTE

If you use the command `describe` in one Ingress object you will see that there is a field called **Default backend**, what's that? If a user tries to access a URL that does not match any of these rules, then the user is redirected to the service specified as the **default backend**

But this configures all the URL's that the user uses, what if you want to use different rules for each URL?

```

ingress.yaml

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  rules:
    - host: wear.my-online-store.com
      http:
        paths:
          - backend:
              serviceName: wear-service
              servicePort: 80
    - host: watch.my-online-store.com
      http:
        paths:
          - backend:
              serviceName: watch-service
              servicePort: 80
  backend:
    serviceName: wear-service
    servicePort: 80

```

So in this case the path is indiferent, what really matters is the URL that the users uses to access to the application. It depends on how application wants to manage their services. However, you can specify in

addition different paths to this configuration to add more complexity to the configuration.

11.10.5.2 Ingress Resources updates in last K8s Versions

1. **apiVersion:** as changed to networking.k8s.io/v1
2. **pathType:** there are 3 path types
 - **Exact:** Matches the URL path exactly and with case sensitivity
 - **Prefix:** Matches based on a URL path prefix split by /. Matching is case sensitive and done on a path element by element basis. A path element refers to the list of labels in the path split by the / separator.
 - **ImplementationSpecific:** With this path type, matching is up to the IngressClass. Implementations can treat this as a separate pathType or treat it identically to Prefix or Exact path types.
3. **serviceName & servicePort**

How does pathType Works?

Kind	Path(s)	Request Path	Matches?
Prefix	/	(all paths)	Yes
Exact	/	(all paths)	No, only matches "/"
Prefix	/bar	/bar, /bar/temp	Yes
Exact	/bar	/bar, /bar/temp	No, only matches "/bar"
Exact	/bar/	/bar, /bar	No, only matches "/bar/"
Prefix	/bar/	/bar	Yes
Prefix	/wear	/watch	No, uses default backend
Prefix	/, /wear	/watch	Yes, it match "/"
Exact	/, /wear	/watch	No
Exact	/, /wear	/	Yes

```

ingress.yaml

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
spec:
  rules:
    - host: wear.my-online-store.com
      http:
        paths:
          - pathType: Prefix
            path: /
            backend:
              service:
                name: wear-service
                port:
                  number: 80
    - host: my-online-store.com
      http:
        paths:
          - pathType: Prefix
            path: /watch
            backend:
              service:
                name: watch-service
                port:
                  number: 80
        - pathType: Prefix
          path: /
          backend:
            service:
              name: watch-service
              port:
                number: 80

```

11.10.5.3 kubectl create ingress in the last versions

```
$ kubectl create ingress --help
```

```
$ kubectl create ingress <ingress-name> \
--rule="wear.my-online.store.com*=wear-service:80*
```

11.10.5.4 Ingress annotations and rewrite target

Imagine an streaming video service using K8s, it has the main page on the URL `http://watch-myapp.com` and the different videos on `http://video-myapp.com/videoN`. So we must configure Ingress to achieve it, because when users visit the main page and click on a video, his request should be forwarded internally to the video URL. So we want to "Rewrite" the URL when the request is passed on to the watch or wear applications. We do not want to pass in the same path that was typed in. So when we specify the `rewrite-target` option, it rewrites the URL by replacing whatever is under `rules->http->paths->path` with the `rewrite-target`.

EXAMPLE

```
ingress.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: ingress-wear
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - http:
    paths:
    - pathType: Prefix
      path: /wear
      backend:
        service:
          name: wear-service
          port:
            number: 80
```

11.10.6 Lab X - Playing with Ingress 1

11.10.6.1 Question 1

To know the **Ingress Controller** deployed in the Cluster you should find for **Pods** or **Deployments**, because they are deployed as Pods running a container with the nginx controller solution.

```
$ kubectl get po -A | grep -i controller
```

```
$ kubectl get deploy --all-namespaces | grep -i controller
```

11.10.6.2 Question 6

To see where the Ingress Object is deployed, now yes, you should find for Ingress:

```
$ kubectl get ing -A
```

11.10.6.3 Question 8

To see What is the Host configured on the Ingress Resource, we can do it by two ways:

- **Describe**

```
$ kubectl get ing <ing-name> -n <namespace-name> | gre
```

- **Get with jsonpath**

```
$ kubectl get ing <ing-name> -n <namespace-name> -o jsonpath='{...host}'
```

WARNING

As it happens, if there are not hosts defined it means **All hosts(*)**, and it only can be seen it in the **describe**, because in the YAML or JSON it would be empty.

11.10.6.4 Question 9

To know the backend configured in the /wear path on the Ingress:

```
$ kubectl describe ing <ing-name> -n <namespace-name> | grep -i "/wear" -A 3
```

You can also do it by **jsonpath**:

```
$ kc get ing ingress-wear-watch -o jsonpath='{...paths}' -n app-space
```

```
$ kc get ing ingress-wear-watch -o jsonpath='{...path[0].backend.service.name}' -n app-space
```

11.10.6.5 Question 10

To get what path is hosting the backend service video-service:

```
$ kc get ing ingress-wear-watch -o \
  jsonpath='{range ...paths[*]}.{.path}{"\t"}{.backend.service.name}{"\n"}' -n app-space
```

```
$ kc describe ing ingress-wear-watch 1 grep -i "video" -A 5
```

11.10.6.6 Question 11

WARNING

If the requirement does not match any of the configured paths what service are the requests and **the default backend is not configured** the requirement won't go any service.

11.10.6.7 Question 13

To check the function of the ingress you can use the IP configured / assigned to it with curl:

```
$ kc describe ing <ingres-name> | grep -i IP -A 3
```

```
$ curl http://Ingress_IP/path
```

You can try the different pathes.

Also, you can look for services IP's and Ports. If they are of type ClusterIp you only can access them by their IP and Port, but if they are NodePort, you can access them also using any Node IP with the NodePort assigned. In that case they are three ClusterIP, but you can access the content using:

```
$ curl http://SERVICE_IP:port
```

11.10.6.8 Question 14

The best way to edit when running a K8s Object is using `kubectl edit`, but you can also try with `kubectl apply -f`, `kubectl delete` & `kubectl create...`

NOTE

Do not be anxious, the assignation of an IP to a Host Ingress Configuration can take time, so wait a little and you will see the IP once creating magically:

```
$ kubectl describe ing <ing-name> | grep -i IP -A 2
```

To check use a curl to the new IP.

11.10.6.9 Question 18

You are requested to add a new path to your ingress to make the food delivery application available to your customers, so to do it, you only have to add more paths / backends to the configuration.

WARNING

In this case, the service you have to configure as backend is in the same Namespace, so you can configure it using their Object Name. But if the service is not in the same Namespace, **you should create a new Ingress in the Namespace of the Service**, because Ingress do not accept as service.name the FQDN.

11.10.6.10 Question 21

You are requested to add a new path to your ingress to make the food delivery application available to your customers, so to do it, you only have to add more paths / backends to the configuration.

NOTE

Copy directly from kubectl and the edit

```
$ kubectl get ing <ingres-name> -o yaml > ingress-custom.yaml
```

Custom ingress:

```
ingress-custom.yaml

apiVersion: v1
items:
- apiVersion: networking.k8s.io/v1
  kind: Ingress
  metadata:
    annotations:
      nginx.ingress.kubernetes.io/rewrite-target: /
      nginx.ingress.kubernetes.io/ssl-redirect: "false"
    creationTimestamp: "2023-06-06T07:40:13Z"
    generation: 1
    name: ingress-wear-watch
    namespace: app-space
    resourceVersion: "739"
    uid: 96c7912a-4329-48cb-930f-24bf2fc1b3a7
  spec:
    rules:
    - http:
        paths:
        - backend:
            service:
              name: wear-service
              port:
                number: 8080
            path: /wear
            pathType: Prefix
        - backend:
            service:
              name: video-service
              port:
                number: 8080
            path: /watch
            pathType: Prefix
        - backend:
            service:
              name: food-service
              port:
                number: 8080
            path: /eat
            pathType: Prefix
    status:
      loadBalancer:
        ingress:
        - ip: 10.104.79.231
  kind: List
  metadata:
    resourceVersion: ""
```

Test the configuration using:

```
$ curl http://<INGRESS_IP>/eat
```

11.10.6.11 Question 22

So as we said, Ingress and Service must be in the same Namespace, so we have to create a new Ingress in the critical-space Namespace.

To change the current Namespace:

```
$ kubectl config set-context --current --namespace critical-space
```

```
$ kubectl get svc  
==> pay-service / ports:
```

So now we have to create the new Ingress in critical-space Namespace:

```
$ kubectl create ing --help
```

Since we are not taking into account hosts, so the rules we define in our ingress will be applied to **All Hosts (*)** we will use the simplest configuration helper.

```
$ kubectl create ingress pay-ingress --rule="/pay=pay-service:8282"
```

When you create the ingress, an IP is assigned to it, and when you try to access it from its IP to the path /pay:

```
$ curl http://ING_IP/pay
```

You notice that it is not working as expected. There still is the page 404 error. So let's take a look at the logs in order to find out why.

```
$ kc get po  
==> webapp-pay-3463
```

```
$ kc logs webapp-pay-3463  
==> * Running on http://10.244.0.11:8080/ (Press CTRL+C to quit)  
10.244.0.9 - - [06/Jun/2023 08:57:07] "GET /pay HTTP/1.1" 404 -  
10.244.0.9 - - [06/Jun/2023 08:57:30] "GET /pay/ HTTP/1.1" 404 -  
10.244.0.9 - - [06/Jun/2023 08:57:33] "GET /pay HTTP/1.1" 404 -  
10.244.0.9 - - [06/Jun/2023 08:57:38] "GET //pay HTTP/1.1" 404 -
```

11.10.6.12 Question 23

So the requests are arriving!!! It is working!! But the app seems to not resolve properly the requests. So you talk with the fabricant and they tell you that the app responds in the endpoint "/", not in the endpoint "/pay". So we have to use **url rewriting annotation**.

So:

```
$ kubectl edit ingress pay-ingress
```

```
pay-ingress.yaml
apiVersion: ...
...
metadata:
...
annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
```

So with this annotation, the paths will be rewritten in the backend to look for "/" in the application. And if now we try, we will get the correct page!

```
$ curl http://INGRESS_IP/pay
```

```
$ kc logs webapp-pay-3463
==> 10.244.0.9 - - [06/Jun/2023 09:09:48] "GET / HTTP/1.1" 200 -
```

11.10.7 Lab X - Playing with Ingress 2

11.10.7.1 Question 1

We have deployed two applications. Explore the setup.

```
$ alias kc=kubectl
```

```
$ kc get all -A
```

11.10.7.2 Question 3

Create a CM for the Ingress Controller. Make it easy:

```
$ kubectl create cm --help
```

```
$ kubectl create cm <cm-name>
```

You also can do it by defining an empty cm YAML file, then creating it from the file... Make it easy!

11.10.7.3 Question 4

We create the ServiceAccounts objects creating at first two files and copying the basic configuration from K8s Doc or from this manual. Then we save and we create them using:

```
$ kubectl create -f <file-sa.yaml>
```

```
$ kubectl get sa
```

11.10.7.4 Question 5

Read good the logs, if you pay attention on them, they have always a lot of information:

```
Error from server (BadRequest): error when creating "/root/ingress-controller.yaml":  
Service in version "v1" cannot be handled as a Service:  
strict decoding error: unknown field "spec.ports[0].nodeport"
```

nodeport does not exists!!!! It exists nodePort, 15 minutes thinking about would it could be, here is the answer. So you should read more carefully the logs. Also you are so tired, maybe you should rest for a time.

NOTE

Remember that if the Service it's not provided, you can easily configure one using:

```
$ kubectl expose --help
```

11.10.7.5 Question 7

```
$ kc create ing app-ingress \  
--rule=/wear=wear-service:8080 \  
--rule=/watch=video-service:8080
```

```
$ kc get ing  
==> ADDRESS: xxx
```

```
$ curl http://XXX/wear
```

12 CheatSheet

12.1 K8s Best Practices

Here are some links to check K8s best practices:

- [Oficial kubectl Usage Conventions](#)
- [10 K8s Best Practices](#)
- [Oficial K8s Best Practices](#)

12.2 Get

Get extra info:

```
$ kubectl get resource resource_name -o wide
```

Get/describe based on labels:

```
$ kubectl get/describe resource -l tier=busy
```

Get using jsonpath:

- Basic

```
$ kubectl get resource resource_name -o jsonpath='{..labels}',
```

- Range

```
$ kubectl get resource resource_name -o jsonpath='{range ..containers[*]}{.image}{"\t"},'
```

Retrieve events:

```
$ kubectl describe resource resource_name | grep -i events -a10
```

12.3 Create Objects

As you might have seen already, it is a bit difficult to create and edit YAML files from scratch. We can use **imperative** commands to help us in generating a YAML template. Or **declarative** commands to help us to just create the objects. We can use imperative as well, but it's recommended to use declarative for creation.

12.3.1 Imperative

12.3.1.1 General Resources

Use the helper from `create` command ([Oficial Doc](#)):

```
$ kubectl create <resource_type> --help
```

NOTE

You can follow this procedure in order to create custom Object:

```
$ kubectl create <resource_type> <resource_name> ... -o yaml --dry-run=client \
> template.yaml

$ vim template.yaml

$ kubectl create -f template.yaml
```

From template:

```
$ kubectl create -f resource_file.yaml [-f resource_file_2.yaml]
```

Replace an object with new config:

```
$ kubectl replace -f resource_file.yaml [-f resource_file_2.yaml]
```

```
$ kubectl replace --force -f resource_file.yaml [-f resource_file_2.yaml]
```

NOTE

When changes can not be overridden, you can use the following technique:

- Run the `kubectl edit <resource_type> <resource_name>`
- Save changes, until they cannot be applied.
- Changes will be saved in a file like: `/tmp/kubectl-edit-12345.yaml`, then execute: Replace an object with new config:

```
$ kubectl replace --force -f /tmp/kubectl-edit-12345.yaml
```

12.3.1.2 Pods

Most used - `run` ([Official Doc](#)):

```
$ kubectl run --help
```

```
$ kubectl run <pod_name> --image=nginx \
[--port=xxx] [--env="VAR=value"] [--env="VAR2=value2"] [--dry-run] \
[--command -- <command>]
```

NOTE

You can follow this procedure in order to create custom Pod:

```
$ kubectl run <pod-name> --image=<image>:<tag> -o yaml --dry-run=client \
> pod.yaml

$ vim pod.yaml

$ kubectl create -f pod.yaml
```

12.3.1.3 Services

Take a replication controller, service or pod and expose it as a new Kubernetes Service ([Official Doc](#)).

```
$ kubectl expose --help

$ kubectl expose <obj-type> <obj-name> [--port=port] [--protocol=TCP|UDP] \
[--target-port=number-or-name] [--name=name] \
[--external-ip=external-ip-of-service] [--type=type]
```

NOTE

By default when `kubectl expose` is used, a ClusterIP service is created, but you can change it using the `--type` label.

12.3.2 Declarative

Apply:

```
$ kubectl apply -f resource_file.yaml [-f resource_file_2.yaml]
```

12.4 Delete

```
$ kubectl delete <resource_type> <resource_name_1> ... <resource_name_n>
```

12.5 Policies

[Official K8s Doc](#)

Kubernetes policies are configurations that manage other configurations or runtime behaviors. Kubernetes offers various forms of policies, for example:

- **NetworkPolicies:** can be used to restrict ingress and egress traffic for a workload.

- **LimitRanges:** manage resource allocation constraints across different object kinds.
- **ResourceQuotas:** limit resource consumption for a namespace.