CONTENIDOS

1 DESPLIEGUE DE APLICACIONES EN CONTENEDORES (2 HORAS)

- Introducción a los contenedores
- Arquitectura de microservicios
- Tecnologías subyacentes y diferencias entre ellas: docker, cri-o, LXC, ...
- Ciclo de vida en el despliegue de aplicaciones con docker

2 INTRO A KUBERNETES (2 HORAS)

- Características, historia, estado actual del proyecto kubernetes (k8s)
- Arquitectura básica de k8s
- Alternativas para instalación simple de k8s: minikube, kubeadm, k3s
- Instalación con minikube
- Instalación y uso de kubectl
- Despliegue de aplicaciones con k8s

3 DESPLIEGUE DE APLICACIONES CON K8S (1:30 HORAS)

- Pods
- ReplicaSet: Tolerancia y escalabilidad
- Deployment: Actualizaciones y despliegues automáticos

4 COMUNICACIÓN ENTRE SERVICIOS Y ACCESO DESDE EL EXTERIOR (1:30 HORAS)

- Services
- DNS
- Ingress
- Ejemplos de uso y despliegues

5 CONFIGURACIÓN DE APLICACIONES (1 HORA)

- Variables de entorno
- ConfigMaps
- Secrets
- Ejemplo de despliegue parametrizado

6 ALMACENAMIENTO (1:30 HORAS)

- Consideraciones sobre el almacenamiento
- PersistentVolume
- PersistentVolumeClaim
- Ejemplo de despliegue con volúmenes

CONTENIDOS

7 OTROS TIPOS DE DESPLIEGUES (1:30 HORAS)

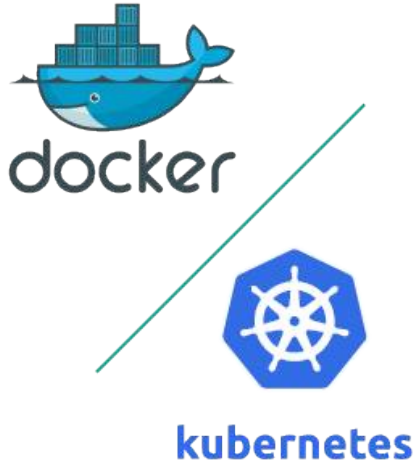
- StatefulSet
- DaemonSet
- AutoScale
- Helm

8 ADMINISTRACIÓN BÁSICA (1 HORA)

- Namespaces
- Usuarios
- RBAC
- Cuotas y límites

9 INSTALACIÓN PASO A PASO (4 HORAS)

- Consideraciones previas: Requisitos hardware, arquitectura física y lógica, entornos y herramientas para el despliegue
- Instalación completa componente a componente en múltiples nodos



MÓDULO 8. ADMINISTRACIÓN BÁSICA

Namespaces

Los Namespaces nos permiten aislar recursos para el uso por los distintos usuarios del cluster, para trabajar en distintos proyectos. A cada namespace se le puede asignar una cuota y definir reglas y políticas de acceso.

```
kubectl get namespaces
NAME          STATUS    AGE
default       Active    1d
kube-public   Active    1d
kube-system   Active    1d
```

Para crear un nuevo namespace:

```
kubectl create ns proyecto1
kubectl describe ns proyecto1
```

O usando una definición yaml:

```yaml
apiVersion: v1
kind: Namespace
metadata:
 name: proyecto1
```

Para crear un recurso en un namespace:

```
kubectl create deployment nginx --image=nginx -n proyecto1
kubectl expose deployment nginx --port=80 --type=NodePort -n proyecto1
kubectl get deploy -n proyecto1
kubectl get deploy --all-namespaces
```

O usando una definición yaml:

```yaml
apiVersion: apps/v1beta1
kind: Deployment
metadata:
 name: nginx
 namespace: proyecto1
 ...
```

Al borrar un namespaces se borran los recursos creados (los volúmenes no se asocian a namespaces):

```
kubectl delete ns proyecto1
```

Usuarios en Kubernetes

Hasta ahora hemos estado trabajando en nuestro cluster con el usuario **admin: `~/.kube/config`**

Pero en un cluster real tenemos distintos usuarios y grupos:

Kebernetes no nos ofrece un una gestión de usuarios similar a otros objetos de la API: ~~**`kubectl create user`**~~

La gestión de usuarios la tiene que realizar el administrador usando distintos mecanismos:

- **Usando certificados**
- Usando tokens
- Autentificación básica
- oAuth2

Autentificación de usuarios basada en certificados

- Kubernetes es configurado con una autoridad certificadora (CA). La podemos encontrar:
  - Normalmente en el nodo master: `/etc/kubernetes/pki/ca.crt` y `/etc/kubernetes/pki/ca.key`
  - En minikube, la encontramos en nuestro equipo: `~/.minikube/ca.crt` y `~/.minikube/ca.key`
- Cualquier certificado firmado por la CA es admitido por nuestro cluster
- Tenemos varias opciones, pero nosotros vamos a usar OpenSSL para realizar la firma.
- Tenemos que tener en cuenta dos campos importantes en el certificado SSL:
  - **Common Name (CN)**: Kubernetes lo interpreta como el **usuario**
  - **Organization (O)**: Kubernetes lo interpreta como el **grupo**
- Vamos a crear el siguiente usuario:

user:usuario1
group: desarrollo

- En el namespace **proyecto1**: `kubectl create namespace proyecto1`

Autentificación de usuarios basada en certificados

- Creamos la clave privada (en `~/.certs`):

```
openssl genrsa -out usuario1.key 2048
```

- Creamos la petición de certificado (CSR):

```
openssl req -new -key usuario1.key -out usuario1.csr -subj "/CN=usuario1/O=desarrollo"
```

- Mandamos el CSR al administrador

- Crea el certificado a partir de CSR firmándolo con el CA:

```
openssl x509 -req -in usuario1.csr -CA ~/.minikube/ca.crt -CAkey
~/.minikube/ca.key -CAcreateserial -out usuario1.crt -days 500
```

Autentificación de usuarios basada en certificados

1. **Suponemos que el usuario se está conectando a un cluster que ya tiene configurado en `~/.kube/config`**

- Añadimos las credenciales del nuevo usuario:

```
kubectl config set-credentials usuario1
--client-certificate=/home/debian/.certs/usuario1.crt
--client-key=/home/debian/.certs/usuario1.key
```

- Añadimos el nuevo contexto:

```
kubectl config set-context usuario1-context --cluster=minikube --
namespace=proyecto1 --user=usuario1
```

Autentificación de usuarios basada en certificados

2. Suponemos que el usuario quiere crear un nuevo fichero de configuración de acceso a un nuevo cluster

```
export KUBECONFIG=~/.kube/micluster.yaml
```

- Añadimos el nuevo cluster

```
kubectl config set-cluster mi_cluster
--certificate-authority=/home/debian/.minikube/ca.crt --embed-certs=true --
server=https://192.168.99.100:6443
```

- Añadimos las credenciales del nuevo usuario:

```
kubectl config set-credentials usuario1
--client-certificate=/home/debian/.certs/usuario1.crt
--client-key=/home/debian/.certs/usuario1.key
```

- Añadimos el nuevo contexto:

```
kubectl config set-context usuario1-context --cluster=mi_cluster --
namespace=proyecto1 --user=usuario1
```

Autentificación de usuarios basada en certificados

Seguimos usando nuestro fichero de configuración `~/.kube/config`

```
export KUBECONFIG=~/.kube/config
```

- Mostramos los contextos definidos en el fichero de configuración:

```
kubectl config get-contexts
CURRENT   NAME                CLUSTER   AUTHINFO   NAMESPACE
          usuario1-context    minikube   usuario1   proyecto1
*         minikube            minikube   minikube
```

- Elegimos el nuevo contexto:

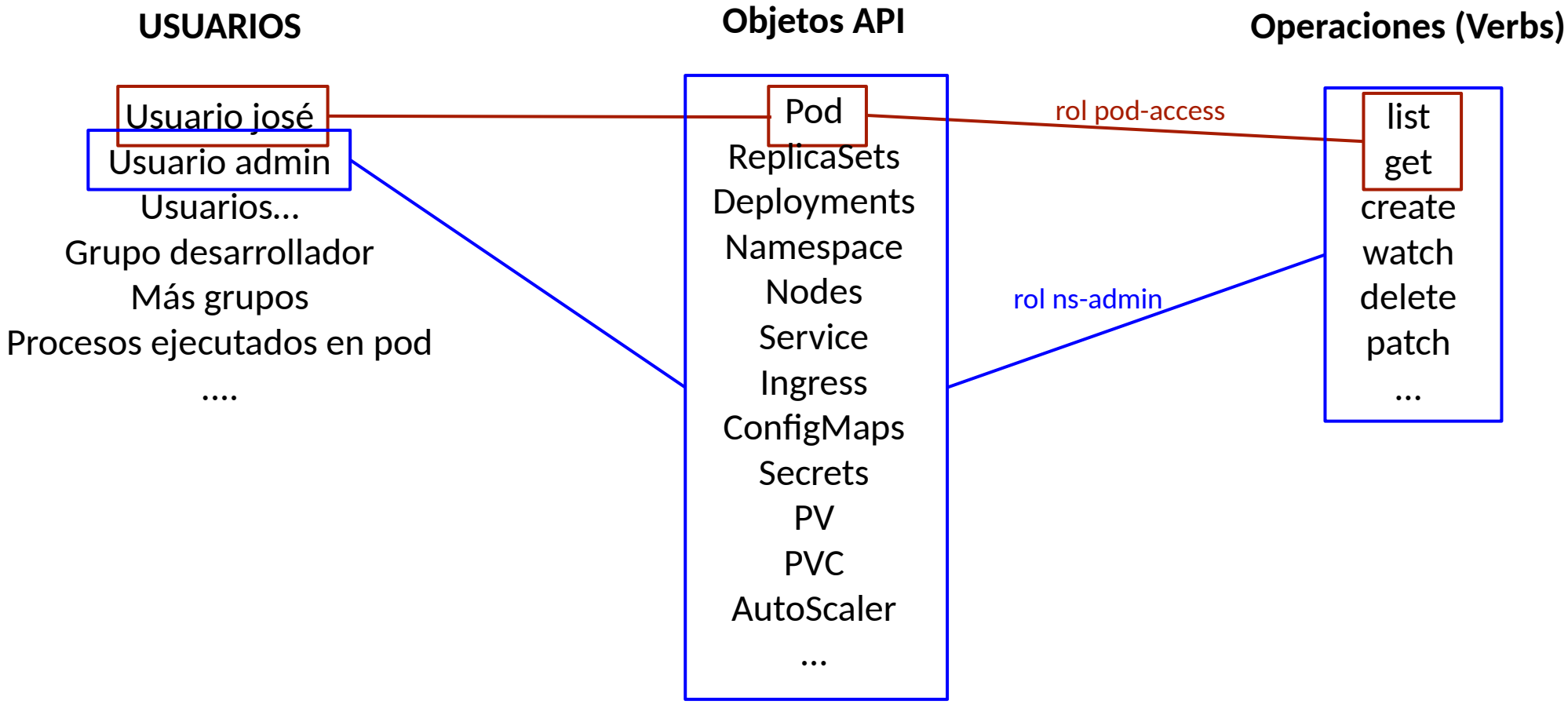
```
kubectl config use-context usuario1-context
```

No tenemos permiso para visualizar la lista de Pods!!!!!
Necesitamos gestionar autorizaciones RBAC

```
kubectl get pods
Error from server (Forbidden): pods is forbidden: User "usuario1" cannot list
resource "pods" in API group "" in the namespace "proyecto1"
```

RBAC

En nuestro cluster kubernetes tenemos 3 grupos importantes:



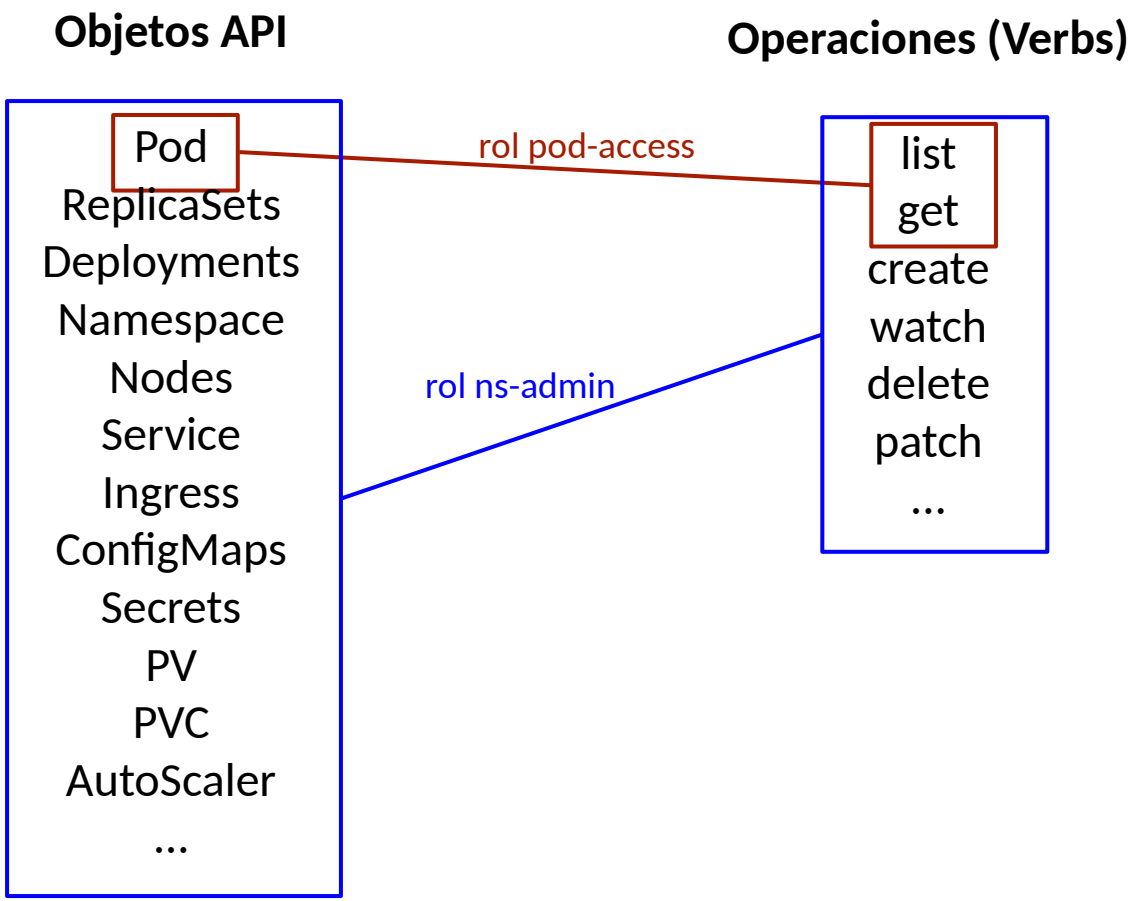
RBAC (Role Based Access Control): Es un sistema de autorización basado en conecta los tres grupos anteriores.

RBAC. Ejemplo de Operaciones

`kubectl run --image=bitnami/mongodb my-mongodb`	deployments: create
`kubectl get deployments -w`	deployments: get, list, watch
`kubectl delete deployment my-mongodb`	deployments: get, delete
`kubectl edit deployment my-mongodb mypod`	deployments: get, patch
`kubectl expose deployment my-mongodb --port=27017 --type=NodePort`	deployments: get services: create
`kubectl exec -ti mypod bash`	pods: get pods/exec: create

RBAC: Roles

Los roles nos permiten establecer un conjunto de operaciones permitidas sobre un conjunto de recursos API **en un namespace**.



RBAC: Roles

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
   namespace: test
   name: pod-access
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list"]
```

```yaml
kind: Role
apiVersion:
rbac.authorization.k8s.io/v1beta1
metadata:
   namespace: test
   name: ns-admin
rules:
 - apiGroups: ["*"]
   resources: ["*"]
   verbs: ["*"]
```

- Para un namespace (`namespace: test`),
- indicamos los recursos API (`resources: ["pods"]`),
- también hay que indicar el grupo del recurso (`apiGroups: [""]`), el grupo "" indica el "`core`",
- a los que permitimos las siguientes operaciones (`verbs: ["get", "list"]`)
- Se pueden usar un comodín "*"

Los grupos de la API nos clasifica los distintos recursos de la API. vienen indicado el URL. veamos algún ejemplo:

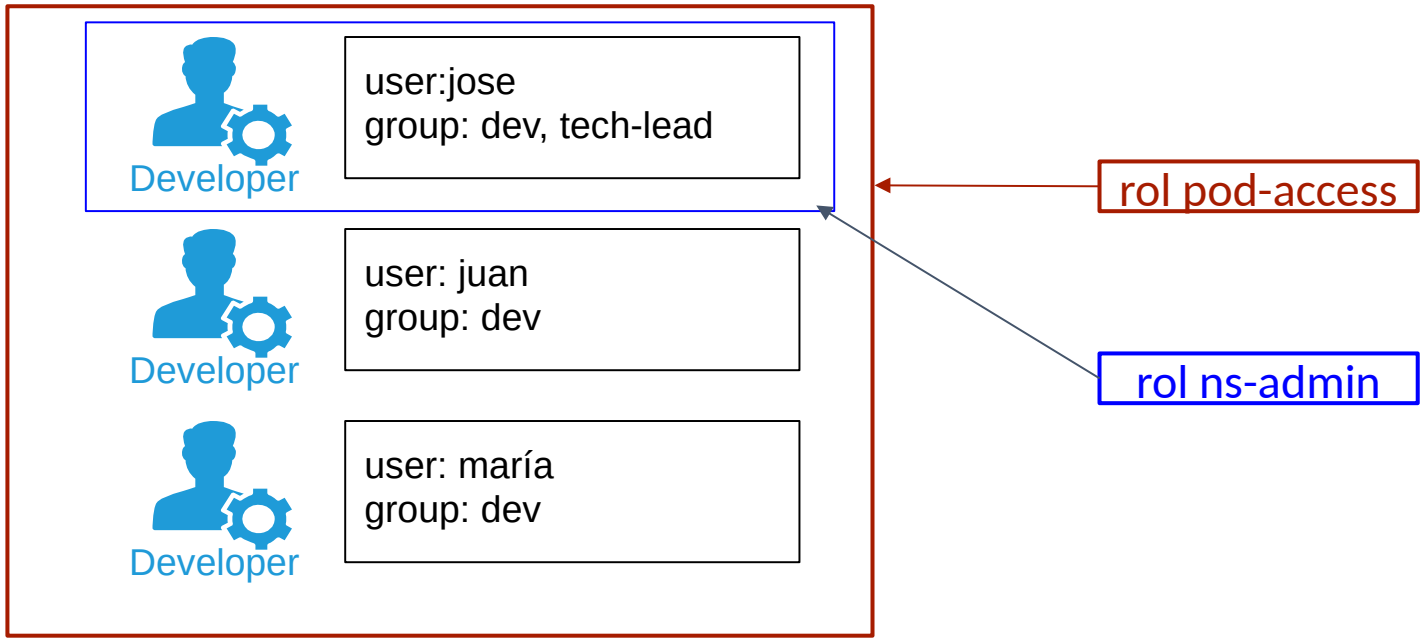
- El recurso `pod` está en el grupo `core`
- El recurso `deployment` en el grupo `apps`

Group	Version	Kind
core	v1	Pod

Group	Version	Kind
apps	v1	Deployment

RBAC: RoleBindings

El recurso RoleBindings permite asociar cada Role a un usuario o conjunto de usuarios.



RBAC: RoleBinding

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
 name: deployment-manager-binding
 namespace: test
subjects:
- kind: Group
 name: dev
 apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: Role
 name: pod-access
 apiGroup: rbac.authorization.k8s.io
```

```yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: ns-admin
 namespace: test
subjects:
- kind: User
 name: jose
 apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: Role
 name: ns-admin
 apiGroup: rbac.authorization.k8s.io
```

- Podemos indicar **distintos y varios** subjects (User,Group,...)
- Sólo podemos indicar **un role por bilding**

Ejercicio 1: Role y RoleBinding

Vamos a crear los roles como administrador:

```
kubectl config use-context minikube

kubectl create -f role-deployment-manager.yaml
kubectl get role -n proyecto1
kubectl describe role -n proyecto1

kubectl create -f rolebinding-deployment-manager.yaml
kubectl describe rolebinding -n proyecto1
```

Ahora podemos acceder como usuario1 y comprobamos si podemos trabajar con los recursos que hemos autorizado:

```
kubectl config use-context usuario1-context
kubectl get pods
No resources found.
```

RBAC: ClusterRole y ClusterRoleBinding

Si los **Role** y los **RoleBinding** nos permiten gestionar las autorizaciones a nivel de n**amespace**, los **ClusterRole** y los **ClusterRoleBinding** nos permiten gestionar las autorizaciones a nivel del **cluster**.

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
   namespace: test
   name: pod-access
rules:
 - apiGroups: [""]
   resources: ["pods"]
   verbs: ["get", "list"]

kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
 name: deployment-manager-binding
 namespace: test
subjects:
- kind: Group
 name: dev
 apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: Role
 name: pod-access
 apiGroup: rbac.authorization.k8s.io
```

```yaml
kind: ClusteRole
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:

   name: ns-admin
rules:
 - apiGroups: ["*"]
   resources: ["*"]
   verbs: ["*"]

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: ns-admin

subjects:
- kind: User
 name: jose
 apiGroup: rbac.authorization.k8s.io
roleRef:
 kind: Role
 name: ns-admin
 apiGroup: rbac.authorization.k8s.io
```

RBAC: ClusterRole y ClusterRoleBilding

Por defecto en nuestra instalación tenemos creados algunos ClusterRole y ClusterRoleBilding:

```
kubectl get clusterrole
kubectl get clusterrolebinding
```

- Por ejemplo **cluster-admin:** Para los miembros del grupo **system:masters**. Pueden hacer cualquier operación en el cluster. Por lo tantos si creas un usuario de este grupo será administrador del cluster(`openssl req ... -subj "/CN=admin2/O=system:masters"`)
- Los ClusterRole **admin, view y edit**, están pensado para ser asignado a usuarios en namespace (en un RoleBilding también se pueden asociar ClusterRole)
- Muchos de los ClusteRole son del sistema, están nombrado con el prefijo **system:...** Las modificaciones a estos recursos pueden dar lugar a fallos en el cluster. Corresponden a los diferentes componentes del cluster (kube-controller-manager, kube-proxy,...)

Default ClusterRole	Default ClusterRoleBinding	Description
cluster-admin	**system:masters** group	Allows super-user access to perform any action on any resource. When used in a **ClusterRoleBinding**, it gives full control over every resource in the cluster and in all namespaces. When used in a **RoleBinding**, it gives full control over every resource in the rolebinding's namespace, including the namespace itself.
admin	None	Allows admin access, intended to be granted within a namespace using a **RoleBinding**. If used in a **RoleBinding**, allows read/write access to most resources in a namespace, including the ability to create roles and rolebindings within the namespace. It does not allow write access to resource quota or to the namespace itself.
edit	None	Allows read/write access to most objects in a namespace. It does not allow viewing or modifying roles or rolebindings.
view	None	Allows read-only access to see most objects in a namespace. It does not allow viewing roles or rolebindings. It does not allow viewing secrets, since those are escalating.

Using RBAC Authorization

RBAC: ServiceAccount

Como hemos visto hemos gestionado las autorizaciones a usuarios y grupos. También podemos gestionar las autorizaciones de los procesos que se ejecutan en un Pod. Cuando un proceso ejecutado en un Pod necesita interactuar con la API de kubernetes (tiller de helm, monitorización con prometheus,...) también debemos limitar el nivel de acceso que tiene.

Para definir un proceso que se ejecuta en un pod y que vamos a controlar sus permisos utilizamos el objeto ServiceAccount.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account
```

- Cada vez que creamos un ServiceAccount, se crea automáticamente un API token y se guarda en el cluster.
- Podemos usarlo como "**subject**" en los RoleBinding y en los ClusterRoleBinding.
- Los ServiceAccounts los usamos en las declaraciones Pod/RS/Deployment:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: my-service-account
```

- Si no se especifica, se usará pel ServiceAccount "**default**"
- El API token es montado en el contendor.

RBAC: ServiceAccount

Veamos un ejemplo: instalamos Prometheus con helm:

```
helm install \
    --namespace=monitoring \
    --name=prometheus \
    --version=7.0.0 \
    stable/prometheus
```

kubectl get pod -n monitoring

```
NAME                                            READY   STATUS RESTARTS   AGE
...
prometheus-server-559fbf685c-cgmvm              2/2  Running   2         19s
```

kubectl get serviceaccount -n monitoring

```
NAME                         SECRETS   AGE
default                      1         19s
...
prometheus-server            1         19s
```

kubectl describe serviceaccount prometheus-server -n monitoring

```
...
Tokens:             prometheus-server-token-xx2pq
```

kubectl get secrets -n monitoring

```
NAME                                       TYPE                                  DATA   AGE
...
prometheus-server-token-xx2pq              kubernetes.io/service-account-token   3      19s
```

kubectl describe secret prometheus-server-token-xx2pq -n monitoring

kubectl describe pod prometheus-server-559fbf685c-cgmvm -n monitoring

```
...
     Mounts:
     /data from storage-volume (rw)
     /var/run/secrets/kubernetes.io/serviceaccount from prometheus-server-token-xx2pq (ro)
...
```

ResourceQuotas

Por cada **namespace** podemos indicar una cuotas de uso de recursos utilizando el objeto ReourceQuotas. En la cuota podemos limitar el uso de:

- Hardware: CPU y memoria
- Recursos de la API
- Almacenamiento
- ...

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
 name: team-vision-resquota
 namespace: team-vision
spec:
 hard: # Quotas go here
   requests.cpu: 100
   requests.memory: 400Gi
   pods: 6
   configmaps: 5
   persistentvolumeclaims: 3
   requests.storage: 500Gi
```

¿Cómo indicamos la CPU y la memoria que requiere un POD?

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: database
spec:
 containers:
 - name: db
   image: mysql
   resources: # requests here
     requests:
       memory: "64Mi"
       cpu: "250m"
...
```

Limits

Necesitamos un objeto que nos permita limitar el uso de recurso (memoria y CPU) de un pod. Vamos a usar un objeto **Limits.** Casos de uso:

- Si ponemos un `request.cpu=100`, significa que la suma de núcleos de CPU no puede superar 100, ¿pero si queremos limitar que cualquier pod no use más de un núcleo?
- Queremos limitar el uso de recursos:

```
requests:
  memory: "512Mi"
  cpu: "5"
```

```
kubectl top pod
NAME          CPU(cores)
MEMORY(bytes)
tensorflow    20           6Gi
```

Podemos limitar el uso de recursos en el Pod:

```yaml
apiVersion: v1
kind: Pod
metadata:
 name: database
spec:
 containers:
 - name: db
   image: mysql
   resources:
      requests:
        memory: 64Mi
        cpu: 250m
      limits:
        memory: 128Mi
        cpu: 500m
```

Podemos limitar el uso de recursos en el namespace:

```yaml
kind: LimitRange
metadata:
 name: mem-min-max-demo-lr
spec:
 limits:
 - max:
     cpu: 1
     memory: 512Mi
   type: Container
```

EJEMPLO 2: Quotas y Limits

kubectl create -f quota.yaml

kubctl get resourcequota
kubectl describe resourcequota mem-cpu-demo

```
...
Resource      Used  Hard
--------      ----  ----
limits.cpu    0    2
limits.memory 0    2Gi
requests.cpu  0    1
requests.memory  0     1Gi
```

kubectl describe ns default

kubectl create -f pod1.yaml

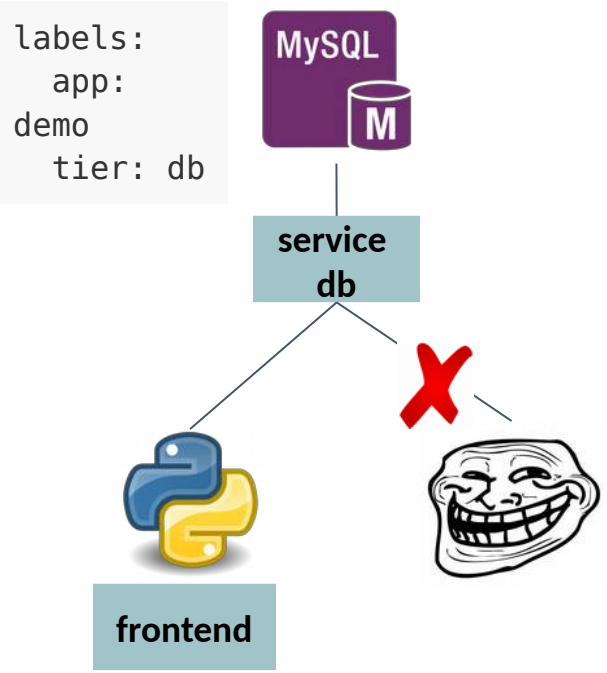
kubectl describe resourcequota mem-cpu-demo

```
...
Resource      Used   Hard
--------      ----   ----
limits.cpu    800m   2
limits.memory 800Mi  2Gi
requests.cpu  400m   1
requests.memory  600Mi  1Gi
```

kubectl create -f pod2.yaml

```
Error from server (Forbidden): error when creating "pod2.yaml": pods "quota-mem-cpu-demo-2" is forbidden: exceeded
quota: mem-cpu-demo, requested: requests.memory=700Mi, used: requests.memory=600Mi, limited: requests.memory=1Gi
```

NetworkPolicy



Queremos que el servicio que ofrece acceso a la base de datos, sólo sea accesible desde el frontend. Para ello vamos a usar el recurso NetworkPolicy.

```yaml
apiVersion: networking.k8s.io/v1
 kind: NetworkPolicy
 metadata:
   name: db-frontend-allow
 spec:
   podSelector:
     matchLabels:
       app: demo
       tier: db
   ingress:
     - from:
       - podSelector:
           matchLabels:
             app: demo
             tier: frontend
         ports:
         - protocol: TCP
           port: 3306
```

Los pod destinos, la base de datos

Los pod que pueden acceder, la app frontend

Puerto al que se puede conectar