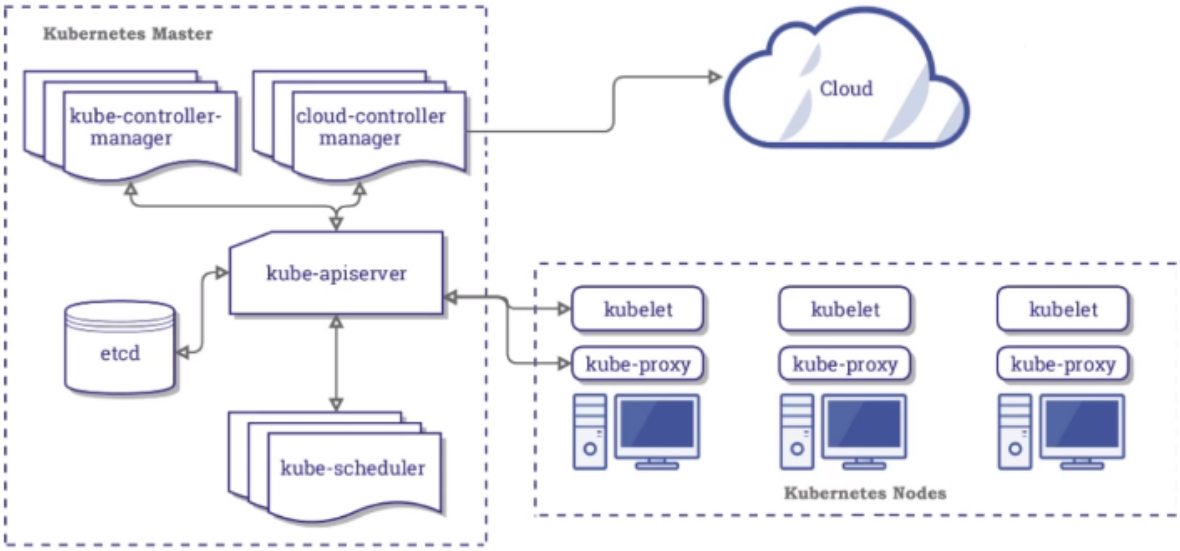
Curso kubernetes

- Curso en UDEMY: https://www.udemy.com/course/kubernetes-de-principiante-a-experto/learn/lecture/17672506?start=15#content
- Codigo del curso: https://github.com/ricardoandre97/k8s-resources
- Resumen arquitectura kubernetes https://kubernetes.io/es/docs/concepts/_print/

Arquitectura Kubernetes



Servicios del master

- [COMPLETAR]
- Apiserver - Comunica con el usuario
- scheduler - programador
- etcd - base de datos
- cloud controller manager - se usa para conectarse con servicios cloud (amazon, azure...)

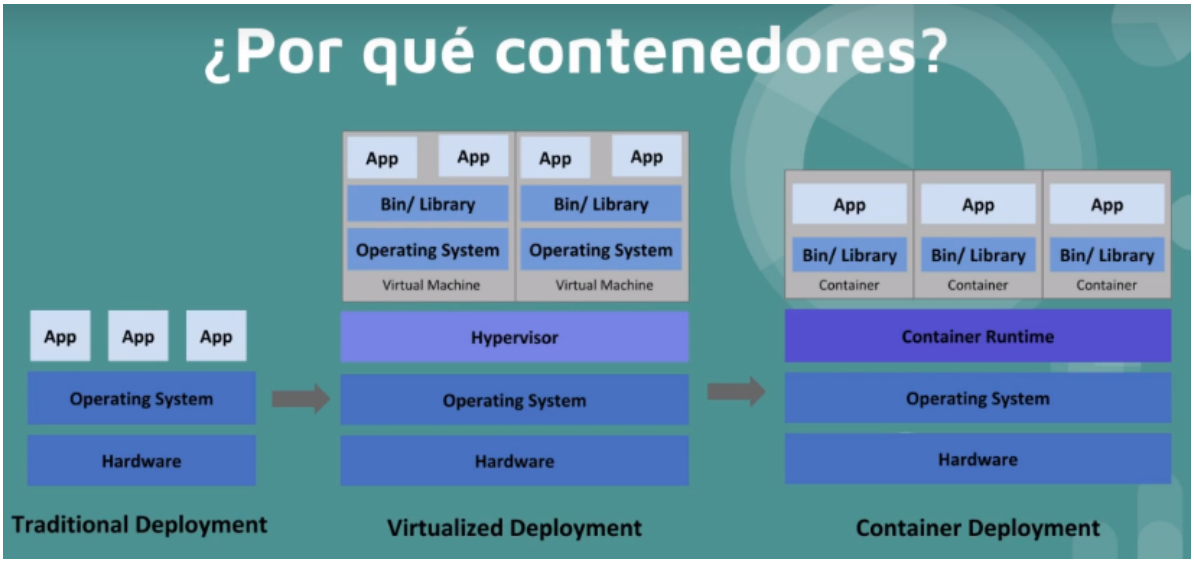
Servicios de los nodos

- Kubelet - Controla los nodos y se comunica con el master, informando del estado del nodo, levantandolos...
- Kube-proxy - maneja las redes de los nodos
- Container Runtime - Crea los contenedores en cada maquina

Herramientas para k8s:

- **kubetcl**: servicio para controlar k8s
- **minikube**: app para crear cluster en local. Requiere *kubectl*.

Pods en Kubernetes VS Contenedores de Docker



Contenedores ¿Como funcionan?

Cuando creamos un contenedor crea todos estos namespaces, que son independientes en cada contenedor:

- **IPC - Inter process communication**: Permite que los distintos procesos dentro del mismo contenedor puedan verse.
- **Cgroup**: Sistema para controlar los recursos de un contenedor (RAM, memoria...)
- **Network**: Cada contenedor crea su propia red aislada del resto
- **Mount**: Permiten montar sistemas de archivos externos
- **PID**: Gestor de pid de procesos
- **User**: Gestor de usuarios
- **UTS - Unix timesharing system**: Permite dar un hostname unico al contenedor

Gracias a ellos un contenedor es totalmente aislado del resto de contenedores. No obstante gracias a Docker el proceso de comunicar contenedores se facilitó gracias a la red bridge para establecer comunicación entre ellos.

Pods: Qué es y como funciona

Básicamente un pod permite compartir namespaces entre contenedores, pudiendo compartir un IPC, un cgroup... entre varios contenedores.

Básicamente lo que hace al crearse un pod es

- Crea un contenedor (llamemosle P) temporal
- Ese contenedor comparte sus namespaces de Network, IPC y UTS con los contenedores Docker que formen el pod.
- Gracias a ello, los contenedores pueden comunicarse a través de TCP/IP ya que ambos tienen la misma IP, por lo que pueden referirse a través de localhost
- Una vez configura los contenedores del pod, el contenedor temporal P se apaga
- **Un pod seria un grupo de contenedores que comparten los siguientes namespaces**:
    - Comparten el namespace **Network** por lo que tienen la misma ip y red
    - Comparten el namespace **IPC** por lo que pueden ver procesos entre si
    - También comparte el **UTS** por lo que tienen el mismo hostname
- No obstante, el pod no comparte el resto de namespaces por lo que el Cgroup Mount y users será independiente para cada contenedor.
- En el fondo un POD es un objeto que envuelve los contenedores docker que lo componen y que sirve para compartir ciertos namespaces entre contenedores.

Pods

Documentacion oficial: https://kubernetes.io/es/docs/concepts/workloads/pods/pod/

https://kubernetes.io/docs/reference/kubectl/cheatsheet/

- **Crear un pod**: Actualmente se recomienda el uso de *generadores*. Generalmente se usa un archivo de imagen docker para crear el pod que queremos:

  `kubeclt run --generator=run-pod/v1 podtest --image=nginx:alpine`

  Esto crea el pod `podtest` en version `v1` a partir de la imagen `nginx:alpine`. Además crea un contenedor docker con la imagen indicada que estará asociado al pod y se basará en la imagen que indicamos.
- **Ver pods**: `kubectl get pods`. Obtendremos el nombre, si está ready ,el status... podemos ver solo uno con `kubectl get pod <nombre>`. Además, e READY indica cuantos contenedores lo forman y cuantos contenedores están ready
- **Ver logs de pod**: `kubectl describe pod <nombrepod>`. Muestra toda la información del pod y los eventos (el log)
- **Ver recursos de la api**: `kubectl api-resources` muestra los recursos de la api kubectl, incluyendo pods...
- **Borrar pods**: `kubectl delete pod <nombre1> <nombre2>...`
- **Obtener manifiesto de un pod en yml**: con `kubectl get pod <nombre> -o yaml` obtiene la información que kubectl le envia a kubernetes para crear un pod.
- **Acceder a un Pod**: (NOTA: las ips de los pods solo están accesibles desde dentro del cluster.). `kubectl exec -ti <nombrepod> --sh` permite entrar en una terminal dentro del contenedor que forma el pod. En caso de que el pod este formado por varios contenedores el comando varia, debiendo incluir el nombre del contenedor al que queremos ingresar: `kubectl exec -ti <nombrePod> -c <nombreContenedor> --sh`
- **Ver logs de un pod**: Con `kubeclt logs <nombrepod>` podemos ver el log del pod. Con `-f` permite seguir los logs.

Manifiesto de un pod

Con un manifiesto podemos configurar varios pods a la vez, definiendo todo lo que queremos, manfiesto de recursos, etc...

Se crea en extension yml de forma similar a un `docker-compose`

```
# Tipo de api. Para ver las existentes con `kubectl api-versions`
apiVersion: v1
# Tpo de objeto de la api. Se pueden ver los recursos de la api con `kubectl api-
resources`
kind: Pod
# Metadata del pod incluyendo elnombre
metadata:
  name: hello
# en el Spec incluimos que contenedores vamos a incluir
spec:
  containers:
  - name: hello # Nombre del contenedor
    image: busybox # Imagen en que se basa
    command: ['sh', '-c', 'echo "Hello, Kubernetes!" && sleep 3600'] # Comando del
contenedor que sobreescribe el CMD del contenedor
- - -
# Aqui podemos seguir definiendo pods para crear más de uno por archivo
```

Si esto lo guardamos en un archivo podemos ejecutarlo con kubectl mediante `kubectl apply -f <nombreArchivo>` de forma que kubectl creará el pod

Pods con mas de un contenedor

Podemos crear pods con mas de un contenedor aunque el modelo de un contenedor por pod es el más utilizado:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello
spec:
  containers:
  - name: cont1 # Nombre contenedor 1
    image: python_3.6-alpine
    command: "Levanta un server en 8081"
  - name: cont2 # Nombre contenedor 2
    image: python_3.6-alpine
    command: "Levanta un server en 8082"
```

Tendremos dos contenedores corriendo en el pod, con la misma imagen y compartiendo los namespaces de Network, IPC y UTS, pero no el sistema de archivos ni usuarios.

Si intentasemos levantar los dos contenedores en el mismo puerto **NO PODRIAMOS** ya que al compartir la misma IP, estamos intentando levantar dos servicios con el mismo puerto.

Si lo levantasemos, al ver la infomacion con `kubectl get pods` obtendriamos:

```
NAME        READY       STATUS      RESTARTS        AGE
nombrePod   1/2         Error       0               1s
```

El error se debe a que los dos contenedores tratan de levantar un servicio en el mismo puerto:

```
--POD-------------------------------------------------------
|   IP POD (COMPARTIDA POR LOS CONTENEDORES)                |
|                                                           |
|   - CONTENEDOR 1 ----------   - CONTENEDOR 2 ----------   |
|   |   PYTHON ALPINE       |   |   PYTHON ALPINE       |   |
|   |   IP_POD:8081         |   |   IP_POD:8082         |   |
|   -------------------------   -------------------------   |
-------------------------------------------------------------
```

La diferencia cond ocker es que en este caso desde el contenedor 1 sí podemos acceder al contenedor 2 ya que comparten red: el LOCALHOST de ambos contenedores es el mismo.

Labels en pods

Los labels son metadata. Los más utilizados son `app` y `env` que indican que app es y el entorno en el que se despliegan.

```
apiVersion: v1
kind: Pod
metadata:
  name: podtest1
  labels:
    app: front
    env: dev
```

```
spec:
  containers:
  - name: cont1
    image: python_3.6-alpine
- - -
apiVersion: v1
kind: Pod
metadata:
  name: podtest2
  labels:
    app: backend
    env: dev
spec:
  containers:
  - name: cont1
    image: python_3.6-alpine
```

Esto sirve para poder filtrar por el valor de estos labels. Por ejemplo podemos filtrar aquellos cuya app es backend con `kubectl get pods -l app=backend`. Además, objetos de alto nivel como los tipo `deployment` solo conocen los pods por los labels con lo que serán necesarios para gestionarlos.

El label obligatorio sería `app`

Problemas de los pods

- **No se recuperan solos**: si cae, no se levanta otro automaticamente, y
- **No pueden replicarse por si mismos**: necesitan que alguien los controle para poder crear replicas de éstos.
- **No pueden actualizarse automaticamente**: necesitan otros objetos de más alto nivel

Objetos de kubernetes

Ver https://kubernetes.io/es/docs/concepts/overview/working-with-objects/kubernetes-objects/

Hay que entender que un POD es uno de los muchos objetos de Kubernetes. Cada uno puede definirse usando un Un objeto de Kubernetes es un "registro de intención" -- una vez que has creado el objeto, el sistema de Kubernetes se pondrá en marcha para asegurar que el objeto existe. Al crear un objeto, en realidad le estás diciendo al sistema de Kubernetes cómo quieres que sea la carga de trabajo de tu clúster; esto es, el **estado deseado** de tu clúster.manifiesto según el tipo de objeto a crear.

Cada objeto de Kubernetes incluye dos campos como objetos anidados que determinan la configuración del objeto: el campo de objeto *spec* y el campo de objeto *status*. El campo *spec*, que es obligatorio, describe el *estado deseado* del objeto -- las características que quieres que tenga el objeto. El campo *status* describe el *estado actual* del objeto, y se suministra y actualiza directamente por el sistema de Kubernetes. En cualquier momento, el Plano de Control de Kubernetes gestiona de forma activa el estado actual del objeto para que coincida con el estado deseado requerido.

Para describir objetos de kubernetes podemos usar la linea de comandos o manifiestos:

```
apiVersion: apps/v1 # Usa apps/v1beta2 para versiones anteriores a 1.9.0
kind: Deployment # Objeto tipo Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # indica al controlador que ejecute 2 pods
  template:
```

```
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

En el archivo `.yaml` del objeto de Kubernetes que quieras crear, obligatoriamente tendrás que indicar los valores de los siguientes campos (como mínimo):

- `apiVersion` - Qué versión de la API de Kubernetes estás usando para crear este objeto
- `kind` - Qué clase de objeto quieres crear
- `metadata` - Datos que permiten identificar unívocamente al objeto, incluyendo una cadena de texto para el `name`, UID, y opcionalmente el `namespace`
- También deberás indicar el campo `spec` del objeto. El formato del campo `spec` **es diferente según el tipo de objeto de Kubernetes**, y contiene campos anidados específicos de cada objeto. Puede verse aqui https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/

ReplicaSet

https://kubernetes.io/es/docs/concepts/workloads/controllers/replicaset/

Un ReplicaSet garantiza que un número específico de réplicas de un pod se está ejecutando en todo momento

Se trata de un objeto que permite crear réplicas de pods, controlando que siempre haya un numero de replicas que tiene configurado. Para ello controla mediante el `label` que pods hay levantados.

Además incluye en los pods que controla un label `owner` para indicar qué replicaSet controla estos pods, a fin de que solamente él los gestione y no lo capture otro replicaset que busque pods con el mismo label.

```
apiVersion: apps/v1 # A diferencia de los pods, la apiVersion es apps/v1 ya que su
apiAppGroup es `apps`
kind: ReplicaSet # Tipo de objeto
metadata:
  name: frontend # Nombre del replicaset
  # Estos son labels del objeto replicaset
  labels:
    app: guestbook
    tier: frontend
# Este spec es qué queremos que haga el replicaset
spec:
  # modifica las réplicas según tu caso de uso
  replicas: 3 # Numero de replicas MINIMAS
  # EL selector se utiliza para indicar qué pods tiene que controlar
  selector:
    matchLabels:
      tier: frontend # Queremos los pods con label 'tier=frontend'
  # A partir de aqui será informacion de los pods que controle el replicaset
  template:
    metadata: # Esta será la metadata que incluirá en los pods que gestione
      labels:
        tier: frontend # Los pods tendran un label 'tier=frontend'
     # Este será el spec de los pods que gestione el replicaset
    spec:
      containers:
      - name: php-redis
```

```
        image: gcr.io/google_samples/gb-frontend:v3
```

Para ejecutarlo usaremos `kubectl apply -f <nombreArchivo>`

Para obtener los replicaset podemos usar `kubectl get replicaset` o `kubectl get rs` (este nombre corto se obtiene con el listado `kubectl api-resources` en el apartado SHORTNAMES)

```
javier@DG-LP-21:~$ kubectl api-resources
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED   KIND
bindings                                       v1                                     true         Binding
componentstatuses                 cs           v1                                     false        ComponentStatus
configmaps                        cm           v1                                     true         ConfigMap
endpoints                         ep           v1                                     true         Endpoints
```

ownerReferences

La relacion entre replicasets y pods se realiza con los **owner references**. En la metadata de los pods podemos ver unos labels donde se incluye esta informacion bajo el nombre de `ownerReferences`:

```
[ricardo@localhost replicaSet]$ kubectl get pod rs-test-s924d -
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: "2019-12-20T00:05:13Z"
  generateName: rs-test-
  labels:
    app: pod-label
  name: rs-test-s924d
  namespace: default
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: rs-test
    uid: 07ff57f0-48b1-4660-a78e-ac84883b50b6
  resourceVersion: "62789"
  selfLink: /api/v1/namespaces/default/pods/rs-test-s924d
  uid: b921be4b-73df-478d-ae56-4f1fec7e604d
```

Si inspeccionamos el replicast vemos que su UID coincide con el que aparece en el del ownerReference del pod:

```
[ricardo@localhost replicaSet]$ kubectl get rs rs-test -o yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"ReplicaSet","metadata":{"
"default"},"spec":{"replicas":2,"selector":{"matchLabels":{"ap
pec":{"containers":[{"command":["sh","-c","echo cont1 \u003e i
lpine","name":"cont1"},{"command":["sh","-c","echo cont2 \u003
6-alpine","name":"cont2"}]}}}}
  creationTimestamp: "2019-12-20T00:05:13Z"
  generation: 2
  labels:
    app: rs-test
  name: rs-test
  namespace: default
  resourceVersion: "63667"
  selfLink: /apis/apps/v1/namespaces/default/replicasets/rs-te
  uid: 07ff57f0-48b1-4660-a78e-ac84883b50b6
```

Errores de Replicaset: herencia incorrecta

Puede suceder que un replicaset adopte pods que no le corresponda. Por ejemplo, si creamos manualmente unos pods que cumplan el criterio de busqueda de un replicaset, ese RS los va a adoptar autormaticamente, cumplan o no con el resto de criterios de los pods a crear por el RS (metadata, imagen del contenedor....)

Por ejemplo:

- Creamos dos pods planos con label `app` = perro

- Creamos el replicaset que maneje 3 replicas de pods con `app`=perro
- El RS adopta los 2 pods creados antes que, aunque tengan el mismo label no tienen porque estar basados en la misma imagen o cumplir sus criterios y además crea uno más que SI cumple con las condiciones del ReplicaSet

Es por ello que es importante no usar pods `planos` a mano para evitar que un replicaset posteriormente los adopte, haciendo que no cumplan las condiciones del replicaset, pero a ojos de este, sí cuenten como una instancia.

Otros errores de Replicaset

- **El replicaset no puede cambiar los pods una vez creados**. Si queremos actualizar por ejemplo la imagen, si el RS ya ha creado el numero de replicas que se le requieren, no va a crear nuevas replicas, por lo que los pods preexistentes quedarán con la imagen antigua con la que se crearon. Si luego se borra alguno, los nuevos pods sí tendrán la nueva configuración.

Deployment

https://kubernetes.io/es/docs/concepts/workloads/controllers/deployment/

Un deployment es un objeto de más alto nivel que el ReplicaSet y que controla estos

```
DEPLOYMENT  --> REPLICASET  --> POD
                            |-> POD
```

Un controlador de *Deployment* proporciona actualizaciones declarativas para los Pods y los ReplicaSets. Cuando describes el *estado deseado* en un objeto Deployment, el controlador del Deployment se encarga de cambiar el estado actual al estado deseado de forma controlada. Puedes definir Deployments para crear nuevos ReplicaSets, o eliminar Deployments existentes y adoptar todos sus recursos con nuevos Deployments.

```yaml
apiVersion: apps/v1 # api del objeto Deployment
kind: Deployment # tipo de objeto
metadata:
  name: nginx-deployment # Se crea un Deployment denominado `nginx-deployment`
  labels:
    app: nginx
spec:
  replicas: 3 # El Deployment crea tres Pods replicados,
  selector:
    matchLabels:
      app: nginx # identifica los Pods que debe gestionar. En este caso, en este caso
por el label `app: nginx`
  template: # define como van a ser los pods que maneje el deployment
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

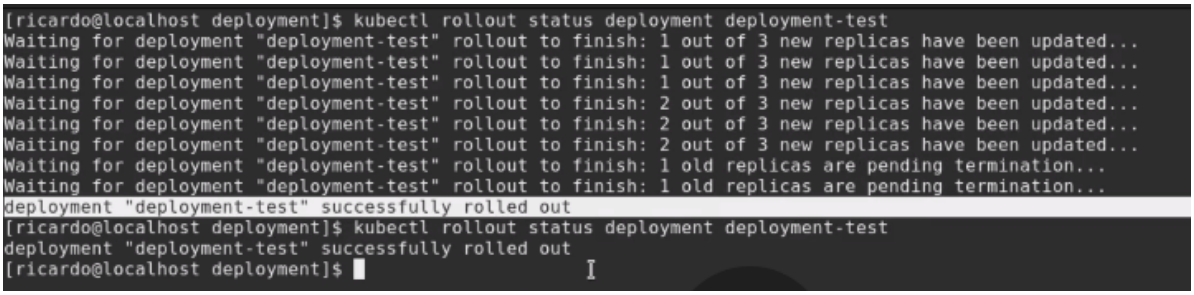
Cuando se crea el Deployment, se crea el Replicaset que cumple sus criterios y las replicas que debe disponer.

Podemos manejar los deployments con los mismos comandos: `kubectl get deployments`, `kubectl get deployment --show-labels`, delete...

En caso de los Deployment, no tienen owners ya que son objetos de más alto nivel, por lo que si vemos su info con `kubectl get deployment <name> -o yaml` no van a tener `ownerReferences`

Cambios en Deployment

A diferencia del ReplicaSet, en caso de haber cambios, el Deployment sí que actualiza los cambios. Lo que hace al cambiar la configuracion y lanzarlo, es matar los pods de la versión anterior y lanzarlos con la nueva configuracion. Por ello, si realizamos un `apply` después de hacer un cambio al yml, el replicaset y los pods que genera se adaptan al cambio



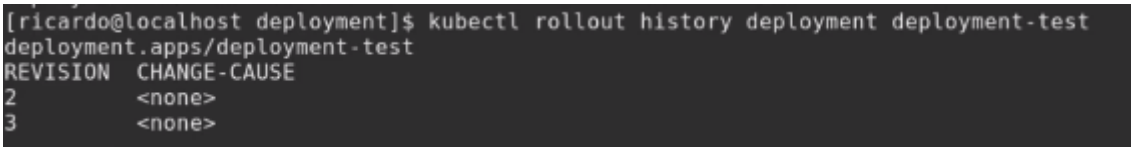
Lo que vemos que hace es:

- Arranca un pod de la nueva version y comprueba si está ok
- Si está ok, borra un pod de la versión anterior para mantener siempre el mismo numero de réplicas.
- Lo va haciendo con el resto de replicas, hasta tener el numero de replicas deseado con la nueva versión

De esta forma, a diferencia del ReplicaSet, sí que refleja los cambios del deployment

Historico de despliegues

Cada vez que realizamos un cambio en un deployment, se genera un replicaset nuevo, pero los anteriores **no se eliminan**. Es por ello que podemos consultar las revisiones de los deployment con `kubectl rollout hystory deployment <nombre>` :



Si ademas vemos los Replicaset que se han creado en cada deployment tenemos que han sido dos:

La idea de mantener estos Replicaset es poder volver a una revisión anterior.

Por defecto, se mantienen 10 replicaset pero el valor puede modificarse con `spec.revisionHistoryLimit`

Change-cause

El apartado **Change-cause** permite saber qué se cambió en el deployment, que deberá indicarse de alguna forma:

- **Mediante un flag** `--record` en la orden `kubectl apply` que registrará el comando como la causa del cambio
- **Crear una anotación en el deployment**: en la metadata del deployment, incluir un campo `metadata:annotations:kubernetes.io/change-cause:`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    kubernetes.io/change-cause: Causa del cambio
  name: nginx-deployment
  labels:
    app: nginx
```

- Existe otra forma con otra anotacion, pero es más tedioso (ver en la doc).

Rollback de un deployment

En ocasiones necesitas revertir un Deployment; por ejemplo, cuando el Deployment no es estable, como cuando no para de reiniciarse. Por defecto, toda la historia de despliegue del Deployment se mantiene en el sistema de forma que puedes revertir en cualquier momento (se puede modificar este comportamiento cambiando el límite de la historia de revisiones de modificaciones).

Para ello podemos usar el comando: `kubectl rollout undo <nombreDeployment> --to-revision=2` indicando que queremos volver a la revisión 2 del deployment.

Services

https://kubernetes.io/docs/concepts/services-networking/service/

Los servicios son abstracciones que observa los pods segun su label para servir de puente entre el usuario y los pods, de forma que podamos utilizarlos desde un solo punto de entrada, independientemente de que pod concreto utilicemos. Es un "balanceador" que distribuye la carga de peticiones entre los pods del mismo tipo, para poder ofrecer al usuario un solo punto de entrada (*endpoint*)

```
-----------------------------------
| |--------| |--------| |-------| |
| | POD 1  | | POD 2  | | POD 3 | |  <------- ENDPOINT <----- SERVICE <----- USER
| |--------| |--------| |-------| |           Ips Pods        Ip Fija
|-----------------------------------
  IPs variables para cada POD
```

Endpoints

Un servicio va a tener una IP única que **no cambia durante el tiempo**. Aunque las IPs de los pods cambie, el servicio puede consultarlos, dado que al crear un servicio se crea un **endpoint**, a los que el servicio va a asociar las ips de los pods. Podemos decir que un **endpoint** es un listado de ips d elos pods que cumplen la condicion del servicio, ya que el servicio actualiza automaticamente este listado cada vez que un pod cae o se levanta.

Creación

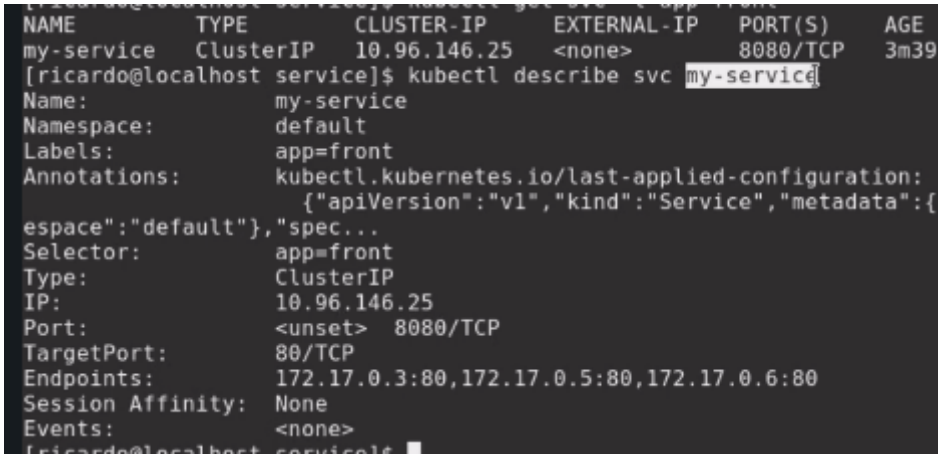
```yaml
apiVersion: v1 # Version de la apiversion
kind: Service # Tipo de objeto: servicio
metadata: # metadata del servicio
  name: my-service
  labels:
    app: front
spec:
  type: ClusterIP # Tipo de servicio (ver mas adelante)
  selector:
    app: MyApp # que label vamos a utilizar para observar los pods
  ports: # Puerto del servicio
```

```
    - protocol: TCP
      port: 80 # Puerto que el servicio expone
      targetPort: 9376 # Puerto DEL POD a consumir
```

Con esto se crea un servicio en el puerto 80 que consume los pods con el label `app=MyApp` a través del puerto 9376.

Ojo: El servicio solamente se encarga de enrutar con una ip fija los pods. Para mantener las replicas y controlarlas seguimos necesitando un Deployment (con su consecuencte ReplicaSet)

Un ejemplo de informacion de un servicio:



Tenemos:

- El **nombre, labels y anotaciones** del servicio
- El **selector** que se usa para elegir los pods (en este caso app=front)
- La **IP y el puerto de escucha** del servicio (IP / Port)
- El **puerto de escucha de los pods** (TargetPort)
- Los **endpoint** (es decir, las ips de los pods que cumplen la condicion del servicio)

De la misma forma que en el ReplicaSet, no se recomienda crear Pods planos, y aque si cumple el criterio de busqueda del servicio, este lo adoptara aunque no cumpla con el spec del servicio

Servicios y DNS

El servicio hereda un dns con el nombre del servicio, de forma que si hacemos una llamada a `http://nombreServicio:PuertoEscucha` el servicio redirige a los pods.

Tipos de servicios

https://kubernetes.io/docs/concepts/services-networking/service/

Existen distintos tipos de servicios en función de qué queremos que haga y que exponga.

- **ClusterIp**: Es una IP virtual que k8s le asigna al servicio , mantenida por Kubernetes, pero interna al clúster (no vamos a poder acceder desdefuera del cluster). Solo sirve para la comunicacion interna entre servicios. Es el valor por defecto de los servicios.
- **NodePort**: Sirve para exponer un servicio fuera del cluster a través de un puerto de un nodo. Lo que hace es crear un ClusterIp y además abrir un puerto de un nodo para poder acceder a los servicios de los pods. Nodeport expone por defecto un rango de puertos desde 30000-32767, por lo que usando la ip del cluster y el puerto que expone el NodePort podemos acceder al servicio de los pods.
- **LoadBalancer**: Es un balanceador de carga. Solo se crean balanceadores externos en servicios cloud (amazon, Azure, Google Cloud...). Lo que hace es provisionar un balanceador de carga, que abre nodeports en cada nodo de forma que, accediendo al balanceador él se encarga de elegir que nodo utilizar. Para entrar al nodo usa un **servicio Nodeport** que a su vez usa un ClusterIp, por lo que crear un LoadBalancer requiere crear los servicios NodePort.

Despliegue de servicios

Para hacer un despliegue, solo necesitaremos un documento `yaml` donde deberemos definir:

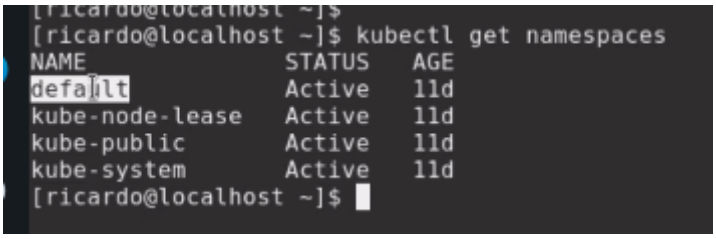
- Un objeto **Deployment** que maneje:
  - El nº de replicas
  - El criterio de seleccion de pods
  - El contenedor o contenedores en los que se basa nuestros pods
- Un objeto **Service** que exponga y mantenga fija la IP de los pods manejados por el **Deployment**. Si queremos exponerlos, deberemos hacerlo usando un objeto NodePort.

Namespaces y Context

Namespaces

Es una separación lógica que nos brinda un scope/entorno. Permite aislar objetos de kubernetes en distintos contextos. Permite una mejor organización, crear distintos entornos en un mismo clúster y sobre todo asignar recursos por namespace, de forma que podemos definir numero de pods, RAM consumida, usuarios con acceso....

Namespaces por defecto



Por defecto k8s genera 3 namespaces:

- **default**: el namespace donde caen todos los objetos creados que no tiene un namespace definido. Cualquier operacion de obtener objetos, borrar, etc... que no indique un namespace concreto realiza la operación sobre este namespace.
- **kube-public**: es una convención de k8s al que todos los usuarios por defecto pueden acceder. No se recomienda usarlo
- **kube-system**: contiene los objetos del cluster de kubernetes. No deberiamos crear ni borrar objetos de este namespace.

Crear namespaces

https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/

Las dos mas usadas son:

- Usando la linea de comandos: `kubectl create namespace <nombre>`.
- Utilizando un `yaml`:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: <insert-namespace-name-here>
  labels:
    name: <nombre>
```

Realizando un `kubectl apply -f <nombreArchivo>` creamos el namespace del archivo.

Objetos de un namespace

Para crear objetos en un namespace sirve con incluirlo a la hora de crear los objetos. Para ello, en el propio archivo yaml de definicion del objeto podemos incluir el namespace al que queremos que pertenezca:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: <nombreNamespace> # Nombre del namespace al que queremos asociarlo
  name: nginx-deployment
  labels:
    app: nginx
```

DNS en los servicios de un namespace

Recordemos que cuando creamos un servicio podemos acceder a través de dns del propio servicio, usando su nombre. En caso de que los servicios vivan en un namespace, los dns se crean con **el nombre del servicio + nombre namespace + svc.cluster.local**. Por ejemplo:

```
apiVersion: v1
kind: Service
metadata:
  namespace: test
  name: my-service
  ...
```

Este servicio `my-service` en el namespace `test` estaría accesible en `http://my-service.test.svc.cluster.local`

Si no le pasamos este nombre de namespace y el sufijo svc.cluster.local, trata de acceder al servicio pero dentro del namespace default.

Context

https://kubernetes.io/es/docs/concepts/configuration/organize-cluster-access-kubeconfig/

Un elemento *context* en un archivo kubeconfig se utiliza para agrupar los parámetros de acceso bajo un nombre apropiado. Cada contexto tiene tres parámetros: clúster, Namespace y usuario. Por defecto, la herramienta de línea de comandos `kubectl` utiliza los parámetros del *contexto actual* para comunicarse con el clúster.

Podemos revisar los contextos que tenemos consultando la configuracion de kubernetes en `.kube/config` o con la orden `kubectl config view`. Por ejemplo:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0t...
    server: https://192.168.6.153:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
```

```
users:
```

Vemos que tenemos un contexto `kubernetes-admin@kubernetes`, sin namespace definido, que aparece como current-context. Esto significa que a la hora de trabajar, por defecto estamos trabajando sobre ese contexto, que en nuestro caso es un server en la ip 192.168.6.153:6443.

Podemos crear distintos contextos y asociarlos a namespaces, de forma que eligiendo uno como `current-context` nos permita trabajar en un namespace por defecto sin tener que pasar `-n` de continuo.

```
# set a context utilizing a specific username and namespace.
kubectl config set-context gce --user=cluster-admin --namespace=foo \
  && kubectl config use-context gce
```

Límites en pods

Cuando definimos pods sin limites, estos pueden utilizar todos los recursos de los nodos, pudiendo tumbar los nodos. Para evitar esto, podemos definir limites de RAM (en unidades de espacio (b, Mb, Gb) y CPU (en millicores, es decir, milesimas partes de core, o en tantos por 1 de cpu).

Para definir los límites de un pod, deberemos definir dos parámetros fundamentales:

- **Limits**: Sería cual es el límite de recursos de los que va a disponer el pod.
- **Request**: Dice cuantos recursos van a estar garantizados, es decir, que siempre va a tener disponibles y dedicados esos recursos.

En caso que el pod sobrepase el límite, k8s se va a encargar de gestionar el pod (normalmente reiniciándolo) según la política establecida por k8s.

Un ejemplo de pod limitado con limites y request:

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources: # Recursos definidos para el pod
      requests: # Request
        memory: "64Mi" # 64 megas
        cpu: "250m" # 250 milicores
      limits: # Limits
        memory: "128Mi"
        cpu: "0.5"  # Modo alternativo, pedimos el 50% de una cpu
```

Como diferencia entre el limite de CPU y RAM es que en caso de limites de CPU, nunca van a poder sobrepasar el limite del nodo. Ademas si ningun nodo tiene la CPU o la RAM solicitada por un pod, el pod se quedará en estado `pending`, dado que el scheduler de kubernetes no puede encajarlo en ningun nodo.

Quality of Services Classes

https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/https://kubernetes.io/docs/tasks/configure-pod-container/quality-service-pod/

Se trata de clases que clasifican los pods dependiendo de su configuracion en limites. Cuando k8s crea un pod asigna una de las clases QoS en funcion de sus limites, pudiendo ser:

- **Guaranteed**: Mismo valor de limites y request: el pod siempre tiene garantizado el limite.
- **Burstable**: Aquellos cuyo límite es > que la request, pudiendo aumentar en algun momento
- **BestEffort**: Aquellos que no tienen definidos los límites. Son los mas peligrosos ya que pueden tumbar los nodos

Objetos k8s para control de límites

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/

LimitRange

https://kubernetes.io/es/docs/concepts/policy/limit-range/

Permiten definir limites por defecto para los pods creados sin limites definidos. Además permite controlar configuraciones a nivel de objeto, indicando el minimo o máximo de recursos. En caso de tratar de crear pods que superen estos limites, k8s impedirá la creación de los pods en función de los límites establecidos por LimitRange.

- **Definicion de valores por defecto a pods BestEffort (sin limites definidos)**: Podemos definir un limitrange que controle los valores por defecto de los pods en caso de crearse sin indicar límites de forma expresa

  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-namespace/

  ```yaml
  apiVersion: v1
  kind: Namespace # Creamos un namespace donde va a aplicar el LimitRange
  metadata:
    name: dev
    labels:
      name: dev
  ---
  apiVersion: v1
  kind: LimitRange
  metadata:
    name: cpu-min-max-demo-lr
    namespace: dev # Lo definimos en el namespace dev
  spec:
    limits:
    - default: # Indica los limites por default
        memory: 512Mi
      defaultRequest: # Indica las request por default
        memory: 256 Mi
      type: Container # Tipo aplicado: Contenedores dentro del pod
  ```

  Una vez creado podemos usarlo con apply: `kubeclt apply -f <archivoYaml>`

- **Definicion de valores máximos y mínimos**: Podmeos definir los valores de memoria y cpu maximos y minimos para cada pod creado dentro del namespace definido:

  https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-constraint-namespace/

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: mem-min-max-demo-lr
  namespace: dev # Lo definimos en el namespace dev
spec:
  limits:
  - max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

ResourceQuota

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.21/#resourcequota-v1-core

A diferencia de un **LimitRange**, que afecta a los pods a nivel de objeto, un **ResourceQuota** afecta a nivel de namespace, marcando el máximo de cpus y ram que utilizan los objetos definidos en un namespace, que a su vez tendrán definidos sus límites mediante los limitRange.

- **LimitRange** > define recursos a nivel de POD
- **ResourceQuota** > define recursos a nivel de Namespace

Un ejemplo de resourceQuota para un namespace sería:

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-memory-cpu-namespace/

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: mem-cpu-demo
  namespace: nombreNamespace
spec:
  hard:
    # Especificaciones para todo el namespace
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

> **NOTA**: Cuando definimos un ResourceQuota en un namespace deberemos definir en cada POD su limit y request para que el resourceQuota pueda sumarlo y ver que no se superan.

Además de limitar los recursos, podemos limitar el **total de pods que se pueda crear en el namespace**.

https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/quota-pod-namespace/

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-demo
  namespace: nombreNamespace
spec:
  hard:
    pods: "2"  # Limitamos el numero de pods en el namespace
```

en este caso, cuando creemos un deployment o un replicaService, si el numero de replicas intenta superar al numero de pods definidos por el resourceQuota, k8s no nos permitirá crear el pod ya que no se puede superar la quota definida en el ResourceQuota.

Probes (diagnosticos)

Las Probes son diagnosticos realizados por el servicio *Kubelet* que corre en cada pod. Kubelet define distintos Probes dentro de cada pod para comprobar de forma periódica el estado del mismo. Esto puede hacerse de tres formas:

- Kubelet puede ejecutar un comando,
- También hacer una llamada TCP a un puerto del contenedor del pod
- Por último puede hacer una llamada http al pod.

Para ello se deben incluir en la definicion del pod, indicando el tipo de operacion y sus condiciones.

TIpos

Existen distintos tipos de probes, según lo que queramos comprobar:

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

- **Liveness**: Es una prueba que ejecuta kubelet en el contenedor en un intervalo de tiempo y que asegura que el contenedor está respondiendo como debería responder.
- **Readyness**: es una prueba que se ejecuta en el pod cuando se crea, antes de que se incluya como un endpoint válido. No s ayuda a garantizar que está listo para recibir tráfico. Por ejemplo, si un pod no supera una prueba readyness **NO VA A REINICIARSE**. Simplemente se elimina del listado de endpoints válidos hasta que supere la prueba de nuevo, dado que simplemente puede estar saturado de forma temporal. Estas pruebas siguen siendo compatibles con pruebas Liveness.
- **Startup**: Si el startup se define en el manifiesto, el Liveness y el Readyness no se ejecutarán. Este tipo de probe se utiliza en servicios que tardan mucho tiempo en levantarse. Lo que hace es indicar que se espere a que se levante completamente. Una vez lo pase, ya se ejecutarán los liveness y los readyness.

Liveness

Un ejemplo de manifiesto de un probe Liveness con ejecucion de comando sería:

```
apiVersion: v1
kind: Pod # Recordemos que los probes se ejecutan a nivel de pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox # Imagen del contenedor
    args:
    - /bin/sh
    - -c
    livenessProbe: # Esta es la probe de tipo liveness
      exec:
        command: # En este caso ejecuta el comando `cat /tmp/healty` cada 5 segundos
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

En este caso, mientras el comando `cat /tmp/healty` sea correcto el pod seguirá arriba

Por otro lado, un Liveness con ejecución de **peticion TCP** sería:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: k8s.gcr.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe: # Indicamos la prueba tipo liveness
      tcpSocket: # Indicamos que será una llamada tcp
        port: 8080 # Indicamos el puerto
      initialDelaySeconds: 15
      periodSeconds: 20
```

Y una por **llamada http**:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet: # Indicamos que es una llamada http GET
        path: /healthz # Incluimos el path
        port: 8080 # Incluimos el puerto
        httpHeaders: # Podemos incluir custom headers
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3 # Delay inicial al arrancar el pod
      periodSeconds: 3 # Ejecución periódica tras arrancar
```

Readyness

Un ejemplo de prueba readyness podría ser:

```
readinessProbe:
  exec:
    command:
    - cat
    - /tmp/healthy
  initialDelaySeconds: 5
  periodSeconds: 5
```

ConfigMaps & Environment Variable

Variables de entorno

https://kubernetes.io/docs/tasks/inject-data-application/define-environment-variable-container/

Podemos definir variables de entorno directamente sobre nuestros pods:

```
apiVersion: v1
kind: Pod
metadata:
  name: envar-demo
  labels:
    purpose: demonstrate-envars
spec:
  containers:
  - name: envar-demo-container
    image: gcr.io/google-samples/node-hello:1.0
    env:                                    # Define que son variables de entorno
    - name: DEMO_GREETING                   # Key
      value: "Hello from the environment"   # Value
    - name: GREETING
      value: "Warm greetings to"
    - name: HONORIFIC
      value: "The Most Honorable"
    - name: NAME
      value: "Kubernetes"
    command: ["echo"]
    args: ["$(GREETING) $(HONORIFIC) $(NAME)"] # Podemos usar la variables de entorno
directamente
```

Estas variables de entorno son accesibles desde cualquier parte del contenedor del pod, estarán disponibles globalmente en el contenedor.

Capturar variables de entorno y incluirlas dentro del pod

En este caso podemos utilizar la propia información del pod para definir variables de entorno dentro del mismo. Simplemente a la hora de definirlas debemos referirlas a la propiedad del pod . Si obtenemos la definicion del pod con `kubectl get pods <nombre> -o yaml` y sus propiedades, podemos reintroducirlas como variables de entorno referenciandolas en el manifiesto del pod.

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-envars-fieldref
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "sh", "-c"]
      args:
      - while true; do
          echo -en '\n';
          printenv MY_NODE_NAME MY_POD_NAME MY_POD_NAMESPACE;
          printenv MY_POD_IP MY_POD_SERVICE_ACCOUNT;
          sleep 10;
        done;
      env:
        - name: MY_NODE_NAME
          valueFrom:
            fieldRef:
```

```
              fieldPath: spec.nodeName # Indicamos que la variable de entorno obtenga el
valor de
                                       # la propiedad spec.nodeName del pod
        - name: MY_POD_NAME
          valueFrom:
            fieldRef:
              fieldPath: metadata.name
        - name: MY_POD_NAMESPACE
          valueFrom:
            fieldRef:
              fieldPath: metadata.namespace
        - name: MY_POD_IP
          valueFrom:
            fieldRef:
              fieldPath: status.podIP
        - name: MY_POD_SERVICE_ACCOUNT
          valueFrom:
            fieldRef:
              fieldPath: spec.serviceAccountName
  restartPolicy: Never
```

ConfigMap

https://kubernetes.io/es/docs/concepts/configuration/configmap/

Un ConfigMap es un objeto de la API que permite almacenar la configuración de otros objetos utilizados. Aunque muchos objetos de kubernetes que tienen un `spec`, un ConfigMap tiene una sección `data` para almacenar items, identificados por una clave, y sus valores. La idea es separar las configuraciones de los pods para que éstos consuman la configuración de los Configmaps en lugar de declararla ellos mismos.

Para crearlos podemos:

- Hacerlo desde un **archivo** o una **carpeta** con varios archivos

```
# Create the local directory
mkdir -p configure-pod-container/configmap/

# Download the sample files into `configure-pod-container/configmap/` directory
wget https://kubernetes.io/examples/configmap/game.properties -O configure-pod-
container/configmap/game.properties
wget https://kubernetes.io/examples/configmap/ui.properties -O configure-pod-
container/configmap/ui.properties

# Create the configmap
kubectl create configmap game-config --from-file=configure-pod-container/configmap/
```

- Utilizar un manifiesto**: como por ejemplo

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  player_initial_lives: "3"
  ui_properties_file_name: "user-interface.properties"
  # file-like keys
  game.properties: | # Con | indicamos que se define como un archivo
    enemy.types=aliens,monsters
    player.maximum-lives=5
```

```
  user-interface.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
```

Para que un pod consuma un configmap podemos: (https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/)

- **Argumento en la linea de comandos** como entrypoint de un contenedor
- **Variable de entorno de un contenedor** como por ejemplo:

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      env:
        - name: SPECIAL_LEVEL_KEY # Nombre que tendrá la variable de entorno dentro del
pod
          valueFrom:
            configMapKeyRef: # Indicamos que lo lea del config map
              name: special-config # Nombre del configMap
              key: special.how # Nombre del valor que queremos consumir del configmap
  restartPolicy: Never
```

- **Como fichero en un volumen de solo lectura**, para que lo lea la aplicación

```
apiVersion: v1
kind: Pod
metadata:
  name: dapi-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "ls /etc/config/" ]
      volumeMounts:
      - name: config-volume
        mountPath: /etc/config # Ruta del volumen donde ira el configmap dentro del
contenedor
  volumes:
    - name: config-volume # Nombre de volumen de configuracion
      configMap:
        name: special-config # Nombre del configMap
        items: # Elementos del configmap que vamos a consumir. SI NO LO DEFINIMOS LOS
MONTA TODOS
        - key: # Nombre del elemento en el configmap
          path: # Nombre del elemento que va a tener en el archivo dentro del pod
  restartPolicy: Never
```

- **Escribir el código para ejecutar dentro de un Pod que utiliza la API para leer el ConfigMap**

Estos diferentes mecanismos permiten utilizar diferentes métodos para modelar los datos que se van a usar. Para los primeros tres mecanismos, el kubelet utiliza la información del ConfigMap cuando lanza un contenedor (o varios) en un Pod.

Para el cuarto método, tienes que escribir el código para leer el ConfigMap y sus datos. Sin embargo, como estás utilizando la API de kubernetes directamente, la aplicación puede suscribirse para obtener actualizaciones cuando el ConfigMap cambie, y reaccionar cuando esto ocurra. Accediendo directamente a la API de kubernetes, esta técnica también permite acceder al ConfigMap en diferentes namespaces.

Además, estos mecanismos son compatibles, pudiendo definir unas variables de entorno en el pod de ambas formas simultáneamente:

```
apiVersion: v1
kind: ConfigMap # Configmap como archivo
metadata:
  name: nginx-config
  labels:
    app: front
data:
  nginx: |
    server {
        listen       9090;
        server_name  localhost;
        location / {
            root   /usr/share/nginx/html;
            index  index.html index.htm;
        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   /usr/share/nginx/html;
        }
    }
---
apiVersion: v1
kind: ConfigMap # Configmap como clave valor
metadata:
  name: vars
  labels:
    app: front
data:
  db_host: dev.host.local
  db_user: dev_user
  script: | # definimos configmap como archivo usando |
    echo DB host es $DB_HOST y DB user es $DB_USER > /usr/share/nginx/html/test.html
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-test
  labels:
    app: front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: front
  template:
    metadata:
      labels:
        app: front
    spec:
      containers:
        - name: nginx
          image: nginx:alpine
```

```
          env:
            - name: DB_HOST # Montamos una variable de entorno usando configmap `vars`
              valueFrom:
                configMapKeyRef:
                  name: vars
                  key: db_host
            - name: DB_USER
              valueFrom:
                configMapKeyRef:
                  name: vars
                  key: db_user
          volumeMounts:
          - name: nginx-vol # Montamos como volumen el archivo del configMap nginx-vol
            mountPath: /etc/nginx/conf.d
      volumes:
        - name: nginx-vol # Definimos el volumen donde irá montado el nginx-vol
          configMap:
            name: nginx-config # Decimos el configmap que se montará
            items:
            - key: nginx # Indicamos la key que usará y el path donde se montará
              path: default.conf
```

Secrets

https://kubernetes.io/docs/concepts/configuration/secret/

Un secret de kubernetes es un configmap creado para guardar información sensible, estando totalmente aislado del pod, pudiendo montarse como variable de entorno o como volumen. Puede haber distintos tipos (Basic, TLS, SSH (ver documentacion)).

Creación

Los secretos pueden definirse por linea de comandos o desde archivos, como ConfigMaps, aunque es importante que **su nombre debe poder ser un nombre de subdominio DNS**.

- Desde la linea de comandos

```
echo -n 'admin' > ./username.txt            # Creamos los archivos
echo -n '1f2d1e2e67df' > ./password.txt

kubectl create secret generic db-user-pass \    # Se los pasamos al secreto al crearlos
  --from-file=./username.txt \
  --from-file=./password.txt
```

- Desde un manifiesto en texto plano:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data: # Usando data SE GUARDA COMO TEXTO PLANO
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

- Desde un manifiesto usando **stringdata** para que vaya codificado: En este caso, aunque lo pasemos como texto plano, el secret LO CODIFICA A BASE 64

```
# For example, if your application uses the following configuration file:

apiUrl: "https://my.api.com/api/v1"
username: "<user>"
password: "<password>"

# You could store this in a Secret using the following definition:

apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  config.yaml: | # Archivo que creará con los datos
    apiUrl: "https://my.api.com/api/v1"
    username: <user>
    password: <password>
```

No obstante en este caso, seguimos guardando como texto plano en el manifiesto. Para hacerlo de forma totalmente segura podemos **usar variables de entorno** que contengan los valores:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
stringData:
  username: $USER
  password: $PASSWORD
```

Para cambiar las variables de entorno en kubernetes, podemos usar la orden **envsubs** de k8s que lo que hace es leer un archivo y sustituir las variables de entorno que estén definidas en el sistema por sus valores:

- Imaginemos un entorno con $USER = javier y $PASSWORD = contraseña
- Usamos `envsubs` para incluir los valores en el manifiesto: `envsubst < ManifiestodelSecret.yaml > archivodesalida.yaml`
- De esta forma, el archivo de salida será el mismo con las variables sustituidas, con lo que en Git no tenemos porque guardar los datos sensibles:

  ```
  piVersion: v1
  kind: Secret
  metadata:
    name: mysecret
  type: Opaque
  stringData:
    username: javier
    password: contraseña
  ```

Uso en pods

Podemos montar los secretos **como volumenes** o **como variables de entorno**. Para ello:

```
# Creamos el secreto
apiVersion: v1
kind: Secret
```

```yaml
metadata:
  name: secret1
type: Opaque
stringData:
  username: admin
  password: "12345678"
---
# Creamos el pod que lo consume
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: nginx:alpine
    env:
      - name: USERTEST # CONSUMO DE SECRETOS COMO VARIABLES DE ENTORNO
        valueFrom:
          secretKeyRef: # Indicamos que vamos a crear de un secreto
            name: secret1  # Indicamos de qué secreto
            key: username # Indicamos la key
      - name: PASSWORDTEST
        valueFrom:
          secretKeyRef:
            name: secret1
            key: password
    volumeMounts: # CONSUMO DE SECRETOS COMO ARCHIVOS
    - name: test # Nombre del volumen que se montará para montar el archivo del secreto
      mountPath: "/opt" # Será la carpeta del pod donde creará el archivo (creará uno
por key)
      readOnly: true
  volumes: # Aqui es donde usamos el archivo del secreto montado dentro del pod
  - name: test # Nombre del volumen que vamos a usar
    secret: # Indicamos que es un secreto
      secretName: secret1 # nombre del secreto que vamos a usar
      items:
      - key: username
        path: user.txt
```

Persistencia de datos

Volúmenes

https://kubernetes.io/docs/concepts/storage/volumes/

Los pods son objetos sin estado: se montan y desmontan sin importar los datos que existen. En caso de querer persistir los datos, necesitamos **volúmenes**, que serán objetos abstractos e independientes de los pods, para que se guarde cuando los pods se desmonten (de forma parcida a Docker)

Existen distintos tipos de volumenes:

- **EmptyDir:** Es un directorio vacio que se crea en el Pod y existe mientras el pod exista. Si el pod muere, el directorio muere. Por ejemplo, podemos usar un emptydir para usar como caché:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
```

```
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache # volumen que se va a montar en el pod
      name: cache-volume # Nombre del volumen montado
  volumes:
  - name: cache-volume # Referenciado  arriba
    emptyDir: {} # Tipo de volumen emptydir. Solo existe mientras existe el pod
```

- **HostPath**: Hace referencia a un path dentro del nodo donde está definido el pod. En este caso si se elimina el Pod y se levanta otro con el mismo hostpath puede consumir su información. La información **reside en el nodo**, por lo que será un path del nodo, con lo que si el pod trabaja en el nodo1, cae y se levanta en el nodo2, consumirá la carpeta del nodo2, que no tiene porqué ser la misma información.

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: k8s.gcr.io/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      path: /data # directory location on host
      type: Directory # this field is optional
```

- **CloudVols**: Son volumenes cloud (por ejemplo amazon EBS). En este caso consumimos volumenes de archivos de cloud que son usables por todos los pods. Para configurarlos necesita **PVs** y **PVC**
  - **PV** (persistent volume) será un objeto que representa la definición del volumen que se conectará al disco cloud.
  - **PVC** (persistent volume claiming) será un objeto que representa la reclamación de espacio en un PV que usa un disco cloud para almacenar.
- Existen además otros tipos de volumenes:
  - `awsElasticBlockStore` - AWS Elastic Block Store (EBS)
  - `azureDisk` - Azure Disk
  - `azureFile` - Azure File
  - `cephfs` - CephFS volume
  - `cinder` - Cinder (OpenStack block storage) (**deprecated**)
  - `csi` - Container Storage Interface (CSI)
  - `fc` - Fibre Channel (FC) storage
  - `flexVolume` - FlexVolume
  - `flocker` - Flocker storage
  - `gcePersistentDisk` - GCE Persistent Disk
  - `glusterfs` - Glusterfs volume
  - `iscsi` - iSCSI (SCSI over IP) storage
  - `local` - local storage devices mounted on nodes.
  - `nfs` - Network File System (NFS) storage

- `photonPersistentDisk` - Photon controller persistent disk. (This volume type no longer works since the removal of the corresponding cloud provider.)
- `portworxVolume` - Portworx volume
- `quobyte` - Quobyte volume
- `rbd` - Rados Block Device (RBD) volume
- `scaleIO` - ScaleIO volume (**deprecated**)
- `storageos` - StorageOS volume
- `vsphereVolume` - vSphere VMDK volume

PV y PVs

https://kubernetes.io/docs/concepts/storage/persistent-volumes/

PV

https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistent-volumes

Podemos crear un PV (Persistent Volume) usando un manifiesto. Por defecto se crean como **HostPath**

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: foo-pv
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data" # Carpeta del host a la que se refiere el PV. Podría ser una
carpeta cloud
                      # de aws, azure...
```

PVCs

https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims

Vamos a crear un PVC que reclame el PV creado antes mediante un manifiesto:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 8Gi # Cuanto espacio queremos reclamar
  storageClassName: slow
  selector: # selectors para elegir el PV concreto (ver doc)
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

Recordemos que los **PV SON EXCLUSIVOS**, por lo que si un PVC usa un PV, un PV no va a poder ser utilizado por otros PVC.

Para asociar PV y PVCs podemos hacerlo de varias formas.

- Si no indicamos concretamente el nombre del PV, k8s buscará uno que cumpla nuestros requisitos y lo reclamará.
- Si usamos un manifiesto, podemos utilizar selectors para indicar las caracteristicas del PV que queremos reclamar.

Uso de PVCs en pods

Podemos asociar el PVC al pod en su manifiesto:

```yaml
apiVersion: v1
kind: PersistentVolume # PV
metadata:
  name: test-pv
  labels:
    mysql: ready
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mysql"
---
apiVersion: v1
kind: PersistentVolumeClaim # PVC
metadata:
  name: test-pvc
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  selector:
    matchLabels:
      mysql: ready
---
apiVersion: apps/v1
kind: Deployment # Definición del deployment del pod
metadata:
  name: mysql
  labels:
    app: mysql
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mysql # Selector de pods por app
  template: # datos del template
    metadata:
      labels:
        app: mysql
    spec:
      containers: # Datos de los contenedores que forman los pods
```

```
      - name: mysql
        image: mysql:5.7
        env:
          - name: MYSQL_ROOT_PASSWORD
            value: "12345678"
        volumeMounts: # Volumenes que montará y donde dentro del pod
          - mountPath: "/var/lib/mysql"
            name: vol-mysql
      volumes:
        - name: vol-mysql # Definición de los volumenes a usar
          persistentVolumeClaim: # Definición del PVC que va a usar
            claimName: test-pvc
```

StorageClass y provisionamiento dinámico

https://kubernetes.io/docs/concepts/storage/storage-classes/

Sirve para provisionar espacio a los PVCs en caso de no encontrar PVs compatibles. Esto es que **si el PVC no encuentra un PV compatible, k8s lo crea y lo asocia**. Esto se hará por medio del storageClass.

Lo interesante es que podemos definir StorageClasses de Amazon EVS de forma que obtenga espacio de forma dinámica.

Un manifiesto tipo de StorageClass:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs # este provisioner indica dónde o como se va a
persistir
parameters:
  type: gp2
reclaimPolicy: Retain # Definimos la reclaimPolicie para ver qué pasa si el PVC muere
allowVolumeExpansion: true # Permite que sea expansible
mountOptions:
  - debug
volumeBindingMode: Immediate # Indica cundo debe realizarse el binding y el
provisionamiento dinamico
```

Provisioner

Cada storage class tiene asociado un provisioner que es el que determina el plugin que es usado para provisionar los PVs:

olume Plugin	Internal Provisioner	Config Example
AWSElasticBlockStore	✓	AWS EBS
AzureFile	✓	Azure File
AzureDisk	✓	Azure Disk
CephFS	-	-
Cinder	✓	OpenStack Cinder
FC	-	-
FlexVolume	-	-
Flocker	✓	-
GCEPersistentDisk	✓	GCE PD
Glusterfs	✓	Glusterfs
iSCSI	-	-
Quobyte	✓	Quobyte
NFS	-	-
RBD	✓	Ceph RBD
VsphereVolume	✓	vSphere
PortworxVolume	✓	Portworx Volume
ScaleIO	✓	ScaleIO
StorageOS	✓	StorageOS
Local	-	Local

Reclaim Policies

https://kubernetes.io/docs/concepts/storage/storage-classes/#reclaim-policy

Será la política de reclamación de espacio de un PVC. Esto sirve para controlár qué pasa cuando muere un PVC, que va a dejar de necesitar por tanto un PV que se conecte al disco cloud. k8s ofrece varias alternativas:

- **Retain**: si se elimina el PVC, el PV y la información en el cloud no se elimina PERO NO PUEDE SER CONSUMIDA POR OTRO PVC. Se guarda a efectos de poder disponer de ella de otra forma
- **Recycle**: Al eliminar el PVC, el PV no se elimina PERO SÍ LA INFORMACIÓN EN EL DISCO CLOUD, por lo que otro PVC podrá utilizar el PV.
- **Delete**: Cuando el PVC se borra, se borra el PV y el volumen cloud.

Linkeo entre PV y PVS a storageClassName

Veamos como se linkea un PV y PVS a un storageClassName:

```
apiVersion: v1
kind: PersistentVolume # PV
metadata:
  name: test-pv
  labels:
```

```
    mysql: ready
spec:
  storageClassName: manual # StorageClass manual
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mysql"
---
apiVersion: v1
kind: PersistentVolumeClaim # PVC
metadata:
  name: test-pvc
spec:
  storageClassName: manual # StorageClass manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  selector:
    matchLabels:
      mysql: ready
```

Role Base Access Control: Users & Groups

https://kubernetes.io/docs/reference/access-authn-authz/rbac/

Permite definir permisos y controlar el acceso de los usuarios y grupos, deifnir roles...

Gestión de usuarios: Certificados

https://kubernetes.io/docs/reference/access-authn-authz/certificate-signing-requests/

https://kubernetes.io/docs/tasks/administer-cluster/certificates/

Kubernetes no proporciona un sistema de gestión de usuarios. Por el contrario, proporciona sistemas para autenticar a los usuarios en k8s. De forma general, suele utilizarse un sistema de certificados:

- En k8s existe un certificado que sirve para firmar otros certificados. Se denomina C.A. Si firmamos un certificado de un usuario (CSR) con el certificado CA, k8s permitirá usar el certificado para identificar el certificado. Por lo que los pasos serían:
    - Generar el CA
    - Obtener el certificado del usuario (CSR). k8s tomará el campo CN del certificado como el nombre de usuario y el campo O (Organization) como el grupo.
    - Firmar el CSR con CA para que k8s lo identifique.

Roles & ClusterRoles

https://kubernetes.io/docs/reference/access-authn-authz/rbac/#role-and-clusterrole

Rol y ClusterRol son plantillas en los que definimos reglas sobre recursos y permisos. La diferencia es que un **Rol se define para un namespace y un ClusterRol se define para un cluster completo**.

- *Ejemplo de rol*:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default # Indicamos el namespace
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"] # Recursos sobre los que tiene permisos
  verbs: ["get", "watch", "list"] # Qué puede hacer con los recursos
```

- *Ejemplo de ClusterRol*:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  # "namespace" omitted since ClusterRoles are not namespaced
  name: secret-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  # at the HTTP level, the name of the resource for accessing Secret
  # objects is "secrets"
  resources: ["secrets"] # Recursos sobre los que se tiene permisos
  verbs: ["get", "watch", "list"] # QUé permisos se tienen
```

RoleBinding& ClusterRoleBinding

https://kubernetes.io/docs/reference/access-authn-authz/rbac/#rolebinding-and-clusterrolebinding

Sirve para unir Role y ClusterRole a los usuarios. Para ello indicaremos el rol/ClusterRol y el *subject*, que será a quién le aplicará ese rol/clusterRol, pudiendo ser usuarios, grupos, etc...

Para asegurarnos que RBAC está activado deberemos comprobarlo mediante `kubectl cluster-info | grep authortization-mode` indicando si está incluido. En caso de no estarlo deberiamos activarlo.

- *Ejemplo de RoleBinding asociado a nivel de usuario*

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pods in the "default" namespace.
# You need to already have a Role named "pod-reader" in that namespace.
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects: # Indica a quién afecta el rol
# You can specify more than one "subject"
- kind: User # Especificamos el binding a nivel usuario (campo CN del certificado)
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef: # Indica el rol
  # "roleRef" specifies the binding to a Role / ClusterRole
  kind: Role # this must be Role or ClusterRole
  name: pod-reader # this must match the name of the Role or ClusterRole you wish to
bind to
  apiGroup: rbac.authorization.k8s.io
```

- *Ejemplo de ClusterRoleBinding asociado a nivel de grupo*

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This cluster role binding allows anyone in the "manager" group to read secrets in any
namespace.
kind: ClusterRoleBinding
metadata:
  name: read-secrets-global
subjects:
- kind: Group # Especificamos el binding a nivel grupo (campo O del certificado)
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole # this must be Role or ClusterRole
  name: secret-reader # this must match the name of the Role or ClusterRole you wish to
bind to
  apiGroup: rbac.authorization.k8s.io
```

ServiceAccount

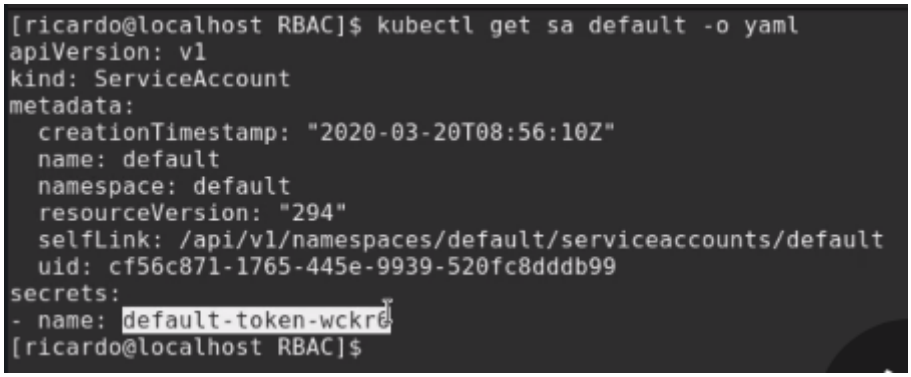
https://kubernetes.io/docs/tasks/configure-pod-container/configure-service-account/

Se trata de una forma de gestionar los permisos de los Pods sobre los recursos del cluster. Son objetos que consumen Roles y ClusterRoles a través de bindings y que se asignan a los pods. Funciona de la siguiente forma:

- Definiremos roles y cluster roles con los permisos que queremos otorgar
- Definiremos ServiceAccount para poder asociarlos a los pods
- Realizaremos un role-binding entre el Rol y el ServiceAccount para otorgrar permisos al ServiceAccount
- Realizaremos un binding entre el Pod y el ServiceAccount para otorgar permisos al pod
- Cuando el POD use la API, va a leer un token que incluye el ServiceAccount asociado para verificar su autenticidad, y verificará los permisos otorgados a través del role-binding que existe entre el serviceAccount y el role.

Descripción de un ServiceAccount

Todos los namespaces tienen un serviceAccount por defecto. Si vemos qué contiene un service account vemos que es simplemente un objeto asociado a un secret, que es un token:



Para ver el token asociado al serviceAccount, podemos observar el token con `kubectl get secrets <nombre>`. Si lo decodificamos vemos que dentro estará:

- la clave pública del certificado,
- el namespace asociado
- otro token en JWT con la definición del serviceAccount

k8s se encarga de realizar la creación de estos token cuando creamos los service accounts.

Creación de un ServiceAccount

De forma sencilla, un ServiceAccount se puede crear con un manifiesto simple del tipo:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-robot
```

Al realizar un `apply`, k8s se encarga de incluir el token automáticamente.

Asociacion entre Pods y ServiceAccounts

Si no definimos un serviceAccount para un pod, por defecto se asocia al serviceacount por defecto. Si queremos configurar un serviceAccoun concreto en nuestros pods deberemos:

- Definir el ServiceAccount
- Definir un Deployment que genere y gestione los pods
- Definir el Rol al que queremos asociar el serviceAccount
- Definir el RoleBinding que asocie el serviceAccount al Role

```
---
# definimos el SERVICE ACCOUNT que vamos a usar
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-sa
---
# definimos el DEPLOYMENT que va a gestionar los pods
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test
  labels:
    app: front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: front
  template:
    metadata:
      labels:
        app: front
    spec:
```

```
      serviceAccountName: my-sa # Indicamos el serviceAccount que queremos que usen
      containers:
      - name: nginx
        image: nginx:alpine
---
# Indicamos el ROL que vamos a asociar al serviceAccount
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: sa-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["apps"] # "" indicates the core API group
  resources: ["deployments", "replicasets"]
  verbs: ["get", "watch", "list"]
---
# Indicamos la ROLEBINDING que asocie el Rol y el ServiceAccount
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sa-pods
  namespace: default
subjects:
- kind: ServiceAccount # Tipo de objeto al que asociamos el rol
  name: my-sa # "name" is case sensitive
  apiGroup: ""
roleRef: # Rol que queremos asociar
  kind: Role #this must be Role or ClusterRole
  name: sa-reader # this must match the name of the Role or ClusterRole you wish to bind
to
  apiGroup: rbac.authorization.k8s.io
```

Exponer apps fuera del cluster

Ingress

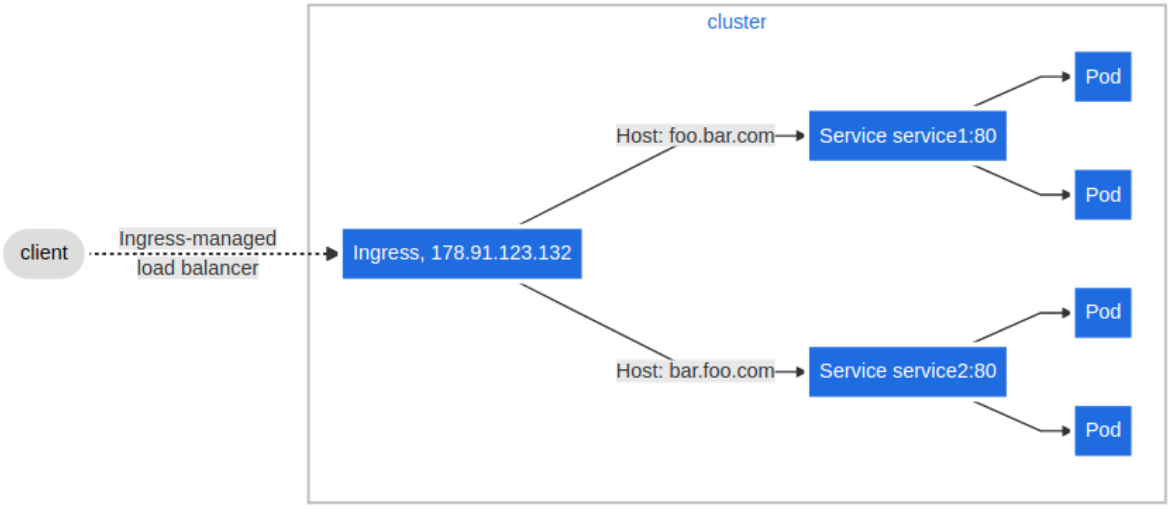
https://kubernetes.io/docs/concepts/services-networking/ingress/

Se trata de un objeto que permite a un cliente de fuera del cluster acceder a los servicios tipo Nodeport o LoadBalancer dentro del cluster. Ingress permite que podamos disponer un solo acceso a todos los servicios, mediante la redirección de las rutas web a los servicios.

Un Ingress puede manejar distintos servicios internos del cluster, mediante el mapeo de distintos paths de entrada hacia distintos servicios del cluster. Se pueden definir mediante un manifiesto como el siguiente:

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress # Tipo ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: foo.bar.com # Host de acceso 1
    http:
      paths: # Paths asociados al host 1
      - pathType: Prefix
        path: "/" # Path para acceder al service 1: http://foo.bar.com/
        backend:
          service:
            name: service1 # servicio al que se asocia esa ruta
            port:
              number: 80
  - host: bar.foo.com # Host de acceso 2. SI ELIMINAMOS EL HOST, APLICARIA A TODOS
    http:
      paths: # Paths asociados al host 2
      - pathType: Prefix
        path: "/" # Path para acceder al service 2: http://bar.foo.com/
        backend:
          service:
            name: service2 # servicio al que se asocia esa ruta
            port:
              number: 80
```

En este ejemplo, se mapean dos servicios accesibles mediante distinto host:



Recordemos que estos servicios deberán ser de tipo **NodePort** para que exponga un puerto.

Mediante la definición de las reglas podemos jugar y configurar de forma flexible la entrada al cluster:

- **Opciones para definir paths**: https://kubernetes.io/docs/concepts/services-networking/ingress/#path-types
- **Uso de Wildcards**: https://kubernetes.io/docs/concepts/services-networking/ingress/#hostname-wildcards

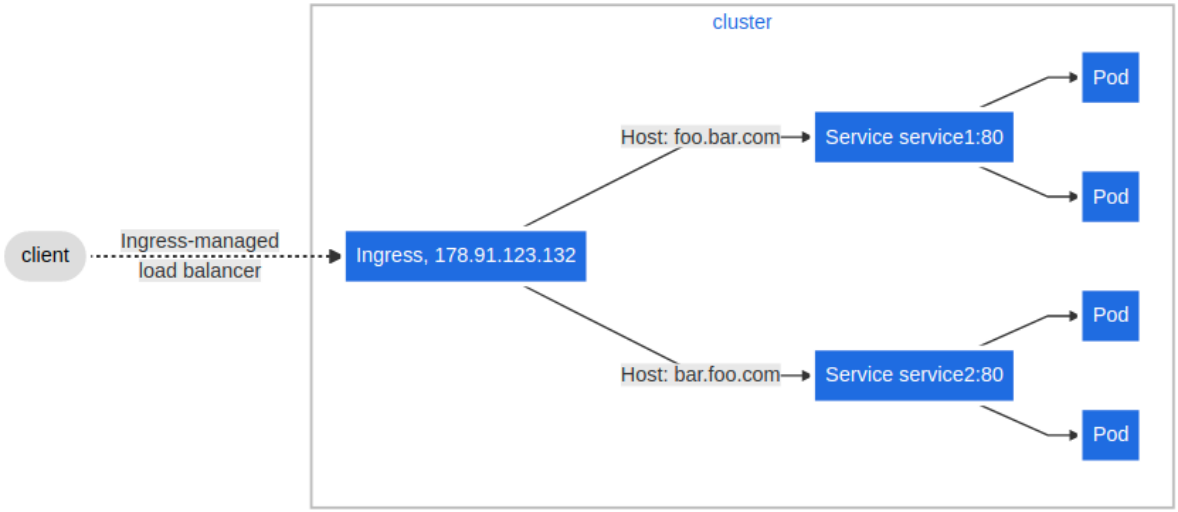
Tipos de Ingress

Podemos definir distintos tipos de Ingress según los especificados por k8s:

- **Ingress definido por un unico servicio**: Existen casos en los que queremos exponer un solo servicio como punto de entrada de todo el cluster. Para ello podemos especificarlo sin reglas:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
spec:
  defaultBackend:
    service:
      name: test
      port:
        number: 80
```

- **Fanout simple**: Una configuración de fanout enruta el tráfico desde una única dirección IP a más de un Servicio, basándose en el URI HTTP que se solicita. Un Ingress le permite mantener el número de balanceadores de carga al mínimo. Por ejemplo, una configuración como èsta:




```
    Puede definirse con un manifiesto como este:
```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: name-virtual-host-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - pathType: Prefix
        path: "/"
        backend:
          service:
            name: service1
            port:
              number: 80
  - host: bar.foo.com
    http:
```

```
    paths:
    - pathType: Prefix
      path: "/"
      backend:
        service:
          name: service2
          port:
            number: 80
```

- **TLS**: Puede asegurar un Ingress especificando un Secreto que contenga una clave privada TLS y un certificado. El recurso Ingress sólo admite un único puerto TLS, el 443, y asume la terminación TLS en el punto de entrada (el tráfico hacia el Servicio y sus Pods está en texto plano). Si la sección de configuración TLS en un Ingress especifica diferentes hosts, se multiplexan en el mismo puerto según el nombre de host especificado a través de la extensión SNI TLS (siempre que el controlador Ingress soporte SNI). El secreto TLS debe contener claves denominadas tls.crt y tls.key que contienen el certificado y la clave privada a utilizar para TLS. Por ejemplo:

```
# Definimos el secreto
apiVersion: v1
kind: Secret
metadata:
  name: testsecret-tls
  namespace: default
data:
  tls.crt: base64 encoded cert
  tls.key: base64 encoded key
type: kubernetes.io/tls
---
# Definimos el Ingress especnginx, o GCE).ificando el secret
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: tls-example-ingress
spec:
  tls:
  - hosts:
      - https-example.foo.com
    secretName: testsecret-tls
  rules:
  - host: https-example.foo.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 80
```

- **Load balancing**: Un controlador de Ingress es arrancado con algunas configuraciones de política de balanceo de carga que aplica a todos los Ingress, como el algoritmo de balanceo de carga, el esquema de pesos del backend, y otros. Los conceptos de equilibrio de carga más avanzados (por ejemplo, sesiones persistentes, pesos dinámicos) aún no están expuestos a través del Ingress. En su lugar, puede obtener estas características a través del equilibrador de carga utilizado para un Servicio.

> También vale la pena señalar que, aunque los controles de salud no están expuestos directamente a través del Ingress, existen conceptos paralelos en Kubernetes, como las sondas de preparación, que le permiten lograr el mismo resultado final. Por favor, revise la documentación específica del controlador para ver cómo manejan las comprobaciones de salud (por ejemplo: nginx, or GCE).

IngressController

https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

Ingress permite definir las reglas, pero para definir un punto de acceso al cluster necesitamos un **IngressController**.

Se trata del objeto que permite controlar el punto de entrada al cluster y que escucha las reglas del Ingress para redefinir el trafico. Básicamente son Deployments configurados para escuchar los puertos y que serán los puntos de acceso de nuestro cluster. En Kubernetes hay tres tipos soportados por k8s de ingresControllers: AWS, GCE, y nginx, aunque existen controladores adicionales de terceros. Para ello definimos objetos **Servicios Nodeport** para poder exponer los IngressController a través de una Ip pública del cluster.

Puedes definir más de un IngresController, debiendo estar indicado qué **IngressClass** está utilizando (ver en siguiente punto).

Para ver como se crean, deberemos ingresar en la documentación de cada uno de ellos.

IngressClass (compartiendo un Ingress)

Para usar un mismo Ingress por distintos controladores, deberemos definir la **IngressClass**. Cada Ingress debe especificar una clase, una referencia al recurso IngressClass quue contiene configuracion adicional, incluyendo el nombre del controlador que debe implementar la clase.

```yaml
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: external-lb
spec:
  controller: example.com/ingress-controller
  parameters:
    apiGroup: k8s.example.com
    kind: IngressParameters # Parametros concretos para este IngressClass
    name: external-lb
```

Mediante un objeto **IngressParameters** podemos definir propiedades concretas para esta clase (ejemplo).

IngressClass por defecto

Podemos marcar un IngressClass como el IngressClass por defecto seteando la anotacion `ingressclass.kubernetes.io/is-default-class` a `true`