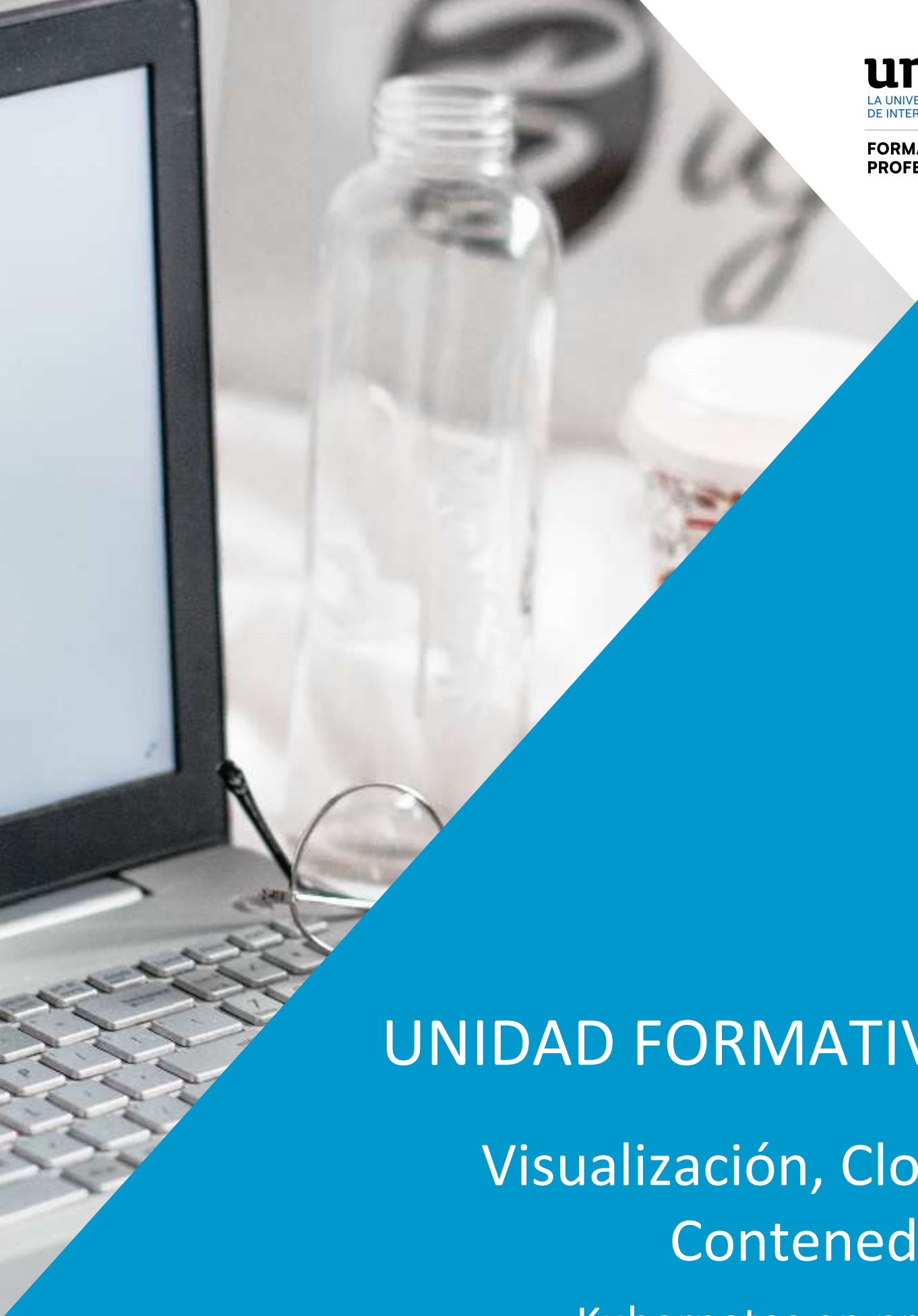


unir
LA UNIVERSIDAD DE INTERNET

FORMACIÓN PROFESIONAL

UNIDAD FORMATIVA 3

Visualización, Cloud y Contenedores

Kubernetes en práctica

Índice

Kubernetes en práctica

En esta segunda parte del curso de Kubernetes nos vamos a centrar en los mecanismos que nos van a permitir productivizar nuestras aplicaciones.

Ya hemos visto cómo desplegar contenedores encapsulados en Pods dentro del clúster. Sin embargo, un aspecto clave para cualquier aplicación es el **almacenamiento de datos**. A continuación, veremos las principales opciones que nos ofrece Kubernetes para almacenar la información de las aplicaciones y qué grado de persistencia ofrece cada una.

También aprenderemos a desacoplar la configuración de las aplicaciones de la definición de los Pods. Esto nos permitirá reutilizar nuestros Pods en diferentes entornos de despliegue, pero con configuraciones distintas, haciendo nuestros desarrollos más reutilizables.

Hasta ahora hemos visto cómo desplegar nuestros Pods en el clúster de manera individual. En un entorno real, querremos tener varias réplicas de nuestros Pods ejecutándose a la vez de una manera consistente: estudiaremos cómo los objetos ***ReplicaSets*** de Kubernetes nos ayudarán con este propósito.

Si lo que queremos es ejecutar tareas de mantenimiento o supervisión, los ***DaemonSets*** se aseguran de que en todos los nodos del clúster, o en aquellos que hemos seleccionado, se ejecute un Pod determinado.

Otro de los aspectos más relevantes a la hora de desplegar aplicaciones en Kubernetes es cómo van a ser expuestos nuestros Pods. Aquí es donde entran en juego los objetos ***Services***, que nos permitirán exponer un conjunto de Pods a través de una única IP y puerto, ya sea de manera interna o externa al clúster.

Objetivos

- Conoceremos cómo funciona la persistencia en Kubernetes.
- Aprenderemos cómo se maneja la configuración y cómo se guardan los secretos en el clúster.
- Conoceremos las diferentes alternativas de despliegue en un clúster de Kubernetes.

Volúmenes y Configuración

Importante

Por no extender las explicaciones de cada nuevo concepto o parámetro que se introduce a los manifiestos, recordemos que *kubectl* viene con un completísimo manual que nos permite obtener documentación de cualquier elemento del *yaml* de un manifiesto:

```
$ kubectl explain pods.spec.containers.volumeMounts
KIND:     Pod
VERSION:  v1

RESOURCE: volumeMounts <[]Object>

DESCRIPTION:
     Pod volumes to mount into the container's filesystem. Cannot be updated.

     VolumeMount describes a mounting of a Volume within a container.

FIELDS:
   mountPath     <string> -required-
     Path within the container at which the volume should be mounted. Must not
     contain ':'.

   mountPropagation      <string>
     mountPropagation determines how mounts are propagated from the host to
     container and the other way around. When not set, MountPropagationNone is
     used. This field is beta in 1.10.

...
```

Almacenamiento de datos en Kubernetes

En Kubernetes, los volúmenes son directorios accesibles por todos los contenedores que forman parte de un Pod. Las modificaciones realizadas en el sistema de ficheros locales de los contenedores **se perderán cuando se reinicien**; sin embargo, la información en los volúmenes sí se mantendrá tras un reinicio del contenedor.

Además de los volúmenes existen otros tipos de recursos para el almacenamiento de la configuración. En general, podemos clasificar el almacenamiento de Kubernetes en base a la **infraestructura** que lo respalda y la **persistencia** que ofrece:

- Persistencia local al Pod o al nodo (emptyDir, hostPath).
- Compartición de ficheros (nfs).
- Asociados a proveedor cloud (awsElasticBlockStore, azureDisk, gcePersistentDisk).
- Sistemas de ficheros distribuidos (glusterfs, cephfs).
- Almacenamiento orientado a un propósito específico (gitRepo, configMap, secret).

Volúmenes emptyDir

Los volúmenes de tipo **emptyDir** nos permitirán compartir ficheros entre los contenedores de un Pod. Como su nombre indica, estos se crean vacíos. Todos los contenedores que forman parte del Pod podrán tener acceso de lectura y escritura a los ficheros del volumen, aunque cada contenedor podrá montar el volumen en una ruta diferente. Los volúmenes **emptyDir** están ligados a la vida útil del Pod, es decir, los datos almacenados serán borrados cuando el Pod sea eliminado del nodo.

Para configurar un volumen de este tipo en la definición YAML de un Pod, lo añadiremos en la propiedad **spec.volumes**, donde le daremos un nombre e indicaremos el tipo. Además, en cada uno de los contenedores que vaya a hacer uso del volumen, deberemos indicar en spec.containers.volumeMounts su nombre, la ruta local donde se montará y, opcionalmente, si queremos solamente acceso de lectura desde el contenedor.

En el siguiente ejemplo se define un volumen de tipo **emptyDir** en un Pod, que será utilizado por dos contenedores, cada uno en una ruta distinta y uno de ellos con solo acceso de lectura:

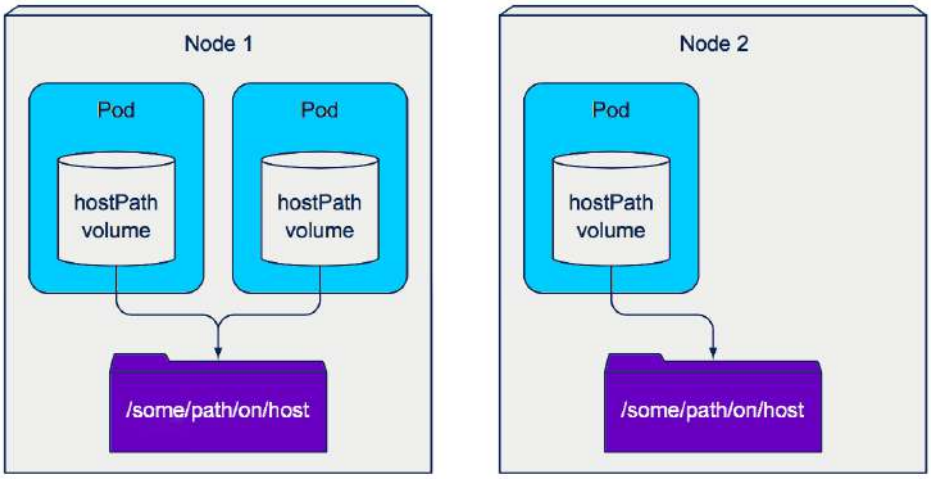
```
apiVersion: v1
kind: Pod
metadata:
  name: web-applicaton
spec:
  containers:
  - name: generador-html
    image: ubuntu:alpine
    volumeMounts:
      - name: html-content
        mountPath: /var/web-site
  - name: servidor-web
    image: nginx:alpine
    volumeMounts:
      - name: html-content
        mountPath: /usr/share/nginx/html
        readOnly: true
  volumes:
  - name: html-content
    emptyDir: {}
```

Los volúmenes emptyDir se crearán en el disco del nodo que esté alojando al Pod; esto quiere decir que el rendimiento dependerá del tipo de disco que se utilice. Sin embargo, podemos indicarle a Kubernetes que queremos utilizar un sistema de ficheros en memoria (***tmpfs***), el cual será muchísimo más rápido, aunque deberemos tener en cuenta los límites de memoria de los contenedores. La definición del volumen quedaría de la siguiente manera:

```
...
  volumes:
  - name: html-content
    emptyDir:
      medium: Memory
```

Volúmenes hostPath

Por lo general, los Pods no deberían acceder al sistema de ficheros del nodo. Sin embargo, en ocasiones tendremos algunos Pods que realizan tareas a nivel del sistema que sí necesitarán acceder al sistema de ficheros de los nodos. Este tipo de Pods habitualmente actuarán como demonios (*daemons*) y, seguramente, serán desplegados mediante un *DaemonSet*.



Un volumen *hostPath* monta ficheros del nodo contenedor del Pod.

Los volúmenes de tipo **hostPath** referencian un directorio o archivo específico del sistema de ficheros del nodo, permitiendo a los Pods que lo monten, acceder a su contenido en la ruta local especificada. Es importante tener en cuenta que dicha ruta será la misma en todos los nodos y deberemos saber previamente si existe o no y si tenemos los permisos necesarios.

Algunos ejemplos de casos de uso son:

- El contenedor necesita acceder a la información interna de Docker (/var/lib/Docker).
- Se quiere ejecutar cAdvisor en el contenedor para el análisis de recursos (/sys).

```
---
apiVersion: v1
kind: Pod
metadata:
  name: app-daemon
spec:
  containers:
  - image: ubuntu:alpine
    name: daemon-container
    volumeMounts:
    - mountPath: /data/node
      name: data-node
  volumes:
  - name: data-node
    hostPath:
      path: /data
      type: DirectoryOrCreate
```

Reto

Encuentra los posibles valores del campo ***type***.

Volúmenes persistentes

La mayoría de nuestras aplicaciones necesitarán que su información persista más allá de la vida útil de un Pod e independiente del nodo en el que se aloje. Los volúmenes vistos hasta ahora no nos permitían esta persistencia, ya que estaban ligados a la vida del Pod o del Nodo.

Por ello, para que nuestros datos sean persistentes y accesibles desde cualquiera de los nodos, necesitaremos disponer de algún tipo de almacenamiento conectado a la red. Estos podrían ser volúmenes alojados en un proveedor de la nube, recursos compartidos con NFS o sistemas distribuidos de ficheros. A continuación, veremos cómo utilizar algunas de estas opciones.

Volúmenes awsElasticBlockStore

Cuando utilicemos un clúster de Kubernetes que se ejecute en una instancia de EC2 de AWS, podremos utilizar un volumen de tipo ***awsElasticBlockStore*** para montar un volumen EBS de Amazon en los Pods. Además, podremos inicializarlos previamente al despliegue de los Pods y, una vez que estos últimos sean eliminados, el volumen simplemente se desmontará, conservando todo su contenido.

Para poder utilizar este tipo de volúmenes, deberemos crearlo previamente en AWS, ya sea mediante la consola web de AWS o su cliente de línea de comandos:

```
$ aws ec2 create-volume --availability-zone=eu-west-1a \
   --size=10 --volume-type=gp2
```

Veamos un ejemplo de configuración YAML, donde indicaremos el identificador en AWS y el tipo del sistema de ficheros:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: sample-app
spec:
  containers:
  - image: my-app:latest
    name: app-container
    volumeMounts:
    - mountPath: /data
      name: volumen-ecs
  volumes:
  - name: volumen-ecs
    awsElasticBlockStore:
      volumeID: "<volume id>"
      fsType: ext4
```

Volúmenes gcePersistentDisk

De igual forma, si nuestro clúster se está ejecutando en Google Kubernetes Engine (GKE), entonces podremos crear discos persistentes de GCE y montarlos en nuestros Pods. Para ello, crearemos previamente un disco en la misma zona que nuestro clúster:

```
$ gcloud compute disks create --size=1GiB --zone=europe-west1-b mongodb
```

La configuración en nuestro fichero YAML es similar a la anterior; simplemente crearemos un volumen de tipo gcePersistentDisk indicando el nombre en GCE y el tipo de sistema de archivos:

```
volumes:
- name: mongodb-data
  gcePersistentDisk:
    pdName: mongodb
    fsType: ext4
```

Volúmenes NFS

Cuando nuestro clúster de Kubernetes está ejecutándose en servidores propios, en lugar de utilizar alguna de las soluciones de Kubernetes en la nube, tenemos varias opciones para montar un almacenamiento persistente externo. Una de ellas es montar recursos compartidos NFS en los Pods.

Deberemos disponer de nuestro propio servidor NFS con el recurso a compartir previamente creado y configurado. Este, además, podrá ser montado por varios Pods simultáneamente. Veamos un ejemplo de cómo lo definiríamos en YAML:

```
volumes:
- name: volumen-nfs
  nfs:
    server: 192.168.0.200
    path: /data/shared-data
```

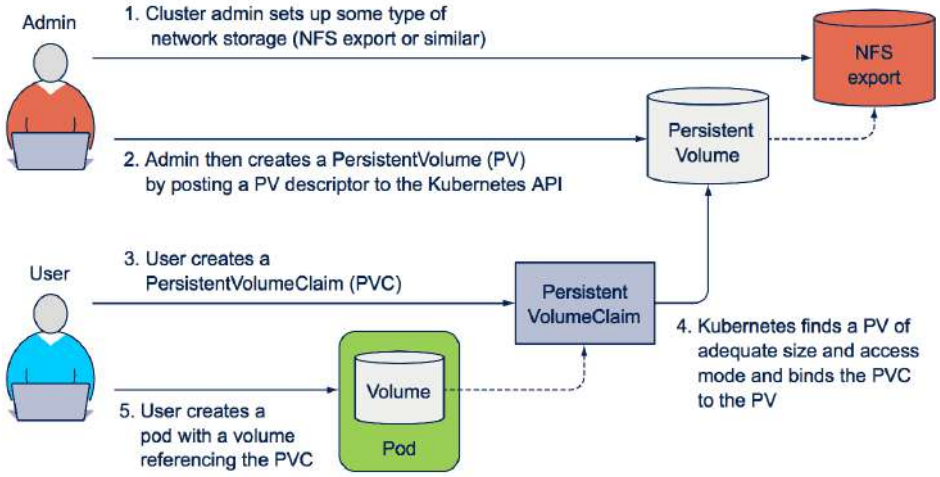
Kubernetes *PersistentVolume*

Hasta ahora, los tipos de volúmenes persistentes que hemos visto necesitaban de cierto conocimiento de la infraestructura por parte de los desarrolladores al definir los Pods. Por ejemplo, para utilizar un volumen de tipo NFS había que conocer cuál era el servidor.

Kubernetes nos permite desacoplar el método de almacenamiento, ocultando la infraestructura subyacente a los desarrolladores. Estos últimos solo tendrán que solicitar la cantidad de almacenamiento requerido para sus aplicaciones y será Kubernetes el encargado de ponerlo a disposición de los Pods.

Por lo tanto, serán los administradores del clúster los encargados de configurar el almacenamiento disponible, indicando el tamaño y los modos de acceso permitidos, y registrarlo en Kubernetes utilizando recursos de tipo ***PersistentVolume***.

Posteriormente, cuando un usuario necesite almacenamiento lo solicitará por medio de un manifiesto ***PersistenVolumeClaim***, indicando cuánta capacidad necesita y el modo de acceso requerido. Finalmente, será Kubernetes el encargado de asociarle un PersistentVolume adecuado.



Pasos de la creación y uso de un PersistentVolume.

Creación de un PersistentVolume

Antes de crear el recurso *PersistentVolume*, deberemos tener disponible el almacenamiento físico. Una vez dado este paso, crearemos el manifiesto en formato YAML, donde deberemos indicar la capacidad, los modos de acceso que queremos permitir y la referencia al disco.

Los modos de acceso permitidos son:

- **ReadWriteOne (RWO).** Únicamente se permite que un nodo monte el volumen para lectura y escritura.
- **ReadOnlyMany (ROX).** Está permitido que múltiples nodos monten el volumen para solo lectura.
- **ReadWriteMany (RWX).** Varios nodos pueden montar el volumen tanto para lectura como para escritura.

Veamos un ejemplo de creación de un *PersistentVolume* para un disco de tipo *Elastic Block Storage* (EBS) previamente creado en AWS:

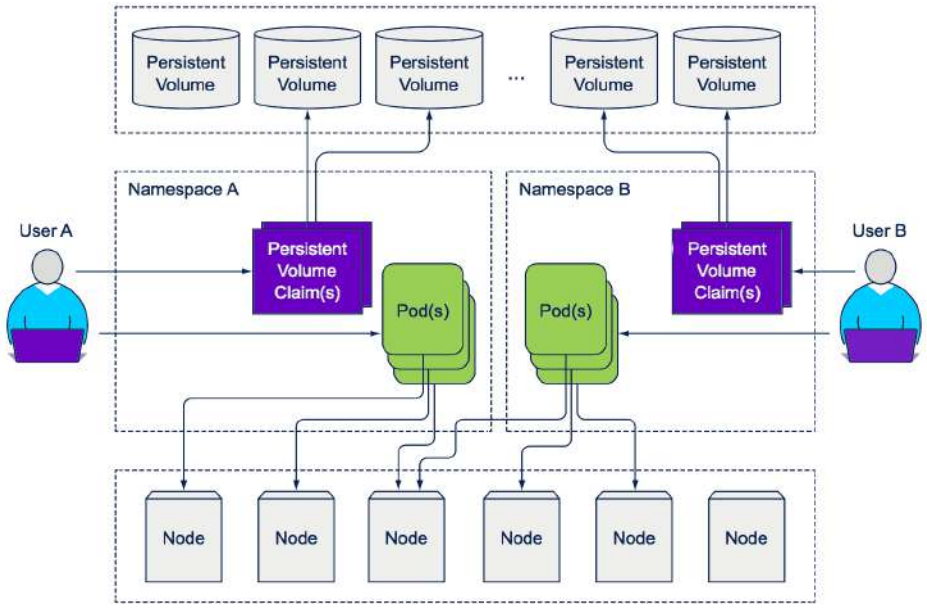
```
---
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongodb-pv
  labels:
    type: amazonEBS
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadOnlyMany
  awsElasticBlockStore:
    volumeID: "vol-12312313"
    fsType: ext4
```

```
$ kubectl create -f mongodb-pv.yaml
persistentvolume "mongodb-pv" created
$ kubectl get persistentvolume
NAME        CAPACITY   RECLAIMPOLICY   ACCESSMODES   STATUS      CLAIM
mongodb-pv   1Gi        Retain          RWO,ROX       Available
```

¡Ojo! Si estamos realizando pruebas con Minikube o Docker Desktop, podemos definirnos un PersistentVolume utilizando un volumen de tipo hostPath, en lugar de referenciar uno externo:

```
...
  hostPath:
    path: /tmp/mongodb
```

Los *PersistentVolume* no pertenecen a ningún *Namespace*, sino que son recursos a nivel de clúster, al igual que ocurre con los nodos.



Los PV son globales al clúster, los PVC pertenecen a un namespace.

Reclamación de un *PersistentVolume* mediante *PersistentVolumeClaims*

Antes de poder utilizar en los Pods el almacenamiento que acabamos de configurar, deberemos **reclamarlo** mediante un ***PersistentVolumentClaim***. Este proceso se realiza de forma **previa e independiente** a la creación de los Pods. A diferencia de los *PersistentVolume*, que eran globales al clúster, estos objetos de reclamo o *claims* existen a nivel de Namespace. Esto quiere decir que solo los Pods creados en dicho Namespace podrán utilizarlo.

Para hacer esta solicitud de almacenamiento, crearemos un fichero en formato YAML con su definición, en la que indicaremos la cantidad de almacenamiento requerido, así como los modos de acceso:

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-pvc
spec:
  resources:
    requests:
      storage: 1Gi
  accessModes:
  - ReadWriteOnce
  selector:
    matchLabels:
      type: "amazonEBS"
```

```
$ kubectl create -f mongodb-pvc.yaml
persistentvolumeclaim "mongodb-pvc" created
```

Al listar los ***pvc*** veremos que aparece como enlazado (*bound*) con el PersistentVolume previamente creado. Una vez enlazado, este último ya no estará disponible:

```
$ kubectl get pvc
NAME        STATUS    VOLUME      CAPACITY   ACCESSMODES   AGE
mongodb-pvc   Bound     mongodb-pv   1Gi      RWO,ROX      3s

$ kubectl get pv
NAME        CAPACITY   ACCESSMODES   STATUS    CLAIM             AGE
mongodb-pv   1Gi      RWO,ROX      Bound     default/mongodb-pvc   1m
```

Utilizando PersistentVolumeClaim en la definición del Pod

Ya por último, solamente nos quedaría referenciar nuestro objeto PersistentVolumeClaim en la definición del Pod para poder utilizarlo. Esto lo haremos en la propiedad ***spec.volumes***, al igual que vimos con el resto de volúmenes persistentes.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: mongodb
spec:
  containers:
  - image: mongo
    name: mongodb
    volumeMounts:
    - name: mongodb-data
      mountPath: /data/db
    ports:
    - containerPort: 27017
      protocol: TCP
  volumes:
  - name: mongodb-data
    persistentVolumeClaim:
      claimName: mongodb-pvc
```

Aprovisionamiento dinámico con StorageClass

Acabamos de ver cómo utilizar PersistentVolumesClaims para desacoplar la infraestructura de almacenamiento de la definición de los Pods. Sin embargo, hace falta que los administradores:

- Creen previamente el almacenamiento y lo disponibilicen.
- Que lo registren en el clúster mediante recursos *PersistenVolumes*.

Kubernetes soporta el aprovisionamiento **dinámico** del almacenamiento externo creando, además, su objeto PersistentVolume, eliminando la necesidad de crear y registrar previamente el almacenamiento.

Para ello, definiremos recursos ***StorageClass*** y será Kubernetes el encargado de crear el PersistentVolume cada vez que se solicite almacenamiento mediante un PersistentVolumeClaim. Kubernetes soporta el aprovisionamiento dinámico para la mayoría de los proveedores de nube.

Al definir un *StorageClass* especificaremos el proveedor a utilizar (provisioner), así como los parámetros (parameters) que se utilizarán cuando se realice el aprovisionamiento dinámico. Cada proveedor tendrá su propio conjunto de parámetros (para más información, consulta la documentación en Storage Classes). El siguiente ejemplo crea un StorageClass para discos EBS en AWS:

```
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: gp2-ebs-sc
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  fsType: ext4
```

```
$ kubectl create -f gp2-ebs-sc.yaml
storageclass "gp2-ebs-sc" created

$ kubectl get storageclass
NAME          PROVISIONER          AGE
gp2-ebs-sc     kubernetes.io/aws-ebs   8m
```

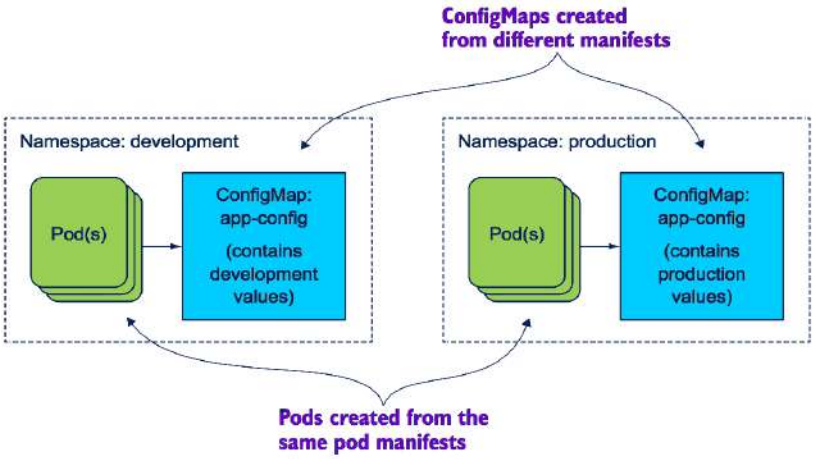
Para especificar que queremos un aprovisionamiento dinámico con el StorageClass que acabamos de crear, utilizaremos la propiedad ***spec.storageClassName*** de la definición del PersistentVolumenClaim. El uso de este en la definición de los Pods será igual a lo visto anteriormente.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: aws-pvc-sc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
  storageClassName: gp2-ebs-sc
```

ConfigMaps

Kubernetes también nos permite desacoplar de los Pods sus opciones de configuración, de manera que sean reutilizables en diferentes entornos. Separaremos la configuración en objetos de tipo ***ConfigMap***, que estarán compuestos por pares clave/valor. A la hora de desplegar nuestros Pods, estos se combinarán con los valores de configuración del *ConfigMap* antes de ejecutarse.

Los ConfigMap serán referenciados mediante su nombre en la definición de los Pods. Ello nos permite utilizar los mismos Pods en diferentes entornos creando diferentes ConfigMaps en cada Namespace:



Despliegue de un Pod en varios entornos con diferentes ConfigMaps.

Creación de un ConfigMap

Podemos crear un *ConfigMap* de forma imperativa utilizando el comando ***kubectl create configmap***. A este último deberemos pasarle por parámetro tanto el nombre que queremos asociar al ConfigMap como la fuente de datos de donde leer los valores de configuración.

Para la fuente de datos, tenemos tres opciones:

- Podemos crear el ConfigMap a partir de los ficheros de un directorio local. Todos aquellos ficheros del directorio cuyo nombre sea una clave válida serán añadidos al ConfigMap, y los subdirectorios serán ignorados.

```
$ kubectl create configmap ejemplo-dir-config --from-file=app-settings/
$ kubectl get configmaps ejemplo-dir-config -o yaml
...
data:
  database.properties: |
    connection=mongodb://username:pa$$w0rd@mongodb.sample.com:27017/admin
  smtp.properties: |
    smtp.server=sample-server.com
    smtp.port=587
    smtp.user=admin
    smtp.password=pa$$w0rd
```

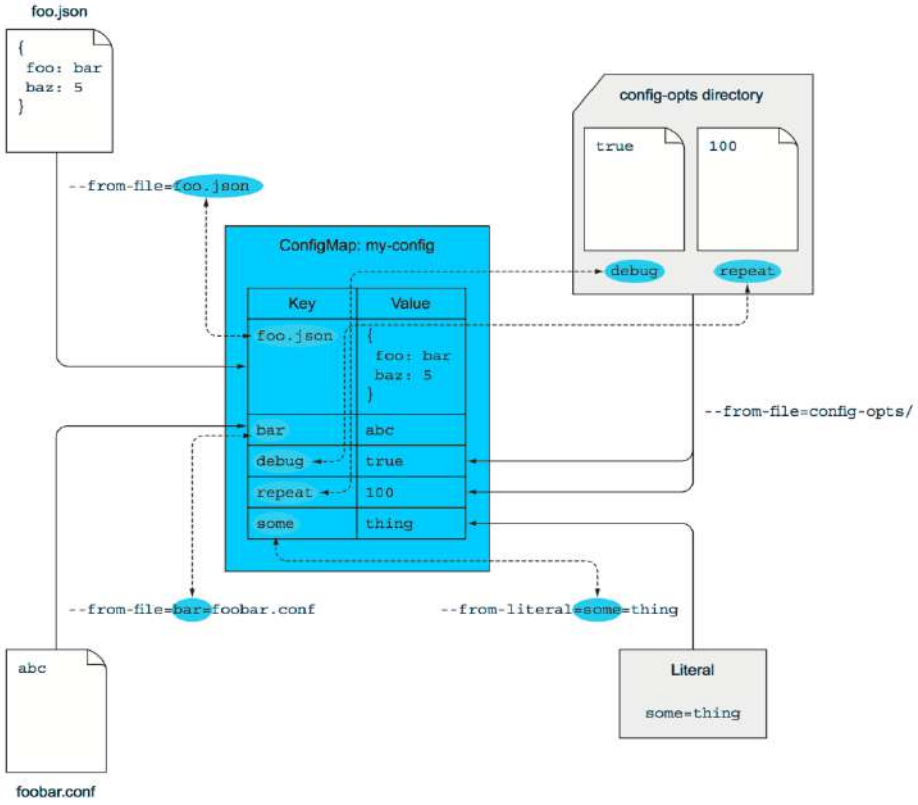
- A partir de un fichero específico.

```
$ kubectl create configmap ejemplo-env-file \
    --from-env-file=app-settings/smtp.properties

$ kubectl get configmap ejemplo-env-file -o yaml
...
data:
  smtp.server: sample-server.com
  smtp.port: 587
  smtp.user: admin
  smtp.password: pa$$w0rd
```

- O mediante literales.

```
$ kubectl create configmap ejemplo-literal-config \
   --from-literal=parametro1=valor1 \
   --from-literal=parametro2=valor2

$ kubectl get configmaps ejemplo-literal-config -o yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: ejemplo-literal-config
  namespace: default
data:
  parametro1: valor1
  parametro2: valor2
```

Por supuesto, podemos crear un ConfigMap utilizando diferentes tipos de fuentes de datos. En la siguiente imagen se muestra cómo se compondría un ConfigMap con varias fuentes y cuáles serían las claves utilizadas según el origen de datos:



Creación de un ConfigMap desde ficheros, directorios y literales.

Por supuesto, también podemos crear un ConfigMap de forma **declarativa**, definiendo el recurso en el fichero de metadatos YAML y creándolo con kubectl create. Su definición deberá incluir la sección **data** con el mismo formato que hemos visto en los ejemplos.

Cómo utilizar los ConfigMaps

Principalmente existen tres formas para tener disponibles las configuraciones almacenadas en los ConfigMaps en nuestros Pods. Según nuestras necesidades utilizaremos una u otra, o incluso una mezcla de ellas.

- Una opción sería **montar el contenido de los ConfigMaps en el sistema de ficheros** de los contenedores del Pod, creando un fichero por clave.

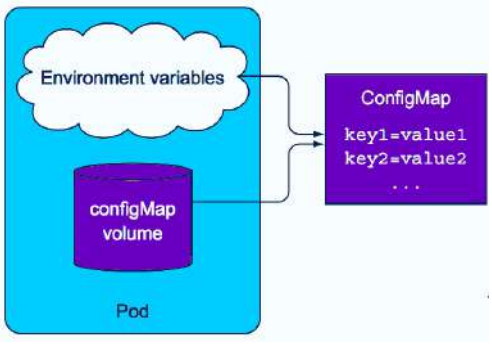
```
...
spec:
  containers:
  - name: app-server
    ...
    volumeMounts:
    - name: dir-config
      mountPath: /etc/my-app/config
      readOnly: true
    ...
  volumes:
  - name: dir-config
    configMap:
      name: ejemplo-dir-config
```

- También podremos establecer de manera dinámica **variables de entorno** con los valores almacenados en el ConfigMap.

```
apiVersion: v1
kind: Pod
metadata:
  name: app-pod
spec:
  containers:
    - image: some-image
      envFrom: # Carga todo el contenido del ConfigMap como variables
      - prefix: config_ # prefijo para todas las variables que van dentro
        configMapRef:
          name: ejemplo-literal-config
    - name: test-container
      env:    # Variables de entorno, una por una
        - name: DB_CONFIG
          valueFrom:
            configMapKeyRef:
              name: ejemplo-dir-config
              key: database.properties
        - name: SMTP_SERVER
          valueFrom:
            configMapKeyRef:
              name: ejemplo-env-file
              key: smtp.server
```

- Y, por último, podemos **establecer dinámicamente los argumentos** del comando de inicio de un contenedor con los valores del ConfigMap. Para ello, definiremos las variables (como acabamos de ver) y utilizaremos la propiedad ***spec.containers.args*** para establecerla.

```
spec:
  containers:
    - name: test-container
      env:
        ...
      args: ["$(SMTP_SERVER)"]
```



Los ConfigMaps se utilizarán mediante variables de entorno o volúmenes.

Ejemplo

Pinchando en este enlace podemos ver un ejemplo de uso de ConfigMaps para configurar un almacenamiento en memoria con Redis.

Secrets

Kubernetes nos ofrece los objetos ***Secret*** para el almacenamiento de información sensible como son las contraseñas, tokens de seguridad, claves ssh, etc. Los objetos Secrets se definen a nivel de Namespace y se almacenarán en la base de datos etcd. La información de los valores estará codificada en **base64**. Es decir, **no se verá en formato legible. Sin embargo, no se cifrará.**

Creación de un Secret

Podemos crear recursos Secret con pares clave/valor a partir de literales, ficheros o directorios, de la misma forma que vimos para los *ConfigMaps*. Para crearlos, utilizaremos el comando ***kubectl create secret generic***:

```
$ kubectl create secret generic db-secret \
   --from-literal=username=admin \
   --from-literal=password='p4$$w0rd'

$ kubectl create secret generic keys-secret \
   --from-file=privatekey=/path/.ssh/id_rsa \
   --from-file=publickey=/path/.ssh/id_rsa.pub

$ kubectl get secret db-secret -o yaml
apiVersion: v1
kind: Secret
type: Opaque
metadata:
  name: db-secret
  namespace: default
data:
  password: cDQkJHcwcmQ=
  username: YWRtaW4=

$ kubectl describe secrets keys-secret
Name:        keys-secret
Namespace:   default
Labels:      <none>
Annotations:  <none>

Type: Opaque
Data
====
id_rsa:     1050 bytes
id_rsa.pub: 1679 bytes
```

Cómo utilizar los Secrets

La utilización de los Secrets en los Pods es también muy similar a lo visto para los ConfigMaps. Veamos algunos ejemplos.

- Pasado de las claves como variables de entorno.

```
env:
- name: DB_PASSWORD
  valueFrom:
    secretKeyRef:
      name: db-secret
      key: password
```

- Montado de un volumen con las claves como ficheros: los volúmenes montados para los Secrets será en tipo tmpfs, es decir, sistemas de ficheros en memoria.

```
spec:
  containers:
    ...
  volumes:
  - name: keys
    secret:
      secretName: keys-secret
```

ReplicaSets, DaemonSets y Servicios

Hasta ahora hemos visto cómo podemos crear Pods de manera individual en nuestro clúster. Sin embargo, normalmente querremos tener más de una copia de nuestros Pods ejecutándose al mismo tiempo. Los principales motivos para tener una replicación de nuestros Pods son:

- **Redundancia**: al tener múltiples instancias de nuestros Pod en ejecución, conseguiremos que el sistema sea tolerante a fallos.
- **Escalabilidad**: nuestros sistemas podrán atender más peticiones o cargas de trabajo simultáneamente.
- **Fragmentación** (sharding): será posible gestionar diferentes partes de los procesos y cálculos de forma paralela en diferentes réplicas.

Además, nos interesará poder aplicar políticas y configuraciones a todo el conjunto de contenedores de un Pod de forma conjunta: conectividad, balanceo de carga, etc.

Por todo esto, Kubernetes nos ofrece una serie de mecanismos para gestionar los Pods de otras maneras, cuyas diferencias veremos a continuación.

ReplicaSets

Los *ReplicaSets* serán los encargados de gestionar nuestros Pods a lo largo del clúster, garantizando que en todo momento se están ejecutando el número y tipo de Pods deseados. Todos aquellos Pods gestionados por un ReplicaSets serán replanificados automáticamente en caso de fallo del nodo en el que se están ejecutando.

A la hora de definir un objeto ReplicaSet, deberemos incluir la propia definición de los Pods que se gestionarán y, por supuesto, cuál es el número de réplicas deseado. Tendrá lugar un **proceso de reconciliación**, que se ejecutará constantemente, observará el estado actual y lo comparará con el deseado. En caso de encontrar diferencias, actuará en consecuencia creando los Pods necesarios. Además, dicho proceso de reconciliación del ReplicaSets deberá ser capaz de identificar cuáles son los Pods que gestiona.

Los ReplicaSets son **idóneos para microservicios sin estado** y, habitualmente, utilizaremos un balanceador de carga para repartir el tráfico entre los Pods gestionados por el ReplicaSet. Al escalar hacia abajo, reduciendo el número de réplicas, el ReplicaSet elegirá uno de sus Pods al azar para su eliminación, por lo que nuestra aplicación deberá estar preparada para no verse afectada por ello.

Definición y creación de ReplicaSets

Al igual que con el resto de los objetos de Kubernetes, definiremos a los ReplicaSets mediante ficheros YAML:

```
---
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: web-rs
  labels:
            env: dev
            role: web
spec:
  replicas: 4
  selector:
            matchLabels:
              role: web
  template:
            metadata:
              labels:
                role: web
            spec:
              containers:
              - name: nginx-server
                image: nginx
```

La sección ***spec*** es obligatoria, y allí indicaremos el número de réplicas deseado y la ***template*** (plantilla) que define los Pods que serán creados por el ReplicaSet. En la definición de los Pods será obligatorio definir qué etiquetas tendrán, las cuales serán utilizadas para identificar a los Pods que pertenecen al ReplicaSet. No deberíamos utilizar etiquetas que ya estén siendo utilizadas, ya que los Pods existentes podrían ser adoptados por el ReplicaSet.

Por último, en la propiedad ***spec.selector*** definiremos el selector que se utilizará para identificar a los Pods del ReplicaSet. Este debería utilizar un subconjunto de las etiquetas definidas en la plantilla de los Pods. En caso de no definir ningún selector, se utilizarán todas las etiquetas definidas en la plantilla del Pod.

Veamos un sencillo **ejemplo de definición de un objeto ReplicaSet**:

Una vez que tengamos creado el fichero de configuración del ReplicaSet en formato YAML, usaremos el comando kubectl apply para enviar la definición del ReplicaSet al API de Kubernetes y crear el nuevo recurso.

Una vez creado, el controlador del ReplicaSet detectará que aún no existe ningún Pod en ejecución según el selector definido y creará tantos Pods como réplicas hayamos especificado:

```
$ kubectl apply -f web-rs.yaml
replicaset.apps/web-rs created

$ kubectl get pods
NAME         READY   STATUS            RESTARTS   AGE
web-rs-6j7kq   1/1     Running           0          14s
web-rs-6q6f4   1/1     Running           0          14s
web-rs-k5mw4   1/1     Running           0          14s
web-rs-tr5ld   0/1     ContainerCreating   0          14s

$ kubectl describe rs web-rs
Name:         web-rs
Namespace:    default
Selector:     role=web
Labels:       env=dev
              role=web
Replicas:     4 current / 4 desired
Pods Status:  4 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  role=web
  Containers:
   nginx-server:
    Image:        nginx
```

Si queremos encontrar todos aquellos Pods que son gestionados por un ReplicaSet, podemos listarlos filtrándolos por el selector que tiene configurado.

```
$ kubectl get pods --selector role=web
```

En caso de que tengamos un Pod y no estemos seguros de si está gestionado por un ReplicaSet u otro controlador, podremos obtener su definición en formato YAML y revisar en la sección ***ownerReferences*** quién es el propietario:

```
$ kubectl get pods web-rs-k5mw4 -o yaml
...
  ownerReferences:
  - apiVersion: apps/v1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicaSet
    name: web-rs
...
```

Borrado de ReplicaSets

Al eliminar un ReplicaSet, por defecto también serán eliminados automáticamente aquellos Pods gestionados por él. En caso de que no queramos que se eliminen sus Pods, deberemos utilizar la opción ***--cascade=false***. De esta manera, los Pods seguirán ejecutándose y podremos inspeccionarlos y eliminarlos manualmente cuando ya no los necesitemos:

$ **kubectl delete rs** web-rs **--cascade=false**

Escalado de Pods

Existen diferentes tipos de objetos en Kubernetes que mantienen funcionando un conjunto de réplicas de Pods. Acabamos de ver los ReplicaSets, más adelante también veremos los *Deployments*. Y, además, existen los StatefulSets, o ReplicationControllers.

> Reto
>
> Usando la documentación presente en Kubernetes.io, encuentra las diferencias entre estos cuatro componentes.

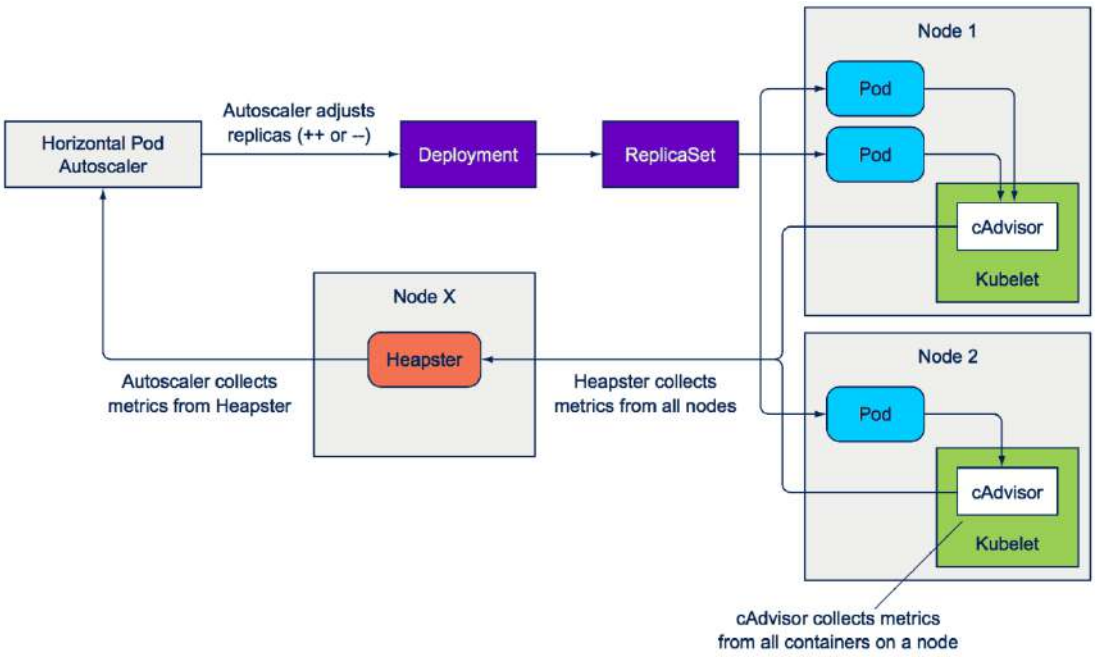
Este escalado puede ser realizado de forma manual, ya sea de forma declarativa mediante la modificación del manifiesto de nuestro componente, o de forma imperativa, usando el comando ***kubectl scale:***

$ **kubectl scale** replicasets web-rs **--replicas=6**

Horizontal Pod Autoscaler (HPA)

El objeto **Horizontal Pod Autoscaler** (HPA) nos permite modificar automáticamente, en base a ciertas métricas, el número de réplicas de Pods de los recursos de tipo ReplicaSet, Deployment, StatefulSet, o ReplicationController.

Por defecto, Kubernetes nos permite utilizar el consumo tanto de CPU como de memoria en la definición de métricas para el autoescalado, sin embargo, también sería posible definir métricas personalizadas e, incluso, externas. En la siguiente imagen podemos ver resumido su funcionamiento:



Arquitectura del *Horizontal Pod Autoscaler* (HPA).

El comando ***kubectl autoscale*** nos permite crear de manera imperativa un Horizontal Pod Autoscaler (HPA), indicando el tipo de controlador que autoescalará, el mínimo y el máximo del número de réplicas permitidas, así como el porcentaje de uso de CPU que disparará el autoescalado.

$ **kubectl autoscale rs** nombre-replicaset **--min=2 --max=5 --cpu-percent=80**

El fichero de configuración YAML incluirá una sección ***spec.scaleTargetRef*** en la que indicaremos una referencia al objeto que queremos autoescalar, así como los parámetros que definirán el autoescalado:

```
---
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: web-rs-scaler
spec:
  scaleTargetRef:
    kind: ReplicaSet
    name: web-rs
  minReplicas: 3
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Actualmente, el autoescalado basado en el uso de memoria está en versión beta (autoscaling/v2beta2). Consulta la documentación oficial del API para más detalles sobre su configuración.

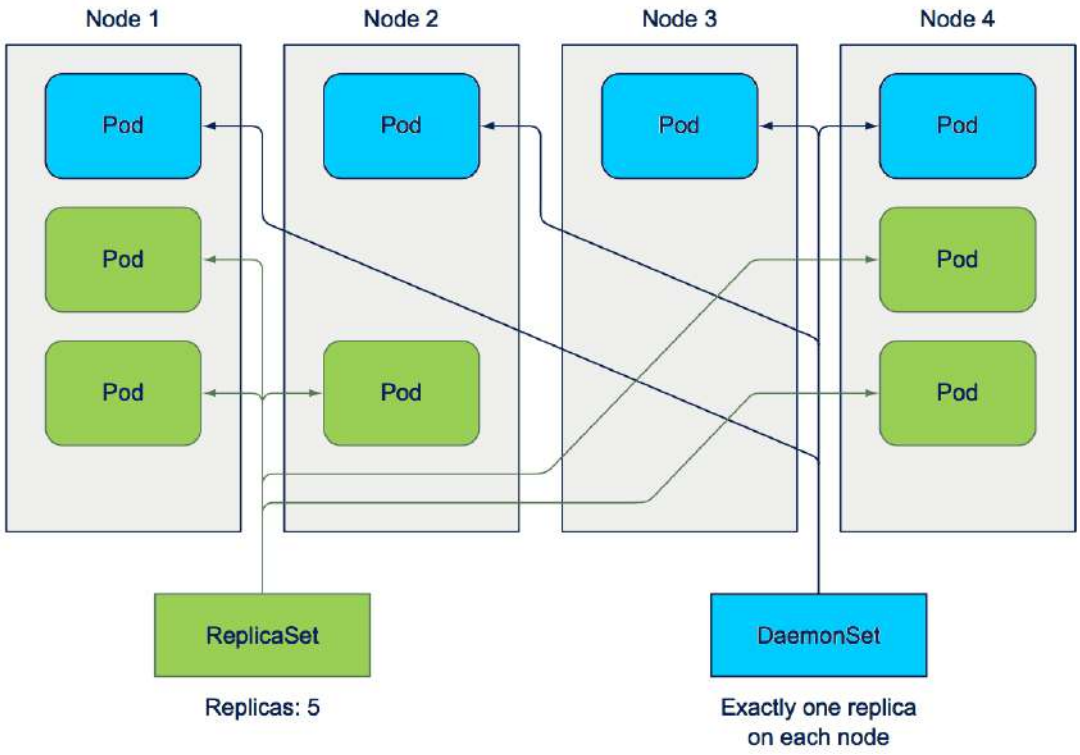
Importante

Cuando estemos utilizando un HPA no deberíamos escalar manualmente estos objetos, sino que deberemos dejar que sea el propio HPA el que modifique el número de réplicas en dichos objetos. Realmente, ambos objetos están desacoplados, por lo que técnicamente podríamos hacerlo, sin embargo, es posible que haya conflictos y obtengamos comportamientos no esperados.

DaemonSets

Acabamos de ver cómo los ReplicaSets nos permiten ejecutar cierto número de Pods distribuidos a lo largo de un clúster de Kubernetes, siendo el propio ReplicaSet el que decide dónde ejecutar los Pods. Sin embargo, a veces necesitaremos ejecutar un Pod en todos y cada uno de los nodos del clúster o, al menos, en un conjunto específico de ellos.

Los ***DaemonSets*** se aseguran de que en todos los nodos del clúster, o en aquellos que hemos seleccionado, se ejecute un Pod determinado. Además, se encargan de ejecutar un nuevo Pod cuando nuevos nodos son añadidos al clúster. Ejemplos de **casos de uso** de los DaemonSets serían la recolección de logs o la monitorización de los recursos de cada nodo.



Diferencias entre ReplicaSets y DaemonSets desplegando los Pods.

Definición de un DaemonSet

Al igual que el resto de los objetos, necesitaremos especificar la versión, el tipo de objeto y los metadatos. La versión utilizada para los DaemonSets es *apps/v1*. Además, deberemos incluir una sección *spec* en la que al menos incluyamos la template con la definición de los Pods que se van a ejecutar en los nodos. Esta definición es similar a la que ya vimos para los Pods.

Veamos un ejemplo de un DaemonSet:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: ssd-monitor
spec:
  selector:
    matchLabels:
      app: ssd-monitor
  template:
    metadata:
      labels:
        app: ssd-monitor
    spec:
      containers:
      - name: main
        image: ssd-monitor-image
```

```
$ kubectl apply -f ssd-monitor.yaml
daemonset "ssd-monitor" created
$ kubectl describe daemonset ssd-monitor
$ kubectl get pods --selector=ssd-monitor -o wide
```

Ejecución de Pods en (solo) ciertos nodos

Cuando usamos los DaemonSets, normalmente querremos desplegar el Pod en todos los nodos, sin embargo, a veces solamente desearemos hacerlo en algunos de ellos (por ejemplo, para monitorizar los nodos que poseen una característica determinada). Para limitar los nodos donde desplegar los Pods tenemos **dos alternativas**: utilizar un selector de nodos o especificar la afinidad de nodo.

Selectores de nodo (nodeSelector)

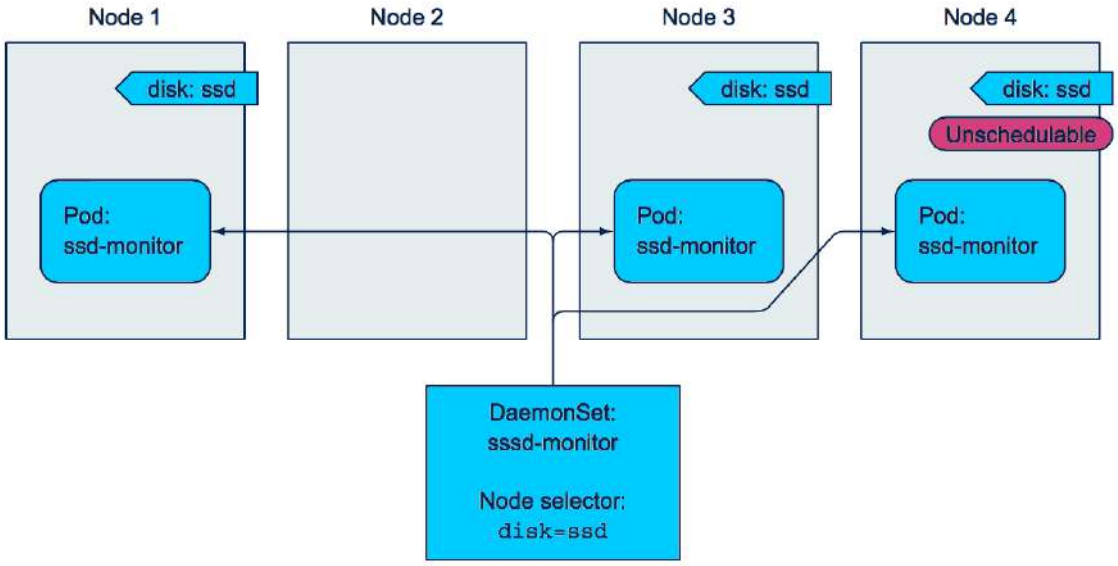
Imaginemos que queremos monitorizar con un Pod todos aquellos nodos que utilizan discos SSD. Lo primero que deberemos hacer es etiquetar de alguna manera los nodos que utilizan ese tipo de disco. Esto lo haremos con kubectl label como ya vimos.

```
$ kubectl label nodes <nobre-nodo> disk=ssd
$ kubectl get nodes --selector disk=ssd
```

Ahora ya podemos definir el selector en la definición del DaemonSet. Esto lo haremos en la propiedad ***spec.template.spec.nodeSelector*** donde indicaremos la lista de etiquetas del selector. Veamos qué cambios haríamos en el ejemplo anterior:

```
...
spec:
  ...
  template:
    spec:
      nodeSelector:
        disk: ssd
      containers:
      - name: main
        image: ssd-monitor-image
```

La siguiente imagen muestra cómo los Pods solamente serían desplegados en aquellos Pods que coinciden con el selector definido:



Los selectores de nodos permiten desplegar los Pods solo en ciertos nodos.

Afinidad de nodo (affinity)

La selección de nodos en base a la afinidad es similar a los selectores de nodo, ya que restringe los nodos en base a sus etiquetas. Sin embargo, la afinidad establece reglas que deben cumplirse en el momento de la planificación de los nodos, pero no después. Estas se definen en la propiedad ***spec.template.spec.affinity***. Veamos un ejemplo:

```
spec:
 affinity:
  nodeAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchExpressions:
     - key: kubernetes.io/e2e-az-name
       operator: In
       values:
       - e2e-az1
       - e2e-az2
```

Para más información acerca de los selectores de nodo y la afinidad, consulta la documentación oficial de Kubernetes.

Borrado de un DaemonSet

Para borrar un DaemonSet, utilizaremos el comando ***kubectl delete daemonset*** pasándole el nombre, o bien ***kubectl delete*** con el fichero YAML que habíamos utilizado para su creación. Como ocurría con los ReplicaSet, una vez que borramos un DaemonSet, todo los Pods gestionados por él son también eliminados, a menos que utilicemos el argumento *--cascade=false*.

```
$ kubectl delete -f sample-daemonset.yaml  --cascade=false
```

Servicios

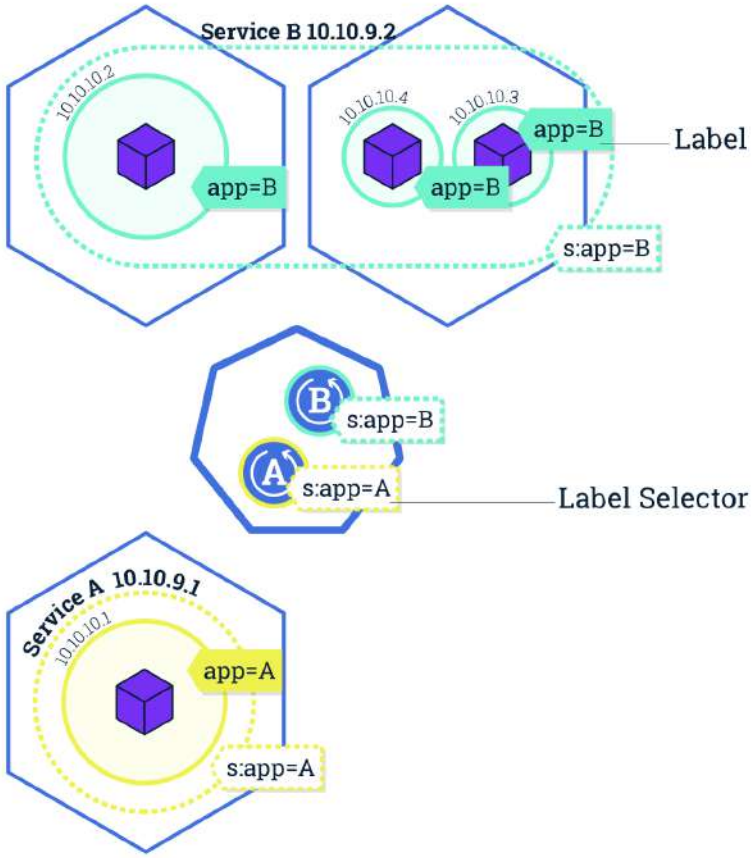
En Kubernetes, todos los Pods del clúster tendrán asociada una IP interna única. Además, todos los Pods de un nodo tendrán conectividad con el resto de Pods del clúster a través de sus direcciones IP internas, independientemente del nodo en el que estén.

Sin embargo, cada vez que se cree de nuevo un Pod, porque se haya recreado debido a un fallo o una actualización, se le asignará una nueva dirección IP. Por ello, necesitamos una manera de reconciliar estos cambios, para que nuestras aplicaciones puedan comunicarse entre sí y seguir funcionando.

Kubernetes nos ofrece una abstracción denominada ***Service*** que permite a las aplicaciones desplegadas en el clúster comunicarse fácilmente entre sí y con el exterior.

Los *Service* siempre estarán apuntando a Pods del clúster y nunca a otros objetos como los Deployments o los ReplicaSets. La manera de determinar a qué Pods apunta un Servicio es mediante las etiquetas de los Pods, utilizando selectores. Esto nos ofrece una **gran flexibilidad** a la hora de configurar los servicios, ya que es indiferente a cómo fueron creados dichos Pods.

En el siguiente escenario tenemos dos Servicios, A y B, que utilizan selectores de la etiqueta app para apuntar a los Pods de los Deployments A y B respectivamente:



Los Servicios se emparejan con Pods utilizando etiquetas y selectores.

Aunque todos los Pods tengan una dirección IP única, esta no será expuesta al exterior a menos que sea a través de un Servicio. La manera en que los Pods son expuestos dependen del tipo de servicio que utilicemos.

ClusterIP

Un servicio de tipo ***ClusterIP*** distribuirá las peticiones recibidas entre los Pods a los que apunta.

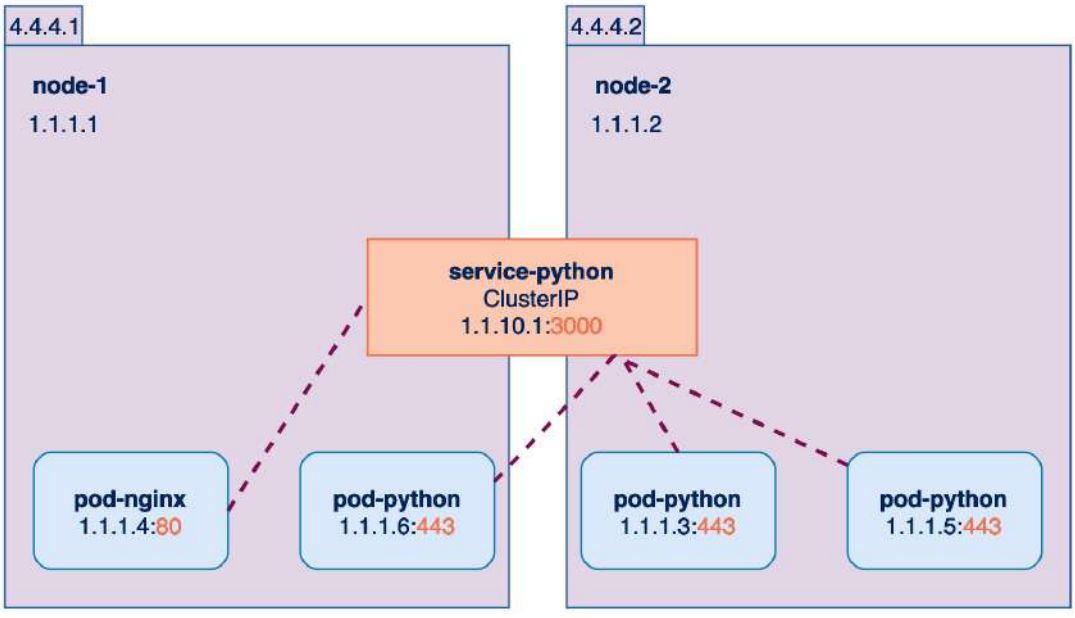
En la definición YAML de un servicio de tipo ClusterIP especificaremos un selector de Pods para indicar a qué Pods redirigir el tráfico y, además, los puertos que queremos utilizar en la propiedad ***spec.ports***. Deberemos especificar:

- El puerto en que estará disponible el servicio (**port**).
- El puerto del contenedor al que se redirigirán las peticiones (*targetPort*).
- El nombre que le queremos dar al puerto (**name**), pero solo es obligatorio cuando se define más de uno.

El siguiente ejemplo crearía un servicio interno al clúster que atiende peticiones TCP en el puerto 3000 y las redirige al puerto 443 de uno de los Pods seleccionados por el selector:

```
---
apiVersion: v1
kind: Service
metadata:
  name: service-python
spec:
  selector:
    run: pod-python
  ports:
  - port: 3000
    protocol: TCP
    targetPort: 443
  type: ClusterIP

$ kubectl apply -f service-python.yaml
service/service-python created
```

Ahora cualquier Pod del clúster puede fácilmente enviar peticiones a los Pods definidos por el selector a través del servicio creado, sin verse afectado por los cambios de IP al recrear los Pods:



ClusterIP distribuye peticiones internas entre los Pods.

Una vez creado el servicio mediante el fichero YAML, si listamos los servicios en el namespace, veremos en la columna CLUSTER-IP la dirección IP interna que se le ha asignado al servicio:

```
$ kubectl get services
```

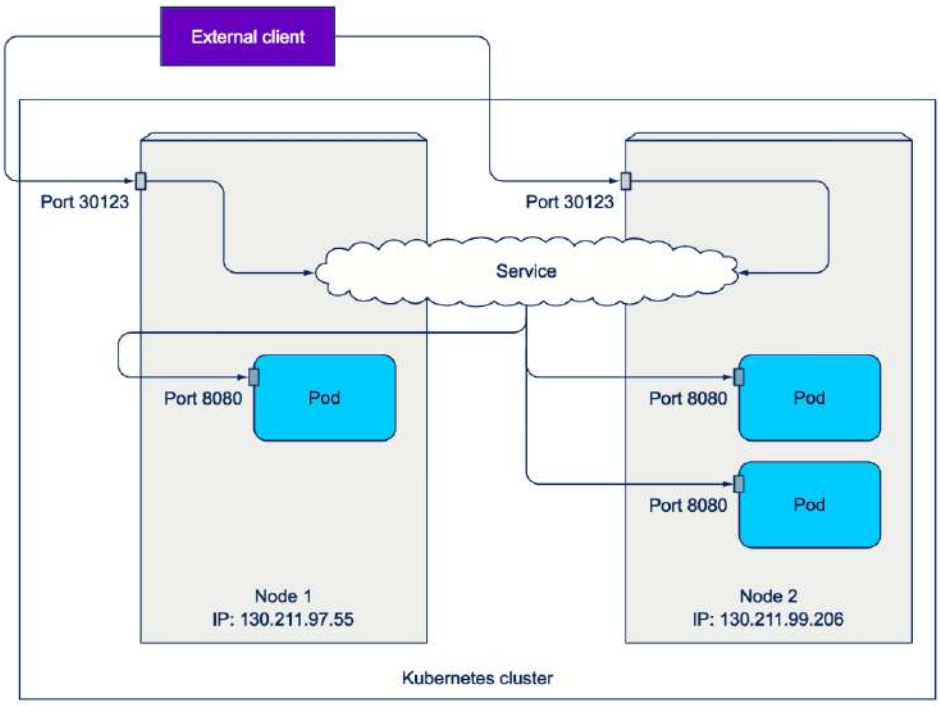
```
NAME          TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
kubernetes    ClusterIP  10.96.0.1       <none>        443/TCP    35d
service-python ClusterIP 10.97.135.121   <none>        3000/TCP   14s
```

NodePort

Los servicios de tipo NodePort ya nos permiten hacer disponibles nuestros Pods fuera del clúster. La definición YAML será similar a la anterior, pero ahora deberemos especificar también el puerto en el que cada uno de los nodos del clúster (***spec.ports.nodePort***) atenderá las peticiones que serán redirigidas al servicio.

```
...
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8080
    nodePort: 30123
  selector:
    app: web
  type: NodePort
```

Ahora los Pods de nuestra aplicación web serán accesibles a través de las IPs internas y externas de los nodos en el puerto 30123.



NodePort permite acceder a través de la IP pública de los nodos.

Minikube / Docker Desktop

En el caso de estar usando una instalación local de Kubernetes, la 'dirección externa' del clúster será 127.0.0.1

LoadBalancer

La mayoría de los proveedores de nube ofrecen la creación de balanceadores de carga en su infraestructura. Cuando nuestro clúster está desplegado en uno de estos proveedores, podemos aprovechar dicha característica y crear automáticamente un balanceador de carga para nuestros servicios.

Al crear un servicio de tipo LoadBalancer, se creará un balanceador de carga con su propia IP pública, el cual distribuirá las peticiones recibidas a las IP públicas de todos los nodos. En caso de que nuestro clúster esté ejecutándose en un entorno que no soporta balanceadores de carga, el servicio se comportará **exactamente igual que el tipo NodePort.**

```
---
apiVersion: v1
kind: Service
…
  selector:
    app: web
  type: LoadBalancer
```

Si listamos los servicios, veremos que ahora la dirección externa mostrada será la del balanceador de carga:

```
$ kubectl get services service-web-lb
NAME          TYPE          CLUSTER-IP      EXTERNAL-IP     PORT(S)       AGE
service-web-lb   LoadBalancer   10.103.219.103   130.222.53.125   80:30123/TCP   22s
```

ExternalName

Los servicios de tipo ***ExternalName*** nos permiten crear un alias para un servicio externo al clúster. En este tipo de servicios no utilizaremos ningún selector ni obtendrán ninguna dirección IP del clúster. Simplemente mapearán el nombre del servicio con un DNS externo, el cual se especificará como un registro CNAME en la definición YAML.

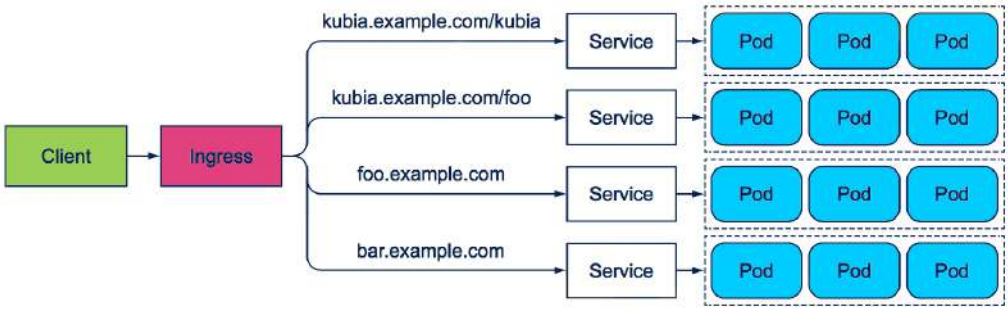
```
---
apiVersion: v1
kind: Service
metadata:
  name: servicio-externo
spec:
  type: ExternalName
  externalName: servidor.remoto.com
  ports:
  - port: 80
```

Accederemos al servicio de la misma forma que los anteriores, sin embargo, ahora la redirección será a nivel de DNS.

Exposición de múltiples servicios con Ingress

Además de los servicios de tipo *NodePort* y *LoadBalancer*, disponemos de otra manera de acceder a nuestros Pods desde fuera del clúster. Kubernetes nos ofrece una abstracción más, denominada ***Ingress***, para el balanceo de carga en el clúster.

Los objetos *Ingress* actuarán como punto de entrada al clúster, permitiéndonos utilizar la misma dirección IP para exponer múltiples servicios, mediante HTTP(S), a través de las reglas de enrutamiento.



Exposición de múltiples servicios mediante un único recurso *Ingress.*

Cuando los clientes envían peticiones HTTP a un recurso Ingress, la petición se redirige a un servicio determinado en base a ciertas reglas previamente configuradas en el objeto. Un recurso Ingress solamente expone puertos HTTP y HTTPS; si queremos exponer otros puertos deberemos utilizar un servicio de tipo NodePort o LoadBalancer.

En el siguiente ejemplo se define un sencillo recurso Ingress con una regla para redirigir las peticiones al host *app.ejemplo.com* al servicio web-rs en su puerto 80:

```
---
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ejemplo-ingress
spec:
  rules:
  - host: app.ejemplo.com
    http:
      paths:
      - path: /
        backend:
          serviceName: web-rs
          servicePort: 80

$ kubectl get ingresses
NAME            HOSTS          ADDRESS         PORTS    AGE
ejemplo-ingress   app.ejmplo.com   192.168.99.100   80       29m
```

Para más información, consulta la documentación oficial de Kubernetes.

LA UNIVERSIDAD EN INTERNET | FORMACIÓN PROFESIONAL

PROEDUCA