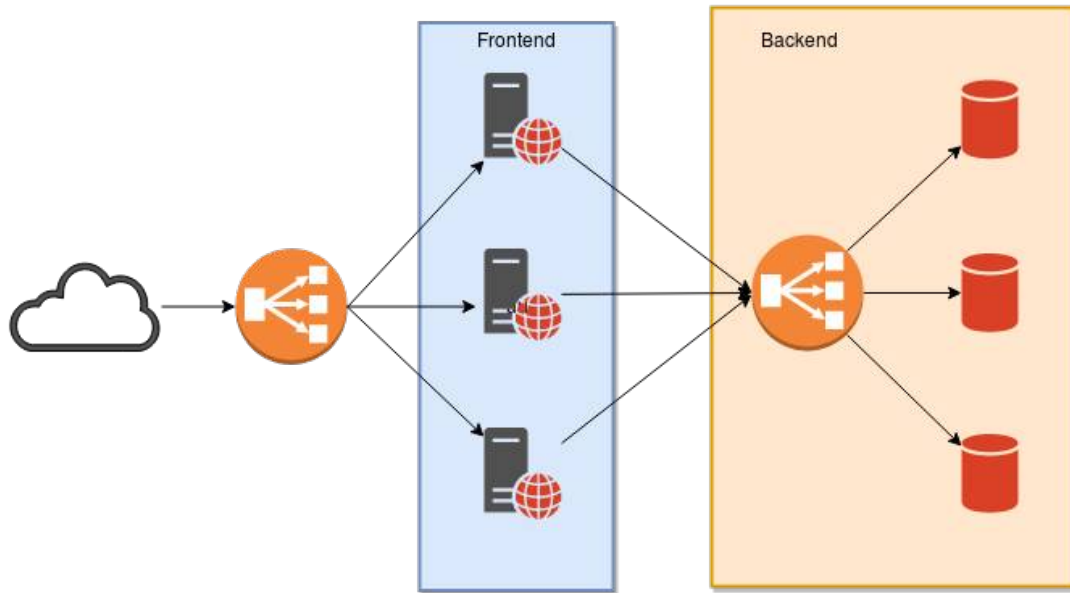


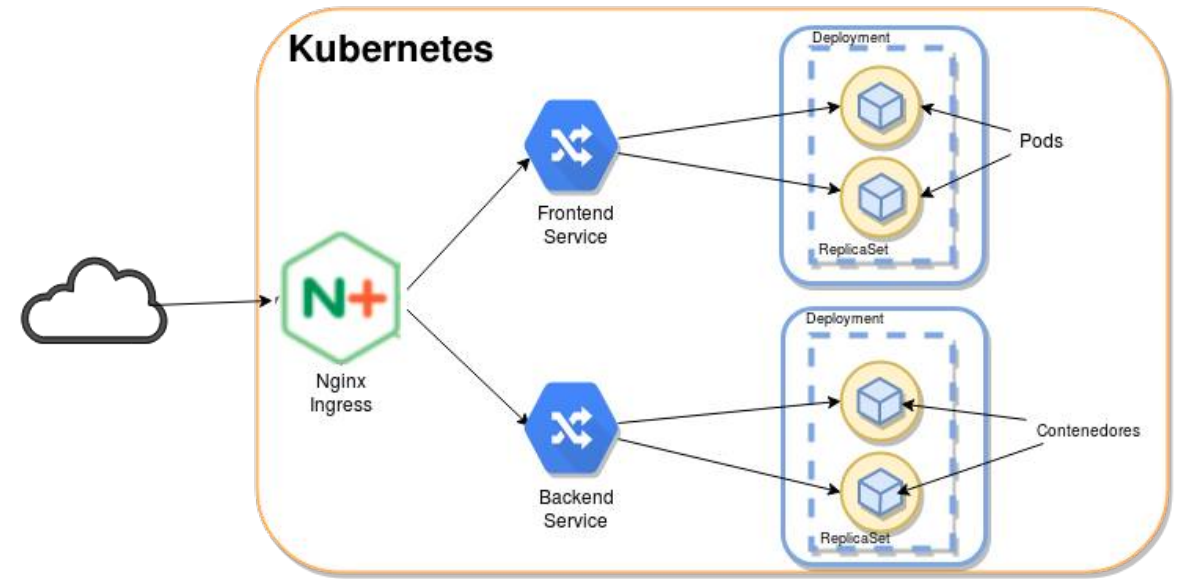
UNIDAD 2

DESPLIEGUE DE APLICACIONES EN KUBERNETES

Despliegue de aplicaciones en Kubernetes



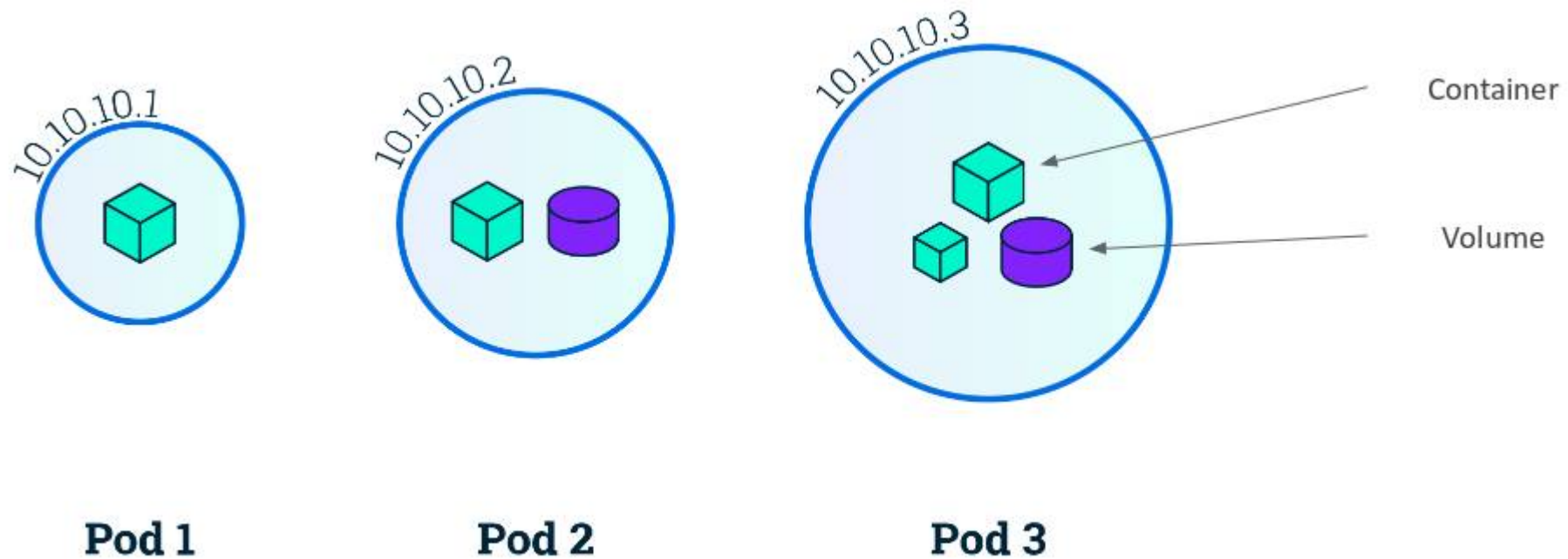
- Máquinas físicas/virtuales
- Balanceadores de carga
- Proxys inversos



- Pods
- ReplicaSets
 - Que no haya caída del servicio
 - Tolerancia a errores
 - Escalabilidad dinámica
- Deployments
 - Actualizaciones continuas
 - Despliegues automáticos
- Services
 - Acceso a los pods
 - Balanceo de carga
- Ingress
 - Acceso por nombres
- Otros recursos
 - Migraciones sencillas
 - Monitorización
 - Control de acceso basada en Roles
 - Integración y despliegue continuo

Pod

La unidad más pequeña de kubernetes son los **Pods**, con los que podemos correr **contenedores**. Un pod representa un **conjunto de contenedores** que comparten **almacenamiento y una única IP**. Los pods son **efímeros**, cuando se destruyen se pierde toda la información que contenía. Si queremos desarrollar aplicaciones persistentes tenemos que utilizar **volúmenes**.



Pod

Por lo tanto, aunque Kubernetes es un orquestador de contenedores, **la unidad mínima de ejecución son los pods:**

- Si seguimos el principio de un proceso por contenedor, nos evitamos tener sistemas (como máquinas virtuales) ejecutando docenas de procesos,
- pero en determinadas circunstancias necesito más de un proceso para que se ejecute mi servicio.

Por lo tanto parece razonable que podamos tener más de un contenedor compartiendo almacenamiento y direccionamiento, que llamamos **Pod**. Además existen más razones:

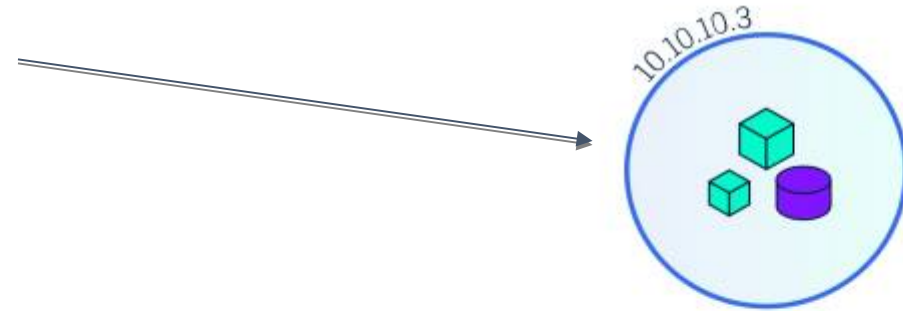
- Kubernetes puede trabajar con distintos contenedores (Docker, Rocket, cri-o,...) por lo tanto es necesario añadir una capa de abstracción que maneje las distintas clases de contenedores.
- Además esta capa de abstracción añade información adicional necesaria en Kubernetes como por ejemplo, políticas de reinicio, comprobación de que la aplicación esté inicializada (readiness probe), comprobación de que la aplicación haya realizado alguna acción especificada (liveness probe), ...

Podemos crear un pod directamente con kubectl:

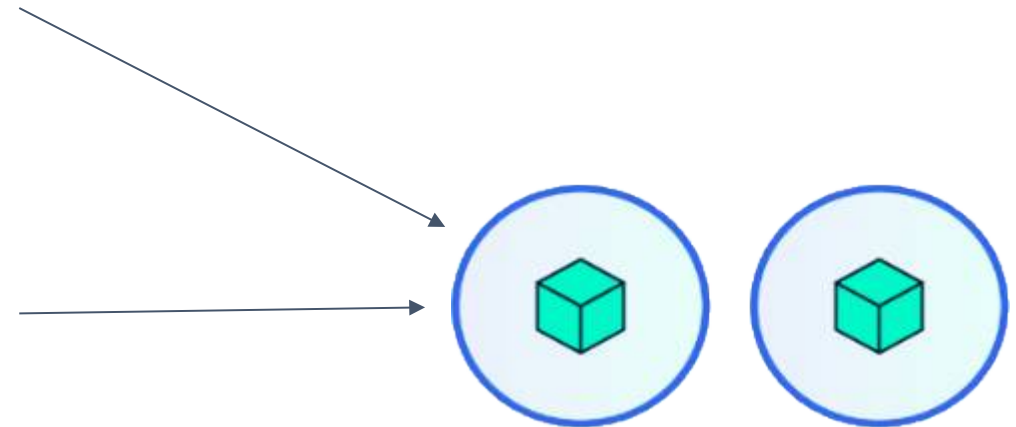
```
kubectl run nginx --image=nginx
```

¿Pod multicontenedor o múltiples pods?

1. Un servidor web nginx con un servidor de aplicaciones PHP-FPM, lo podemos implementar en un pod, y cada servicio en un contenedor.



2. Una aplicación WordPress con una base de datos mariadb, lo implementamos en dos pods diferenciados, uno para cada servicio.



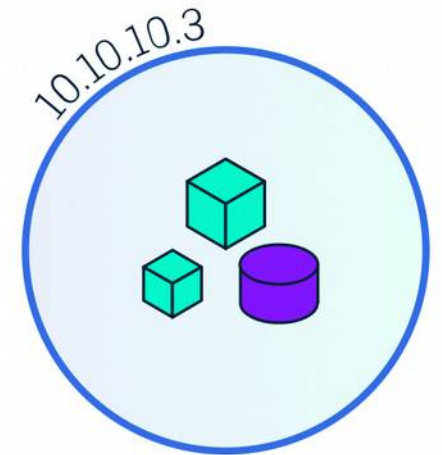
3. Un servidor de base de datos mongodb master con slaves.

Algunas notas sobre pods multicontenedores...

La razón principal por la que los Pods pueden tener **múltiples contenedores** es para admitir aplicaciones auxiliares que ayudan a una aplicación primaria. Ejemplos típicos de estas aplicaciones pueden ser las que envían o recogen datos externos (por ejemplo de un repositorio) y los servidores proxy. El ayudante y las aplicaciones primarias a menudo necesitan comunicarse entre sí. Normalmente, esto se realiza a través de un sistema de archivos compartido o mediante la interfaz de red de bucle de retorno, localhost. **Un ejemplo de este patrón es un servidor web junto con un programa auxiliar que sondea un repositorio Git en busca de nuevas actualizaciones.**

Para seguir aprendiendo...

- [Communicate Between Containers in the Same Pod Using a Shared Volume](#)
- [Multi-container pods and container communication in Kubernetes](#)



Describiendo objetos k8s: Pod

Una de la forma de describir los objetos en k8s es usando un fichero escrito en yaml, por ejemplo podemos describir un pod:

```
apiVersion: v1 # required
kind: Pod # required
metadata: # required
  name: nginx # required
  namespace: default
  labels:
    app: nginx
    service: web
spec: # required
  containers:
    - image: nginx:1.16
      name: nginx
      imagePullPolicy: Always
```

Para más información acerca de la estructura de la definición de los objetos de Kubernetes:

[Understanding Kubernetes Objects.](#)

- `apiVersion: v1`: La versión de la API que vamos a usar.
- `kind: Pod`: La clase de recurso que estamos definiendo.
- `metadata`: Información que nos permite identificar unívocamente al recurso.
- `spec`: Definimos las características del recurso. En el caso de un pod indicamos los contenedores que van a formar el pod, en este caso sólo uno.

- Las imágenes se guardan en un registro interno.
- Se pueden utilizar registros públicos (google, docker hub,...) y registros privados.
- La política por defecto es `IfNotPresent`, que se baja la imagen si no está en el registro interno. Si queremos forzar la descarga:
 - `imagePullPolicy: Always`
 - Omitir `imagePullPolicy` y usar la etiqueta `:latest`
 - Omitir `imagePullPolicy` y la etiqueta de la imagen

Trabajando con Pod: EJEMPLO 1

Para crear el pod desde el fichero yaml:

```
kubectl create -f pod.yaml
```

Y podemos ver que el pod se ha creado:

```
kubectl get pods
```

Podemos modificar las características de cualquier recurso de kubernetes una vez creado, por ejemplo podemos modificar la definición del pod de la siguiente manera:

```
kubectl edit pod nginx  
KUBE_EDITOR="nano" kubectl edit pod nginx
```

Si queremos saber en qué nodo del cluster se está ejecutando:

```
kubectl get pod -o wide
```

Para obtener información más detallada del pod:

```
kubectl describe pod nginx
```

Para eliminar el pod:

```
kubectl delete pod nginx
```

Para más información acerca de los pod puedes leer: la [documentación de la API](#) y la [guía de usuario](#).

Accediendo a los pod con kubectl

Para obtener los logs del pod:

```
kubectl logs nginx
```

Si quiero conectarme al contenedor:

```
kubectl exec -it nginx -- /bin/bash
```

Podemos acceder a la aplicación, redirigiendo un puerto de localhost al puerto de la aplicación:

```
kubectl port-forward nginx 8080:80
```

Y accedemos al servidor web en la url `http://localhost:8080`.

Labels

Las [Labels](#) nos permiten etiquetar los recursos de kubernetes (por ejemplo un pod) con información del tipo clave/valor.

Para obtener las labels de los pods que hemos creado:

```
kubectl get pods --show-labels
```

Los Labels lo hemos definido en la sección metada del fichero yaml, pero también podemos añadirlos a los pods ya creados:

```
kubectl label pods nginx service=web --overwrite=true
```

Los Labels me van a permitir seleccionar un recurso determinado, por ejemplo para visualizar los pods que tienen un label con un determinado valor:

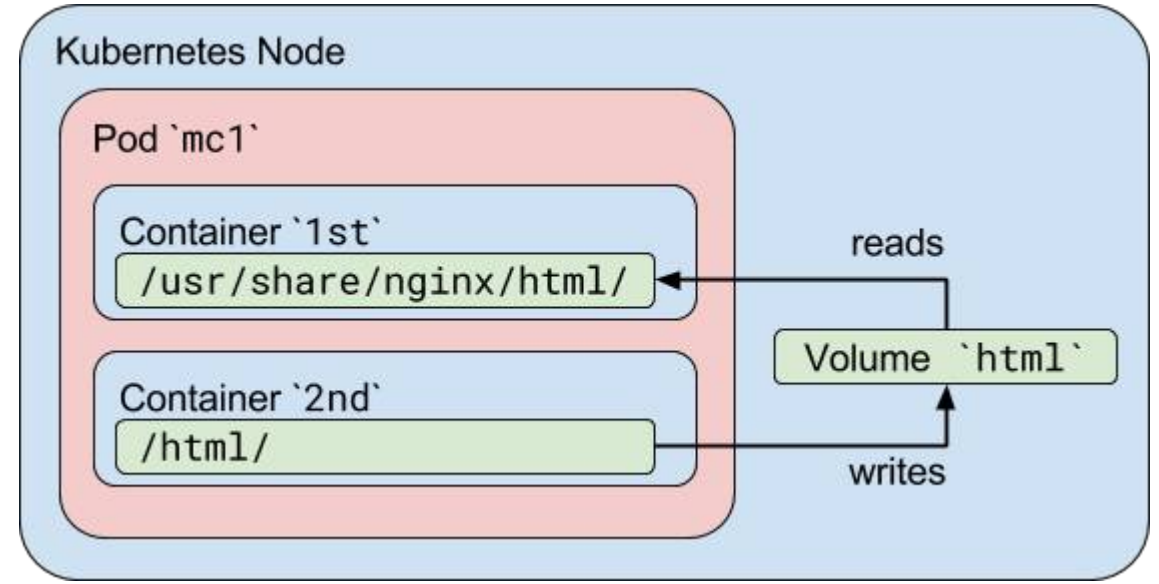
```
kubectl get pods -l service=web
```

También podemos visualizar los valores de los labels como una nueva columna:

```
kubectl get pods -Lservice
```

Pod multicontenedores. EJEMPLO 2

```
apiVersion: v1
kind: Pod
metadata:
  name: mc1
spec:
  volumes:
  - name: html
    emptyDir: {}
  containers:
  - name: 1st
    image: nginx
    volumeMounts:
    - name: html
      mountPath: /usr/share/nginx/html
  - name: 2nd
    image: debian
    volumeMounts:
    - name: html
      mountPath: /html
  command: ["/bin/sh", "-c"]
  args:
  - while true; do
      date >> /html/index.html;
      sleep 1;
    done
```



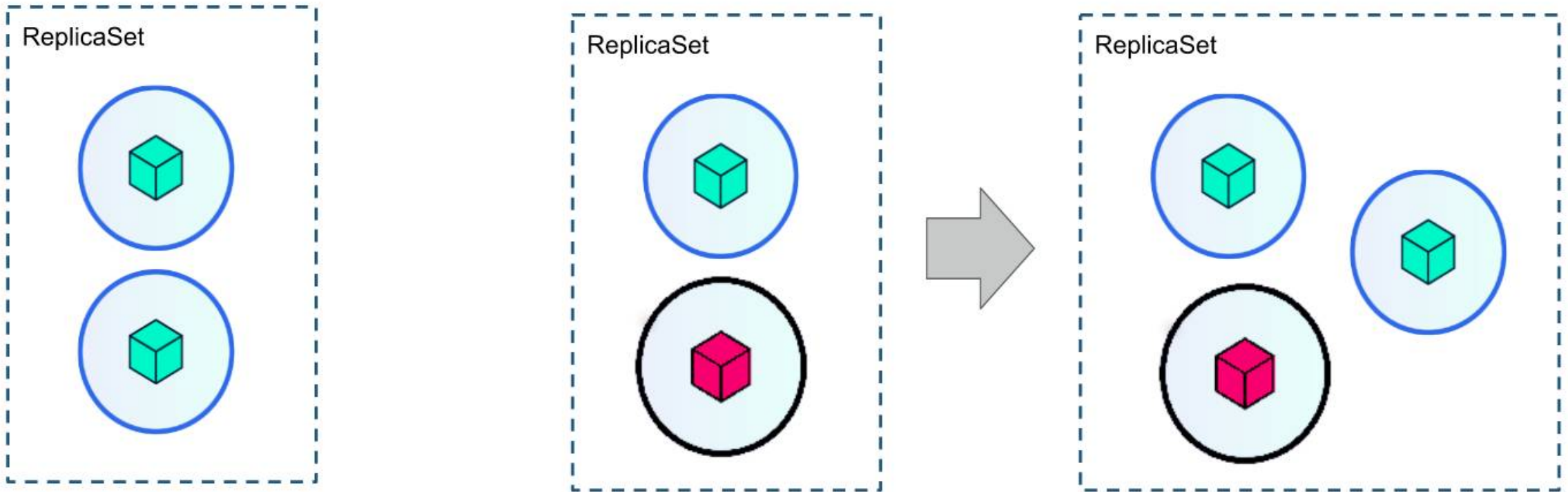
```
kubectl exec mc1 -c 1st -- /bin/cat /usr/share/nginx/html/index.html
```

```
kubectl exec mc1 -c 2nd -- /bin/cat /html/index.html
```

```
kubectl port-forward mc1 8080:80
```

ReplicaSet

ReplicaSet es un recurso de Kubernetes que asegura que siempre se ejecute un número de réplicas de un pod determinado. Por lo tanto, nos asegura que un conjunto de pods siempre están funcionando y disponibles. Nos proporciona las siguientes características: **Tolerancia a fallos** y **Escalabilidad dinámica**.



Describiendo objetos k8s: ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  namespace: default
spec:
  Replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx
          name: nginx
```

- replicas: Indicamos el número de pos que siempre se van a estar ejecutando.
- selector: Seleccionamos el recurso que va controlar el replicaset por medio de las etiquetas
- template: El recurso ReplicaSet contiene la definición de un pod.

Trabajando con ReplicaSet: EJEMPLO 3

Creamos el ReplicaSet:

```
kubectl create -f nginx-rs.yaml
```

```
kubectl get rs
```

```
kubectl get pods
```

¿Qué pasaría si borro uno de los pods que se han creado? Inmediatamente se creará uno nuevo para que siempre estén ejecutándose los pods deseados, en este caso 2:

```
kubectl delete pod nginx-5b2rn
```

```
kubectl get pods
```

Para escalar el número de pods:

```
kubectl scale rs nginx --replicas=5
```

```
kubectl get pods --watch
```

Como anteriormente vimos podemos modificar las características de un ReplicaSet con la siguiente instrucción:

```
kubectl edit rs nginx
```

Por último si borramos un ReplicaSet se borrarán todos los pods asociados:

```
kubectl delete rs nginx
```

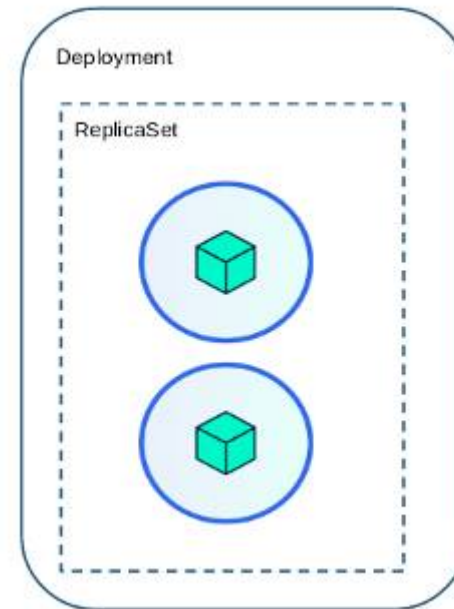
Para más información acerca de los ReplicaSet puedes leer: la [documentación de la API](#) y la [guía de usuario](#).

Deployment

Deployment es la unidad de más alto nivel que podemos gestionar en Kubernetes. Nos permite definir diferentes funciones:

- Control de réplicas
- Escalabilidad de pods
- Actualizaciones continuas
- Despliegues automáticos
- Rollback a versiones anteriores

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



Describiendo objetos k8s: Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
  namespace: default
  labels:
    app: nginx
spec:
  revisionHistoryLimit: 2
  strategy:
    type: RollingUpdate
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx
          name: nginx
          ports:
            - name: http
              containerPort: 80
```

El despliegue de un Deployment crea un ReplicaSet y los Pods correspondientes. Por lo tanto en la definición de un Deployment se define también el ReplicaSet asociado. En la práctica siempre vamos a trabajar con Deployment. Los atributos relacionados con el Deployment que hemos indicado en la definición son:

- **revisionHistoryLimit**: Indicamos cuántos ReplicaSets antiguos deseamos conservar, para poder realizar rollback a estados anteriores. Por defecto, es 10.
- **strategy**: Indica el modo en que se realiza una actualización del Deployment:
 - **recreate**: elimina los Pods antiguos y crea los nuevos;
 - **RollingUpdate**: va creando los nuevos pods, comprueba que funcionan y se eliminan los antiguos.

Trabajando con Deployment: EJEMPLO 4

Cuando creamos un Deployment, se crea el ReplicaSet asociado y todos los pods que hayamos indicado.

```
kubectl create -f nginx-deployment.yaml  
kubectl get deploy,rs,pod
```

Como ocurría con los replicaSets los Deployment también se pueden escalar, aumentando o disminuyendo el número de pods asociados:

```
kubectl scale deployment nginx --replicas=4
```

Otras operaciones:

```
kubectl port-forward deploy/nginx 8080:80  
kubectl logs deploy/nginx
```

Si eliminamos el Deployment se eliminarán el ReplicaSet asociado y los pods que se estaban gestionando.

```
kubectl delete deployment nginx
```

Para más información acerca de los Deployment puedes leer: la [documentación de la API](#) y la [guía de usuario](#).

Actualización de objetos k8s

Para modificar un Deployment (o cualquier objeto k8s):

- Modificando el parámetro directamente. `kubectl set`
- Modificando el fichero yaml y aplicando el cambio. `kubectl apply -f`
- Modificando la definición del objeto: `kubectl edit deployment ...`

Por ejemplo para hacer una actualización de la aplicación:

- `kubectl set image deployment nginx nginx=nginx:1.16 -all`
- Modifico el fichero `deployment.yaml` y ejecuto:
 - `kubectl apply -f deployment.yaml`
- `kubectl edit deployment nginx`

Actualización y Rollout de la aplicación

```
kubectl set image deployment nginx nginx=nginx:1.16 --all
```

Comprobamos que se ha creado un nuevo ReplicaSet, y unos nuevos pods con la nueva versión de la imagen.

```
kubectl get rs  
kubectl get pods
```

La opción `--all` fuerza a actualizar todos los pods aunque no estén inicializados.

Si queremos volver a la versión anterior de nuestro despliegue, tenemos que ejecutar:

```
kubectl rollout undo deployment nginx
```

Y comprobamos como se activa el antiguo ReplicaSet y se crean nuevos pods con la versión anterior de nuestra aplicación:

```
kubectl get rs
```

EJEMPLO 5: Desplegando aplicación mediaWiki

Vamos a desplegar la aplicación mediawiki:

```
kubectl apply -f mediawiki-deploy.yaml --record  
kubectl get all
```

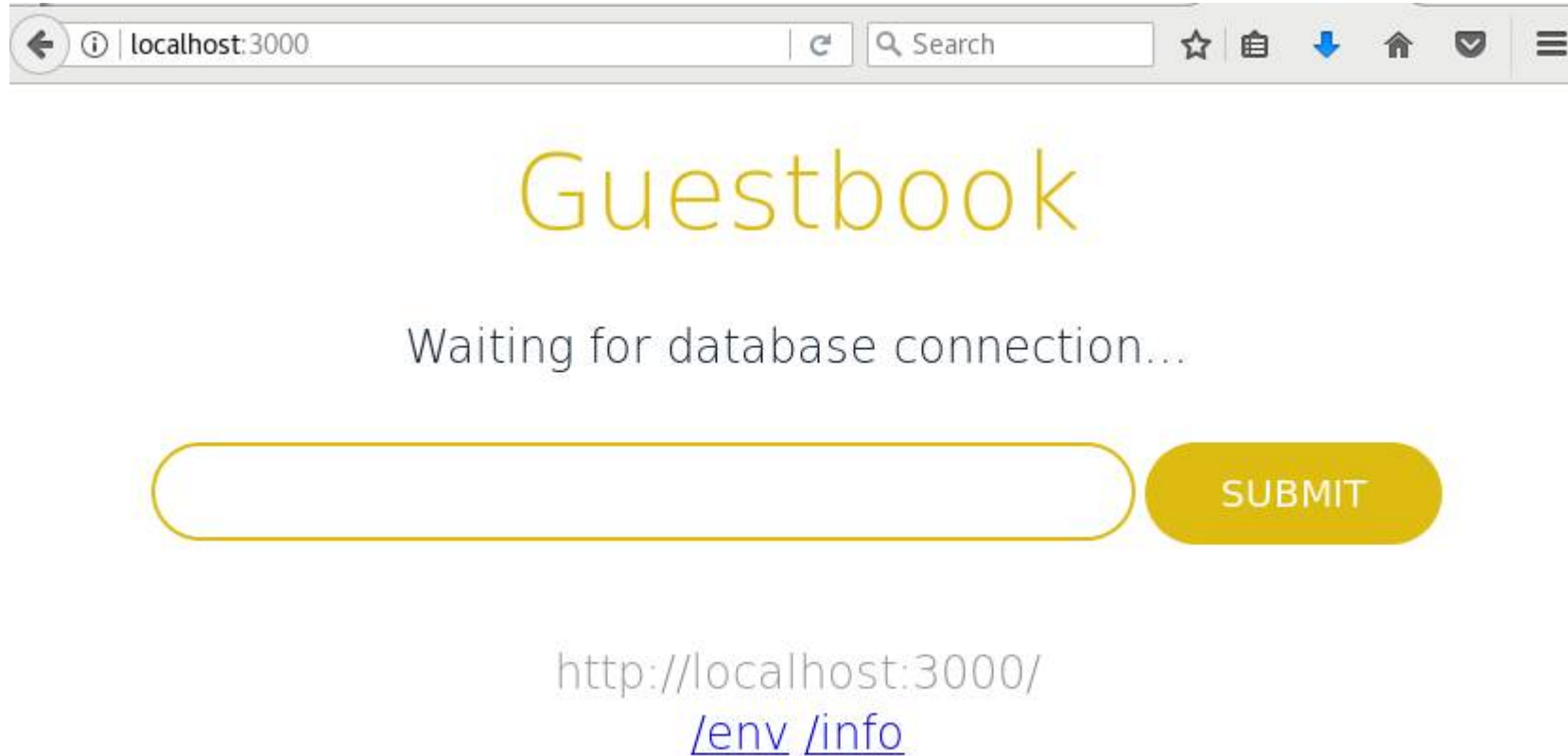
Comprobamos el historial de actualizaciones y desplegamos una nueva versión:

```
kubectl rollout history deployment/mediawiki  
kubectl set image deployment/mediawiki mediawiki=mediawiki:1.27 --all --record  
kubectl get all
```

Ahora vamos a desplegar una versión que da un error (versión no existe). ¿Podemos volver al despliegue anterior?

```
kubectl rollout history deployment/mediawiki  
kubectl set image deployment/mediawiki mediawiki=mediawiki:2 --all --record  
kubectl rollout undo deployment/mediawiki  
kubectl get all
```

EJEMPLO 6: Desplegando la aplicación GuestBook (Parte 1)



A screenshot of a web browser window. The address bar shows 'localhost:3000'. The page title is 'Guestbook'. Below the title, it says 'Waiting for database connection...'. There is a large, empty, rounded rectangular input field with a yellow border. To the right of the input field is a yellow button with the text 'SUBMIT'. Below the input field, the URL 'http://localhost:3000/' is displayed, followed by two links: '/env' and '/info'.

Guestbook

Waiting for database connection...

SUBMIT

<http://localhost:3000/>
[/env](#) [/info](#)

<https://github.com/kubernetes/examples/tree/master/guestbook>

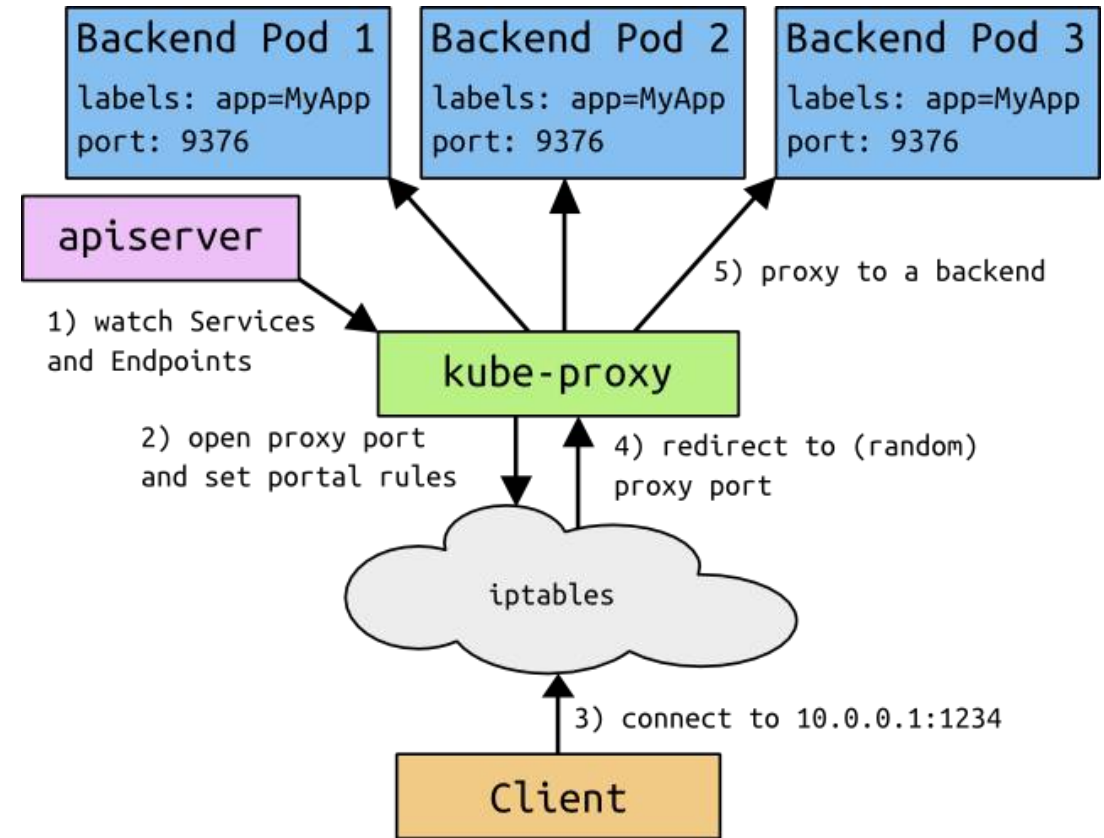
Services

Los servicios ([services](#)) nos permiten acceder a nuestra aplicaciones.

- Un servicio es una abstracción que define un conjunto de pods que implementan un micro-servicio. (Por ejemplo el *servicio frontend*).
- Ofrecen una dirección virtual (CLUSTER-IP) y un nombre que identifica al conjunto de pods que representa, al cual nos podemos conectar.
- La conexión al servicio se puede realizar desde otros pods o desde el exterior (mediante la generación aleatoria de un puerto).

Services

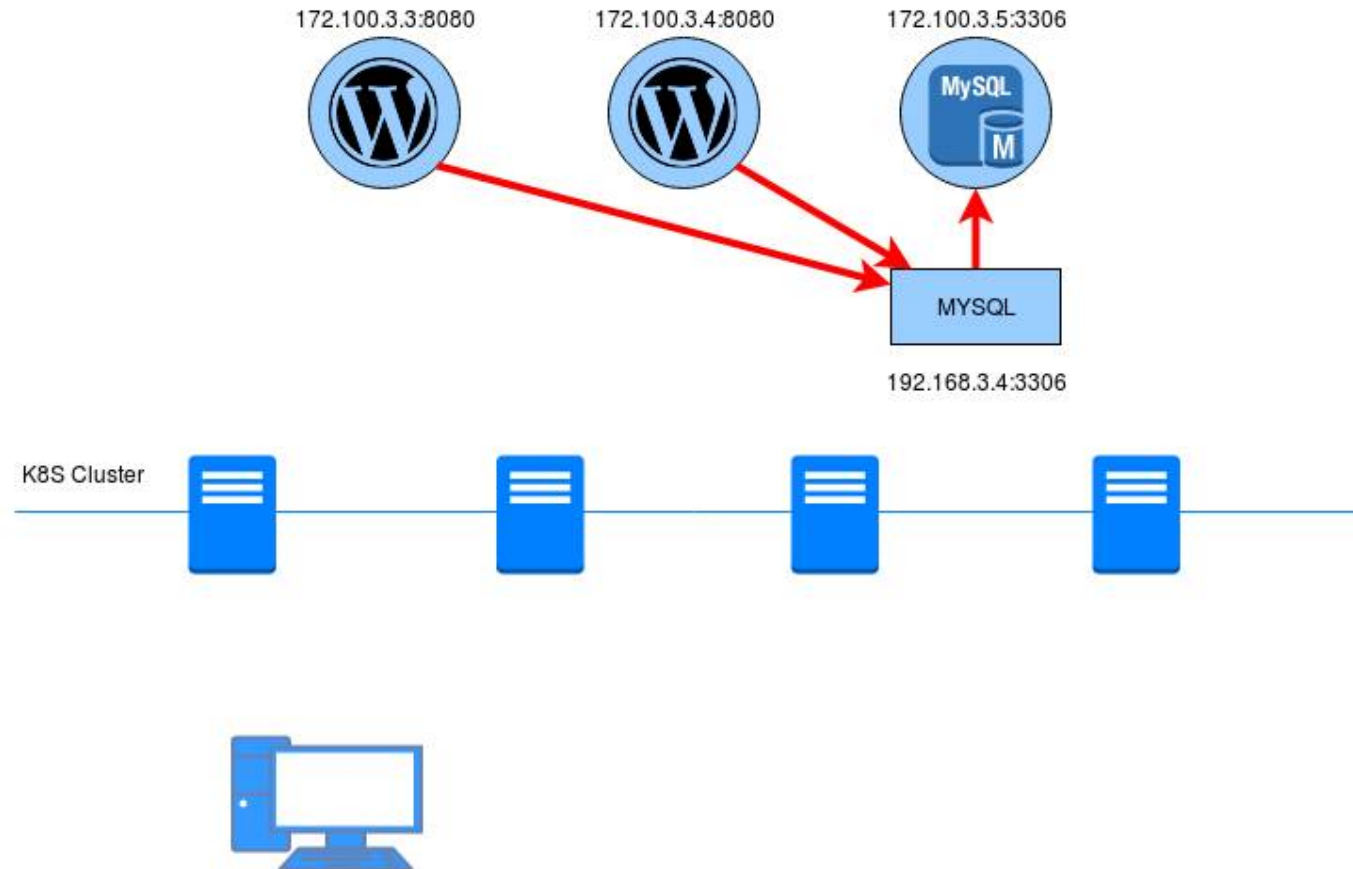
- Los servicios se implementan con iptables.
- El componente kube-proxy de Kubernetes se comunica con el servidor de API para comprobar si se han creado nuevos servicios.
- Cuando se crea un nuevo servicio, se le asigna una nueva ip interna virtual (IP-CLUSTER) que permite conexiones desde otros pods.
- Además podemos habilitar el acceso desde el exterior, se abre un puerto aleatorio que permite que accediendo a la IP del cluster y a ese puerto se acceda al conjunto de pods.
- Si tenemos más de un pod el acceso se hará siguiendo una política round-robin.



Services ClusterIP

- **ClusterIP:** Solo permite el acceso interno entre distintos servicios. Es el tipo por defecto. Podemos acceder desde el exterior con la instrucción `kubectl proxy`, puede de ser gran ayuda para los desarrolladores.

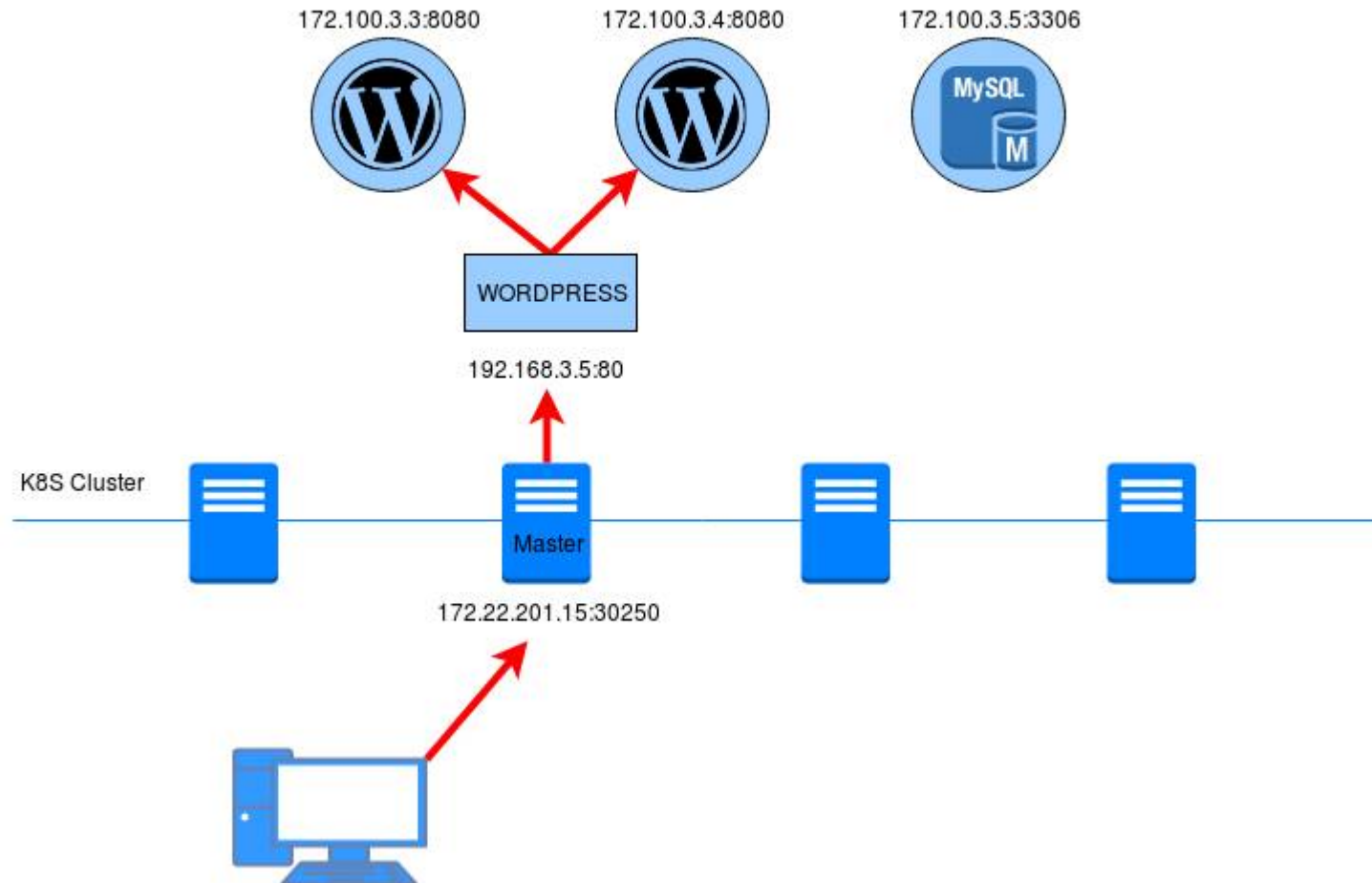
Services - ClusterIP



Services NodePort

- **NodePort:** Abre un puerto, para que el servicio sea accesible desde el exterior. Por defecto el puerto generado está en el rango de 30000:40000. Para acceder usamos la ip del servidor master del cluster y el puerto asignado.

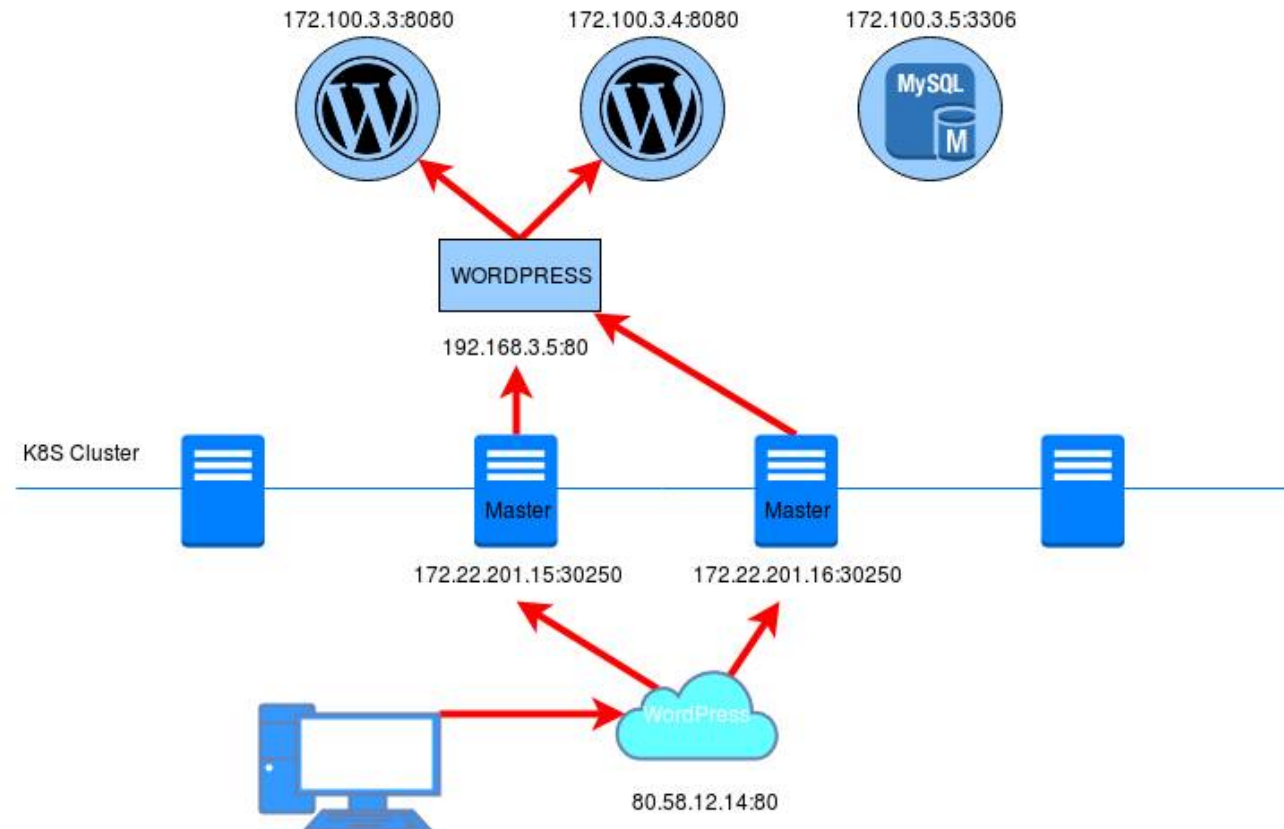
Services - NodePort



Services LoadBalancer

- **LoadBalancer:** Este tipo sólo está soportado en servicios de cloud público (GKE, AKS o AWS). El proveedor asignará un recurso de balanceo de carga para el acceso a los servicios. si usamos un cloud privado, como OpenSatck necesitaremos un plugin para configurar el funcionamiento.

Services - LoadBalancer



EJEMPLO 7: Describiendo objetos k8s: Services ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
spec:
  type: ClusterIP
  ports:
    - name: http
      port: 80
      targetPort: http
  selector:
    app: nginx
```

```
kubectl create -f deployment.yaml
```

```
kubectl create -f service_ci.yaml
```

También podríamos haber creado el servicio sin usar el fichero yaml, de la siguiente manera:

```
kubectl expose deployment/nginx --port=80 --type=ClusterIP
```

Podemos ver el servicio que hemos creado:

```
kubectl get svc
```

Puede ser bueno acceder desde exterior, por ejemplo en la fase de desarrollo de una aplicación para probarla:

```
kubectl proxy
```

Y accedemos a la URL:

```
http://localhost:8001/api/v1/namespaces/<NAMESPACE>/
services/<SERVICE NAME>:<PORT NAME>/proxy/
```

EJEMPLO 7: Describiendo objetos k8s: Services NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  namespace: default
spec:
  type: NodePort
  ports:
    - name: http
      port: 80
      targetPort: http
  selector:
    app: nginx
```

```
kubectl create -f service_np.yaml
```

También podríamos haber creado el servicio sin usar el fichero yaml, de la siguiente manera:

```
kubectl expose deployment/nginx --port=80 --type=NodePort
```

Podemos ver el servicio que hemos creado:

```
kubectl get svc
```

Desde el exterior accedemos a:

```
http://<IP_MASTER>:<PUERTO_ASIGNADO>
```

Para más información acerca de los Services puedes leer:
la [documentación de la API](#) y la [guía de usuario](#).

EJEMPLO 8: Desplegando la aplicación GuestBook (Parte 2)



Guestbook

SUBMIT

<http://172.22.201.15:30355/>
[/env](#) [/info](#)