Kubernetes

minikube

KUBERNETES

Este curso tiene como **objetivo** establecer las características que deben tener los sistemas que ofrecen las aplicaciones web para que cumplan con los requisitos esperados.

KUBERNETES

Kubernetes es un software de orquestación de contenedores desarrollado inicialmente por Google, pero que hoy en día es un proyecto libre independiente utilizado en gran cantidad de entornos diferentes y que se ha convertido en muchos casos en la solución preferida para orquestar aplicaciones basadas en contenedores en entornos en producción.

Necesidades a la que nos enfrentamos:

- **Escalabilidad**
- **Tolerancia a fallos**
- **Actualizaciones**

KUBERNETES

- **Actualizaciones** se pueden hacer en momentos en los que el uso es mínimo y, si es necesario una interrupción del servicio, se puede programar para un momento determinado en que tenga muy poco impacto. Este tipo de aplicaciones no se suelen modificar habitualmente.
- Es fácil determinar los **recursos** necesarios para que la aplicación funcione de forma adecuada, porque ni el uso de la aplicación se dispara en unos instantes, ni el número de empleados de una empresa varía de forma abrupta.

Es muy difícil determinar los **recursos** necesarios para prestar servicios a una demanda muy variable e idealmente, el servicio no puede interrumpirse nunca lo que dificulta las **actualizaciones**. Su uso va a variar en el tiempo con picos muy elevados en cuanto a **escalabilidad**.

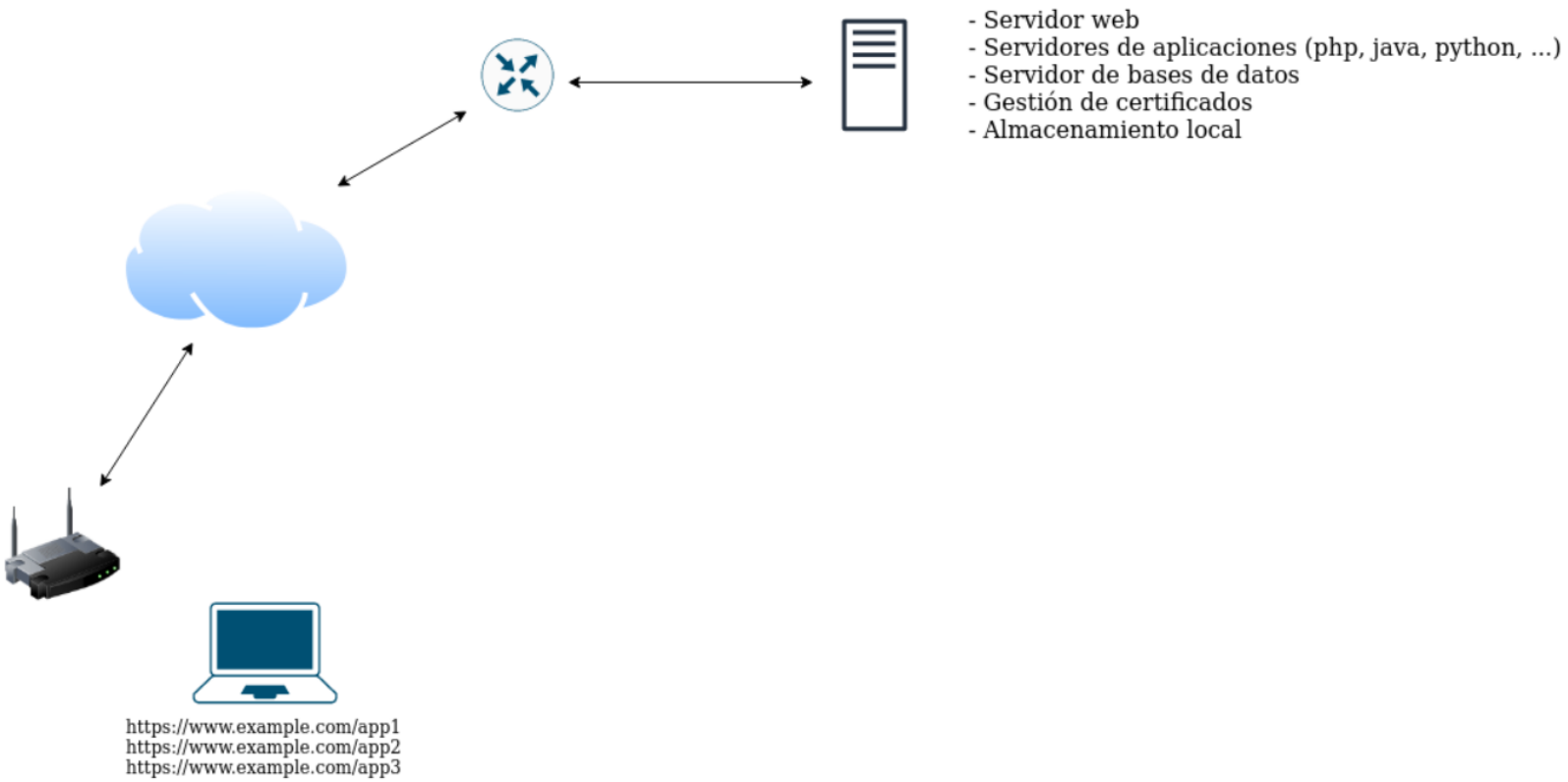
KUBERNETES

KUBERNETES

Evolucionamos pasito a pasito…

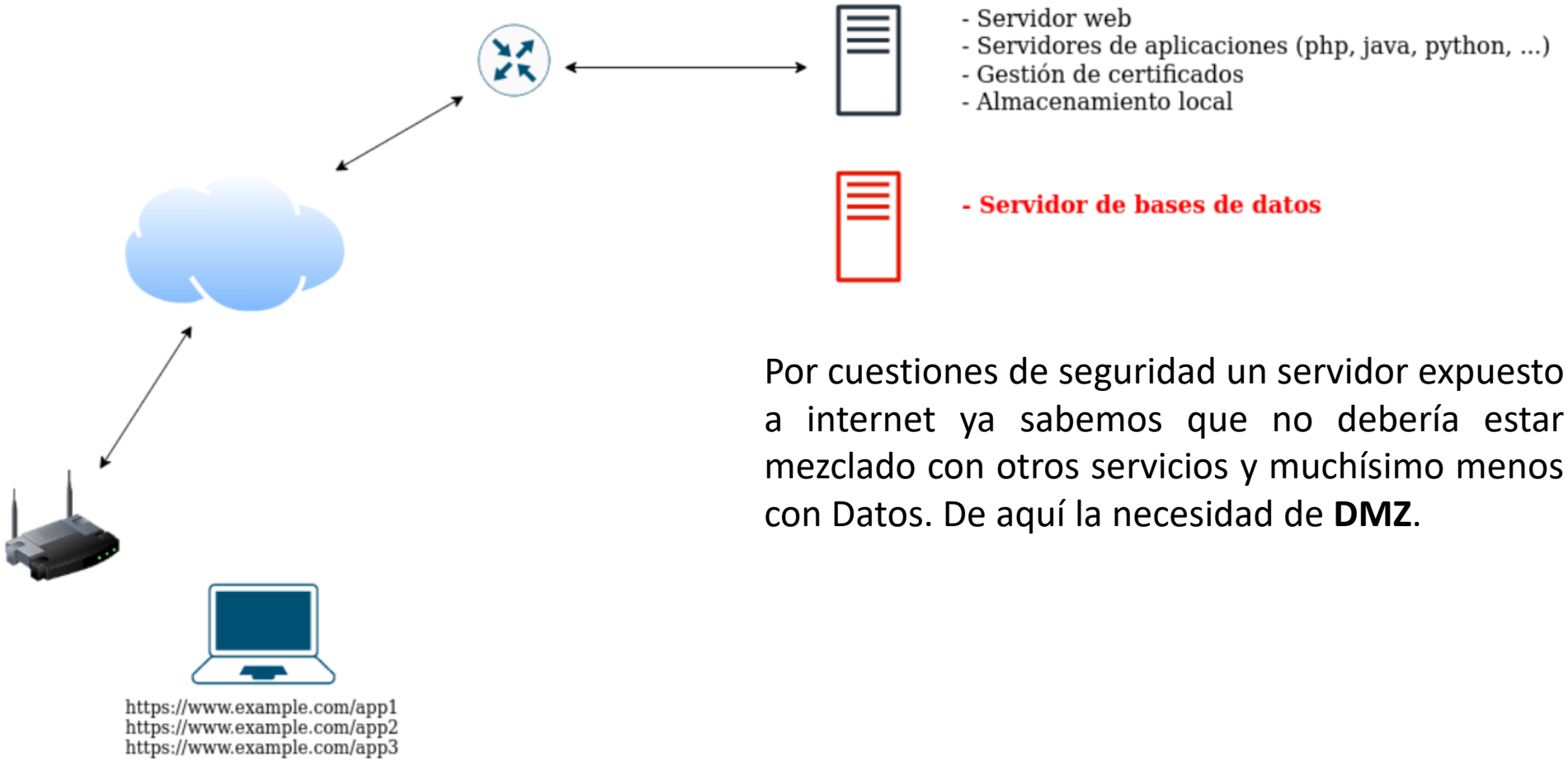
KUBERNETES

Punto de partida



KUBERNETES

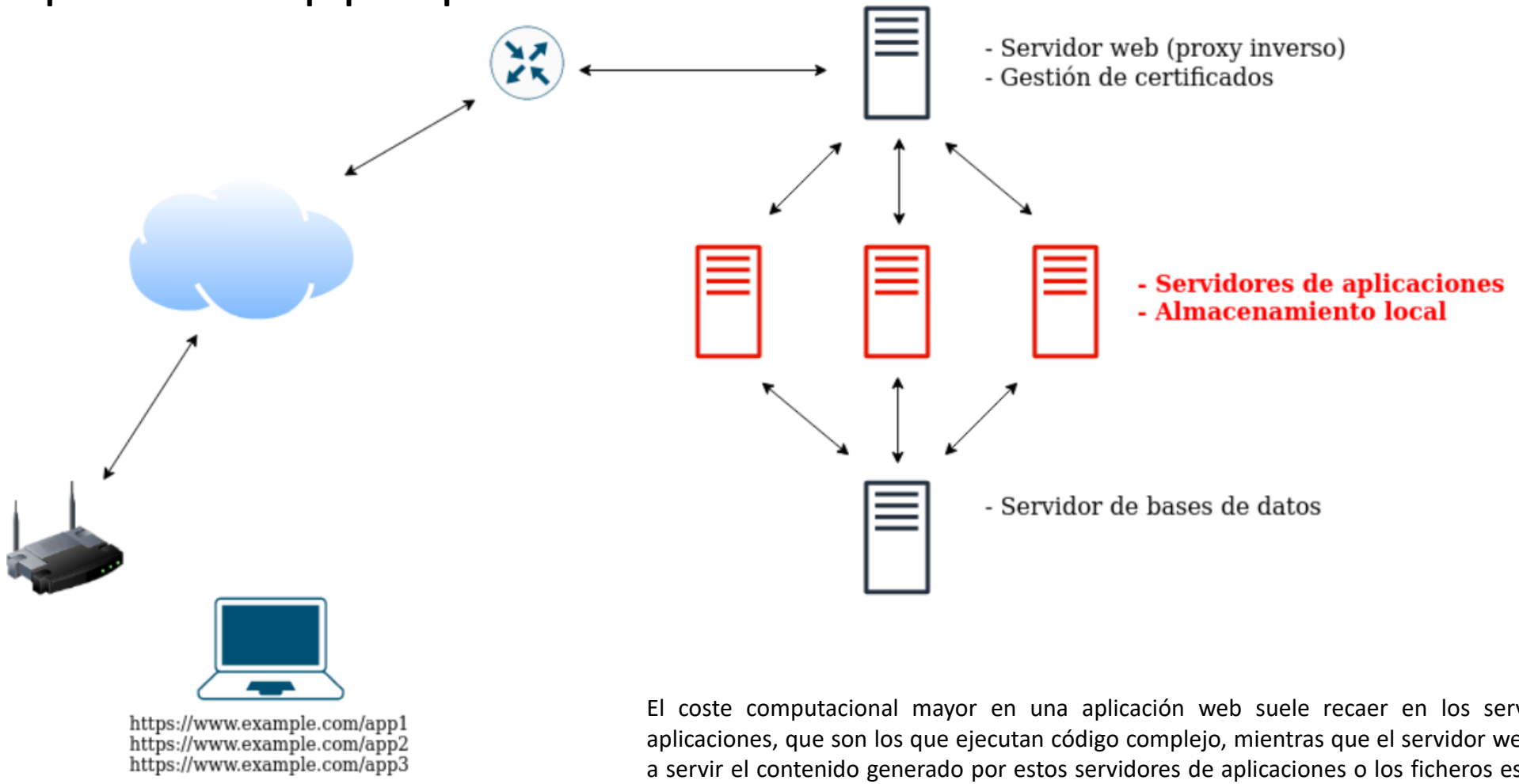
Servidor de bases de datos separado



Por cuestiones de seguridad un servidor expuesto a internet ya sabemos que no debería estar mezclado con otros servicios y muchísimo menos con Datos. De aquí la necesidad de **DMZ**.

KUBERNETES

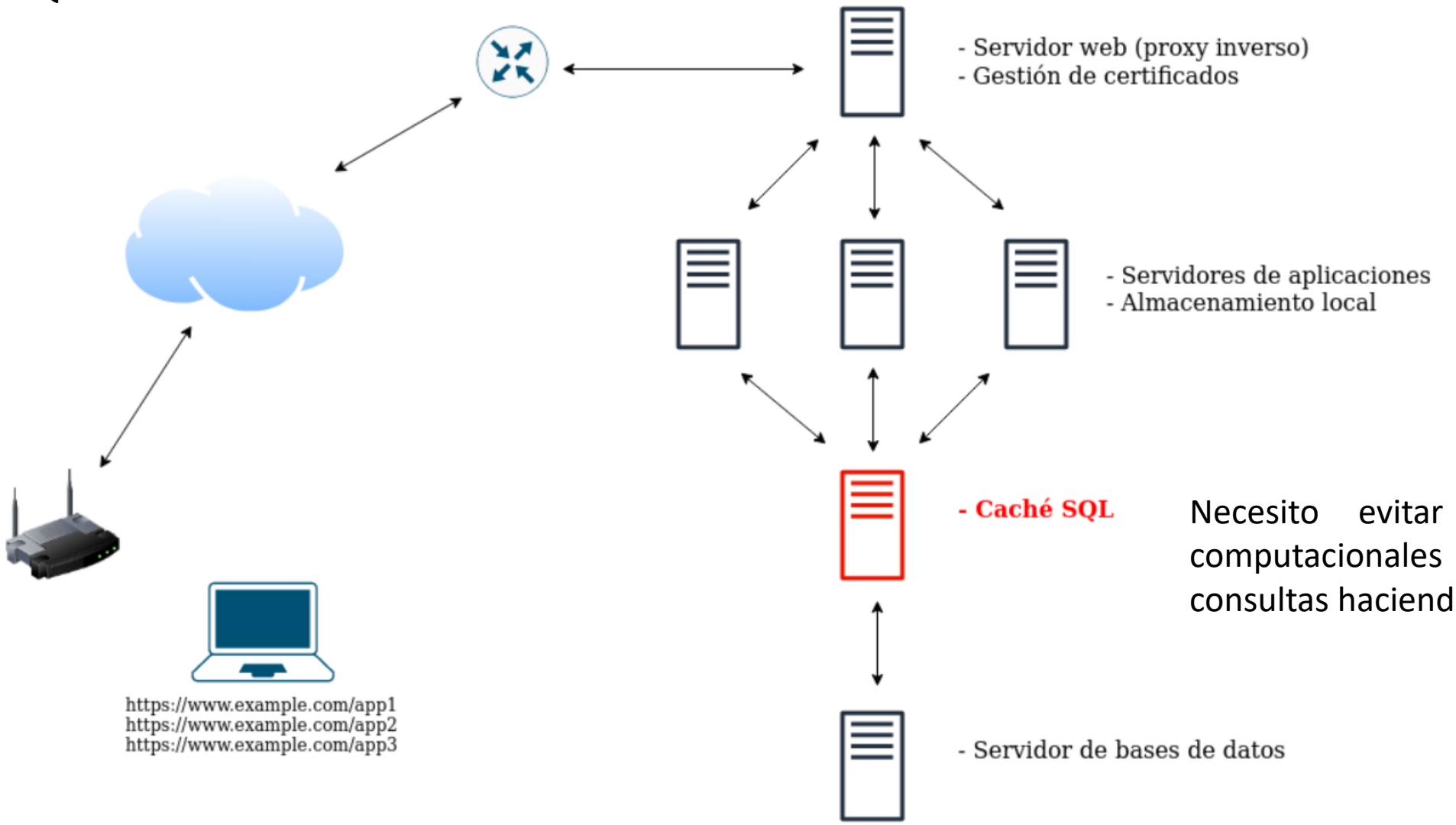
Servidores de aplicaciones en equipos separados



El coste computacional mayor en una aplicación web suele recaer en los servidores de aplicaciones, que son los que ejecutan código complejo, mientras que el servidor web se limita a servir el contenido generado por estos servidores de aplicaciones o los ficheros estáticos del sitio web. Al servir tres aplicaciones web diferentes desde el mismo equipo, podemos tener importantes interacciones entre ellas y que un aumento de uso de una aplicación, repercuta negativamente en las otras.

KUBERNETES

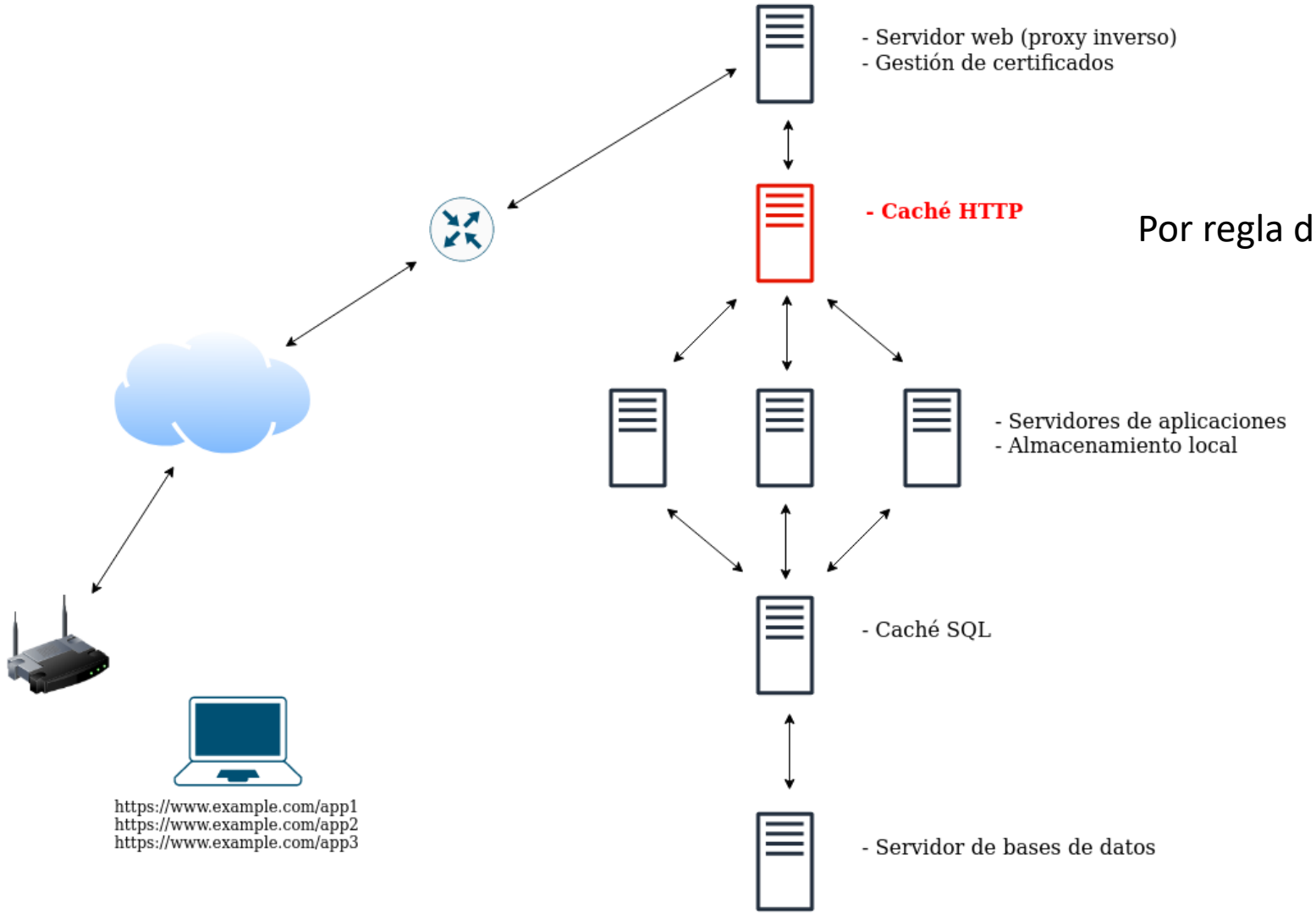
Caché SQL



Necesito evitar los costes computacionales de las consultas haciendo un cacheo.

KUBERNETES

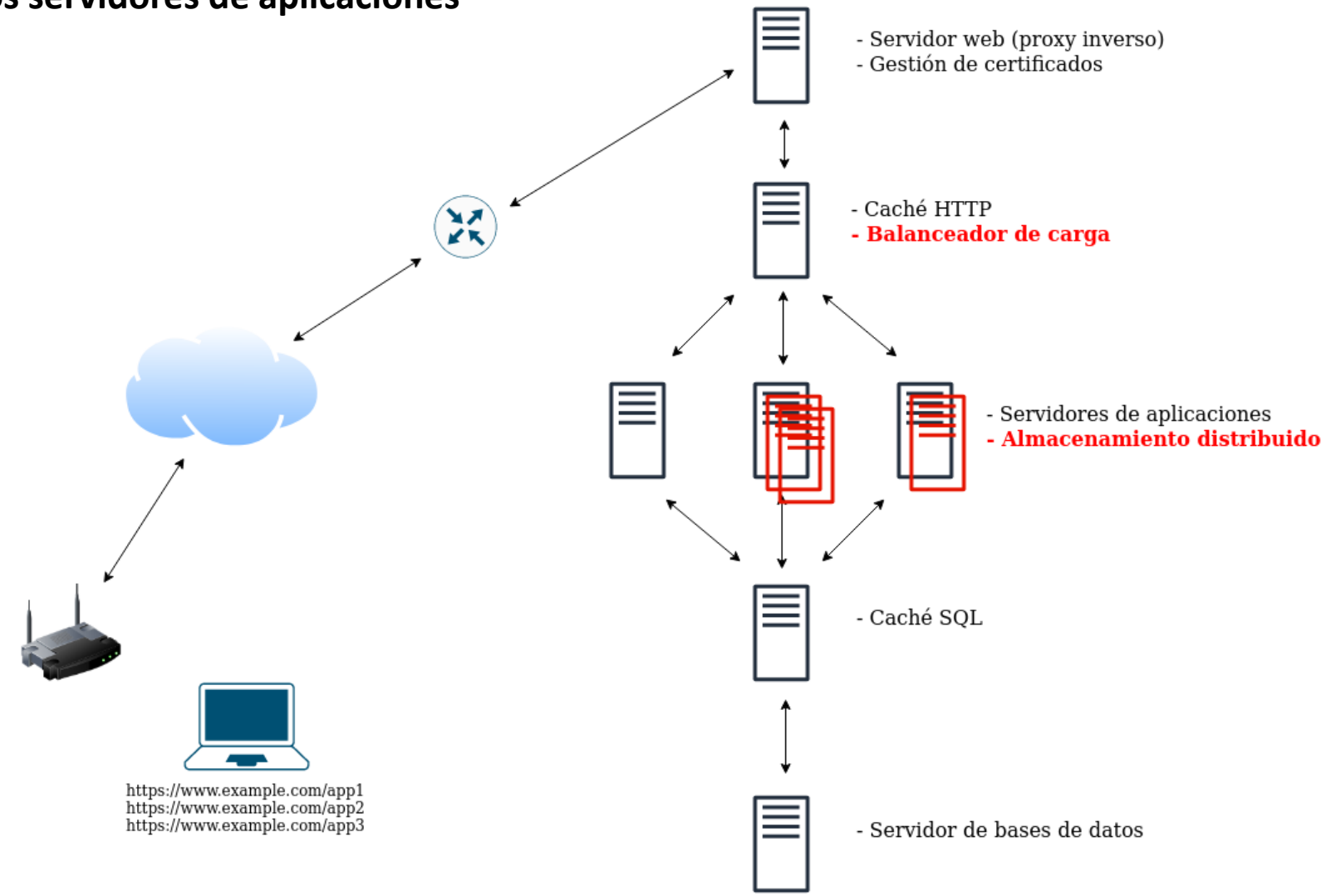
Caché HTTP



Por regla de tres.

KUBERNETES

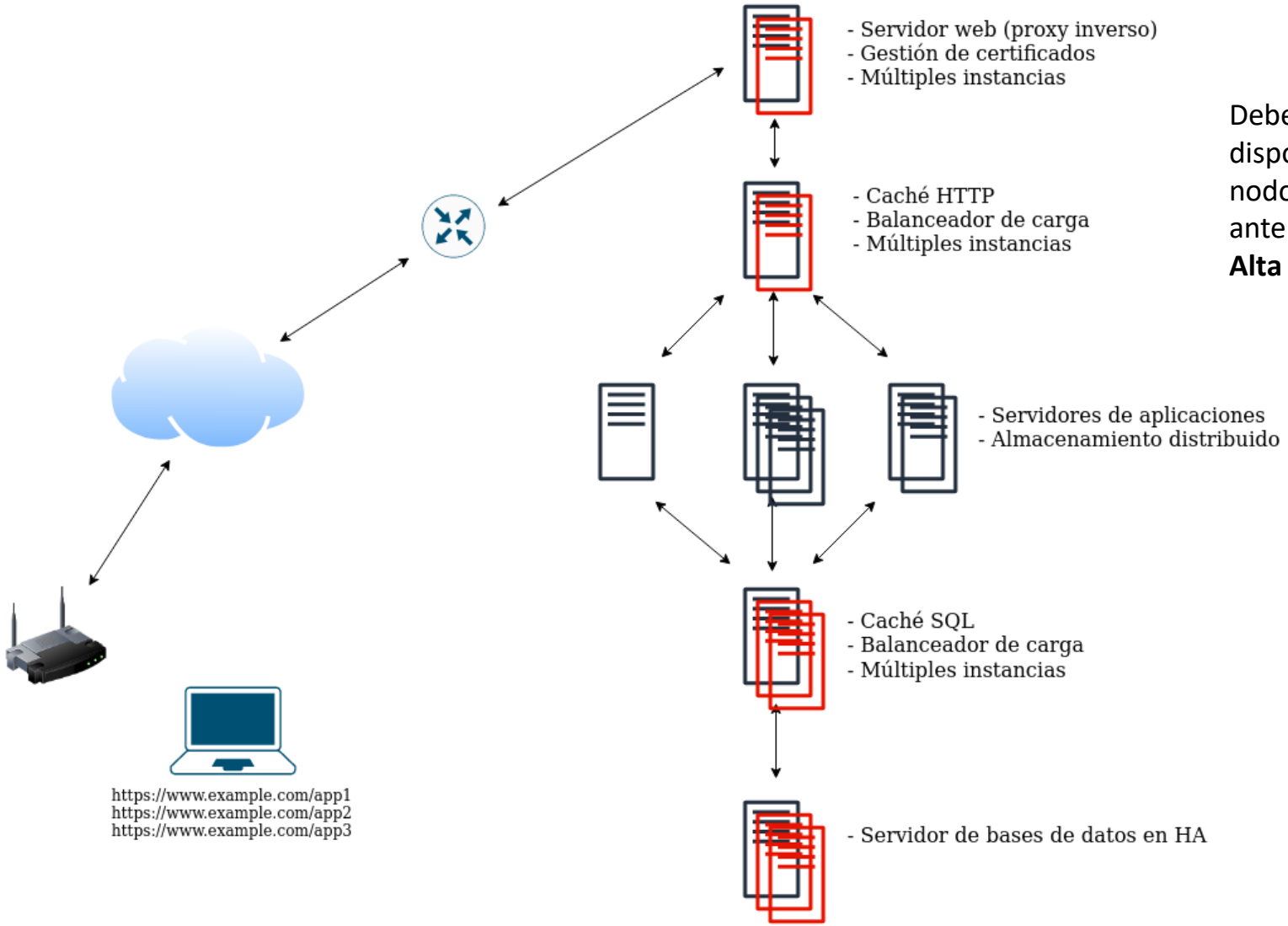
Varios servidores de aplicaciones



Si la demanda de alguna de las aplicaciones varía de forma importante, se puede utilizar escalado horizontal, por lo que el almacenamiento entre los servidores de aplicación de la misma aplicación tiene que estar distribuido de forma que garantice el uso concurrente y se deben repartir las peticiones a los diferentes servidores de aplicación a través de un balanceador de carga.

KUBERNETES

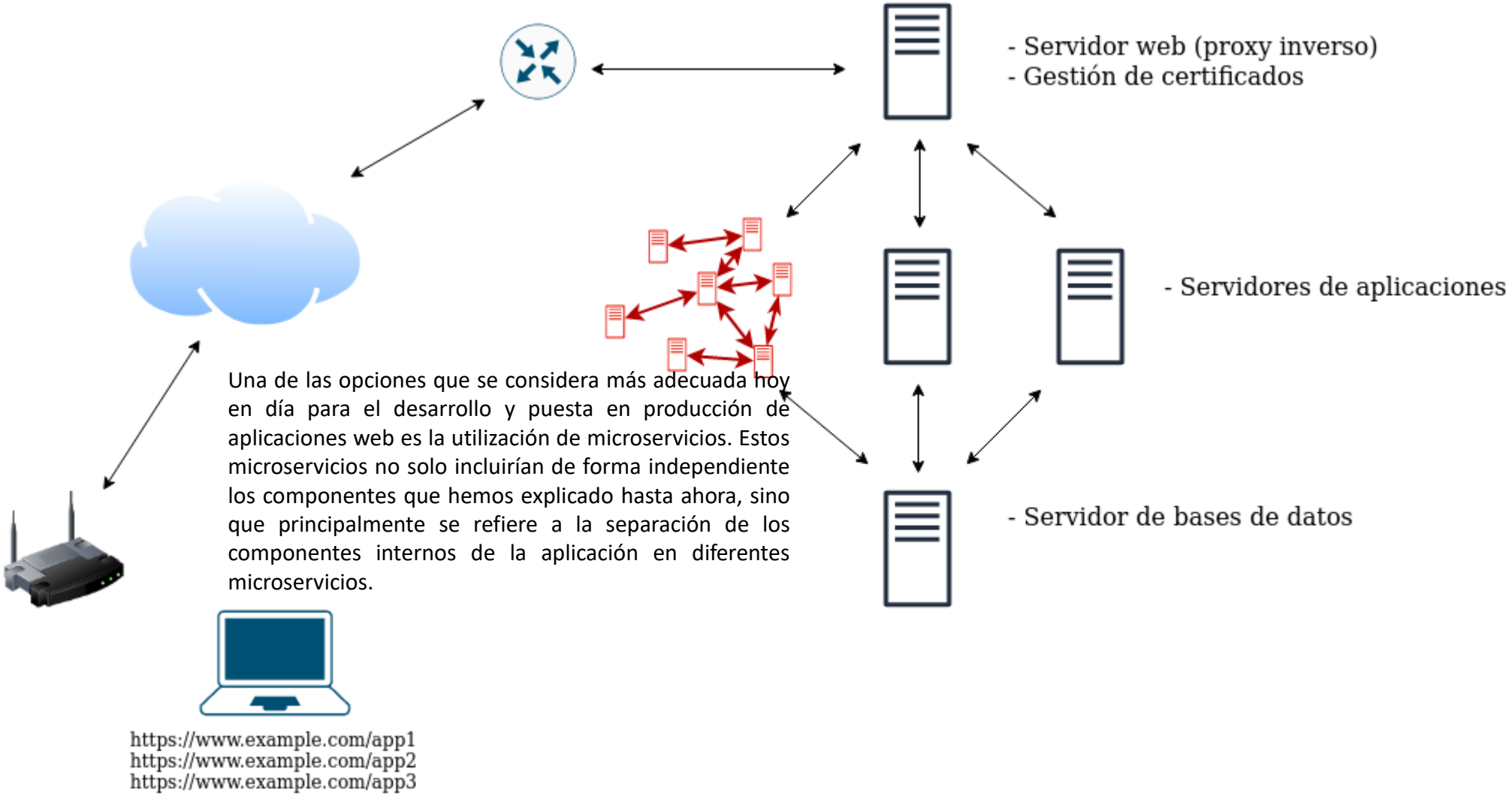
Alta disponibilidad en el resto de componentes



Debemos crear una arquitectura en la que la disponibilidad nunca dependa de uno solo nodo y el sistema pueda responder siempre ante incidencias puntuales en cualquier nivel: **Alta disponibilidad**

KUBERNETES

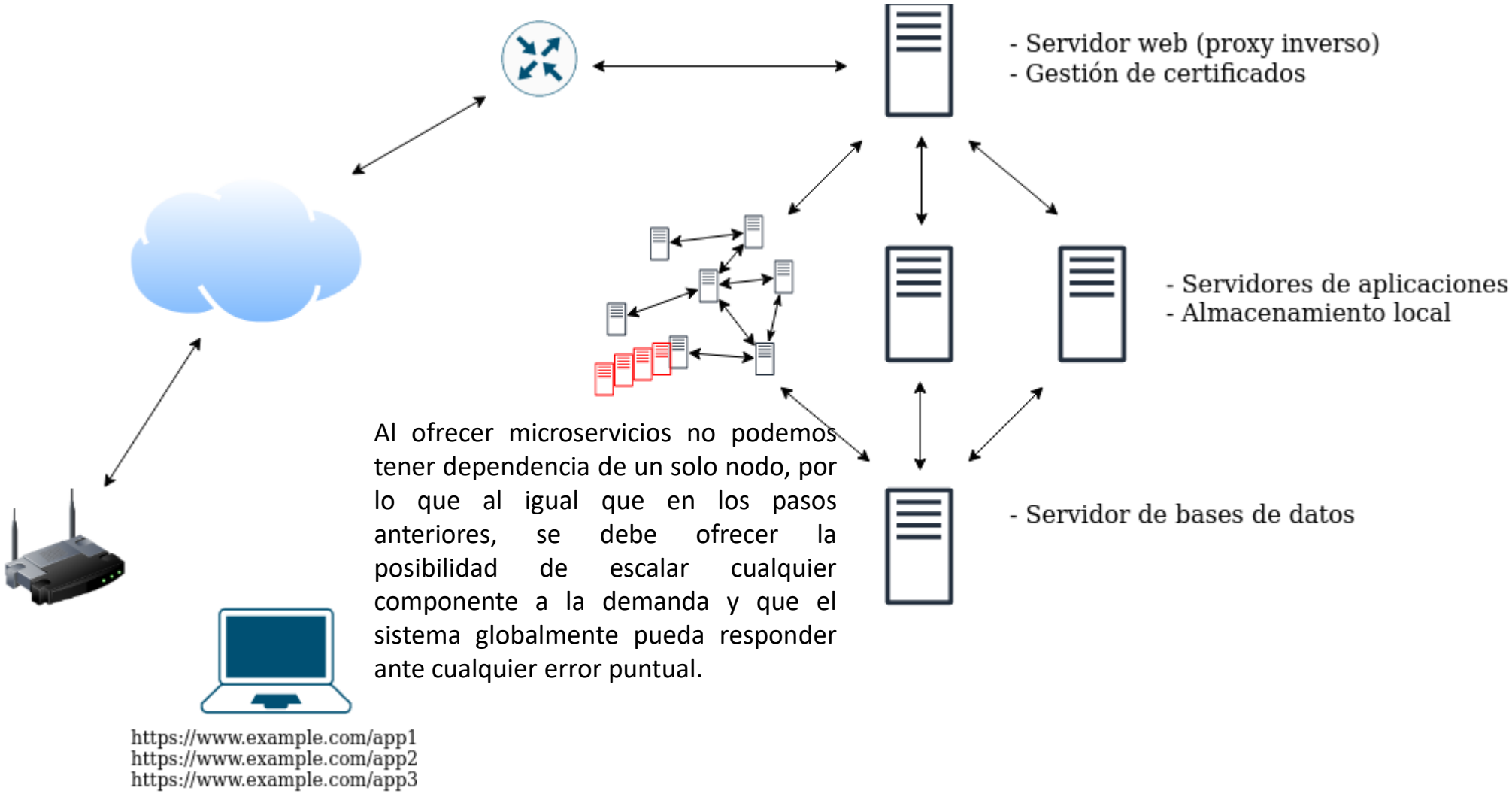
Microservicios y aplicaciones "tradicionales"



Una de las opciones que se considera más adecuada hoy en día para el desarrollo y puesta en producción de aplicaciones web es la utilización de microservicios. Estos microservicios no solo incluirían de forma independiente los componentes que hemos explicado hasta ahora, sino que principalmente se refiere a la separación de los componentes internos de la aplicación en diferentes microservicios.

KUBERNETES

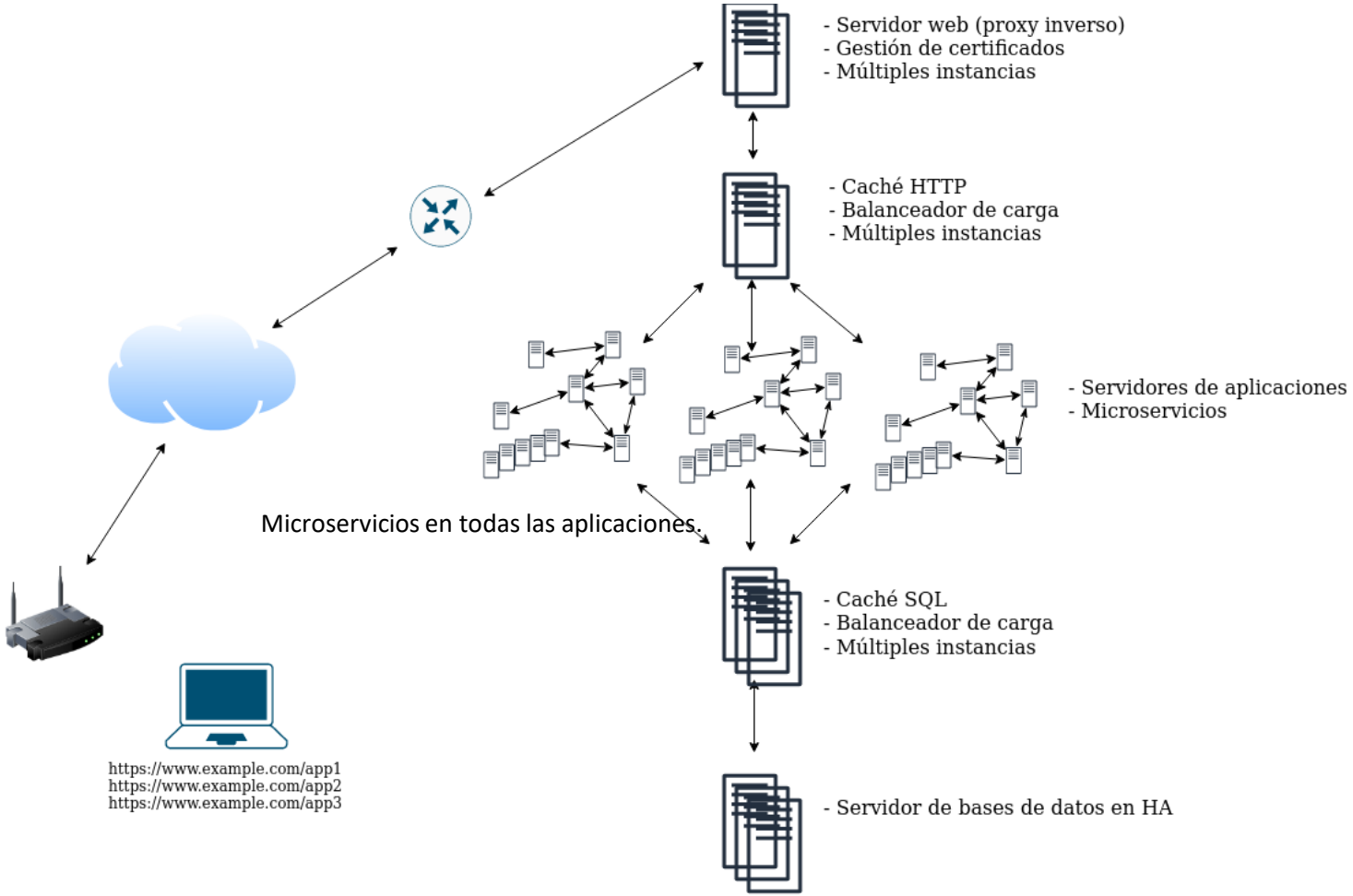
Escalabilidad en los microservicios



Al ofrecer microservicios no podemos tener dependencia de un solo nodo, por lo que al igual que en los pasos anteriores, se debe ofrecer la posibilidad de escalar cualquier componente a la demanda y que el sistema globalmente pueda responder ante cualquier error puntual.

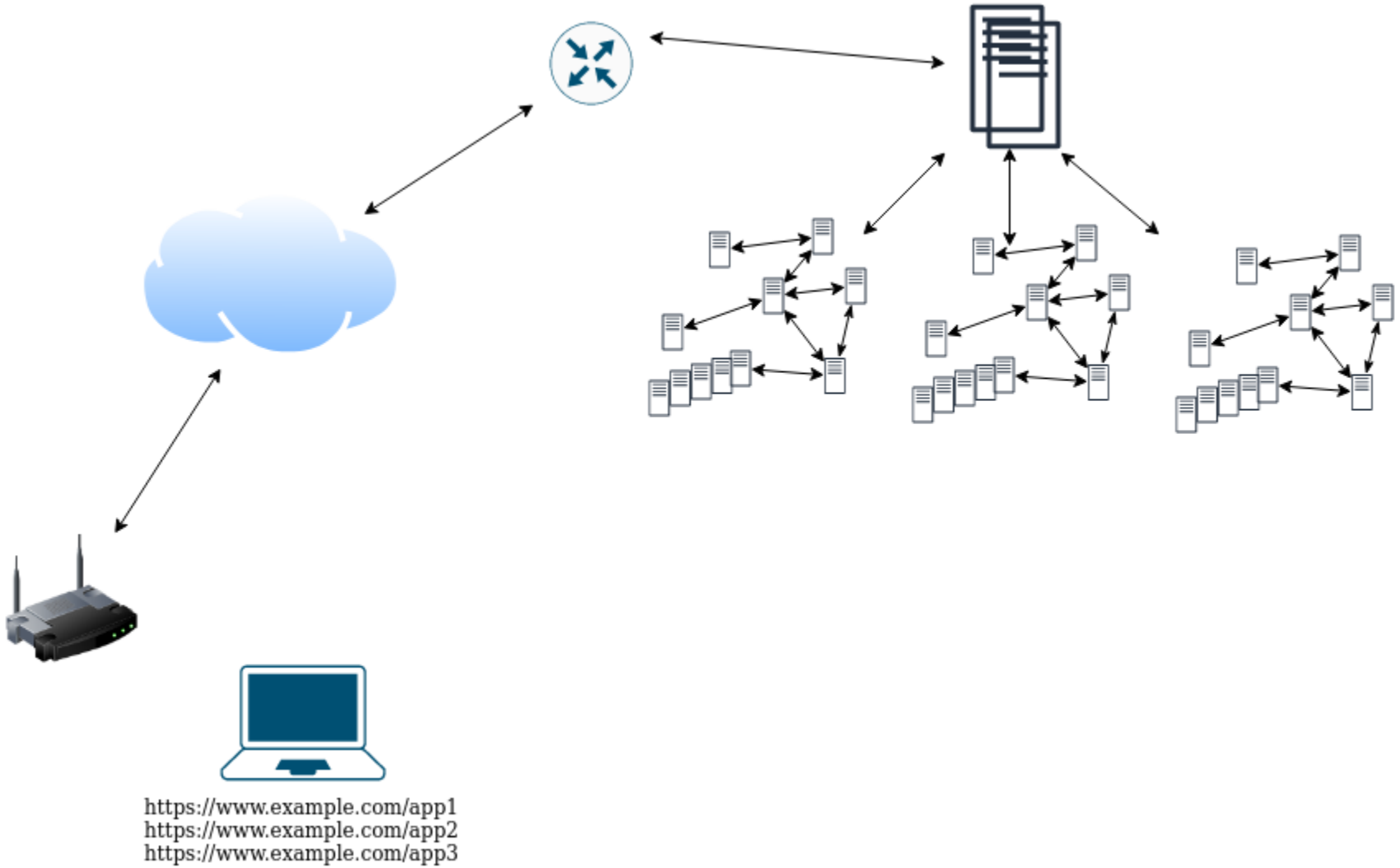
KUBERNETES

Microservicios en todas las aplicaciones



KUBERNETES

Todo en microservicios



- Microservicios

Todo definido internamente en microservicios.

KUBERNETES

KUBERNETES

Nodos

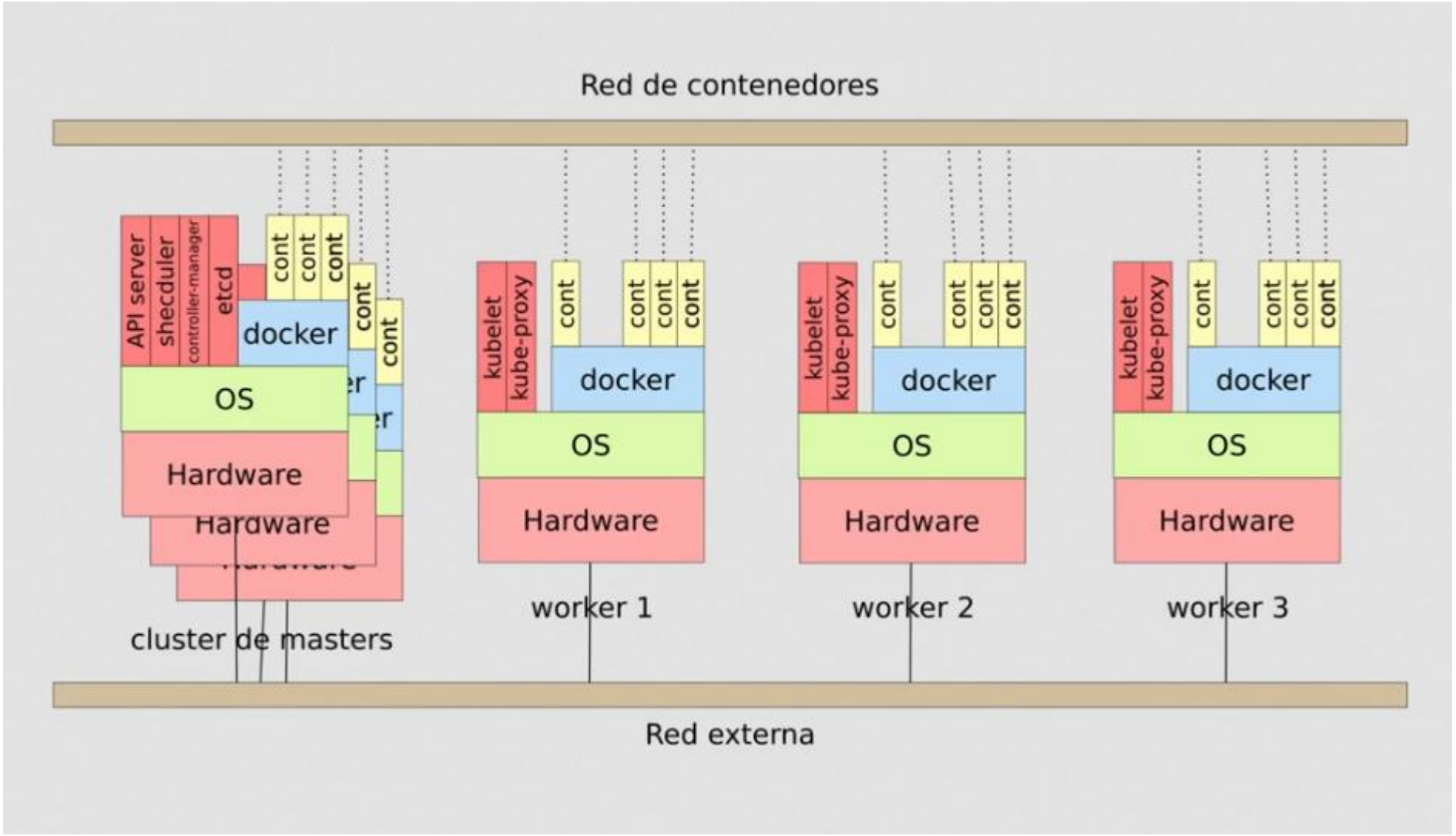
k8s es un software que se instala en varios nodos que se gestionan de forma coordinada, es decir, un clúster de nodos. Aunque es posible en casos muy peculiares instalar algunos nodos sobre sistemas Windows, la situación normal es que se trate de un cluster de nodos linux. No es necesario que todos los nodos tengan la misma versión y ni siquiera que sean la misma distribución, aunque en muchos casos sí lo sea por simplicidad en el despliegue y mantenimiento.

Los nodos del clúster pueden ser máquinas físicas o virtuales, pero quizás lo más habitual es que se traten de instancias de nube de infraestructura, es decir, máquinas virtuales ejecutándose en algún proveedor de IaaS (AWS, GCP, OpenStack, etc.). ***Serverless***

Se distingue entre dos tipos de nodos:

- Los nodos ***master***: Son los que ejecutan los servicios principales de k8s y ordenan a los otros nodos los contenedores que deben ejecutar. Como el uso del término master es últimamente muy controvertido en los paises de habla inglesa, se está cambiando su denominación por *control plane* node.
- Los nodos ***worker***: Son los que reciben las órdenes de los controladores y en los que se ejecutan los contenedores de las aplicaciones.

KUBERNETES


Red de contenedores
API server
shecduler
controller-manager
etcd
cont
cont
cont
cont
cont
docker
OS
Hardware
Hardware
cluster de masters
kubelet
kube-proxy
cont
cont
cont
cont
docker
OS
Hardware
worker 1
kubelet
kube-proxy
cont
cont
cont
cont
docker
OS
Hardware
worker 2
kubelet
kube-proxy
cont
cont
cont
cont
docker
OS
Hardware
worker 3
Red externa

KUBERNETES

Instalación

minikube

Minikube permite desplegar localmente un "cluster" de Kubernetes con un solo nodo. Minikube es un proyecto oficial de Kubernetes y es probablemente la solución más adecuada para aprender a usar k8s, ya que es un proyecto maduro y muy sencillo de instalar. Los requisitos mínimos para instalar minikube en nuestro equipo son:

- 2 CPUs
- 2GiB de memoria
- 20GiB de espacio libre en disco
- Un sistema de virtualización o de contenedores instalado:
  - Docker
  - Hyperkit
  - Hyper-V
  - KVM
  - Parallels
  - Podman
  - VirtualBox
  - VMWare

https://minikube.sigs.k8s.io/docs/drivers/.

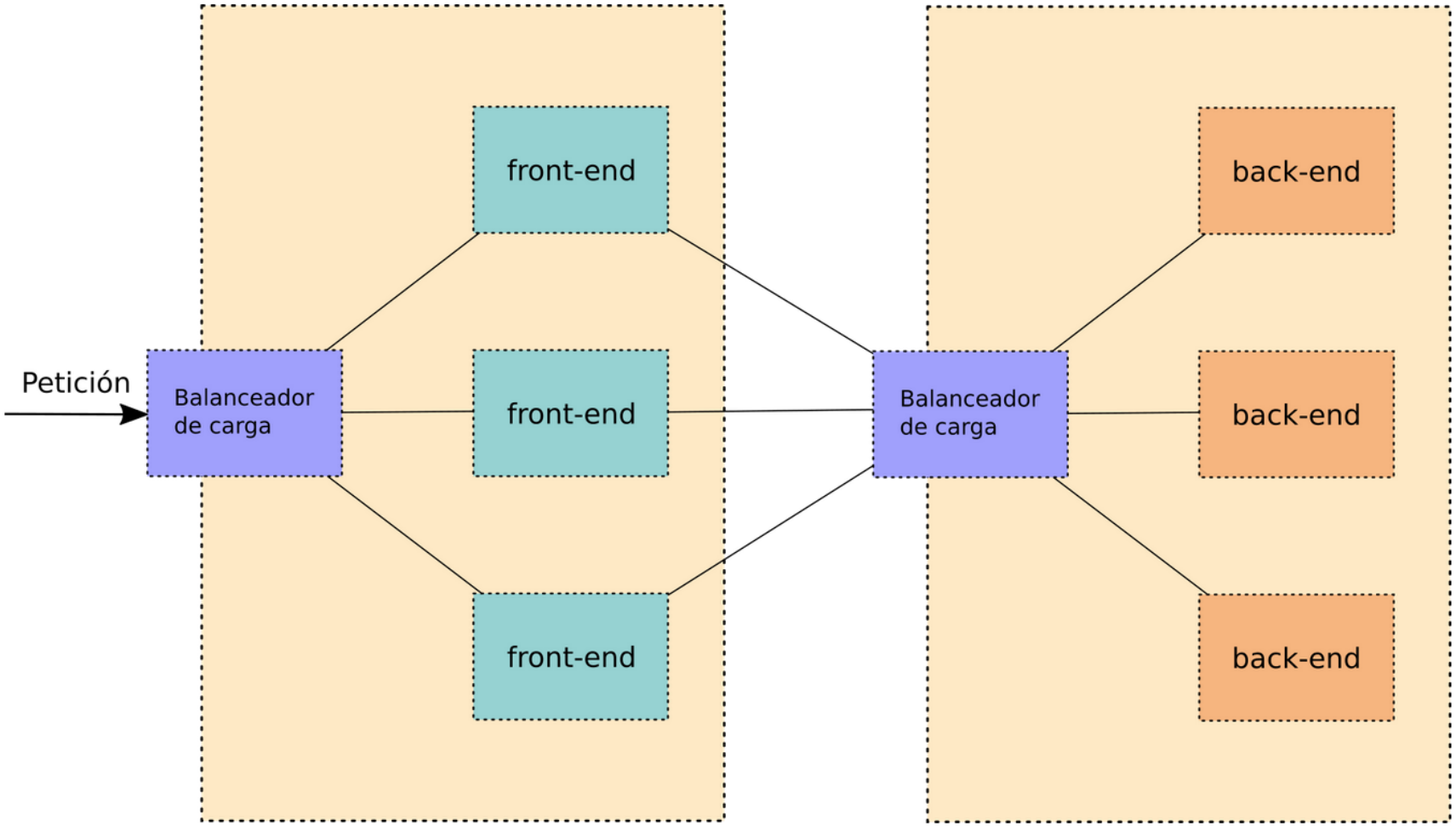
KUBERNETES

Tarea

1. Instala minikube.
2. ¿Qué cliente vamos a utilizar para gestionar minikube?. Instalar.

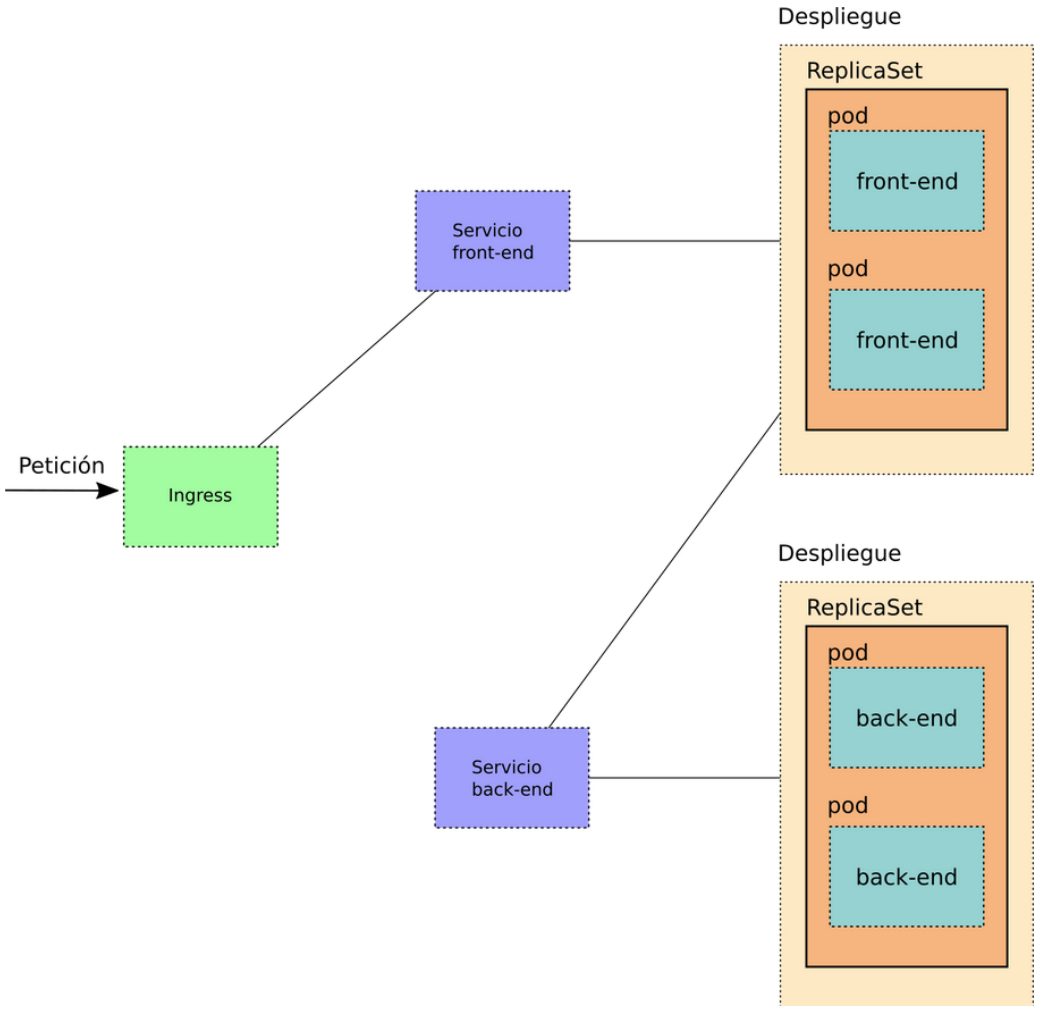
KUBERNETES

Modelo tradicional



KUBERNETES

Modelo Kubernetes



KUBERNETES

Elementos Kubernetes

- Pods: ejecutan los contenedores
- ReplicaSets:
  - Se encargan de que no haya caída del servicio
  - Gestionan la tolerancia a fallos
  - Proporcionan escalabilidad dinámica
- Deployments:
  - Gestionan las actualizaciones continuas
  - Realizan despliegues automáticos
- Services:
  - Gestionan el acceso a los pods
  - Balancean la carga entre los Pods disponibles
- Ingress:
  - Gestionan el acceso desde el exterior a través de nombre

KUBERNETES

PODS

KUBERNETES

Pod es una envoltura que contiene uno o varios contenedores (en la mayoría de los casos un solo contenedor). De forma genérica, un Pod representa un conjunto de contenedores que comparten almacenamiento y una única IP.

- En la mayoría de los casos y siguiendo el principio de un proceso por contenedor, evitamos tener sistemas (como máquinas virtuales) ejecutando docenas de procesos, por lo que lo más habitual será tener un Pod en cuyo interior se define un contenedor que ejecuta un solo proceso.

Algunos ejemplos pueden ser: ejecución en modo demonio de un servidor web, ejecución de un servidor de aplicaciones Java, ejecución de una tarea programada, ejecución en modo demonio de un servidor DNS, etc.

- En determinadas circunstancias será necesario ejecutar más de un proceso en el mismo "sistema", como en los casos de procesos fuertemente acoplados, en esos casos, tendremos más de un contenedor dentro del Pod. Cada uno de los contenedores ejecutando un solo proceso, pero pudiendo compartir almacenamiento y una misma dirección IP como si se tratase de un sistema ejecutando múltiples procesos.

Un ejemplo típico de un Pod multicontenedor es un servidor web nginx con un servidor de aplicaciones PHP-FPM, que se implementaría mediante un solo Pod, pero ejecutando un proceso de nginx en un contenedor y otro proceso de php-fpm en otro contenedor.

KUBERNETES

Para crear un Pod podemos hacerlo:

- De forma **imperativa**: mediante `kubectl run pod-nginx --image=nginx`

Un Pod tiene otros muchos parámetros asociados, que en este caso quedarán sin definir o Kubernetes asumirá los valores por defecto. Sin embargo es mucho más habitual trabajar con los objetos de Kubernetes de manera declarativa.

- De forma **declarativa**: definiendo los objetos de forma detallada a través de un fichero en formato YAML.

KUBERNETES

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

• **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 el recurso:

  • **name**: Nombre del pod

  • **Labels**: nos permiten etiquetar los recursos de Kubernetes (por ejemplo un pod) con información del tipo clave/valor.

• **spec**: Definimos las características del recurso. En el caso de un Pod indicamos los contenedores que van a formar el Pod (sección containers), en este caso sólo uno.

  • **image**: La imagen desde la que se va a crear el contenedor

  • **name**: Nombre del contenedor.

  • **imagePullPolicy**: Las imágenes se guardan en un registro interno. Se pueden utilizar registros públicos (google o docker hub son los más usados) 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 desde el repositorio externo, tendremos que indicar imagePullPolicy:Always.

KUBERNETES

Crear directamente el Pod desde el fichero yaml:	kubectl create -f pod.yaml
Ver el estado en el que se encuentra y si está o no listo:	kubectl get pods
Información sobre los Pods, como por ejemplo, saber en qué nodo del cluster se está ejecutando, ip:	kubectl get pod -o wide
Información más detallada del Pod (equivalente al inspect de docker):	kubectl describe pod pod-nginx
Editar el Pod y ver todos los atributos que definen el objeto, la mayoría de ellos con valores asignados automáticamente por el propio Kubernetes y podremos actualizar ciertos valores:	kubectl edit pod pod-nginx
Obtener directamente los logs al igual que se hace en docker:	kubectl logs pod-nginx
Ejecuta alguna orden adicional en el Pod, podemos utilizar el comando exec, por ejemplo, en el caso particular de que queremos abrir una shell de forma interactiva:	kubectl exec -it pod-nginx -- /bin/bash
Acceder a la aplicación, redirigiendo un puerto de localhost al puerto de la aplicación:	kubectl port-forward pod-nginx 8080:80 Y accedemos al servidor web en la url http://localhost:8080.

KUBERNETES

Vamos a crear nuestro primer Pod, y para ellos vamos a desplegar una imagen que nos ofrece un servidor web con una página estática. Para ello realiza los siguientes pasos:

1.Crea un fichero yaml con la descripción del recurso Pod, teniendo en cuenta los siguientes aspectos:
   1. Indica nombres distintos para el Pod y para el contenedor.
   2. La imagen que debes desplegar es iesgn/test_web:latest.
   3. Indica una etiqueta en la descripción del Pod.

2.Crea el Pod.

3.Comprueba que el Pod se ha creado y está corriendo.

4.Obtén información detallada del Pod creado.

5.Accede de forma interactiva al Pod y comprueba los ficheros que están en el DocumentRoot (usr/local/apache2/htdocs/).

6.Crea una redirección con kubectl port-forward utilizando el puerto de localhost 8888 y sabiendo que el Pod ofrece el servicio en el puerto 80. Accede a la aplicación desde un navegador.

7.Muestra los logs del Pod y comprueba que se visualizan los logs de los accesos que hemos realizado en el punto anterior.

8.Elimina el Pod, y comprueba que ha sido eliminado.

KUBERNETES

REPLICASET

KUBERNETES

ReplicaSet controla un conjunto de Pods y es el responsable de que estos Pods siempre estén ejecutándose (**Tolerancia a fallos**) y de aumentar o disminuir las réplicas de dicho Pod (**Escalabilidad dinámica**).

KUBERNETES

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: replicaset-nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx
          name: contenedor-nginx
```

Algunos de los parámetros definidos ya lo hemos estudiado en la definición del Pod. Los nuevos parámetros de este recurso son los siguientes:

- **replicas**: Indicamos el número de Pods que siempre se deben estar ejecutando.
- **selector**: Seleccionamos los Pods que va a controlar el ReplicaSet por medio de las etiquetas. Es decir este ReplicaSet controla los Pods cuya etiqueta app es igual a nginx.
- **template**: El recurso ReplicaSet contiene la definición de un Pod. Fíjate que el Pod que hemos definido en la sección template tiene indicado la etiqueta necesaria para que sea seleccionado por el ReplicaSet (app: nginx).

KUBERNETES

Crear directamente el Replicaset desde el fichero yaml:	kubectl apply -f rs.yaml
Ver los recursos que se han creado:	kubectl get rs,pods
Si queremos obtener información detallada del recurso ReplicaSet que hemos creado:	kubectl describe rs nombre
Para escalar el número de pods: (Puede aumentarse o disminuirse y puede hacerse vía comandos o en el archivo yaml)	kubectl scale rs nombre --replicas=5
Si borramos un ReplicaSet se borrarán todos los Pods asociados:	kubectl delete rs nombre O kubectl delete -f rs.yaml

KUBERNETES

Como indicamos en el contenido de este módulo, no se va a trabajar directamente con los Pods (realmente tampoco vamos a trabajar directamente con los ReplicaSet, en el siguiente módulo explicaremos los Deployments que serán el recurso con el que trabajaremos). En este ejercicio vamos a crear un ReplicaSet que va a controlar un conjunto de Pods. Para ello, realiza los siguientes pasos:

1. Crea un fichero yaml con la descripción del recurso ReplicaSet, teniendo en cuenta los siguientes aspectos:
    1. Indica nombres distintos para el ReplicaSet y para el contenedor de los Pods que va a controlar.
    2. El ReplicaSet va a crear 3 réplicas.
    3. La imagen que debes desplegar es iesgn/test_web:latest.
    4. Indica de manera adecuada una etiqueta en la especificación del Pod que vas a definir que coincida con el selector del ReplicaSet.
2. Crea el ReplicaSet.
3. Comprueba que se ha creado el ReplicaSet y los 3 Pods.
4. Obtén información detallada del ReplicaSet creado.
5. Vamos a probar la tolerancia a fallos: Elimina uno de los 3 Pods, y comprueba que inmediatamente se ha vuelto a crear un nuevo Pod.
6. Vamos a comprobar la escalabilidad: escala el ReplicaSet para tener 6 Pods de la aplicación.
7. Elimina el ReplicaSet y comprueba que se han borrado todos los Pods.

KUBERNETES

DEPLOYMENT

KUBERNETES

Cuando queramos desplegar una aplicación en Kubernetes no vamos a gestionar pods ni replicaset directamente, sino que vamos a crear un recurso de tipo Deployment. ¿Qué ocurre cuando creamos un nuevo recurso Deployment?

- La creación de un Deployment conlleva la creación de un ReplicaSet que controlará un conjunto de Pods creados a partir de la versión de la imagen que se ha indicado.
- Si hemos desarrollado una nueva versión de la aplicación y hemos creado una nueva imagen con la nueva versión, podemos modificar el Deployment indicando la **nueva versión** de la imagen. En ese momento se creará un nuevo ReplicaSet que controlará un nuevo conjunto de Pods creados a partir de la nueva versión de la imagen (habremos desplegado una nueva versión de la aplicación).
- Por lo tanto podemos decir que un Deployment va guardando un historial con los ReplicaSet que se van creando al ir cambiado la versión de la imagen. El ReplicaSet que esté activo en un determinado momento será el responsable de crear los Pods con la versión actual de la aplicación.
- Si tenemos un historial de ReplicaSet según las distintas versiones de la imagen que estamos utilizando, podemos, de una manera sencilla, volver a una versión anterior de la aplicación (**Rollback**).

Por cada servicio o microservicio que necesite nuestra aplicación crearemos un Deployment para desplegarlo.

KUBERNETES

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: deployment-nginx
  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: contendor-nginx
        ports:
        - name: http
          containerPort: 80
```

La creación 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 (los parámetros replicas, selector y template). 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. Es decir, cuando modificamos la versión de la imagen del Deployment, se crea un ReplicaSet nuevo y ¿qué hacemos con los pods?:

- •Recreate: elimina los Pods antiguos y crea los nuevos; es la opción por defecto.
- •RollingUpdate: va creando los nuevos Pods, comprueba que funcionan y se eliminan los antiguos.

Además, hemos introducido un nuevo parámetro al definir el contenedor del pod: con el parámetro ports hemos indicado el puerto que expone el contenedor (containerPort) y le hemos asignado un nombre (name).

KUBERNETES

Crear directamente el deployment desde el fichero yaml:	kubectl apply -f deployment.yaml
Ver los recursos que se han creado:	kubectl get deploy,rs,pod
Si queremos obtener información detallada del recurso Deployment que hemos creado:	kubectl describe deployment nombre
Para escalar el número de deployment: (estamos escalando el ReplicaSet asociado en ese momento)	kubectl scale deployment nombre --replicas=4
Si queremos ver los logs generados en los Pods de un Deployment	kubectl logs deployment/nombre
Si eliminamos el Deployment se eliminarán el ReplicaSet asociado y los Pods que se estaban gestionando.	kubectl delete deployment nombre
Si queremos acceder a la aplicación, podemos utilizar la opción de port-forward sobre el despliegue (de nuevo recordamos que no es la forma adecuada	kubectl port-forward deployment/nombre puerto:puerto_pod

KUBERNETES

En esta actividad vamos a crear un Deployment de una aplicación web. Sigamos los siguientes pasos:

1.Crea un fichero yaml con la descripción del recurso Deployment, teniendo en cuenta los siguientes aspectos:

- •Indica nombres distintos para el Deployment y para el contenedor de los Pods que va a controlar.
- •El Deployment va a crear 2 réplicas.
- •La imagen que debes desplegar es iesgn/test_web:latest.
- •Indica de manera adecuada una etiqueta en la especificación del Pod que vas a definir que coincida con el *selector* del Deployment.

2.Crea el Deployment.

3.Comprueba los recursos que se han creado: Deployment, ReplicaSet y Pods.

4.Obtén información detallada del Deployment creado.

- •Crea un una redirección utilizando el port-forward para acceder a la aplicación, sabiendo que la aplicación ofrece el servicio en el puerto 80, y accede a la aplicación con un navegador web.

6.Accede a los logs del despliegue para comprobar el acceso que has hecho en el punto anterior.

7.Elimina el Deployment y comprueba que se han borrado todos los recursos creados.

KUBERNETES

El equipo de desarrollo ha creado una primera versión preliminar de una aplicación web y ha creado una imagen de contenedor con el siguiente nombre: iesgn/test_web:version1.
Vamos a desplegar esta primera versión de la aplicación, para ello:
1.Crea un fichero yaml (puedes usar el de la actividad anterior) para desplegar la imagen: iesgn/test_web:version1.
2.Crea el Deployment, recuerda la opción que nos permite registrar los comandos que vamos a ejecutar a continuación para ir actualizando el despliegue.
3.Crea una redirección utilizando el port-forward para acceder a la aplicación, sabiendo que la aplicación ofrece el servicio en el puerto 80, y accede a la aplicación con un navegador web.
Nuestro equipo de desarrollo ha seguido trabajando y ya tiene lista la versión 2 de nuestra aplicación, han creado una imagen que se llama: iesgn/test_web:version2. Vamos a actualizar nuestro despliegue con la nueva versión, para ello:
1.Realiza la actualización del despliegue utilizando la nueva imagen.
2.Comprueba los recursos que se han creado: Deployment, ReplicaSet y Pods.
3.Visualiza el historial de actualizaciones.
4.Crea una redirección utilizando el port-forward para acceder a la aplicación, sabiendo que la aplicación ofrece el servicio en el puerto 80, y accede a la aplicación con un navegador web.
Finalmente después de un trabajo muy duro, el equipo de desarrollo ha creado la imagen iesgn/test_web:version3 con la última versión de nuestra aplicación y la vamos a poner en producción, para ello:
1.Realiza la actualización del despliegue utilizando la nueva imagen.
2.Comprueba los recursos que se han creado: Deployment, ReplicaSet y Pods.
3.Visualiza el historial de actualizaciones.
4.Crea una redirección utilizando el port-forward para acceder a la aplicación, sabiendo que la aplicación ofrece el servicio en el puerto 80, y accede a la aplicación con un navegador web.
¡Vaya!, parece que esta versión tiene un fallo, y no se ve de forma adecuada la hoja de estilos, tenemos que volver a la versión anterior:
1.Ejecuta la instrucción que nos permite hacer un rollback de nuestro despliegue.
2.Comprueba los recursos que se han creado: Deployment, ReplicaSet y Pods.
3.Visualiza el historial de actualizaciones.
4.Crea una redirección utilizando el port-forward para acceder a la aplicación, sabiendo que la aplicación ofrece el servicio en el puerto 80, y accede a la aplicación con un navegador web.

KUBERNETES

En esta tarea vamos a desplegar una aplicación web que requiere de dos servicios para su ejecución. La aplicación se llama GuestBook y necesita los siguientes servicios:

•La aplicación Guestbook es una aplicación web desarrollada en python que es servida en el puerto 5000/tcp. Utilizaremos la imagen iesgn/guestbook.

•Esta aplicación guarda la información en una base de datos no relacional redis, que utiliza el puerto 6379/tcp para recibir las conexiones. Usaremos la imagen redis.

Por lo tanto si tenemos dos servicios distintos, tendremos dos ficheros yaml para crear dos recursos Deployment, uno para cada servicio. Con esta manera de trabajar podemos obtener las siguientes características:

1.Cada conjunto de Pods creado en cada despliegue ejecutarán un solo proceso para ofrecer el servicio.

2.Cada conjunto de Pods se puede escalar de manera independiente. Esto es importante, si identificamos que al acceder a alguno de los servicios se crea un cuello de botella, podemos escalarlo para tener más Pods ejecutando el servicio.

3.Las actualizaciones de los distintos servicios no interfieren en el resto.

4.Lo estudiaremos en un módulo posterior, pero podremos gestionar el almacenamiento de cada servicio de forma independiente.

Por lo tanto para desplegar la aplicaciones tendremos dos ficheros.yaml:

- guestbook-deployment.yaml
- redis-deployment.yaml

Para realizar el despliegue realiza los siguientes pasos:

1.Usando los ficheros anteriores crea los dos Deployments.

2.Comprueba que los recursos que se han creado: Deployment, ReplicaSet y Pods.

- Crea una redirección utilizando el port-forward para acceder a la aplicación, sabiendo que la aplicación ofrece el servicio en el puerto 5000, y accede a la aplicación con un navegador web.
- ¿Qué aparece en la página principal de la aplicación?. Aparece el siguiente mensaje: Waiting for database connection.... Por lo tanto podemos indicar varias conclusiones:
- Hasta ahora no estamos accediendo de forma "normal" a las aplicaciones. El uso de la opción port-forward es un mecanismo que realmente nos posibilita acceder a la aplicación, pero utilizando un proxy. Deberíamos acceder a las aplicaciones usando una ip y un puerto determinado.
- Parece que tampoco hay acceso entre los Pods de los distintos despliegues. Parece que los Pods de la aplicación guestbook no pueden acceder al Pod donde se está ejecutando la base de datos redis.

KUBERNETES

SERVICES

KUBERNETES

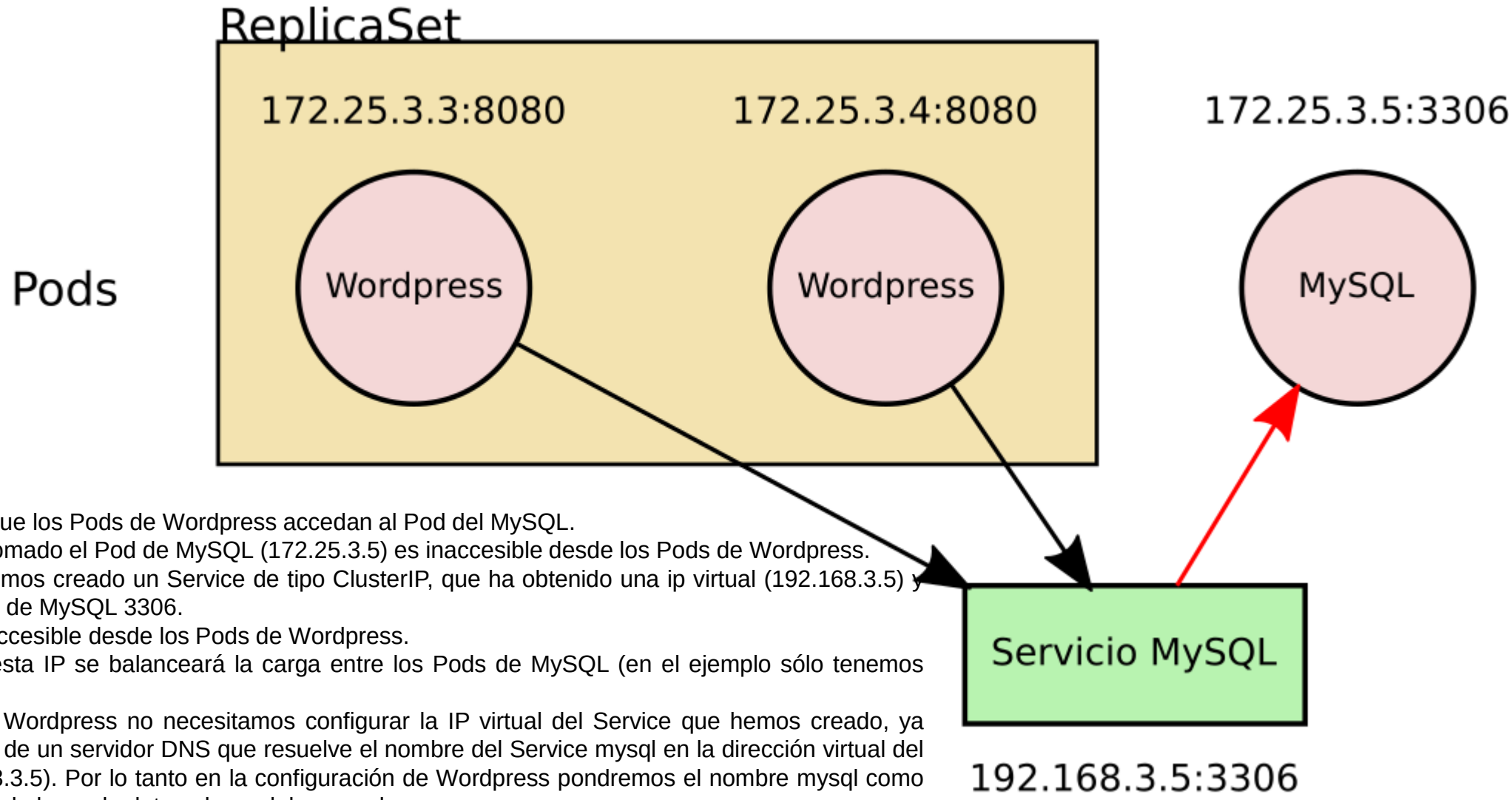
Los servicios (Services) nos permiten acceder a las aplicaciones que hemos desplegado en el cluster.

- Un Service es una abstracción que **nos permite acceder a un conjunto de pods** (que se han creado a partir de un Deployment) que implementan una aplicación (Por ejemplo: acceder a un servidor web, a una servidor de base de datos, a un servicio que forma parte de una aplicación, ...).

- A cada Pod se le asigna una IP a la que no se puede acceder directamente, por lo tanto necesitamos un Service que nos ofrece **una dirección virtual (CLUSTER-IP) y un nombre** que identifica al conjunto de Pods que representa, al cual nos podemos conectar.

KUBERNETES

ClusterIP



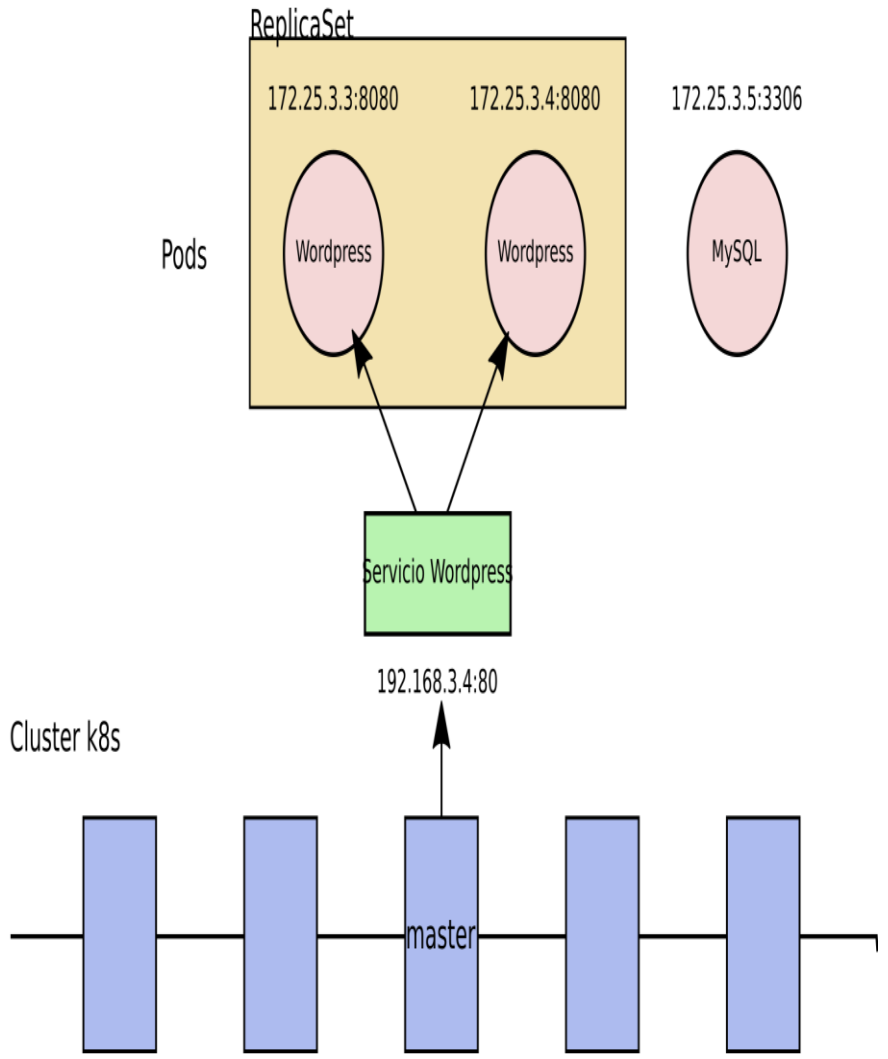
1.Necesitamos que los Pods de Wordpress accedan al Pod del MySQL.
2.La IP que ha tomado el Pod de MySQL (172.25.3.5) es inaccesible desde los Pods de Wordpress.
3.Por lo tanto hemos creado un Service de tipo ClusterIP, que ha obtenido una ip virtual (192.168.3.5) y expone el puerto de MySQL 3306.
4.Esta IP sí es accesible desde los Pods de Wordpress.
5.Al acceder a esta IP se balanceará la carga entre los Pods de MySQL (en el ejemplo sólo tenemos uno).
6.Además en el Wordpress no necesitamos configurar la IP virtual del Service que hemos creado, ya que disponemos de un servidor DNS que resuelve el nombre del Service mysql en la dirección virtual del Service (192.168.3.5). Por lo tanto en la configuración de Wordpress pondremos el nombre mysql como host del servidor de base de datos al que debe acceder.

KUBERNETES

NodePort

1. Necesitamos que los Pods de Wordpress sean accesibles desde el exterior, para que podamos acceder a la aplicación.
2. La IP que han tomado los Pods de Wordpress (172.25.3.3, ...) no son accesibles desde el exterior. Además comprobamos que estos Pods están ofreciendo el servicio en el puerto 8080
3. Por lo tanto, hemos creado un Service de tipo NodePort que ha obtenido una IP virtual (192.168.3.4) y expone el puerto 80.
4. Al acceder a esta IP al puerto 80 se balanceará la carga entre los Pods de Wordpress, accediendo a las IPs de los Pods de Wordpress al puerto 8080.
5. El Service NodePort ha asignado un puerto de acceso aleatorio (entre el 30000 - 40000) que nos permite acceder a la aplicación mediante la IP del nodo master. En el ejemplo si accedemos a 10.0.2.4:30453 estaremos accediendo al Service que nos permitirá acceder a la aplicación

.



KUBERNETES

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

Veamos la descripción:

•Vamos a crear un recurso Service (parámetro kind) y lo nombramos como nginx (parámetro name). Este nombre será importante para la resolución dns.
•En la especificación del recurso indicamos el tipo de Service (parámetro type).
•A continuación, definimos el puerto por el que va a ofrecer el Service y lo nombramos (dentro del apartado port: el parámetro port y el parámetro name). Además, debemos indicar el puerto en el que los Pods están ofreciendo el Service (parámetro targetPort), en este caso, hemos usado el nombre del puerto (http) que indicamos en el recurso Deployment:
... ports: - name: http containerPort: 80 ...
•Por ultimo, seleccionamos los Pods a los que vamos acceder y vamos a balancear la carga seleccionando los Pods por medio de sus etiquetas (parámetro selector).

KUBERNETES

Crear directamente el Servicio desde el fichero yaml:	kubectl apply –f servicio.yaml
Ver los recursos que se han creado:	kubectl get all
Si queremos obtener información detallada del recurso Service que hemos creado:	kubectl describe service nombre
Si queremos ver los logs generados en los Pods de un service	kubectl logs service/nombre
Si eliminamos el service se eliminarán el deployment asociado	kubectl delete service nombre

KUBERNETES

Hasta ahora tenemos dos opciones principales para acceder a nuestras aplicaciones desde el exterior:

1.Utilizando Services del tipo NodePort: Esta opción no es muy viable para entornos de producción ya que tenemos que utilizar puertos aleatorios desde 30000-40000.

2.Utilizando Services del tipo LoadBalancer: Esta opción sólo es válida si trabajamos en un proveedor Cloud que nos ofrece un balanceador de carga para cada una de las aplicaciones, en cloud público puede ser una opción muy cara.

La solución puede ser utilizar un Ingress controller que nos permite utilizar un proxy inverso (HAproxy, nginx, traefik,...) que por medio de reglas de encaminamiento que obtiene de la API de Kubernetes, nos permite el acceso a nuestras aplicaciones por medio de nombres.

KUBERNETES

Para ver los *addons* que nos ofrece minikube podemos ejecutar:

minikube addons list

Para activar el Ingress Controller ejecutamos:

minikube addons enable ingress

Para comprobar si tenemos instalado el componente, podemos visualizar los Pods creados en el namespace ingress-nginx. Este espacio de nombre se ha creado para desplegar el controlador de ingress

Por lo tanto al ejecutar:

kubectl get pods -n ingress-nginx ... ingress-nginx-controller-558664778f-shjzp 1/1 Running 0 ...

Debe aparece un Pod que se llama ingress-nginx-controller-..., si es así, significa que se ha instalado un Ingress Controller basado en el proxy inverso nginx.

KUBERNETES

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: nginx
spec:
  rules:
  - host: www.example.org
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: nginx
            port:
              number: 80
```

Hemos indicado el tipo de recurso Ingress (kind) y le hemos puesto un nombre (name). A continuación en la especificación del recurso vamos a poner una regla que relaciona un nombre de host con un Service que me permita el acceso a una aplicación:

- **host**: Indicamos el nombre de host que vamos a usar para el acceso. Este nombre debe apuntar a la ip del nodo master.
- **path**: Indicamos el path de la url que vamos a usar, en este caso sería la ruta raíz: /. Esto nos sirve por si queremos servir la aplicación en una ruta determinada, por ejemplo: www.example.org/app1.
- **pathType**: No es importante, nos permite indicar cómo se van a trabajar con las URL.
- **backend**: Indicamos el Service al que vamos a acceder. En este caso indicamos el nombre del Service (service/name) y el puerto del Service (service/port/number).

KUBERNETES

Crear directamente el recurso ingress desde el fichero yaml:	kubectl apply -f ingress.yaml
Ver los recursos que se han creado:	kubectl get ingress
Si queremos obtener información detallada del recurso ingress que hemos creado:	kubectl describe ingress/nginx

KUBERNETES

Una vez que tenemos creado el despliegue de la aplicación, que realizamos en Despliegue de la aplicación GuestBook, vamos a crear los Services correspondientes para acceder a ella:

Service para acceder a la aplicación

1. El primer Service que vamos a crear nos va a permitir acceder a la aplicación GuestBook desde el exterior, para ello crea un fichero yaml con la definición del Service.
2. Comprueba el puerto que le han asignado al Service para acceder desde el exterior.
3. Accede a la ip del nodo master y al puerto asignado desde un navegador web para ver la aplicación.
4. Responde la siguiente pregunta: ¿Por qué aparece el mensaje de error: **Waiting for database connection...**?

Service para acceder a la base de datos.

Tienes que poner el tipo del Service, el puerto del servicio será el 6379 y el nombre del puerto de la base de datos que hemos asignado en el Deployment es redis-server.
Nota: No cambies el nombre del Service, ya que la aplicación guestbook va a acceder por defecto a la base de datos usando el nombre redis.

1. Elabora el fichero yaml con la definición del Service, y créalo.
2. Lista los Services que has creado.
3. Accede a la ip del nodo master y al puerto asignado desde un navegador web para ver la aplicación. Comprueba que funciona sin ningún problema.

Ingress:

1.Activa el *addon* ingress en minikube para instalar el Ingress Controller.
2.Crea La definición del recurso Ingress con los datos sugeridos, y crea el recurso Ingress.
3.Modifica el fichero /etc/hosts de tu ordenador para configurar la resolución estática.
4.Accede a la aplicación usando el nombre que has asignado.

KUBERNETES

Despliegues parametrizados: Variables de entorno

```
env:
  - name: MYSQL_ROOT_PASSWORD
    value: my-password
```

Limitaciones:

- Los valores de las variables de entorno están escritos directamente en el fichero yaml. Estos ficheros yaml suelen estar en repositorios git y lógicamente no es el sitio más adecuado para ubicarlos.
- Por otro lado, escribiendo los valores de las variables de entorno directamente en los ficheros, hacemos que estos ficheros no sean reutilizables en otros despliegues y que el procedimiento de cambiar las variables sea tedioso y propenso a errores, porque hay que hacerlo en varios sitios.

KUBERNETES

Variables de entorno

Soluciones:

- ConfigMap permite definir un diccionario (clave,valor) para guardar información que se puede utilizar para configurar una aplicación.

```
kubectl create cm mariadb --from-literal=root_password=my-password \
                          --from-literal=mysql_usuario=usuario     \
                          --from-literal=mysql_password=password-user \
                          --from-literal=basededatos=test
```

```
...
    spec:
      containers:
        - name: contenedor-mariadb
          image: mariadb
          ports:
            - containerPort: 3306
              name: db-port
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                configMapKeyRef:
                  name: mariadb
                  key: root_password
            - name: MYSQL_USER
              valueFrom:
                configMapKeyRef:
                  name: mariadb
                  key: mysql_usuario
            - name: MYSQL_PASSWORD
              valueFrom:
                configMapKeyRef:
                  name: mariadb
                  key: mysql_password
            - name: MYSQL_DATABASE
              valueFrom:
                configMapKeyRef:
                  name: mariadb
                  key: basededatos
```

KUBERNETES

Variables de entorno

Soluciones:

- Los Secrets permiten guardar información sensible que será **codificada** o **cifrada**.

```
kubectl create secret generic mariadb --from-literal=password=my-password
```

```
...
    spec:
      containers:
        - name: mariadb
          image: mariadb
          ports:
            - containerPort: 3306
              name: db-port
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: mariadb
                  key: password
```

KUBERNETES

En un ejemplo del módulo anterior: Ejemplo completo: Desplegando y accediendo a la aplicación Temperaturas habíamos desplegado una aplicación formada por dos microservicios que nos permitía visualizar las temperaturas de municipios.

Recordamos que el componente frontend hace peticiones al componente backend utilizando el nombre temperaturas-backend, que es el nombre que asignamos al Service ClusterIP para el acceso al backend.

Vamos a cambiar la configuración de la aplicación para indicar otro nombre.

Podemos configurar el nombre del servidor backend al que vamos acceder desde el frontend modificando la variable de entorno TEMP_SERVER a la hora de crear el despliegue del frontend.

Por defecto el valor de esa variable es:

TEMP_SERVER temperaturas-backend:5000

Vamos a modificar esta variable en el despliegue del frontend y cambiaremos el nombre del Service del backend para que coincidan, para ello realiza los siguientes pasos:

1.Crea un recurso ConfigMap con un dato que tenga como clave SERVIDOR_TEMPERATURAS y como contenido servidor-temperaturas:5000.

2.Modifica el fichero de despliegue del frontend: frontend-deployment.yaml para añadir la modificación de la variable TEMP_SERVER con el valor que hemos guardado en el ConfigMap.

3.Realiza el despliegue y crea el Service para acceder al frontend.

4.Despliega el microservicio backend.

5.Modifica el fichero backend-srv.yaml para cambiar el nombre del Service por servidor-temperaturas y crea el Service.

6.Accede a la aplicación usando el puerto asignado al Service NodePort del frontend o creando el recurso Ingress.

KUBERNETES

ALMACENAMIENTO

Ya lo hemos comentando en anteriores módulos, pero es importante tener en cuenta que **los Pods son efímeros**, es decir, cuando un Pod se elimina se pierde toda la información que tenía. Evidentemente, cuando creamos un nuevo Pod no contendrá ninguna información adicional a la propia aplicación.

La solución consiste en usar Volúmenes. Con el uso de dichos volúmenes vamos a conseguir varias cosas:

1. Si un Pod guarda su información en un volumen, está no se perderá. Por lo que podemos eliminar el Pod sin ningún problema y cuando volvamos a crearlo mantendrá la misma información. En definitiva, los volúmenes proporcionan almacenamiento adicional o secundario al disco que define la imagen.
2. Si usamos volúmenes, y tenemos varios Pods que están ofreciendo un servicio, estos Pods tendrán la información compartida y por tanto todos podrán leer y escribir la misma información.
3. También podemos usar los volúmenes dentro de un Pod, para que los contenedores que forman parte de él puedan compartir información.

KUBERNETES

ALMACENAMIENTO

Tipos de volúmenes

- Proporcionados por proveedores de cloud: AWS, Azure, GCE, OpenStack, etc
- Propios de Kubernetes:
  - configMap: Para usar un configMap como un directorio desde el Pod.
  - emptyDir: Volumen efímero con la misma vida que el Pod. Usado como almacenamiento secundario o para compartir entre contenedores del mismo Pod.
  - hostPath: Monta un directorio del host en el Pod (usado excepcionalmente, pero es el que nosotros vamos a usar con minikube).
  - ...
- Habituales en despliegues "on premises": glusterfs, cephfs, iscsi, nfs, etc.

KUBERNETES

ALMACENAMIENTO

Aprovisionamiento estático PersistentVolumen (PV).
Tenemos tres modos de acceso, que dependen del backend que vamos a utilizar:

- ReadWriteOnce: read-write solo para un nodo (RWO)
- ReadOnlyMany: read-only para muchos nodos (ROX)
- ReadWriteMany: read-write para muchos nodos (RWX)

Las políticas de reciclaje de volúmenes también dependen del backend y son:

- Retain: El PV no se elimina, aunque el PVC se elimine. El administrador debe borrar el contenido para la próxima asociación.
- Recycle: Reutilizar contenido. Se elimina el contenido y el volumen es de nuevo utilizable.
- Delete: Se borra después de su utilización.

A modo de resumen, ponemos en la siguiente tabla los modos de acceso de algunos de los sistemas de almacenamiento más usados:

KUBERNETES

ALMACENAMIENTO

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv1
spec:
  storageClassName: manual
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /data/pv1
```

KUBERNETES

ALMACENAMIENTO

Aprovisionamiento dinámico PersistentVolumen (PVC).

Cuando el desarrollador necesita almacenamiento para su aplicación, hace una petición de almacenamiento creando un recurso PersistentVolumenClaim (PVC) y de forma dinámica se crea el recurso PersistentVolume que representa el volumen y se asocia con esa petición. De otra forma explicado, cada vez que se cree un PersistentVolumenClaim, se creará bajo demanda un PersistentVolumen que se ajuste a las características seleccionadas.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: pvc1
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

KUBERNETES

ALMACENAMIENTO

Una vez que hemos solicitado el almacenamiento, y se ha asignado un volumen (ya sea de forma dinámica o estática), vamos a definir el uso que se va a hacer de este volumen.
Como ejemplo, vamos a definir un Pod que utilice dicho volumen, para ello vamos a crear un fichero yaml con la siguiente definición:

```
kind: Pod
apiVersion: v1
metadata:
  name: task-pv-pod
spec:
  volumes:
    - name: task-pv-storage
      persistentVolumeClaim:
       claimName: pvc1
  containers:
    - name: task-pv-container
      image: nginx
      ports:
        - containerPort: 80
          name: "http-server"
      volumeMounts:
        - mountPath: "/usr/share/nginx/html"
          name: task-pv-storage
```

KUBERNETES

ALMACENAMIENTO

Siguiendo la guía explicada en el Ejemplo 2: Gestión dinámica de volúmenes, vamos a crear un servidor web que permita la ejecución de scripts PHP con almacenamiento persistente.

Para realizar esta actividad vamos a usar asignación
dinámica de volúmenes y puedes usar, como modelos, los ficheros del ejemplo 2.

Realiza los siguientes pasos:

1. Crea un fichero yaml para definir un recurso PersistentVolumenClaim que se llame pvc-webserver y para solicitar un volumen de 2Gb.
2. Crea el recurso y comprueba que se ha asociado un volumen de forma dinámica a la solicitud.
3. Crea un fichero yaml para desplegar un servidor web desde la imagen php:7.4-apache, asocia el volumen al Pod que se va a crear e indica el punto de montaje en el del servidor: /var/www/html.
4. Despliega el servidor y crea un fichero info.php en /var/www/html, con el siguiente contenido: <?php phpinfo(); ?>.
5. Define y crea un Service NodePort, accede desde un navegador al fichero info.php y comprueba que se visualiza de forma correcta.
6. Comprobemos la persistencia: elimina el Deployment, vuelve a crearlo y vuelve a acceder desde el navegador al fichero info.php. ¿Se sigue visualizando?

KUBERNETES

Despliegue de aplicaciones con Helm

Helm, que es un software que nos permite empaquetar aplicaciones completas y gestionar el ciclo completo de despliegue de dicha aplicación.
Helm usa un formato de empaquetado llamado **charts**. Un chart es una colección de archivos que describen un conjunto de recursos que nos permite desplegar una aplicación en Kubernetes.
Los **charts** son distribuidos en distintos repositorios, que podremos dar de alta en nuestra instalación de Helm. Para buscar los distintos charts y los repositorios desde los que se distribuyen podemos usar la página Artifact Hub.

KUBERNETES

Despliegue de aplicaciones con Helm

Para instalar el chart ejecutamos la siguiente instrucción:

```
helm install serverweb bitnami/nginx --set service.type=NodePort
```

Como vemos hemos nombrado el chart desplegado (`serverweb`), indicado el chart (`bitnami/nginx`) y, en este caso, hemos redefinido el parámetro `service.type`.

Cuando se despliega el chart se nos ofrece información que nos muestra cómo acceder a la aplicación:

```
NOTES:
** Please be patient while the chart is being deployed **

NGINX can be accessed through the following DNS name from within your cluster:

    serverweb-nginx.default.svc.cluster.local (port 80)

To access NGINX from outside the cluster, follow the steps below:

1. Get the NGINX URL by running these commands:

    export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" services serverweb-nginx)
    export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresses[0].address}")
    echo "http://${NODE_IP}:${NODE_PORT}"
```

Si queremos acceder a la aplicación desde el exterior debemos ejecutar las tres últimas instrucciones, que nos muestran la ip de nuestro cluster y el puerto asignado al Service NodePort.

Siempre podemos volver a ver esta información ejecutando la siguiente instrucción:

```
helm status serverweb
```

Podemos comprobar los Deployments que hemos realizado con Helm, ejecutando:

```
helm ls
NAME            NAMESPACE       REVISION        UPDATED                                 STATUS          CHART           APP VERSION
serverweb       default         1               2021-06-29 19:11:15.975016119 +0200 CEST        deployed        nginx-9.3.0     1.21.0
```

Y podemos comprobar también los recursos que se han creado en el cluster:

```
kubectl get all
NAME                                READY   STATUS    RESTARTS   AGE
```

KUBERNETES

Añadir un repositorio a helm	helm repo add "stable" "https://charts.helm.sh/stable" --force-update
Listar los repositorios de helm	helm repo list
Actualizar la lista de charts ofrecidos	helm repo update
Buscar charts	helm search repo nombre-chart
Instalar un chart	helm install nombre repo/chart --set service.type=NodePort Ej: helm install serverweb bitnami/nginx --set service.type=NodePort
Desinstalar una aplicación completa	helm delete nombre

KUBERNETES

Vamos a instalar el CMS Wordpress usando Helm. Para ello, realiza los siguientes pasos:

1. Instala la última versión de Helm.
2. Añade el repositorio de bitnami
3. Busca el chart de bitnami para la instalación de Wordpress.
4. Busca la documentación del chart y comprueba los parámetros para cambiar el tipo de Service y el nombre del blog.
5. Instala el chart definiendo el tipo del Service como NodePort y poniendo tu nombre como nombre del blog.
6. Comprueba los Pods, ReplicaSet, Deployment y Services que se han creado.
7. Accede a la aplicación.

KUBERNETES

OTRAS CARGAS DE TRABAJO

Una característica de una aplicación que es muy importante para Kubernetes es si se trata de una aplicación con estado (*stateful*) o sin estado (*stateless*).

Stateless: Una aplicación sin estado es aquella en la que las peticiones son totalmente independientes unas de otras y no necesita ninguna referencia de una petición anterior. Un ejemplo de una aplicación sin estado sería un servicio DNS, en el que cada vez que se realiza una petición es totalmente independiente de las anteriores o posteriores que se hagan. Las aplicaciones sin estado son perfectas para desplegarse el Kubernetes, ya que se ajustan perfectamente a un Deployment y se pueden escalar y balancear sin problemas, ya que cada pod responderá a las peticiones que reciba de forma independiente al resto.

Stateful: Por contra, las aplicaciones con estado son aquellas en las que una petición puede verse afectada por el resultado de las anteriores y a su vez puede afectar a las posteriores (por eso se dice que tiene estado). Una base de datos sería el paradigma de una aplicación con estado, puesto que cada modificación que hagamos a la base de datos puede afectar a las consultas posteriores. Una aplicación con estado no se ajusta bien a un Deployment de Kubernetes, ya que de forma general, un cluster de pods independientes no puede tener en cuenta el estado de la aplicación correctamente.