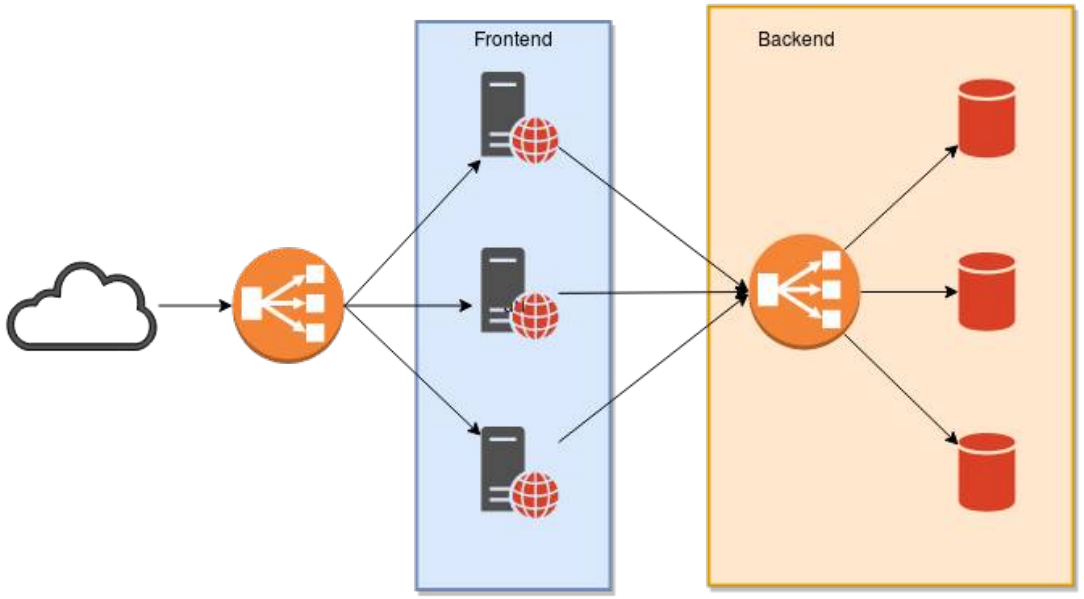
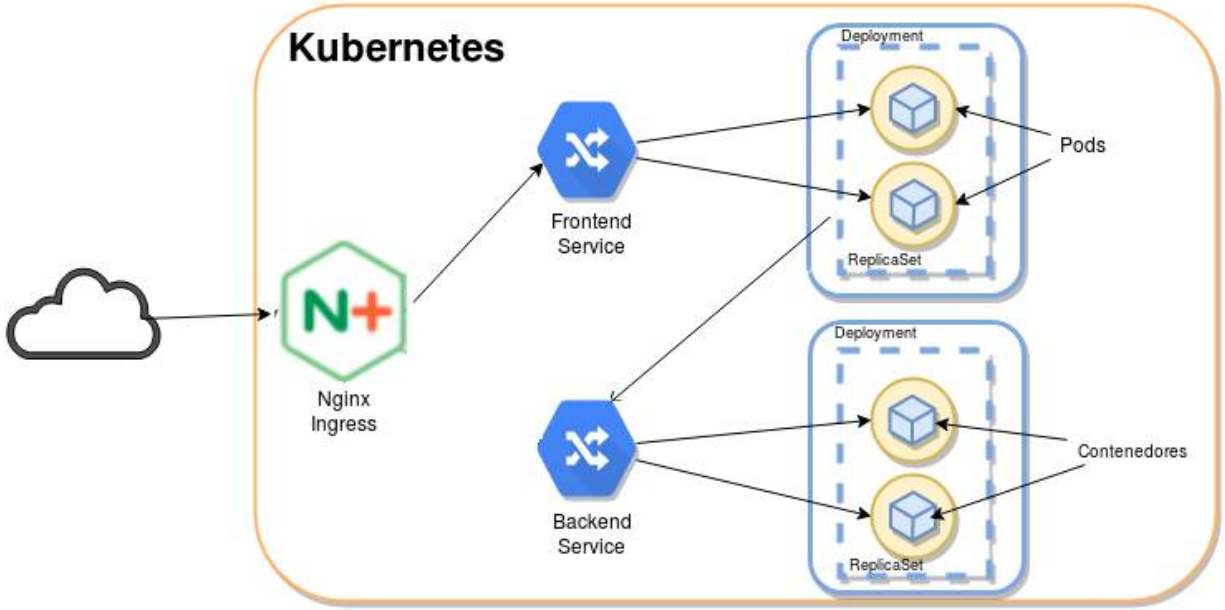
UNIDAD 3
DESPLIEGUE DE APLICACIONES EN KUBERNETES

Noviembre 2020

Despliegue de aplicaciones en Kubernetes



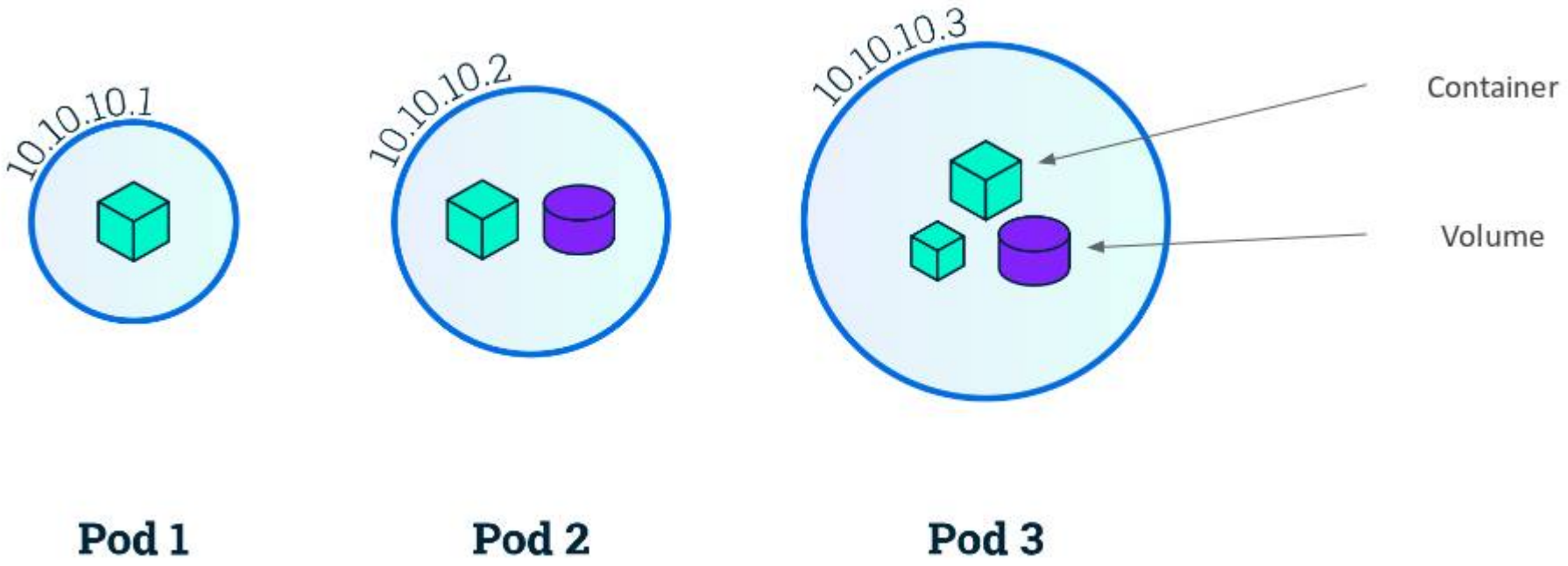
- Máquinas físicas/virtuales
- Balanceadores de carga
- Proxys inversos



- Pods
- ReplicaSets
  - Que no haya caída del servicio
  - Tolerancia a errores
  - Escabilidad dinámica
- Deployments
  - Actualizaciones continúas
  - 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:

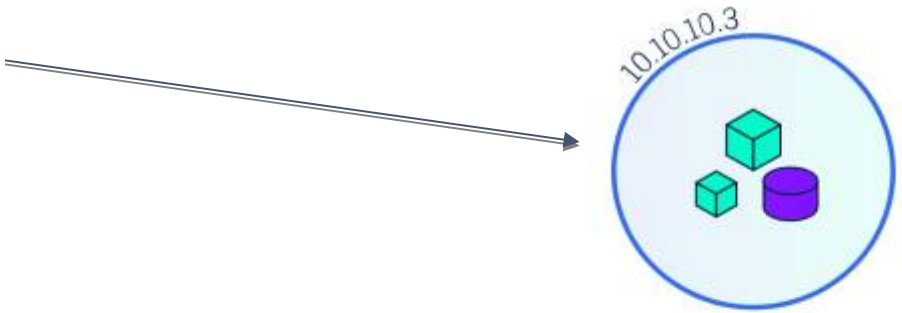
- Kubernetes puede trabajar con distintos contenedores (Docker, Rocket, cri-o,...) por lo tanto es necesario añadir una capa de abstracción que maneje las distintas clases de contenedores.
- Además esta capa de abstracción añade información adicional necesaria en Kubernetes como por ejemplo, políticas 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
```

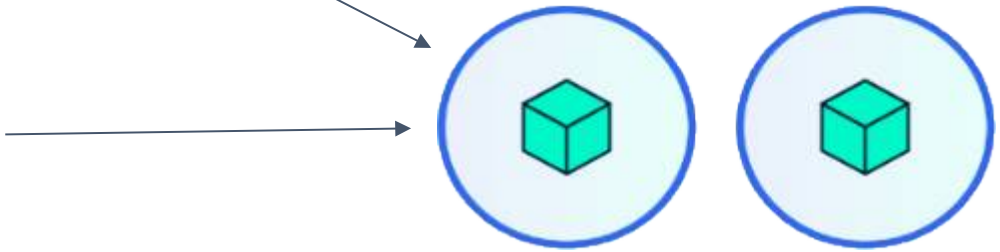
¿Pod multicontenedor o múltiples pods?

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.

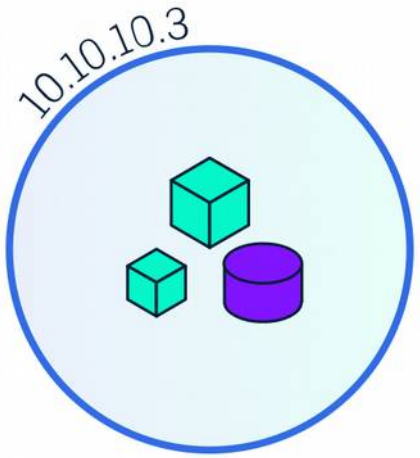


Algunas notas sobre pods multicontenedores...

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
- 7 container design patterns you need to know



Para más información acerca de los pod puedes leer: la documentación de la API y la guía de usuario.

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:

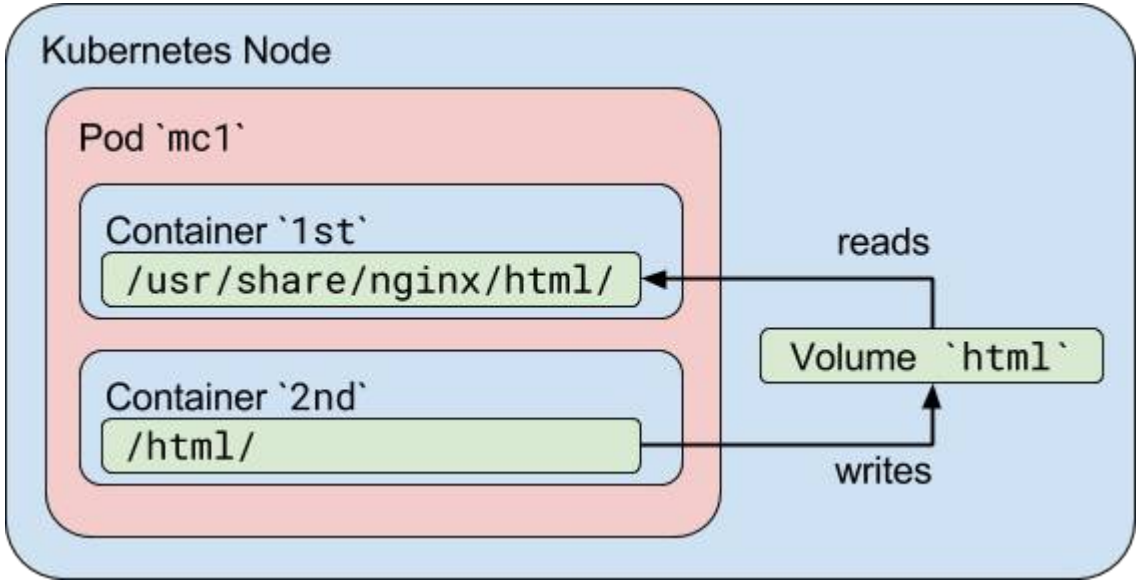
```yaml
apiVersion: v1 # required
kind: Pod # required
metadata: # required
 name: nginx # required
 namespace: default
 labels:
   app: nginx
   service: web
spec: # required
 containers:
   - image:  nginx:1.16
     name:  nginx
     imagePullPolicy: Always
```

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.

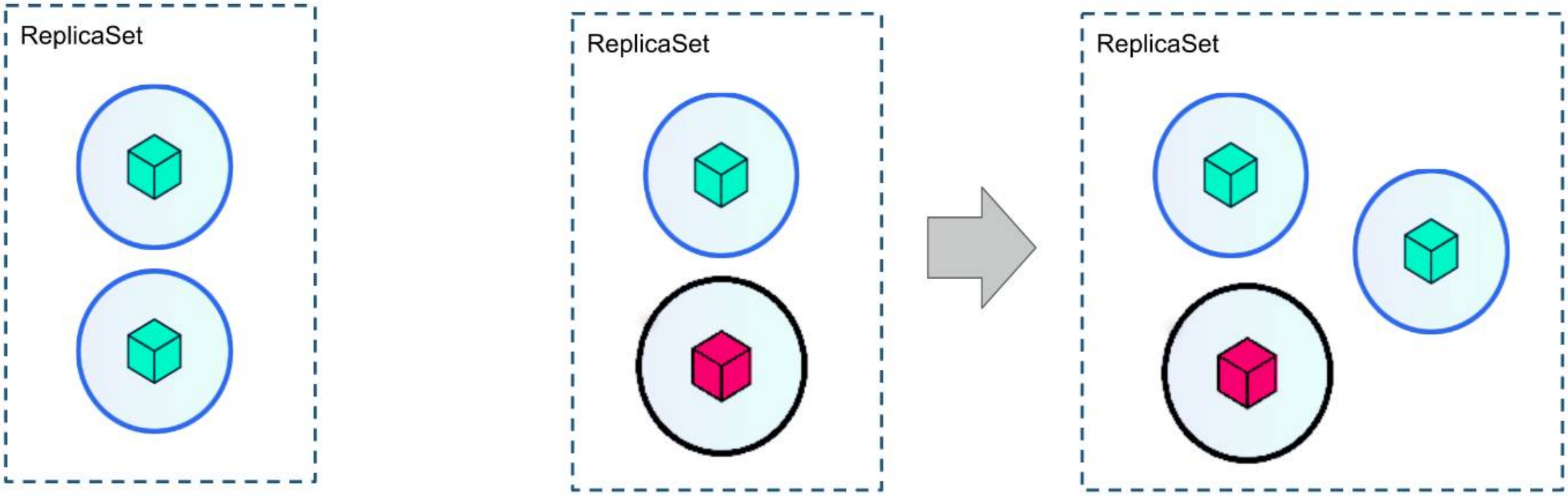
- Las imágenes se guardan en un registro interno.
- Se pueden utilizar registros públicos (google, docker hub,...) y registros privados.
- La política por defecto es `IfNotPresent`, que se baja la imagen si no está en el registro interno. Si queremos forzar la descarga:
  - `imagePullPolicy: Always`
  - Omitir `imagePullPolicy` y usar la etiqueta `:latest`
  - Omitir `imagePullPolicy` y la etiqueta de la imagen
- Las Labels nos permiten etiquetar los recursos de kubernetes (por ejemplo un pod) con información del tipo clave/valor.

- **EJEMPLO 1: Trabajando con Pod**
- **EJEMPLO 2: Pod multicontenedores**



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

```yaml
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.

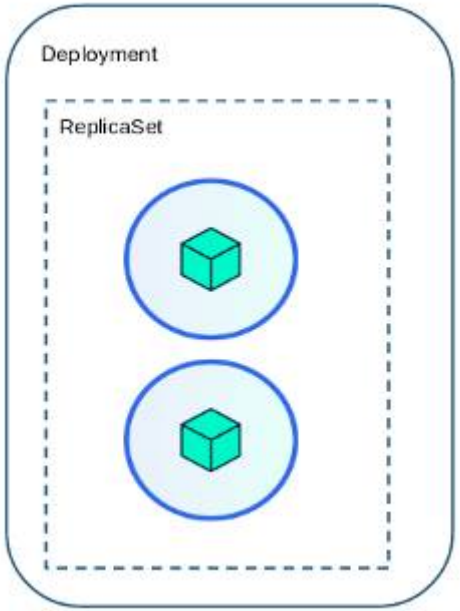
Para más información acerca de los ReplicaSet puedes leer: la documentación de la API y la guía de usuario.

Ejemplo 3: Trabajando con ReplicaSet

Deployment

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

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




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

Describiendo objetos k8s: Deployment

```yaml
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.

Para más información acerca de los Deployment puedes leer: la documentación de la API y la guía de usuario.

Ejemplo 4: Trabajando con Deployment

Manejo de objetos k8s

- **Comandos imperativos**

El usuario opera directamente con los objetos de la API. Ejemplos:

```
kubectl create deployment nginx --image nginx
kubectl expose deployment/nginx --port=80 –type=NodePort
kubectl get all
```

- **Configuración imperativa de objetos**

Especificamos un comando imperativo, pero la definición del objeto está guardada en un fichero Yaml. Posteriormente no podremos modificar el objeto, habrá que bortrarlo y crearlo de nuevo. Ejemplos:

```
kubectl create -f deploy.yaml
kubectl delete -f deploy.yaml
```

- **Configuración declarativa de objetos**

No se definen las acciones a realizar. Cuando se aplica la configuración del objeto estamos indicando un estado deseado al que queremos llegar. Posteriormente si la definición cambia, podremos cambiar el objeto. Recomendado en producción. Ejemplo:

```
kubectl apply -f deploy.yaml
```

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:

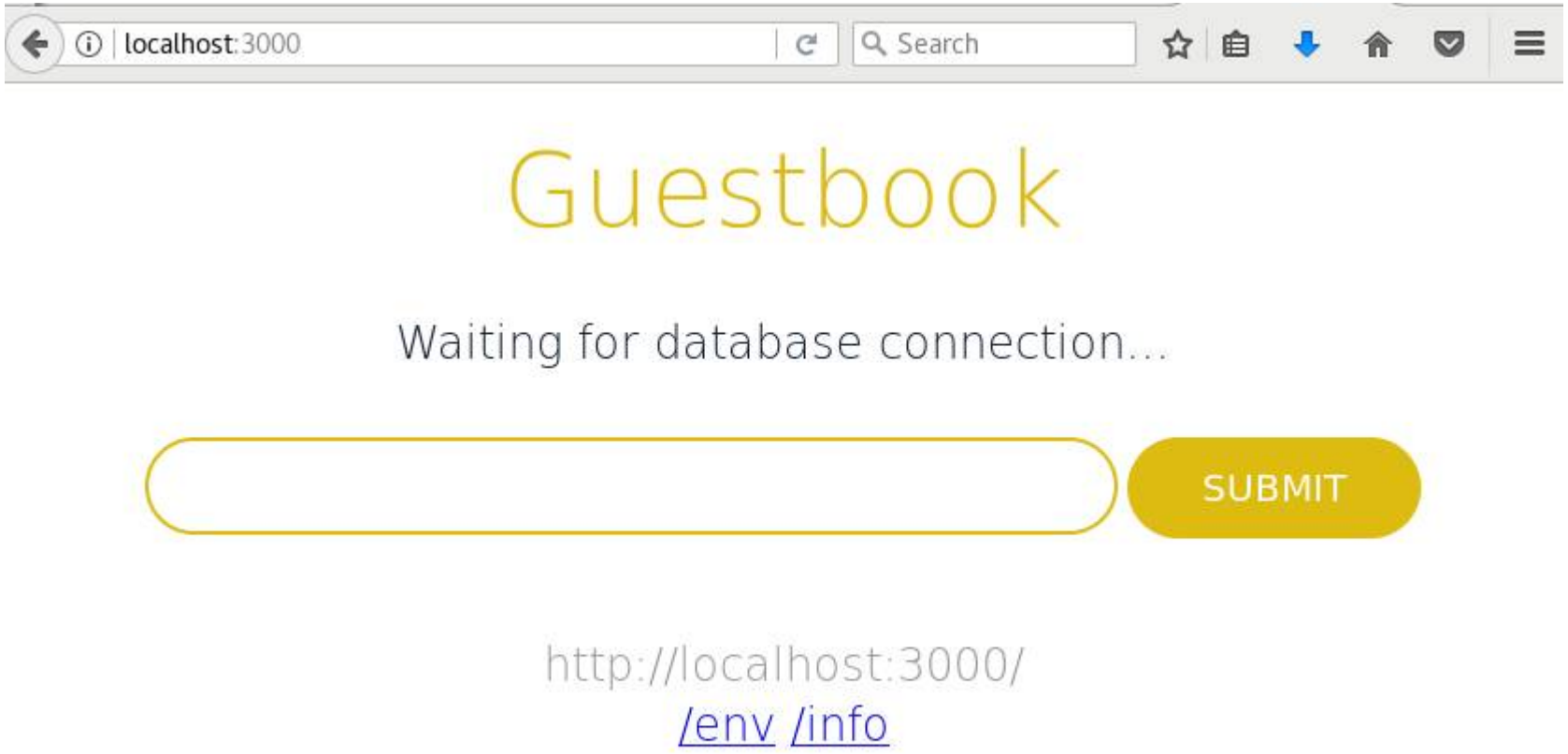
```
kubectl rollout undo deployment nginx
```

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

EJEMPLO 6: Desplegando la aplicación GuestBook (Parte 1)



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

DEPLOYMENT STRATEGIES

When it comes to production, a ramped or blue/green deployment is usually a good fit, but proper testing of the new platform is necessary.

Blue/green and shadow strategies have more impact on the budget as it requires double resource capacity. If the application lacks in tests or if there is little confidence about the impact/stability of the software, then a canary, a/b testing or shadow release can be used.

If your business requires testing of a new feature amongst a specific pool of users that can be filtered depending on some parameters like geolocation, language, operating system or browser features, then you may want to use the a/b testing technique.

Container Solutions

Strategy	ZERO DOWNTIME	REAL TRAFFIC TESTING	TARGETED USERS	CLOUD COST	ROLLBACK DURATION	NEGATIVE IMPACT ON USER	COMPLEXITY OF SETUP
RECREATE version A is terminated then version B is rolled out	✗	✗	✗	■□□	■■■	■■■	□□□
RAMPED version B is slowly rolled out and replacing version A	✓	✗	✗	■□□	■■■	■□□	■□□
BLUE/GREEN version B is released alongside version A, then the traffic is switched to version B	✓	✗	✗	■■■	□□□	■■□	■■□
CANARY version B is released to a subset of users, then proceed to a full rollout	✓	✓	✗	■□□	■□□	■□□	■■□
A/B TESTING version B is released to a subset of users under specific condition	✓	✓	✓	■□□	■□□	■□□	■■■
SHADOW version B receives real world traffic alongside version A and doesn't impact the response	✓	✓	✗	■■■	□□□	□□□	■■■

https://github.com/ContainerSolutions/k8s-deployment-strategies