Kubernetes

2. Tipos de Workloads

Índice

Deployments y ReplicaSets

Statefulsets

DaemonSets

Jobs y CronJobs

Características básicas de los pods:

- **Asignación de recursos**
- **Estado de salud**

Objetivos

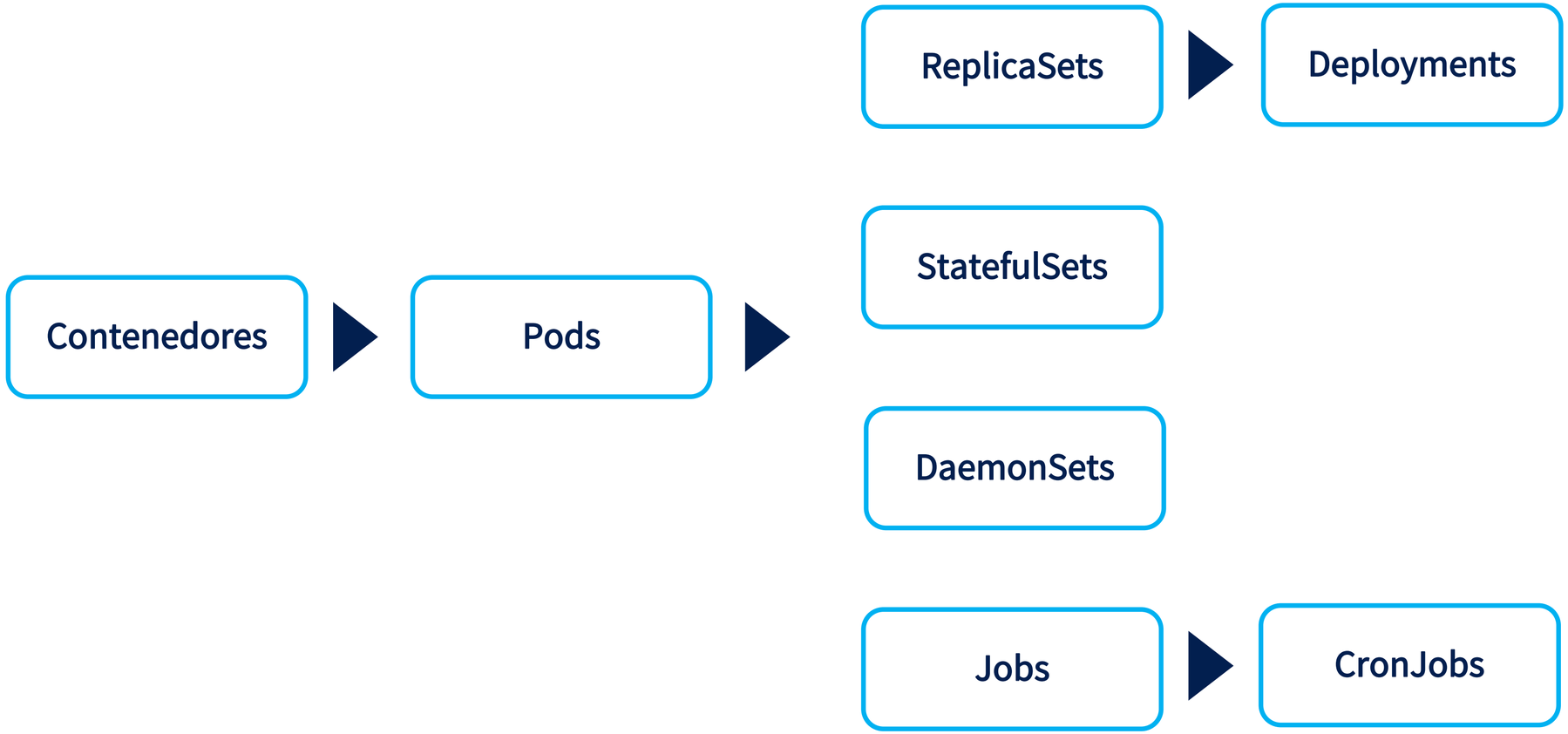
Aprender a crear objetos de Kubernetes

Conocer todas las posibilidades de un Pod

Profundizar en los diferentes tipos de workloads

Adquirir práctica y visualizar los resultados

Mapa Conceptual



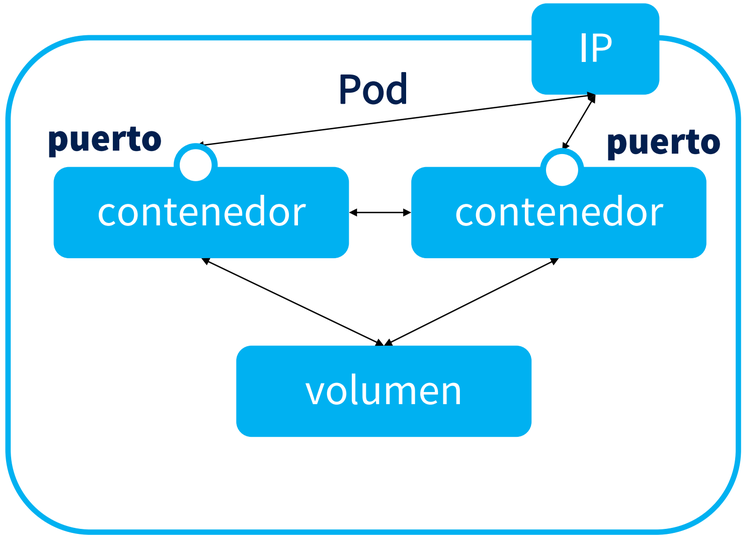
Recordamos: Objetos YAML en Kubernetes

Todos los objetos tienen la misma estructura principal:

```
apiVersion: <La API de mi objeto / el catálogo>
kind: <El tipo de objeto dentro del catálogo>
metadata:
    name: <Nombre del objeto>
spec:
  <
  ....
  Todas las variables del objeto
  ....
  >
```

Recordamos: Pods

Un pod es el objeto básico de cargas de trabajo en Kubernetes.



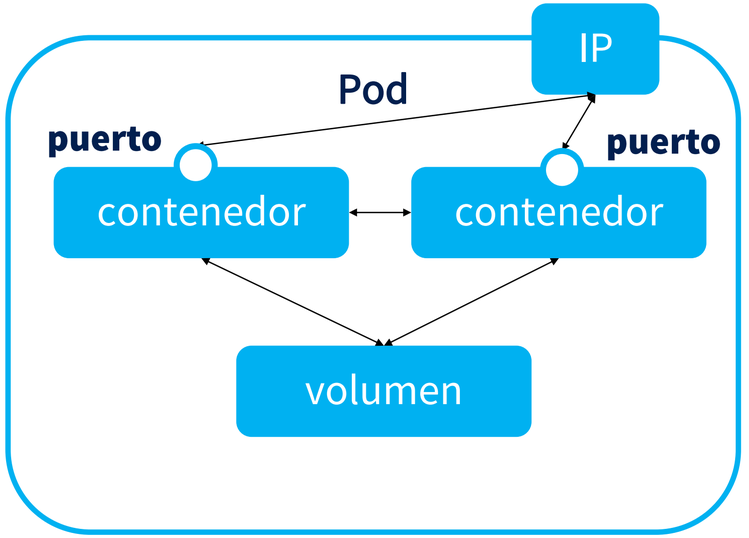
Algunas características

- Los contenedores del mismo pod siempre se ejecutan en el mismo nodo
- Están aislados del resto de contenedores que corren en el mismo nodo
- Los contenedores del mismo pod se pueden comunicar utilizando *localhost* o 127.0.0.1
- Comparten una misma IP para comunicarse con otros pods
- Pueden compartir directorios entre sí. Esto no se puede hacer entre distintos pods

Limitaciones de un pod

El Pod es un objeto efímero. Una vez se destruye, no se vuelve a crear. Esto implica:

- Si el proceso del contenedor termina por cualquier razón (un error por ejemplo), nuestra aplicación no se vuelve a levantar.
- Si el nodo en el que está el pod falla, el pod no se levantará de nuevo en otro nodo.
- Si desplegamos una versión errónea de nuestra aplicación, no tenemos manera de volver a la última versión correcta.

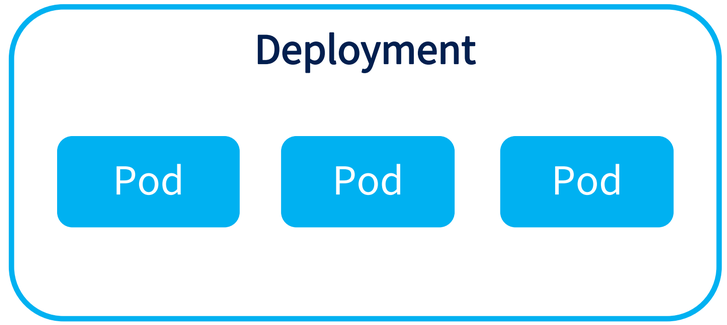


Pero todo esto son características que nos ofrecía Kubernetes. Existen otros objetos básicos que nos permitirán desplegar pods de manera mucho más efectiva: los deployments.

Deployments

El objeto Deployment mejora el objeto Pod, añadiéndole alta disponibilidad:

- Asegura que siempre están disponible el número de pods deseado
- Se encarga de que las actualizaciones sean seguras
- Guarda versiones anteriores para poder volver a ellas



En definitiva, es un objeto superior a pod, que se encarga de crear pods y mantenerlos disponibles.

Todos los pod creados por un Deployment son exactamente iguales. Son réplicas unos de otros.

Deployments

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
spec:
    selector:
        matchLabels:
            app: nginx
    template:
        metadata:
            labels:
                app: nginx
        spec:
            containers:
            - name: nginx
              image: nginx
              ports:
              - containerPort: 80
```

Deployments

apiVersion es apps/v1. Es la librería principal para cargas de trabajo. La que usaremos todo este módulo

El spec tiene 2 campos importantes:

selector: El Deployment tiene que saber de alguna manera qué pods debe controlar. En selector le estamos diciendo que controle todos los pods que tengan la etiqueta app: nginx

template: No es más que la plantilla de los pods que debe crear el Deployment. Como hemos visto antes, los pods tendrán el campo metadata (en este caso añadimos las etiquetas para que coincida con el matchLabels del Deployment). Y también su propio campo spec, con la lista de contenedores.

Deployments

Vamos a desplegar este Deployment.

Volvemos al entorno de prueba que vimos en el primer módulo:

https://www.katacoda.com/courses/kubernetes/playground

Recuerda pulsar el botón launch.sh para lanzar el clúster de Kubernetes.

Deployments

Vamos a desplegar este Deployment.

Primero creamos un archivo nuevo con nano llamado deployment.yaml

```
nano deployment.yaml
```

Copiamos el código anterior. Guardamos con ctrl+O y salimos con ctrl+X

Deployments

Enviamos este archivo YAML a Kubernetes con

kubectl apply -f deployment.yaml

A continuación visualizamos lo que hemos desplegado:

kubectl get deployment

kubectl get pod

Vemos que se ha creado el deployment nginx-deployment, y que este a desplegado un pod que comienza con el nombre nginx-deployment y otros caracteres.

Deployments

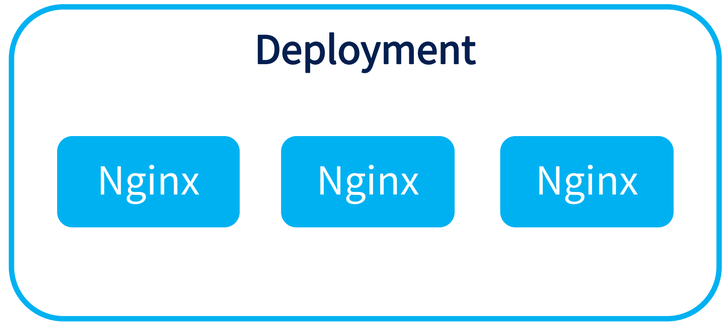
Otro campo importante es **replicas**.

En él especificamos el número de pods que queremos que estén disponibles. Kubernetes se encargará de hacer lo necesario para que siempre tengamos ese número disponible.

Por defecto despliega sólo 1 réplica. Si queremos que nuestro servicio sea de alta disponibilidad, debemos desplegar más.

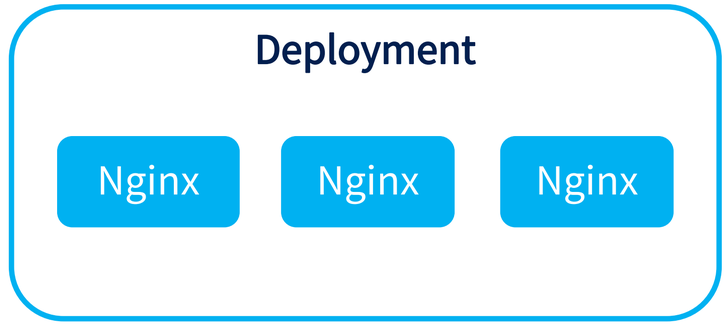
Vamos a modificar el archivo deployment.yaml y añadir el campo replicas como en la siguiente diapositiva.

Aplicamos los cambios con kubectl apply -f deployment.yaml



Deployments

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
spec:
    selector:
        matchLabels:
            app: nginx
    replicas: 3    # Levanta 3 pods
    template:
        metadata:
            labels:
                app: nginx
    spec:
        containers:
        - name: nginx
          image: nginx
          ports:
          - containerPort: 80
```



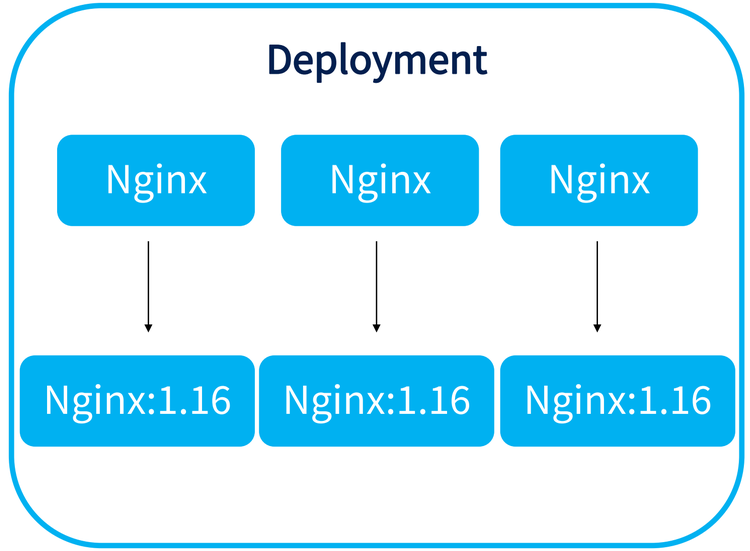
Deployments

Vamos a probar a cambiar la versión de nginx:

Editamos el objeto YAML del deployment y cambiamos el campo image del contenedor a nginx:1.16

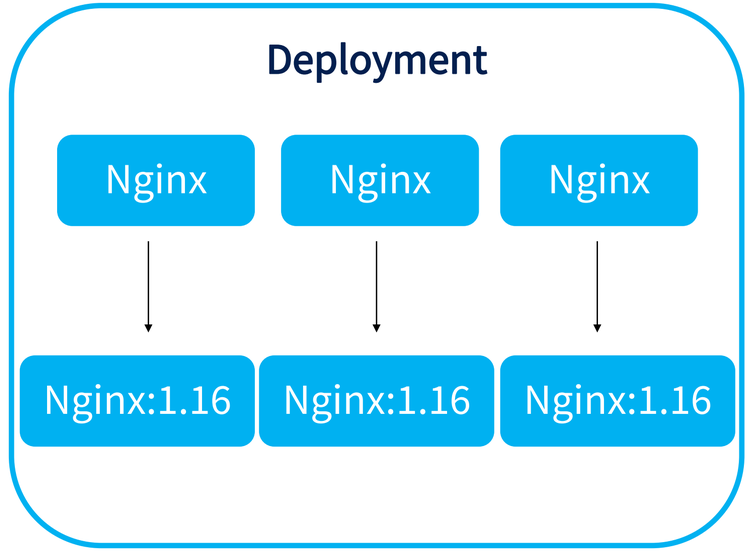
Vemos que el Deployment está actualizando los contenedores uno a uno. Hasta que no está listo un contenedor nuevo con la imagen nginx:1.16, no elimina uno anterior.

Esta estrategia de despliegue se denomina RollingUpdate.



Deployments

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
spec:
    selector:
        matchLabels:
            app: nginx
    replicas: 3    # Levanta 3 pods
    template:
        metadata:
            labels:
                app: nginx
    spec:
        containers:
        - name: nginx
          image: nginx:1.16
          ports:
          - containerPort: 80
```



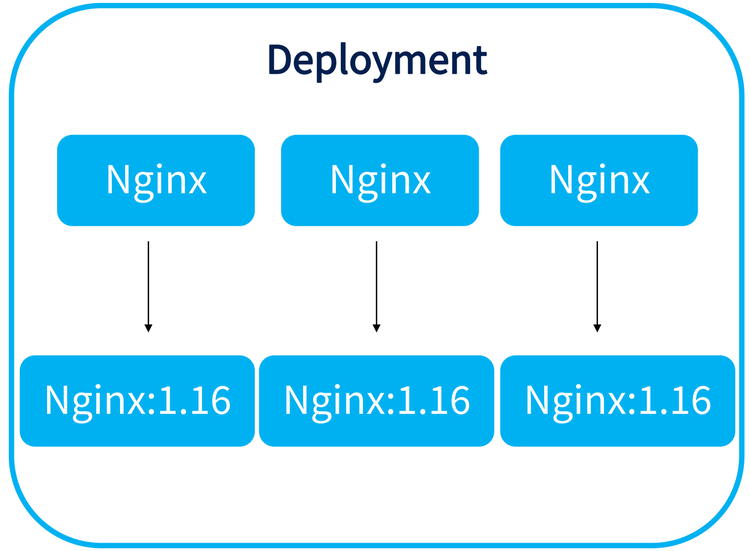
Deployments

En un Deployment es posible controlar la estrategia con la que se despliegan nuevas versiones.

Con el campo strategy, dentro del spec del Deployment, podemos especificar que sea RollingUpdate (por defecto), o Recreate (en el que elimina todos los pods actuales antes de desplegar los nuevos.

Por ejemplo:

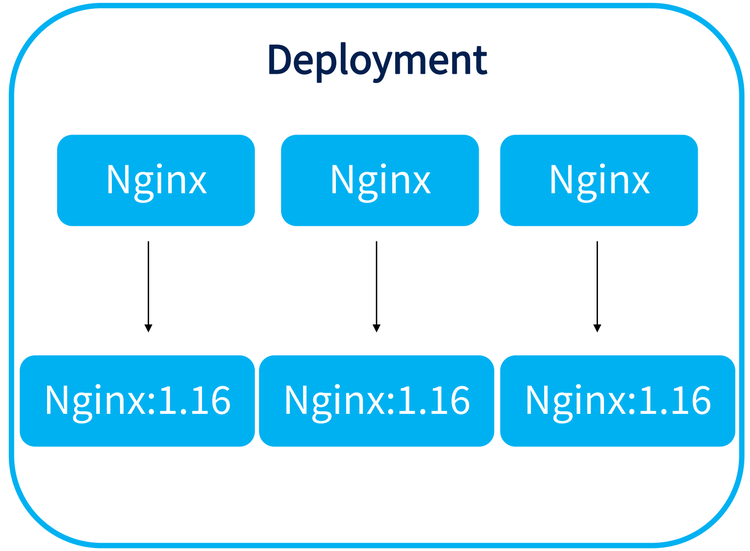
```
strategy:
    type: Recreate
```



Deployments

Añade en deployment.yaml **este campo, dentro del** spec **del deployment, al mismo nivel que** replicas**,** selector **y** template**.**

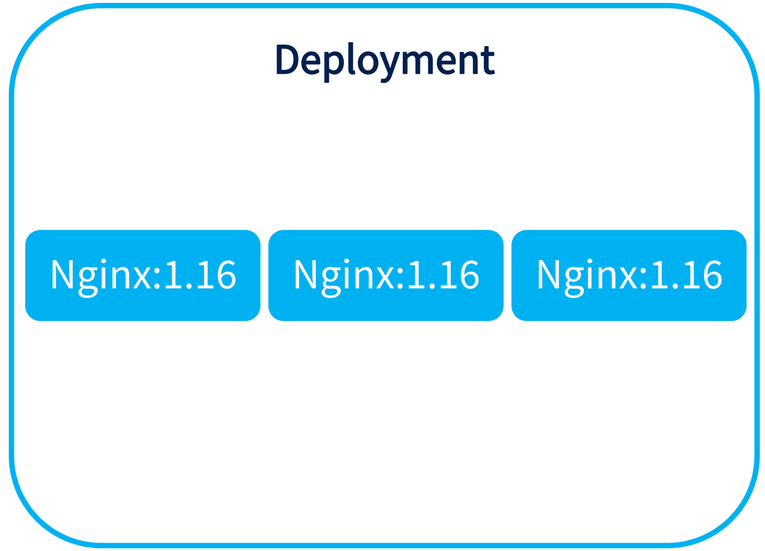
```
spec:
    selector:
        matchLabels:
            app: nginx
    strategy:
        type: Recreate
    replicas: 3
    template:
.....
```



Deployments

Por tanto dentro del spec de un Deployment tendremos:

- selector: Etiquetas para controlar qué pods pertenecen a este deployment. Las etiquetas incluidas en selector.matchLabels siempre deben estar presentes en template.metadata.labels
- strategy: Tipo RollingUpdate o Recreate
- replicas: Número de pods
- template: La plantilla para los pods que crea este Deployment. Dentro de este tendremos un metadata con los labels iguales a los del campo selector.matchLabels y un campo spec con la especificación del pod.



Deployments

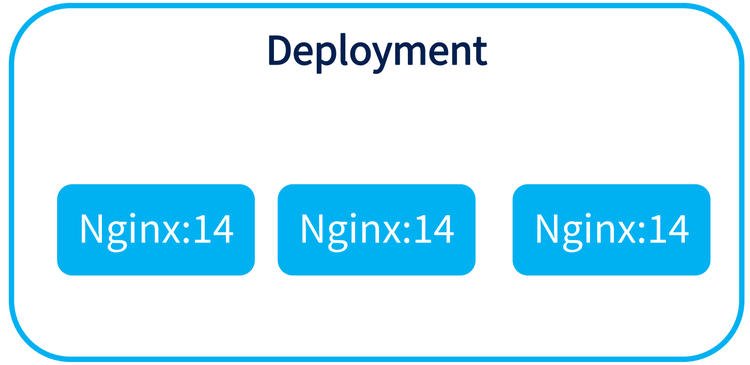
Entonces, ¿qué son esos números y letras al final del nombre de cada pod?

Indican la versión del Deployment de la que forma parte cada pod.

Verás que el primer conjunto de números es igual en todos los pods del Deployment.

nginx-deployment-**1564180365**-70iae

Cuando has cambiado la imagen de Nginx, habrás visto que estos números que eran los mismos para los 3 pods, han cambiado para los nuevos



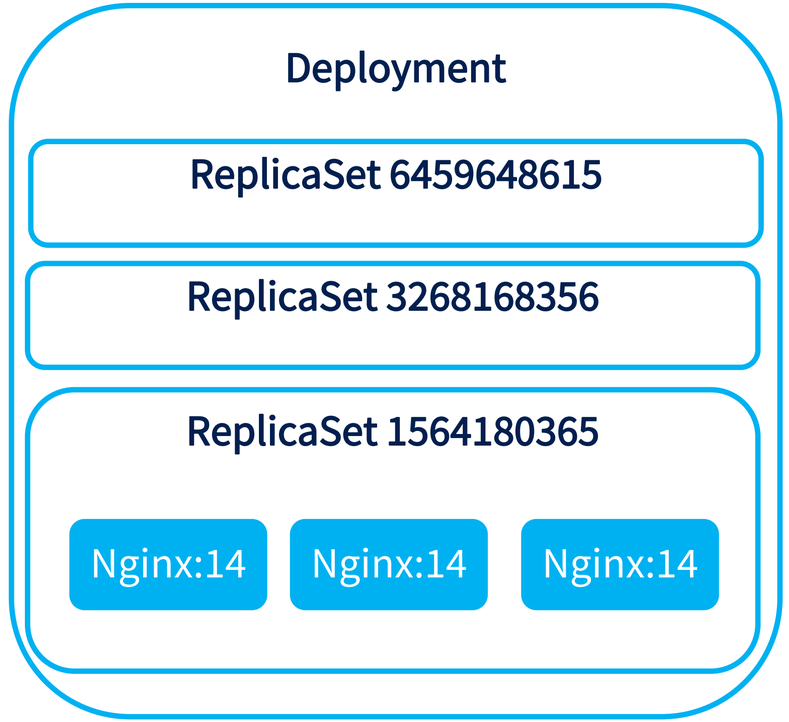
Deployments

Esos números identifican una versión del deployment. Son iguales para todos los pods que comparten la misma versión.

En realidad, esos números identifican un objeto de Kubernetes llamado **ReplicaSet**. Es un objeto que nuestro deployment crea automáticamente cada ves que se despliega una nueva versión.

Los ReplicaSet son útiles porque contienen el código de cada versión de nuestro Deployment. Nos permiten volver atrás como veremos en otro módulo.

Aunque todos los pods estén en la última versión, los ReplicaSets de las anteriores siguen ahí, con 0 pods.



Deployments

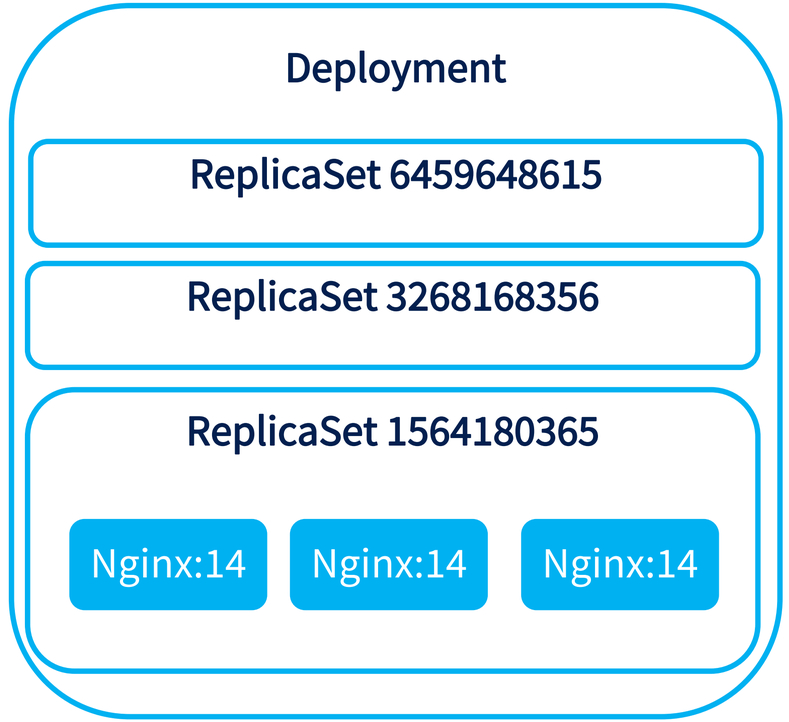
Puedes listar los ReplicaSet con

```
kubectl get replicaset
```

Obtendrás una lista con un ReplicaSet por cada vez que has actualizado el Deployment.

En realidad, el Deployment no crea los Pods. El Deployment sólo crea ReplicaSets, y los controla para que desplieguen las nuevas versiones con la estrategia que hayamos especificado, y para que tengan el número de réplica que queremos.

Los ReplicaSet son los que se encargan de recrear nuestros pod cuando se caen.



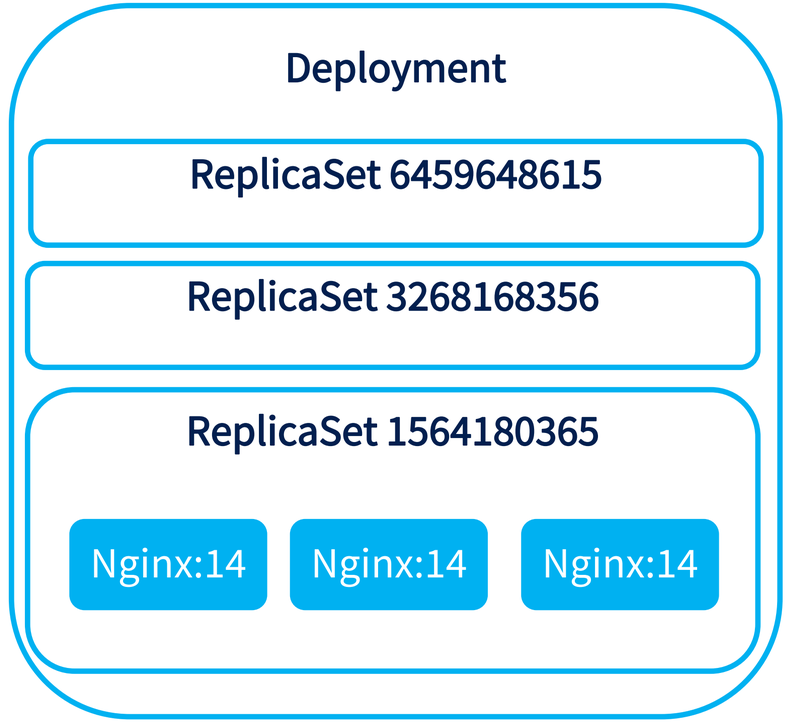
Deployments

¿Por qué no hemos hablado antes de los ReplicSet?

Bien, es porque en Kubernetes no se crean ReplicaSets ‘a mano’. Los ReplicaSets son controlados por cada Deployment y no debemos modificarlos nunca.

nginx-deployment-1564180365-70iae

En cuanto al segundo conjunto de números y letras, identifica a cada pod dentro del mismo ReplicaSet, para que todos tengan finalmente un nombre único.

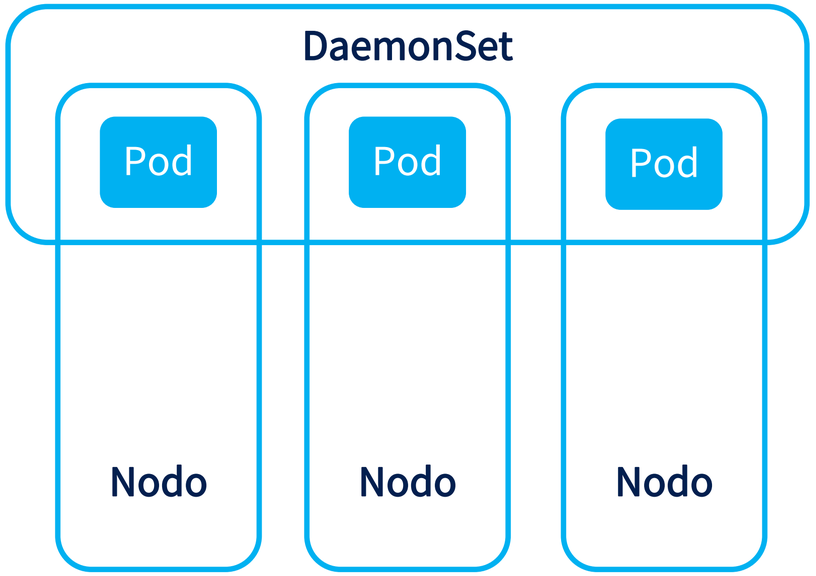


DaemonSets

Los DaemonSet son muy parecidos a los Deployments. La única diferencia con estos es que, en lugar de elegir nosotros el número de réplicas, se levanta automáticamente un pod en cada nodo. Siempre tendremos exactamente un pod de nuestro DaemonSet en cada nodo.

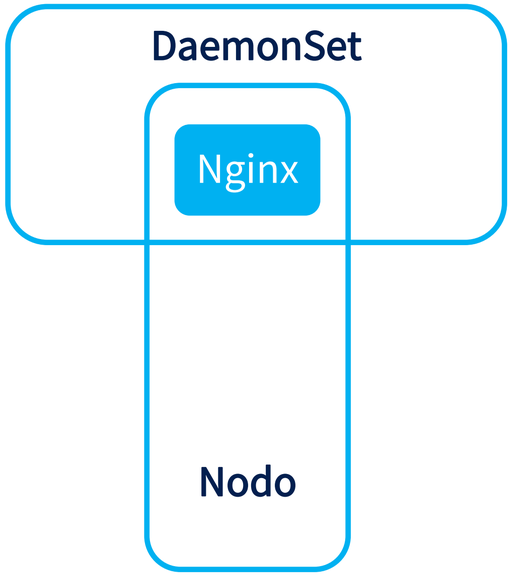
Si se une uno nuevo al clúster, automáticamente el DaemonSet levanta un Pod en él.

El nombre de DaemonSet viene porque los procesos de sistema en Linux se llaman daemons, y con este objeto lo que conseguimos es desplegar procesos de sistema comunes a todos los nodos del clúster.



DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
    name: nginx-daemonset
spec:
    selector:
        matchLabels:
            app: nginx-daemonset
    template:
        metadata:
            labels:
                app: nginx-daemonset
        spec:
            containers:
            - name: nginx
              image: nginx
              ports:
              - containerPort: 80
```

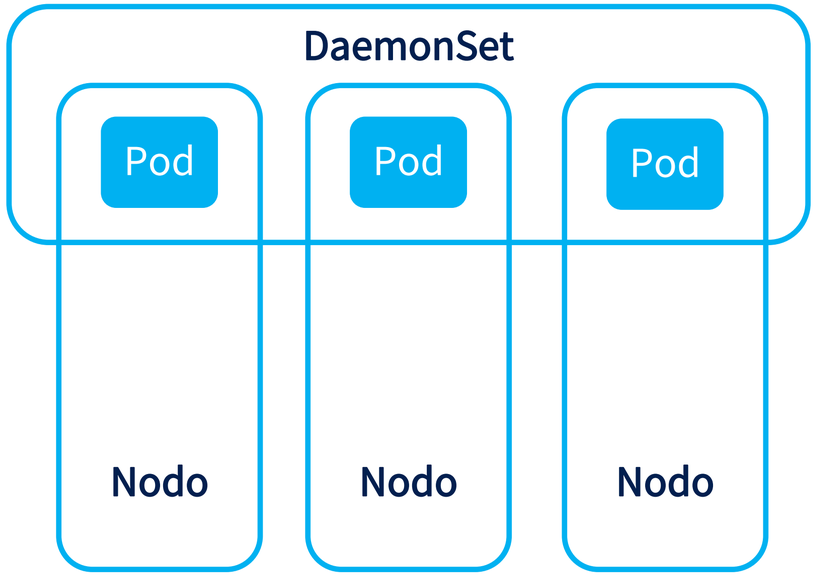


DaemonSets

No es muy útil desplegar un nginx en cada nodo. Los daemonSet no se suelen usar para desplegar aplicaciones, sino procesos necesarios en los nodos.

Por ejemplo, un contenedor que monitorice el estado de salud del nodo. O un contenedor que lea los logs de todos los demás contenedores que existen en el nodo, y los reenvíe a nuestro sistema de almacenamiento de logs. O que aplique normas de seguridad de red en el nodo.

Es posible excluir algunos nodos del despliegue de los DaemonSets, como por ejemplo el nodo master en el entorno de prueba. Lo veremos en siguientes módulos.

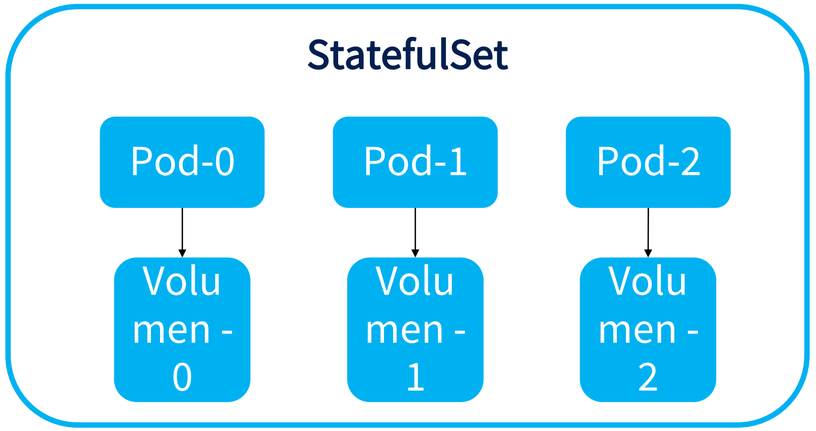


StatefulSet

Hasta ahora hemos estado hablando de aplicaciones **stateless** o sin estado.

Stateless significa que no almacenan información necesaria para la app en el disco. Una aplicación **stateless** almacena toda la información en otro sistema externo, por ejemplo una base de datos.

La ventaja de que toda la información esté en un sistema distinto es que no me tengo que preocupar por si una réplica se cae, ya que no necesito levantarla de nuevo en el mismo nodo ni con el mismo disco, sino que puedo levantarla en cualquier sitio libre siempre que tenga acceso a la base de datos. Mi servicio no notará ningún cambio.



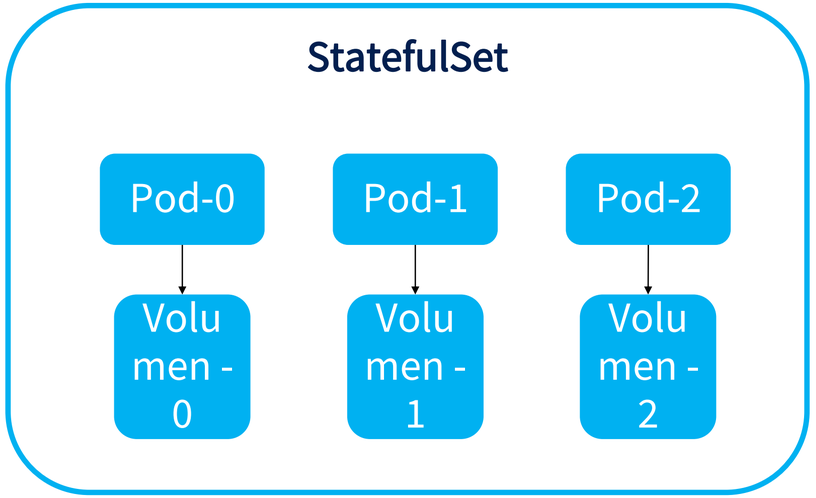
StatefulSet

Lo opuesto es que mi aplicación sea **stateful**. Esto es, que almacena información necesaria en el disco del mismo sistema.

Guardando la información de los usuarios que se han creado en una de mis réplicas en su disco, si esta falla todos esos usuarios no podrían usar mi app hasta que se levantase esa réplica es ese nodo con ese disco.

Lo inteligente en servicios web es que siempre sean stateless para que puedan recuperarse de fallos y escalar sin mayor complicación.

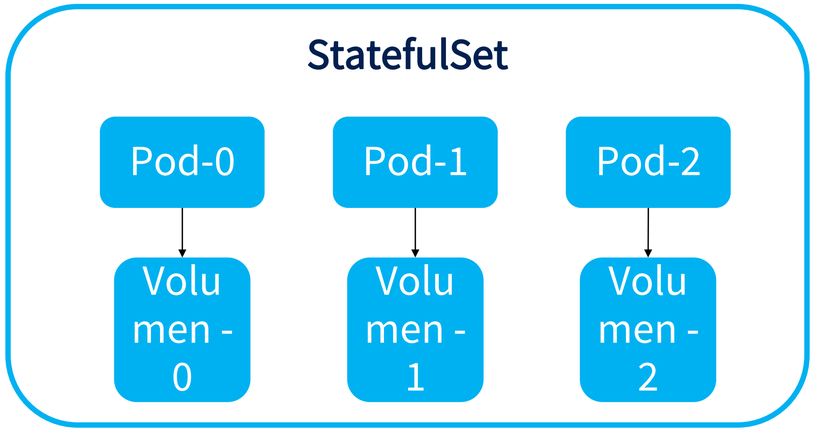
Pero esto no siempre es posible para todas las aplicaciones. Por ejemplo, podemos levantar una base de datos en Kubernetes (recuerda que Kubernetes vale para cualquier tipo de carga de trabajo). La base de datos **necesariamente necesita su propio disco para funcionar**, es donde residen sus datos y su valor.



StatefulSet

Los statefulsets proporcionan características que son necesarias para este tipo de aplicaciones stateful:

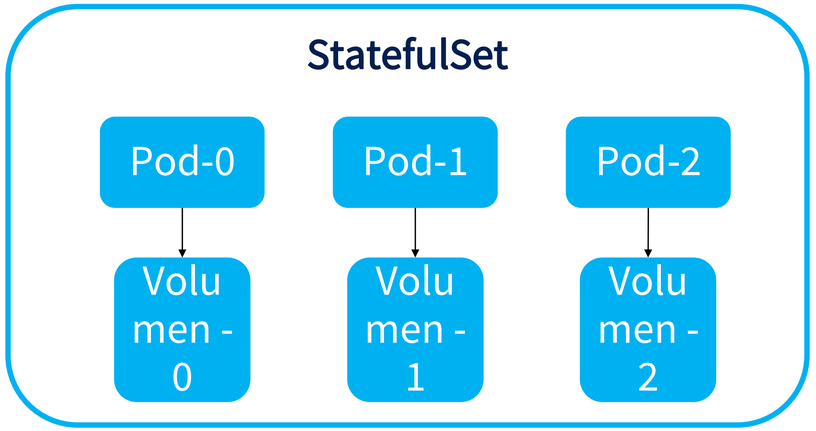
1. **Nombres** de red **únicos** para cada réplica
2. **Almacenamiento** persistente para cada una de ellas (opcional)
3. Un sistema de **despliegue ordenado** (cuando comience el despliegue de la réplica con id 1, Kubernetes asegura que la réplica con id 0 va a estar funcionando, y así sucesivamente)



La spec es similar al deployment, salvo que añade el campo obligatorio serviceName (relacionado con temas de red que veremos en el siguiente módulo)

StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
    name: nginx-statefulset
spec:
    selector:
        matchLabels:
            app: nginx-statefulset
    replicas: 3
    serviceName: nginx-statefulset
    template:
        metadata:
            labels:
                app: nginx-statefulset
        spec:
            containers:
            - name: nginx
              image: nginx
              ports:
              - containerPort: 80
```

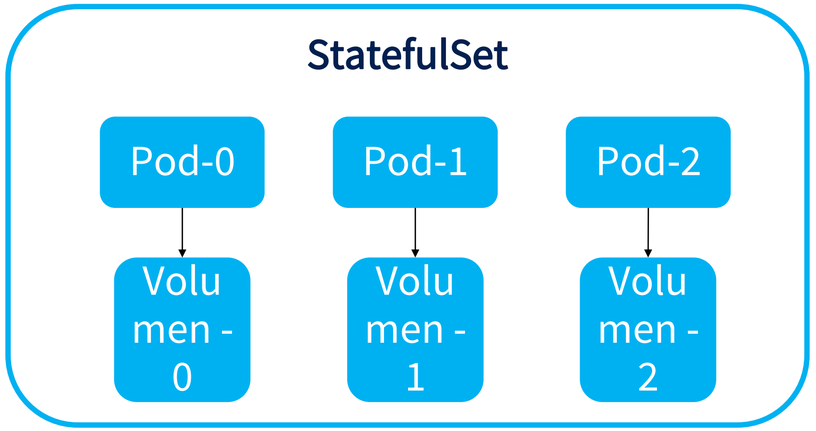


StatefulSet

Listando los pods vemos que existen diferencias claras con el Deployment:

1. Se han levantado las 3 replicas de una en una y en orden ascendente, en lugar de todas a la vez.
2. Los pod no tienen nombres aleatorios. Tienen un nombre fijo. Además, si eliminamos uno de ellos, se recrea con el mismo nombre.
3. Si actualizamos el statefulset para desplegar una imagen distinta por ejemplo, se actualizarán en orden descendente.

La forma de actualizar los statefulsets es configurable.



Jobs

Hay casos de uso que sólo requieren ejecutar una acción y luego liberar los recursos. O ejecutarla periódicamente.

Kubernetes también tiene soporte para este tipo de contenedores. Existen dos objetos que nos permiten ejecutar scripts o ejecutar cargas de trabajo puntuales de forma muy similar a como desplegamos los deployments o los statefulSet.

Un **Job** es un objeto de Kubernetes que despliega uno o más pods, **hasta que el proceso acabe correctamente**. Tiene capacidades para volver a lanzar el pod en caso de que ocurra cualquier error. Y cuando termina correctamente, no lo vuelve a recrear.

Jobs

```
apiVersion: apps/v1
kind: Job
metadata:
    name: hello-world-job
spec:
    template:
        spec:
            containers:
            - name: hello-world
              image: hello-world
            restartPolicy: Never
```

Jobs

Como ves es una definición muy similar a la de un deployment. En este caso no es necesario el uso de selector y labels.

Es necesario modificar el restartPolicy por defecto en el spec del pod. El valor por defecto es Always y no es aceptado en los Jobs, ya que el pod no se debe reiniciar cuando termine correctamente. Puedes cambiarlo a Never u OnFailure.

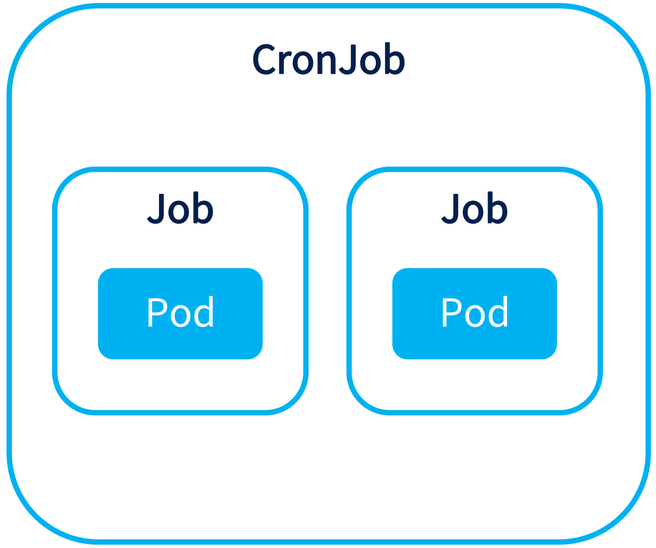
Cambiándolo a Never, el job se encargará de recrear el pod si falla hasta alcanzar un número máximo de intentos que por defecto es 6.

Existen varios parámetros que controlan el comportamiento del job en caso de fallos (número de reintentos, etc.) pero son un tema algo más avanzado que veremos más adelante.

CronJobs

El otro objeto de interés es el CronJob. Como se puede suponer, un CronJob se encarga de crear un job de forma periódica.

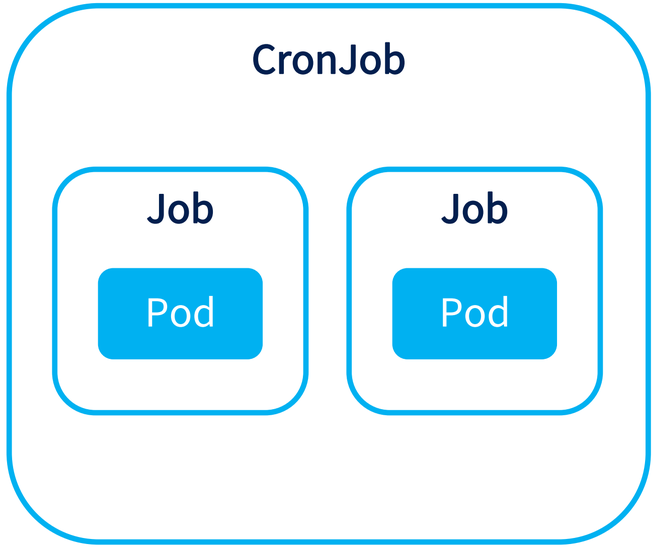
```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello-world-cronjob
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello-world
            image: hello-world
          restartPolicy: Never
```



CronJobs

Como mínimo, dentro del spec tiene:

- schedule: Una expresión en formato cron para definir el periodo,
- jobTemplate: La plantilla para los jobs que creará este cronjob. Efectivamente, dentro de este jobTemplate tendremos a su vez el template del pod que creará ese job.



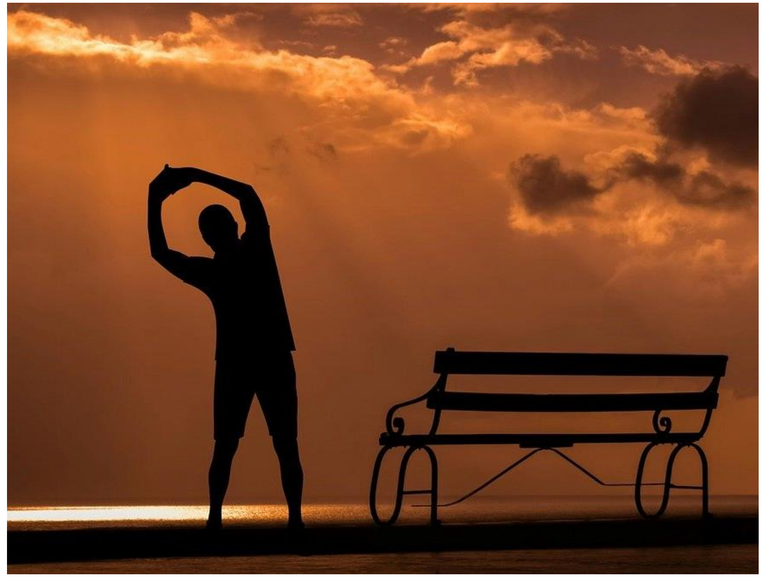
Vemos que el cronjob está creando un job nuevo cada minuto con un ID único. Y el job a su vez crea el pod que ejecuta hello-world.

Descansa 10 minutos

Hasta ahora hemos visto los 4 tipos de workloads que podemos usar en Kubernetes para ejecutar todo tipo de aplicaciones:

1. Deployments, que a su vez crean replicaSets
2. DaemonSets, para ejecutar contenedores en todos los nodos
3. StatefulSets, para aplicaciones imposibles de desplegar con los Deployments
4. Jobs y CronJobs, para cargas de trabajo puntuales

En lo poco que queda del módulo 2, vamos a ver cómo definir mejor nuestros pods para que estos se desplieguen de forma segura.



Mejorando nuestros Pods

Vamos a ver características fundamentales que podemos añadir a nuestros pods.

Éstas son útiles para cualquier pod que creemos, ya sea con un Deployment, DaemonSet, StatefulSet o Job.

1. Cómo asignar recursos (CPU y memoria)
2. Cómo conoces el estado de salud

Asignación de recursos

Asignar una cantidad de memoria y de tiempo de CPU a nuestros pods es una medida básica para que estos no fallen en su ejecución.

Hacerlo es de lo más sencillo, sólo hay que añadir un bloque de este tipo **dentro del YAML de cada contenedor:**

```
resources:
    requests:
        memory: 64M
        cpu: 250m
    limits:
        memory: 128M
        cpu: 500m
```

Asignación de recursos

Requests define la capacidad que Kubernetes reservará en el nodo para ese contenedor.

Si ningún nodo tiene capacidad suficiente, no se levantará. Así mismo, ningún otro pod del mismo nodo puede hacer uso de esos recursos reservados en requests.

Limits está pensado para proteger al nodo (y por tanto al resto de pods que se ejecutan en él), de posibles picos en el uso de CPU y memoria.

Por ejemplo, si por un error mi contenedor empieza a utilizar memoria RAM y llega a los 8G, que es toda la memoria que tiene mi nodo, éste fallará y todos los pods que están en él también.

Para eso, especifico que mi contenedor tiene un límite de 2G. Si quiere utilizar más, Kubernetes lo reiniciará automáticamente. Mi pod fallará por haber llegado al límite de memoria, pero habré salvado a todos los demás pods del nodo.

Asignación de recursos

Los requests y limits de **memoria** se pueden especificar con los formatos: 512M, 512Mi, 0.5G, 0.5Gi. Todos estos son equivalentes.

La **CPU** se mide en número de **cores** que puede utilizar. 0.5 significa que puede utilizar el 50% de un core. Es equivalente a 500m.

Cuidado con utilizar la unidad MB o GB en la memoria. No es aceptada por Kubernetes. La correcta es **Mi o Gi** (es similar a MB o GB, simplemente Mi y Gi representan potencias de 2 mientras que MB y GB representan potencias de 10, pero prácticamente representan los mismos valores). Lo mejor es utilizar M o G.

Asignación de recursos

Vamos a modificar nuestro deployment.yaml para ponerle requests y límites de CPU y memoria.

Añadimos este bloque dentro de la definición del contenedor. Es decir, al mismo nivel que el campo image o ports:

```
resources:
    requests:
        memory: 64M
        cpu: 100m
    limits:
        memory: 128M
        cpu: 200m
```

LivenessProbe

¿Qué pasa si mi aplicación está fallando pero el proceso dentro del contenedor sigue activo? Kubernetes no tendría ninguna manera de saber que tu contenedor está dando fallos y debería ser reiniciado. Si el proceso dentro del contenedor no ha terminado, es imposible conocer que está fallando.

Para eso tenemos otro campo dentro del spec de un pod: **livenessProbe**.

LivenessProbe define **cómo Kubernetes va a preguntar a tu aplicación que está funcionando correctamente**. Si esta prueba falla, Kubernetes entenderá que ese contenedor está fallando y lo reiniciará.

LivenessProbe

Hay varios tipos de pruebas:

- **Una petición HTTP**. Ésta es la más común. Kubernetes realiza una petición periódicamente, por ejemplo cada 30 segundos, al puerto y URL que especifiquemos. Si esta petición HTTP se responde con un 200, la aplicación está bien. Si es otro código distinto, la aplicación está fallando por algún motivo.

```
livenessProbe:
    httpGet:
      path: /healthz
      port: 8080
```

LivenessProbe

- **Ejecutar un comando dentro del contenedor**: Si este comando no acaba con éxito, la aplicación está fallando.

```
livenessProbe:
    exec:
      command:
      - cat
      - /tmp/healthy
```

LivenessProbe

- **Una conexión TCP**: Intenta conectarse a un puerto TCP. Si no lo consigue, entiende que la aplicación está caída.

```
livenessProbe:
    tcpSocket:
      port: 8080
```

LivenessProbe

Hay varios parámetros que podemos configurar en todos ellos:

initialDelaySeconds: Periodo inicial en el que no se realizan las pruebas, mientras la aplicación se levanta del todo (**importante**, porque con él conseguimos que nuestros pods no se reinicien porque nuestro código tarde mucho en levantarse del todo)

periodSeconds: Periodo para realizar la prueba. Por defecto es 10.

failureThreshold: Cuántos fallos de prueba se permiten antes de reiniciar el pod. Por defecto es 3

timeoutSeconds: Cuánto tiempo se espera a que el contenedor responda a cada prueba. Por defecto es 1.

LivenessProbe

Vamos a añadir una livenessProbe a deployment.yaml. Vamos a comprobar que en todo momento Nginx está respondiendo a peticiones http (la página welcome to Nginx que devuelve por defecto).

Lo añadimos de nuevo dentro del YAML de cada contenedor, al mismo nivel que image, ports o resources:

```
livenessProbe:
    httpGet:
        path: /
        port: 80
    initialDelaySeconds: 15
    periodSeconds: 10
```

LivenessProbe

Le hemos dado 15 segundos iniciales para que Nginx pueda levantarse correctamente, y especificado que queremos que compruebe el puerto TCP cada 10 segundos.

Vamos a inspeccionar los logs de un pod de este Deployment para comprobar que le está llegando una petición http al path “/” cada 10 segundos.

De esta forma tan sencilla conseguimos que nuestra aplicación se auto-repare en caso de errores en alguna de sus réplicas.

readinessProbe

Ahora mismo, el pod entra en estado Ready en el mismo instante en que comienza a ejecutarse el contenedor. El estado Ready significa que está lista para recibir peticiones.

Pero en la realidad, puede que la aplicación que hay dentro no esté todavía preparada para funcionar. Normalmente las aplicaciones tardan unos segundos en estar preparadas para funcionar correctamente.

Para esperar a que mi aplicación esté respondiendo correctamente antes de pasar a estado Ready y comenzar a procesar peticiones, está la ReadinessProbe.

readinessProbe

La definición es exactamente igual que LivenessProbe. Por ejemplo, Kubernetes se intentará conectar cada 10 segundos al puerto 80. Cuando lo consiga, significa que mi servidor está de verdad listo para responder peticiones reales.

Todos los parámetros de configuración son exactamente iguales:

```
readinessProbe:
    httpGet:
        path: /
        port: 80
    initialDelaySeconds: 15
    periodSeconds: 10
```

readinessProbe

No debemos confundir esto con el parámetro initialDelaySeconds de livenessProbe.

Éste parámetro simplemente controla que no de por fallo un pod que está iniciándose. Pero no impide que le lleguen peticiones.

La readinessProbe es básica porque impide que lleguen peticiones al pod (estado Ready) antes de que este sea capaz de responderlas.

¡Fin del módulo!

En esta última parte hemos aprendido a configurar de forma correcta nuestros pods:

- Asignándoles CPU y memoria
- Definiendo pruebas de vida para que un pod erróneo no procese peticiones y sea reiniciado en caso de error.

Con todo esto somos capaces de levantar cualquier tipo de aplicación de forma efectiva en Kubernetes.

Y todo simplemente definiendo objetos en YAML, sin haber programado nada ni instalado ningún servicio extra.

Extras

Páginas de la documentación oficial:

Deployments
DaemonSets
StatefulSets
Jobs
Cronjobs

Resources
LivenessProbe y ReadinessProbe

¡Gracias!