07.

Kubernetes

# Multi-Container Pods.

# Multicontainer Pods.

Kubernetes da la oportunidad de mejorar los contenedores con contenedores auxiliares con funcionalidad adicional.

Esto se traduce a poder crear pods con varios contenedores dentro que se pueden comunicar entre si.

***Exploramos*** *el fichero multiContainerPod.yml*
*vi multiContainerPod.yml*
***Crear*** *el Pod a partir del fichero*
*kubectl apply -f multiContainerPod.yml*
***Visualizar*** *los Pods*
*kubectl get pods -n paradigma*
***Accedemos*** *al contenedor de ubuntu y comprobamos que nos comunicamos con el de nginx*
*kubectl exec -it mimulticontainerpod -c ubuntu -n paradigma -- bash*
*curl localhost:80*

08.

Kubernetes

# Observability.

# Liveness and Readiness Probes.

https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-probes/

Kubernetes detecta problemas automáticamente y responde sin la necesidad de una configuración especializada pero a veces necesitamos un control adicional sobre como kubernetes determina el estado del contenedor y las sondas de kubernetes permiten personalizarlo. A partir de esta detección se pueden crear mecanismos mas sofisticados que los habituales para administrar el estado de los contenedores. Existen 2 **sondas**, liveness probe y readiness probe.

**Liveness Probe**

Kubernetes utiliza esta sonda para saber cuándo debe reiniciar un contenedor. Por ejemplo, las **liveness probe** podrían detectar un punto muerto, en el que una aplicación se está ejecutando, pero no puede progresar. Reiniciar un contenedor en ese estado puede ayudar a que la aplicación vuelva a estar disponible.

**Readiness Probe**

Kubernetes utiliza esta sonda para saber cuándo un contenedor está listo para empezar a aceptar tráfico. Una aplicación puede necesitar cargar grandes datos o archivos de configuración durante el arranque, o depender de servicios externos después del arranque. Un pod con contenedores que informan de que no están listos no recibe tráfico a través de los servicios de Kubernetes.

# Liveness.

***Exploramos*** *el fichero livenessPod.yml*

*vi livenessPod.yml*

***Crear*** *el Pod a partir del fichero*

*kubectl apply -f livenessPod.yml*

***Visualizar*** *los Pods*

*kubectl get pods -n paradigma*

***Creamos*** *el fichero que la sonda necesita que exista y vemos como deja de dar error*

*kubectl exec -it milivenesspod -n paradigma -- touch /tmp/live*

***Eliminamos*** *el fichero que la sonda necesita que exista y vemos como comienza a dar error*

*kubectl exec -it milivenesspod -n paradigma -- rm /tmp/live*

# Readiness.

***Exploramos*** *el fichero readinessPod.yml*

*vi readinessPod.yml*

***Crear*** *el Pod a partir del fichero*

*kubectl apply -f readinessPod.yml*

***Visualizar*** *los Pods*

*kubectl get pods -n paradigma*

***Creamos*** *el fichero que la sonda necesita que exista y vemos como deja de dar error*

*kubectl exec -it mireadinesspod -n paradigma -- touch /tmp/readi*

***Eliminamos*** *el fichero que la sonda necesita que exista y vemos como comienza a dar error*

*kubectl exec -it mireadinesspod -n paradigma -- rm /tmp/readi*

# Container Logging.

https://kubernetes.io/docs/concepts/cluster-administration/logging/

Al administrar contenedores, a veces es necesario obtener **logs** de los contenedores para obtener una idea de lo que está sucediendo dentro de un contenedor.

***Exploramos*** *el fichero logs.yml*
*vi logs.yml*
***Crear*** *el Pod a partir del fichero*
*kubectl apply -f logs.yml*
***Visualizar*** *los Pods*
*kubectl get pods -n paradigma*
***Visualizar*** *los logs del pod*
*kubectl logs milogspod -n paradigma*
***Visualizar*** *los logs de un contenedor dentro de un multicontainer*
*kubectl logs mimulticontainerpod -c ubuntu -n paradigma*
***Visualizar*** *los logs del pod utilizando el comando* ***describe***
*kubectl describe pod milogspod -n paradigma*

# Monitoring Aplications.

https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/

La monitorización es una parte importante de la administración de cualquier infraestructura de aplicación.
A continuación os muestro como ver el uso de recursos de pods y nodos usando el comando **kubectl top**

Desplegamos varios pods con diferentes usos de recursos para ver después como mostrarlo.

***Habilitamos*** *el servidor de métricas de minikube*
*minikube addons enable metrics-server*
***Exploramos*** *el fichero resourceConsumerBig.yml*
*vi resourceConsumerBig.yml*
***Crear*** *el Pod a partir del fichero*
*kubectl apply -f resourceConsumerBig.yml*
***Exploramos*** *el fichero resourceConsumerSmall.yml*
*vi resourceConsumerSmall.yml*
***Crear*** *el Pod a partir del fichero*
*kubectl apply -f resourceConsumerSmall.yml*

A continuación ejecutamos los comandos para ver los datos de uso de recursos del cluster

***Vemos** el uso de recursos de todos los pods de un Namespace*

*kubectl top pods -n paradigma*

***Vemos** el uso de recursos de un pod en concreto*

*kubectl top pods miresourceconsumerbig -n paradigma*

***Vemos** el uso de recursos de todos los pods del cluster*

*kubectl top pods --all-namespaces*

09.

Kubernetes

# Pod Design

# Labels.

https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Las **labels** son pares **clave/valor** que se adjuntan a los objetos, como los pods. Se utilizan para especificar los atributos de identificación de los objetos que son significativos y relevantes para los usuarios. Pueden utilizarse para organizar y seleccionar subconjuntos de objetos. Pueden adjuntarse a los objetos en el momento de su creación y, posteriormente, añadirse y modificarse en cualquier momento. Cada objeto puede tener definido un conjunto de labels de clave/valor. Cada clave debe ser única para un objeto determinado.

***Exploramos*** *el fichero labelProduccionPod.yml y labelDesarrolloPod.yml*
*vi labelProduccionPod.yml y vi labelDesarrolloPod.yml*
***Crear*** *los Pods a partir del fichero*
*kubectl apply -f labelProduccionPod.yml*
*kubectl apply -f labelDesarrolloPod.yml*
***Visualizar*** *los Pods mostrando las labels*
*kubectl get pods -n paradigma –show-labels*
***Exploramos*** *el pod y podemos ver las labels también*
*kubectl describe pod milabelproduccionpod -n paradigma*
*kubectl describe pod milabeldesarrollopod -n paradigma*

# Selectors.

https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

Los **selectors** se utilizan para identificar y seleccionar un grupo específico de objetos utilizando sus **labels**. Una forma de utilizar los **selectors** es usarlos con **kubectl get** para recuperar una lista específica de objetos. Podemos especificar un selector utilizando el flag **l**

*kubectl get pods -n paradigma -l app=miapp*

También podemos encadenar varios selectores utilizando una lista delimitada por comas

*kubectl get pods -n paradigma -l app=miapp,entorno=desarrollo*

# Annotations.

https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/

Las **annotations** son similares a las **labels**, ya que pueden utilizarse para almacenar metadatos personalizados sobre los objetos. Sin embargo, a diferencia de las labels, las annotations no pueden utilizarse para seleccionar o agrupar objetos. Las herramientas externas pueden leer, escribir e interactuar con las annotations.

***Exploramos*** *el fichero annotationPod.yml*
*vi annotationPod.yml*
***Crear*** *los Pods a partir del fichero*
*kubectl apply -f annotationPod.yml*

# Deployment.

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

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.

- Que no haya caída del servicio
- Tolerancia a fallos
- Escalabilidad de pods

***Exploramos*** *el fichero deploymentPod.yml*
*vi deploymentPod.yml*

Vemos que aparecen nuevas etiquetas

***spec.replicas*** *indica el número de pods que queremos ejecutar*
***spec.template*** *define los pods que se crearán dentro del deployment*
***spec.selector*** *el deployment gestionará todos los pods cuyas* ***labels*** *coincidan con ese selector*

***spec.strategy*** *Indica el modo en que se realiza una actualización del Deployment:* ***Recreate****: elimina los Pods antiguos y crea los nuevos.* ***RollingUpdate****: actualiza los Pods a la nueva versión*

***Crear*** *el Deployment a partir del fichero*

*kubectl apply -f deployment.yml*

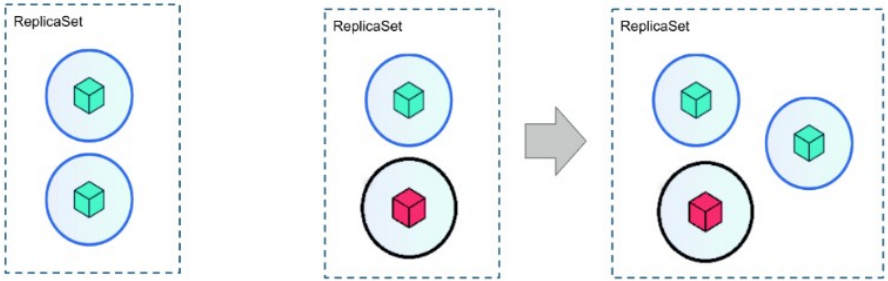
***Visualizar*** *los Deployments*

*kubectl get deploy -n paradigma*

***Visualizar*** *los Pods que ha generado el deployment*

*kubectl get pods -n paradigma*

Si se elimina uno de los pods creados, automáticamente Kubernetes se encarga de crear otro pod para mantener en todo momento el número de réplicas que hemos especificado.



# Jobs.

https://kubernetes.io/docs/concepts/workloads/controllers/jobs-run-to-completion/

Kubernetes proporciona la capacidad de ejecutar fácilmente cargas de trabajo en un clúster distribuido, pero no todas las cargas de trabajo deben ejecutarse constantemente. Con los **Jobs**, podemos ejecutar cargas de trabajo de contenedores hasta que se completen y luego cerrar el contenedor.

***Exploramos*** *el fichero job.yml*
*vi job.yml*
***Crear*** *el job a partir del fichero*
*kubectl apply -f job.yml*
***Visualizar*** *los Jobs*
*kubectl get jobs -n paradigma*
***Visualizar*** *los pods que ha generado el job*
*kubectl get pods -n paradigma*
***Visualizar*** *los logs del pod*
*kubectl logs mijob -n paradigma*

Vamos a explicar 2 etiquetas que hemos visto en la definición del Job y que se utilizan para gestionar los fallos de un Pod.

**restartPolicy** indica la política de reinicio que se va a aplicar cuando el Pod falla. En el caso de los jobs existen 2 políticas.

*OnFailure* el Pod permanece en el mismo nodo y el contenedor se vuelve a ejecutar. Por lo tanto, tu aplicación debe poder gestionar el caso en que se reinicia de forma local.

*Never* el controlador del Job arranca un nuevo pod. Esto quiere decir que tu aplicación debe ser capaz de gestionar el caso en que se reinicia en un nuevo pod. En particular, debe ser capaz de gestionar los ficheros temporales, los bloqueos, los resultados incompletos, y cualquier otra dependencia de ejecuciones previas.

**backoffLimit** indica el número de reintentos que queremos que se realicen antes de considerar el Job como fallido.

# CronJobs.

https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/

Los **CronJobs** nos permiten hacer lo mismo que con un job, pero volver a ejecutar la carga de trabajo regularmente de acuerdo con un cronograma.

***Exploramos*** *el fichero cronjob.yml*
*vi cronjob.yml*
***Crear*** *el job a partir del fichero*
*kubectl apply -f cronjob.yml*
***Visualizar*** *los CronJobs*
*kubectl get cronjobs -n paradigma*
***Visualizar*** *los Jobs que ha generado el cronjob*
*kubectl get jobs -n paradigma*
***Visualizar*** *los pods que ha generado el job*
*kubectl get pods -n paradigma*
***Visualizar*** *los logs del pod*
*kubectl logs microjob -n paradigma*

A continuación vemos la sintaxis del Schedule

```
# ┌──────────── minute (0 - 59)
# │ ┌──────────── hour (0 - 23)
# │ │ ┌──────────── day of the month (1 - 31)
# │ │ │ ┌──────────── month (1 - 12)
# │ │ │ │ ┌──────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                   7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * *
```

Por ejemplo, la línea siguiente establece que la tarea debe iniciarse todos los viernes a medianoche, así como el día 13 de cada mes a medianoche.

*0 0 13 * 5*