Kubernetes: primeros pasos
Pods

KEEPCODING
Tech School

Primeros Pasos en Kubernetes

- En esta sección:
  - Conceptos Básicos (yaml)
  - Objetos
    - Namespaces
    - Pods
    - Controllers
  - Más sobre kubectl
  - Ejercicios

¿Qué podíamos hacer con Docker + Compose?

- Lanzar / Crear contenedores (docker) para ejecutar aplicaciones.
- Orquestar múltiples contenedores para que trabajen e interactúen entre sí.
- Crear imágenes de docker

Conceptos: "imágenes", "contenedores".

¿Qué podemos hacer con Kubernetes?

- Orquestar / Lanzar **pods** que ejecuten nuestras aplicaciones e interactúen entre sí.

- Crear **servicios** para exponer los pod a otros pods o al exterior.

- Crear otros muchos tipos de **objetos** para integrar nuestras aplicaciones en entornos productivos.

- Utilizar ficheros "**yaml**" para declarar todos nuestros objetos.

Conceptos: "yaml", "objetos", "pods", "servicios"

Conceptos básicos: YAML

- Lenguaje declarativo “Human Readable”
- Indentación por espacios
- Comentarios con almohadillas
- Listas con guiones
- Los strings pueden ir sin comillas, pero podemos usarlas si queremos.

XML vs JSON vs YAML

XML

```xml
<person>
  <firstname>Tom<firstname>
  <lastname>Smith<lastname>
  <year>1982<year>
  <favorites>
    <value>tennis</value>
    <value>golf</value>
  </favorites>
</person>
```

JSON

```json
{
  "person": {
    "firstname": "Tom",
    "lastname": "Smith",
    "year": 1982,
    "favorites": ["tennis", "golf"]
  }
}
```

YAML

```yaml
person:
  firstname: Tom
  lastname: Smith
  year: 1982
  favorites:
    - tennis
    - golf
```

Fuente: http://sangsoonam.github.io/2017/03/13/yaml-vs-json.html

Listas (secuencias / arrays)

Para crear una secuencia, lista o “array”, usa el “-”.
La indentación es importante, indica a qué elemento pertenece.

YAML

```yaml
- foo
- bar
-
  - baz
   - qux
```

JSON

```json
[
  "foo",
  "bar",
  [
    "baz",
    "qux"
  ]
]
```

Map

Para crear elementos de tipo <key><value> usaremos “:”

YAML

```yaml
foo: value
bar:
  - baz
  - qux
```

JSON

```json
{
  "foo": "value",
  "bar": [
    "baz",
    "qux"
  ]
}
```

Multilínea

Para introducir en un elemento un contenido multilínea podemos usar el símbolo “|”. Hay otros modificadores interesantes, como ">".

YAML

```yaml
YAML: |
  YAML Ain't Markup Language

  YAML is a human friendly data serialization

  standard for all programming languages.
```

JSON

```json
{
  "YAML": "YAML Ain't Markup Language\n\nYAML is a human friendly data serialization\nstandard for all programming languages."
}
```

Comentarios

Utilizamos la almohadilla para los comentarios.
La indentación también es importante.

YAML

```yaml
# This is a comment
YAML: |
  YAML Ain't Markup Language
```

JSON

```json
{

  "YAML": "YAML Ain't Markup Language"

}
```

Documentación oficial

KEEPCODING
Tech School

Documentación oficial de Kubernetes

- Muy útil y de calidad.
- Bien organizada
- Buscador excelente, multitud de ejemplos.
- Es necesario saber navegar por los documentos y utilizar el buscador para las certificaciones oficiales (CKAD, CKA, CKS).

- Los más importantes
  - > Página principal:
    - https://kubernetes.io/docs/home/
  - > API Reference docs, por ejemplo para 1.31:
    - https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.31/

Objetos en Kubernetes

KEEPCODING
Tech School

Objetos de Kubernetes

- Son abstracciones que representan el estado del sistema.
- Definen el **estado deseado** del sistema ("*desired state*" / "*record of intent*")
- Muchos tipos de objetos tienen su especificación (estado deseado) y un **estado actual** (desired state & status), proporcionado por kubernetes (`kubectl describe xxx`).
- Son entidades persistentes (definiciones almacenadas dentro de la base de datos interna (etcd)).
- Podemos definirlos mediante YAML

Objetos de Kubernetes

Campos requeridos:

- **apiVersion**: Qué versión del API
- **kind**: Tipo de objeto
- **metadata**: Permite identificar unívocamente al objeto

Cada tipo de objeto (**en cada versión de la API**) tiene sus propios parámetros, características y funcionalidades.

```yaml
apiVersion: apps/v1 # Version de la API (va cambiando)
kind: Deployment  # TIPO: Deployment
metadata: # Metadatos del Deployment
  name: nginx-deployment
spec: # Specificacion del DEPLOYMENT
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # indica al controlador que ejecute 2 pods
  template:
    metadata: # Metadatos del POD
      labels:
        app: nginx
    spec: # Especificación del POD
      containers: # Declaración de los contenedores del POD
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80
```

API Reference: https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.31/

Objetos de Kubernetes

- Metadatos
  - **Name** : Nombre del recurso
  - **Namespace** : Namespace asociado al recurso (opcional en el yaml, pero mandatory a nivel de objeto, **OJO**!)
  - **Labels**: Etiquetas
  - **Annotations**: key-value map, se suelen usar para modificar el comportamiento de otros componentes o de kubernetes

Objetos de Kubernetes

- Labels (etiquetas)
  - Son metadatos del tipo key-value que podemos adjuntar en los objetos de kubernetes (como por ejemplo los Pods).
  - Pensado para identificar recursos por información que pueda ser útil para los usuarios.
  - No están pensados para almacenar información.
  - Máximo 63 caracteres, han de empezar y acabar por un carácter alfanumérico y pueden contener '/', '.' y '-'
  - Se pueden filtrar con kubectl
  - Suelen ir de la mano de los **selectores**

```
$ kubectl get pods -l environment=production,tier=frontend
$ kubectl get pods -l 'environment in (production)'
```

Objetos de Kubernetes

- Annotations
  - Similares a los labels (metadatos de tipo key=value), pero no se suelen utilizar para identificar los recursos en sí, sino para incluir información que pueda ser usada por otros componentes o procesos.
  - Pueden almacenar mucha más cantidad de información.

```
$ kubectl annotate pods foo description='my frontend'
```

Namespaces

KEEPCODING
Tech School

Namespaces

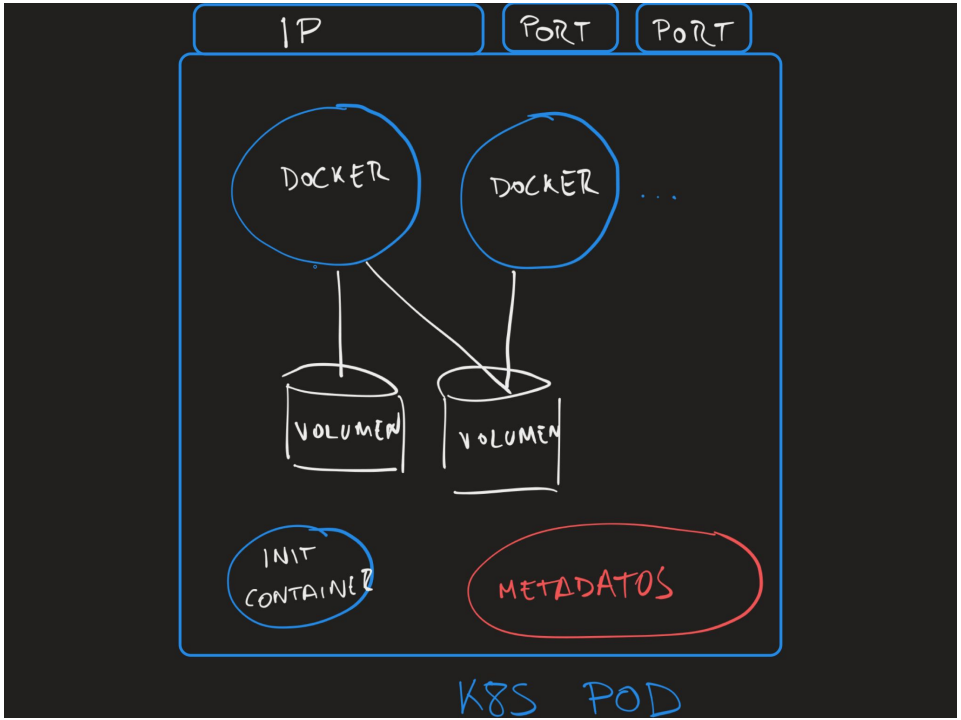
- Permite aislamiento de grupos de recursos dentro de un cluster. Proporciona a los recursos su espacio de nombres (para identificarlos).
- Nombres de recurso únicos dentro del mismo namespace.
- Proporciona separación lógica de recursos (multitenancy).
- Útiles para utilizarse en entornos con muchos usuarios distribuidos entre equipos o proyectos.
- Podemos tener distintas aplicaciones separadas o múltiples entornos de la misma aplicación.
- Existen 2 namespaces por defecto: **default** y **kube-system**.
- Los objetos de Kubernetes pueden ser “namespaced” o “non-namespaced”.

```
$ kubectl get ns
$ kubectl create ns test
$ kubectl -n test get pods
$ kubectl -n kube-system get pods
$ kubectl api-resources --namespaced=true
$ kubectl api-resources --namespaced=false
```

Pods

KEEPCODING
Tech School

PODS



Pods

- Basic building block / Unidad de ejecución básica en Kubernetes.
- Representa procesos (contenedores) que se ejecutan en el clúster.

- Encapsula **uno o varios contenedores** que corren en el mismo nodo, **compartiendo recursos de almacenamiento y de red (comparten la misma IP y volúmenes)**.

- Al igual que los contenedores de aplicaciones individuales, se les considera **entidades efímeras**.
- **Por lo general diseñaremos PODs pero no crearemos PODs directamente.**

- https://kubernetes.io/docs/concepts/workloads/pods/

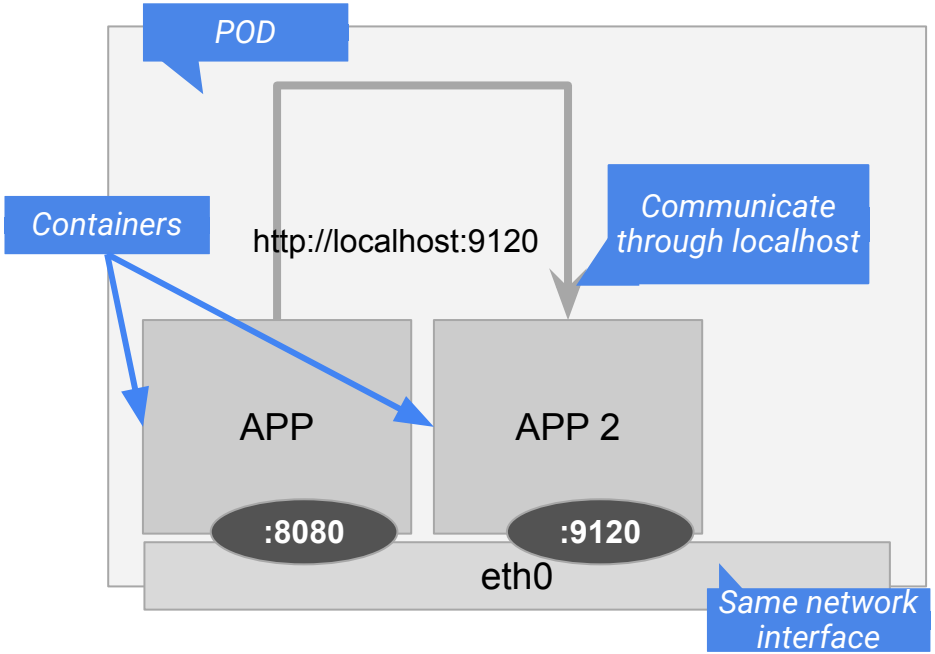
Pods - Ciclo de vida

- Ciclo de vida:
  - Cuando creamos un objeto pod via YAML simplemente lo declaramos dentro del cluster (en un namespace).
  - Es Kubernetes el encargado de elegir en qué nodo tendrá que correr, ejecutarlo, reiniciarlo, eliminarlo, etc. Kubernetes se encarga del ciclo de vida de los objetos.
  - Un pod define uno o varios "containers" (repetimos!).
  - Fases del ciclo de vida: Pending, Init, Running, Succeeded, Failed, etc.
  - Scheduling solo ocurre una vez en la vida del pod (mover un pod a otro nodo quiere decir destruirlo y crear uno nuevo en otro).
  - El reinicio de un pod consiste en destruirlo y dejar que el sistema cree uno nuevo (*)

- **PRÁCTICA**: get pod -w + crear un deployment + destruir alguno de los pods.

Networking en un POD

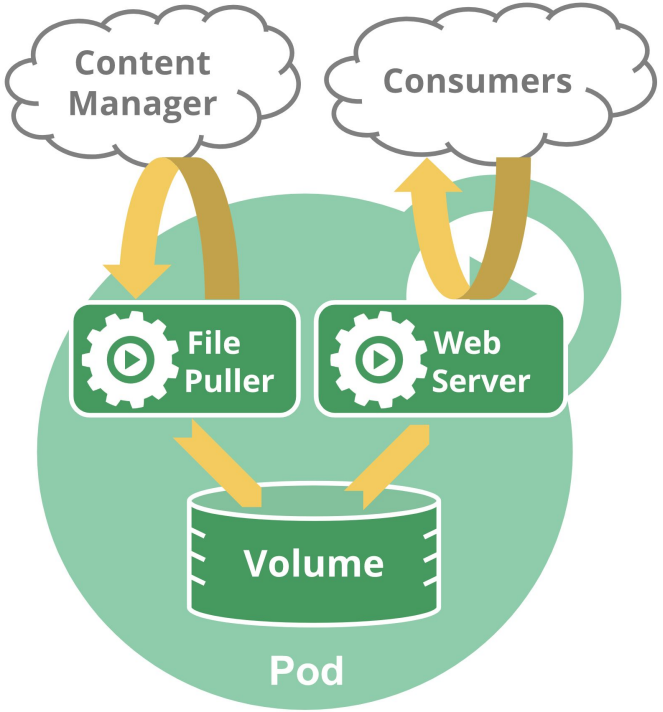
- A cada pod se le asigna una dirección IP única (de la "*overlay network*").
- Todos los contenedores dentro del POD comparten el namespace de red, incluyendo la dirección IP y los puertos de red.
- Los contenedores dentro de un pod pueden comunicarse con otros directamente usando “localhost”.
- Hay que coordinar el uso de los puertos de red disponibles.
- Si se lanza en modo "**HostNetwork: true**" el pod utiliza el interfaz de red real del host (OJO!)




```
$ kubectl get pod -o=wide
$ kubectl describe pod xxx
```

Almacenamiento en un POD

- Un Pod puede tener varios **Volúmenes** de almacenamiento compartidos.
- Todos los contenedores en el Pod pueden acceder a esos volúmenes compartidos, de esta manera gracias a ellos podemos compartir datos entre los contenedores del Pod.
- Los volúmenes también pueden ser de tipo persistente, de tal forma que si el contenedor se reinicia nos puede valer para persistir información.



https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/

Almacenamiento en un POD

- Objetos relacionados con el almacenamiento:
  - **Volumen** → Definidos a nivel de POD. Pueden ser efímeros o persistentes, dependiendo del tipo elegido. Parecidos a los volúmenes de docker. **No son objetos de Kubernetes**, sino parte de la especificación del POD.
  - **Persistent Volume (PV)** → Volumen persistente, ciclo de vida manejado por Kubernetes. Puede estar fuera del cluster, en un servicio de storage externo.
  - **Persistent Volume Claim (PVC)** → Petición de PV para creación automática. Ciclo de vida manejado por Kubernetes.

- Los volúmenes habrán de **montarse** en los contenedores para poder ser utilizados.

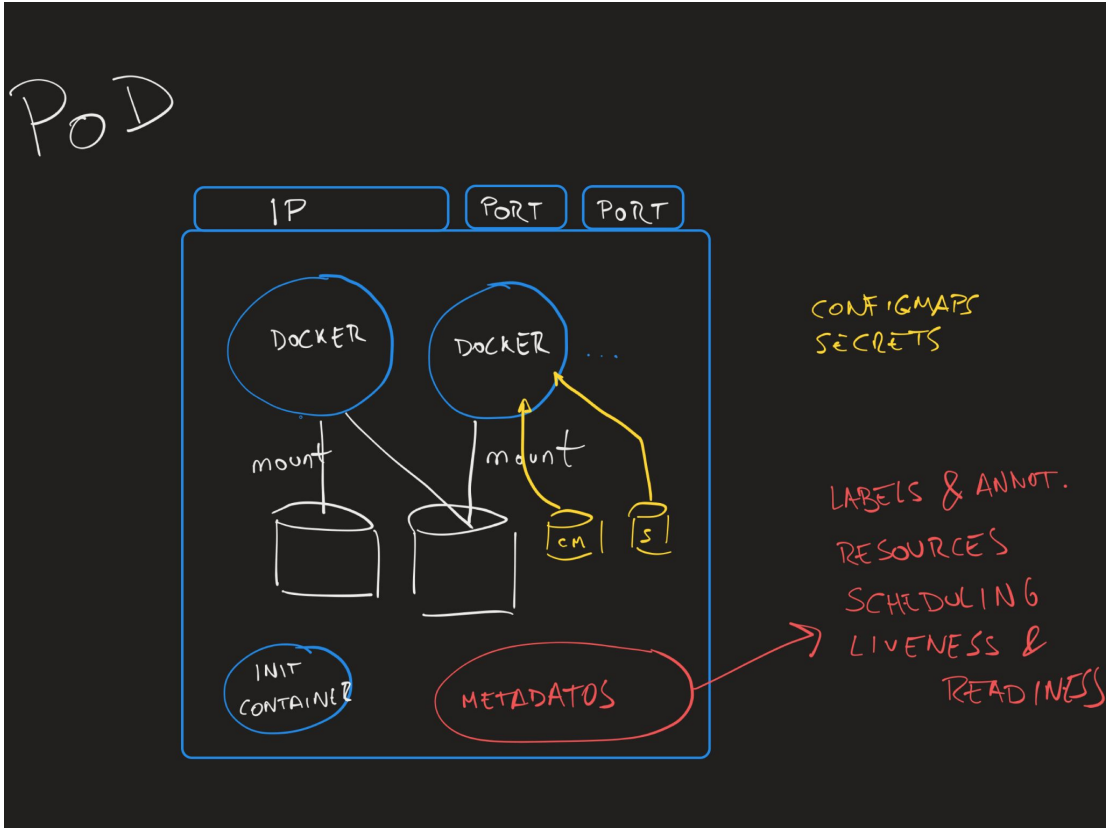
Configuración de los contenedores de los PODs

- Podremos dotar de configuración a los contenedores de los PODs mediante **ConfigMaps** y **Secrets**.
- A través de ConfigMaps y Secrets podremos dotar a los contenedores de ficheros de configuración, variables de entorno, etc.
- Similar a cuando en Docker utilizamos bind-mounts o variables de entorno.

PODS Funcionalidad

- En la definición de nuestros Pods podremos incluir:
  - **Init Containers** → Se lanzan antes de los contenedores principales. Han de terminar para poder lanzar los contenedores principales.
  - Toda la relacionada con lanzar contenedores vista en docker: entrypoint, cmd, usuario, imagen, volúmenes (persistencia), variables de entorno, control CPU y memoria, política de reinicio, etc.
  - Networking: puertos (informativos), usar red del host.
  - Autocuración y **chequeo** estado (liveness & readiness probes)
  - Control **scheduling**: NodeSelector / Affinity rules / Tolerations
  - Otras opciones: configuración DNS, /etc/hosts, …

Visión de un POD



Pod Spec

- **`containers`** → **definición contenedores**
- `dnsConfig`→ control DNS
- `dnsPolicy`→ control DNS
- `hostAliases`→ /etc/hosts
- **`hostNetwork`** → equivalente a net: host en docker
- `hostName`→ darle un hostname específico al pod
- `initContainers`→ definición init containers (**misma spec que containers**).
- `nodeName` → scheduling
- `nodeSelector` → scheduling
- `prirorityClassName` → tipo de prioridad
- **`restartPolicy`** → política reinicio (default = always)
- `schedulerName`→ scheduling
- `securityContext`→ seguridad
- `serviceAccountName` → seguridad (RBAC)
- `subdomain`→ control DNS
- `tolerations` → scheduling
- **`volumes`** → declaración volúmenes

¿Reconocemos opciones de docker?

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.31/#podspec-v1-core

Container Spec

- `args` → equivalente al CMD en docker.
- `command` → equivalente al ENTRYPOINT.
- `env` → variables de entorno.
- `envFrom` → variables de entorno.
- `image` → imagen del docker.
- `imagePullPolicy` → política de descarga.
- `livenessProbe` → chequeos.
- `name` → nombre del contenedor.
- `ports` → detalles de puertos que usará el contendor (descriptivo).
- `readinessProbe` → chequeos.
- `resources` → control de CPU y memoria.
- `volumeMounts` → puntos de montaje volúmenes.
- `workingDir` → cambiar el directorio de ejecución de la imagen (WORKDIR).

¿Reconocemos opciones de docker?

¿Creéis que faltan settings interesantes que conocemos de docker (por ejemplo -it)? Seguro que sí! → mirad la spec oficial.

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.31/

Ejemplo de un POD

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! &&
sleep 30']
```

```
$ kubectl apply -f pod.yaml
$ kubectl get pods
$ kubectl describe pod myapp-pod
$ kubectl logs myapp-pod
```

Vemos alguna similitud con un 'docker run'?

Usa `kubectl run xxx --image=nginx --dry-run=client -o=yaml` para crear el esqueleto ;-)

[Ejemplos de kubectl run]

Ejemplo de un POD

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod2
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c']
    args:
      - |
        echo 'Hello Kubernetes!'
        sleep 30
```

```
$ kubectl apply -f pod.yaml
$ kubectl get pods
$ kubectl describe pod myapp-pod
$ kubectl logs myapp-pod
```

Ejemplo de un POD (log contínuo)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod3
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox

    command: ['sh', '-c']
    args:
      - |
        while true; do
          echo "Hello Kubernetes"
          sleep 1
        done
```

```
$ kubectl apply -f pod.yaml
$ kubectl get pods
$ kubectl describe pod myapp-pod
$ kubectl logs -f myapp-pod
```

Variables de entorno

- Se definen en la sección env de los containers.
- Se pueden cargar desde fuentes externas (ConfigMaps, Secrets)

```
kubectl run nginx2 --image=nginx --env VAR1=valor1 --env VAR2=valor2
--dry-run=client -o=yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: nginx2
  name: nginx2
spec:
  containers:
  - name: nginx2
    env:
    - name: VAR1
      value: valor1
    - name: VAR2
      value: valor2
    image: nginx
```

Init Containers

- Se lanzan antes que el contenedor principal
- Han de terminar de forma satisfactoria.
- Pueden usar una imagen diferente a la del contenedor principal.
- Muy útiles para realizar comprobaciones, inicializar o preparar el entorno.

- Ejemplos de PODs con Init Containers:
  - https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#init-containers-in-use

Init Containers

- DEMO!

Chequeo de los PODs

- Kubernetes monitoriza y cuida de nuestros PODs mediante 2 métodos:
  - Liveness Probe: ¿Da señales de vida?
    - Sí → Ok.
    - No → Se reinicia el pod.
  - Readiness Probe: ¿Está preparado para procesar tráfico?
    - Sí → El pod se añade como endpoint en los servicios asociados.
    - No → El pod se quita de la lista de endpoints en los servicios asociados.

Liveness probe

- Si no pasa el check Kubernetes reiniciará el POD
- Pueden ser checks HTTP, TCP, comandos, etc.
- Definido a nivel de container!
- Si el POD necesita tiempo durante el arranque añade un '`startupProbe`'

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/liveness
    args:
    - /server
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

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

Readiness probe

- Si no pasa el check, los servicios de Kubernetes no mandarán tráfico al POD.
- Pueden ser checks HTTP, comandos, etc.
- Definido a nivel de container!
- Campos:
  - tcpSocket
  - exec
  - httpGet
  - … (spec aquí)

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: readiness
  name: readiness-http
spec:
  containers:
  - name: readiness
    image: k8s.gcr.io/liveness
    args:
    - /server
    readinessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

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

Liveness & Readiness probe

- DEMO!

NodeSelector

- Ofrece el nivel más básico de scheduling (programación) de PODs.
- NodeSelector nos permite indicar en qué tipo de host debería correr el **POD**.
- Se relaciona con los **labels** de los nodos.

- Definido a nivel de POD!

Nota: También existe NodeName, pero es tan restrictivo que casi no se utiliza.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

NodeSelector

- Demo!

Estado de los PODs

- Un POD comienza su ciclo de vida en estado (phase) **Pending** hasta que se lanza. De ahí pasa a **Init** donde se inicializa, descargan imágenes, se prepara entorno de red y almacenamiento y se lanzan los "init containers" (si los hubiera).
- Tras la fase init se lanzan los contenedores principales y se pasa a la fase **Running (X/Y)**, donde X es el número de contenedores del POD ya **Ready** e Y es el número de contenedores totales del pod, por ejemplo:

```
NAME        READY     STATUS    RESTARTS   AGE
myapp-pod   1/1       Running   0          9m
```

- Si el contenedor termina pasará a **Succeeded** o **Failed**

Estado de los PODs

- Para analizar el por qué de un estado que no esperamos utilizamos `'kubectl describe POD'` ó `'kubectl logs -c container_name POD'` donde podremos ver los detalles (state & phase) así como las razones asociadas.

- Los contenedores de dentro del POD también tienen sus estados:
  - Waiting (se está inicializando)
  - Running (contenedor en ejecución)
  - Terminated (contenedor finalizado)
  - Ready (true o false)

PODs: Resumen inicial

- Basic building block → uno o varios contenedores.
- Funcionalidades más importantes:
  - init containers
  - liveness y readiness
  - scheduling básico (NodeSelector)
  - gestión de recursos (cpu / memoria) → lo veremos más adelante

- DEMOS:
  - Init containers
  - Liveness & Readiness probes
  - NodeSelector

Controladores de Pods

KEEPCODING
Tech School

Trabajando con Pods

- Realmente, rara vez trabajaremos directamente con Pods en Kubernetes. Los Pods están diseñados para ser efímeros y dispensables.
- **Un Pod no puede “auto-curarse”**, si un Pod por alguna razón falla y es eliminado, no volverá a ejecutarse automáticamente en ningún momento, sin embargo para poder dar capacidades de resiliencia y auto-curación Kubernetes usa un nivel de abstracción superior llamado “Controller”.
- Un **Controller** puede crear y manejar varios Pods directamente por nosotros, encargándose de la replicación, del versionado y otorgando de las capacidades de autocuración. Por ejemplo, si un nodo falla, Kubernetes a través del controller automáticamente colocará ese Pod (uno nuevo en realidad) en otro nodo.
- Ejemplos de Controllers:
    - **Deployment**
    - **StatefulSet**
    - **DaemonSet**

Comando kubectl

KEEPCODING
Tech School

kubectl

- Comandos kubectl más comunes:
  - **`get / describe`** → fundamentales
  - **`config`** → manejar configuración y contextos de kubectl
  - **`logs`** → acceso a logs de los pods / contenedores
  - **`run`** → para crear un pod directamente (muy útil con --dry-run=client -o=yaml)
  - **`exec`** → similar a docker exec ('--' para separar el comando del resto de opciones)
  - **`(create / apply / delete) -f fichero.yaml`**
  - **`edit`** → modifica recursos (abre un editor con el yaml obtenido del sistema)
  - **`scale`** → cambia número de réplicas (para deployments, statefulsets, etc).
  - **`top <pod>`** : obtiene métricas del pod (requiere metrics server)
  - **`label / annotate / taint`** → manejar labels, annotations y taints
  - ---
  - **`create <resource>`** : crea directamente algunos tipos de objetos
  - **`expose`** : crea servicios directamente.
  - **`cluster-info`**
  - **`explain`** : describe tipos de recursos (a nivel de ayuda)
  -

kubectl

- Otros comandos:
  - **`port-forward`** → conectar máquina local con pod o servicio. **MUY ÚTIL**
  - **`cp`** → permite copiar ficheros desde / hacia máquina local (requiere 'tar' en imagen).
  - **`attach`** → permite unirnos al contenedor del pod
  - **`drain / cordon / uncordon`** → mantenimiento nodos
  - **`diff -f fichero.yaml`** → compara estado actual del cluster con lo que supondría el fichero yaml.
  - **`set / rollout`** → aplicar cambios (rolling updates)
  - **`replace`** → elimina y re-crea el recurso
  - **`patch`** → aplica cambios en algunos casos complicados

kubectl

- Modificadores más típicos:
  - `-n namespace` → ejecuta el comando en un determinado namespace
  - `-A` → Todos los namespaces (all)
  - `-l` → filtrado de labels
  - `-o wide|yaml|...` → cambia el formato de salida (wide da columnas extra con más info)
  - `-c container-name` → cuando un pod tiene varios contenedores permite apuntar al que queramos.
  - `-f fichero.yaml` → Fichero a utilizar (apply, create, delete)
  - `-it` → Similar a docker run / exec (stdin / tty). Usado en `kubectl exec -it ...`
  - `-w` → lr mostrando los cambios (watch) (típico con `kubectl get pod|svc`)
  - **--dry-run** client → Simula la ejecución del comando, sin enviarlo al servidor.

kubectl

- Cheat sheets:
  - https://kubernetes.io/docs/reference/kubectl/cheatsheet/
  - https://jamesdefabia.github.io/docs/user-guide/kubectl-cheatsheet/
- De docker a kubectl (trucos para usuarios de docker)
  - https://kubernetes.io/docs/reference/kubectl/docker-cli-to-kubectl/
- Generalmente no lanzaremos cargas de trabajo directamente con kubectl sino que lo haremos a través de ficheros YAML.

kubectl

- Docker style:

```
kubectl run -i --tty busybox --image=busybox --restart=Never -- sh
kubectl run nginx --image=nginx --command -- <cmd> <arg1> ... <argN>
kubectl exec -it my-pod -- bash
```

- Formatos de salida (-o): yaml, json, wide, …
- ¿Múltiples contenedores en el pod? → `-c container-name`
- Aprendiendo desde kubectl:

```
kubectl run nginx --image=nginx --dry-run=client -o=yaml > pod-nginx-spec.yaml
```

kubectl

- Generando ficheros YAML desde kubectl:

```
# pod
kubectl run --help → MUY ÚTIL

kubectl run nginx --image=nginx --dry-run=client -o=yaml > pod-nginx-spec.yaml
# con CMD (args)
kubectl run nginx --image=nginx --dry-run=client -o=yaml -- echo "hola caracola"

# con Entrypoint (command)
kubectl run nginx --image=nginx --dry-run=client -o=yaml --command -- echo "hola caracola"
```

kubectl

- Generando ficheros YAML desde kubectl:

```
# deployment
kubectl create deployment dep1 --image=nginx --dry-run=client -o=yaml > dep1.yaml

# servicio
kubectl expose deployment dep1 --type=LoadBalancer --port=8080 --targetPort=80 \
  --dry-run=client -o=yaml > svc-lb-dep1.yaml
```

kubectl

- Y más...

```
kubectl create job (or cronjob)
kubectl create configmap (or secret)...
kubectl create role (or clusterrole or rolebinding)... --dry-run=client -o=yaml
kubectl create serviceaccount
kubectl create namespace
kubectl create --help # :)
```

- NOTA: No se pueden crear directamente objetos del tipo **DaemonSet** o **StatefulSet**, pero siempre podemos crear un Deployment y adaptar el manifiesto.

kubectl - troubleshooting

- ¿Necesitas más detalles de algún objeto (IP, nodo asociado, etc.)? Prueba algo como:

```
kubectl get <tipo-objeto> -o=wide
kubectl describe <tipo-objeto> nombre-objeto
```

- ¿Algo no funciona y necesitas saber por qué?
  - Describe el objeto

```
kubectl describe <tipo-objeto> nombre-objeto
```

  - Analiza los logs de los pods

```
kubectl logs nombre_pod [-c nombre_contenedor]
```

kubectl

- Filtrado con kubectl
  - `-l` → Para labels
  - `--field-selector` (para otro tipo de filtrado).
  - Ejemplos:

```
kubectl get pods -l mylabel=xxx
kubectl get pods -l mylabel # devuelve pods con ese label, sin importar el valor


kubectl get pods --field-selector=status.phase=Running
kubectl get pods --all-namespaces --field-selector spec.nodeName=node02
```

kubectl

- Filtrando salida con filtrado json (-o jsonpath)
  - Ejemplos:

```
kubectl get pod nginx -o jsonpath='{.metadata.labels}'

# similar a
kubectl get pod nginx  -o json | jq -c '.metadata.labels'

# Obtener InternalIP de los nodos en una línea
kubectl get nodes -o jsonpath='{
$.items[*].status.addresses[?(@.type=="InternalIP")].address }'

# Obtener el nodePort de un servicio
kubectl get -o jsonpath="{.spec.ports[0].nodePort}" services nginx
```

Más ejemplos en https://kubernetes.io/docs/reference/kubectl/jsonpath/

kubectl

- Crear ficheros YAML desde recursos.

```
kubectl get deployment xxx -o=yaml > backup_xxx.yaml
```

- Modificar o eliminar algo que está en ejecución guardándolo previamente en YAML (cuidado!)

```
kubectl get deployment xxx -o=yaml > backup_xxx.yaml
# editamos o cambiamos cosas en backup_pod.yaml
kubectl apply -f backup_xxx.yaml
```

- Recordad que generalmente NO tocaremos pods sino recursos de alto nivel!!!!

Los pods no se pueden cambiar (excepto algunos parámetros vía 'kubectl patch' o labels con 'kubectl label'). Solamente eliminar y recrear.

Ejercicios básicos sobre pods

KEEPCODING
Tech School

Ejercicios con Pods

1. Crea un namespace que se llame “keepcoding” y un pod con la imagen nginx en este namespace.
2. Crea el mismo POD pero ahora haciéndolo con un YAML.
3. Lanza un POD con Busybox (con kubectl) que ejecute el comando “env”
4. Haz lo mismo que (3) pero con un YAML.
5. Prepara el YAML para crear un namespace, pero sin crearlo. Intenta hacerlo con kubectl (pistas: --dry-run, -o=yaml, …)
6. Obtén los pods de todos los namespaces **(kubectl get --help)**
7. Crea un POD con la imagen nginx definiendo el puerto 80 **(kubectl run --help)**

> **Prueba además la opción --expose!**

Ejercicios con Pods

8. Obtén la dirección IP del pod anterior. Hazle un curl desde otro POD que levantes.
9. Obtén el YAML de un pod.
10. Crea un pod con busybox que escriba “Hola Keepcoding!” y salga.
11. Haz un describe del pod
12. Obtén los logs del pod
13. ¿Si el pod se reinició, cómo obtienes los logs anteriores al reinicio? (***kubectl logs --help***)
14. Ejecuta una shell interactiva dentro del pod de nginx.
15. Crea un POD de nginx que tenga una variable de entorno NAME=keepcoding. Visualízala.

Ejercicios con Pods

16. Crea 3 pods, con nombres nginx1, nginx2 y nginx3 que tengan el label app=v1.
17. Muestra con un comando los pods junto con sus labels (***kubectl get --help***)
18. Crea un servicio del tipo ClusterIP que apunte a los 3 nginx creados (app=v1).
19. Desde un pod "busybox" prueba la resolución DNS y acceso al servicio creado (prueba desde el mismo namespace y desde uno diferente).
20. Haz que ahora el label del pod nginx2 pase a ser app=v2 (***kubectl label --help***), y observa los cambios en los endpoints del servicio que creaste.
21. Obtener el label ‘app’ de los pods.
22. Muestra los pods que tengan el label ‘app=v2’
23. Quítale el label a los pods anteriores.

Ejercicios con Pods

24. Crea un Pod que despliegue en un nodo que tenga el label "ssd=true".
25. Añadir un annotation a los 3 pods con una descripción (ej. description="devops"). (`kubectl annotate --help`)
26. Muestra las 'annotations' del pod nginx1
27. Quita las 'annotations' a todos los pods anteriores.
28. Elimina todos los pods y déjalo todo limpio.
29. Crea un POD con nginx con un **liveness Probe** que sea ejecutar el comando “ls”.
30. Cambia el check para que no empiece hasta 15 segundos después de la creación, y el intervalo de chequeos sea de 10 segundos.

Ejercicios con Pods

31. Cambia el Check para que sea de tipo HTTP en el puerto 80 y en la raíz.
32. Crea un POD con **2 contenedores**, en uno nginx y en otro redis.

KEEPCODING

Tech School

Madrid | Barcelona | Bogotá