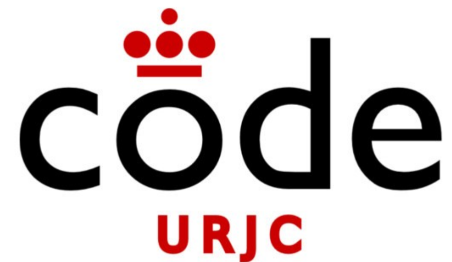


Máster Universitario en Ingeniería Informática

Computación en la Nube

Mejores prácticas en Kubernetes

Universidad
Rey Juan Carlos

Iván Chicano
Correo: ivan.chicano@urjc.es

©2022 Iván Chicano Capelo
Algunos derechos reservados
Este documento se distribuye bajo la licencia
"Atribución 4.0 Internacional" de Creative Commons,
disponible en
https://creativecommons.org/licenses/by/4.0/deed.es

Mejores prácticas

code URJC

- Namespaces
- Control de recursos
- Control de salud
- Servicios externos
- Otras prácticas

Mejores prácticas

- **Namespaces**
- Control de recursos
- Control de salud
- Servicios externos
- Otras prácticas

Namespaces

code URJC

- La mayoría de los recusos Kubernetes se asocian a un **namespace**
  - Dos recursos se pueden llamar igual si están en namespaces distintos
  - Facilita la gestión de recursos por diferentes equipos / usuarios

```
$ kubectl get namespaces
```

```
NAME           STATUS    AGE
default        Active    1d
kube-system    Active    1d
kube-public    Active    1d
```

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

Namespaces

- Especificar un namespace en un comando

```
$ kubectl --namespace=<namespace-name> get pods
```

- Establecer el namespace para todos los comandos

```
kubectl config set-context $(kubectl config current-context) \
  --namespace=<namespace-name>
```

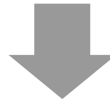
```
$ kubectl config view | grep namespace:
```

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

Namespaces

- Creación de un namespace nuevo

```
$ kubectl create namespace development
```



```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: development
  labels:
    name: development
```

```
$ kubectl apply -f namespace-dev.yaml
```

Namespaces

code URJC

- Levantar un recurso en un namespace distinto

```
$ kubectl apply -f pod.yaml --namespace=development
```

- O incluido dentro del manifiesto

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
  namespace: development
  labels:
    name: mypod
spec:
  containers:
  - name: mypod
    image: nginx
```

Si se intenta usar el argumento - -namespace para levantarlo, el comando falla

Namespaces

code URJC

- **Herramientas adicionales a kubectl**
  - **kubens:** Cambio ágil de cluster al que está conectado kubectl

```
$ kubectx minikube
Switched to context "minikube".
$ kubectx -
Switched to context "oregon".
```

  - **kubectx:** Cambio ágil de namespace por defecto

```
$ kubens kube-system
Context "test" set.
Active namespace is "kube-system".
$ kubens -
Context "test" set.
Active namespace is "default".
```

https://github.com/ahmetb/kubectx/

Namespaces

code URJC

- Por defecto los namespaces están ocultos para otros namespaces en el mismo clúster.
  - Pero no están totalmente aislados.
  - Dado un servicio Kubernetes de nombre “simpleservice”, si otro pod se quiere conectar a este puede usar su nombre como DNS.

```
/ # curl http://simpleservice:9876/info
```

  - Si es servicio “simpleservice” está en otro namespace, por ejemplo “test”, podemos usar el siguiente nombre como DNS.

```
/ # curl http://simpleservice.test:9876/info
```

Namespaces

- Si se crea un namespace que se asigna a un TLD como “com”, “es” u “org”, y se crea un servicio con el mismo nombre que un sitio web como “google”, Kubernetes interceptará las solicitudes a “google.com” y las enviará a su servicio.

Namespaces

code URJC

- ¿Cuántos namespaces crear?
  - Demasiados y se meten en medio.
  - Muy pocos y perdemos los beneficios.

Namespaces

- **Solución**: Depende de la organización y/o proyecto.
  - Equipos pequeños (5-10 microservicios): Solo usar “default” o tener uno de producción y otro de desarrollo.
  - Equipos con crecimiento rápido (10+ microservicios con múltiples subequipos con sus microservicios propios): Usar múltiples namespaces para producción y desarrollo. Cada subequipo puede decidir tener su propio namespace.

Namespaces

code URJC

- **Solución**: Depende de la organización y/o proyecto.
  - Gran empresa: Como mínimo cada equipo tiene su namespace (o incluso varios, para desarrollo y producción). Buena idea configurar acceso (RBAC) y límites de recursos.
  - Multinacional (con colaboración con empresas externas): Múltiples clústeres.

Mejores prácticas

code URJC

- Namespaces
- **Control de recursos**
- Control de salud
- Servicios externos
- Otras prácticas

Control de recursos

code URJC

- Cuando levantamos un Pod en Kubernetes, este selecciona un nodo para ejecutar el Pod.
- Dichos nodos tienen una capacidad de CPU y memoria limitada.
- Por defecto, si no indicamos límites de recursos de computación, un nodo se puede quedar sin memoria o CPU.
- Las aplicaciones puede que consuman más recursos de las que deberían.
- Es más fácil levantar más replicas que arreglar el código.
- Aplicaciones paran, el nodo se bloquea...

Requests and limits

code URJC

- Las Request (peticiones) y Limits (límites) son mecanismos que nos ofrece Kubernetes para solicitar y limitar el consumo de recursos en los Pods.
- Con las Requests, Kubernetes garantiza que el contenedor tendrá acceso a esa cantidad de recursos (mínimo).
- Kubernetes usará esta información para decidir en qué nodo poner el Pod.
- Con los Limits, Kubernetes asegura que los recursos que el contenedor consume no pasan del límite (máximo).

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

Requests and limits

code URJC

- Kubernetes distingue dos tipos de recurso de computación: CPU y memoria.
- La CPU se mide en unidades de CPU. 1 unidad de CPU es equivalente a 1 núcleo de CPU (físico o virtual).
- Se pueden solicitar unidades fraccionales de CPU (0.5 CPUs) o en formato de milinúcleos (1 CPU = 1000m).
- La memoria se mide en bytes.
- Se puede expresar en potencias de 10 (K, M…) o en potencias de 2 (Ki, Mi…).

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

Requests and limits

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: app
    image: images.my-company.example/app:v4
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
  - name: log-aggregator
    image: images.my-company.example/log-aggregator:v6
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

Requests and limits

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

- Las peticiones y límites se definen por contenedor

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

Requests and limits

- No puedes solicitar más recursos de los que están disponibles en tus nodos. Si solicitas recursos de más, tus pods no se levantarán (pending).

  Si tu aplicación se acerca a los límites de CPU, Kubernetes empezará a “estrangular” (throttle) el contenedor, reduciendo su rendimiento.

- Si tu aplicación llega al límite de memoria, el contenedor será finalizado (si el Pod es gestionado por un Deployment, se levantará un nuevo Pod).

Requests and limits

- Si tenemos una sonda de vida (Liveness Probe), tenemos que tener cuidado al configurar initialDelaySeconds.
  - Si es demasiado pequeño puede que el Pod se reinicie constantemente.
  - Si es demasiado grande tardaremos mucho en saber si nuestra aplicación está viva o no.
- Recomendable usar el percentil 99 (p99) de tiempos de inicio de la aplicación.
  - Probar múltiples veces cuánto tarda en iniciar la aplicación.

ResourceQuotas y LimitRanges

- Pero, ¿qué pasa si otro equipo que también usa el cluster no configura el uso de recursos de sus contenedores?
- Sus aplicaciones pueden estrangular a las nuestras.
- Reparto injusto de recursos.
- ¿Todo nuestro trabajo no ha valido para nada?
- Para evitar esto, se pueden definir ResourceQuotas y LimitRanges a nivel de Namespace.

ResourceQuotas

code URJC

- Los ResourceQuotas limitan el consumo de recursos por namespace.
- No solo recursos de computación, puede limitar la cantidad de objetos Kubernetes creados en el namespace por tipo.
- Solo vamos a ver limitación de recursos de computación.

https://kubernetes.io/docs/concepts/policy/resource-quotas/

ResourceQuotas

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-resource-quota
spec:
  hard:
    requests.cpu: 250m
    requests.memory: 50Mib
    limits.cpu: 2000m
    limits.memory: 2Gib
```

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

ResourceQuotas

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-resource-quota
spec:
  hard:
    requests.cpu: 250m
    requests.memory: 50Mib
    limits.cpu: 2000m
    limits.memory: 2Gib
```

La suma de las requests en el namespace (CPU o memoria) no pueden superar este umbral

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

ResourceQuotas

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-resource-quota
spec:
  hard:
    requests.cpu: 250m
    requests.memory: 50Mib
    limits.cpu: 2000m
    limits.memory: 2Gib
```

La suma de los limits en el namespace (CPU o memoria) no pueden superar este umbral

https://kubernetes.io/docs/concepts/configuration/manage-resources-containers

ResourceQuotas

code URJC

- Las ResourceQuotas fuerzan a que los contenedores indiquen sus requests y/o limits.
- Si no se incluyen, falla el comando (403).
- Dependiendo de la organización, un equipo puede tener uno o varios namespaces.
  - Si se tiene un namespace de producción, lo común es no poner quotas.
  - Si se tiene un namespace de desarrollo, lo común es tener quotas estrictas.

LimitRanges

code URJC

- Los LimitRanges permiten limitar los recursos consumidos a nivel de contenedor, Pod o PersistentVolume.
- Se evita que se creen contenedores que consumen demasiados recursos (o muy pequeños).
- Permiten asignar valores por defecto de consumo de recursos.

https://kubernetes.io/docs/concepts/policy/limit-range/

LimitRanges

```
apiVersion: v1
kind: LimitRange
metadata:
  name: dev-limit-range
spec:
  limits:
- default:
    cpu: 500m
    memory: 50Mib
  defaultRequest:
    cpu: 100m
    memory: 10Mib
  max:
    cpu: 1000m
    memory: 200Mib
  min:
    cpu: 10m
    memory: 100Kib
type: Container
```

https://kubernetes.io/docs/concepts/policy/limit-range/

LimitRanges

code URJC

```
apiVersion: v1
kind: LimitRange
metadata:
  name: dev-limit-range
spec:
  limits:
- default:
    cpu: 500m
    memory: 50Mib
  defaultRequest:
    cpu: 100m
    memory: 10Mib
  max:
    cpu: 1000m
    memory: 200Mib
  min:
    cpu: 10m
    memory: 100Kib
type: Container
```

Valor por defecto asignado a los limits de los contenedores de los Pods.

https://kubernetes.io/docs/concepts/policy/limit-range/

LimitRanges

code URJC

```
apiVersion: v1
kind: LimitRange
metadata:
  name: dev-limit-range
spec:
  limits:
- default:
    cpu: 500m
    memory: 50Mib
  defaultRequest:
    cpu: 100m
    memory: 10Mib
  max:
    cpu: 1000m
    memory: 200Mib
  min:
    cpu: 10m
    memory: 100Kib
type: Container
```

Valor por defecto asignado a las requests de los contenedores de los Pods.

https://kubernetes.io/docs/concepts/policy/limit-range/

LimitRanges

```
apiVersion: v1
kind: LimitRange
metadata:
  name: dev-limit-range
spec:
  limits:
- default:
    cpu: 500m
    memory: 50Mib
  defaultRequest:
    cpu: 100m
    memory: 10Mib
  max:
    cpu: 1000m
    memory: 200Mib
  min:
    cpu: 10m
    memory: 100Kib
type: Container
```

Limits máximos que un Pod puede asignar.

https://kubernetes.io/docs/concepts/policy/limit-range/

LimitRanges

code URJC

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: dev-limit-range
spec:
  limits:
- default:
    cpu: 500m
    memory: 50Mib
  defaultRequest:
    cpu: 100m
    memory: 10Mib
  max:
    cpu: 1000m
    memory: 200Mib
  min:
    cpu: 10m
    memory: 100Kib
type: Container
```

Requests mínimos que un Pod puede asignar.

https://kubernetes.io/docs/concepts/policy/limit-range/

Mejores prácticas

- Namespaces
- Control de recursos
- **Control de salud**
- Servicios externos
- Otras prácticas

Control de salud

- Un sistema bien diseñado debe ser tolerante a fallos.
  - Es muy difícil (o imposible) diseñar un sistema sin errores.
- En un sistema distribuido puede (y suele) haber muchas partes independientes en funcionamiento
  - Que se necesitan entre sí.
  - Si una falla, debería ser detectado, solucionado y/o rodearla.
- Control de salud (Health check)

Control de salud

- Kubernetes empieza a mandar tráfico a un Pod cuando todos sus contenedores están iniciados.
  - Un contenedor está iniciado cuando su proceso interno se inicia.
  - Si un contenedor crashea, se reinicia.
  - Puede ser insuficiente:
    - La aplicación entra en punto muerto (deadlock).
    - Al iniciar la aplicación carga muchos datos.
    - O requiere de un servicio externo para iniciar.
    - Etc.

Control de salud

- Kubernetes ofrece dos tipos de controles de salud.
- Sonda de disponibilidad (Readiness Probe)
- Sonda de vida (Liveness Probe)

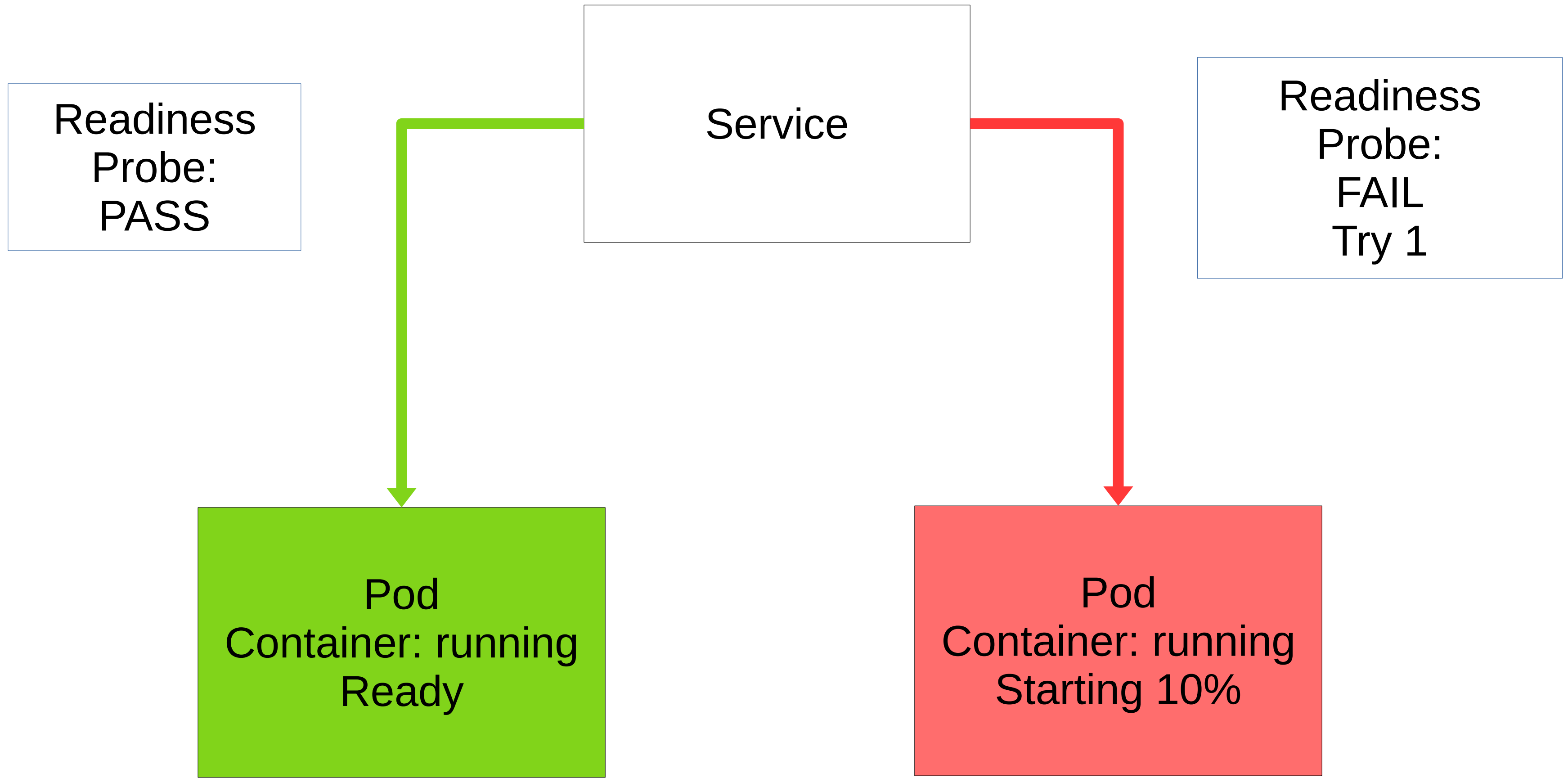
https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Readiness

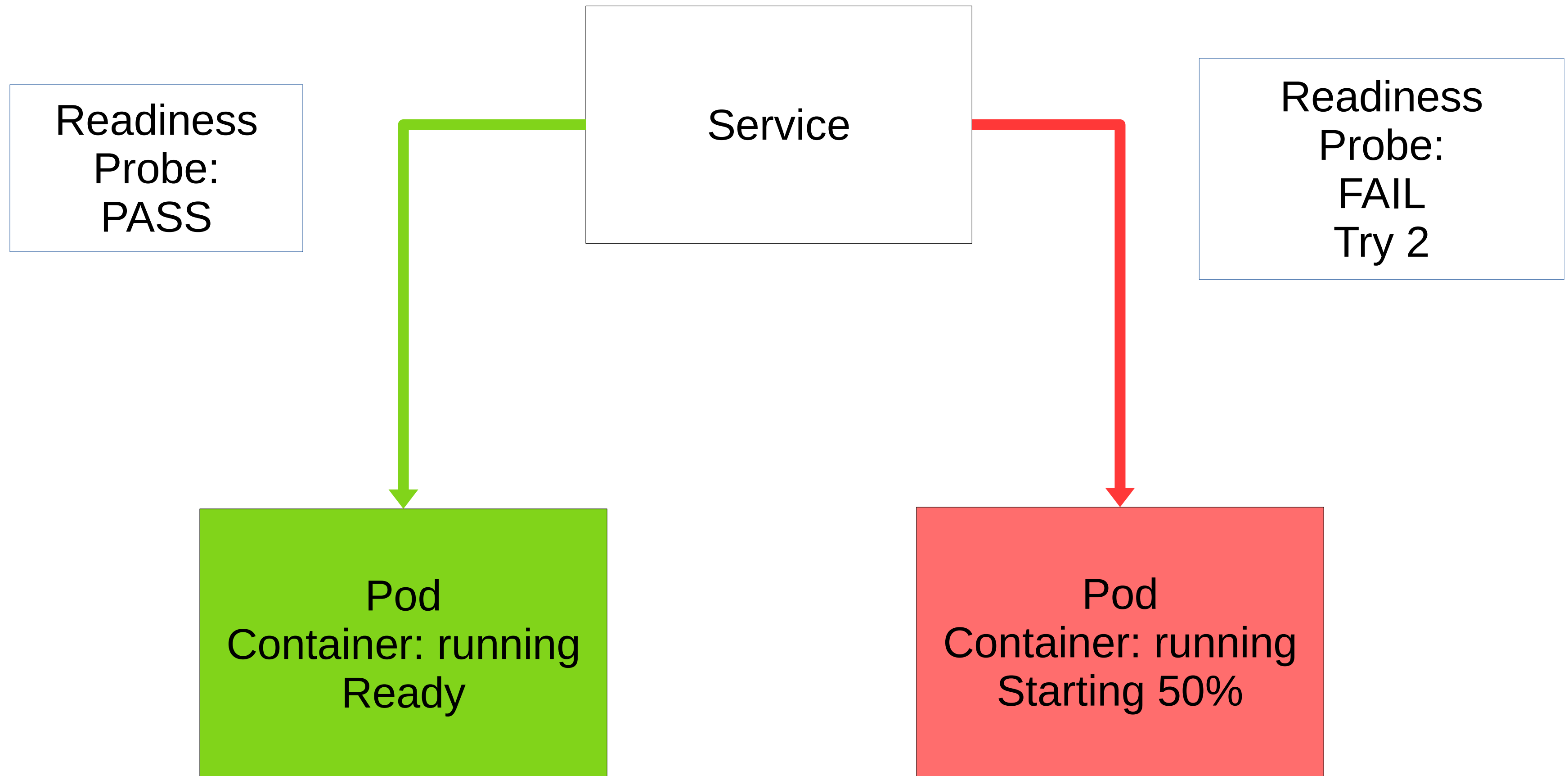
code URJC

- La sonda de disponibilidad (Readiness Probe) comunica a Kubernetes que el contenedor está listo para recibir tráfico.
- Si la sonda falla, el contenedor no recibirá tráfico a través de servicios Kubernetes.
- Si la sonda detecta que el contenedor está listo, informa al servicio.
- Deben ser independientes (no depender de servicios externos como bases de datos o cachés).

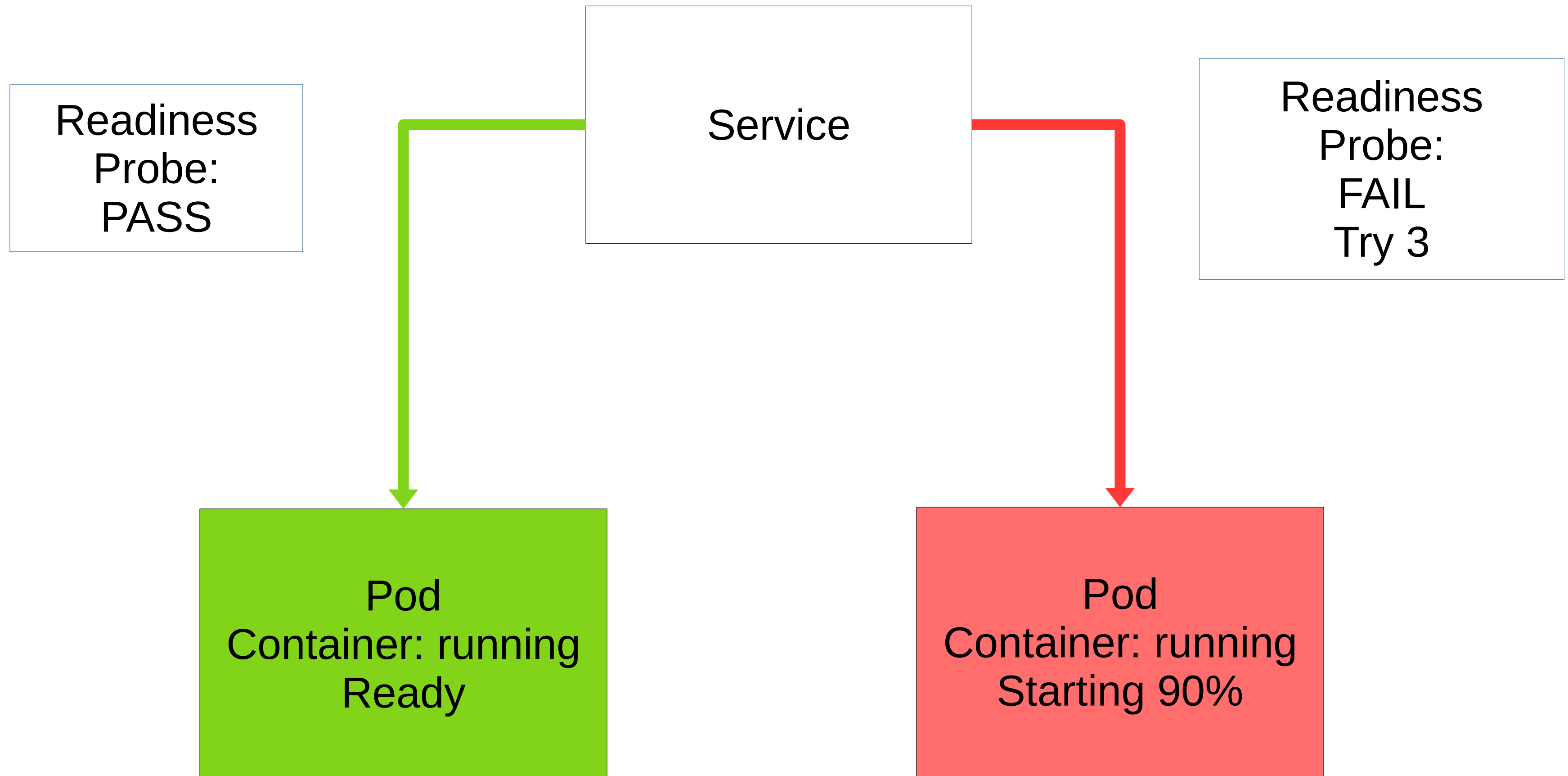
Readiness



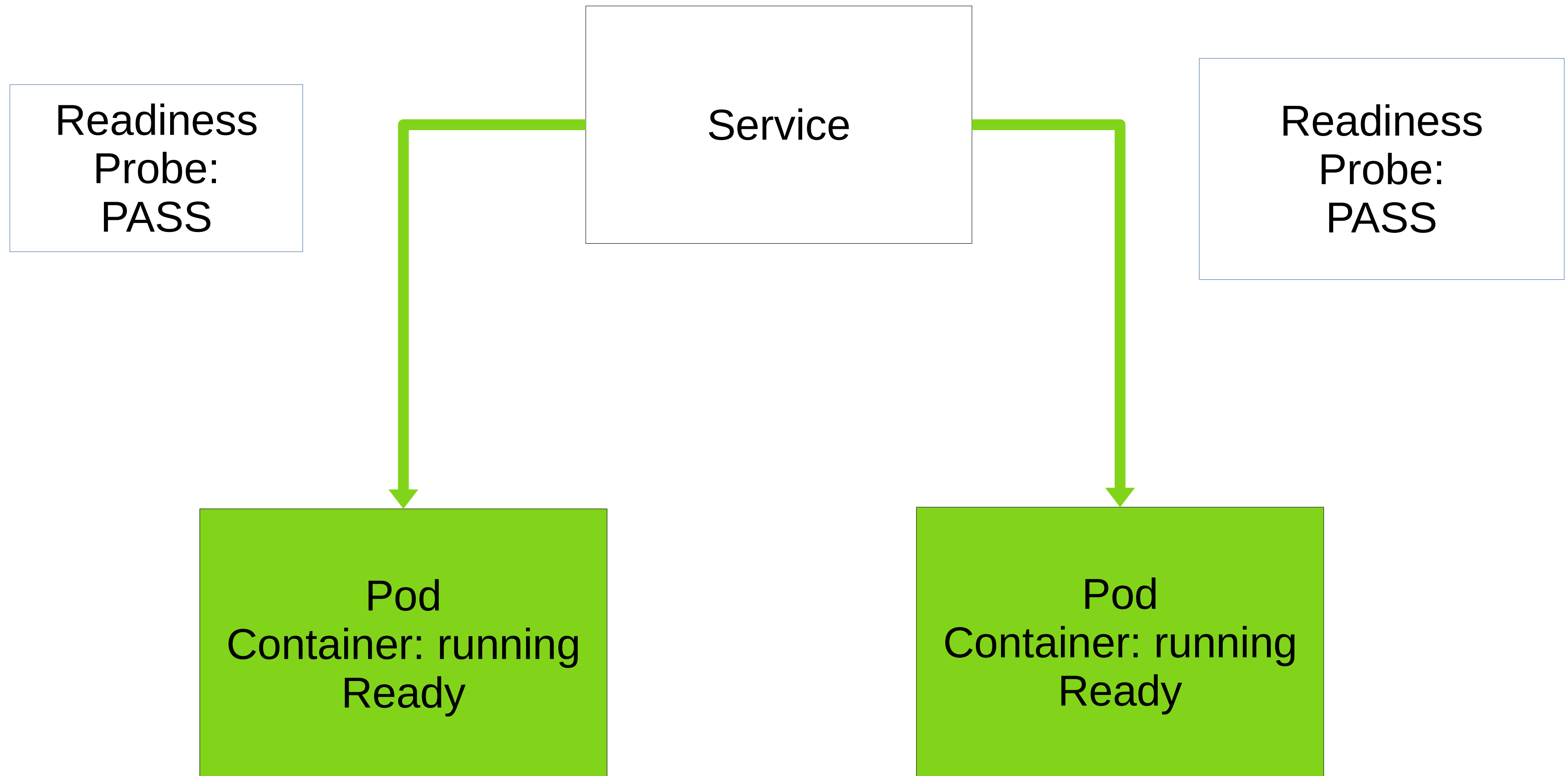
Readiness



Readiness



Readiness

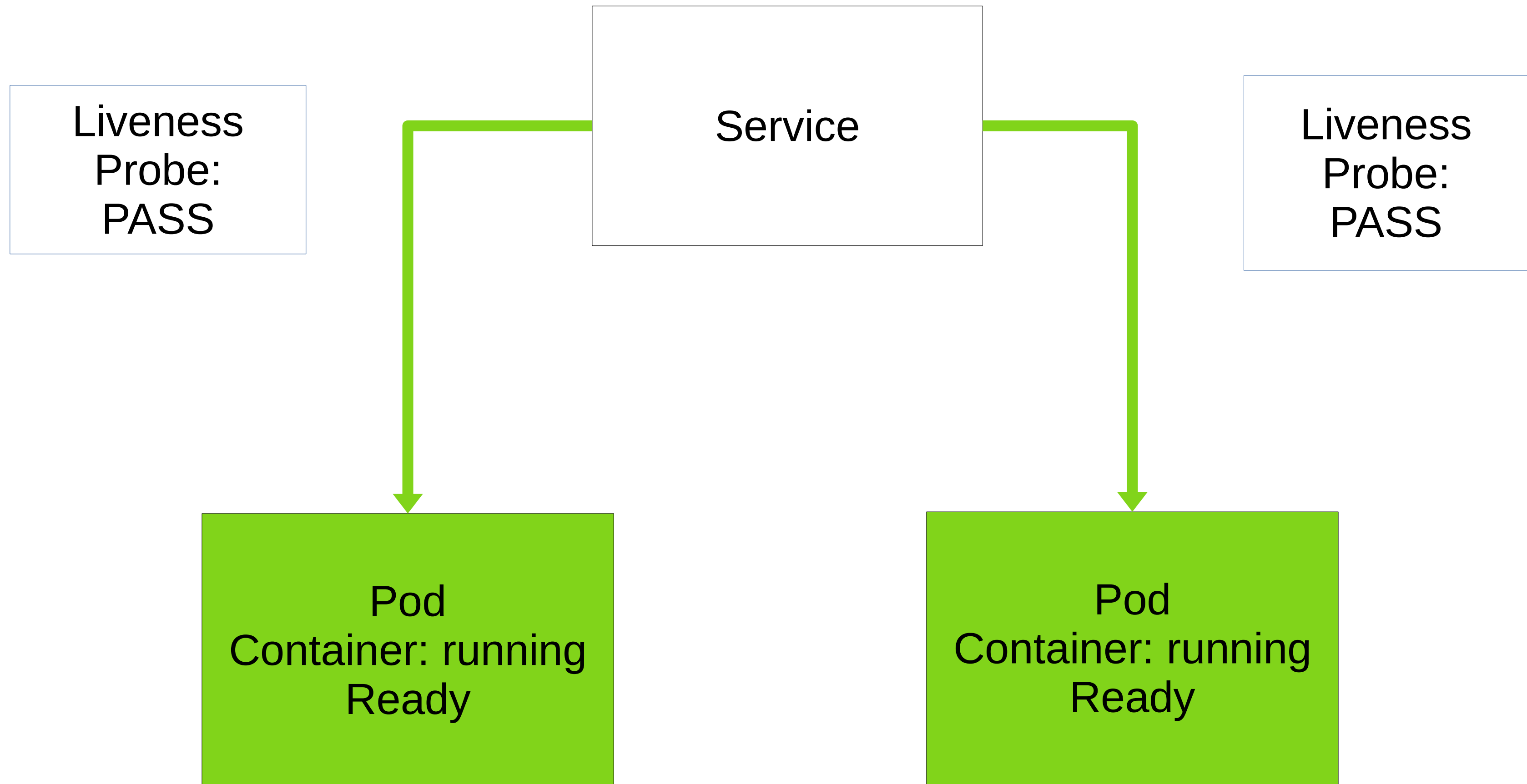


Liveness

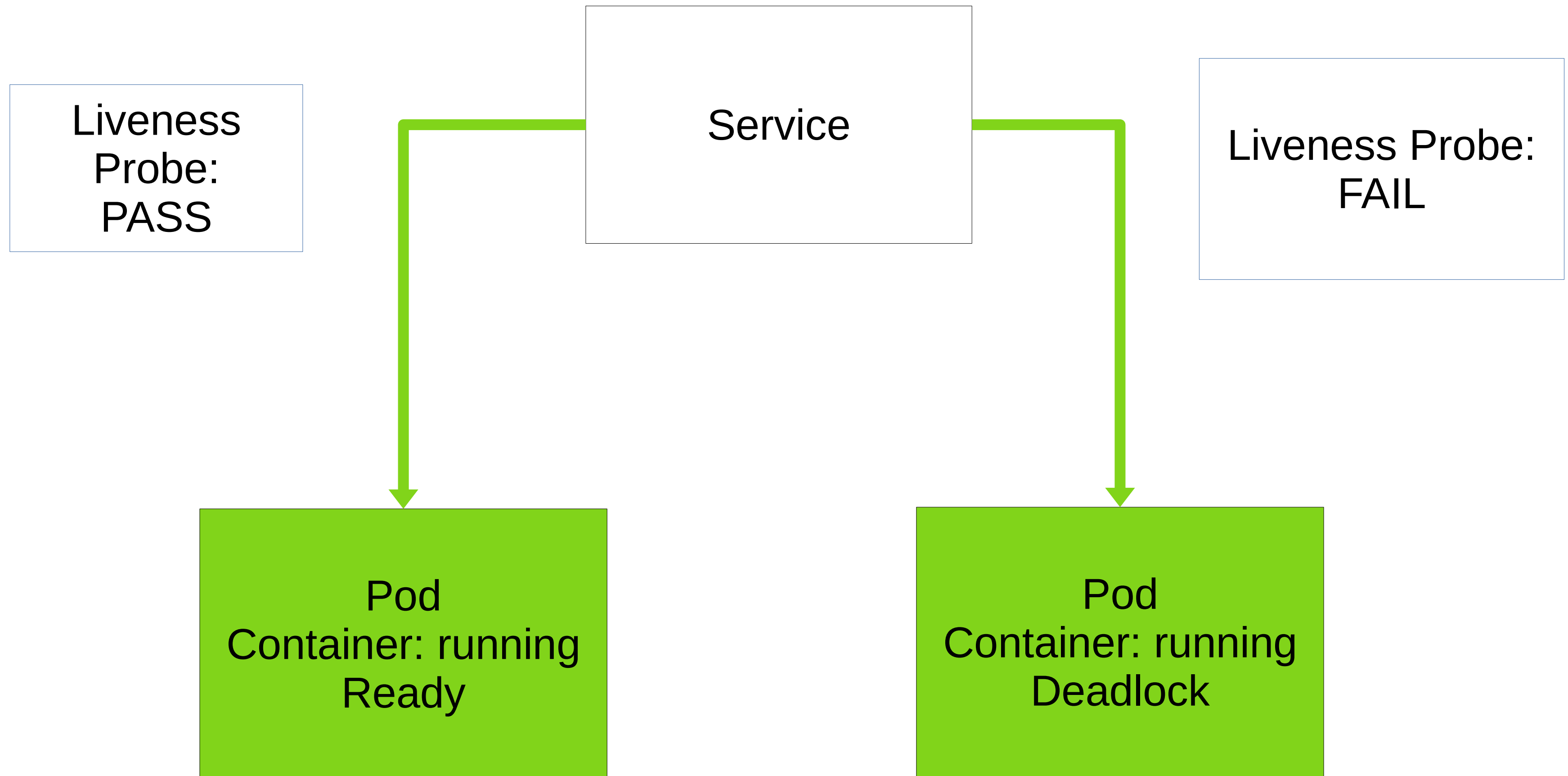
code URJC

- La sonda de vida (Liveness Probe) comunica a Kubernetes si el contenedor ha tenido algún problema y debe reiniciarse.
- Si la sonda falla, el contenedor se reiniciará.
- Si la sonda detecta vida, el contenedor sigue ejecutándose.

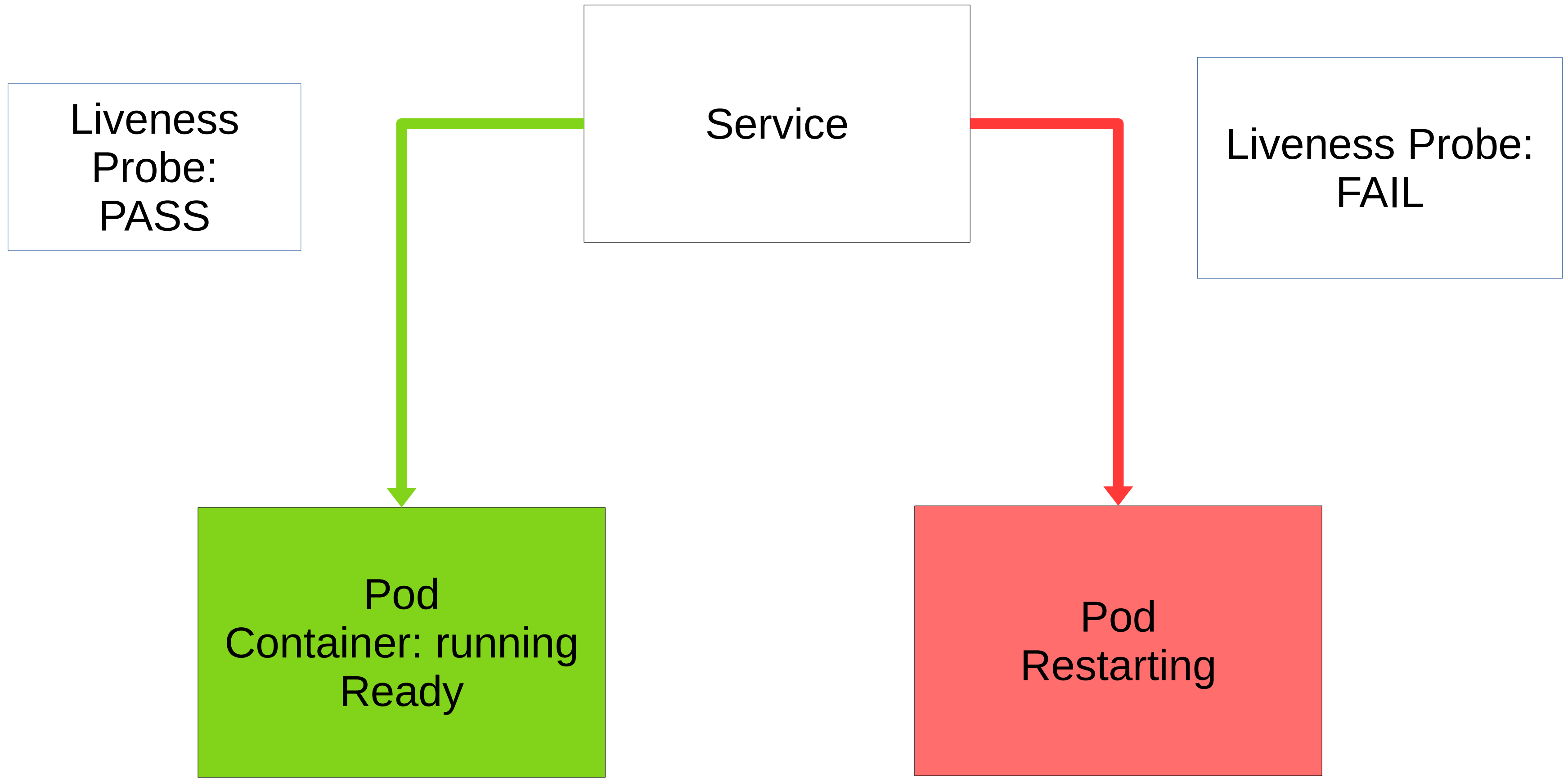
Liveness



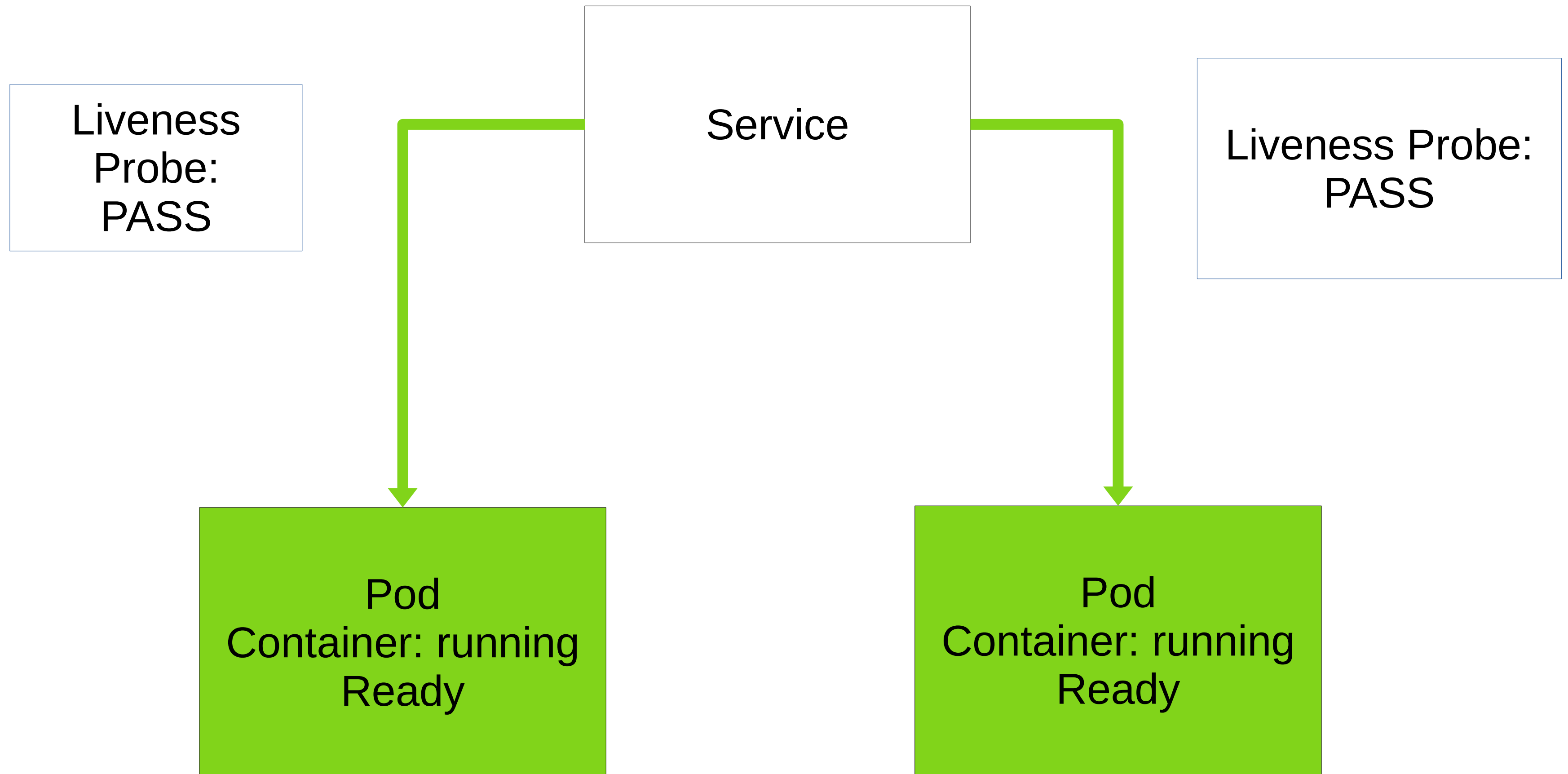
Liveness



Liveness



Liveness



Tipos de sondas

- Las sondas de vida y disponibilidad pueden ser de distintos tipos
  - Comando
  - HTTP
  - TCP

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

Tipos de sondas - Comando

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

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

Tipos de sondas - Comando

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Sonda de vida (Liveness Probe) de tipo comando.

Ejecuta comando cat /tmp/healthy

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

Tipos de sondas

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
    readinessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Mismo ejemplo con sonda de disponibilidad (Readiness Probe)

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

Tipos de sondas - Comando

code URJC

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Ejecuta el comando cada 5 segundos (periodSeconds)

Espera 5 segundos antes de lanzar la primera sonda (initialDelaySeconds)

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

Tipos de sondas - Comando

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600
    livenessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 5
      periodSeconds: 5
```

Durante 30 segundos, la sonda pasará (PASS), después la sonda fallará (FAIL) y el contenedor se reiniciará.

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

Tipos de sondas - HTTP

- Si el exit code del comando es 0, la sonda tendrá éxito (saludable, healthy).
- Si el exit code es distinto de 0 la sonda fallará (no saludable, unhealthy).

Tipos de sondas - Comando

```
$ kubectl apply -f
https://k8s.io/examples/pods/probe/exec-liveness.yaml
```

- Antes de que pasen 35 segundos

```
$ kubectl describe pod liveness-exec
```

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  18s   default-scheduler  Successfully assigned
default/liveness-exec to docker-desktop
  Normal  Pulling    18s   kubelet            Pulling image
"registry.k8s.io/busybox"
  Normal  Pulled     16s   kubelet            Successfully pulled image
"registry.k8s.io/busybox" in 2.1286646s
  Normal  Created    16s   kubelet            Created container liveness
  Normal  Started    16s   kubelet            Started container liveness
```

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

Tipos de sondas - Comando

- Después de que pasen 35 segundos

```
$ kubectl describe pod liveness-exec
```

```
Type      Reason     Age                     From                Message
  ----      ------     ----                    ----                -------
  Normal    Scheduled  2m17s                   default-scheduler   Successfully assigned
default/liveness-exec to docker-desktop
  Normal    Pulled     2m15s                   kubelet             Successfully pulled image
"registry.k8s.io/busybox" in 2.1286646s
  Normal    Pulling    63s (x2 over 2m17s)  kubelet             Pulling image
"registry.k8s.io/busybox"
  Normal    Created    62s (x2 over 2m15s)  kubelet             Created container liveness
  Normal    Started    62s (x2 over 2m15s)  kubelet             Started container liveness
  Normal    Pulled     62s                     kubelet             Successfully pulled image
"registry.k8s.io/busybox" in 590.2588ms
  Warning  Unhealthy  18s (x6 over 103s)   kubelet             Liveness probe failed: cat: can't
open '/tmp/healthy': No such file or directory
  Normal    Killing    18s (x2 over 93s)    kubelet             Container liveness failed
liveness probe, will be restarted
```

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

Tipos de sondas - Comando

code URJC

- Si cambiamos livenessProbe por readinessProbe
- Antes de que pasen 35 segundos

```
$ kubectl describe pod readiness-exec
```

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  20s   default-scheduler  Successfully assigned
default/readiness-exec to docker-desktop
  Normal  Pulling    20s   kubelet            Pulling image
"registry.k8s.io/busybox"
  Normal  Pulled     20s   kubelet            Successfully pulled image
"registry.k8s.io/busybox" in 787.8317ms
  Normal  Created    20s   kubelet            Created container readiness
  Normal  Started    20s   kubelet            Started container readiness
```

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

Tipos de sondas - Comando

code URJC

- Después de que pasen 35 segundos

```
$ kubectl describe pod liveness-exec
```

```
Events:
  Type     Reason     Age                    From               Message
  ----     ------     ----                   ----               -------
  Normal   Scheduled  2m48s                  default-scheduler  Successfully assigned 
default/readiness-exec to docker-desktop
  Normal   Pulling    2m48s                  kubelet            Pulling image 
"registry.k8s.io/busybox"
  Normal   Pulled     2m48s                  kubelet            Successfully pulled 
image "registry.k8s.io/busybox" in 787.8317ms
  Normal   Created    2m48s                  kubelet            Created container 
readiness
  Normal   Started    2m48s                  kubelet            Started container 
readiness
  Warning  Unhealthy  39s (x21 over 2m14s)   kubelet            Readiness probe failed: 
cat: can't open '/tmp/healthy': No such file or directory
```

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

Tipos de sondas - HTTP

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: registry.k8s.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/

Tipos de sondas - HTTP

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

Petición HTTP GET cada 3 segundos al servidor ejecutándose en el contenedor.

Ruta: /healthz

Puerto: 8080

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

Tipos de sondas - HTTP

code URJC

- Si el estado de la respuesta de la petición está en el rango 200 o 300 la sonda devolverá éxito (saludable, healthy).
- Si el estado de la respuesta es distinto la sonda fallará (no saludable, unhealthy).
- Por defecto usa HTTP, se puede configurar para HTTPS.

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

- Solo permite GET, si necesitas otro método HTTP usa sondas de tipo comando con curl.

Tipos de sondas - HTTP

```
livenessProbe:
    exec:
      command:
         - curl
         - -X POST
         - http://localhost/healthz
      initialDelaySeconds: 5
      periodSeconds: 5
```

Tipos de sondas - TCP

```
apiVersion: v1
kind: Pod
metadata:
  name: goproxy
  labels:
    app: goproxy
spec:
  containers:
  - name: goproxy
    image: registry.k8s.io/goproxy:0.1
    ports:
    - containerPort: 8080
    readinessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 5
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 20
```

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

Mejores prácticas

code URJC

- Namespaces
- Control de recursos
- Control de salud
- **Servicios externos**
- Otras prácticas

Servicios externos

code URJC

- Es altamente probable que tu aplicación en Kubernetes necesite algún servicio externo al clúster.
  - Bases de datos
  - APIs externas (Clima, mapas, etc.)
  - Servicios en migración
- Por lo general usas el endpoint que te ofrece este servicio directamente en el código.
- Pero no siempre es el caso.
  - Por ejemplo, bases de datos pueden tener distintos endpoints para distintas instancias.

Servicios externos

code URJC

- Un servicio Kubernetes permite abstraer el acceso a Pods utilizando un selector.
- Pero se pueden configurar sin selector para abstraer otros servicios, como por ejemplo servicios externos.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

https://kubernetes.io/docs/concepts/services-networking/service/#services-without-selectors

Servicios externos

code URJC

- Los Service de tipo ExternalName permiten asociar un Service a un nombre DNS externo.

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: ExternalName
  externalName: my.database.example.com
```

https://kubernetes.io/docs/concepts/services-networking/service/#externalname

Servicios externos

- Pero, ¿y si el servicio externo tiene múltiples endpoints?
- ¿Usar un ConfigMap?
  - Se introduce como variable de entorno en la aplicación.
- Aunque puede funcionar, tiene problemas.
  - Hay que crear el ConfigMap.
  - Adaptar el código para leer variables de entorno.
  - Si cambia el endpoint, puede ser necesario reiniciar los contenedores en ejecución para que se actualicen las variables de entorno.

EndpointSlice

code URJC

- Para asociar un Service sin selector a un servicio externo, es necesario un objeto EndpointSlice.
- Representan un subconjunto de endpoints que implementan un Service.
- Se suelen usar para servicios en la red interna de la organización pero externos al clúster o al namespace.
  - Para servicios externos a la red hay opciones más sencillas.
  - También se puede usar para servicios externos si es necesario configurar determinados aspectos.

https://kubernetes.io/docs/concepts/services-networking/service/#services-without-selectors

EndpointSlice

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-1
  labels:
    kubernetes.io/service-name: my-service
addressType: IPv4
ports:
  - name: ''
    appProtocol: http
    protocol: TCP
    port: 9376
endpoints:
  - addresses:
      - "10.4.5.6"
      - "10.1.2.3"
```

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

EndpointSlice

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-1
  labels:
    kubernetes.io/service-name: my-service
addressType: IPv4
ports:
  - name: ''
    appProtocol: http
    protocol: TCP
    port: 9376
endpoints:
  - addresses:
      - "10.4.5.6"
      - "10.1.2.3"
```

Por convenio es recomendable utilizar el nombre del Service como prefijo del nombre del EndpointSlice

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

EndpointSlice

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-1
  labels:
    kubernetes.io/service-name: my-service
addressType: IPv4
ports:
  - name: ''
    appProtocol: http
    protocol: TCP
    port: 9376
endpoints:
  - addresses:
      - "10.4.5.6"
      - "10.1.2.3"
```

El valor de este label tiene que ser el nombre del Service

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

EndpointSlice

code URJC

```
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-1
  labels:
    kubernetes.io/service-name: my-service
addressType: IPv4
ports:
  - name: ''
    appProtocol: http
    protocol: TCP
    port: 9376
endpoints:
  - addresses:
      - "10.4.5.6"
      - "10.1.2.3"
```

Vacío porque el puerto 9376 no es un puerto con protocolo conocido.
Si por ejemplo se usara el puerto 80, se pondría http.

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

EndpointSlice

```yaml
apiVersion: discovery.k8s.io/v1
kind: EndpointSlice
metadata:
  name: my-service-1
  labels:
    kubernetes.io/service-name: my-service
addressType: IPv4
ports:
  - name: ''
    appProtocol: http
    protocol: TCP
    port: 9376
endpoints:
  - addresses:
      - "10.4.5.6"
      - "10.1.2.3"
```

Direcciones IP a las que el Service se conecta (no es necesario especificar orden)

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

EndpointSlice

- En el código, podemos referenciar a este servicio.

```
http://my-service
```

- Mejor que tener en el código las Ips.

```
http://10.4.5.6
```

- O que sacarlas de variables de entorno.

```
http://process.env[“ENDPOINT_SERVICE_ENV”]
```

https://kubernetes.io/docs/concepts/services-networking/service/#services-without-selectors

Mejores prácticas

code URJC

- Namespaces
- Control de recursos
- Control de salud
- Servicios externos
- **Otras prácticas**

Otras prácticas

code URJC

- No desplegar Pods individualmente.
  - Usar Deployments, StatefulSets, ReplicaSets, etc.
  - Múltiples réplicas en distintos nodos.
- Reducir el tamaño de los contenedores (imagenes pequeñas).
- Usar labels para identificar y organizar recursos.
- Usar RBAC como sistema de autorización.
- Network policies y firewalls para restringir tráfico entre recursos.
- Monitorización de los componentes de k8s del clúster.
- Auditoría de logs.

Otras prácticas

- Usar un servicio de k8s en la nube (EKS en AWS).
  - Reduce la complejidad de gestionar tu propio clúster.
  - Nos ofrecen ellos la infraestructura.
  - Nos permite y facilita el auto escalado.
- Control de versiones para ficheros de configuración.
  - Usar un flujo de trabajo (workflow) para facilitar automaticación de despliegue e integración (CI/CD).
- Apagado grácil de contenedores en caso de caída.

Mejores prácticas

code
URJC

- Checklist de mejores prácticas

https://learnk8s.io/production-best-practices