Kubernetes Self Healing

Prelude

- Kubernetes provides self-healing by default.
- If a POD/container goes down, Kubernetes will instantly redeploy it, matching the so-called **desired state**.

What is self-healing Kubernetes?

The idea behind self-healing Kubernetes is simple:

- If a container fails, Kubernetes automatically redeploys the afflicted container to its desired state to restore operations.

Self-healing Kubernetes has four capabilities:

1. **restart** failed containers
2. **replace** containers that require application updates, such as a new software version
3. **disable** containers that don't respond to predefined health checks
4. **prevent** containers from appearing to users or other containers until they are ready

Expectations from Kubernetes

- Ideally, container detection and restoration should be seamless and immediate, minimize application disruption
- Organisations can specify how Kubernetes performs health checks and what actions it should take after it detects a problem

How does self-healing work with Kubernetes?

Kubernetes clusters are composed of pods -- logical entities where containers deploy

A pod has five possible states:

1. **Pending**. The pod has been created but is not running
2. **Running**. The pod and its containers are running without issue
3. **Succeeded**. The pod completes its container lifecycle properly and it runs and stops normally
4. **Failed**. At least one container within the pod has failed, and the pod is terminated
5. **Unknown**. The pod's state and parameters cannot be determined

---

Kubectl commands can obtain the pods and their status for a given application

```
kubectl get pods
Kubectl get pods -o wide
Kubectl get pods -show-labels
Kunectl describe pod <podName>
```

POD Restart Policies

Restart policies are an important component of self-healing applications, which are automatically repaired when a problem arises.

There are three possible values for a pod's restart policy in Kubernetes:

- Always
- OnFailure
- Never

Always is the default restart policy.

- With this policy, containers will always be restarted if they stop.
- This policy should be used for applications that always need to be running.

OnFailure Policy will always restart containers only if the container process exits with an error code or the container is determined to be unhealthy by a liveness probe.

Never restart policy causes the pod's containers to never be restarted, even if the container exits or a liveness probe fails.

Configuring POD Restart Policies

Kind: Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  restartPolicy: OnFailure
  containers:
  - name: example-container
    image: nginx
```

Kind: Deployment

- In this example, all three replicas of the nginx container in the example deployment will have a restart policy of Always.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: example
  template:
    metadata:
      labels:
        app: example
    spec:
      containers:
      - name: example-container
        image: nginx
        restartPolicy: Always
```

Container Restart Policies

1. **Liveness probe:** whether the application is up and running
    - A liveness probe finds the running status of each container.
    - If a container fails the liveness probe, Kubernetes terminates it and creates a new container according to internally established policies.
    - **Types**: liveness command, liveness HTTP request, TCP liveness probe
2. **Readiness probe:** whether it is ready to accept requests
    - A readiness probe verifies a container's ability to service requests or handle traffic.
    - If a container fails the readiness probe, Kubernetes removes its IP address from the corresponding pod.
    - This makes it unavailable until it is terminated and restarted.
3. **Startup probes:** Only used when some programs need additional startup time on their first initialization.

Configuring Liveness Probe

Type: liveness command

Create a file **exec-liveness.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
```

Anatomy

- In the configuration file, you can see that the Pod has a single Container.
- The **periodSeconds** field specifies that the **kubelet** should perform a liveness probe every 5 seconds.
- The **initialDelaySeconds** field tells the kubelet that it should wait 5 seconds before performing the first probe.
- To perform a probe, the kubelet executes the command: **`cat /tmp/healthy`** in the target container.

- If the command succeeds, it **returns 0**, and the kubelet considers the container to be alive and healthy.
- If the command **returns a non-zero value**, the **kubelet** kills the container and restarts it.

Testing

Note:
When the container starts, it executes this command:

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -f /tmp/healthy; sleep 600"
```

For the first 30 seconds of the container's life, there is a **/tmp/healthy** file.

So during the first 30 seconds, the command **`cat /tmp/healthy`** returns a success code.

After 30 seconds, **`cat /tmp/healthy`** returns a failure code.

Create the Pod:

```
kubectl apply -f exec-liveness.yaml
```

Within 30 seconds, view the Pod events:

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

The output indicates that no liveness probes have failed yet: All is well

```
Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  5s    default-scheduler  Successfully assigned
default/liveness-exec to node2
  Normal  Pulling    5s    kubelet            Pulling image
"registry.k8s.io/busybox"
  Normal  Pulled     3s    kubelet            Successfully pulled image
"registry.k8s.io/busybox" in 1.757781793s (1.757790833s including waiting)
  Normal  Created    3s    kubelet            Created container liveness
  Normal  Started    3s    kubelet            Started container liveness
```

After 35 seconds, view the Pod events again:

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

```
Type     Reason     Age              From             Message
```

```
  ----     ------     ----              ----               -------
  Normal   Scheduled  46s               default-scheduler  Successfully assigned
default/liveness-exec to node2
  Normal   Pulling    46s               kubelet            Pulling image "registry.k8s.io/busybox"
  Normal   Pulled     44s               kubelet            Successfully pulled image
"registry.k8s.io/busybox" in 1.757781793s (1.757790833s including waiting)
  Normal   Created    44s               kubelet            Created container liveness
  Normal   Started    44s               kubelet            Started container liveness
  Warning  Unhealthy  1s (x3 over 11s)  kubelet            Liveness probe failed: cat: can't open
'/tmp/healthy': No such file or directory
  Normal   Killing    1s                kubelet            Container liveness failed liveness probe,
will be restarted
```

Wait another 30 seconds, and verify that the container has been restarted:

```
Type     Reason     Age                From               Message
  ----     ------     ----               ----               -------
  Normal   Scheduled  90s                default-scheduler  Successfully assigned
default/liveness-exec to node2
  Normal   Pulled     88s                kubelet            Successfully pulled image
"registry.k8s.io/busybox" in 1.757781793s (1.757790833s including waiting)
  Warning  Unhealthy  45s (x3 over 55s)  kubelet            Liveness probe failed: cat:
can't open '/tmp/healthy': No such file or directory
  Normal   Killing    45s                kubelet            Container liveness failed
liveness probe, will be restarted
  Normal   Pulling    15s (x2 over 90s)  kubelet            Pulling image
"registry.k8s.io/busybox"
  Normal   Created    14s (x2 over 88s)  kubelet            Created container liveness
  Normal   Pulled     14s                kubelet            Successfully pulled image
"registry.k8s.io/busybox" in 1.349600787s (1.349609497s including waiting)
  Normal   Started    13s (x2 over 88s)  kubelet            Started container liveness
```

Also Note:

- The output shows that RESTARTS has been incremented.
- Note that the RESTARTS counter increments as soon as a failed container comes back to the running state

```
linuxadmin@master:~$ kubectl get pods
NAME            READY   STATUS    RESTARTS      AGE
liveness-exec   1/1     Running   1 (22s ago)   97s
```

Configuring Liveness Probe

Type: liveness HTTP request

```
livenessProbe:
      httpGet:
        path: /index.html
        port: 80
```

Configuring Liveness Probe

Type: TCP liveness probe

Used when finding if MySQL Server is accessible by it's port: 3306

```
livenessProbe:
      tcpSocket:
        port: 3306
```

Configure Readiness Probe

- Sometimes, applications are temporarily unable to serve traffic.
- For example, an application might need to load large data or configuration files during startup, or depend on external services after startup.
- In such cases, you don't want to kill the application, but you don't want to send it requests either.
- Kubernetes provides **readiness probes to detect and mitigate** these situations.
- A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

Note:

Readiness probes run on the container during its whole lifecycle.

Caution:

Liveness probes do not wait for readiness probes to succeed.

If you want to wait before executing a liveness probe you should use LARGER **initialDelaySeconds**.

How to configure?

Readiness probes are configured similarly to liveness probes. The only difference is that you use the **readinessProbe** field instead of the **livenessProbe** field.

```
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: 200
      periodSeconds: 5
    readinessProbe:
     exec:
       command:
       - cat
       - /tmp/healthy
     initialDelaySeconds: 100
     periodSeconds: 5
```

Configuring Startup Probe

```
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:
      httpGet:
        path: /index.html
        port: 80
      failureThreshold: 1
      periodSeconds: 10

   startupProbe:
      httpGet:
        path: /index.html
        port: 80
      failureThreshold: 30
      periodSeconds: 10
```

Using startup probe, the application will have a maximum of **5 minutes (30 * 10 = 300s)** to finish its startup.

Once the startup probe has succeeded once, the liveness probe takes over.

If the startup probe never succeeds, the container is killed after 300s and subject to the pod's restartPolicy.

Kubernetes behaviour?

- To accomplish self-healing, Kubernetes frequently checks the status of pods and their containers.
- If Kubernetes determines that a container has failed or is unresponsive, it terminates and restarts or reschedules the pod as soon as possible: assuming there is sufficient infrastructure available to do so.
- **Time Taken:** Detecting a failed container application or component can take up to five minutes.

A typical containerized environment includes three major layers:

- The **application layer** houses the container entity, along with its code and dependencies.
- The Kubernetes **component layer** is the OS for containers. This layer includes the **kubelet, kube-proxy and container runtime** components that make Kubernetes work.
- The **infrastructure layer** is where servers, disks with container image files and network connectivity operate. Kubernetes handles the Self-healing?

Application Layer

What Layer does Kubernetes support for Self-healing?

- Self-healing operates at the **application layer only**
- That is where Kubernetes deploys and manages containers
- If a pod crashes, Kubernetes can reschedule it

Infrastructure Layer

No Infrastructure Self-healing

- Unfortunately, however, Kubernetes has no provision or mechanism to enable infrastructure self-healing.
- A problem with Kubernetes itself or the infrastructure, such as a failed disk or network switch, could therefore disrupt a containerized application beyond Kubernetes' ability to repair.

Hello Cloud Admins, it's your Job to monitor such failures

- Organisations that implement self-healing Kubernetes should also integrate some form of application performance monitoring to oversee Kubernetes, as well as comprehensive infrastructure monitoring to alert IT admins to issues in the component and infrastructure layers

Component Layer

How to detect Component Layer issues?

- Properly rolling out Kubernetes with tools such as Terraform is not enough
- You need a component that is continuously and proactively monitoring the health status of all Kubernetes components ensuring prompt recovery with minimal impact on the rest of the cluster
- Solutions: Azure Kubernetes Service (AKS), Amazon EC2 Container Service (ECS), Google Kubernetes Engine (GKE), Portainer, Red Hat OpenShift Container Platform

Solution:

- From your Datacenter, move your deployments to Cloud PaaS
