Monitoring Health

In this lesson, we will find out why to monitor the health of services and how to achieve this using Kubernetes Probes.

WE'LL COVER THE FOLLOWING

- Why to Monitor Health?
- Kubernetes Probes
  - Liveness Probe
  - Readiness Probe
- Understanding the Updated Pod Definition
  - Liveness Probe in Action
- Pods Are (Almost) Useless (By Themselves)

Why to Monitor Health?

The go-demo-2 Docker image is designed to fail on the first sign of trouble. In cases like that, there is no need for any health checks. When things go wrong:

- The main process stops.
- The container hosting the main process stops as well.
- Kubernetes restarts the failed container.

However, not all services are designed to fail fast. Even those that are might still benefit from additional health checks. For example, a back-end API can be up and running but, due to a memory leak, serves requests much slower than expected. Such a situation might benefit from a health check that would verify whether the service responds within, for example, two seconds.

Kubernetes Probes

We can exploit Kubernetes **liveness and readiness probes** for that.

Liveness Probe

`livenessProbe` can be used to confirm whether a container should be running. If the probe fails, Kubernetes will kill the container and apply restart policy which defaults to `Always`.

Readiness Probe

We'll leave `readinessProbe` for later since it is directly tied to `Services`.

Instead, we'll explore `livenessProbe`. Both are defined in the same way so the experience with one of them can be easily applied to the other.

Understanding the Updated Pod Definition

Let's take a look at an updated definition of the Pod we used thus far.

```
cat pod/go-demo-2-health.yml
```

The **output** is as follows.

```
apiVersion: v1
kind: Pod
metadata:
  name: go-demo-2
  labels:
    type: stack
spec:
  containers:
  - name: db
    image: mongo:3.3
  - name: api
    image: vfarcic/go-demo-2
    env:
    - name: DB
      value: localhost
    livenessProbe:
      httpGet:
        path: /this/path/does/not/exist
        port: 8080
      initialDelaySeconds: 5
      timeoutSeconds: 2 # Defaults to 1
      periodSeconds: 5 # Defaults to 10
      failureThreshold: 1 # Defaults to 3
```

- **Line 8-12:** Don't get confused by seeing two containers in this Pod. Those two should be defined in separate Pods. However, since that would require knowledge we are yet to obtain, and go-demo-2 doesn't work

require knowledge we are yet to obtain, and go-demo-2 doesn't work without a database, we'll have to stick with the example that specifies two containers. It won't take long until we break it into pieces.

- **Line 16-19:** The additional definition is inside the `livenessProbe`. We defined that the action should be `httpGet` followed with the `path` and the `port` of the service. Since `/this/path/does/not/exist` is true to itself, the probe will fail, thus showing us what happens when a container is unhealthy. The `host` is not specified since it defaults to the Pod IP.
- **Line 20-23:** We declared that the first execution of the probe should be delayed by five seconds (`initialDelaySeconds`), that requests should timeout after two seconds (`timeoutSeconds`), that the process should be repeated every five seconds (`periodSeconds`), and (`failureThreshold`) define how many attempts it must try before giving up .

Liveness Probe in Action

Let's take a look at the probe in action.

```
kubectl create \
    -f pod/go-demo-2-health.yml
```

We created the Pod with the probe. Now we must wait until the probe fails a few times. A minute is more than enough. Once we're done waiting, we can describe the Pod.

```
kubectl describe \
    -f pod/go-demo-2-health.yml
```

The bottom of the **output** contains events. They are as follows.

```
...
Events:
  Type     Reason                 Age              From               Message
  ----     ------                 ----             ----               -------
  Normal   Scheduled              6m               default-scheduler  Successfully assigned g
  Normal   SuccessfulMountVolume  6m               kubelet, minikube  MountVolume.SetUp succe
  Normal   Pulling                6m               kubelet, minikube  pulling image "mongo"
  Normal   Pulled                 6m               kubelet, minikube  Successfully pulled ima
  Normal   Created                6m               kubelet, minikube  Created container
  Normal   Started                6m               kubelet, minikube  Started container
```

```
Normal   Created                 5m (x3 over 6m)  kubelet, minikube  Created container
Normal   Started                 5m (x3 over 6m)  kubelet, minikube  Started container
Warning  Unhealthy               5m (x3 over 6m)  kubelet, minikube  Liveness probe failed:

Normal   Pulling                 5m (x4 over 6m)  kubelet, minikube  pulling image "vfarcic/
Normal   Killing                 5m (x3 over 6m)  kubelet, minikube  Killing container with
Normal   Pulled                  5m (x4 over 6m)  kubelet, minikube  Successfully pulled ima
```

We can see that, once the container started, the probe was executed, and that it failed. As a result, the container was killed only to be created again. In the output above, we can see that the process was repeated three times (`3x over ...`).

Please visit Probe v1 core if you'd like to learn all the available options.

Pods Are (Almost) Useless (By Themselves)

Pods are fundamental building blocks in Kubernetes. In most cases, you will not create Pods directly. Instead, you'll use higher level constructs like Controllers.

Pods are disposable. They are not long lasting services. Even though Kubernetes is doing its best to ensure that the containers in a Pod are (almost) always up-and-running, the same cannot be said for Pods. If a Pod fails, gets destroyed, or gets evicted from a Node, it will not be rescheduled. At least, not without a Controller. Similarly, if a whole node is destroyed, all the Pods on it will cease to exist. Pods do not heal by themselves. Excluding some special cases, Pods are not meant to be created directly.

> 📝 **Do not** create Pods by themselves. Let one of the controllers create Pods for you.

In the next lesson, we will test our understanding of Kubernetes Pods with the help of a quick quiz.