Configure liveliness and readiness probes for pods in AKS cluster.

Configure Liveness, Readiness and Startup Probes

This page shows how to configure liveness, readiness and startup probes for containers.

For more information about probes, see Liveness, Readiness and Startup Probes

The kubelet uses liveness probes to know when to restart a container. For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.

A common pattern for liveness probes is to use the same low-cost HTTP endpoint as for readiness probes, but with a higher failureThreshold. This ensures that the pod is observed as not-ready for some period of time before it is hard killed.

The kubelet uses readiness probes to know when a container is ready to start accepting traffic. One use of this signal is to control which Pods are used as backends for Services. A Pod is considered ready when its Ready condition is true. When a Pod is not ready, it is removed from Service load balancers. A Pod's Ready condition is false when its Node's Ready condition is not true, when one of the Pod's readinessGates is false, or when at least one of its containers is not ready.

The kubelet uses startup probes to know when a container application has started. If such a probe is configured, liveness and readiness probes do not start until it succeeds, making sure those probes don't interfere with the application startup. This can be used to adopt liveness checks on slow starting containers, avoiding them getting killed by the kubelet before they are up and running.

Caution:

Liveness probes can be a powerful way to recover from application failures, but they should be used with caution. Liveness probes must be configured carefully to ensure that they truly indicate unrecoverable application failure, for example a deadlock.
Note:
Incorrect implementation of liveness probes can lead to cascading failures. This results in restarting of container under high load; failed client requests as your application became less scalable; and increased workload on remaining pods due to some failed pods. Understand the difference between readiness and liveness probes and when to apply them for your app.

Before you begin

You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster. It is recommended to run this tutorial on a cluster with at least

two nodes that are not acting as control plane hosts. If you do not already have a cluster, you can create one by using minikube or you can use one of these Kubernetes playgrounds:

- Killercoda
- KodeKloud
- Play with Kubernetes

Define a liveness command

Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. Kubernetes provides liveness probes to detect and remedy such situations.

In this exercise, you create a Pod that runs a container based on the registry.k8s.io/busybox:1.27.2 image. Here is the configuration file for the Pod:

pods/probe/exec-liveness.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/busybox:1.27.2
    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
```

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.

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 https://k8s.io/examples/pods/probe/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:

```
Type    Reason     Age   From               Message
----    ------     ----  ----               -------
Normal  Scheduled  11s   default-scheduler  Successfully assigned default/liveness-exec to node01
Normal  Pulling    9s    kubelet, node01    Pulling image "registry.k8s.io/busybox:1.27.2"
Normal  Pulled     7s    kubelet, node01    Successfully pulled image "registry.k8s.io/busybox:1.27.2"
Normal  Created    7s    kubelet, node01    Created container liveness
Normal  Started    7s    kubelet, node01    Started container liveness
```

After 35 seconds, view the Pod events again:

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

At the bottom of the output, there are messages indicating that the liveness probes have failed, and the failed containers have been killed and recreated.

```
Type     Reason     Age               From               Message
----     ------     ----              ----               -------
Normal   Scheduled  57s               default-scheduler  Successfully assigned default/liveness-exec to node01
Normal   Pulling    55s               kubelet, node01    Pulling image "registry.k8s.io/busybox:1.27.2"
Normal   Pulled     53s               kubelet, node01    Successfully pulled image "registry.k8s.io/busybox:1.27.2"
Normal   Created    53s               kubelet, node01    Created container liveness
Normal   Started    53s               kubelet, node01    Started container liveness
Warning  Unhealthy  10s (x3 over 20s)  kubelet, node01    Liveness probe failed: cat: can't open '/tmp/healthy': No such file or directory
Normal   Killing    10s               kubelet, node01    Container liveness failed liveness probe, will be restarted
```

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

```
kubectl get pod liveness-exec
```

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:

```
NAME           READY     STATUS    RESTARTS   AGE
liveness-exec   1/1       Running   1          1m
```

Define a liveness HTTP request

Another kind of liveness probe uses an HTTP GET request. Here is the configuration file for a Pod that runs a container based on the registry.k8s.io/e2e-test-images/agnhost image.

pods/probe/http-liveness.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-http
spec:
  containers:
  - name: liveness
    image: registry.k8s.io/e2e-test-images/agnhost:2.40
    args:
    - liveness
    livenessProbe:
      httpGet:
        path: /healthz
        port: 8080
        httpHeaders:
        - name: Custom-Header
          value: Awesome
      initialDelaySeconds: 3
      periodSeconds: 3
```

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 3 seconds.
The initialDelaySeconds field tells the kubelet that it should wait 3 seconds before performing the first probe. To perform a probe, the kubelet sends an HTTP GET request to the server that is running in the container and listening on port 8080. If the handler for the server's /healthz path returns a success code, the kubelet considers the container to be alive and healthy. If the handler returns a failure code, the kubelet kills the container and restarts it.

Any code greater than or equal to 200 and less than 400 indicates success. Any other code indicates failure.

You can see the source code for the server in server.go.

For the first 10 seconds that the container is alive, the /healthz handler returns a status of 200. After that, the handler returns a status of 500.

```
http.HandleFunc("/healthz", func(w http.ResponseWriter, r *http.Request) {
    duration := time.Now().Sub(started)
    if duration.Seconds() > 10 {
        w.WriteHeader(500)
        w.Write([]byte(fmt.Sprintf("error: %v", duration.Seconds())))
    } else {
        w.WriteHeader(200)
        w.Write([]byte("ok"))
    }
})
```

The kubelet starts performing health checks 3 seconds after the container starts. So the first couple of health checks will succeed. But after 10 seconds, the health checks will fail, and the kubelet will kill and restart the container.

To try the HTTP liveness check, create a Pod:

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

After 10 seconds, view Pod events to verify that liveness probes have failed and the container has been restarted:

```
kubectl describe pod liveness-http
```

In releases after v1.13, local HTTP proxy environment variable settings do not affect the HTTP liveness probe.

Define a TCP liveness probe

A third type of liveness probe uses a TCP socket. With this configuration, the kubelet will attempt to open a socket to your container on the specified port. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

pods/probe/tcp-liveness-readiness.yaml

```
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: 15
      periodSeconds: 10
    livenessProbe:
      tcpSocket:
        port: 8080
      initialDelaySeconds: 15
      periodSeconds: 10
```

As you can see, configuration for a TCP check is quite similar to an HTTP check. This example uses both readiness and liveness probes. The kubelet will run the first liveness probe 15 seconds after the container starts. This will attempt to connect to the goproxy container on port 8080. If the liveness probe fails, the container will be restarted. The kubelet will continue to run this check every 10 seconds.

In addition to the liveness probe, this configuration includes a readiness probe. The kubelet will run the first readiness probe 15 seconds after the container starts. Similar to the liveness probe, this will attempt to connect to the goproxy container on port 8080. If the probe succeeds, the Pod will be marked as ready and will receive traffic from services. If the readiness probe fails, the pod will be marked unready and will not receive traffic from any services.

To try the TCP liveness check, create a Pod:

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

After 15 seconds, view Pod events to verify that liveness probes:

```
kubectl describe pod goproxy
```

Define a gRPC liveness probe

FEATURE STATE: Kubernetes v1.27 [stable]

If your application implements the gRPC Health Checking Protocol, this example shows how to configure Kubernetes to use it for application liveness checks. Similarly you can configure readiness and startup probes.

Here is an example manifest:

pods/probe/grpc-liveness.yaml

```
apiVersion: v1
kind: Pod
metadata:
```

```
  name: etcd-with-grpc
spec:
  containers:
  - name: etcd
    image: registry.k8s.io/etcd:3.5.1-0
    command: [ "/usr/local/bin/etcd", "--data-dir",  "/var/lib/etcd", "--listen-client-urls",
"http://0.0.0.0:2379", "--advertise-client-urls", "http://127.0.0.1:2379", "--log-level", "debug"]
    ports:
    - containerPort: 2379
    livenessProbe:
      grpc:
        port: 2379
      initialDelaySeconds: 10
```

To use a gRPC probe, port must be configured. If you want to distinguish probes of different types and probes for different features you can use the service field. You can set service to the value liveness and make your gRPC Health Checking endpoint respond to this request differently than when you set service set to readiness. This lets you use the same endpoint for different kinds of container health check rather than listening on two different ports. If you want to specify your own custom service name and also specify a probe type, the Kubernetes project recommends that you use a name that concatenates those. For example: myservice-liveness (using - as a separator).

Note:
Unlike HTTP or TCP probes, you cannot specify the health check port by name, and you cannot configure a custom hostname.
Configuration problems (for example: incorrect port or service, unimplemented health checking protocol) are considered a probe failure, similar to HTTP and TCP probes.

To try the gRPC liveness check, create a Pod using the command below. In the example below, the etcd pod is configured to use gRPC liveness probe.

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

After 15 seconds, view Pod events to verify that the liveness check has not failed:

```
kubectl describe pod etcd-with-grpc
```

When using a gRPC probe, there are some technical details to be aware of:

- The probes run against the pod IP address or its hostname. Be sure to configure your gRPC endpoint to listen on the Pod's IP address.
- The probes do not support any authentication parameters (like -tls).
- There are no error codes for built-in probes. All errors are considered as probe failures.
- If ExecProbeTimeout feature gate is set to false, grpc-health-probe does **not** respect the timeoutSeconds setting (which defaults to 1s), while built-in probe would fail on timeout.

Use a named port

You can use a named port for HTTP and TCP probes. gRPC probes do not support named ports.

For example:

```
ports:
- name: liveness-port
  containerPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
```

Protect slow starting containers with startup probes

Sometimes, you have to deal with applications that require additional startup time on their first initialization. In such cases, it can be tricky to set up liveness probe parameters without compromising the fast response to deadlocks that motivated such a probe. The solution is to set up a startup probe with the same command, HTTP or TCP check, with a failureThreshold * periodSeconds long enough to cover the worst case startup time.

So, the previous example would become:

```
ports:
- name: liveness-port
  containerPort: 8080

livenessProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 1
  periodSeconds: 10

startupProbe:
  httpGet:
    path: /healthz
    port: liveness-port
  failureThreshold: 30
  periodSeconds: 10
```

Thanks to the 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 to provide a fast response to container deadlocks. If the startup probe never succeeds, the container is killed after 300s and subject to the pod's restartPolicy.

Define readiness probes

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 runs on the container during its whole lifecycle.
Caution:

The readiness and liveness probes do not depend on each other to succeed. If you want to wait before executing a readiness probe, you should use initialDelaySeconds or a startupProbe.
Readiness probes are configured similarly to liveness probes. The only difference is that you use the readinessProbe field instead of the livenessProbe field.

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

Configuration for HTTP and TCP readiness probes also remains identical to liveness probes.

Readiness and liveness probes can be used in parallel for the same container. Using both can ensure that traffic does not reach a container that is not ready for it, and that containers are restarted when they fail.

Configure Probes

Probes have a number of fields that you can use to more precisely control the behavior of startup, liveness and readiness checks:

- initialDelaySeconds: Number of seconds after the container has started before startup, liveness or readiness probes are initiated. If a startup probe is defined, liveness and readiness probe delays do not begin until the startup probe has succeeded. If the value of periodSeconds is greater than initialDelaySeconds then the initialDelaySeconds will be ignored. Defaults to 0 seconds. Minimum value is 0.
- periodSeconds: How often (in seconds) to perform the probe. Default to 10 seconds. The minimum value is 1. While a container is not Ready, the ReadinessProbe may be executed at times other than the configured periodSeconds interval. This is to make the Pod ready faster.
- timeoutSeconds: Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1.

- successThreshold: Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness and startup Probes. Minimum value is 1.
- failureThreshold: After a probe fails failureThreshold times in a row, Kubernetes considers that the overall check has failed: the container is *not* ready/healthy/live. Defaults to 3. Minimum value is 1. For the case of a startup or liveness probe, if at least failureThreshold probes have failed, Kubernetes treats the container as unhealthy and triggers a restart for that specific container. The kubelet honors the setting of terminationGracePeriodSeconds for that container. For a failed readiness probe, the kubelet continues running the container that failed checks, and also continues to run more probes; because the check failed, the kubelet sets the Ready condition on the Pod to false.
- terminationGracePeriodSeconds: configure a grace period for the kubelet to wait between triggering a shut down of the failed container, and then forcing the container runtime to stop that container. The default is to inherit the Pod-level value for terminationGracePeriodSeconds (30 seconds if not specified), and the minimum value is 1. See probe-level terminationGracePeriodSeconds for more detail.

Caution:

Incorrect implementation of readiness probes may result in an ever growing number of processes in the container, and resource starvation if this is left unchecked.

HTTP probes

HTTP probes have additional fields that can be set on httpGet:

- host: Host name to connect to, defaults to the pod IP. You probably want to set "Host" in httpHeaders instead.
- scheme: Scheme to use for connecting to the host (HTTP or HTTPS). Defaults to "HTTP".
- path: Path to access on the HTTP server. Defaults to "/".
- httpHeaders: Custom headers to set in the request. HTTP allows repeated headers.
- port: Name or number of the port to access on the container. Number must be in the range 1 to 65535.

For an HTTP probe, the kubelet sends an HTTP request to the specified port and path to perform the check. The kubelet sends the probe to the Pod's IP address, unless the address is overridden by the optional host field in httpGet. If scheme field is set to HTTPS, the kubelet sends an HTTPS request skipping the certificate verification. In most scenarios, you do not want to set the host field. Here's one scenario where you would set it. Suppose the container listens on 127.0.0.1 and the Pod's hostNetwork field is true. Then host, under httpGet, should be set to 127.0.0.1. If your pod relies on virtual hosts, which is probably the more common case, you should not use host, but rather set the Host header in httpHeaders.

For an HTTP probe, the kubelet sends two request headers in addition to the mandatory Host header:

- User-Agent: The default value is kube-probe/1.33, where 1.33 is the version of the kubelet.
- Accept: The default value is */*.

You can override the default headers by defining httpHeaders for the probe. For example:

```
livenessProbe:
  httpGet:
    httpHeaders:
      - name: Accept
        value: application/json

startupProbe:
  httpGet:
    httpHeaders:
      - name: User-Agent
        value: MyUserAgent
```

You can also remove these two headers by defining them with an empty value.

```
livenessProbe:
  httpGet:
    httpHeaders:
      - name: Accept
        value: ""

startupProbe:
  httpGet:
    httpHeaders:
      - name: User-Agent
        value: ""
```

Note:
When the kubelet probes a Pod using HTTP, it only follows redirects if the redirect is to the same host. If the kubelet receives 11 or more redirects during probing, the probe is considered successful and a related Event is created:

```
Events:
  Type     Reason       Age                  From               Message
  ----     ------       ----                 ----               -------
  Normal   Scheduled    29m                  default-scheduler  Successfully assigned
default/httpbin-7b8bc9cb85-bjzwn to daocloud
  Normal   Pulling      29m                  kubelet            Pulling image
"docker.io/kennethreitz/httpbin"
  Normal   Pulled       24m                  kubelet            Successfully pulled image
"docker.io/kennethreitz/httpbin" in 5m12.402735213s
  Normal   Created      24m                  kubelet            Created container httpbin
  Normal   Started      24m                  kubelet            Started container httpbin
```

```
 Warning  ProbeWarning  4m11s (x1197 over 24m)  kubelet            Readiness probe warning:
Probe terminated redirects
```

If the kubelet receives a redirect where the hostname is different from the request, the outcome of the probe is treated as successful and kubelet creates an event to report the redirect failure.

TCP probes

For a TCP probe, the kubelet makes the probe connection at the node, not in the Pod, which means that you can not use a service name in the host parameter since the kubelet is unable to resolve it.

Probe-level terminationGracePeriodSeconds

FEATURE STATE: Kubernetes v1.28 [stable]

In 1.25 and above, users can specify a probe-level terminationGracePeriodSeconds as part of the probe specification. When both a pod- and probe-level terminationGracePeriodSeconds are set, the kubelet will use the probe-level value.

When setting the terminationGracePeriodSeconds, please note the following:

- The kubelet always honors the probe-level terminationGracePeriodSeconds field if it is present on a Pod.
- If you have existing Pods where the terminationGracePeriodSeconds field is set and you no longer wish to use per-probe termination grace periods, you must delete those existing Pods.

For example:

```yaml
spec:
  terminationGracePeriodSeconds: 3600  # pod-level
  containers:
  - name: test
    image: ...

    ports:
    - name: liveness-port
      containerPort: 8080

    livenessProbe:
      httpGet:
        path: /healthz
        port: liveness-port
      failureThreshold: 1
      periodSeconds: 60
      # Override pod-level terminationGracePeriodSeconds #
      terminationGracePeriodSeconds: 60
```