Lesson 1 : Application High Availability with Kubernetes

Describe how Kubernetes tries to keep applications running after failures.

Concepts of Deploying Highly Available Applications

Goal: make applications more robust and resistant to runtime failures, and subsequently disrupt business services / user transactions

In general, HA protect application from failing due to

application bugs

networking issues

cluster resources exhausted

memory leak

Writing Reliable Applications

- Cluster-level HA mitigates worst-case scenarios
- HA not substituation for fixing application-level issues
- Aim for reliability; application security is separate concern
- Applications work with k8s cluster to handle failure scenarios:
  - Tolerates restarts
  - Responds to health probes (startup, readiness, liveness probes)
  - Support multiple simultaneous instances
  - Has well-defined and well-behaved resources usage
  - Operate with restricted privileges
- Example 1: Most HTTP-based application provide endpoint to verify application health. So configure probe to observe this endpoint
- Example 2: Application respond with healthy status only when database is reachable.

Kubernetes Application Reliability

- Application crash → POD crash
- Application crash → POD still running
- POD running → Application not ready (delay, misconfigure)
- Application and POD still running, but missing important file/directory
- Kubernetes – HA technique to improve application reliability:
  - Restart policy: Never, Always, OnFailure
  - Probing: using health probes, cluster can check all the path-to-directory/IP/routing/permission is working
  - Horizontal scaling: when surge in demands, cluster can scale up number of replicas

Guided Exercise: Application High Availability with Kubernetes

You should be able to:

- Explore how the restartPolicy attribute affects crashing pods.
- Observe the behavior of a slow-starting application that has no configured probes.
- Use a deployment to scale the application, and observe the behavior of a broken pod.

Lesson 2 : Application Health Probes

Describe how Kubernetes uses health probes during deployment, scaling, and failover of applications.

Kubernetes Probes

- Enable cluster to determine status of application
- Repeatitively probe for response
- Use case scenarios:
    - Crash mitigation by automatically attempting to restart failing pods
    - Failover and load balancing by sending requests only to healthy pods
    - Monitoring by determining whether and when pods are failing
    - Scaling by determining when new replica is needed to process increase requests

Authoring Probe Endpoints

- Endpoints determine health and status of application
- Application developers write code to generate health probe endpoint
  - https://<application-name>>:8080/healthz
  - Application report successful health probe only if it can connect to configured database
- Endpoint best practice design
  - quick to perform
  - should not execute complex queries or
  - too many network calls
-

Probe Types

- **Readiness Probes** Determines whether the application is ready to serve requests.
- **Liveness Probes** Determine e whether the application container is in a healthy state.
- **Startup Probes** Determines when an application's startup is completed.

Readiness Probes

- Called throughout lifetime of application.
- Determines whether the application is ready to serve requests.
- **If the readiness probe fails**, then Kubernetes **remove the pod's IP address** from the service resource.
- Help detect temporary issues that might affect your applications.

For example:

- application might be temporarily unavailable when it starts, because it must *establish initial network connections, load files in a cache, or perform initial tasks* that *take time to complete*.
- application might occasionally need *to run long batch jobs*, which make it temporarily unavailable to clients.

Liveness Probes

- Called after application's initial start process and throughout lifetime of application
- Determine e whether the application container is in a healthy state.
- **If an application fails its** liveness probe enough times, then the cluster restarts the pod according to its restart policy

For example:

- Detect deadlock (application is running, but unable to make progress)
- Pod appear to be running, application code my not function correctly

Startup Probes

- Determines when an application's startup is completed.
- is not called after the probe succeeds.
- Liveness and readiness probes do not start until startup probe succeeds.
- **If the startup probe does not succeed** after a configurable timeout, then the pod is restarted based on its restartPolicy value.

For example:

- Applications with a long start time.

Type of Tests

- When defining probe, must specify one of following types to perform the test

HTTP GET Each time that the probe runs, the cluster sends a request to the specified HTTP endpoint. The test is considered a success if the request responds with an HTTP response code between 200 and 399. Other responses cause the test to fail.

Container command Each time that the probe runs, the cluster runs the specified command in the container. If the command exits with a status code of 0, then the test succeeds. Other status codes cause the test to fail.

TCP socket Each time that the probe runs, the cluster attempts to open a socket to the container. The test succeeds only if the connection is established

Timings and Thresholds

- All probes include timing variables: periodSeconds and failureThreshold
- **periodSeconds** specifies the frequency of running probes in interval of seconds
- **failureThresholds** define how many failed attempts before probe considered failed

Example: failureThresholds: 3; periodSeconds: 5

- scenario 1: After 15 seconds, if all probes don't get response then probe considered failed

- scenario 2: After 10 seconds, if third probe get response then probe succeeded

- Too short period can cause resources wastage
- Too few probes: false alarm, or may not be accurately gauge the situation

Adding Probes using YAML manifest file

```
apiVersion: apps/v1
kind: Deployment
...output omitted...
spec:
...output omitted...
  template:
    spec:
      containers:
      - name: web-server
        ...output omitted...
        livenessProbe: ❶
          failureThreshold: 6 ❷
          periodSeconds: 10 ❸
          httpGet: ❹
            path: /health ❺
            port: 3000 ❻
```

1. Defines a liveness probe.
2. Specifies how many times the probe must fail before mitigating.
3. Defines how often the probe runs.
4. Sets the probe as an HTTP request and defines the request port and path.
5. Specifies the HTTP path to send the request to.
6. Specifies the port to send the HTTP request over.

Can add to individual pod or deployment

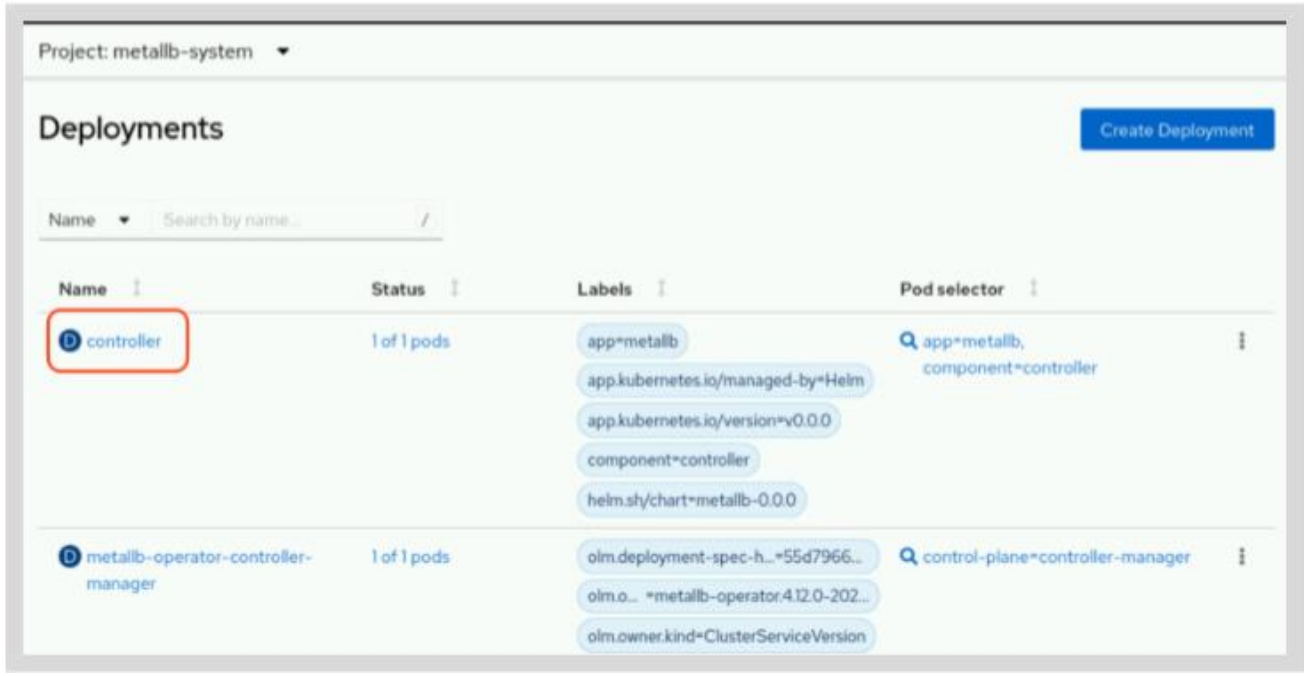
Adding Probes using **oc set probe** command

```
[user@host ~]$ oc set probe deployment/front-end \
--readiness \ ❶
--failure-threshold 6 \ ❷
--period-seconds 10 \ ❸
--get-url http://:8080/healthz ❹
```

1. Defines a readiness probe.
2. Sets how many times the probe must fail before mitigating.
3. Sets how often the probe runs.
4. Sets the probe as an HTTP request, and defines the request port and path.

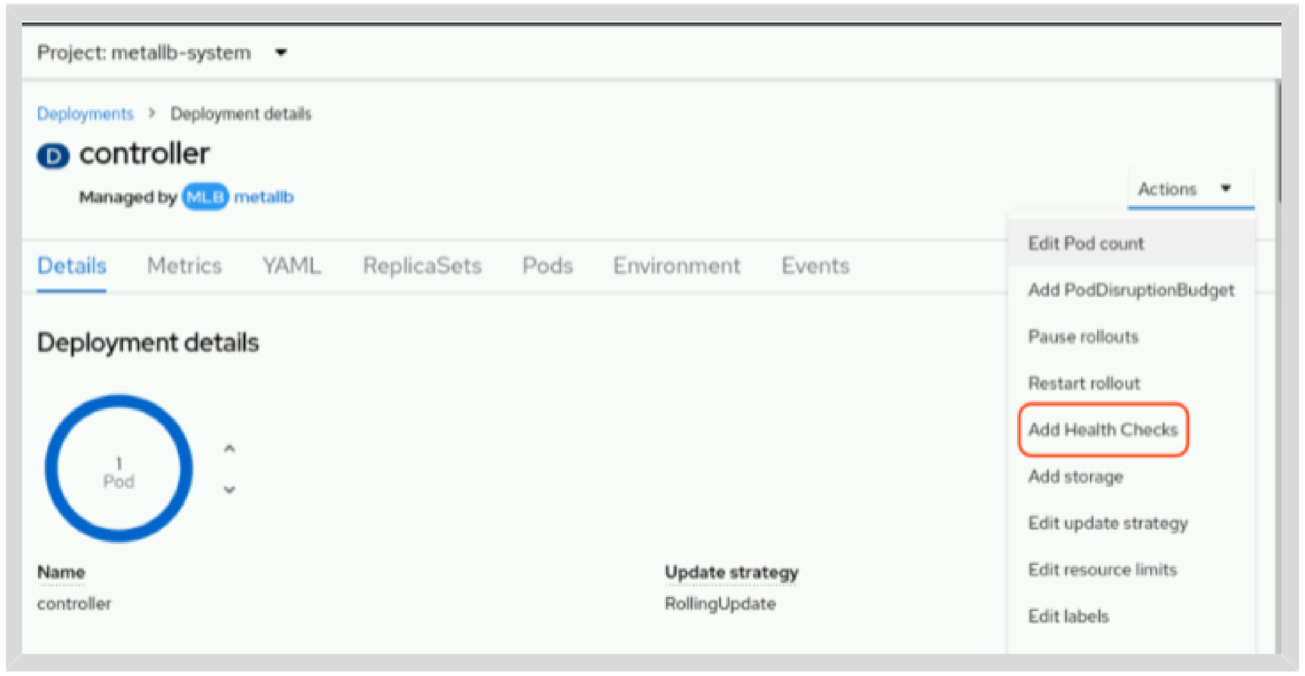
Adding Probes using Web Console

1. Navigate to **Workloads**
2. Select **Deployments** or **Pods**
3. **Actions → Add Health Checks**



Name	Status	Labels	Pod selector
controller	1 of 1 pods	app=metallb app.kubernetes.io/managed-by=Helm app.kubernetes.io/version=v0.0.0 component=controller helm.sh/chart=metallb-0.0.0	app=metallb, component=controller

metallb-operator-controller-manager	1 of 1 pods	olm.deployment-spec-h...=55d7966...
		olm.o... =metallb-operator.4.12.0-202...
		olm.owner.kind=ClusterServiceVersion
			control-plane=controller-manager



Name	Update strategy
controller	RollingUpdate

Guided Exercise: Application Health Probes

You should be able to:

- Observe potential issues with an application that is not configured with health probes.
- Configure startup, liveness, and readiness probes for the application.