Application High Availability with Kubernetes

Objectives

* Describe how Kubernetes tries to keep applications running after failures.

Concepts of Deploying Highly Available Applications

*High availability* (HA) is a goal of making applications more robust and resistant to runtime failures. Implementing HA techniques decreases the likelihood that an application is completely unavailable to users.

In general, HA can protect an application from failures in the following contexts:

* From itself in the form of application bugs
* From its environment, such as networking issues
* From other applications that exhaust cluster resources

Additionally, HA practices can protect the cluster from applications, such as one with a memory leak.

Writing Reliable Applications

At its core, cluster-level HA tooling mitigates worst-case scenarios. HA is not a substitute for fixing application-level issues, but augments developer mitigations. Although required for reliability, application security is a separate concern.

Applications must work with the cluster so that Kubernetes can best handle failure scenarios. Kubernetes expects the following behaviors from applications:

* Tolerates restarts
* Responds to health probes, such as the startup, readiness, and liveness probes
* Supports multiple simultaneous instances
* Has well-defined and well-behaved resource usage
* Operates with restricted privileges

Although the cluster can run applications that lack the preceding behaviors, applications with these behaviors better use the reliability and HA features that Kubernetes provides.

Most HTTP-based applications provide an endpoint to verify application health. The cluster can be configured to observe this endpoint and mitigate potential issues for the application.

The application is responsible for providing such an endpoint. Developers must decide how the application determines its state.

For example, if an application depends on a database connection, then the application might respond with a healthy status only when the database is reachable. However, not all applications that make database connections need such a check. This decision is at the discretion of the developers.

Kubernetes Application Reliability

If an application pod crashes, then it cannot respond to requests. Depending on the configuration, the cluster can automatically restart the pod. If the application fails without crashing the pod, then the pod does not receive requests. However, the cluster can do so only with the appropriate health probes.

Kubernetes uses the following HA techniques to improve application reliability:

* Restarting pods: By configuring a restart policy on a pod, the cluster restarts misbehaving instances of an application.
* Probing: By using health probes, the cluster knows when applications cannot respond to requests, and can automatically act to mitigate the issue.
* Horizontal scaling: When the application load changes, the cluster can scale the number of replicas to match the load.

These techniques are explored throughout this chapter.

Guided Exercise: Application High Availability with Kubernetes

Simulate different types of application failures and observe how Kubernetes handles them.

**Outcomes**

* 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.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that the following conditions are true:

* The reliability-ha project exists.
* The resource files are available in the course directory.
* The classroom registry has the long-load container image.

The long-load container image contains an application with utility endpoints. These endpoints perform such tasks as crashing the process and toggling the server's health status.

[student@workstation ~]$ **lab start reliability-ha**

**Instructions**

1. As the developer user, create a pod from a YAML manifest in the reliability-ha project.
   1. Log in as the developer user with the developer password.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**

*...output omitted...*

* 1. Select the reliability-ha project.
  2. [student@workstation ~]$ **oc project reliability-ha**

Now using project "reliability-ha" on server "https://api.ocp4.example.com:6443".

* 1. Navigate to the lab materials directory and view the contents of the pod definition. In particular, restartPolicy is set to Always.

[student@workstation ~]$ **cd DO180/labs/reliability-ha**

[student@workstation reliability-ha]$ **cat long-load.yaml**

apiVersion: v1

kind: Pod

metadata:

name: long-load

spec:

containers:

- image: registry.ocp4.example.com:8443/redhattraining/long-load:v1

name: long-load

securityContext:

allowPrivilegeEscalation: false

**restartPolicy: Always**

* 1. Create a pod by using the oc apply command.
  2. [student@workstation reliability-ha]$ **oc apply -f long-load.yaml**

pod/long-load created

* 1. Send a request to the pod to confirm that it is running and responding.
  2. [student@workstation reliability-ha]$ **oc exec long-load -- \**
  3. **curl -s localhost:3000/health**

Ok

1. Trigger the pod to crash, and observe that the restartPolicy instructs the cluster to re-create the pod.
   1. Observe that the pod is running and has not restarted.
   2. [student@workstation reliability-ha]$ **oc get pods**
   3. NAME READY STATUS RESTARTS AGE

long-load 1/1 Running **0** 1m

* 1. Send a request to the /destruct endpoint in the application. This request triggers the process to crash.
  2. [student@workstation reliability-ha]$ **oc exec long-load -- \**
  3. **curl -s localhost:3000/destruct**

command terminated with exit code 52

* 1. Observe that the pod is running and restarted one time.
  2. [student@workstation reliability-ha]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

long-load 1/1 Running **1** (34s ago) 4m16s

* 1. Delete the long-load pod.
  2. [student@workstation reliability-ha]$ **oc delete pod long-load**

pod "long-load" deleted

The pod is not re-created, because it was created manually, and not via a workload resource such as a deployment.

1. Use a restart policy of Never to create the pod, and observe that it is not re-created on crashing.
   1. Modify the long-load.yaml file so that the restartPolicy is set to Never.
   2. *...output omitted...*

restartPolicy: **Never**

* 1. Create the pod with the updated YAML file.
  2. [student@workstation reliability-ha]$ **oc apply -f long-load.yaml**

pod/long-load created

* 1. Send a request to the pod to confirm that the pod is running and that the application is responding.
  2. [student@workstation reliability-ha]$ **oc exec long-load -- \**
  3. **curl -s localhost:3000/health**

Ok

* 1. Send a request to the /destruct endpoint in the application to crash it.
  2. [student@workstation reliability-ha]$ **oc exec long-load -- \**
  3. **curl -s localhost:3000/destruct**

command terminated with exit code 52

* 1. Observe that the pod is not restarted and is in an error state.
  2. [student@workstation reliability-ha]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

long-load 0/1 **Error** 0 2m36s

* 1. Delete the long-load pod.
  2. [student@workstation reliability-ha]$ **oc delete pod long-load**

pod "long-load" deleted

1. Because the cluster does not know when the application inside the pod is ready to receive requests, you must add a startup delay to the application. Adding this capability by using probes is covered in a later exercise.
   1. Update the long-load.yaml file by adding a startup delay and use a restart policy of Always. Set the START\_DELAY variable to 60,000 milliseconds (one minute) so that the file looks like the following excerpt:
   2. *...output omitted...*
   3. spec:
   4. containers:
   5. - image: registry.ocp4.example.com:8443/redhattraining/long-load:v1
   6. imagePullPolicy: Always
   7. securityContext:
   8. allowPrivilegeEscalation: false
   9. name: long-load
   10. **env:**
   11. **- name: START\_DELAY**
   12. **value: "60000"**

**restartPolicy: Always**

**Note**

Although numbers are a valid YAML type, environment variables must be passed as strings. YAML syntax is also indentation-sensitive.

For these reasons, ensure that your file appears *exactly* as the preceding example.

* 1. Apply the YAML file to create the pod and proceed within one minute to the next step.
  2. [student@workstation reliability-ha]$ **oc apply -f long-load.yaml**

pod/long-load created

* 1. Within a minute of pod creation, verify the status of the pod. The status shows as ready even though it is not. Try to send a request to the application, and observe that it fails.
  2. [student@workstation reliability-ha]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

long-load 1/1 Running 0 16s

[student@workstation reliability-ha]$ **oc exec long-load -- \**

**curl -s localhost:3000/health**

app is still starting

* 1. After waiting a minute for the application to start, send another a request to the pod to confirm that it is running and responding.
  2. [student@workstation reliability-ha]$ **oc exec long-load -- \**
  3. **curl -s localhost:3000/health**

Ok

1. Use a deployment to scale up the number of deployed pods. Observe that deleting the pods causes service outages, even though the deployment handles re-creating the pods.
   1. Review the long-load-deploy.yaml file, which defines a deployment, service, and route. The deployment creates three replicas of the application pod.

In each pod, a START\_DELAY environment variable is set to 15,000 milliseconds (15 seconds). In each pod, the application responds that it is not ready until after the delay.

[student@workstation reliability-ha]$ **cat long-load-deploy.yaml**

*...output omitted...*

spec:

containers:

- image: registry.ocp4.example.com:8443/redhattraining/long-load:v1

imagePullPolicy: Always

name: long-load

env:

**- name: START\_DELAY**

**value: "15000"**

*...output omitted...*

* 1. Start the load test script, which sends a request to the /health API endpoint of the application every two seconds. Leave the script running in a visible terminal window.
  2. [student@workstation reliability-ha]$ **./load-test.sh**

*...output omitted...*

* 1. In a new terminal window, apply the ~/DO180/labs/reliability-ha/long-load-deploy.yaml file.
  2. [student@workstation ~]$ **oc apply -f \**
  3. **~/DO180/labs/reliability-ha/long-load-deploy.yaml**
  4. deployment.apps/long-load created
  5. service/long-load created

route.route.openshift.io/long-load created

* 1. Watch the output of the load test script as the pods and the application instances start. After a delay, the requests succeed.
  2. *...output omitted...*
  3. Ok
  4. Ok
  5. Ok

*...output omitted...*

* 1. By using the /togglesick API endpoint of the application, put one of the three pods into a broken state.
  2. [student@workstation ~]$ **curl \**
  3. **long-load-reliability-ha.apps.ocp4.example.com/togglesick**

***no output expected***

* 1. Watch the output of the load test script as some requests start failing. Because of the load balancer, the exact order of the output is random.
  2. *...output omitted...*
  3. Ok
  4. app is unhealthy
  5. app is unhealthy
  6. Ok
  7. Ok

*...output omitted...*

Press **Ctrl**+**C** to end the load test script.

* 1. Return to the /home/student/ directory.
  2. [student@workstation reliability-ha]$ **cd /home/student/**

[student@workstation ~]$

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish reliability-ha**

Application Health Probes

Objectives

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

Kubernetes Probes

Health probes are an important part of maintaining a robust cluster. *Probes* enable the cluster to determine the status of an application by repeatedly probing it for a response.

A set of health probes affect a cluster's ability to do the following tasks:

* 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 a new replica is ready to receive requests

Authoring Probe Endpoints

Application developers are expected to code health probe endpoints during application development. These endpoints determine the health and status of the application. For example, a data-driven application might report a successful health probe only if it can connect to the database.

Because the cluster calls them often, health probe endpoints should be quick to perform. Endpoints should not perform complicated database queries or many network calls.

Probe Types

Kubernetes provides the following types of probes: startup, readiness, and liveness. Depending on the application, you might configure one or more of these types.

Readiness Probes

A *readiness probe* determines whether the application is ready to serve requests. If the readiness probe fails, then Kubernetes prevents client traffic from reaching the application by removing the pod's IP address from the service resource.

Readiness probes help to detect temporary issues that might affect your applications. For example, the 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. The application might occasionally need to run long batch jobs, which make it temporarily unavailable to clients.

Kubernetes continues to run the probe even after the application fails. If the probe succeeds again, then Kubernetes adds back the pod's IP address to the service resource, and requests are sent to the pod again.

In such cases, the readiness probe addresses a temporary issue and improves application availability.

Liveness Probes

Like a readiness probe, a *liveness probe* is called throughout the lifetime of the application. Liveness probes determine 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.

Unlike a startup probe, liveness probes are called after the application's initial start process. Usually, this mitigation is by restarting or re-creating the pod.

Startup Probes

A *startup probe* determines when an application's startup is completed. Unlike a liveness probe, a startup probe is not called after the probe succeeds. If the startup probe does not succeed after a configurable timeout, then the pod is restarted based on its restartPolicy value.

Consider adding a startup probe to applications with a long start time. By using a startup probe, the liveness probe can remain short and responsive.

Types of Tests

When defining a probe, you must specify one of the following types of test to perform:

**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 the types of probes include timing variables. The *period seconds* variable defines how often the probe runs. The *failure threshold* defines how many failed attempts are required before the probe itself fails.

For example, a probe with a failure threshold of 3 and period seconds of 5 can fail up to three times before the overall probe fails. Using this probe configuration means that the issue can exist for 10 seconds before it is mitigated. However, running probes too often can waste resources. Consider these values when setting probes.

Adding Probes via YAML

Because probes are defined on a pod template, probes can be added to workload resources such as deployments. To add a probe to an existing deployment, update and apply the YAML file or use the oc edit command. For example, the following YAML excerpt defines a deployment pod template with a probe:

apiVersion: apps/v1

kind: Deployment

*...output omitted...*

spec:

*...output omitted...*

template:

spec:

containers:

- name: web-server

*...output omitted...*

livenessProbe:

failureThreshold: 6

periodSeconds: 10

httpGet:

path: /health

port: 3000

|  |  |
| --- | --- |
|  | Defines a liveness probe. |
|  | Specifies how many times the probe must fail before mitigating. |
|  | Defines how often the probe runs. |
|  | Sets the probe as an HTTP request and defines the request port and path. |
|  | Specifies the HTTP path to send the request to. |
|  | Specifies the port to send the HTTP request over. |

Adding Probes via the CLI

The oc set probe command adds or modifies a probe on a deployment. For example, the following command adds a readiness probe to a deployment called front-end:

[user@host ~]$ **oc set probe deployment/front-end \**

**--readiness \**

**--failure-threshold 6 \**

**--period-seconds 10 \**

**--get-url http://:8080/healthz**

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

Adding Probes via the Web Console

To add or modify a probe on a deployment from the web console, navigate to the **Workloads** → **Deployments** menu and select a deployment.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Click **Actions** and then click **Add Health Checks**.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Click **Edit probe** to specify the readiness type, the HTTP headers, the path, the port, and more.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Guided Exercise: Application Health Probes

Configure health probes in a deployment and verify that network clients are insulated from application failures.

**Outcomes**

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

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that the following conditions are true:

* The reliability-probes project exists.
* The resource files are available in the course directory.
* The classroom registry has the long-load container image.

The registry.ocp4.example.com:8443/redhattraining/long-load:v1 container image contains an application with utility endpoints. These endpoints perform such tasks as crashing the process and toggling the server's health status.

[student@workstation ~]$ **lab start reliability-probes**

**Instructions**

1. As the developer user, deploy the long-load application in the reliability-probes project.
   1. Log in as the developer user with the developer password.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**
   4. Login successful.

*...output omitted...*

* 1. Set the reliability-probes project as the active project.
  2. [student@workstation ~]$ **oc project reliability-probes**

Now using project reliability-probes on server "https://api.ocp4.example.com:6443".

* 1. Apply the long-load-deploy.yaml file to create the pod. Move to the next step within one minute.
  2. [student@workstation ~]$ **oc apply -f \**
  3. **~/DO180/labs/reliability-probes/long-load-deploy.yaml**
  4. deployment.apps/long-load created
  5. service/long-load created

route.route.openshift.io/long-load created

* 1. Verify that the pods take several minutes to start by sending a request to a pod in the deployment.
  2. [student@workstation ~]$ **oc exec deploy/long-load -- \**
  3. **curl -s localhost:3000/health**

app is still starting

* 1. Observe that the pods are listed as ready even though the application is not ready.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. long-load-8564d998cc-579nx **1/1** Running 0 30s
  5. long-load-8564d998cc-ttqpg **1/1** Running 0 30s

long-load-8564d998cc-wjtfw **1/1** Running 0 30s

1. Add a startup probe to the pods so that the cluster knows when the pods are ready.
   1. Modify the ~/DO180/labs/reliability-probes/long-load-deploy.yaml YAML file by defining a startup probe. The probe runs every three seconds and triggers a pod as failed after 30 failed attempts. The file should match the following excerpt:
   2. *...output omitted...*
   3. spec:
   4. *...output omitted...*
   5. template:
   6. *...output omitted...*
   7. spec:
   8. containers:
   9. - image: registry.ocp4.example.com:8443/redhattraining/long-load:v1
   10. imagePullPolicy: Always
   11. name: long-load
   12. **startupProbe:**
   13. **failureThreshold: 30**
   14. **periodSeconds: 3**
   15. **httpGet:**
   16. **path: /health**
   17. **port: 3000**
   18. env:

*...output omitted...*

* 1. Scale down the deployment to zero replicas.
  2. [student@workstation ~]$ **oc scale deploy/long-load --replicas 0**

deployment.apps/long-load scaled

**Note**

Red Hat recommends scaling down an application to zero replicas before deleting or changing a deployment. Scaling down to zero replicas stops any new pods from being created while enabling exiting pods to terminate gracefully after finishing all current requests. For multiple deployment updates, scaling down to zero prevents a noisy event log from Kubernetes responding to every change, and conserves cluster resources.

* 1. Apply the updated long-load-deploy.yaml file. Because the YAML file specifies the number of replicas, the deployment is scaled up. Move to the next step within one minute.
  2. [student@workstation ~] **oc apply -f \**
  3. **~/DO180/labs/reliability-probes/long-load-deploy.yaml**
  4. deployment.apps/long-load configured
  5. service/long-load unchanged

route.route.openshift.io/long-load configured

* 1. Observe that the pods do not show as ready until the application is ready and the startup probe succeeds. Wait for the three pods to reach the ready state. Press **Ctrl**+**c** to stop the watch command.
  2. [student@workstation ~]$ **watch oc get pods**
  3. Every 2.0s: oc get pods
  4. NAME READY STATUS RESTARTS AGE
  5. long-load-785b5b4fc8-7x5ln **1/1** Running 0 90s
  6. long-load-785b5b4fc8-f7pdk **1/1** Running 0 90s

long-load-785b5b4fc8-r2nqj **1/1** Running 0 90s

1. Add a liveness probe so that broken instances of the application are restarted.
   1. Start the load test script.
   2. [student@workstation ~]$ **~/DO180/labs/reliability-probes/load-test.sh**
   3. Ok
   4. Ok
   5. Ok

*...output omitted...*

Keep the script running in a visible window.

* 1. In a new terminal window, use the /togglesick endpoint to make one of the pods unhealthy. Move to the next step within one minute.
  2. [student@workstation ~]$ **oc exec \**
  3. **deploy/long-load -- curl -s localhost:3000/togglesick**

***no output expected***

The load test window begins to show app is unhealthy. Because only one pod is unhealthy, the remaining pods still respond with Ok.

* 1. Update the ~/DO180/labs/reliability-probes/long-load-deploy.yaml file to add a liveness probe. The probe runs every three seconds and triggers the pod as failed after three failed attempts. Modify the spec.template.spec.containers object in the file to match the following excerpt.
  2. spec:
  3. *...output omitted...*
  4. template:
  5. *...output omitted...*
  6. spec:
  7. containers:
  8. - image: registry.ocp4.example.com:8443/redhattraining/long-load:v1
  9. *...output omitted...*
  10. startupProbe:
  11. failureThreshold: 30
  12. periodSeconds: 3
  13. httpGet:
  14. path: /health
  15. port: 3000
  16. **livenessProbe:**
  17. **failureThreshold: 3**
  18. **periodSeconds: 3**
  19. **httpGet:**
  20. **path: /health**
  21. **port: 3000**
  22. env:

*...output omitted...*

* 1. Scale down the deployment to zero replicas.
  2. [student@workstation ~]$ **oc scale deploy/long-load --replicas 0**

deployment.apps/long-load scaled

The load test script shows that the application is not available.

* 1. Apply the updated long-load-deploy.yaml file to update the deployment, which triggers the deployment to re-create its pods.
  2. [student@workstation ~]$ **oc apply -f \**
  3. **~/DO180/labs/reliability-probes/long-load-deploy.yaml**
  4. deployment.apps/long-load configured
  5. service/long-load unchanged

route.route.openshift.io/long-load configured

* 1. Wait for the three new pods to reach the ready state. Press **Ctrl**+**c** to stop the watch command.
  2. [student@workstation ~]$ **watch oc get pods**
  3. Every 2.0s: oc get pods
  4. NAME READY STATUS RESTARTS AGE
  5. long-load-785b5b4fc8-8x5ln **1/1** Running 0 70s
  6. long-load-785b5b4fc8-f8pdk **1/1** Running 0 70s

long-load-785b5b4fc8-r9nqj **1/1** Running 0 70s

* 1. Wait for the load test window to show Ok for all responses, and then toggle one of the pods to be unhealthy.
  2. [student@workstation ~]$ **oc exec \**
  3. **deploy/long-load -- curl -s localhost:3000/togglesick**

***no output expected***

The load test window might show app is unhealthy a number of times before the pod is restarted.

* 1. Observe that the unhealthy pod is restarted after the liveness probe fails. After the pod is restarted, the load test window shows only Ok.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. long-load-fbb7468d9-8xm8j 1/1 Running 0 9m42s
  5. long-load-fbb7468d9-k66dm 1/1 Running 0 8m38s

long-load-fbb7468d9-ncxkh 0/1 Running **1** (11s ago) 10m

1. Add a readiness probe so that traffic goes only to pods that are ready and healthy.
   1. Scale down the deployment to zero replicas.
   2. [student@workstation ~]$ **oc scale deploy/long-load --replicas 0**

deployment.apps/long-load scaled

* 1. Use the oc set probe command to add the readiness probe.
  2. [student@workstation ~]$ **oc set probe deploy/long-load \**
  3. **--readiness --failure-threshold 1 --period-seconds 3 \**
  4. **--get-url http://:3000/health**

deployment.apps/long-load probes updated

* 1. Scale up the deployment to three replicas.
  2. [student@workstation ~]$ **oc scale deploy/long-load --replicas 3**

deployment.apps/long-load scaled

* 1. Observe the status of the pods by using a watch command.
  2. [student@workstation ~]$ **watch oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. long-load-d5794d744-8hqlh 0/1 Running 0 48s
  5. long-load-d5794d744-hphgb 0/1 Running 0 48s

long-load-d5794d744-lgkns 0/1 Running 0 48s

The command does not immediately finish, but continues to show updates to the pods' status. Leave this command running in a visible window.

* 1. Wait for the pods to show as ready. Then, in a new terminal window, make one of the pods unhealthy for five seconds by using the /hiccup endpoint.
  2. [student@workstation ~]$ **oc exec \**
  3. **deploy/long-load -- curl -s localhost:3000/hiccup?time=5**

***no output expected***

The pod status window shows that one of the pods is no longer ready. After five seconds, the pod is healthy again and shows as ready.

The load test window might show app is unhealthy one time before the pod is set as not ready. After the cluster determines that the pod is no longer ready, it stops sending traffic to the pod until either the pod is fixed or the liveness probe fails. Because the pod is sick only for five seconds, it is enough time for the readiness probe to fail, but not the liveness probe.

**Note**

Optionally, repeat this step and observe as the temporarily sick pod's status changes.

* 1. Stop the load test and status commands by pressing **Ctrl**+**c** in their respective windows.

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish reliability-probes**

Reserve Compute Capacity for Applications

Objectives

* Configure an application with resource requests so Kubernetes can make scheduling decisions.

Kubernetes Pod Scheduling

The Red Hat OpenShift Container Platform (RHOCP) pod scheduler determines the placement of new pods onto nodes in the cluster. The pod scheduler algorithm follows a three-step process:

**Filtering nodes**

A pod can define a node selector that matches the labels in the cluster nodes. Only labels that match are eligible.

Additionally, the scheduler filters the list of running nodes by evaluating each node against a set of predicates. A pod can define *resource requests* for compute resources such as CPU, memory, and storage. Only nodes with enough available computer resources are eligible.

The filtering step reduces the list of eligible nodes. In some cases, the pod could run on any of the nodes. In other cases, all the nodes are filtered out, so the pod cannot be scheduled until a node with the appropriate prerequisites becomes available.

If all nodes are filtered out, then a FailedScheduling event is generated for the pod.

**Prioritizing the filtered list of nodes**

By using multiple priority criteria, the scheduler determines a weighted score for each node. Nodes with higher scores are better candidates to run the pod.

**Selecting the best fit node**

The candidate list is sorted according to these scores, and the node with the highest score is selected to host the pod. If multiple nodes have the same high score, then one node is selected in a round-robin fashion. After a host is selected, then a Scheduled event is generated for the pod.

The scheduler is flexible and can be customized for advanced scheduling situations. Customizing the scheduler is outside the scope of this course.

Compute Resource Requests

For such applications that require a specific amount of compute resources, you can define a resource request in the pod definition of your application deployment. The resource requests assign hardware resources for the application deployment.

Resource requests specify the minimum required compute resources necessary to schedule a pod. The scheduler tries to find a node with enough compute resources to satisfy the pod requests.

In Kubernetes, memory resources are measured in bytes, and CPU resources are measured in CPU units. CPU units are allocated by using millicore units. A millicore is a CPU core, either virtual or physical, that is split into 1000 units. A request value of "1000 m" allocates an entire CPU core to a pod. You can also use fractional values to allocate CPU resources. For example, you can set the CPU resource request to a 0.1 value, which represents 100 millicores (100 m). Likewise, a CPU resource request with a 1.0 value represents an entire CPU or 1000 millicores (1000 m).

You can define resource requests for each container in either a deployment or a deployment configuration resource. If resources are not defined, then the container specification shows a resources: {} line.

In your deployment, modify the resources: {} line to specify the chosen requests. The following example defines a resource request of 100 millicores (100 m) of CPU and one gigabyte (1 Gi) of memory.

*...output omitted...*

spec:

containers:

- image: quay.io/redhattraining/hello-world-nginx:v1.0

name: hello-world-nginx

resources:

requests:

cpu: "100m"

memory: "1Gi"

If you use the edit command to modify a deployment or a deployment configuration, then ensure that you use the correct indentation. Indentation mistakes can result in the editor refusing to save changes. Alternatively, use the set resources command that the kubectl and oc commands provide, to specify resource requests. The following command sets the same requests as the preceding example:

[user@host ~]$ **oc set resources deployment hello-world-nginx \**

**--requests cpu=10m,memory=1gi**

The set resource command works with any resource that includes a pod template, such as the deployments and job resources.

Inspecting Cluster Compute Resources

Cluster administrators can view the available and used compute resources of a node. For example, as a cluster administrator, you can use the describe node command to determine the compute resource capacity of a node. The command shows the capacity of the node and how much of that capacity is allocatable. It also displays the amount of allocated and requested resources on the node.

[user@host ~]$ **oc describe node master01**

Name: master01

Roles: control-plane,master,worker

*...output omitted...*

Capacity:

cpu: 8

ephemeral-storage: 125293548Ki

hugepages-1Gi: 0

hugepages-2Mi: 0

memory: 20531668Ki

pods: 250

Allocatable:

cpu: 7500m

ephemeral-storage: 114396791822

hugepages-1Gi: 0

hugepages-2Mi: 0

memory: 19389692Ki

pods: 250

*...output omitted...*

Non-terminated Pods: (88 in total)

... Name CPU Requests CPU Limits Memory Requests Memory Limits ...

... ---- ------------ ---------- --------------- -------------

... controller-... 10m (0%) 0 (0%) 20Mi (0%) 0 (0%) ...

... metallb-... 50m (0%) 0 (0%) 20Mi (0%) 0 (0%) ...

... metallb-... 0 (0%) 0 (0%) 0 (0%) 0 (0%) ...

Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)

Resource **Requests** Limits

-------- **--------** ------

cpu **3183m (42%)** 1202m (16%)

memory **12717Mi (67%)** 1350Mi (7%)

*...output omitted...*

RHOCP cluster administrators can also use the oc adm top pods command. This command shows the compute resource usage for each pod in a project. You must include the --namespace or -n options to specify a project. Otherwise, the command returns the resource usage for pods in the currently selected project.

The following command displays the resource usage for pods in the openshift-dns project:

[user@host ~]$ **oc adm top pods -n openshift-dns**

NAME CPU(cores) MEMORY(bytes)

dns-default-5kpn5 1m 33Mi

node-resolver-6kdxp 0m 2Mi

Additionally, cluster administrators can use the oc adm top node command to view the resource usage of a cluster node. Include the node name to view the resource usage of a particular node.

[user@host ~]$ **oc adm top node master01**

NAME CPU(cores) CPU% MEMORY(bytes) MEMORY%

master01 1250m 16% 10268Mi 54%

Limit Compute Capacity for Applications

Objectives

* Configure an application with resource limits so Kubernetes can protect other applications from it.

Memory and CPU requests that you define for containers help Red Hat OpenShift Container Platform (RHOCP) to select a compute node to run your pods. However, these resource requests do not restrict the memory and CPU that the containers can use. For example, setting a memory request at 1 GiB does not prevent the container from consuming more memory.

Red Hat recommends that you set the memory and CPU requests to the peak usage of your application. In contrast, by setting lower values, you overcommit the node resources. If all the applications that are running on the node start to use resources above the values that they request, then the compute nodes might run out of memory and CPU.

In addition to requests, you can set memory and CPU *limits* to prevent your applications from consuming too many resources.

Setting Memory Limits

A memory limit specifies the amount of memory that a container can use across all its processes.

As soon as the container reaches the limit, the compute node selects and then kills a process in the container. When that event occurs, RHOCP detects that the application is not working any more, because the main container process is missing, or because the health probes report an error. RHOCP then restarts the container according to the pod restartPolicy attribute, which defaults to Always.

RHOCP relies on Linux kernel features to implement resource limits, and to kill processes in containers that reach their memory limits:

**Control groups (cgroups)**

RHOCP uses control groups to implement resource limits. Control groups are a Linux kernel mechanism for controlling and monitoring system resources, such as CPU and memory.

**Out-of-Memory killer (OOM killer)**

When a container reaches its memory limit, the Linux kernel triggers the OOM killer subsystem to select and then kill a process.

You must set a memory limit when the application has a memory usage pattern that you must mitigate, such as when the application has a memory leak. A memory leak is a bug in the application, which occurs when the application uses some memory but does not free it after use. If the leak appears in an infinite service loop, then the application uses more and more memory over time, and can end up consuming all the available memory on the system. For these applications, setting a memory limit prevents them from consuming all the node's memory. The memory limit also enables OpenShift to regularly restart applications to free up their memory when they reach the limit.

To set a memory limit for the container in a pod, use the oc set resources command:

[user@host ~]$ **oc set resources deployment/hello --limits memory=1Gi**

In addition to the oc set resources command, you can define resource limits from a file in the YAML format:

apiVersion: apps/v1

kind: Deployment

*...output omitted...*

spec:

containers:

- image: registry.access.redhat.com/ubi9/nginx-120:1-86

name: hello

resources:

requests:

cpu: 100m

memory: 500Mi

**limits:**

cpu: 200m

**memory: 1Gi**

When RHOCP restarts a pod because of an OOM event, it updates the pod's lastState attribute, and sets the reason to OOMKilled:

[user@host ~]$ **oc get pod hello-67645f4865-vvr42 -o yaml**

*...output omitted...*

status:

*...output omitted...*

containerStatuses:

- containerID: cri-o://806b...9fe7

image: registry.access.redhat.com/ubi9/nginx-120:1-86

imageID: registry.access.redhat.com/ubi9/nginx-120:1-86@sha256:1403...fd34

**lastState:**

terminated:

containerID: cri-o://bbc4...9eb2

exitCode: 137

finishedAt: "2023-03-08T07:56:06Z"

**reason: OOMKilled**

startedAt: "2023-03-08T07:51:43Z"

name: hello

ready: true

restartCount: 1

*...output omitted...*

To set a memory limit for the container in a pod from the web console, select a deployment, and click **Actions** → **Edit resource limits**.

|  |
| --- |
| A screenshot of a phone  Description automatically generated |

Set memory limits by increasing or decreasing the memory on the **Limit** section.

|  |
| --- |
| A screenshot of a computer error message  Description automatically generated |

Setting CPU Limits

CPU limits work differently from memory limits. When the container reaches the CPU limit, RHOCP inhibits the container's processes, even if the node has available CPU cycles. The application continues to work, but at a slower pace.

In contrast, if you do not set a CPU limit, then the container can consume as much CPU as is available on the node. If the node's CPU is under pressure, for example because several containers are running CPU-intensive tasks, then the Linux kernel shares the CPU resource between all these containers, according to the CPU requests value for the containers.

You must set a CPU limit when you require a consistent application behavior across clusters and nodes. For example, if the application runs on a node where the CPU is available, then the application can execute at full speed. On the other hand, if the application runs on a node with CPU pressure, then the application executes at a slower pace.

The same behavior can occur between your development and production clusters. Because the two environments might have different node configurations, the application might run differently when you move it from development to production.

**Note**

Clusters can have differences in hardware configuration beyond what limits observe. For example, two clusters' nodes might have CPUs with equal core count and unequal clock speeds.

Requests and limits do not account for these hardware differences. If your clusters differ in such a way, take care that requests and limits are appropriate for both configurations.

By setting a CPU limit, you mitigate the differences between the configuration of the nodes, and you experience a more consistent behavior.

To set a CPU limit for the container in a pod, use the oc set resources command:

[user@host ~]$ **oc set resources deployment/hello --limits cpu=200m**

You can also define CPU limits from a file in the YAML format:

apiVersion: apps/v1

kind: Deployment

*...output omitted...*

spec:

containers:

- image: registry.access.redhat.com/ubi9/nginx-120:1-86

name: hello

resources:

requests:

cpu: 100m

memory: 500Mi

**limits:**

**cpu: 200m**

memory: 1Gi

To set a CPU limit for the container in a pod from the web console, select a deployment, and click **Actions** → **Edit resource limits**.

|  |
| --- |
| A screenshot of a phone  Description automatically generated |

Set CPU limits by increasing or decreasing the CPU on the **Limit** section.

|  |
| --- |
| A screenshot of a computer  Description automatically generated |

Viewing Requests, Limits, and Actual Usage

By using the RHOCP command-line interface, cluster administrators can view compute usage information on individual nodes. The oc describe node command displays detailed information about a node, including information about the pods that are running on the node. For each pod, it shows CPU requests and limits, as well as memory requests and limits. If you do not specify a request or limit, then the pod shows a zero for that column. The command also displays a summary of all the resource requests and limits.

[user@host ~]$ **oc describe node master01**

Name: master01

Roles: control-plane,master,worker

*...output omitted...*

Non-terminated Pods: (88 in total)

... Name CPU Requests CPU Limits Memory Requests Memory Limits ...

... ---- ------------ ---------- --------------- -------------

... controller-... 10m (0%) 0 (0%) 20Mi (0%) 0 (0%) ...

... metallb-... 50m (0%) 0 (0%) 20Mi (0%) 0 (0%) ...

... metallb-... 0 (0%) 0 (0%) 0 (0%) 0 (0%) ...

*...output omitted...*

Allocated resources:

(Total limits may be over 100 percent, i.e., overcommitted.)

Resource Requests **Limits**

-------- -------- **------**

cpu 3183m (42%) **1202m (16%)**

memory 12717Mi (67%) **1350Mi (7%)**

*...output omitted...*

The oc describe node command displays requests and limits. The oc adm top command shows resource usage. The oc adm top nodes command shows the resource usage for nodes in the cluster. You must run this command as the cluster administrator.

The oc adm top pods command shows the resource usage for each pod in a project.

The following command displays the resource usage for the pods in the current project:

[user@host ~]$ **oc adm top pods -n openshift-console**

NAME CPU(cores) MEMORY(bytes)

hello-67645f4865-vvr42 121m 309Mi

intradb-6f8d7cfffb-fz55b 0m 440Mi

To visualize the consumption of resources from the web console, select a deployment, and click the **Metrics** tab. From this tab, you can view the usage for memory, CPU, the file system, and incoming and outgoing traffic.

|  |
| --- |
| A screenshot of a computer screen  Description automatically generated |

Application Autoscaling

Objectives

* Configure a horizontal pod autoscaler for an application.

Kubernetes can autoscale a deployment based on current load on the application pods, by means of a HorizontalPodAutoscaler (HPA) resource type.

A horizontal pod autoscaler resource uses performance metrics that the OpenShift Metrics subsystem collects. The Metrics subsystem comes preinstalled in OpenShift. To autoscale a deployment, you must specify resource requests for pods so that the horizontal pod autoscaler can calculate the percentage of usage.

The autoscaler works in a loop. Every 15 seconds by default, it performs the following steps:

* The autoscaler retrieves the details of the metric for scaling from the HPA resource.
* For each pod that the HPA resource targets, the autoscaler collects the metric from the metric subsystem.
* For each targeted pod, the autoscaler computes the usage percentage, from the collected metric and from the pod resource requests.
* The autoscaler computes the average usage and the average resource requests across all the targeted pods. It establishes a usage ratio from these values, and then uses the ratio for its scaling decision.

The simplest way to create a horizontal pod autoscaler resource is by using the oc autoscale command, for example:

[user@host ~]$ **oc autoscale deployment/hello --min 1 --max 10 --cpu-percent 80**

The previous command creates a horizontal pod autoscaler resource that changes the number of replicas on the hello deployment to keep its pods under 80% of their total requested CPU usage.

The oc autoscale command creates a horizontal pod autoscaler resource by using the name of the deployment as an argument (hello in the previous example).

The maximum and minimum values for the horizontal pod autoscaler resource accommodate bursts of load and avoid overloading the OpenShift cluster. If the load on the application changes too quickly, then it might help to keep several spare pods to cope with sudden bursts of user requests. Conversely, too many pods can use up all cluster capacity and impact other applications that use the same OpenShift cluster.

To get information about horizontal pod autoscaler resources in the current project, use the oc get command. For example:

[user@host ~]$ **oc get hpa**

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS ...

hello Deployment/hello <unknown>/80% 1 10 1 ...

scale Deployment/scale 60%/80% 2 10 2 ...

**Important**

The horizontal pod autoscaler initially has a value of <unknown> in the TARGETS column. It might take up to five minutes before <unknown> changes to display a percentage for current usage.

A persistent value of <unknown> in the TARGETS column might indicate that the deployment does not define resource requests for the metric. The horizontal pod autoscaler does not scale these pods.

Pods that are created by using the oc create deployment command do not define resource requests. Using the OpenShift autoscaler might therefore require editing the deployment resources, creating custom YAML or JSON resource files for your application, or adding limit range resources to your project that define default resource requests.

In addition to the oc autoscale command, you can create a horizontal pod autoscaler resource from a file in the YAML format.

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: hello

spec:

minReplicas: 1

maxReplicas: 10

metrics:

- resource:

name: cpu

target:

averageUtilization: 80

type: Utilization

type: Resource

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: hello

|  |  |
| --- | --- |
|  | Minimum number of pods. |
|  | Maximum number of pods. |
|  | Ideal average CPU usage for each pod. If the global average CPU usage is above that value, then the horizontal pod autoscaler starts new pods. If the global average CPU usage is below that value, then the horizontal pod autoscaler deletes pods. |
|  | Reference to the name of the deployment resource. |

Use the oc apply -f *hello-hpa*.yaml command to create the resource from the file.

The preceding example creates a horizontal pod autoscaler resource that scales based on CPU usage. Alternatively, it can scale based on memory usage by setting the resource name to memory, as in the following example:

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: hello

spec:

minReplicas: 1

maxReplicas: 10

metrics:

- resource:

name: **memory**

target:

averageUtilization: 80

*...output omitted...*

To create a horizontal pod autoscaler resource from the web console, click the **Workloads** → **HorizontalPodAutoscalers** menu. Click **Create HorizontalPodAutoscaler** and customize the YAML manifest.

**Note**

If an application uses more overall memory as the number of replicas increases, then it cannot be used with memory-based autoscaling.

Guided Exercise: Application Autoscaling

Configure an autoscaler for an application and then load test that application to observe scaling up.

**Outcomes**

You should be able to manually scale up a deployment, configure a horizontal pod autoscaler resource, and monitor the autoscaler.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that all resources are available for this exercise. It also creates the reliability-autoscaling project.

[student@workstation ~]$ **lab start reliability-autoscaling**

**Instructions**

1. Log in to the OpenShift cluster as the developer user with the developer password. Use the reliability-autoscaling project.
   1. Log in to the OpenShift cluster.
   2. [student@workstation ~]$ **oc login -u developer -p developer \**
   3. **https://api.ocp4.example.com:6443**
   4. Login successful.

*...output omitted...*

* 1. Set the reliability-autoscaling project as the active project.
  2. [student@workstation ~]$ **oc project reliability-autoscaling**

*...output omitted...*

1. Create the loadtest deployment, service, and route. The deployment uses the registry.ocp4.example.com:8443/redhattraining/loadtest:v1.0 container image that provides a web application. The web application exposes an API endpoint that creates a CPU-intensive task when queried.
   1. Review the ~/DO180/labs/reliability-autoscaling/loadtest.yml resource file that the lab command prepared. The container specification does not include the resources section that you use to specify CPU requests and limits. You configure that section in another step. Do not change the file for now.
   2. apiVersion: v1
   3. kind: List
   4. metadata: {}
   5. items:
   6. - apiVersion: apps/v1
   7. kind: Deployment
   8. *...output omitted...*
   9. spec:
   10. containers:
   11. - image: registry.ocp4.example.com:8443/redhattraining/loadtest:v1.0
   12. name: loadtest
   13. readinessProbe:
   14. failureThreshold: 3
   15. httpGet:
   16. path: /api/loadtest/v1/healthz
   17. port: 8080
   18. scheme: HTTP
   19. periodSeconds: 10
   20. successThreshold: 1
   21. timeoutSeconds: 1
   22. - apiVersion: v1
   23. kind: Service
   24. *...output omitted...*
   25. - apiVersion: route.openshift.io/v1
   26. kind: Route

*...output omitted...*

* 1. Use the oc apply command to create the application.
  2. [student@workstation ~]$ **oc apply -f \**
  3. **~/DO180/labs/reliability-autoscaling/loadtest.yml**
  4. deployment.apps/loadtest created
  5. service/loadtest created

route.route.openshift.io/loadtest created

* 1. Wait for the pod to start. You might have to rerun the command several times for the pod to report a Running status. The name of the pod on your system probably differs.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

loadtest-65c55b7dc-r4s4s 1/1 **Running** 0 49s

1. Configure a horizontal pod autoscaler resource for the loadtest deployment. Set the minimum number of replicas to 2 and the maximum to 20. Set the average CPU usage to 50% of the CPU requests attribute.

The horizontal pod autoscaler does not work, because the loadtest deployment does not specify requests for CPU usage.

* 1. Use the oc autoscale command to create the horizontal pod autoscaler resource.
  2. [student@workstation ~]$ **oc autoscale deployment/loadtest --min 2 --max 20 \**
  3. **--cpu-percent 50**

horizontalpodautoscaler.autoscaling/loadtest autoscaled

* 1. Retrieve the status of the loadtest horizontal pod autoscaler resource. The unknown value in the TARGETS column indicates that OpenShift cannot compute the current CPU usage of the loadtest deployment. The deployment must include the CPU requests attribute for OpenShift to be able to compute the CPU usage.
  2. [student@workstation ~]$ **oc get hpa loadtest**
  3. NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

loadtest Deployment/loadtest **<unknown>**/50% 2 20 2 74s

* 1. Get more details about the resource status. You might have to rerun the command several times. Wait three minutes for the command to report the warning message.
  2. [student@workstation ~]$ **oc describe hpa loadtest**
  3. Name: loadtest
  4. Namespace: reliability-autoscaling
  5. *...output omitted...*
  6. Conditions:
  7. Type Status Reason Message
  8. ---- ------ ------ -------
  9. AbleToScale True SucceededGetScale the HPA controller was able to get the target's current scale
  10. ScalingActive False **FailedGetResourceMetric** the HPA was unable to compute the replica count: failed to get cpu utilization: **missing request for cpu**
  11. Events:
  12. Type ... Message
  13. ---- ... -------
  14. *...output omitted...*
  15. Warning ... **failed to get cpu utilization: missing request for cpu**

*...output omitted...*

* 1. Delete the horizontal pod autoscaler resource. You re-create the resource in another step, after you fix the loadtest deployment.
  2. [student@workstation ~]$ **oc delete hpa loadtest**

horizontalpodautoscaler.autoscaling "loadtest" deleted

* 1. Delete the loadtest application.
  2. [student@workstation ~]$ **oc delete -f \**
  3. **~/DO180/labs/reliability-autoscaling/loadtest.yml**
  4. deployment.apps "loadtest" deleted
  5. service "loadtest" deleted

route.route.openshift.io "loadtest" deleted

1. Add a CPU resource section to the ~/DO180/labs/reliability-autoscaling/loadtest.yml file. Redeploy the application from the file.
   1. Edit the ~/DO180/labs/reliability-autoscaling/loadtest.yml file, and configure the CPU limits and requests for the loadtest deployment. The pods need 25 millicores to operate, and must not consume more that 100 millicores.

You can compare your work with the completed ~/DO180/solutions/reliability-autoscaling/loadtest.yml file that the lab command prepared.

*...output omitted...*

spec:

containers:

- image: registry.ocp4.example.com:8443/redhattraining/loadtest:v1.0

name: loadtest

readinessProbe:

failureThreshold: 3

httpGet:

path: /api/loadtest/v1/healthz

port: 8080

scheme: HTTP

periodSeconds: 10

successThreshold: 1

timeoutSeconds: 1

**resources:**

**requests:**

**cpu: 25m**

**limits:**

**cpu: 100m**

*...output omitted...*

* 1. Use the oc apply command to deploy the application from the file.
  2. [student@workstation ~]$ **oc apply -f \**
  3. **~/DO180/labs/reliability-autoscaling/loadtest.yml**
  4. deployment.apps/loadtest created
  5. service/loadtest created

route.route.openshift.io/loadtest created

* 1. Wait for the pod to start. You might have to rerun the command several times for the pod to report a Running status. The name of the pod on your system probably differs.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

loadtest-667bdcdc99-vhc9x 1/1 **Running** 0 36s

1. Manually scale the loadtest deployment by first increasing and then decreasing the number of running pods.
   1. Scale up the loadtest deployment to five pods.
   2. [student@workstation ~]$ **oc scale deployment/loadtest --replicas 5**

deployment.apps/loadtest scaled

* 1. Confirm that all five application pods are running. You might have to rerun the command several times for all the pods to report a Running status. The name of the pods on your system probably differ.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE
  4. loadtest-667bdcdc99-5fcvh 1/1 **Running** 0 43s
  5. loadtest-667bdcdc99-dpspr 1/1 **Running** 0 42s
  6. loadtest-667bdcdc99-hkssk 1/1 **Running** 0 43s
  7. loadtest-667bdcdc99-vhc9x 1/1 **Running** 0 8m11s

loadtest-667bdcdc99-z5n9q 1/1 **Running** 0 43s

* 1. Scale down the loadtest deployment back to one pod.
  2. [student@workstation ~]$ **oc scale deployment/loadtest --replicas 1**

deployment.apps/loadtest scaled

* 1. Confirm that only one application pod is running. You might have to rerun the command several times for the pods to terminate.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

loadtest-667bdcdc99-vhc9x 1/1 **Running** 0 11m

1. Configure a horizontal pod autoscaler resource for the loadtest deployment. Set the minimum number of replicas to 2 and the maximum to 20. Set the average CPU usage to 50% of the CPU request attribute.
   1. Use the oc autoscale command to create the horizontal pod autoscaler resource.
   2. [student@workstation ~]$ **oc autoscale deployment/loadtest --min 2 --max 20 \**
   3. **--cpu-percent 50**

horizontalpodautoscaler.autoscaling/loadtest autoscaled

* 1. Open a new terminal window and run the watch command to monitor the oc get hpa loadtest command. Wait five minutes for the loadtest horizontal pod autoscaler to report usage in the TARGETS column.

Notice that the horizontal pod autoscaler scales up the deployment to two replicas, to conform with the minimum number of pods that you configured.

[student@workstation ~]$ **watch oc get hpa loadtest**

Every 2.0s: oc get hpa loadtest workstation: Fri Mar 3 06:26:24 2023

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

loadtest Deployment/loadtest **0%/50%** 2 20 **2** 52s

Leave the command running, and do not interrupt it.

1. Increase the CPU usage by sending requests to the loadtest application API.
   1. Use the oc get route command to retrieve the URL of the application.
   2. [student@workstation ~]$ **oc get route loadtest**
   3. NAME HOST/PORT ...

loadtest **loadtest-reliability-autoscaling.apps.ocp4.example.com** ...

* 1. Send a request to the application API to simulate additional CPU pressure on the container. Do not wait for the curl command to complete, and continue with the exercise. After a minute, the command reports a timeout error that you can ignore.
  2. [student@workstation ~]$ **curl \**
  3. **loadtest-reliability-autoscaling.apps.ocp4.example.com/api/loadtest/v1/cpu/1**
  4. <html><body><h1>504 Gateway Time-out</h1>
  5. The server didn't respond in time.

</body></html>

* 1. Watch the output of the oc get hpa loadtest command in the second terminal. After a minute, the horizontal pod autoscaler detects an increase in the CPU usage and deploys additional pods.

**Note**

The increased activity of the application does not immediately trigger the autoscaler. Wait a few moments if you do not see any changes to the number of replicas.

You might need to run the curl command multiple times before the application uses enough CPU to trigger the autoscaler.

The CPU usage and the number of replicas on your system probably differ.

Every 2.0s: oc get hpa loadtest workstation: Fri Mar 3 07:20:19 2023

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

loadtest Deployment/loadtest **220%/50%** 2 20 **9** 16m

* 1. Wait five minutes after the curl command completes. The oc get hpa loadtest command shows that the CPU load decreases.

**Note**

Although the horizontal pod autoscaler resource can be quick to scale up, it is slower to scale down.

Every 2.0s: oc get hpa loadtest workstation: Fri Mar 3 07:23:11 2023

NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

loadtest Deployment/loadtest **0%/50%** 2 20 9 18m

* 1. **Optional**: Wait for the loadtest application to scale down. It takes five additional minutes for the horizontal pod autoscaler to scale down to two replicas.
  2. Every 2.0s: oc get hpa loadtest workstation: Fri Mar 3 07:29:12 2023
  3. NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

loadtest Deployment/loadtest 0%/50% 2 20 **2** 24m

* 1. Press **Ctrl**+**C** to quit the watch command. Close that second terminal when done.

**Finish**

On the workstation machine, use the lab command to complete this exercise. This step is important to ensure that resources from previous exercises do not impact upcoming exercises.

[student@workstation ~]$ **lab finish reliability-autoscaling**

Lab: Configure Applications for Reliability

Deploy and troubleshoot a reliable application that defines health probes, compute resource requests, and compute resource limits so it can run N instances per node; and configure a horizontal pod autoscaler that will scale to a maximum of N instances.

**Outcomes**

You should be able to add resource requests to a Deployment object, configure probes, and create a horizontal pod autoscaler resource.

As the student user on the workstation machine, use the lab command to prepare your system for this exercise.

This command ensures that all resources are available for this exercise. It also creates the reliability-review project and deploys the longload application in that project.

[student@workstation ~]$ **lab start reliability-review**

**Instructions**

The API URL of your OpenShift cluster is https://api.ocp4.example.com:6443, and the oc command is already installed on your workstation machine.

Log in to the OpenShift cluster as the developer user with the developer password.

Use the reliability-review project for your work.

1. The longload application in the reliability-review project fails to start. Diagnose and then fix the issue. The application needs 512 MiB of memory to work.

After you fix the issue, you can confirm that the application works by running the ~/DO180/labs/reliability-review/curl\_loop.sh script that the lab command prepared. The script sends requests to the application in a loop. For each request, the script displays the pod name and the application status. Press **Ctrl**+**C** to quit the script.

* 1. Log in to the OpenShift cluster.
  2. [student@workstation ~]$ **oc login -u developer -p developer \**
  3. **https://api.ocp4.example.com:6443**
  4. Login successful.

*...output omitted...*

* 1. Set the reliability-review project as the active project.
  2. [student@workstation ~]$ **oc project reliability-review**

*...output omitted...*

* 1. List the pods in the project. The pod is in the Pending status. The name of the pod on your system probably differs.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

longload-64bf8dd776-b6rkz 0/1 **Pending** 0 8m1s

* 1. Retrieve the events for the pod. No compute node has enough memory to accommodate the pod.
  2. [student@workstation ~]$ **oc describe pod longload-*64bf8dd776-b6rkz***
  3. Name: longload-64bf8dd776-b6rkz
  4. Namespace: reliability-review
  5. *...output omitted...*
  6. Events:
  7. Type Reason Age From Message
  8. ---- ------ ---- ---- -------

Warning FailedScheduling 8m default-scheduler 0/1 nodes are available: 1 **Insufficient memory**. preemption: 0/1 nodes are available: 1 No preemption victims found for incoming pod.

* 1. Review the resource requests for memory. The longload deployment requests 8 GiB of memory.
  2. [student@workstation ~]$ **oc get deployment longload -o \**
  3. **jsonpath='{.spec.template.spec.containers[0].resources.requests.memory}{"\n"}'**

8Gi

* 1. Set the memory requests to 512 MiB. Ignore the warning message.
  2. [student@workstation ~]$ **oc set resources deployment/longload \**
  3. **--requests memory=512Mi**

deployment.apps/longload resource requirements updated

* 1. Wait for the pod to start. You might have to rerun the command several times for the pod to report a Running status. The name of the pod on your system probably differs.
  2. [student@workstation ~]$ **oc get pods**
  3. NAME READY STATUS RESTARTS AGE

longload-5897c9558f-cx4gt 1/1 **Running** 0 86s

* 1. Run the ~/DO180/labs/reliability-review/curl\_loop.sh script to confirm that the application works.
  2. [student@workstation ~]$ **~/DO180/labs/reliability-review/curl\_loop.sh**
  3. 1 curl: (7) Failed to connect to master01.ocp4.example.com port 30372: Connection refused
  4. 2 longload-5897c9558f-cx4gt: app is still starting
  5. 3 longload-5897c9558f-cx4gt: app is still starting
  6. 4 longload-5897c9558f-cx4gt: app is still starting
  7. 5 longload-5897c9558f-cx4gt: **Ok**
  8. 6 longload-5897c9558f-cx4gt: **Ok**
  9. 7 longload-5897c9558f-cx4gt: **Ok**
  10. 8 longload-5897c9558f-cx4gt: **Ok**

*...output omitted...*

Press **Ctrl**+**C** to quit the script.

Hide Solution

1. When the application scales up, your customers complain that some requests fail. To replicate the issue, manually scale up the longload application to three replicas, and run the ~/DO180/labs/reliability-review/curl\_loop.sh script at the same time.

The application takes seven seconds to initialize. The application exposes the /health API endpoint on HTTP port 3000. Configure the longload deployment to use this endpoint, to ensure that the application is ready before serving client requests.

* 1. Open a new terminal window and run the ~/DO180/labs/reliability-review/curl\_loop.sh script.
  2. [student@workstation ~]$ **~/DO180/labs/reliability-review/curl\_loop.sh**
  3. 1 longload-5897c9558f-cx4gt: Ok
  4. 2 longload-5897c9558f-cx4gt: Ok
  5. 3 longload-5897c9558f-cx4gt: Ok
  6. 4 longload-5897c9558f-cx4gt: Ok

*...output omitted...*

Leave the script running and do not interrupt it.

* 1. Scale up the application to three replicas.
  2. [student@workstation ~]$ **oc scale deployment/longload --replicas 3**

deployment.apps/longload scaled

* 1. Watch the output of the curl\_loop.sh script in the second terminal. Some requests fail because OpenShift sends requests to the new pods before the application is ready.
  2. *...output omitted...*
  3. 22 longload-5897c9558f-cx4gt: Ok
  4. 23 longload-5897c9558f-cx4gt: Ok
  5. 24 longload-5897c9558f-cx4gt: Ok
  6. 25 curl: (7) Failed to connect to master01.ocp4.example.com port 30372: Connection refused
  7. 26 curl: (7) Failed to connect to master01.ocp4.example.com port 30372: Connection refused
  8. 27 longload-5897c9558f-cx4gt: Ok
  9. 28 curl: (7) Failed to connect to master01.ocp4.example.com port 30372: Connection refused
  10. 29 longload-5897c9558f-cx4gt: Ok
  11. 30 curl: (7) Failed to connect to master01.ocp4.example.com port 30372: Connection refused
  12. 31 longload-5897c9558f-tpssf: app is still starting
  13. 32 longload-5897c9558f-kkvm5: app is still starting
  14. 33 longload-5897c9558f-cx4gt: Ok
  15. 34 longload-5897c9558f-tpssf: app is still starting
  16. 35 longload-5897c9558f-tpssf: app is still starting
  17. 36 longload-5897c9558f-tpssf: app is still starting
  18. 37 longload-5897c9558f-cx4gt: Ok
  19. 38 longload-5897c9558f-tpssf: app is still starting
  20. 39 longload-5897c9558f-cx4gt: Ok
  21. 40 longload-5897c9558f-cx4gt: Ok

*...output omitted...*

Leave the script running and do not interrupt it.

* 1. Add a readiness probe to the longload deployment. Ignore the warning message.
  2. [student@workstation ~]$ **oc set probe deployment/longload --readiness \**
  3. **--initial-delay-seconds 7 \**
  4. **--get-url http://:3000/health**

deployment.apps/longload probes updated

* 1. Scale down the application back to one pod.
  2. [student@workstation ~]$ **oc scale deployment/longload --replicas 1**

deployment.apps/longload scaled

If scaling down breaks the curl\_loop.sh script, then press **Ctrl**+**c** to stop the script in the second terminal. Then, restart the script.

* 1. To test your work, scale up the application to three replicas again.
  2. [student@workstation ~]$ **oc scale deployment/longload --replicas 3**

deployment.apps/longload scaled

* 1. Watch the output of the curl\_loop.sh script in the second terminal. No request fails.
  2. *...output omitted...*
  3. 92 longload-7ddcc9b7fd-72dtm: Ok
  4. 93 longload-7ddcc9b7fd-72dtm: Ok
  5. 94 longload-7ddcc9b7fd-72dtm: Ok
  6. 95 longload-7ddcc9b7fd-qln95: Ok
  7. 96 longload-7ddcc9b7fd-wrxrb: Ok
  8. 97 longload-7ddcc9b7fd-qln95: Ok
  9. 98 longload-7ddcc9b7fd-wrxrb: Ok
  10. 99 longload-7ddcc9b7fd-72dtm: Ok

*...output omitted...*

Press **Ctrl**+**C** to quit the script.

Hide Solution

1. Configure the application so that it automatically scales up when the average memory usage is above 60% of the memory requests value, and scales down when the usage is below this percentage. The minimum number of replicas must be one, and the maximum must be three. The resource that you create for scaling the application must be named longload.

The lab command provides the ~/DO180/labs/reliability-review/hpa.yml resource file as an example. Use the oc explain command to learn the valid parameters for the hpa.spec.metrics.resource.target attribute. Because the file is incomplete, you must update it first if you choose to use it.

To test your work, use the oc exec deploy/longload — curl localhost:3000/leak command to sends an HTTP request to the application /﻿leak API endpoint. Each request consumes an additional 480 MiB of memory. To free this memory, you can use the ~/DO180/labs/reliability-review/free.sh script.

* 1. Before you create the horizontal pod autoscaler resource, scale down the application to one pod.
  2. [student@workstation ~]$ **oc scale deployment/longload --replicas 1**

deployment.apps/longload scaled

* 1. Edit the ~/DO180/labs/reliability-review/hpa.yml resource file. You can retrieve the parameters for the resource attribute by using the oc explain hpa.spec.metrics.resource and oc explain hpa.spec.metrics.resource.target commands.
  2. apiVersion: autoscaling/v2
  3. kind: HorizontalPodAutoscaler
  4. metadata:
  5. name: longload
  6. labels:
  7. app: longload
  8. spec:
  9. maxReplicas: **3**
  10. minReplicas: **1**
  11. scaleTargetRef:
  12. apiVersion: apps/v1
  13. kind: Deployment
  14. name: longload
  15. metrics:
  16. - type: Resource
  17. resource:
  18. **name: memory**
  19. **target:**
  20. **type: Utilization**

**averageUtilization: 60**

* 1. Use the oc apply command to deploy the horizontal pod autoscaler.
  2. [student@workstation ~]$ **oc apply -f ~/DO180/labs/reliability-review/hpa.yml**

horizontalpodautoscaler.autoscaling/longload created

* 1. In the second terminal, run the watch command to monitor the oc get hpa longload command. Wait for the longload horizontal pod autoscaler to report usage in the TARGETS column. The percentage on your system probably differs.
  2. [student@workstation ~]$ **watch oc get hpa longload**
  3. Every 2.0s: oc get hpa longload workstation: Fri Mar 10 05:15:34 2023
  4. NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

longload Deployment/longload **13%/60%** 1 3 1 75s

Leave the command running and do not interrupt it.

* 1. To test your work, run the oc exec deploy/longload — curl localhost:3000/leak command in the first terminal for the application to allocate 480 MiB of memory.
  2. [student@workstation ~]$ **oc exec deploy/longload -- curl -s localhost:3000/leak**

longload-7ddcc9b7fd-72dtm: consuming memory!

* 1. In the second terminal, after two minutes, the oc get hpa longload command shows the memory increase. The horizontal pod autoscaler scales up the application to more than one replica. The percentage on your system probably differs.
  2. Every 2.0s: oc get hpa longload workstation: Fri Mar 10 05:19:44 2023
  3. NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

longload Deployment/longload **145%/60%** 1 3 **2** 5m18s

* 1. To test your work, run the ~/DO180/labs/reliability-review/free.sh script in the first terminal for the application to release the memory. Ensure that the pod that frees the memory is the same pod that was consuming memory. Execute the free.sh script several times if necessary.
  2. [student@workstation ~]$ **~/DO180/labs/reliability-review/free.sh**

longload-7ddcc9b7fd-72dtm: releasing memory!

* 1. In the second terminal, after ten minutes, the oc get hpa longload command shows the memory decrease. The horizontal pod autoscaler scales down the application to one replica. The percentage on your system probably differs.
  2. Every 2.0s: oc get hpa longload workstation: Fri Mar 10 05:19:44 2023
  3. NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS AGE

longload Deployment/longload **12%/60%** 1 3 **1** 15m28s

Press **Ctrl**+**C** to quit the watch command. Close that second terminal when done.

Hide Solution

Summary

* A highly available application is resistant to scenarios that might otherwise make it unavailable.
* Kubernetes and RHOCP provide HA features, such as health probes, that help the cluster to route traffic only to working pods.
* Resource requests and limits help to keep cluster node resource usage balanced.
* Horizontal pod autoscalers automatically add or remove replicas based on current resource usage and specified parameters.