Container Orchestration Using Kubernetes

Understanding Kubernetes Workloads

Learning Objectives

By the end of this lesson, you will be able to:

- Create a Kubernetes pod to comprehend its lifecycle
- Configure a namespace with a resource quota to manage resources within a Kubernetes container
- Deploy a multitier application to understand the deployment process in Kubernetes
- Design and implement a horizontal pod autoscaler (HPA) to achieve scalability
- Construct a Kubernetes ReplicaSet to achieve high availability

Overview of Workloads

Overview

A workload is an application running on Kubernetes.

It consists of one or more components working together, running inside a set of pods.

It defines and manages how applications run on a cluster, representing the processes or applications running in containers.

It handles tasks such as deploying, scaling, and managing pods.

Workload Resources

Kubernetes provides several built-in workload resources:

- Deployment and ReplicaSet
- StatefulSet
- DaemonSet
- Job and CronJob

Users can add in a third-party workload resource if they want a specific behavior that is not part of Kubernetes' core.

Workload Resources

Deployment and ReplicaSet

- A Deployment manages a set of pods to run an application workload, usually one that doesn't maintain state.
- A ReplicaSet aims to maintain a stable set of replica pods running at any given time.

StatefulSet

- A StatefulSet runs a group of pods and maintains a sticky identity for each of those pods.
- It manages applications that need persistent storage or a stable, unique network identity.

Workload Resources

DaemonSet

- A DaemonSet defines pods that provide node-local facilities.
- It ensures all (or some) nodes run a copy of pod, adds pods to new nodes, removes them from deleted nodes, and cleans up pods when the DaemonSet is deleted.

Job and CronJob

- A Job creates one or more pods to execute a task until completion, ensuring that the specified number of successful completions is achieved.
- A CronJob executes one-time Jobs on a repeating schedule.

Understanding Pods

Overview of Pods

Pods are the smallest deployable units of computing that users can create and manage in Kubernetes.

A pod is a group of one or more containers with shared storage, network, resources, and a specification for running the containers.

Its contents are always co-located, co-scheduled, and run in a shared context.

It models an application-specific **logical host** that contains one or more application containers that are relatively tightly coupled.

Using Pods

Pods are not created directly; instead, they are created using a Deployment or Job for workloads. If pods need to track state, the **StatefulSet** resource is used to create stateful pods.

Pods in a Kubernetes cluster can be used for:

Running a single container

Running multiple containers working together

Pods and Controllers

New pods are assigned to a cluster node, with a controller managing their replication, updates, and recovery from failures.

Some examples of workload resources that manage one or more pods are:

Pod Templates

Pod templates are specifications that help users to create pods. They are included in workload resources such as **Deployments**, **Jobs**, and **DaemonSets.**

The sample below is a basic pod manifest with a single-container template:

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
# Name of the pod
spec:
  containers:
# List of Containers
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Pods Templates

Controllers for workload resources create and manage pods from a pod template.

Each workload resource implements its own rules for handling changes to the pod template.

Modifying the pod template or switching to a new pod template has no direct effect on the pods that already exist.

The **kubelet** does not directly observe or manage any of the details around pod templates and updates.

Pod Update and Replacement

Pod update operations like **patch** and **replace** have some limitations:

- The metadata about a pod is immutable.
- If the **metadata.deletionTimestamp** is set, no new entry can be added to the **metadata.finalizers** list.
- Pod updates may not change fields other than **spec.containers[*].image, spec.initContainers[*].image, spec.activeDeadlineSeconds,** or **spec.tolerations**.
- Only two updates are allowed for **spec.activeDeadlineSeconds**: setting an unassigned field with a positive number and updating the field to a smaller non-negative value.

Resource Sharing and Communication

Pods enable data sharing and communication among their constituent containers, facilitating efficient data exchange and communication within the pod.

Storage in pods

All containers in the pod can access the shared volumes, allowing those containers to share data.

Pod networking

Within a pod, containers share an IP address and port space and can find each other via localhost.

Privileged Mode for Containers

Any container in a pod can enable **Privileged Mode** using the privileged flag on the security context of the container spec.

This is useful for containers that want to use operating system administrative capabilities such as manipulating the network stack or accessing hardware devices.

Sidecar Container of a Pod

The sidecar containers of a pod support functionalities such as logging, monitoring, service discovery, security, and more.

Quick Check

You are managing a Kubernetes cluster and need to ensure that a workload consisting of multiple containers is always running and automatically recovers from failure. The workload should scale based on traffic. Which Kubernetes object is the best fit for managing this requirement?

A. Pod

B. ReplicaSet

C. Deployment

D. DaemonSet

Understanding Pod Lifecycle

Pods Lifecycle

Pods are created, assigned a unique ID (UID), and scheduled to nodes where they remain until termination (according to restart policy) or deletion.

- If a node dies, the pods scheduled to that node are scheduled for deletion after a timeout period.
- Kubernetes uses a higher-level abstraction called controller, which handles the work of managing the relatively disposable pod instances.
- A given pod (as defined by a UID) is never **rescheduled** to a different Node.
- A pod can be replaced by a new, near-identical pod with even the same name, if desired but with a different UID.

Pods and Volumes

A multi-container pod contains a file puller and a web server that use a persistent volume for shared storage between the containers.

Source: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/

Phases of a Pod

Phases of a Pod

A pod phase represents its stage in the lifecycle, indicated by the **phase** field in its **podStatus** object. Possible phase values include:

Value	Description
Pending	The pod has been accepted by the Kubernetes cluster, but one or more of the containers has not been set up and made ready to run.
Running	The pod has been bound to a node, and all the containers have been created.
Succeeded	All containers in the pod have terminated successfully and will not be restarted.

Phases of a Pod

When a node in a Kubernetes cluster fails or loses connection, Kubernetes marks all pods on that node as failed. Possible pod phases include:

Value	Description
Failed	All containers in the pod have terminated, and at least one container has terminated in failure.
Unknown	This phase typically occurs due to an error in communicating with the node where the pod should be running.

Container States

Kubernetes tracks the state of each container inside a pod. Each state has a specific meaning:

Running
The **Running** status indicates that a container is executing without issues.

Waiting
If a container is not in either the **Running** or **Terminated** state, it is **Waiting**.

Terminated
A container in the terminated state began execution and then either ran to completion or failed for some reason.

Container Restart Policy

The **spec** of a pod has a **restartPolicy** field with possible values **Always, OnFailure,** and **Never**.

1. The **restartPolicy** applies to all containers in the pod.

2. **restartPolicy** only refers to restarts of the containers by the **kubelet** on the same node.

Pod Conditions

A pod has a **PodStatus**, which has an array of **PodConditions** through which the pod has passed or not. Following are the **PodConditions**:

- PodScheduled
- ContainersReady
- Initialized
- Ready
- PodReadyToStart Containers (beta feature)

Pod Conditions

Each **podConditions** is described by several fields that offer insights into the Pod's status and the reasons behind any state transitions. The key fields are outlined below:

Field Name	Description
type	Represents the **name** of the pod condition
status	Indicates whether that condition is applicable, with possible values, such as **True**, **False**, or **Unknown**
lastProbeTime	Represents the **timestamp** of when the pod condition was **last probed**
lastTransitionTime	Represents the **timestamp** for when the pod **last transitioned** from one status to another
reason	Represents **machine-readable**, **UpperCamelCase** text indicating the reason for the condition's last transition
message	Represents **human-readable** message indicating details about the last status transition

Termination of Pods

Pods represent running processes on cluster nodes, so it's crucial to give them a time called grace period to clean up the running resources rather than being abruptly killed.

If the **kubelet** or the container runtime's management service is restarted while waiting for processes to terminate, the cluster retries from the start, including the full original grace period.

Termination of Pods

In Kubernetes, pod termination follows a controlled process designed to ensure graceful shutdown and minimal disruption to services. The following outlines the process of terminating a pod:

Termination signal

A **SIGTERM** signal is sent to the Pod's containers to initiate shutdown.

Grace period

Kubernetes waits for the pod to stop, clean up the resources, and prevent new traffic.

PreStop hook

If configured, the **PreStop** hook runs custom logic before shutdown.

Forcible termination

If the Pod does not shut, Kubernetes forcefully stops it with **SIGKILL**.

Pod deletion

After termination, the Pod and associated resources are deleted from the cluster

Forced Pod Termination

Forced deletions can be disruptive for some workloads and their pods, but Kubernetes provides options to manage this process effectively, which includes:

2

Graceful deletion: Modifying the **--grace-period=<seconds>** flag to a desired time, ensuring clean shutdown within required time

1

Immediate force deletion: Using **--force** with **--grace-period=0** to immediately delete a pod

3

StatefulSets caution: Deleting StatefulSet pods, requiring additional steps for proper management

Forced pod termination is useful when Kubernetes pods are stuck in a terminating state.

Assisted Practice

Creating a Kubernetes Pod **Duration: 10 Min.**

Problem statement:

You have been assigned a task to create and describe a Kubernetes pod to understand its lifecycle.

Outcome:

By completing this demo, you will be able to create and describe a Kubernetes pod by defining its configuration, deploying it using a YAML file, and inspecting its lifecycle and status.

Note: Refer to the demo document for detailed steps:
01_Creating_a_Kubernetes_Pod

Assisted Practice: Guidelines

Steps to be followed:

1. Create and describe a Kubernetes pod

ASSISTED PRACTICE

Assisted Practice

Creating a Multi-Container Pod **Duration: 10 Min.**

Problem statement:

You have been assigned a task to create a multi-container pod in a Kubernetes cluster that allows you to run and interact with multiple containers within a single pod.

Outcome:

By completing this demo, you will be able to create, manage, and access a multi-container Kubernetes pod, allowing multiple containers to interact within the same pod environment.

Note: Refer to the demo document for detailed steps:
02_Creating_a_Multi-Container_Pod

Assisted Practice: Guidelines

Steps to be followed:

1. Create and access a multi-container pod

ASSISTED PRACTICE

Quick Check

You have a pod running a critical database. It enters the **Terminating** phase but remains stuck. The database container hasn't released file locks in the persistent volume. What is the best way to forcefully delete the Pod while minimizing data corruption risks?

A. Restart the **kubelet** to kill the pod immediately

B. Use the **kubectl delete** command with a **--grace-period=0** to force delete the pod

C. Wait for the pod's default termination grace period to end and allow Kubernetes to handle it

D. Delete the associated persistent volume manually and then delete the pod

Understanding Init Containers

What Are Init Containers?

These are specialized containers that run before the main application containers in a pod, ensuring that essential preconditions are met before the application containers start.

Key characteristics include:

- Init containers run one by one, and each must complete successfully before the next one starts.
- They can use different images and may contain different utilities or setups that are not needed in the main container.
- They are added under the **initContainers** field in the pod's spec, as an array, similar to the app containers field.
- If an init container fails, the **kubelet** keeps restarting it until it succeeds.
- The status of init containers can be viewed in the **.status.initContainerStatuses** field.

Init Containers: Use Cases

Database migration

Init containers are frequently used to execute database migrations before an application starts, ensuring compatibility between the application and the database schema.

Configuration management

An init container can generate or download configuration files needed by the main application container.

Dependency check

In distributed systems, init containers can ensure that necessary services (such as databases or message brokers) are running and available before the application container starts.

Init Containers: Use Cases

Security and permissions

Init containers can set the necessary file permissions or create directories needed by the main application container.

Network setup

In some cases, init containers can be used to configure networking components before the application starts.

Init vs. Regular Containers

Aspect	Init containers	Regular containers
Execution order	Run sequentially; each must finish before the next starts.	Run concurrently or sequentially, depending on the pod spec
Lifecycle	Run once to completion, then exit	Run for the entire lifetime of the pod
Resource handling	Handle different resource requests and limits	Use standard resource requests and limits throughout their runtime
Purpose	Perform initial setup tasks, such as setting up configurations and waiting for dependencies	Handle the main application workload of the pod
Restart policy	Retry restart on failure until completion (based on init container logic)	Perform restart based on the pod's **restartPolicy** setting

Advantages of Init Containers

Init containers can contain utilities or custom code for a setup that is not present in an app image.

1

The application image builder and deployer roles can work independently without the need to jointly build a single app image.

2

Advantages of Init Containers

Init containers offer a mechanism to block or delay app container startup until a set of preconditions are met.

3

Init containers can run with a different view of the filesystem than app containers in the same pod.

4

Init containers can securely run utilities or custom code that would otherwise make an app container image less secure.

5

Init Containers in Use

This example defines a simple pod that has two **init** containers:

```
apiVersion: v1
Kind:  pod
Metadata:
   name:  myapp-pod
   labels:
     app: myapp
   spec:
     containers:
     - name: myapp-container
       image: busybox:1.28
       command: [‘sh’,’-c’,’echo The app os running! 66 sleep3600
    initContainers:
       - name: init01
       image: busybox:1.28
       command: [‘sh’, ’-c’ , ”sleep 60; done”]
     - name: init02
       image: busybox:1.28
       command: [‘sh’, ’-c’, ”sleep 120; done”]
```

Init Containers in Use

The following are the commands used to start and check the status of the pod:

```
kubectl apply -f myapp.yaml

The output is similar to this:

pod/myapp-pod created

And check on its status with:

kubectl get -f myapp.yaml

The output is similar to this:

NAME         READY     STATUS     RESTARTS   AGE
myapp-pod    0/1       Init:0/2   0          6s

or for more details:

kubectl describe -f myapp.yaml
```

Assisted Practice

Configuring a Pod Using an Init Container **Duration: 10 Min.**

Problem statement:

You have been assigned a task to create and configure a pod using an init container to design more complex and flexible workflows for Kubernetes applications.

Outcome:

By completing this demo, you will be able to configure a Kubernetes pod using an Init container, deploy services, and verify the pod's state to understand the Init container's role in managing dependencies before the main container runs.

Note: Refer to the demo document for detailed steps:
03_Configuring_a_Pod_Using_an_Init_Container

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a pod
2. Create services
3. Verify the pod’s state

ASSISTED PRACTICE

Quick Check

You have a Pod that needs to perform a database migration before starting the main application. The migration requires creating several tables and validating connections. Which of the following best describes how you can ensure the migration completes before the main application container starts?

A. Use a **livenessProbe** in the main container to check for database availability

B. Set the **restartPolicy** of the main container to **OnFailure**

C. Define an init container that handles the migration tasks before the main container starts

D. Configure a sidecar container to perform the database migration

Managing Pod and Container Resources

Resource Management for Pods and Containers

In Kubernetes, while defining pod resources it is possible to define resources for individual containers. The most common resources to specify in a pod are CPU and memory (RAM).

Kubernetes components for resource management

01

kube-scheduler
Determines which node a pod should run on and handles the initial scheduling based on resource requests

02

kubelet
Manages pods and containers on a node and enforces resource limits while the pod is running

Request and Limit

These are key concepts for managing the resources allocated to containers within pods, allowing efficient resource distribution across the cluster while maintaining performance and stability.

Request

- Specify the minimum amount of CPU or memory that a container needs to operate
- Ensure that enough resources are allocated to containers to run efficiently
- Act as a lower threshold for resource allocation by not scheduling pods until the node does not have requested resources

Limit

- Specify the maximum amount of CPU or memory that a container is allowed to consume
- Prevent a container from over-consuming resources
- Terminate or limit container activities on exceeding the maximum resource usage threshold

Request and Limit

The following is an example of pod spec with request and limit:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  containers:
  - name: my-container
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Request and Limit

In the example:

- The request is for 64Mi of memory and 250m of CPU (250 millicores, or 25% of a single core).
- The limit is 128Mi of memory and 500m of CPU (50% of a core).

Behavior of Kubernetes:

- If the node does not have at least 64Mi of memory and 250m of CPU available, the pod will not be scheduled.
- If the container tries to use more than 500m of CPU, Kubernetes will throttle the usage. If it exceeds 128Mi of memory, it will be killed.

Resource Types

CPU and **memory** (RAM) are both resource types.

- CPU represents compute processing and is specified in units of Kubernetes CPUs.
- It is measured in units of cores.

- Memory refers to the amount of RAM a container can use on a node.
- It is measured in bytes (often MiB or GiB).

Resource Requests and Limits of Pod and Container

Each container of a pod can specify one or more of the following:

Resource Units in Kubernetes

CPU resource units

In Kubernetes, one CPU is equivalent to **one physical CPU core or one virtual core,** depending on whether the node is a physical host or a virtual machine running inside a physical machine.

Memory resource units

Memory can be expressed as a plain integer or as a fixed-point number using one of these suffixes:
Ei, Pi, Ti, Gi, Mi, Ki.

Resource Units in Kubernetes

```
    apiVersion: v1
    Kind:  pod
    Metadata:
       name:  Frontend
      spec:
        containers:
         -  name: app
            image: images.my company.example/app;V4
          resources:
                requests:
                  memory: “64Mi”
                  cpu: ‘250m”
                limits:
                  memory: “128Mi”
                  cpu: “500m”
         -  name: log-aggregator
            image: images.my company.example/log-
aggregator;v6
            resources:
                requests:
                  memory: “64Mi”
                  cpu: ‘250m”
                limits:
                  memory: “128Mi”
                  cpu: “500m”
```

- In this example, the pod has two containers.
- Each container has a request of 0.25 CPU and 64MiB (2^{26} bytes) of memory.
- Each container has a limit of 0.5 to 1 CPU and 128 MiB to 256 MiB of memory.

How Pods with Resource Limits Run

When the **kubelet** starts a container of a pod, it passes the CPU and memory limits to the container runtime.

Local Ephemeral Storage

It refers to the temporary disk storage a container uses on its node. This storage is non-persistent and is lost when the Pod is deleted or restarted.

Kubernetes supports two ways to configure local ephemeral storage on a node:

Single filesystem

In this configuration, place the different kinds of ephemeral local data (**emptyDir** volumes, writeable layers, container images, logs) in one filesystem.

Two filesystems

In this setup, two filesystems are used on the node. The first filesystem handles ephemeral data from running pods. The other is backed by another storage device, which is used exclusively for container images and writable layers.

Local Ephemeral Storage

The local ephemeral storage usage can be managed for containers within pods using the **ephemeral-storage** resource type

Each container of a Pod can specify either or both of the following:

spec.containers[ ].resources.requests.ephemeral-storage

This specification defines the minimum amount of ephemeral storage a container requires.

spec.containers[ ].resources.limits.ephemeral-storage

This defines the maximum amount of ephemeral storage a container can use, preventing over usage of node resources.

Local Ephemeral Storage

In the following example, the pod has two containers:

```
apiVersion: v1
 Kind:  pod
 Metadata:
    name:  Frontend
   spec:
     containers:
      -  name: app
         image: images.my company.example/app;V4
       resources:
             requests:
               ephemeral-storage: “2Gi”
             limits:
               ephemeral-storage: “4Gi”
               cpu: “500m”
      -  name: log-aggregator
         image: images.my company.example/log-aggregator;v6
         resources:
             requests:
               ephemeral-storage: “2Gi”
             limits:
               ephemeral-storage: “4Gi”
```

How Ephemeral Storage Pods Are Scheduled

In Kubernetes, the **kube-scheduler** schedules pods with ephemeral storage requests, ensuring the selected node has sufficient local ephemeral storage capacity for the Pod's needs.

The scheduler verifies whether the node meets the ephemeral storage requests of the pod.

The scheduler ensures that the total sum of ephemeral storage requests from all scheduled pods on the node is less than the node's available storage capacity.

If a Pod exceeds its allocated ephemeral storage limit while running, the **kubelet** on the node tracks the usage and may trigger eviction to free up resources.

In this way, the scheduler sums up the ephemeral storage requests and limits across all containers within a Pod. The Pod is scheduled only if a node can meet the total request.

Ephemeral Storage Consumption Management

If the **kubelet** is managing local ephemeral storage as a resource, it measures the use of storage in:

If a pod uses more ephemeral storage than allowed, it receives an eviction signal from **kubelet**.

How the Kubelet Measures Pod Storage Use

The **kubelet** supports different ways to measure pod storage use:

Periodic scanning

Filesystem project quota

How the Kubelet Measures Pod Storage Use

Periodic scanning

The **kubelet** performs regular scans on storage used by Pods, specifically in **emptyDir** volumes, container log directories, and writable container layers. These scans measure the total disk space consumed.

Limitation: If a file is deleted but remains open by a process, its node persists, but the kubelet won't recognize it as in-use space during scans. The kubelet doesn't track open file descriptors for such deleted files, potentially missing them in storage calculations.

How the Kubelet Measures Pod Storage Use

Filesystem project quota

Project quotas are an operating system feature used to track storage consumption on filesystems. When enabled, project quotas can monitor storage usage for specific directories, like those created by **emptyDir** volumes.

Limitation: It is not available in Kubernetes v1.30. In Kubernetes v1.31, project quotas allow monitoring of storage used on filesystems like XFS and ext4fs, but they do not enforce limits, only track usage.

Quick Check

As a DevOps engineer, you notice one of your containers handling temporary data exceeds its ephemeral storage limit and gets evicted. What action should you take to avoid this?

A. Set a lower CPU limit

B. Increase the ephemeral-storage limit

C. Use a sidecar container for logging

D. Adjust memory requests

Resource Quota

Overview of Resource Quota

Problem statement

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Solution

Resource quotas are a tool for administrators to address this concern.

Explanation

A **ResourceQuota** object sets limits on the number and amount of resources used per namespace, including object creation and compute resource usage.

Resource Quota

Resource quota works as follows:

- Teams use separate namespaces, managed via RBAC.
- Each namespace gets a **ResourceQuota** from the admin.
- Users create resources within namespaces, monitored by the quota system to stay within the **ResourceQuota** limits.
- Resource creation exceeding a quota results in a **403 FORBIDDEN** error with details of the breach.
- For CPU and memory, users must define requests or limits if quotas are active; non-compliance may prevent pod creation.

Enabling Resource Quota

This feature is typically enabled by default in many Kubernetes distributions.

The Kubernetes API server must have the **--enable-admission-plugins=** flag configured with **ResourceQuota** as one of its arguments.

Compute Resource Quota

The following are the supported resource types to limit the total sum of compute resources that can be requested in a given namespace:

Resource name	Description
limits.cpu	Across all pods in a non-terminal state; the sum of CPU limits cannot exceed this value.
limits.memory	Across all pods in a non-terminal state; the sum of memory limits cannot exceed this value.
requests.cpu	Across all pods in a non-terminal state; the sum of CPU requests cannot exceed this value.

Compute Resource Quota

Resource name	Description
requests.memory	Across all pods in a non-terminal state; the sum of memory requests cannot exceed this value.
hugepages-<size>	Across all pods in a non-terminal state; the number of huge page requests of the specified size cannot exceed this value.
cpu	Same as **requests.cpu**
memory	Same as **requests.cpu**

Compute Resource Quota

The following is the example for setting resource quota for compute resources:

```
apiVersion: v1
kind: ResourceQuota
metadata:
     name: compute-resources
     namespace: test
spec:
     hard:
         requests.cpu: "1"
         requests.memory: 1Gi
         limits.cpu: "2"
         limits.memory: 2Gi
         requests.nvidia.com/gpu: 4
```

Storage Resource Quota

The following are the supported resource types to limit the total sum of storage resources that can be requested in a given namespace:

Resource name	Description
requests.storage	Across all persistent volume claims; the sum of storage requests cannot exceed this value.
persistentvolumeclaims	The total number of **PersistentVolumeClaims** that can exist in the namespace
<storage-class-name>.storageclass.storage.k8s.io/requests.storage	Across all persistent volume claims associated with the **<storage-class-name>**; the sum of storage requests cannot exceed this value.
<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims	Across all persistent volume claims associated with the **<storage-class-name>**; the total number of persistent volume claims that can exist in the namespace

Object Count Quota

This allows administrators to limit the number of specific types of objects, like Pods, ConfigMaps, or PersistentVolumeClaims, within a Kubernetes namespace.

The following syntax can be used to set the quota for the total number of a particular resource kind in the Kubernetes API:

1. **count/<resource>.<group>** for resources from non-core groups

2. **count/<resource>** for resources from the core group

Object Count Quota

The following are examples of referencing various resource types for quota settings:

count/pods: Set a quota on the number of pods

count/deployments.apps: Set a quota on the number of deployments in the apps API group

count/services: Set a quota on the number of services

count/secrets: Set a quota on the number of secrets

Object Count Quota

The following are examples of referencing various resource types for quota settings:

count/configmaps: Set a quota on the number of ConfigMaps

count/jobs.batch: Set a quota on the number of jobs in the batch API group

count/cronjobs.batch: Set a quota on the number of cron jobs in the batch API group

Object Count Quota

The following example demonstrates the setting of resource quota for Kubernetes resources:

```
apiVersion: v1
kind: ResourceQuota
metadata:
     name: compute-resources
     namespace: test
spec:
     hard:
         configmaps: "10"
         persistentvolumeclaims: "4"
         pods: "4"
         replicationcontrollers: "20"
         secrets: "10"
         services: “4"
```

Assisted Practice

Managing Container Resources with a Resource Quota **Duration: 10 Min.**

Problem statement:
You have been assigned a task to create a namespace with a resource quota for managing resources within a Kubernetes container.

Outcome:

By completing this demo, you will be able to manage container resources in Kubernetes by creating a namespace with a resource quota, validating the quota by deploying pods, and observing the enforcement of resource limits.

Note: Refer to the demo document for detailed steps:
04_Managing_Container_Resources_with_a_Resource_Quota

Assisted Practice: Guidelines

Steps to be followed:

1. Create a namespace with a resource quota
2. Validate the resource quota by creating pods

ASSISTED PRACTICE

Static Pods

What Are Static Pods?

Static pods are managed directly by the kubelet daemon on a specific node, without the API server observing them. They are always bound to one kubelet on a specific node.

- The main use of static pods is to run a self-hosted control plane.
- The kubelet automatically tries to create a mirror pod on the Kubernetes API server for each static pod.
- The pods running on a node are visible on the API server but cannot be controlled from there.

Control Plane Pods

Pods that form the control plane, like the Kube-apiserver, Kube-scheduler, Kube-controller-manager, and etcd, operate as static pods overseen by the Kubelet service.

In the control-plane node, the control-plane pods manifest can be found in **/etc/kubernetes/manifests**

The **kubelet** on the control plane node uses this directory by running it with --pod-manifest-path=/etc/kubernetes/manifests/

Assisted Practice

Managing Static Pods **Duration: 10 Min.**

Problem statement:

You have been assigned a task to create, manage, and delete static pods in Kubernetes, highlighting its operational independence from the control plane.

Outcome:

By completing this demo, you will be able to create, manage, and delete static pods in Kubernetes, understanding their lifecycle and behavior when managed independently from the control plane.

Note: Refer to the demo document for detailed steps:
05_Managing_Static_Pods

Assisted Practice: Guidelines

Steps to be followed:

1. Create a static pod in the worker node
2. Try to delete the pod from the control plane
3. Delete the static pod in the worker node

ASSISTED PRACTICE

Deployment

What Is Deployment?

A Deployment provides declarative updates for pods and ReplicaSets.

Users set a target configuration for a Deployment, and the controller adjusts the current state to match it gradually.

Users can configure Deployments to generate new ReplicaSets or replace old ones, inheriting their resources.

Creating a Deployment

The following Kubernetes YAML manifest creates a Deployment resource named **nginx-deployment** that manages up to three replicas of a pod, each labeled with **app: nginx**:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
# Name of Deployment
  labels:
    app: nginx
spec:
  replicas: 3
# Number of pods
  selector:
    matchLabels:
      app: nginx
  template:
# pod Specifications
    << POD Template >>
```

Use Cases

Typical use cases for Deployments are:

1. Create a Deployment to rollout a ReplicaSet
2. Declare the new state of pods
3. Rollback to an earlier Deployment revision
4. Scale up the Deployment to facilitate more load
5. Pause the rollout of a Deployment
6. Use the status of the Deployment
7. Clean up older ReplicaSets

Deployment and ReplicaSets

- Deployment creates a ReplicaSet that creates three replicated pods.
- Deployment updates the pods by creating a new ReplicaSet and scaling it up to three replicas
- Deployment maintains the changes as Revision and for each revision, a ReplicaSet is maintained.

Purpose of ReplicaSet

A ReplicaSet's purpose is to maintain a stable set of replica pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical pods.

How a ReplicaSet Works

A ReplicaSet is defined with fields, including a selector that specifies how to identify pods.

A ReplicaSet is linked to its pods via the pods' **metadata.ownerReferences** field, which specifies by what resource the current object is owned.

A ReplicaSet identifies new pods to acquire by using its selector.

ReplicaSet in Use

A ReplicaSet ensures that a specified number of pod replicas are running at any given time.

Deployment provides declarative updates to pods along with a lot of other useful features.

Deployment is a higher-level concept that manages ReplicaSets.

Deployment is recommended over directly using ReplicaSets.

Using a Deployment

Example:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
 replicas: 3
 selector:
     matchLabers:
          tier: frontend
 template:
  metadata:
      labels:
          tier: frontend
   spec:
       container:
           name: php-redis
           image: grc.io/google_samples/gb-
frontened:v3
```

When to Use a ReplicaSet

To check for the pods brought up, use:

kubectl get pods

The pod information will appear as shown below:

Name	Ready	Status	Restarts	Age
frontend-b2zdv	1/1	Running	0	6m36s
frontend-vcmts	1/1	Running	0	6m36s
frontend-wtsmm	1/1	Running	0	6m36s

Why Use a ReplicaSet

It is used to ensure that a specified number of pod replicas are always running. It automatically creates and manages replicas of pods, ensuring high availability and fault tolerance.

The following command is used to retrieve detailed information about a specific pod in a Kubernetes cluster in YAML format:

kubectl get pods frontend-b2zdv -o yaml

Output:

```
apiVersion: app/v1
Kind: pod
Metadata:
 creationTimestamp: “2020-02-12T07:06:162”
  generateName: frontend
  labels:
    tier: frontend
name: frontend-b2zdv
Namespace: default
 ownerreferences:
 apiVersion: apps/v1
 blockOwnersDeletion: true
 controller: true
 kind: replicaSet
 name: frontend
 uid: f391f6db-bb9b-4c09-ae74-6a1f77f3d5cf
```

The output will be similar to the given image, with the frontend ReplicaSet's info set in the metadata's ownerReferences field.

Pod Template

Pod templates are specifications for creating pods and are included in workload resources such as Deployments, Jobs, and DaemonSets.

The **.spec.template** is a pod template that is also required to have labels in place.

For the template's Restart Policy field, the only allowed value for **.spec.template.spec.restartPolicy** is **Always** (default).

Labels and Selectors

The label selectors help the users to identify a set of objects. It the core grouping primitives in Kubernetes.

1. The **.spec.selector** field is a label selector. These are the labels used to identify potential pods to acquire.

2. In the ReplicaSet, **.spec.template.metadata.labels** must match **spec.selector**, or it will be rejected by the API.

Replicas

Specify the number of pods that should run concurrently by setting **.spec.replicas**. The ReplicaSet will create or delete its pods to match this number.

If **.spec.replicas** is not specified, it sets to 1 by default.

Working with Deployments

To delete a Deployment and all its pods, use **kubectl delete**. The Garbage Collector will automatically delete all the dependent pods by default.

When using the REST API or the **client-go** library, ensure that **propagationPolicy** is set to **Background** or **Foreground** in the -d option.

Working with Deployments

Isolating pods from a ReplicaSet

- pods can be removed from a ReplicaSet by changing their labels.
- This technique may be used to remove pods from Service for debugging, data recovery, and so on.
- pods that are removed in this way will be replaced automatically.

Scaling Deployments

Scale a Deployment by updating the .spec.replicas value. To scale down, the ReplicaSet controller uses an algorithm to determine which pods to remove first.

1. Pending (and unschedulable) pods are scaled down first.

2. If the **controller.kubernetes.io/pod-deletion-cost** annotation is set, the pod with the lower value will come first.

3. Pods on nodes with more Replicas come before pods on nodes with fewer replicas.

4. If the pods' creation times differ, the pod that was created more recently comes before the older pod.

Scaling Deployments

An application must be scaled when the load on the application grows or shrinks.

An example of this is shown below:

Example

```
// To decrease the number of NGINX replicas from 3 to 1 perform:
kubectl scale deployment/my-nginx --replicas=1

deployment.apps/my-nginx scaled

// Now, you only have one pod managed by the deployment.

kubectl get pod -l app=nginx

NAME                                READY      STATUS     RESTARTS    AGE
my-nginx-2035384211-j5fhi    1/1         Running    0             30m

// To have the system automatically choose the number of NGINX replicas as needed,
ranging from 1 to 3:

kubectl autoscale deployment/my-nginx --min=1 --max=3

horizontalpodautoscaler.autoscaling/my-nginx autoscaled
```

Updating Labels

Update labels on existing pods and other resources as needed before creating new ones with kubectl label. To label all NGINX pods, use the following command:

```
kubectl label pod -l app=nginx tier=fe

pod/my-nginx-2035384211-j5fhi labeled
pod/my-nginx-2035384211-u2c7e labeled
pod/my-nginx-2035384211-u3t6x labeled

// This filters all pods with the label "app=nginx" and then labels them
with "tier=fe". To see the pod you labeled, run:

kubectl get pod -l app=nginx -L tier

NAME                        READY     STATUS    RESTARTS   AGE       TIER
my-nginx-2035384211-j5fhi   1/1       Running   0          23m       fe
my-nginx-2035384211-u2c7e   1/1       Running   0          23m       fe
my-nginx-2035384211-u3t6x   1/1       Running   0          23m       fe
```

Updating Annotations

Annotations provide non-essential metadata accessible to API clients like tools and libraries, and can be added using kubectl annotate, as shown below:

```
kubectl annotate pod my-nginx-v4-9gw19 description='my frontend running
nginx'
kubectl get pod my-nginx-v4-9gw19 -o yaml

apiVersion: v1
kind: pod
metadata:
  annotations:
    description: my frontend running nginx
```

kubectl Apply

- kubectl apply pushes configuration changes to the cluster.
- It compares the version of the configuration that is being pushed with the previous version.
- It applies changes made without overwriting any automated changes to properties that have not been specified.

```
kubectl apply -f https://k8s.io/examples/application/nginx/nginx-
deployment.yaml
deployment.apps/my-nginx configured
```

kubectl Edit

Resources can be updated using kubectl edit.

An example of this is shown below:

Example

```
kubectl edit deployment/my-nginx

// This is similar to first getting the resource, editing it in the text
editor, and then applying the resource with the updated version:

kubectl get deployment my-nginx -o yaml > /tmp/nginx.yaml
vi /tmp/nginx.yaml
# do some edit, and then save the file

kubectl apply -f /tmp/nginx.yaml
deployment.apps/my-nginx configured

rm /tmp/nginx.yaml
```

Disruptive Updates

The **replace --force** option changes resource fields that cannot be updated once initialized. It deletes and re-creates the resource.

The process to modify the original configuration file is shown below:

```
kubectl replace -f https://k8s.io/examples/application/nginx/nginx-
deployment.yaml --force

deployment.apps/my-nginx deleted
deployment.apps/my-nginx replaced
```

Horizontal Pod Autoscaler (HPA)

A ReplicaSet can also be a target for **horizontal pod autoscalers (HPA)**. Thus, a Deployment can be auto-scaled by an HPA. Here is an example of HPA targeting the Deployment:

```yaml
apiVersion: autoscaling/v1
kind: horizontalpodAutoscaler
metadata:
    name: fronted-scalar
spec:
    scaleTargetRef:
    kind: Deployment
    name: frontend
    minReplicas: 3
    maxReplicas: 10
    targetCPUUtilizationpercentage: 50
```

Horizontal Pod Autoscaler (HPA)

Save this manifest into **hpa.yaml** and submit it to a Kubernetes cluster to create the defined HPA that autoscales the target Deployment, depending on the CPU usage of the replicated pods:

Alternatively, use the **kubectl autoscale** command to accomplish the same:

```
kubectl autoscale rs frontend --max=10 --min=3 --cpu-percent=50
```

Alternatives to ReplicaSet

The following is a list of alternatives to Kubernetes ReplicaSet:

1. Deployment
2. Bare pods
3. Job
4. DaemonSet
5. ReplicationController

Assisted Practice

Deploying Multitier Applications Using Kubernetes **Duration: 10 Min.**

Problem statement:
You have been assigned a task to deploy and verify a multitier WordPress and MySQL application on Kubernetes for managing, scaling, and maintaining them.

Outcome:

By completing this demo, you will be able to deploy a multitier application in Kubernetes by creating and exposing MySQL and WordPress deployments, managing services, and verifying successful deployment through web access.

Note: Refer to the demo document for detailed steps:
06_Deploying_Multitier_Applications_Using_Kubernetes

Assisted Practice: Guidelines

Steps to be followed:

1. Create a deployment for MySQL
2. Create a deployment for WordPress
3. Expose the service for WordPress and MySQL deployment
4. Verify the deployment of the application

ASSISTED PRACTICE

Assisted Practice

Deploying a Voting Application **Duration: 10 Min.**

Problem statement:
You have been assigned a task to deploy a voting application using Kubernetes pods.

Outcome:

By completing this demo, you will be able to deploy an application in Kubernetes by creating a namespace, deploying application resources, and verifying the application's deployment through Kubernetes commands and web access.

Note: Refer to the demo document for detailed steps:
07_Deploying_a_Voting_Application

Assisted Practice: Guidelines

Steps to be followed:

1. Create a namespace
2. Create an application for deployment
3. Verify the deployment of the application

ASSISTED PRACTICE

Quick Check

You have deployed a Kubernetes application using a Deployment, which ensures high availability by managing multiple replicas of your application. Midway through the application's lifecycle, you realize that the traffic load has increased significantly, and you need to increase the number of replicas to handle the additional load. What should you do to scale the application efficiently in Kubernetes?

A. Create additional pods manually to handle the traffic increase

B. Scale the number of pods by editing the Deployment configuration and increasing the replica count

C. Delete the current Deployment and create a new one with a higher number of replicas

D. Convert the Deployment into a DaemonSet to manage the increased load across all nodes

Perform Rolling Updates on Deployments

Deployment Strategies

Deployment strategies determine the method of replacing active pods in Kubernetes during configuration changes. The following are the two strategy types that are widely used:

1. **Recreate strategy** kills all the pods managed by the Deployment, prior to starting a new pod.

2. **Rolling update strategy** runs old as well as new configurations alongside each other.

Deployment Strategies

1 Transition of pods in RollingUpdate strategy

Performing a Rolling Update

To enable the rolling update feature of a Deployment, **.spec.strategy.type** must be set to RollingUpdate. This is shown in the YAML file given below:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
 replicas: 3
 strategy:
      rollingUpdate:
           maxSurge: 25%
           maxUnavailable: 1
      type: RollingUpdate
 selector:
     matchLabers:
          tier: frontend
 template:
  metadata:
      labels:
          tier: frontend
   spec:
       container:
           name: php-redis
           image: grc.io/google_samples/gb-frontened:v3
```

Deployment Strategies

2 Transition of pods in Recreate strategy

Performing a Recreate

To enable the rolling update feature of a Deployment, **.spec.strategy.type** must be set to Recreate.

This is shown in the YAML file given below:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
 replicas: 3
 strategy:
      type: Recreate
 selector:
     matchLabers:
          tier: frontend
 template:
  metadata:
      labels:
          tier: frontend
   spec:
       container:
           name: php-redis
           image: grc.io/google_samples/gb-frontened:v3
```

Rollout Commands

To get the rollout status of the latest Deployment rolling update, use **kubectl rollout status**.

Demo

```
kubectl rollout status deploy/frontend
```

Upon completion of rollout, the output will be as shown below:

Demo

```
deployment ”frontend" successfully rolled out
```

Canary Deployments

Multiple labels may be needed to differentiate Deployments of the same component with different releases or configurations, followed by a canary rollout of the new app version alongside the old one.

A track label distinguishes releases; for instance, **stable** denotes the main stable release.

```
name: frontend
     replicas: 3
     ...
     labels:
        app: guestbook
        tier: frontend
        track: stable
     ...
     image: gb-frontend:v3
// Then, create a new release of the guestbook frontend that carries the
track label with a different value so that two sets of pods do not overlap:
    name: frontend-canary
     replicas: 1
```

Canary Deployments

A new release of the guestbook frontend carrying the track label with a different value (canary) may be created to avoid overlap. This is shown below:

Demo

```
    labels:
       app: guestbook
       tier: frontend
       track: canary
    ...
    image: gb-frontend:v4
```

The front-end service spans both sets of replicas by selecting the common subset of their labels to ensure that the traffic is redirected to both applications.

Demo

```
selector:
    app: guestbook
    tier: frontend
```

Assisted Practice

Managing Kubernetes Deployments and Version Control **Duration: 10 Min.**

Problem statement:
You have been assigned a task to create a Kubernetes deployment, upgrade the image version, and revert to the previous version, demonstrating effective version control within a Kubernetes cluster.

Outcome:

By completing this demo, you will be able to manage Kubernetes deployments, upgrade container images, and use version control to seamlessly switch between deployment versions, ensuring stable and maintainable applications.

Note: Refer to the demo document for detailed steps:
08_Managing_Kubernetes_Deployments_and_Version_Control

Assisted Practice: Guidelines

Steps to be followed:

1. Create a Kubernetes deployment
2. Upgrade the image version
3. Switch back to the previous version

ASSISTED PRACTICE

Quick Check

You have just updated your application image in a Kubernetes Deployment and want to roll out the new version while ensuring there is no downtime. How can you perform a rolling update for this Deployment?

A. Delete the old Deployment and create a new one with the updated image

B. Manually scale down the old Deployment and scale up a new one with the updated image

C. Update the image in the Deployment spec and let Kubernetes handle the rolling update automatically

D. Create a new ReplicaSet with the updated image and delete the previous ReplicaSet once the new one is ready

Perform Rollbacks on Deployments

Performing a Rollback on a Deployment

Rolling back a Deployment is a three-step process:

Step 1: Find the Deployment revision, and list all the revisions of a Deployment

```
kubectl rollout history deployment <deployment-name>

This returns a list of Deployment revisions:
deployment "<deployment-name>"
REVISION        CHANGE-CAUSE
1               ...
2               ...
...
```

Performing a Rollback on a Deployment

The details of a specific revision are given below:

```
Kubectl rollout history deployment <deployment-name> --revision=1

// This returns the details of that revision:

deployment "<deployment-name>" with revision #1
pod Template:
Labels:       foo=bar
Containers:
app:
 Image:        ...
 Port:         ...
 Environment:  ...
 Mounts:       ...
 Volumes:      ...
```

Performing a Rollback on a Deployment

Step 2: Roll back to a specific revision

```
// Set the revision number you get from Step 1 in --to-revision

kubectl rollout undo deployment <deployment-name> --to-revision=<revision>

// If it succeeds, the command returns the following:

deployment "<deployment-name>" rolled back
```

Performing a Rollback on a Deployment

Step 3: Watch the progress of the DaemonSet rollback

```
// kubectl rollout undo deployment tells the server to start rolling back the
Deployment. The real rollback is done asynchronously inside the cluster control
plane.

kubectl rollout status deployment/<deployment-name>
```

After completion of rollback, the output will be as shown below:

```
deployment "<deployment-name>" successfully rolled out
```

Assisted Practice

Creating and Configuring the Metrics Server **Duration: 10 Min.**

Problem statement:
You have been assigned a task to create and configure the metrics server in the Kubernetes cluster to identify the top nodes, pods, and containers.

Outcome:

By completing this demo, you will be able to deploy and configure the Kubernetes metrics server, monitor node resource usage, and verify the metrics server functionality for analyzing cluster performance.

Note: Refer to the demo document for detailed steps:
09_Creating_and_Configuring_the_Metrics_Server

Assisted Practice: Guidelines

Steps to be followed:

1. Create a deployment
2. Configure the metrics server
3. Verify the metrics server deployment

ASSISTED PRACTICE

DaemonSet

DaemonSet: Introduction

A DaemonSet ensures that all the Nodes run a copy of a pod.

Writing a DaemonSet Spec

The daemonset.yaml file shown below describes a DaemonSet that runs the fluentd-elasticsearch Docker image:

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
```

Writing a DaemonSet Spec

```
# This toleration is to have the DaemonSet
runnable on master Nodes; remove it if your
masters can't run the pod
- key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image:
quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
```

```
volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath:
/var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Required Fields

As with all other Kubernetes configurations, a DaemonSet needs apiVersion, kind, and metadata fields. A DaemonSet also needs a .spec section.

Note

The DaemonSet object's name needs to be a valid DNS subdomain name.

Pod Template

The **.spec.template** is a pod template.

This template has the same schema as a pod. It is nested and does not have an apiVersion.

A pod template in a DaemonSet must set relevant labels as well as necessary fields.

The RestartPolicy must be unspecified or set to *Always*.

Pod Selector

The **.spec.selector** field is a pod selector. The **.spec.selector** object consists of two fields:

1. matchLabels
2. matchExpressions

The result is ANDed when both the fields are specified.

Running Pods on Select Nodes

If **.spec.template.spec.nodeselector** is set, the DaemonSet Controller will create pods on Nodes, matching the Node selector.

How Daemon Pods Are Scheduled

The DaemonSet Controller creates and schedules a DaemonSet.

It results in:

1. Inconsistent pod behavior
2. Loss of control over decisions when preemption is enabled with respect to scheduling

Running Pods on Select Nodes

The DaemonSet Controller schedules DaemonSets and binds the pod to the target host only when creating or modifying the DaemonSet pod.

Changes are not made to the spec.template of the DaemonSet.

```
nodeAffinity:
  necessaryDuringSchedulingIgnoredDuringExecution:
    nodeselectorTerms:
    - matchFields:
      - key: metadata.name
        operator: In
        values:
        - target-host-name
```

Using Labels

Labels allow one to slice and dice resources along any dimension specified by a label.

```
kubectl apply -f examples/guestbook/all-in-one/guestbook-all-in-one.yaml
kubectl get pod -Lapp -Ltier -Lrole

NAME                          READY     STATUS    RESTARTS   AGE       APP         TIER       ROLE
guestbook-fe-4nlpb            1/1       Running   0          1m        guestbook   frontend   <none>
guestbook-fe-ght6d            1/1       Running   0          1m        guestbook   frontend   <none>
guestbook-fe-jpy62            1/1       Running   0          1m        guestbook   frontend   <none>
guestbook-redis-master-5pg3b  1/1       Running   0          1m        guestbook   backend    master
guestbook-redis-slave-2q2yf   1/1       Running   0          1m        guestbook   backend    slave
guestbook-redis-slave-qgazl   1/1       Running   0          1m        guestbook   backend    slave
my-nginx-divi2                1/1       Running   0          29m       nginx       <none>     <none>
my-nginx-o0ef1                1/1       Running   0          29m       nginx       <none>     <none>

kubectl get pod -lapp=guestbook,role=slave

NAME                          READY     STATUS    RESTARTS   AGE
guestbook-redis-slave-2q2yf   1/1       Running   0          3m
guestbook-redis-slave-qgazl   1/1       Running   0          3m
```

Assisted Practice

Configuring a DaemonSet **Duration: 10 Min.**

Problem statement:
You have been assigned a task to configure a DaemonSet within Kubernetes for efficient application deployment and management.

Outcome:

By completing this demo, you will be able to configure a Kubernetes DaemonSet to ensure that a specific application is deployed and consistently runs across all nodes in the cluster, enhancing workload distribution and availability.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines

Steps to be followed:

1. Create and configure a DaemonSet

ASSISTED PRACTICE

Quick Check

You are tasked with creating a DaemonSet to deploy a logging agent on every node of your Kubernetes cluster. However, you want to ensure that this agent does not run on the master nodes. How would you achieve this in your DaemonSet spec?

A. Use a pod affinity rule to ensure the pods only run on worker nodes

B. Set a **replicas** field to limit the number of pods to the number of worker nodes

C. Add a node selector in the DaemonSet spec to exclude nodes labeled node-role.kubernetes.io/master

D. Use a Deployment with a node affinity rule to target the worker nodes

Perform Rolling Updates on a DaemonSet

DaemonSet Update Strategy

A DaemonSet has two types of update strategies:

OnDelete

- Users update a DaemonSet template.
- A new DaemonSet pod is created only after the old DaemonSet pod is deleted.

RollingUpdate

- Users update a DaemonSet template.
- The old DaemonSet pod is killed and a new DaemonSet pod is created automatically.

Performing a Rolling Update

To enable the rolling update feature of a DaemonSet, **.spec.updateStrategy.type** must be set to RollingUpdate. This is shown in the YAML file given below:

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
 spec:
      tolerations:
```

Performing a Rolling Update

```
# This toleration is to have the DaemonSet runnable on master nodes. Remove it if your masters can't
run the pod.
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Performing a Rolling Update

After verifying the update strategy of the DaemonSet manifest, create the DaemonSet.

Demo

```
kubectl create -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml

// Alternatively, use kubectl apply to create the same DaemonSet if you plan to update the
DaemonSet with kubectl apply.

kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml
```

To check the update strategy, use the following command:

Demo

```
// Check the update strategy of your DaemonSet, and make sure it's set to RollingUpdate.

kubectl get ds/fluentd-elasticsearch -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}' -n
kube-system
```

Performing a Rolling Update

Use the following command to check the DaemonSet manifest if the DaemonSet is not created in the system.

Demo

```
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset.yaml --dry-
run=client -o go-template='{{.spec.updateStrategy.type}}{{"\n"}}'
```

The output will be as shown below:

Demo

```
RollingUpdate

// If the output isn't RollingUpdate, go back and modify the DaemonSet object or
manifest.
```

Updating a DaemonSet Template

DaemonSet can be updated by applying a new YAML file as shown below:

Demo

```yaml
    apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd-elasticsearch
  namespace: kube-system
  labels:
    k8s-app: fluentd-logging
spec:
  selector:
    matchLabels:
      name: fluentd-elasticsearch
  updateStrategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  template:
    metadata:
      labels:
        name: fluentd-elasticsearch
    spec:
      tolerations:
      # This toleration is to have the DaemonSet runnable
on master Nodes. Remove it if your master Nodes can't run the pod.
- key: node-role.kubernetes.io/master
```

Demo

```yaml
effect: NoSchedule
      containers:
      - name: fluentd-elasticsearch
        image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        volumeMounts:
        - name: varlog
          mountPath: /var/log
        - name: varlibdockercontainers
          mountPath: /var/lib/docker/containers
          readOnly: true
      terminationGracePeriodSeconds: 30
      volumes:
      - name: varlog
        hostPath:
          path: /var/log
      - name: varlibdockercontainers
        hostPath:
          path: /var/lib/docker/containers
```

Commands

To update DaemonSets using configuration files, use **kubectl apply**

Demo

```
//Declarative command
kubectl apply -f https://k8s.io/examples/controllers/fluentd-daemonset- update.yaml
```

To update DaemonSets using imperative commands, use **kubectl edit**

Demo

```
//Imperative command
kubectl edit ds/fluentd-elasticsearch -n kube-system
```

To update a container image in the DaemonSet template, use **kubectl set image**

Demo

```
kubectl set image ds/fluentd-elasticsearch fluentd-
elasticsearch=quay.io/fluentd_elasticsearch/fluentd:v2.6.0 -n kube-system
```

Commands

To get the rollout status of the latest DaemonSet Rolling Update, use **kubectl rollout status**

```
kubectl rollout status ds/fluentd-elasticsearch -n kube-system
```

Upon completion of rollout, the output will be as shown below:

```
daemonset "fluentd-elasticsearch" successfully rolled out
```

Troubleshooting

When the rolling update is not completed due to nonavailability of resources, find the nodes that do not have DaemonSet pods scheduled.

This can be done by comparing the output of kubectl get Nodes, where user will get the following output:

Demo

```
kubectl get pod -l name=fluentd-elasticsearch -o wide -n kube-system
```

Troubleshooting

Broken rollout

- If a recent DaemonSet template update is broken, DaemonSet rollout will not progress.
- Example: If the container is crash looping or the container image does not exist (often due to a typo)

Clock skew

If .spec.minReadySeconds is specified in the DaemonSet, clock skew between the master and the Nodes will make the DaemonSet unable to detect the right rollout progress.

Clean up

To delete DaemonSet from a namespace, use the command shown below:

```
kubectl delete ds fluentd-elasticsearch -n kube-system
```

Quick Check

You want to ensure that a specific monitoring agent is running on every node in your Kubernetes cluster. However, this agent should only run on nodes labeled as monitoring-enabled=true. Which Kubernetes resource and approach would you use to achieve this?

A. Create a Deployment and set the replicas field equal to the number of nodes

B. Create a DaemonSet and use a node selector to target nodes with the label monitoring-enabled=true

C. Use a ReplicaSet with a pod affinity rule to ensure the agent runs on specific nodes

D. Create a StatefulSet and assign the agent to each node manually by using node names

Rollbacks

Performing a Rollback on a DaemonSet

The following demonstrates the three-step process of rolling back a DaemonSet:

Step 1: Find the DaemonSet revision, and list all the revisions of a DaemonSet

```
kubectl rollout history daemonset <daemonset-name>

This returns a list of DaemonSet revisions:
daemonsets "<daemonset-name>"
REVISION        CHANGE-CAUSE
1               ...
2               ...
...
```

Performing a Rollback on a DaemonSet

The details of a specific revision are given below:

Demo

```
Kubectl rollout history daemonset <daemonset-name> --revision=1

// This returns the details of that revision:

daemonset "<daemonset-name>" with revision #1
pod Template:
Labels:       foo=bar
Containers:
app:
 Image:        ...
 Port:         ...
 Environment:  ...
 Mounts:       ...
 Volumes:      ...
```

Performing a Rollback on a DaemonSet

Step 2: Roll back to a specific revision

```
// Set the revision number you get from Step 1 in --to-revision

kubectl rollout undo daemonset <daemonset-name> --to-revision=<revision>

// If it succeeds, the command returns the following:

daemonset "<daemonset-name>" rolled back
```

Performing a Rollback on a DaemonSet

Step 3: Watch the progress of the DaemonSet rollback

```
// kubectl rollout undo daemonset tells the server to start rolling back the
DaemonSet. The real rollback is done asynchronously inside the cluster control plane.
// To watch the progress of the rollback, input the command given below:

kubectl rollout status ds/<daemonset-name>
```

After completion of rollback, the output will be as shown below:

Demo

```
daemonset "<daemonset-name>" successfully rolled out
```

DaemonSet Revisions

To see what is stored in each revision, find the DaemonSet revision raw resources.

```
kubectl get controllerrevision -l <daemonset-selector-key>=<daemonset-
selector-value>

// This returns a list of controller revisions:

NAME                                CONTROLLER                    REVISION
AGE
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-
name>     1          1h
<daemonset-name>-<revision-hash>   DaemonSet/<daemonset-
name>     2          1h
```

Every controller revision stores annotations and a template of a DaemonSet revision.

Quick Check

You have updated a DaemonSet to deploy a new version of a monitoring agent across your nodes, but the new version is causing issues. You need to quickly roll back to the previous stable version. How would you perform a rollback on the DaemonSet?

A. Delete the current DaemonSet and manually create a new one using the previous image version

B. Use the kubectl rollout undo command to revert the DaemonSet to the previous version

C. Edit the DaemonSet spec manually to use the old image and reapply it

D. Use the kubectl scale command to scale down the DaemonSet and manually create pods with the old image

Application Configuration

Overview

An application's configuration is everything that is likely to vary between deploys (staging, production, developer environments, and so on). This includes:

1. Resource handles to the database, Memcached, and other backing services
2. Credentials to external services such as Amazon S3 or Twitter
3. Per-deploy values such as canonical hostname for the deploy

Twelve-Factor Principles

Kubernetes has evolved and is heavily influenced by twelve-factor principles:

Twelve-Factor Principles

Kubernetes has evolved and is heavily influenced by twelve-factor principles:

Configuring Applications in Kubernetes

To configure the apps in Kubernetes, use:

Environment variables for configuration

ConfigMap

Secret

Using Environment Variables for Configuration

Here is an example of how to use environment variables by specifying them directly within the pod specification:

```
apiVersion: v1
Kind: pod
Metadata:
    name: pod1
Spec:
    containers:
        name: nginx
        image: nginx
        env:
            name: ENVVAR1
            value: value 1
            name: ENVVAR2
            value: value2
```

Using Environment Variables for Configuration

Define two variables **ENVVAR1** and **ENVVAR2** in **spec.containers.env** with the values **value1** and **value2,** respectively.

Use the **kubectl apply** command to submit the pod Information to Kubernetes:

```
$ kubectl apply -f
https://raw.githubusercontent.com/abhirockzz/kubernetes-in-a-nutshell/master/configuration/kin-config-envvar-in-pod.yaml
pod/pod1 created
```

Use grep to filter the variable(s):

```
$ kubectl exec pod1 -it -- env | grep
ENVVAR
ENVVAR1=value1
ENVVAR2=value2
```

Quick Check

You are deploying an application in Kubernetes that requires database credentials. These credentials need to be securely passed to the application pods without hardcoding them in the container image. What is the most appropriate way to configure this in Kubernetes?

A. Store the credentials directly in the pod spec as environment variables

B. Hardcode the credentials in the container image before deployment

C. Use a Kubernetes Secret to store the credentials and reference it in the pod spec

D. Pass the credentials as command-line arguments when creating the pod

ConfigMap and Secrets

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs. pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

- It allows decoupling of environment-specific configuration from the container images so that the applications are easily portable.
- It is not designed to hold large chunks of data. The data stored in a ConfigMap cannot exceed 1 MiB.

ConfigMap Object

A ConfigMap is an API object that lets users store configurations for other objects to use. Unlike most Kubernetes objects that have a **spec**, a ConfigMap has **data** and **binaryData** fields.

The **data** field is designed to contain UTF-8 strings, while the **binaryData** field is designed to contain binary data as base64-encoded strings.

Each key under the **data** or the **binaryData** field must consist of **alphanumeric characters**, **hyphen** (**-**), **underscore** (**_**), or **period** (**.**).

The keys that are stored in the **data** must not overlap with the keys in the **binaryData** field.

ConfigMaps and Pods

```
apiVersion: v1
Kind: configmap
Metadata:
    name: game-demo
data:
    # property-like keys; each key map to a simple
value
    player_initial_lives: 3
    ui_properties_file_name: “user-
interface.properties”

    #file-like keys
    game.properties:
    enemy.types=aliens, monster
    players.maximum-lives=5
    user-interface.properties::
    color.good=purple1
    color.bad=yellow
    allow.textmode=true
```

- The users can create a pod specification that refers to a ConfigMap.
- The spec can configure the container(s) in that pod based on the ConfigMap data.
- The pod and the ConfigMap must be in the same namespace.
- The example shows a ConfigMap that has some keys with single values and other keys where the value looks like a fragment of a configuration format.

ConfigMaps and Pods

There are four different ways that a **ConfigMap** is used to configure a container inside a pod:

Define **command** and **args** inside a container

Use **Environment variables** for a container

Add a file in **read-only** volume for the application to read

Write **code** to run inside the pod that uses the Kubernetes API to read a ConfigMap

Using ConfigMaps

ConfigMaps can be mounted as data volumes. They may also be used by other parts of the system, without having a direct exposure to the pod.

The most common way to use ConfigMaps is to configure settings for containers running in a pod in the same namespace. A ConfigMap can also be used separately.

Using ConfigMaps

ConfigMaps allows you to inject configuration data into the pods as files. The following are the ways to configure pods to use a ConfigMap as a volume:

- Create a ConfigMap or use an existing one
- Modify the pod definition to add a volume under **.spec.volumes[]**
- Add a **.spec.containers[].volumeMounts[]** to each container that needs the ConfigMap
- Modify the image or command line so that the program looks for files in that directory

Using ConfigMaps

Each ConfigMap used needs to be referred to in **.spec.volumes**.

If there are multiple containers in the pod, each container needs its own **volumeMounts** block, but only one **.spec.volumes** is needed per ConfigMap.

Mounted ConfigMaps are updated automatically. When a ConfigMap currently consumed in a volume is updated, projected keys are updated as well.

The kubelet checks whether the mounted ConfigMap is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the ConfigMap.

Immutable ConfigMaps

This feature is controlled by the **ImmutableEphemeralVolumes** feature gate. An immutable ConfigMap can be created by setting the **immutable** field to **true**.

```
apiVersion: v1
Kind: ConfigMap
Metadata:
    ....
Data:
    ....
Immutable: true
```

Once a ConfigMap is marked as immutable, it is not possible to revert the change or mutate the contents of the **data** or the **binaryData** field.

Secrets

Kubernetes Secrets store and manage credentials like passwords, tokens, and SSH keys.

pods use Secrets by referencing them, and there are three ways to do this:

As files in a volume mounted on one or more of its containers

As container environment variables

When images are pulled for the pod by the kubelet

Types of Secrets

Kubernetes does not impose any constraints on the type name. However, if a built-in type is being used, all the requirements defined for that type must be met. The types are as follows:

- Opaque secrets
- SSH authentication secrets
- Service account token secrets
- TLS secrets
- Docker config secrets
- Bootstrap token secrets
- Basic authentication secret

Types of Secrets

Specify a Secret's type via the **type** field in the Secret resource or through kubectl flags, which affects its validations and Kubernetes-imposed constraints.

Built-in type	Usage
Opaque	arbitrary user-defined data
kubernetes.io/service-account-token	service account token
kubernetes.io/dockercfg	serialized **~/.dockercfg** file
kubernetes.io/dockerconfigjson	serialized **~/.docker/config.json** file
kubernetes.io/basic-auth	credentials for basic authentication
kubernetes.io/ssh-auth	credentials for SSH authentication
kubernetes.io/tls	data for a TLS client or server
bootstrap.kubernetes.io/token	bootstrap token data

Creating a Secret

There are several options to create a Secret:

1. Use **kubectl** command
2. Use config file
3. Use **kustomize**

Editing a Secret

Example

```
# please edit the object below. Lines beginning with a # will be ignored, and
an empty file will abort the edit. If an error occurs while saving this file will
be
#reopened with the relevant failure

apiVersion: v1
data:
    username: YWRtaW4=
    password: MWYyZDFlMmU2NRm
Kind: Secret
Metadata:
    annotations: game-demo
    kubectl.kubernetes.io/last-applied-configuration: {..}
    creationTimestamp: 2016-01-22T18:41:56Z
name: mysecret
    namespace: default
    resourceVersion: “164619”
    uid: cfee02d6-c137-11e5-8d73-42010af00002
Type: Opaque
```

An existing Secret may be edited with the following command:
kubectl edit secrets mysecret

This opens the default editor and allows to update the base64 encoded Secret values in the **data** field.

How to Use Secrets

Use Secrets as data volumes or env vars for pod containers; systems can also access Secrets indirectly.

To access Secret data within a pod:

- Use a pre-existing Secret or create a new one
- Change the pod definition to add a volume under **.spec.volumes[]**
- Specify a **.spec.containers[].volumeMounts[]** to each container that needs the Secret
- Alter the command line or image so that the program looks for files in the specific directory

Usage of Secrets as Environment Variables

To use a Secret in a pod's environment variable:

Create a Secret or use an existing one. Multiple pods can reference the same Secret

Modify the pod definition in each container. The environment variable that consumes the Secret key should populate the Secret's name and key in **env[].valueFrom.secretKeyRef**

Modify the image or command line so that the program looks for values in the specified environment variables

Using Secrets

An example of a pod that uses Secrets from environment variables:

```
apiVersion: v1
kind: pod
metadata:
 name: secret-env-pod
spec:
 containers:
  name: mycontainer
  image: redis
  env:
   name: SECRET-USERNAME
   valueFrom:
    secretKeyRef:
     name: mysecret
     key: username
    name: SECRET_PASSWORD
     valueFrom:
      secretKeyRef:
       name: mysecret
       key: password
restartPolicy: Never
```

Using Secrets

The Secret keys appear as normal environment variables inside a container that consumes a Secret in the environment variables. The following is the result of commands executed inside the container:

	Output:
echo $SECRET_USERNAME	**Admin**
echo $SECRET_PASSWORD	**1f2d1e2e67df**

Environment variables are not updated after a Secret update.

Immutable Secrets and ConfigMaps

The Kubernetes feature **Immutable Secrets and ConfigMaps** provides an option to set individual Secrets and ConfigMaps as immutable:

1. Protects from accidental (or unwanted) updates that could cause applications outages

2. Improves performance of the cluster by significantly reducing load on kube-apiserver

Assisted Practice

Configuring ConfigMaps **Duration: 10 Min.**

Problem statement:
You have been assigned a task to configure ConfigMaps to enhance the flexibility, security, and manageability of your applications, making them adaptable to different environments.

Outcome:

By completing this demo, you will be able to configure ConfigMaps in Kubernetes to manage environment variables and volumes for pods, enabling flexible and scalable deployment configurations.

Note: Refer to the demo document for detailed steps:
11_Configuring_ConfigMaps

Assisted Practice: Guidelines

Steps to be followed:

1. Add a ConfigMap entry to the pod
2. Create pods with services

ASSISTED PRACTICE

Assisted Practice

Creating Jobs **Duration: 10 Min.**

Problem statement:
You have been assigned a task to create jobs in Kubernetes for efficient task management within the cluster.

Outcome:

By completing this demo, you will be able to configure and create Kubernetes Jobs for automating and managing tasks effectively.

Note: Refer to the demo document for detailed steps:
12_Creating_Jobs

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Configure and set up the pod files

ASSISTED PRACTICE

Quick Check

You have an application running in Kubernetes that requires both non-sensitive configuration data (such as API endpoints) and sensitive information (such as API keys). How to provide these data to your application pods?

A. Store both the API endpoints and API keys in a ConfigMap

B. Use Secrets to store both the API endpoints and API keys for added security

C. Store the API endpoints in a ConfigMap and the API keys in a Secret

D. Hardcode the API endpoints and API keys in the application container image.

Health Monitoring

Overview

Node Problem Detector collects information about node problems from various daemons and reports these conditions to the API server as **NodeCondition** and **Event**.

Recommendations and Restrictions

When running the Node Problem Detector, expect an extra resource overhead on each node.

01 A resource limit is set for the Node Problem Detector.

02 The kernel log grows relatively slowly.

03 Even under high load, the resource usage is acceptable.

Quick Check

You have deployed a web application in Kubernetes and want to ensure that Kubernetes can detect when the application is unhealthy and automatically restart the pod if needed. Which approach should you use to monitor the health of your application?

A. Use a readiness probe to check if the pod is running correctly

B. Configure a liveness probe to regularly check the health of the application within the pod

C. Set a high resource limit so that the pod doesn't get terminated due to resource constraints

D. Monitor the application logs manually for errors and restart the erroneous pods

Achieving Scalability

Overview

When deploying a horizontally scalable application to Kubernetes, ensure the following configurations:

1. Resource limits and requests
2. Node and pod, affinities and anti-affinities
3. Health checks
4. Deployment strategies
5. pod disruption budgets
6. Horizontal pod autoscaling

Resource Requests and Limits

Resource requests and limits dictate how much memory and CPU the application may use.

It allows **autoscalers** to estimate the capacity and ensure that the cluster expands (and contracts) as the demand changes.

It is vital to set limits and requests on a **Deployment**.

If there are no limits, the Kubernetes **scheduler** cannot ensure that workloads are spread evenly across the nodes.

Affinities and Anti-affinities

Affinities and anti-affinities enable users to distribute or restrict workload based on a set of rules or constraints.

The **scheduler** can use affinities and anti-affinities to assign the best node.

There are two kinds of affinity or anti-affinity: **preferred** and **required**.

The pods can be placed according to specific requirements, with a combination of two types of affinity.

Scaling Applications

Kubernetes offers scalability by auto-adjusting workloads based on past resource use, with autoscaling spanning three dimensions:

Horizontal pod autoscaler (HPA)	Adjusts the number of replicas of an application
Cluster autoscaler	Adjusts the number of nodes of a cluster
Vertical pod putoscaler (VPA)	Adjusts the resource requests and limits of a container

Scaling Applications

The autoscalers operate on one of two Kubernetes layers:

Pod level

The **HPA** and **VPA** methods take place at the pod level. Both HPA and VPA scale the available resources or instances of the container.

Cluster level

The **cluster autoscaler** falls under the cluster level, where it scales up or down the number of nodes inside the cluster.

Health Checks

Kubernetes has two types of health checks:

Readiness probe	Notifies Kubernetes that the pod is ready to start receiving requests
Liveness probe	Notifies Kubernetes that the pod is still running as expected

Assisted Practice

Configuring Pods Using Liveness Probes **Duration: 10 Min.**

Problem statement:
You have been assigned a task to create and configure a pod using liveness probes to ensure the stability and reliability of the applications running inside the pod.

Outcome:

By completing this demo, you will be able to configure and use liveness probes in Kubernetes to monitor and automatically restart unresponsive pods, ensuring high availability and resilience of applications.

Note: Refer to the demo document for detailed steps:
13_Configuring_Pods_Using_Liveness_Probes

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a pod using liveness probes
2. Describe the pod

ASSISTED PRACTICE

Pod Disruption Budgets

Disruptions in Kubernetes clusters are the norm, not the exception.

The following are some of the possible reasons for disruptions with pods:

Removal of a spot instance on AWS that is out of service

.

Removal of underutilized nodes by autoscaler

Enforcement of new infrastructure configuration by the infrastructure team for replacing nodes within a cluster

Pod Disruption Budgets

The job of the pod disruption budget is to ensure that there is always a minimum number of pods ready for deployment.

Key points:

- With the pod disruption budget, it is possible to specify **minAvailable** or **maxUnavailable**.
- The values may be a percentage of the desired **replica count** or an **absolute number**.
- With pod disruption budget, Kubernetes can reject voluntary disruptions.

Horizontal Pod Autoscalers

Kubernetes control plane can scale applications based on their resource utilization. As pods become busier, the control plane brings up new replicas to share the load automatically.

Periodically, the controller monitors metrics provided by the metrics-server to scale the deployment up or down based on the pods' load.

It is possible to scale while configuring a Horizontal Pod Autoscaler based on CPU and memory usage. But custom metrics servers can help extend the metrics available.

Using HPA for Dynamic Workload Management

HPA works in a check, update, and check again style loop.

The loop works as follows:

1. Monitors the metrics server for resource usage
2. Calculates the desired number of replicas required based on the collected resource usage
3. Decides to scale up the application to the desired number of replicas
4. Changes the desired number of replicas

Since HPA is continuously monitoring, the process repeats from Step 1.

Assisted Practice

Configuring Horizontal Pod Autoscaling (HPA) **Duration: 10 Min.**

Problem statement:
You have been assigned a task to create and configure horizontal pod autoscaling to optimize performance and implement efficient resource utilization.

Outcome:

By completing this demo, you will be able to configure Horizontal Pod Autoscaling (HPA) in Kubernetes, enabling dynamic scaling of pod replicas based on CPU utilization, ensuring efficient resource management and application performance.

Note: Refer to the demo document for detailed steps:
14_Configuring_Horizontal_Pod_Autoscaling_(HPA)

Assisted Practice: Guidelines

Steps to be followed:

1. Create HPA in the master node
2. Check the deployment
3. Verify the HPA

ASSISTED PRACTICE

Quick Check

You have a web application running in a pod that occasionally stops responding. To automatically detect and restart the pod when it becomes unresponsive, you want to configure a liveness probe. Which method should you use to correctly configure the liveness probe?

A. Add a readiness probe that checks the pod's health periodically

B. Configure a liveness probe to send an HTTP GET request to a specific endpoint of the application

C. Manually monitor the pod logs and restart the pod when it stops responding

D. Use a startup probe to check if the pod is responding after deployment

Building Self-Healing Pods

Introduction: No Time for Downtime

An efficient application must run non-stop, 24/7, despite technical problems, updates, or natural events.

Kubernetes facilitates the smooth working of the application by abstracting machines physically. It is a container orchestration tool. The pods and containers in Kubernetes are self-healing.

Pod Readiness

An application can inject extra feedback or signals into podStatus: **pod readiness**.

To use this, set **readinessGates** in the pod's **spec** to specify a list of additional conditions that the **kubelet** evaluates for pod readiness.

Readiness gates are determined by the current state of **status.condition** fields for the pod.

Status for Pod Readiness

In case of a pod that uses custom conditions, it is evaluated to be ready only when both the statements given below apply:

All containers in the pod are ready.

All conditions specified in **readinessGates** are **true**.

The Three Container States: Waiting

Waiting: Created but not running

A container in the waiting stage can still run operations such as pulling images or applying Secrets.

To check the status of waiting, use:

```
kubectl describe pod [podname]
```

Along with the pod's state, a message and reason for the state are displayed as:

```
...
state: Waiting
reason: ErrImagePull
...
```

The Three Container States: Running

Running:

These are the containers running without issues.

The given command is run before the pod enters the running state:

postStart

Running pods display the time of starting of the container:

```
...
state: Running
started: Sat, 30 Jan 2021 16:48:38 +0530
...
```

The Three Container States: Terminated

Terminated:

It is a container that has failed or completed its execution.

The given command is executed before the pod is moved to be terminated:

preStop

Terminated pods will display the time of the entrance of the container:

```
...
State: Terminated
Reason: Completed
Exit Code: 0
Started: Mon, 11 Jan 2021 12:23:45 +0530
Finished: Mon, 18 Jan 2021 21:32:54 +0530
...
```

Self-Healing in Kubernetes

The pod phase in Kubernetes offers insights into the pod's placement. It can have:

- **Pending pods:** The container is waiting to be started.
- **Running pods:** The container has been created, and processes are running in it.
- **Succeeded pods:** The pods have completed the container lifecycle.
- **Failed pods:** The container either exited with non-zero status or was terminated by the system.
- **Unknown pods:** The state of the pod could not be obtained.

Container Probes

The kubelet can optionally perform and react to three kinds of probes on running containers:

livenessProbe	Indicates whether the container is running
readinessProbe	Indicates whether the container is ready to respond to requests
startupProbe	Indicates whether the application within the container is started

Self-Healing in Kubernetes

Each probe provides one of three results:

Failure	The container failed the diagnostic.
Unknown	The diagnostic failed, so no action can be taken.
Success	The container passed the diagnostic.

Self-Healing in Kubernetes

There are four distinct ways to inspect a container using a probe. Each probe must specify one of the following four mechanisms:

exec	Executes a specified command inside the container
gRPC	Performs a remote procedure call using gRPC
httpGet	Performs an HTTP request against the pod's IP address on a specified port and path
tcpSocket	Performs a TCP check against the pod's IP address on a specified port

Quick Check

You have deployed a pod in Kubernetes, and you want to ensure it automatically restarts if the application inside it crashes due to an unexpected error. Which restartPolicy should you set in the pod spec to achieve this self-healing behavior?

A. Never

B. OnFailure

C. Manual

D. Always

Manifest Management and Common Templating Tools

Overview of Manifest Management

There is a need to use templates in Kubernetes manifests for managing common parameters. These include:

Overview of Manifest Management

Manifest management involves deploying static YAML files, with no built-in templating.

Helm addresses this limitation as follows:

- **Helm** is the package manager for Kubernetes.
- It has templating capabilities.
- It is the Kubernetes equivalent of **yum** or **apt**.

- **Helm** deploys charts.
- It is a collection of all pre-configured, versioned application resources that may be deployed as a single unit.

Quick Check

You need to manage a large number of Kubernetes manifests across different environments (for example, development, staging, and production) while ensuring consistency and reusability. Which of the following tools would be most appropriate to simplify and manage manifest configurations with templating?

A. Use kubectl to manually create different manifest files for each environment

B. Use Helm to create templated manifests and manage configuration values for different environments

C. Use a text editor to manually update each manifest for different environments

D. Use kubeadm to automatically generate environment-specific manifests

Key Takeaways

- Node Isolation is used to ensure that only specific pods run on nodes with certain isolation, security, or regulatory properties.
- Topology spread constraints manage pod distribution across a cluster's failure domains like regions and nodes.
- A ReplicaSet ensures that a specified number of pod replicas are running at any given time.
- The Kubernetes control plane scales apps by resource use, auto-adding replicas for busier pods.

Installing MySQL and WordPress in a Kubernetes Environment

Duration: 30 minutes

Project agenda: To deploy a MySQL service and install a WordPress application, ensuring a robust, integrated web platform that leverages secure database management and dynamic content management capabilities

Description:
Your team lead has given you the task of deploying a MySQL and WordPress application with specific requirements. You need to add all users using ConfigMaps and incorporate all sensitive data using secrets. The service should be configured to run on NodePort. Additionally, ensure that the WordPress application only deploys once the MySQL service is verified to be up and running.

LESSON-END PROJECT

Installing MySQL and WordPress in a Kubernetes Environment

Duration: 30 minutes

Steps to perform:

1. Create a secret for storing the MySQL password securely
2. Create a MySQL YAML file
3. Deploy the WordPress application
4. Access the WordPress application using NodePort

Expected Deliverables: A WordPress Application running successfully

LESSON-END PROJECT

Thank You