Container Orchestration Using Kubernetes

Kubernetes Cluster

Learning Objectives

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

- Analyze the cluster architecture to gain insights into how interconnected nodes or servers collaborate
- Summarize the concept of nodes to understand the various operations performed on them
- Provide a detailed description of a controller to enhance understanding
- Explain the workings of kubeadm for simplifying the setup and management of Kubernetes clusters
- Classify the API server to better comprehend its role and significance within the system

Cluster Architecture Overview

Kubernetes Cluster

A Kubernetes cluster is a set of nodes for running containerized applications. It contains a control plane and one or more nodes.

Control Plane

A control plane is responsible for maintaining the desired state of the cluster. It manages all the applications running in the cluster and the container images associated with the applications.

Nodes are the components that run the applications and workloads associated with them.

A control plane is the master node in Kubernetes.

Components of Control Plane

The control plane has five important components:

1. kube-apiserver
2. etcd
3. kube-scheduler
4. kube-controller-manager
5. cloud-controller-manager

Nodes

Kubernetes places containers in pods to run the workload. These pods run on nodes. Nodes are physical or virtual machines that are part of the cluster.

A node has two components:

kubelet

kube-proxy

Container Runtime

Container runtime is the software responsible for running containers. Kubernetes supports many container runtimes including Docker, containerd, and CRI-O.

The two most common ways to add nodes to an API server are:

Kubelet registers itself with the control plane.

Users manually add a node object.

Container Runtime

Following the node registration, the control plane checks the validity of the new node object. A sample JSON manifest from which a node is being created is shown below:

Demo

```
{
 "kind":  "node",
 "apiVersion": "v1"
 "metadata": {
    "name":  "10.240.79.157"
    "label":    {
       "name":   "myfirst-k8s-node"
     }
 }
 }
```

An unhealthy node is ignored for any cluster activity.

Control Plane-Node Communication

Kubernetes has a **hub and spoke** API pattern. All the API usage from nodes (or pods) terminates at the API server.

There are two communication paths from the control plane to the node:

1. Apiserver to kubelet
2. Apiserver to nodes, pods, and services

Quick Check

A DevOps engineer needs to identify the primary data store used by Kubernetes to maintain the state and configuration of the cluster. Which of the following can be considered the primary data store of Kubernetes?

A. MySQL

B. PostgreSQL

C. etcd

D. Redis

Configuring a Cluster

General Configuration Tips

The latest stable API version must be specified while defining the configurations.

Store the configuration files in version control

Use YAML and not JSON to write configuration files

Group related objects in a single file

Call kubectl commands on a directory

Creating pods

Pods can be created using:

Deployment

Deployment is the most preferred way to create pods. It creates a ReplicaSet and specifies a strategy to replace pods.

Jobs

Jobs create pods that perform a specific job and exit.

Naked pods

Naked pods are pods created directly using a definition file. They are not bound to a ReplicaSet or deployment.

Guidelines for Creating a Service

Follow these guidelines to create a service:

1. Avoid using **hostNetwork** for the same reasons as **hostPort**
2. Use **headless** services for service discovery when kube-proxy load balancing is not needed
3. Create a service before its corresponding backend workloads
4. Add a DNS server as a cluster add-on to manage new services and create DNS records
5. Specify a hostPort for a pod only when it is necessary

Using Labels

Labels are key-value pairs that are attached to objects, such as pods.

Omit release-specific labels from their selector to make a service span multiple deployments

Define and use labels that identify semantic attributes {app: myapp, tier: frontend, phase: test, deployment: v3}

Select appropriate pods for the resources using labels

Container Images

When the kubelet attempts to pull a specified image, the **imagePullPolicy** and the tag of the image are affected. Here are some examples:

imagePullPolicy: **IfNotPresent:** The image gets pulled if it is not already present locally.

imagePullPolicy: **Always:** kubelet queries the container image registry to resolve the name to an image digest.

imagePullPolicy: **Never:** The image is never assumed to exist locally.

Using kubectl

The **kubectl apply** maintains the changes applied to a live object. **kubectl apply** follows declarative management.

To create single-container deployments and services, use **kubectl create deployment** and **kubectl expose**

For **get** and **delete** operations, use label selectors and not specific object names

Quick Check

A DevOps engineer is setting up a Kubernetes cluster for a microservices application. They need to configure the cluster, create pods, set up services, use labels, ensure correct container images, and manage the cluster with kubectl. What steps should the engineer follow to achieve this?

A. Configure the cluster and create pods using imperative commands and use labels, define services, and manage with kubectl

B. Use a single YAML file for all configurations and resources

C. Create pods first and then use kubectl for services and labels

D. Set up services before creating pods and manage with kubectl

Managing and Administering Clusters

Cluster Administration and Management

Cluster administration and management involves four steps:

Planning a cluster

The following are some considerations for Kubernetes cluster planning:

- Decide between local Kubernetes testing and high-availability multi-node clusters
- Choose between a hosted Kubernetes service like Google Kubernetes Engine or self-hosting a cluster
- Opt for on-premises or cloud-based clusters as Kubernetes doesn't natively support hybrid setups
- Decide if cluster management or active development of project code is the goal
- Familiarize oneself with the essential Kubernetes components necessary for running a cluster

Optional Cluster Services

The following are optional services for Kubernetes clusters:

- Integrating DNS
- Logging and monitoring cluster activity

Managing and Administering a Kubernetes Cluster

Duration: 20 minutes

Problem Statement:

You have been given a task to verify cluster certificates, create a namespace, and access clusters using the Kubernetes API to ensure secure communication, proper resource segmentation, and effective management within the cluster.

Outcome:

By the end of this demo, you will be able to manage your Kubernetes cluster nodes effectively by verifying certificates for secure communication, creating namespaces for resource organization, and using the Kubernetes API for resource management, ensuring optimal cluster performance, security, and scalability.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

Duration: 20 minutes

1. Verify the certificates of the cluster
2. Create a namespace
3. Access clusters using Kubernetes API

ASSISTED PRACTICE

Quick Check

A DevOps engineer is tasked with exposing a deployment within a Kubernetes cluster to allow external traffic to access the application. They decide to create a Kubernetes service to achieve this. Which command should the engineer use to create a Kubernetes service?

A. kubectl create deployment my-app --image=my-app:latest

B. kubectl expose deployment my-app --type=LoadBalancer --port=80 --target-port=8080

C. kubectl run my-app --image=my-app:latest --port=80

D. kubectl get svc

Node

Node

To run the workload, Kubernetes places the containers in pods and runs them on nodes. A node can be a physical or a virtual machine.

A cluster can have several nodes.

The control plane manages every node.

A node is made up of the kubelet, a container runtime, and the kube-proxy.

Adding a Node

There are two main ways to add nodes to the API server:

- Self-register
- Manually add the nodes

Node Name Uniqueness

The node name identifies a node and must, therefore, be unique. Two nodes cannot have the same name at the same time.

Kubernetes assumes that a resource with the same name is the same object.

An instance using the same name is assumed to have the same state.

Self-Registration of Nodes

When the kubelet flag **--register-node** is true (the default), the kubelet will attempt to register itself with the API server. To accommodate self-registration, use the following options to start the kubelet:

--kubeconfig

Set a path to credentials to authenticate itself to the API server

--register-with-taints

Register the node with the given list of taints

--cloud-provider

Talk to a cloud provider to read metadata about itself

--register-node

Register the node automatically with the API server

Self-Registration of Nodes

Following is the code to enable kubelet on a node for self-registration with the control plane:

Kubeadm join command

```
kubeadm join --token abcdef.1234567890abcdef --discovery-
token-unsafe-skip-ca-verification controlplane-IP:6443
```

Manual Node Administration

Node objects can be created and modified manually using the following steps:

1. Set kubelet **flag --register-node** to **false**
2. Use Kubectl to create and modify node objects
3. Note that the **--register-node** setting has no effect on modification of node objects
4. Use labels on nodes with node selectors on pods to control scheduling

Manual Node Administration

The following JSON manifest provides an example of creating a Kubernetes node with associated labels, which is useful for customizing the cluster's configuration.

Example

```
{
  "kind":  "node",
  "apiVersion": "v1"
  "metadata": {
     "name":  "10.240.79.157"
     "label":      {
        "name":   "myfirst-k8s-node"
      }
  }
  }
```

To create a node from a JSON manifest, use:
kubectl apply –f node.json

Node Status

A node's status contains the following information:

To view the status of a node, use:

```
kubectl describe node <insert-node-name-here>
```

Addresses

The output of the **kubectl describe node** can have three fields. Based on the cloud provider or the bare metal configuration, the usage of these fields will differ:

HostName

The hostname, as reported by the node's kernel, that can be overridden via the kubelet --hostname-override parameter

External IP

The IP address of the node that is externally routable

Internal IP

The IP address of the node that is routable only within the cluster

Conditions

The conditions field describes the status of all running nodes. Some examples of conditions are given below:

node Condition	Description
Ready	True if the node is healthy and ready to accept pods; False if the node is not healthy and not accepting pods; Unknown if there is no response during the last node-monitor-grace-period
DiskPressure	True if pressure exists on the disk size, that is, if the disk capacity is low; otherwise, false
MemoryPressure	True if pressure exists on the node memory, that is, if the node memory is low; otherwise, false
PIDPressure	True if pressure exists on the processes, that is, if there are too many processes on the node; otherwise, false stop
NetworkUnavailable	True if the network for the node is not correctly configured; otherwise, false

Conditions

A node condition is represented as a JSON object. The structure below describes a healthy node:

Example

```
"conditions": [
{
type": "Ready",
"status":  "True",
"reason": "KubeletReady",
"message": "kubelet is posting ready status",
"LastHeartbeatTime": "2019-06-05T18:38:352",
"LastTransitionTime" "2019-06-05T11:41:272"
}

]
```

Capacity and Allocatable Blocks

Capacity and allocatable blocks describe the resources available on the node. The resources include CPU, memory, and the maximum number of pods that can be scheduled on the node.

Capacity block specifies the total number of resources present in the node.

An **allocatable block** specifies the number of available resources on a node.

Info

Info provides general information about the node, such as kernel version, Kubernetes version, Docker version, and OS name.

kubelet is used to gather information about a node.

Heartbeats

They are used to determine the availability of a node. Heartbeats can be the updates of **nodestatus** or the Lease object.

The **nodeStatus** gets updated when there is a change in status or if there has been no update for a configured interval.

Lease objects are created and updated every 10 seconds.

Node Controller

Node controller is a Kubernetes control plane component that manages nodes.

Keeps the node controller's internal list of nodes up-to-date

Assigns a CIDR block to the node when it is registered

Monitors the node’s health

Rate limits on Eviction

The node eviction behavior changes when a node becomes unhealthy. Following are a few scenarios that trigger a change in the eviction behavior:

The eviction rate is reduced if the fraction of unhealthy nodes is at the least **--unhealthy-zone-threshold.**

The eviction rate is stopped if the cluster is small (≤ **--large-cluster-size-threshold** nodes). Otherwise, the eviction rate is reduced to **--secondary-node-eviction-rate.**

Node’s Resource Capacity

The node objects track information about the resource capacity of a node.

Self-registering nodes: The node capacity is reported when the node is registered.

Manually added nodes: The node capacity must be set when the node is added.

Graceful Node Shutdown

Kubernetes supports the Graceful node feature. This feature detects a node system shutdown and terminates pods running on the node.

The Graceful node shutdown feature depends on the system.

The Graceful node shutdown is controlled using the **GracefulnodeShutdown** feature gate.

Graceful Node Shutdown

Kubelet carries out the termination process in the two phases:

1. Terminates regular pods running on the node
2. Terminates critical pods running on the node

Understanding the Working of nodes

Duration: 20 minutes

Problem Statement:

You have been given a task to manage the nodes in a Kubernetes cluster, including verifying their status, deleting and registering worker nodes, and identifying node conditions and resources to ensure the cluster operates efficiently.

Outcome:

By the end of this demo, you will be able to manage the nodes in your Kubernetes cluster effectively. This includes verifying node status, deleting and registering worker nodes, and identifying node conditions and resource allocations, ensuring optimal cluster performance and scalability.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Duration: 20 minutes

Steps to be followed:

1. Verify the status of a node
2. Delete a worker node
3. Register a worker node using a config file
4. Identify the node conditions, capacity, and allocatable resources

ASSISTED PRACTICE

Launching a Pod and Establishing an Associated Service

Duration: 10 minutes

Problem Statement:

You have been asked to create and manage a deployment and its associated service in a Kubernetes cluster to ensure the application is deployed and accessible correctly.

Outcome:

By the end of this demo, you will be able to create and manage a deployment and service in your Kubernetes cluster, ensuring the application is deployed and accessible correctly through a defined deployment object and label selector.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Duration: 10 minutes

Steps to be followed:

1. Create a deployment object
2. Create a service with label selector for deployment

ASSISTED PRACTICE

Quick Check

A system administrator needs to expand a Kubernetes cluster by adding a new node. The node name must be unique and self-registered with accurate status reporting. Which steps should the administrator follow?

A. Manually assign a unique name to the node and update the kubelet configuration file to include the node's name and capacity

B. Use the kubectl command to add the node and rely on the Node Controller for status updates

C. Configure the kubelet to self-register the node with a unique name and use heartbeats for status updates

D. Assign a unique name to the node manually and use manual node administration to monitor node status

Control Plane Node Communication

Control Plane Node Communication

The control plane catalogs the communication paths between the control plane and the Kubernetes cluster. This helps users to customize the installation and harden network configuration.

Communication from Node to Control Plane

Kubernetes has a hub and spoke pattern, which ensures that all API usage from nodes terminates at the API server.

Step 01

The API server is configured to listen for remote connections on a secure HTTPS port with one or more forms of client authentication enabled.

Step 02

Nodes should be provisioned with a public root certificate for the cluster in such a way that they can connect securely to the API server with valid credentials.

Step 03

The components of the control plane communicate with the cluster API server over the secure port.

Control Plane to Node

The communication from control plane to nodes can follow two paths:

1

From the API server to the kubelet process

2

From the API server to any node, pod, or service

Connection from API Server to kubelet

The connections from the API server to the kubelet serve three purposes:

Attaching (through kubectl) to running pods

Fetching logs for pods

Providing kubelet's port-forwarding functionality

Connection from API Server to Nodes, Pods, and Services

- The API server can connect to a node, pod, or service. These connections that default to a plain HTTP connection are neither encrypted nor authenticated.
- The connections can run over a secure HTTPS. This is done by prefixing HTTPS: to the node, pod, or service name in the API URL.

Connections do not validate the certificate provided by the HTTPS endpoint. They do not provide client credentials.

Konnectivity Service

Konnectivity service provides a TCP-level proxy for the control plane to cluster communication.

It consists of two parts:

Konnectivity server

Konnectivity agents

Quick Check

A DevOps engineer is ensuring secure communication between various components in a Kubernetes cluster. They need to confirm the communication from the API server to the kubelet for managing nodes. Which service facilitates this connection?

A. NodePort service

B. kube-proxy

C. Konnectivity service

D. ClusterIP service

Understanding Controllers

Controller Pattern

Controllers are control loops that watch the state of the cluster and then make or request changes where needed.

Tracks at least one Kubernetes resource type

Represents the desired state of the objects

Moves the resource's current state come closer to that desired state

Control via API Server

Built-in controllers interact with the cluster API server and manage the system state.

The Job controller is an example of a Kubernetes built-in controller.

Controllers can update the objects that are used to configure them.

Direct Control

Direct control is employed when controllers need to make changes to things outside of the cluster.

Controllers find the desired state from the API server.

Controllers communicate directly to bring the current state closer in line.

Running Controllers

There are two ways of running controllers:

Inside the
kube-controller-manager

Outside the
kube-controller-manager

Quick Check

A Kubernetes administrator needs to implement a custom resource controller that can manage resources through the API server. This controller should efficiently handle the lifecycle of custom resources without direct intervention. Which approach should the administrator take to achieve this?

A. Implement direct control over resources using manual scripts

B. Use the Controller Pattern to build a custom controller that interacts with the API server

C. Manage resources by directly modifying etcd entries

D. Deploy a dedicated node to handle custom resource management

Importance of Cloud Controller Manager

Cloud Infrastructure

Cloud infrastructure technologies:

- Help run Kubernetes on public, private, and hybrid clouds
- Embed cloud-specific control logic to run on different cloud platforms
- Use a plugin mechanism that lets various cloud providers integrate their platforms with Kubernetes

Cloud Controller Manager Types

There are many types of controllers inside the Cloud Controller Manager, each of which has a role to play. The three important controllers are as follows:

Node Controller

Route Controller

Service Controller

Quick Check

A company is migrating its infrastructure to a hybrid cloud environment and needs to ensure seamless integration between its on-premises data center and various public cloud services. Which component of Kubernetes should the company utilize to embed cloud-specific control logic and ensure compatibility across different cloud platforms?

A. Kubelet

B. Kubernetes API server

C. Cloud Controller Manager

D. etcd

Working with Kubeadm

Kubeadm

The Kubeadm toolkit is used to bootstrap a best-practice Kubernetes Cluster. Kubeadm provides **kubeadm init** and **kubeadm join** to achieve this.

Kubeadm acts as an installer and a building block that helps to get a minimum viable cluster up and running.

Kubeadm Init Command Options

The **kubeadm init [flags]** command initializes and sets up a Kubernetes control plane node. The various options available for this command are:

1. --apiserver-advertise-address
2. --apiserver-bind-port
3. --apiserver-cert-extra-sans
4. --cert-dir
5. --certificate-key
6. --config
7. --control-plane-endpoint
8. --cri-socket

Kubeadm Init Command Options

9 --dry-run

10 --experimental-patches

11 --feature-gates

12 --h, --help

13 --ignore-preflight-errors

14 --image-repository

15 --node-name

16 --pod-network-cidr

Kubeadm Init Command Options

17 --service-cidr

18 --service-dns-domain

19 --skip-certificate-key-print

20 --skip-phases

21 --skip-token-print

22 --token

23 --token-ttl duration

24 --upload-certs

Init Command Workflow

To bootstrap a Kubernetes control plane, kubeadm init follows these steps:

Step 1

Run a series of preflight checks to validate the system state before making changes

Step 2

Set up identities for each cluster component by generating a self-signed CA

Step 3

Write kubeconfig files in /etc/kubernetes/ for the kubelet and controller-manager

Step 4

Generate static pod manifests for the API server, controller-manager, and scheduler

Init Command Workflow

Step 5

Apply labels and taints to the control plane node

Step 6

Generate the token that additional nodes can use to register themselves with a control plane

Step 7

Make necessary configurations for allowing node joining with the bootstrap tokens and TLS

Step 8

Install a DNS server and the kube-proxy addon components through the API server

Using Init Phases with kubeadm

To create a control plane in phases, use **kubeadm init phase** command:

```
sudo kubeadm init phase control-plane controller-manager --help
```

Certain control plane phases have unique flags. To get a list of available options, add **–help.** Here is an example:

```
sudo kubeadm init phase control-plane --help
```

Using Custom Images

By default, **kubeadm** pulls images from **registry.k8s.io**. If the requested Kubernetes version is a CI label, **gcr.io/k8s-staging-ci-images** is used.

A configuration file helps override this behavior. Kubernetes allows the following customization:

To provide **kubernetesVersion,** which affects the version of the images.

To provide an alternative **imageRepository** to be used instead of **registry.k8s.io**

To provide a specific **imageRepository** and **imageTag** for etcd or CoreDNS

Setting the Node Name

Based on the machine's host address, **kubeadm** assigns a node name. The **--node-name** flag may be used to override this setting.

The **--node-name** flag passes the appropriate **--hostname-override** value to the kubelet.

Running Kubeadm Without Internet Connection

To run kubeadm without an internet connection, the required control plane images must be pre-pulled.

The images can be listed and pulled using the **kubeadm config images** sub-command:

Demo

```
kubeadm config images list
kubeadm config images pull
```

Kubeadm Join

The **kubeadm join** command initializes and joins a Kubernetes worker node to the cluster. The **kubeadm join [api-server-endpoint] [flags]** command supports the following options:

1. --apiserver-advertise-address
2. --apiserver-bind-port
3. --certificate-key
4. --config
5. --control-plane
6. --cri-socket
7. --discovery-file
8. --discovery-token

Kubeadm Join

9 --discovery-token-ca-cert-hash

10 --experimental-patches

11 --discovery-token-unsafe-skip-ca-verification

12 --h, --help

13 --ignore-preflight-errors

14 --node-name

15 --skip-phases

16 --tls-bootsrap-token

17 --token

The Join Workflow

The command **kubeadm join** bootstraps a Kubernetes worker node or a control plane node and adds it to the cluster. This action is completed in three steps for worker nodes which are as follows:

Step 01

kubeadm downloads required cluster information from the API server.

Step 02

kubelet starts the TLS bootstrapping process.

Step 03

kubeadm configures the local kubelet to connect to the API server.

Using Join Phases with Kubeadm

Kubeadm allows users to join a node to the cluster in phases using **kubeadm join phase**.

Certain phases have unique flags. To get a list of available options, add **--help.**

An example is given below:

Example

```
kubeadm join phase kubelet-start --help
```

Working with Kubeadm

Duration: 10 minutes

Problem Statement:

You have been given a task to generate the necessary tokens for kubeadm to securely join nodes to the Kubernetes cluster, ensuring secure communication and proper integration of the nodes within the cluster environment.

Outcome:

By the end of this demo, you will be able to generate the necessary tokens for kubeadm, enabling the secure addition of nodes to your Kubernetes cluster. This will ensure that your nodes can securely join the cluster and participate in managing containerized applications.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Duration: 05 minutes

Steps to be followed:

1. Generate tokens for kubeadm

ASSISTED PRACTICE

Kubernetes Dashboard

The following diagram shows the Kubernetes Architecture:

Kubernetes Dashboard

Dashboard is a web-based Kubernetes user interface. It allows users to perform the given tasks in the Kubernetes cluster:

1. Deploy containerized applications
2. Troubleshoot deployed application
3. Manage the cluster resources
4. Get an overview of applications
5. Create or modify individual Kubernetes resources (such as deployments, Jobs, and DaemonSets)

Kubernetes Dashboard

Here is an overview of application pods:

Source: https://kubernetes.io/images/docs/ui-dashboard.png

Kubernetes Dashboard

View container logs of pod

Source: https://kubernetes.io/images/docs/ui-dashboard-logs-view.png

Quick Check

A system administrator is setting up a Kubernetes cluster without internet access. They need to initialize the control plane node and set the node name. What should they do?

A. Use kubeadm init --node-name and pre-pull the required images

B. Run kubeadm init with default options and set the node name later

C. Execute kubeadm init --offline and let kubeadm assign a default node name

D. Use kubeadm init followed by kubeadm join

Launching the Kubernetes Dashboard

Duration: 10 minutes

Problem Statement:

You have been given a task to set up a Kubernetes dashboard for managing and monitoring your Kubernetes cluster. This involves deploying the dashboard, validating the creation of necessary resources, and accessing the dashboard through the master node.

Outcome:

By the end of this demo, you will be able to set up and access the Kubernetes dashboard, validate resources, confirm the service type, and log in using the master node IP, enhancing your Kubernetes cluster management skills.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

Duration: 10 minutes

1. Implement the dashboard deployment
2. Validate the pod, service, and deployment creation
3. Confirm the dashboard service type
4. Access the master node IP
5. Log in to the service dashboard
6. Access the Kubernetes dashboard

ASSISTED PRACTICE

Managing a Cluster Using Kubelet

Managing Clusters

The kubelet is the primary node agent that runs on each node. It can register the node with the API server using a hostname, a flag to override the hostname, or specific logic for a cloud provider.

A container manifest may be provided to the kubelet using the following:

- podSpec
- File
- HTTP endpoint
- HTTP server

Kubelet Authentication

Requests to the kubelet's HTTPS endpoint that are not rejected by other configured authentication methods are treated as anonymous requests.

They will be given a username **system:anonymous** and a group **system:unauthenticated**.

To disable anonymous access and send **401 Unauthorized** responses to unauthenticated requests:

Start the kubelet with the **--anonymous-auth=false** flag

Kubelet Authentication

To enable **X509 client certificate authentication** to the kubelet's HTTPS endpoint:

- Start the kubelet with the **--client-ca-file** flag, providing a CA bundle to verify client certificates
- Start the apiserver with **--kubelet-client-certificate** and **--kubelet-client-key** flags

Kubelet Authentication

Here are a few steps to **enable API bearer tokens** to be used to authenticate to the kubelet's HTTPS Endpoint:

1. Ensure the **authentication.k8s.io/v1** API group is enabled in the API server
2. Start the kubelet with the **--authentication-token-webhook** and **–kubeconfig** flags
3. The kubelet calls the **TokenReview** API on the configured API server to determine user information from bearer tokens

Kubelet Authorization

Any request that is successfully authenticated is then authorized. The default authorization mode is **AlwaysAllow**, which allows all requests.

Access to the kubelet API can be subdivided. Subdivisions occur under the following conditions:

- Anonymous auth is enabled, but it limits the ability of anonymous users to call the kubelet API.
- Bearer token auth is enabled, but it limits the ability of arbitrary API users to call the kubelet API.
- Client certificate auth is enabled, but only some of the client certificates must be allowed to use the kubelet API.

Kubelet Authorization

To subdivide access to the kubelet API, delegate authorization to the API server:

Start the kubelet with the **--authorization-mode=Webhook** and the **--kubeconfig** flags

Ensure the **authorization.k8s.io/v1beta1** API group is enabled in the API server

The kubelet calls the **SubjectAccessReview** API on the configured API server to determine whether each request is authorized.

Kubelet Authorization

To authorize API requests using request attributes, kubelet adopts an approach like the API server. The verb is determined from the incoming request's HTTP web.

HTTP verb	Request verb
POST	create
GET.HEAD	get
PUT	update
PATCH	patch
DELETE	delete

Quick Check

A DevOps engineer is tasked with securing a Kubernetes cluster. They need to ensure that kubelet on each node is properly authenticated and authorized to manage resources within the cluster. Which steps should the engineer take?

A. Use HTTP for kubelet communication and manually verify each request

B. Configure kubelet with client certificate authentication and set up RBAC for authorization

C. Allow anonymous access to the kubelet and use API tokens for authentication

D. Disable kubelet authentication and rely on network firewalls for security

Role-Based Access Controller

What Is RBAC?

Role-based access controller (RBAC) is used to configure fine-grained and specific sets of permissions within a Kubernetes cluster.

Extensions or declarations can define roles and permissions.

Subject	Operations	Resources
User	List, get, create, update, delete, watch, patch, post, put	pods, nodes, configmaps, secrets, deployment
Group		
Service account		

Permissions are purely additive (there are no **deny** rules).

Operations and Subjects

Access control connects operations and subjects.

HTTP verbs sent to the API represent the operations on resources.

Subjects are the actors in the Kubernetes API and RBAC.

API Resource vs. Non-Resource Requests

The Kubernetes API server can add additional attributes as a resource or a non-resource request. Some of the attributes that can be added are:

Request Type	Attribute	Description
Authentication	User	The user string
	Group	The list of group names
	Extra	A map of arbitrary keys
API	Non-resources or API resources flag	

API Resource vs. Non-Resource Requests

Request Type	Attribute	Description
API resource request	API request verb	Lowercase resource verb
	Namespace	The namespace
	API group	The API group being accessed
	Resource	The resources ID
	Subresource	The subresources
Non-resource request	HTTP request verb	Lowercased HTTP method
	Request path	Non-resources request path
Verbs	Common API resource request	Get, list, watch, create, update, patch, delete, delete collection
	Special API resource request	Use, bind, escalate, impersonate, userextras

RBAC Objects

RBAC objects in Kubernetes enable control and management of actions within a cluster, offering granular control over permissions and access. RBAC API declares four kinds of Kubernetes object:

Object	Scope	Description
Role	Namespace	Grants verbs to resources within namespace
Rolebinding	Namespace	Maps role with user or group or service account
ClusterRole	Cluster or non-namespace	Grants verbs to resources of all namespaces in the cluster
ClusterRolebinding	Cluster or non-namespace	Maps cluster role with user or group or service account
Verbs (actions): List, get, create, update, delete, watch, patch, post, put Resources (API objects): pods, nodes, configmaps, secrets, deployment, jobs, etc.		

RBAC Objects

The following diagram explains the user facing RBAC objects and their relations:

Role

A Role always sets permissions within a particular namespace; when creating a Role, one must specify the namespace it belongs in.

The following Role **pod-reader** provides read permissions such as **get** and **list** to all pods in the **default** namespace.

Example

```
kubeadm join apiVersion: rbac.authorization.k8s.io/v1

kind: Role
metadata:
      namespace: default
      name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "list"]
 kubelet-start --help
```

RoleBinding

RoleBinding creation requires a specific subject to be bound to a Role within a namespace to define access permissions.

The following RoleBinding **read-pods** allows **jane** to read pods in the **default** namespace:

Example

```
apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding
metadata:
      name: read-pods
      namespace: default
subjects:
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
# "roleRef" specifies the binding to a Role / ClusterRole
    kind: Role #this must be Role or ClusterRole
    name: pod-reader # this must match the name of the Role or
ClusterRole you wish to bind
    apiGroup: rbac.authorization.k8s.io
```

ClusterRole

A ClusterRole defines cluster-wide access permissions and is not namespace-specific.

The following ClusterRole **secret-reader** allows get, watch, and list actions on resources **Secrets:**

Example

```
apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole
metadata: # "namespace" omitted since ClusterRoles are not namespaced
     name: secret-reader
rules:
- apiGroups: [""] # # at the HTTP level, the name of the resource for
accessing Secret
   # objects is "secrets"
   resources: ["secrets"]
   verbs: ["get", "watch", "list"]
```

Cluster RoleBinding

A ClusterRoleBinding can be used to grant permissions across a whole cluster.

The following ClusterRoleBinding **read-secret-global** allows any user in the group **manager** to read secrets in any namespace:

Example

```
apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding
metadata:
      name: read-secrets-global
subjects:
- kind: Group
  name: manager # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
   kind: ClusterRole
   name: secret-reader
   apiGroup: rbac.authorization.k8s.io
```

Authentication Methods for Kubernetes

There are several authentication mechanisms available in Kubernetes. Some of these include:

- X509 Client Certs
- Bearer Token
- Static Token File
- Bootstrap Tokens
- Service Account

Implementing RBAC Using Namespaces

Duration: 10 minutes

Problem Statement:

You have been given a task to configure role-based access control (RBAC) in a Kubernetes cluster. This includes creating a namespace, generating necessary cryptographic keys and certificates, defining roles and role bindings, setting up user credentials, and verifying the assigned roles.

Outcome:

By the end of this demo, you will be able to set up RBAC, create namespaces, define roles and bindings, and manage secure access control in a Kubernetes cluster.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Duration: 05 minutes

Steps to be followed:

1. Create a namespace in master node
2. Generate an RSA private key and certificate requests
3. Create a role
4. Create a rolebinding service
5. Setup credentials for the user
6. Verify the roles

ASSISTED PRACTICE

Quick Check

A security engineer is implementing Role-Based Access Control (RBAC) in a Kubernetes cluster to enhance security. The engineer needs to ensure that only authenticated users can perform specific actions on API resources. Which steps should the engineer follow to achieve this?

A. Set up network policies to control access and use service accounts for authentication

B. Configure RBAC to define roles and role bindings and use token-based authentication for users

C. Allow all users to access the cluster and rely on manual monitoring of their actions

D. Use basic authentication methods and create ClusterRoles with unrestricted access

API Server

Overview of API Server

The API server is the core of the Kubernetes control plane. It exposes an HTTP API that lets end users, clusters, and external components communicate with one another.

OpenAPI Specification

The Kubernetes API server uses the **/openapi/v2** endpoint to provide OpenAPI v2 spec. The response format can be requested using request headers, as shown below:

Header	Possible values	Notes
Accept-Encoding	gzip	Not supplying this header is also acceptable
Accept	application/com.github.proto-openapi.spec.v2@v1.0+protobuf	Mainly for intra-cluster use
	application/json	Default
	*	Serves application/json

API Groups and Versioning

Kubernetes supports multiple API versions that help to restructure resource representations and eliminate fields.

API group, resource type, namespace, and name distinguish the API resources.

API Resources

The command **kubectl api-resources** lists all supported API resources on the server, as seen in the following image:

```
➜  ~ kubectl api-resources
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED   KIND
bindings                                       v1                                     true         Binding
componentstatuses                 cs           v1                                     false        ComponentStatus
configmaps                        cm           v1                                     true         ConfigMap
endpoints                         ep           v1                                     true         Endpoints
events                            ev           v1                                     true         Event
limitranges                       limits       v1                                     true         LimitRange
namespaces                        ns           v1                                     false        Namespace
nodes                             no           v1                                     false        Node
persistentvolumeclaims            pvc          v1                                     true         PersistentVolumeClaim
persistentvolumes                 pv           v1                                     false        PersistentVolume
pods                              po           v1                                     true         Pod
podtemplates                                   v1                                     true         PodTemplate
replicationcontrollers            rc           v1                                     true         ReplicationController
resourcequotas                    quota        v1                                     true         ResourceQuota
secrets                                        v1                                     true         Secret
serviceaccounts                   sa           v1                                     true         ServiceAccount
services                          svc          v1                                     true         Service
mutatingwebhookconfigurations                  admissionregistration.k8s.io/v1        false        MutatingWebhookConfiguration
validatingwebhookconfigurations                admissionregistration.k8s.io/v1        false        ValidatingWebhookConfiguration
customresourcedefinitions         crd,crds     apiextensions.k8s.io/v1                false        CustomResourceDefinition
apiservices                                    apiregistration.k8s.io/v1              false        APIService
controllerrevisions                            apps/v1                                true         ControllerRevision
daemonsets                        ds           apps/v1                                true         DaemonSet
deployments                       deploy       apps/v1                                true         Deployment
replicasets                       rs           apps/v1                                true         ReplicaSet
statefulsets                      sts          apps/v1                                true         StatefulSet
tokenreviews                                   authentication.k8s.io/v1               false        TokenReview
localsubjectaccessreviews                      authorization.k8s.io/v1                true         LocalSubjectAccessReview
```

API Resources

The following table lists some more commands for viewing the API resources:

Request Detail	Command
Print the supported API resources with more information	kubectl api-resources -o wide
Print the supported namespaced resources	kubectl api-resources --namespaced=true
Print the supported non-namespaced resources	kubectl api-resources --namespaced=false
Print the supported API resources sorted by a column	kubectl api-resources --sort-by=name

API Changes

Kubernetes is designed to adapt to changes.

1

Supports the addition of new API resources and resource fields

2

Provides compatibility for official Kubernetes APIs

API Extension

The Kubernetes API can be extended in one of two ways:

1. Using custom resources
2. Implementing an aggregation layer

Quick Check

A Kubernetes administrator needs to add a custom resource definition (CRD) called Widget to the cluster. They must ensure it is properly versioned and accessible via the API server. What steps should they take?

A. Modify the API server code directly and deploy using kubectl apply

B. Create a new API group and version for the Widget CRD, define it, and update the OpenAPI spec using kubectl apply -f

C. Add the Widget CRD to an existing API group without versioning and update using kubectl create

D. Use a third-party tool to manage Widget CRDs without integrating into the API server

Achieving High Availability

Kubernetes Setup

A highly available Kubernetes cluster can be set up with kubeadm by applying two different approaches, namely, using **stacked control plane nodes** and an **external etcd cluster**.

Steps

1. Create a kube-apiserver load balancer; the name of the load balancer must resolve to DNS
2. Test the connection after adding the first control plane node to the load balancer
3. Add the remaining control plane nodes to the load balancer target group

Stacked Control Plane and etcd Nodes

Source: https://kubernetes.io/images/kubeadm/kubeadm-ha-topology-stacked-etcd.svg

Stacked Control Plane and etcd Nodes

Steps for the first control plane node:

Stacked Control Plane and etcd Nodes

To set the Kubernetes version, use **--kubernetes-version** flag. It is recommended that **kubeadm**, **kubelet**, **kubectl,** and **Kubernetes** versions should match.

The **--control-plane-endpoint** flag should be set to the address or DNS and port of the load balancer.

The command to initialize the control plane is given below:

```
sudo kubeadm init --control-plane-endpoint "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT" --
upload-certs
```

Stacked Control Plane and etcd Nodes

To upload the certificates to be shared across all the control plane instances, the **--upload-certs** flag is used.

Use the following command to re-upload the certificates and generate a new decryption key on a control plane node that is joined to the cluster:

```
sudo kubeadm init phase upload-certs --upload-certs
```

Stacked Control Plane and etcd Nodes

To apply the CNI, the CNI provider configuration must correspond to the pod CIDR specified in the kubeadm configuration file. For instance, to apply Weave Net CNI, use the following command:

```
kubectl apply -f "https://cloud.weave.works/k8s/net?k8s-version=$(kubectl version
| base64 | tr -d '\n’)”
```

To watch the pods of the control plane components, use the following command:

```
kubectl get pod -n kube-system -w
```

Setting Up Additional Control Plane Nodes

Execute the join command that is generated by kubeadm when the init command on the first node is run. The command is shown below:

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-token-
ca-cert-hash sha256:7c2e69131a36ae2a042a339b33381c6d0d43887e2de83720eff5359e26aec866 --
control-plane --certificate-key
f8902e114ef118304e561c3ecd4d0b543adc226b7a07f675f56564185ffe0c07
```

External etcd Nodes

Source: https://kubernetes.io/images/kubeadm/kubeadm-ha-topology-external-etcd.svg

External etcd Nodes

Setting up a cluster with external etcd nodes differs from setting up with stacked etcd in one aspect.

Steps

1. Set up etcd and pass the etcd information in the kubeadm config file
2. Set up the first control plane node
3. Set up additional control plane nodes as required

Setting up etcd Cluster

To set up the etcd cluster, set up SSH and copy the files given below from any etcd node in the cluster to the first control plane node:

Example

```
export CONTROL_PLANE="ubuntu@10.0.0.7"

scp /etc/kubernetes/pki/etcd/ca.crt "${CONTROL_PLANE}";

scp /etc/kubernetes/pki/apiserver-etcd.client.crt "${CONTROL_PLANE}";

scp /etc/kubernetes/pki/apiserver-etcd.client.key "${CONTROL_PLANE}";
```

Then, the value of **CONTROL_PLANE** must be replaced with **user@host** of the first control plane node.

Setting up First Control Plane Node

kubeadm-config.yaml with the content, as shown below, must be created:

Example

```
apiversion: kubeadm.k8s.io/v1beta2
Kind: clusterConfiguration
kuberenetesVersion: stable
controlPlaneEndpoint: "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT"
etcd:
    external:
    endpoint:
    https://etcd_0_IP:2379
     https://etcd_1_IP:2379
     https://etcd_2_IP:2379
    caFile: /etc/kubernetes/pki/etcd/ca.crt
    certfilr: /etc/kubernetes/pki/apiserver-etcd-client.crt
    keyfile: /etc/kubernetes/pki/apiserver-etcd-client.key
```

Setting up First Control Plane Node

The following variables in the config template must be replaced with the appropriate values for the cluster:

Example

```
LOAD_BALANCER_DNS
LOAD_BALANCER_PORT
etcd_0_IP
etcd_1_IP
etcd_2_IP
```

Setting up First Control Plane Node

Once the **.yaml** file is created, there are three steps to be followed:

1. On the node, run **sudo kubeadm init --config kubeadm-config.yaml --upload-certs**
2. Write the output **join** commands that are returned to a text file
3. Apply the CNI plugin

Common Tasks After Bootstrapping Control Plane

Install workers

The command generated as the output of **kubeadm init** command must be used to install worker nodes to the cluster. This can be done as shown below:

```
sudo kubeadm join 192.168.0.200:6443 --token 9vr73a.a8uxyaju799qwdjv --discovery-
token-ca-cert-hash
sha256:7c2e69131a36ae2a042a339b33381c6d0d43887e2de83720eff5359e26aec866
```

Common Tasks After Bootstrapping Control Plane

Manual Certificate Distribution

Certificates must be manually copied from the primary control plane node to the joining control plane nodes if the **–upload-certs** flag is not used during initialization.

Manual Certificate Distribution

To distribute certificates using **ssh** and **scp:**

1. Enable ssh-agent on the main device that has access to all other nodes in the system
2. Add the SSH identity to the session
3. Check that the SSH connection between nodes is working correctly
4. Run the script to copy the certificates from the first control plane node to the others
5. Run the script on each control plane node to move the copied certificates from home directory to **/etc/kubernetes/pki** directory

Quick Check

A system administrator needs to set up a highly available Kubernetes cluster with multiple control plane nodes and an external etcd cluster. What steps should they follow?

A. Set up a single control plane node and rely on it for all operations

B. Configure stacked control plane and etcd nodes on each control plane node

C. Set up the first control plane node, configure external etcd nodes, and manually distribute certificates to additional control plane nodes

D. Use only external etcd nodes without additional control plane nodes

Backup and Restoration of etcd Cluster Data

Operating etcd Clusters for Kubernetes

The etcd is a highly-available key-value store that is used as a backing store for all cluster data in Kubernetes.

To have a Kubernetes cluster that uses etcd as its backing store, a backup plan for the data is mandatory.

Note

Kubectl command-line tool must be configured to communicate with the cluster.

Prerequisites for etcd

1. Etcd needs to be run as a cluster of odd members.
2. A distributed etcd system keeps the cluster stable.
3. In an unstable etcd system, a cluster cannot make any changes to its current state.
4. Keeping etcd clusters stable is critical to the stability of Kubernetes clusters.
5. To run an etcd in production, the minimum recommended version is **3.2.10+.**

Starting a Multi-Node etcd Cluster

The durability and availability of etcd significantly increase when it is run as a multi-node cluster.

To start a multi-node etcd cluster, run the following command:

```
etcd --listen-client-
urls=http://$IP1:2379,http://$IP2:2379,http://$IP3:2379,http://$IP4:2379,http://$I
P5:2379 --advertise-client-
urls=http://$IP1:2379,http://$IP2:2379,http://$IP3:2379,http://$IP4:2379,http://$I
P5:2379
```

Then, start the Kubernetes API server with the flag **--etcd-servers=$IP1:2379,$IP2:2379,$IP3:2379,$IP4:2379,$IP5:2379**.

Note

IP<n> variables must be set to the client IP addresses.

Multi-Node etcd Cluster with Load Balancer

To run a load balancing etcd cluster, follow the given three steps:

An etcd cluster must be set up.

A load balancer must be configured.

The Kubernetes API servers must be started with the flag **--etcd-servers** set to **$LB:2379**.

Securing etcd Clusters

To secure etcd clusters, a firewall may be set up or etcd-provided security features may be used. Securing etcd clusters involves:

Securing Communication

Configuring etcd with secure peer communication

Set flags **--peer-key-file=peer.key** and **--peer-cert-file=peer.cert** respectively and use **HTTPS** as the **URL** schema

Configuring etcd with secure client communication

Set flags **--key-file= k8sclient.key** and **--cert-file= k8sclient.cert** respectively and use **HTTPS** as the **URL** schema

Limiting Access of etcd Clusters

The access to etcd cluster must be restricted to the Kubernetes API servers through TLS authentication.

Example

- Consider a key pair **k8sclient.key** and **k8sclient.cert** trusted by the CA **etcd.ca.**
- Etcd configured with **--client-cert-auth** and TLS uses system CAs or the CA passed by **--trusted-ca-file** flag to verify the client certificates.
- Setting **--client-cert-auth** and **--trusted-ca-file** to **true** and **etcd.ca** respectively restricts the access to clients **k8sclient.cert.**

To provide access to Kubernetes API servers, set flags **--etcd-certfile,--etcd-keyfile,** and **--etcd-cafile** to **k8sclient.cert, k8sclient.key,** and **ca.cert,** respectively.

Replacement of a Failed etcd Member

Failed members of a cluster must be replaced to improve the overall health of the cluster. Replacement of a failed member comprises two steps:

Replacement of a Failed etcd Member

Example: Replacing a failed member in a three-member etcd cluster member1=http://10.0.0.1,member2=http://10.0.0.2, and member3=http://10.0.0.3

Scenario: member1 fails, and member4=http://10.0.0.4 gets added as a replacement.

1. Get the member ID of the failed member
2. Remove the failed member
3. Add the new member
4. Start the newly added member on a machine with the IP 10.0.0.4
5. Update the **--etcd-servers** flag and load balancer configuration

Backing up an etcd Cluster

An etcd cluster can be backed up either by using a built-in snapshot or a volume snapshot.

etcd built-in snapshot

Snapshot supported by etcd can be taken from a live member or by copying the **member/snap/db** file from the etcd data directory

Volume snapshot

Snapshots provided by storage volumes on which etcd is running

To create a snapshot of the etcd cluster's data, use the command;

ETCDCTL_API=3 etcdctl --endpoints=https://127.0.0.1:2379 \--cacert=<trusted-ca-file> --cert=<cert-file> --key=<key-file> \snapshot save <backup-file-location>

Scaling up etcd Clusters

Scaling up etcd clusters increases cluster availability.

Scaling does not increase cluster performance or capability.

Restoring an etcd Cluster

Snapshots taken from an etcd process (major and minor version) restore an etcd cluster. Kubernetes supports restoring a version from a different patch version of etcd.

To recover data from a failed cluster, a restore operation must be employed.

To restore an etcd snapshot, use the command;

ETCDCTL_API=3 etcdctl --data-dir <data-dir-location> snapshot restore snapshot.db

Backing Up and Restoring etcd Cluster Data

Duration: 10 minutes

Problem Statement:

You have been given a task to implement a backup and recovery process for a Kubernetes etcd cluster, which involves backing up the etcd cluster data and retrieving the data for restoration.

Outcome:

By the end of this demo, you will be able to back up and recover an etcd cluster in Kubernetes, ensuring data integrity and availability for effective management and disaster recovery.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

Duration: 10 minutes

1. Back up the etcd cluster data
2. Obtain the data from the etcd cluster

ASSISTED PRACTICE

Quick Check

A Kubernetes administrator needs to back up the etcd cluster data to prevent data loss. Which command should they use to create a snapshot of the etcd data?

A. etcdctl save snapshot

B. etcdctl backup –all

C. etcdctl snapshot save <backup-file>

D. etcd backup create <backup-file>

Version Upgrade on Kubernetes Cluster

Version Upgrade on Kubernetes Cluster

At a high level, the upgrade workflow of a Kubernetes cluster comprises the following:

- Primary control plane node upgrade
- Additional control plane nodes upgrade
- Worker nodes upgrade

Determine Version to Upgrade

Enter the commands shown below to find the latest stable 1.28 version using the OS package manager:

Example

```
apt update
apt-cache madison kubeadm
# find the latest 1.28 version in the list
# it should look like 1.28.x--00, where x is the latest
patch
```

Upgrading Control Plane Nodes

The control plane nodes must be updated one at a time. The node selected for upgrade must have the **/etc/kubernetes/admin.conf** file.

Steps

1. Call **kubeadm upgrade**
2. Drain the node
3. Upgrade **kubelet** and **kubectl**
4. Uncordon the node

Upgrading Control Plane Nodes

Upgrade kubeadm for the first control plane node, as shown below:

Example

```
# replace x in 1.28.x--00 with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.28.x--00 && \
apt-mark hold kubeadm
-
# the following method can also be used.
apt-get update && \
apt-get install -y --allow-change-held-packages
kubeadm=1.28.x--00
```

Upgrading Control Plane Nodes

Verify the working of kubeadm and check its version using the commands shown below:

Example

```
#Verify that the download works and has the expected
version:

kubeadm version

#Verify the upgrade plan:

kubeadm upgrade plan
```

Upgrading Control Plane Nodes

Use this command for other control plane nodes:

Example

```
#For other control plane nodes, use:
sudo kubeadm upgrade node
#instead of:
sudo kubeadm upgrade apply
```

Upgrading Control Plane Nodes

To prepare the node for maintenance, mark the unschedulable and evict workloads:

Example

```
# replace <node-to-drain> with the name of your node you are
draining
kubectl drain <node-to-drain> --ignore-daemonsets
```

To upgrade kubelet and kubectl:

Example

```
  # replace x in 1.28.x--00 with the latest patch version
   apt-mark unhold kubelet kubectl && \
   apt-get update && apt-get install -y kubelet=1.28.x--00
kubectl=1.28.x--00 && \
   apt-mark hold kubelet kubectl
   -
   # the following method can also be used
   apt-get update && \
   apt-get install -y --allow-change-held-packages kubelet=1.28.x-
00 kubectl=1.28.x-00
```

Upgrading Control Plane Nodes

To restart the kubelet, use the command:

Example

```
Restart the kubelet:

sudo systemctl daemon-reload
sudo systemctl restart kubelet
```

To uncordon the node, enter the command:

Example

```
# replace <node-to-drain> with the name of your node
kubectl uncordon <node-to-drain>
```

Upgrade Worker Nodes

The upgrade procedure on worker nodes involves the following steps:

Upgrade Worker Nodes

To upgrade kubeadm, the following commands can be used:

Example

```
# replace x in 1.28.x--00 with the latest patch version
apt-mark unhold kubeadm && \
apt-get update && apt-get install -y kubeadm=1.28.x--00 &&
\
apt-mark hold kubeadm
-
# you can also use the following method
apt-get update && \
apt-get install -y --allow-change-held-packages
kubeadm=1.28.x--00
```

Upgrade Worker Nodes

Worker nodes can be upgraded and drained using the following commands:

Example

```
#For worker nodes this upgrades the local kubelet
configuration:
sudo kubeadm upgrade node


#Drain the node
# replace <node-to-drain> with the name of your node you
are draining
kubectl drain <node-to-drain> --ignore-daemonsets
```

Upgrade Worker Nodes

The upgrade procedure for kubelet and kubectl can be done as follows:

Example

```
# replace x in 1.28.x--00 with the latest patch version
apt-mark unhold kubelet kubectl && \
apt-get update && apt-get install -y kubelet=1.28.x--00
kubectl=1.28.x--00 && \
apt-mark hold kubelet kubectl
-
# you can also use the following method
apt-get update && \
apt-get install -y --allow-change-held-packages
kubelet=1.28.x--00 kubectl=1.28.x--00
```

Upgrade Worker Nodes

Here is how to restart the kubelet and uncordon the node:

Example

```
#Restart the kubelet:
sudo systemctl daemon-reload
sudo systemctl restart kubelet

#Uncordon the node
#Bring the node back online by marking it schedulable:

# replace <node-to-drain> with the name of your node
kubectl uncordon <node-to-drain>
```

Verify the Status of the Cluster

After upgrading the kubelet on all the nodes, the status of the cluster must be checked. This is done to verify the availability of all the nodes.

It can be done by running the following command :

Example

```
kubectl get nodes
```

Note

The Status column should display **ready** for all the nodes. The version number should also be updated.

Recovering from a Failure State

If an operation fails, for example, if **kubeadm upgrade** fails, kubeadm can be recovered from a bad state.

Rerun **kubeadm upgrade**

Run **kubeadm upgrade apply --force** without changing the version that the cluster is running

Reset Kubernetes Cluster

The following command reverts changes made by kubeadm init or kubeadm join on the host.

On worker node: **kubeadm reset**

On master node: **kubeadm reset –f**

Upgrading Kubernetes Versions of Control Plane

Duration: 10 minutes

Problem Statement:

You have been given a task to upgrade a Kubernetes cluster. This involves upgrading the control plane, upgrading the worker nodes, and validating the cluster upgrade by creating a pod. Each of these steps must be carefully executed to ensure the cluster operates smoothly and efficiently post-upgrade.

Outcome:

By the end of this demo, you will be able to upgrade the control plane and worker nodes of a Kubernetes cluster and validate the upgrade by creating a pod, ensuring smooth operation and optimal performance.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Duration: 10 minutes

Steps to be followed:

1. Upgrade the control plane
2. Upgrade the worker node
3. Validate the cluster upgrade by creating a pod

ASSISTED PRACTICE

Quick Check

A DevOps engineer needs to gather detailed information about a specific pod within a Kubernetes cluster for troubleshooting purposes. Which command should the engineer use to get detailed information about the pod?

A. kubectl describe pod [pod-name]

B. kubectl get pod [pod-name]

C. kubectl logs pod [pod-name]

D. kubectl exec -it [pod-name] -- /bin/bash

Managing Kubernetes Objects

Kubernetes Objects

Kubernetes objects are persistent entities that represents the state of the cluster. They describe:

Available resources

Containerized applications that are running

Policies around how the applications behave

Understanding Kubernetes Objects

Kubernetes objects are persistent entities in the system. Kubernetes uses these entities to represent the state of the cluster.

List API Resources

```
kubectl api-resources
```

Resource spec

```
kubectl explain pod.spec.containers
```

Object Spec and Status

A Kubernetes object usually includes two nested object fields, namely, object **spec** and object **status**. They govern the object's configuration.

spec

Describes the **desired state** of the object

status

Describes the **current state** of the object

Managing Kubernetes Objects

The kubectl command provides different ways to create and manage Kubernetes objects. Techniques that may be employed for managing Kubernetes objects are mentioned below:

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Imperative commands	Live objects	Development projects	1+	Lowest
Imperative object configuration	Individual files	Production projects	1	Moderate
Declarative object configuration	Directories of files	Production projects	1+	Highest

Deploying and Verifying Kubernetes Objects

Duration: 10 minutes

Problem Statement:

You have been given a task to set up and verify deployment and pod objects in a Kubernetes cluster. This involves creating deployment configurations, initiating pod objects, and validating their successful operation within the cluster to ensure proper configuration and functionality.

Outcome:

By the end of this demo, you will be able to create and verify deployment and pod objects in a Kubernetes cluster, ensuring configurations are correct for efficient operation.

Note: Refer to the demo document for detailed steps

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

Duration: 05 minutes

1. Create and verify the deployment and pod objects

ASSISTED PRACTICE

Business Benefits of Using Kubernetes Clusters

The advantages of using it are:

1) Simplified Deployment and Scaling: Spotify

Spotify uses Kubernetes for quick deployments and dynamic scaling, enhancing development speed and reducing complexity.

2) Increased Reliability and Availability: Pinterest

Pinterest leverages Kubernetes for automated failover and load balancing, ensuring minimal downtime.

3) Optimized Resource Utilization: Zalando

Zalando employs Kubernetes for automated resource scaling, maximizing efficiency and reducing costs.

Business Benefits of Using Kubernetes Clusters

The advantages of using it are:

4) Enhanced Developer Efficiency: Airbnb

Airbnb uses Kubernetes to streamline development, allowing developers to focus on coding.

5) Cost Savings: The New York Times

The New York Times optimizes cloud resource usage with Kubernetes, significantly reducing costs.

6) Improved Security: The Financial Times

The Financial Times enhances security with Kubernetes' role-based access control and network policies.

Organizations Using Kubernetes Clusters

Notable companies leveraging Kubernetes clusters are as follows:

Google	Uses Kubernetes to manage its large-scale, containerized applications with high efficiency and reliability
Spotify	Utilizes Kubernetes to orchestrate its microservices architecture, enabling seamless scaling and deployment
airbnb	Leverages Kubernetes to ensure high availability and resilience of its infrastructure, supporting millions of users globally
Pinterest	Integrates Kubernetes to streamline its development workflows, enhancing the speed and agility of its e-commerce platform

Quick Check

A developer needs to update the configuration of a Kubernetes object to change its resource limits. They have the updated configuration file ready. Which command should they use to apply these changes to the cluster?

A. kubectl replace –f

B. kubectl edit

C. kubectl patch

D. kubectl apply -f

Key Takeaways

- Kubernetes clusters allow containers to run across multiple machines and environments: virtual, physical, cloud-based, and on-premises.
- Cloud infrastructure technologies lets one run Kubernetes on public, private, and hybrid clouds.
- The kubectl command-line tool supports several different ways to create and manage Kubernetes objects.
- Connections from the API server to a node, pod, or service default to plain HTTP connections are neither authenticated nor encrypted.

Managing Multiple Namespaces with Respective Roles

Duration: 25 minutes

Project agenda: To manage and set permissions for multiple namespaces and roles within a Kubernetes cluster, ensuring structured access control and operational security across different environments

Description: As Kubernetes is adopted within larger organizations, it is crucial to manage access for various teams effectively. In this project, you will simulate an environment where there are two namespaces: Simplilearn and CKA. These namespaces represent different teams or projects. You will set up service accounts, roles, and role bindings to regulate access within these namespaces.

LESSON-END PROJECT

Managing Multiple Namespaces with Respective Roles

Duration: 25 minutes

Steps to perform:

1. Validate the Kubernetes cluster
2. Set up the Simplilearn namespace and users
3. Check API access as ServiceAccount in the Simplilearn namespace
4. Set up the CKA namespace and users
5. Check API access as ServiceAccount in CKA namespace

Expected deliverables: A Kubernetes cluster with the Simplilearn and CKA namespaces, along with the respective service accounts, roles, and role bindings

LESSON-END PROJECT

Thank You