Kubernetes Documents

**understanding-kubernetes-architecture**

**Master Node:** The master node is responsible for controlling and managing the cluster. It consists of several key components:

**API Server:** This is the central control point for the Kubernetes cluster. It exposes the Kubernetes API, which allows users and other components to interact with the cluster.

**Controller Manager:** The controller manager oversees the various controllers that regulate the desired state of the cluster. Examples of controllers include the Replication Controller, Deployment Controller, and StatefulSet Controller.

**Scheduler:** The scheduler is responsible for placing new containers onto nodes based on resource availability and constraints.

**etcd:** This is a distributed key-value store that stores the configuration and state information of the entire cluster. It serves as the cluster's "source of truth" for all cluster data.

**Node (Minion) Node:** Nodes are worker machines responsible for running containers. Each node hosts a set of containers and has the following components:

**Kubelet:** The Kubelet is an agent that runs on each node and communicates with the master node. It ensures that containers are running in a Pod and reports the node's status back to the master.

**Kube Proxy:** Kube Proxy maintains network rules on nodes and enables communication between Pods and services on the cluster.

**Container Runtime:** This is the software responsible for running containers. Kubernetes supports various container runtimes, such as Docker, containerd, and CRI-O.

**All Kubernetes Objects**

**Pod:** The smallest deployable unit in Kubernetes, representing a single instance of a running process. Pods can contain one or more containers.

**Service:** A network abstraction that provides a way to access a set of Pods using a stable IP and port, allowing load balancing and service discovery.

**ReplicaSet:** Ensures a specified number of replica Pods are running at all times. It is often used with Deployments for managing rolling updates.

**Deployment:** Manages the deployment and scaling of Pods. It allows you to declaratively update and roll back applications.

**StatefulSet:** Manages the deployment of stateful applications that require stable network identities and persistent storage.

**DaemonSet:** Ensures that a copy of a specific Pod is running on all or selected nodes in the cluster.

**Job:** Runs a task to completion, such as a batch job, and then terminates. Jobs can be used for one-off or periodic tasks.

**CronJob:** Similar to a Job, but it runs based on a schedule defined using the cron syntax.

**Namespace:** Provides a way to partition resources and isolate workloads within a cluster.

**ConfigMap and Secret:** Store configuration data and sensitive information, respectively, that can be consumed by Pods.

**PersistentVolume (PV) and PersistentVolumeClaim (PVC):** PV represents a physical storage resource in the cluster, while PVC is a request for storage by a user or application.

**ServiceAccount:** Provides an identity for Pods, allowing them to authenticate and interact with other resources in the cluster.

**Role and ClusterRole:** Define a set of permissions (verbs) on specific resources (nouns) within a namespace or cluster-wide.

**RoleBinding and ClusterRoleBinding:** Bind a Role or ClusterRole to a user, group, or ServiceAccount to grant the defined permissions.

**NetworkPolicy:** Defines network access rules for Pods, allowing you to control communication between them.

**PodSecurityPolicy:** Defines security settings for Pods, specifying what security features are allowed.

**HorizontalPodAutoscaler (HPA):** Automatically adjusts the number of Pods based on CPU or custom metrics, ensuring optimal resource utilization.

**VerticalPodAutoscaler (VPA):** Adjusts resource requests and limits of Pods based on historical usage data.

**Endpoints:** Exposes the network address and port of a set of Pods as a single Service.

**Ingress:** Manages external access to services by providing routing rules and exposing services via HTTP and HTTPS.

**PodDisruptionBudget:** Limits the number of Pods that can be simultaneously disrupted during events like node maintenance.

**LimitRange:** Specifies resource limits and usage policies for a namespace.

**Different between replication controller, replicaset and deployment**

In Kubernetes, Replication Controller, ReplicaSet, and Deployment are three different resources used to ensure that a specified number of replicas of a pod are running at all times.

**Replication Controller:** A Replication Controller is the oldest and the simplest form of replication in Kubernetes. It ensures that a specified number of replicas of a pod are running at all times. However, it does not support features such as scaling, rolling updates, or advanced selectors.

**ReplicaSet:** A ReplicaSet is an enhanced version of the Replication Controller. It is designed to handle the limitations of Replication Controllers and provides more features such as advanced selectors, scaling, and rolling updates. A ReplicaSet ensures that a specified number of replicas of a pod are running at all times and replaces the Replication Controller in modern Kubernetes deployments.

**Deployment:** A Deployment is the recommended way to manage the lifecycle of a ReplicaSet. It provides a higher-level abstraction that allows for rolling updates, rollbacks, and scaling. A Deployment manages the ReplicaSet and ensures that the desired state of the ReplicaSet is maintained. It also provides features such as versioning and revision history.

**label and selector in kubernetes**

In Kubernetes, labels and selectors are used to identify and group related objects. Labels are key-value pairs that are attached to Kubernetes objects such as pods, services, and deployments. Selectors are used to filter these objects based on their labels.

Labels:

Labels are used to attach metadata to Kubernetes objects. A label consists of a key-value pair and can be attached to any Kubernetes object. Labels are used to identify and group related objects. For example, you can attach the label "app=frontend" to all the pods and services that belong to the frontend application. You can use labels to organize your resources, create logical groupings, and apply policies to specific groups of resources.

Selectors:

Selectors are used to filter Kubernetes objects based on their labels. A selector is a set of rules that define the criteria for selecting objects based on their labels. For example, you can create a selector that selects all the pods that have the label "app=frontend". Selectors are used by Kubernetes to group related objects and apply policies to specific groups of resources.

Labels and selectors are used together to organize and manage Kubernetes objects. You can use labels to group related objects and selectors to filter them based on their labels. For example, you can create a service that selects all the pods with the label "app=frontend" and exposes them to the outside world. This way, you can ensure that only the pods that belong to the frontend application are exposed and not the other pods in the cluster.

ResourceQuota

In Kubernetes, a "ResourceQuota" is a resource management mechanism that allows you to set limits on the amount of computational resources (such as CPU and memory) and the number of objects (such as Pods, Services, and ConfigMaps) that can be consumed within a namespace. ResourceQuotas are used to prevent a single application or user from monopolizing cluster resources and ensure fair resource allocation.

apiVersion: v1

kind: ResourceQuota

metadata:

name: my-resource-quota

spec:

hard:

cpu: "1"

memory: 1Gi

pods: "10"

services: "5"

**Services**

"Service" is an abstraction that defines a logical set of Pods and a policy by which to access them. Services enable network communication between different sets of Pods, both within the same namespace and across namespaces. They provide a stable IP address and DNS name to access Pods, even as Pods come and go due to scaling, upgrades, or failures.

**ClusterIP:** A ClusterIP Service exposes a set of Pods to other resources within the cluster. It provides an internal IP address that other resources can use to access the Pods. This type of Service is suitable for intra-cluster communication.

**NodePort:** A NodePort Service exposes the Pods on a specific port on each node in the cluster. It creates a mapping from a high-range port on the node to the port on the Pods. This type of Service is often used to expose applications externally, but it's not recommended for production use due to potential security concerns.

**LoadBalancer:** A LoadBalancer Service is typically used in cloud environments. It provisions a cloud-specific load balancer that distributes traffic across the Pods. This type of Service provides external access to the application and automatically balances the load.

**ExternalName:** An ExternalName Service maps a Service to an external DNS name. It's useful when you want to provide a DNS alias for a service that resides outside the cluster, like an external database.

**Taint and toleration**

In Kubernetes, taints and tolerations are used to control which pods are scheduled on which nodes in a cluster.

Taints:

A taint is a property of a node in a Kubernetes cluster that repels pods. Taints can be used to prevent pods from being scheduled on nodes that are not suitable for them. For example, you can add a taint to a node that indicates that it is running critical workloads and should not be used for general-purpose workloads.

Tolerations:

A toleration is a property of a pod in a Kubernetes cluster that allows it to be scheduled on a node with a matching taint. Tolerations are used to specify that a pod is willing to tolerate nodes with certain taints. For example, you can add a toleration to a pod that allows it to be scheduled on a node with a specific taint. This way, you can ensure that only the pods that are suitable for a specific node are scheduled on it.

Taints and tolerations are used together to control which pods are scheduled on which nodes in a Kubernetes cluster. You can use taints to mark nodes as unsuitable for certain types of workloads and tolerations to allow certain pods to be scheduled on those nodes. This way, you can ensure that your workloads are running on the most suitable nodes in your cluster and avoid resource wastage.

Use the following command to apply the taint to the desired node:

kubectl taint nodes <node\_name> run=mypod:NoSchedule

Use the following YAML definition to create the pod with the toleration:

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- name: mycontainer

image: nginx

tolerations:

- key: key1

operator: Equal

value: value1

effect: NoSchedule

**DaemonSets** are used to ensure that some or all of your K8S nodes run a copy of a pod, which allows you to run a daemon on every node.

When you add a new node to the cluster, a pod gets added to match the nodes. Similarly, when you remove a node from your cluster, the pod is put into the trash. Deleting a DaemonSet cleans up the pods that it previously created.

a **"static pod"** is a Pod whose configuration is managed directly by the kubelet on a specific node, rather than being managed through the Kubernetes API server and stored in the cluster's etcd datastore. Static pods are created and managed by the kubelet running on each node, making them a useful approach for running essential system-level services that need to start before the kubelet itself.

Rolling back a Kubernetes Deployment involves reverting to a previous known state of the application when an update or change has caused issues or unexpected behavior. Kubernetes provides a straightforward way to perform rollbacks using the kubectl command-line tool or by editing the Deployment directly.

kubectl rollout undo deployment/<deployment-name>

If you want to roll back to a specific revision, you can use the --to-revision flag:

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

**ConfigMaps** are a way to manage configuration data separately from your application code or Docker images. They allow you to decouple configuration settings, making it easier to manage and update configuration across multiple Pods or containers. ConfigMaps come in two types: literal and file-based. The choice of type depends on the nature of the configuration data you're managing.

apiVersion: v1

kind: ConfigMap

metadata:

name: my-config

data:

DATABASE\_URL: "mysql://user:password@database-host:3306/db"

API\_KEY: "your-api-key"

How to use in Pod

apiVersion: v1

kind: Pod

metadata:

name: my-pod

spec:

containers:

- name: my-app

image: my-image

envFrom:

- configMapRef:

name: my-config

"Secrets" are a way to securely store and manage sensitive information, such as passwords, API tokens, and other confidential data.

kubectl create secret generic db-secret --from-literal=username=dbuser --from-literal=password=Y4nys7f11

Provide reference in pod

apiVersion: v1

kind: Pod

metadata:

name: secret-demo-1

spec:

containers:

- name: demo-container

image: nginx

env:

- name: Username

valueFrom:

secretKeyRef:

name: db-secret

key: username

An "Ingress" is an API object that manages external access to services within a cluster. It provides a way to define rules for routing incoming HTTP and HTTPS traffic to different services based on the requested host or path. Ingress resources enable you to expose your applications to the outside world without exposing the details of individual services.

Key features of Kubernetes Ingress:

**Rules and Paths:** Ingress allows you to define rules that determine how incoming requests are routed to different services based on host names, paths, or other request attributes.

**TLS Termination:** Ingress supports secure traffic using Transport Layer Security (TLS). You can configure SSL termination at the Ingress level to handle HTTPS traffic.

**Load Balancing:** Ingress controllers (explained below) typically integrate with load balancers or reverse proxies to distribute traffic to the appropriate backend services.

**Ingress Controllers:** An Ingress resource itself doesn't provide any routing functionality. Instead, you need to use an Ingress controller, which is a software component responsible for interpreting Ingress rules and routing traffic accordingly. Popular Ingress controllers include Nginx Ingress Controller, Traefik, and HAProxy Ingress.

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-ingress

spec:

rules:

- host: myapp.example.com

http:

paths:

- path: /app

pathType: Prefix

backend:

service:

name: app-service

port:

number: 80

"Network Policies" are a set of rules that define how pods are allowed to communicate with each other across the network within a cluster. Network Policies provide a way to enforce security and isolation at the network level, allowing you to control and restrict communication between pods based on labels, namespaces, and other criteria.

Here are the key features and concepts of Network Policies:

**Isolation and Segmentation:** Network Policies enable you to isolate and segment pods within a namespace, ensuring that only specific pods can communicate with each other.

**Pod Selectors:** Network Policies are defined using pod selectors to specify which pods the policy should apply to. This allows you to target specific groups of pods based on labels.

**Ingress and Egress Rules:** Network Policies include both ingress (incoming) and egress (outgoing) rules. You can define rules that control which pods are allowed to send traffic to and receive traffic from other pods.

**Namespace Scope:** Network Policies are applied at the namespace level. This means that policies are effective only within the same namespace where they are defined.

**Default Policies:** By default, pods within a namespace can communicate with each other without any restrictions. Network Policies allow you to explicitly define rules to restrict this communication.

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-app-to-db

spec:

podSelector:

matchLabels:

app: app-pod

ingress:

- from:

- podSelector:

matchLabels:

app: db-pod

The Network Policy named allow-app-to-db allows pods with the label app: app-pod to receive incoming traffic from pods with the label app: db-pod.

a "StatefulSet" is a workload object designed to manage stateful applications that require unique network identifiers, stable storage, and ordered scaling and updating. StatefulSets are particularly useful for applications like databases, messaging queues, and other stateful workloads where each instance has its own identity and must be managed individually.

Kubernetes Storage Explained

PersistentVolume (PV) and PersistentVolumeClaim (PVC)

We use PV object to represent external storage volume. A single external storage volume can be represented by a single PV. So PV goes with external volumes in 1 to 1 relationship.

PVC goes with Pod in 1 to 1 relationship. The Pods needs a PVC in order to claim ownership of a PV. A valid PVC allows a Pod to mount a PV as its volume.

How PVC binds to PV is defined by Access Mode, with three options. Note that the options are effective for the entire PV

RWO (ReadWriteOnce): allowing the PV to be bound to a single PVC (for read write). This mode is typically used in block storage;

RWM (ReadWriteMany): allowing the PV to be bound to multiple PVCs (for read write). This mode is only supported by file (e.g. NFS) and object storage;

ROM (ReadOnlyMany): allowing the PV bound to multiple PVCs for read only.

When a PVC is released, what to do with the PV is defined as persistentVolumeReclaimPolicy, and the two options (effective at PV level) are:

Delete

Retain

Static Provisioning and Dynamic Provisioning

With static provisioning, the external storage volume must be pre-created. In this context, a PV object represents a pre-created external storage volume. So PVs must be explicit declared. The K8s literature also refers to such PVs as pre-created PV.

With dynamic provisioning, the external storage volume is provisioned dynamically. Therefore, you do not need to explicitly create PVs. By the same token, access mode does not apply.

Instead of PV, now we need to explicitly declare storage class, which specifies how to dynamically provision PVs, with the following properties:

volumeBindingMode defines when the binding and provisioning of a PersistentVolume occurs, with two options:

Immediate (default)

WaitForFirstConsumer (recommended): delays until a Pod using the PVC is created

**Pod.yaml**

apiVersion: v1

kind: Pod

metadata:

name: app

spec:

containers:

- name: app

image: centos

command: ["/bin/sh"]

args: ["-c", "while true; do echo $(date -u) >> /data/out.txt; sleep 5; done"]

volumeMounts:

- name: persistent-storage

mountPath: /data

volumes:

- name: persistent-storage

persistentVolumeClaim:

claimName: ebs-claim

pvc.yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: ebs-claim

spec:

accessModes:

- ReadWriteOnce

storageClassName: ebs-sc

resources:

requests:

storage: 1Gi

**storageclass.yaml**

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

name: ebs-sc

provisioner: ebs.csi.aws.com

parameters:

type: gp2

volumeBindingMode: WaitForFirstConsumer

a "Headless Service" is a type of service that is used to enable DNS-based service discovery for pods in a StatefulSet. Unlike regular services, which provide load balancing and route traffic to one or more pods, a Headless Service does not create a cluster IP and does not load balance traffic. Instead, it exposes the individual pods directly, allowing clients to connect to specific instances using their DNS names.

a "multi-container pod" is a Pod that contains more than one container. These containers share the same network namespace, storage volumes, and lifecycle. Multi-container pods are a way to co-locate tightly coupled, related containers within the same Pod, allowing them to communicate with each other more efficiently.

Helper Containers: One common use case for multi-container pods is to include a helper container that performs tasks like logging, monitoring, or backup operations that are related to the main application container.

Sidecar Containers: A sidecar container is a pattern where a main application container is paired with a supporting container that provides additional functionality. For example, a main application container might have a sidecar container responsible for handling TLS termination, data synchronization, or authentication.

Role-Based Access Control (RBAC) is a fundamental security mechanism in Kubernetes that allows you to control and manage access to resources within a Kubernetes cluster. RBAC enables you to define fine-grained permissions for users, groups, or service accounts, determining who can perform specific actions on specific resources in the cluster.

Key concepts and components of RBAC in Kubernetes:

Roles and ClusterRoles:

Role: A Role is a set of rules that defines what actions (verbs) are allowed on which resources (API groups, resources, and resource names) within a specific namespace.

ClusterRole: A ClusterRole is similar to a Role, but it applies cluster-wide rather than within a single namespace.

RoleBindings and ClusterRoleBindings:

RoleBinding: A RoleBinding binds a Role to a user, group, or service account within a specific namespace.

ClusterRoleBinding: A ClusterRoleBinding binds a ClusterRole to a user, group, or service account cluster-wide.

Verbs and Resources:

Verbs: Verbs specify the actions that a user or group is allowed to perform (e.g., create, get, list, update, delete).

Resources: Resources specify the Kubernetes API resources that actions can be performed on (e.g., pods, services, deployments).

# Define a Role

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: my-namespace

name: pod-reader

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list"]

# Bind the Role to a User

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: pod-reader-binding

namespace: my-namespace

subjects:

- kind: User

name: alice

apiGroup: rbac.authorization.k8s.io

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

In this example, the **pod-reader** Role grants permission to **get** and **list** pods in the **my-namespace** namespace. The **pod-reader-binding** RoleBinding binds this Role to the user **alice**.

a "Service Account" is a resource that provides an identity for pods and processes running within a cluster. Each pod can be associated with a specific service account, which allows it to authenticate with the Kubernetes API server and perform actions based on the permissions granted to that service account through Role-Based Access Control (RBAC).

a "Security Context" is a set of configurations that allow you to control the security settings and permissions for containers running within a Pod. Security contexts are defined at both the Pod and container levels, providing fine-grained control over the behavior of containers in terms of privilege levels, user and group IDs, filesystem permissions, and more.

Pod-Level Security Context: The securityContext field can be defined at the Pod level in the Pod specification. It applies to all containers within the Pod unless overridden at the container level.

Container-Level Security Context: Each container within a Pod can also have its own securityContext, allowing you to customize security settings on a per-container basis.

Capabilities: Security contexts can limit the Linux capabilities of containers. This restricts the actions containers can perform on the host system.

RunAsUser and RunAsGroup: You can specify user and group IDs for the container processes using the runAsUser and runAsGroup fields in the security context. This helps in limiting the privileges of the container.

Demo.yaml

apiVersion: v1

kind: Pod

metadata:

name: security-context-demo

spec:

securityContext:

runAsUser: 1000

runAsGroup: 3000

fsGroup: 2000

volumes:

- name: demo-vol

emptyDir: {}

containers:

- name: sc-demo

image: busybox:1.28

command: [ "sh", "-c", "sleep 1h" ]

volumeMounts:

- name: demo-vol

mountPath: /data/demo

**common pod issue and resolution**

Pods in Kubernetes can encounter various issues, some common ones and their possible resolutions are:

**CrashLoopBackOff:** This error occurs when a container keeps crashing and restarting repeatedly. To resolve this issue, check the container logs to identify the cause of the crash and fix the issue.

**ImagePullBackOff:** This error occurs when a pod is unable to pull the container image from the registry. To resolve this issue, check the image name and make sure it is correct and the registry is accessible.

**Pending:** This error occurs when a pod is stuck in the pending state and is unable to schedule on a node. To resolve this issue, check the available resources on the node and ensure that the pod's resource requests are within the node's capacity.

**Evicted:** This error occurs when a pod is evicted from a node due to resource constraints or other issues. To resolve this issue, check the reason for the eviction and either adjust the resource limits or fix the underlying issue.

**OutOfMemory:** This error occurs when a container runs out of memory. To resolve this issue, increase the memory limit for the container or adjust the application to consume less memory.

**Connection Refused:** This error occurs when a container is unable to connect to a service or endpoint. To resolve this issue, check the endpoint and make sure it is available and accessible.

**ErrImagePull:** This error occurs when a pod is unable to pull an image from the registry due to authentication issues. To resolve this issue, check the credentials and make sure they are correct and have the necessary permissions.

These are just a few common pod issues and resolutions. In general, it is important to monitor pods and their logs regularly to identify and resolve issues quickly to ensure the availability and reliability of the applications running on Kubernetes.

**kubernetes probe**

In Kubernetes, a probe is a diagnostic performed by the kubelet on a running container. It is used to determine the health of a container. Kubernetes supports two types of probes:

**Liveness probe:** This type of probe is used to check if the container is alive. It indicates whether the container is running and responding to requests. If the liveness probe fails, Kubernetes will kill the container and restart it.

**Readiness probe:** This type of probe is used to check if the container is ready to serve requests. It indicates whether the container is fully started and able to handle requests. If the readiness probe fails, Kubernetes will stop sending requests to the container until it passes the probe.

**How to restore if kubernetes master node goes down**

If the Kubernetes master node goes down, there are several steps you can take to restore the cluster:

**Recover the etcd cluster:** The etcd cluster is a key-value store that stores the entire state of the Kubernetes cluster. If the etcd cluster is lost, the entire cluster will be lost. To recover the etcd cluster, you can restore from a backup or create a new etcd cluster and restore the data.

**Restore the master node:** If the master node is lost, you can restore it from a backup or create a new master node. To restore the master node, you need to install all the necessary components, including the API server, etcd, controller manager, and scheduler.

**Rejoin the nodes:** Once the master node is restored, you need to rejoin the worker nodes to the cluster. This can be done by running the kubeadm join command on each node.

**Verify the cluster:** After the master node and worker nodes are restored, you should verify that the cluster is running correctly by checking the status of the Kubernetes components and applications running on the cluster.

It is important to have a disaster recovery plan in place to ensure that your Kubernetes cluster can be restored quickly and efficiently in the event of a failure. This includes regular backups of the etcd cluster and the Kubernetes configuration files, as well as documentation and procedures for restoring the cluster.