**Kubernetes Interview Questions and Answers Part 1**

**1. What is the difference between a Deployment and a StatefulSet in Kubernetes?**

**Answer:**  
A Deployment is used for stateless applications where the state of the application does not need to be preserved. It manages replica sets and ensures that the desired number of pod replicas are running.

A StatefulSet, on the other hand, is used for stateful applications where each pod needs to be uniquely identified and must retain its state even after rescheduling. It assigns a stable identity (DNS name and persistent volume) to each pod.

**Key differences:**

* **Pod Identity:** StatefulSet pods have stable hostnames (e.g., web-0, web-1) while Deployment pods do not.
* **Storage:** StatefulSets are typically used with persistent volume claims (PVCs) that are retained even when pods are deleted.
* **Scaling behavior:** StatefulSets create/delete pods in order, while Deployments do it in parallel.

**Example Use Case:**

* **Deployment**: Web servers, microservices.
* **StatefulSet**: Databases like MySQL, MongoDB, Kafka.

**2. Explain how Kubernetes handles rolling updates and rollbacks.**

**Answer:**  
Kubernetes supports rolling updates using the Deployment controller. It updates pods in a controlled fashion to minimize downtime. The process gradually replaces old pods with new ones, ensuring that some pods are always running.

**Key Concepts:**

* **RollingUpdate Strategy**: Default update strategy in Deployments.
  + maxUnavailable: How many pods can be unavailable during the update.
  + maxSurge: How many extra pods can be created during the update.

**Rollback:**  
If a rollout fails or a bug is introduced, Kubernetes allows you to rollback to the previous version using:

bash

CopyEdit

kubectl rollout undo deployment <deployment-name>

**Commands:**

bash

CopyEdit

kubectl rollout status deployment my-deployment

kubectl rollout history deployment my-deployment

kubectl rollout undo deployment my-deployment

**Benefits:**

* Zero-downtime deployment.
* Version tracking of rollouts.
* Fast recovery from bad deployments.

**3. What are Kubernetes DaemonSets and give a real-world use case?**

**Answer:**  
A DaemonSet ensures that a copy of a pod runs on **every node** (or a subset of nodes) in the cluster.

**Use Cases:**

* Running a node-level monitoring agent (e.g., Prometheus Node Exporter).
* Running a log collector (e.g., Fluentd, Filebeat).
* Running a storage driver or system daemon.

**Behavior:**  
When a new node is added to the cluster, the DaemonSet controller automatically schedules the pod on the new node.

**Sample YAML:**

yaml

CopyEdit

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: fluentd

spec:

selector:

matchLabels:

name: fluentd

template:

metadata:

labels:

name: fluentd

spec:

containers:

- name: fluentd

image: fluent/fluentd

**4. What are the differences between ConfigMap and Secret in Kubernetes?**

**Answer:**  
Both ConfigMap and Secret are used to inject configuration into pods, but they serve different purposes.

| **Feature** | **ConfigMap** | **Secret** |
| --- | --- | --- |
| Purpose | Store non-sensitive config data | Store sensitive data (passwords, tokens) |
| Encoding | Plain text | Base64 encoded |
| Security | Not encrypted | Can be encrypted at rest (KMS) |
| Volume mount | Yes | Yes |
| Environment var | Yes | Yes |

**When to use:**

* **ConfigMap**: App configurations, URLs, filenames.
* **Secret**: Passwords, OAuth tokens, SSH keys.

**Example:**

yaml

CopyEdit

apiVersion: v1

kind: Secret

metadata:

name: db-secret

type: Opaque

data:

username: dXNlcg== # base64 of 'user'

password: cGFzcw== # base64 of 'pass'

**5. What is the role of etcd in a Kubernetes cluster?**

**Answer:**  
etcd is a distributed key-value store used by Kubernetes to store all cluster data. It acts as the **source of truth** for the Kubernetes control plane.

**Responsibilities:**

* Stores configuration data, cluster state, secrets, and service discovery info.
* Supports consistent reads and writes across the cluster.

**Key Features:**

* Highly available and distributed.
* Uses Raft consensus algorithm for leader election and consistency.
* Requires backup and disaster recovery strategy.

**Security Considerations:**

* TLS encryption.
* Authentication and access control.

**Command Example:**  
To view keys in etcd (if directly accessing):

ETCDCTL\_API=3 etcdctl get "" --prefix --keys-only

**6. How does Kubernetes networking work? Explain the Pod-to-Pod communication.**

**Answer:**  
Kubernetes networking is based on a **flat network model**, where all pods can communicate with each other without NAT (Network Address Translation), regardless of the node they’re on.

**Core Networking Rules:**

1. Every pod gets its own IP address.
2. All pods can reach each other directly via IP.
3. Containers within a pod share the same network namespace (localhost).
4. Communication between pods across nodes is handled by CNI plugins (e.g., Calico, Flannel, Cilium).

**Pod-to-Pod Communication Steps:**

* Pod A wants to reach Pod B using Pod B’s IP.
* If both pods are on the same node: traffic stays local.
* If on different nodes: the packet is routed via virtual network interfaces provided by the CNI plugin.

**Example Tools:**

* kubectl exec to test network:

kubectl exec -it pod-a -- curl <pod-b-ip>:<port>

**Why It Matters:**  
Understanding pod communication helps in debugging network issues, setting up network policies, and optimizing inter-service communication.

**7. What are Kubernetes Network Policies and how do they work?**

**Answer:**  
A NetworkPolicy in Kubernetes is a resource used to control traffic flow **at the IP address or port level** between pods.

**Default Behavior:**  
By default, all pods can communicate with all other pods. NetworkPolicies **restrict** this.

**Features:**

* Allow/deny ingress (incoming) or egress (outgoing) traffic.
* Based on labels, namespaces, and IP blocks.
* Require a CNI plugin that supports NetworkPolicy (e.g., Calico, Cilium).

**Example: Allow ingress traffic to pods with label app=db only from app=web:**

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-web-to-db

spec:

podSelector:

matchLabels:

app: db

ingress:

- from:

- podSelector:

matchLabels:

app: web

**Note:** NetworkPolicies are additive — start from "deny all" and explicitly allow traffic.

**8. Explain the Kubernetes control plane components and their responsibilities.**

**Answer:**  
The Kubernetes control plane is the **brain** of the cluster and consists of several components:

1. **kube-apiserver**: Entry point for all REST API requests. Validates and processes requests.
2. **etcd**: Stores all cluster state and configuration data.
3. **kube-scheduler**: Assigns pods to nodes based on resource availability and constraints.
4. **kube-controller-manager**: Runs all controllers (e.g., Node, Deployment, ReplicaSet controllers) to ensure the desired state.
5. **cloud-controller-manager**: Manages cloud provider-specific control loops (like load balancers, volumes).

**Workflow Example:**

* You apply a deployment YAML.
* kube-apiserver receives it.
* etcd stores the desired state.
* kube-controller-manager ensures replicas match.
* kube-scheduler assigns pods to nodes.

This architecture allows scalability, self-healing, and declarative infrastructure.

**9. What is the role of Kubernetes Ingress and how does it differ from a Service?**

**Answer:**  
An Ingress is a Kubernetes resource that manages **external access to services**, typically HTTP/HTTPS traffic. It acts as a **Layer 7 (application layer) reverse proxy**.

**Service vs Ingress:**

| **Feature** | **Service** | **Ingress** |
| --- | --- | --- |
| Layer | L4 (TCP/UDP) | L7 (HTTP/HTTPS) |
| Purpose | Expose internal or external service | Expose multiple services under a single IP |
| Port-based | Yes | No (path/host-based rules) |
| TLS Support | Manually via Service/LoadBalancer | Built-in SSL termination |

**Example Ingress YAML:**

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: web-ingress

spec:

rules:

- host: example.com

http:

paths:

- path: /app1

pathType: Prefix

backend:

service:

name: app1-service

port:

number: 80

**Ingress Controllers** (e.g., NGINX, Traefik) are required to implement ingress rules.

**10. How do you perform debugging in Kubernetes when a pod is not starting?**

**Answer:**  
When a pod fails to start, you can debug it using several tools:

1. **Describe the pod**:

kubectl describe pod <pod-name>

Check events at the bottom — look for image pull errors, scheduling issues, resource limits.

1. **Check logs**:

kubectl logs <pod-name>

If multiple containers:

kubectl logs <pod-name> -c <container-name>

1. **Check status**:

kubectl get pods

Look for status like CrashLoopBackOff, ImagePullBackOff, Pending.

1. **Exec into container** (if running):

kubectl exec -it <pod-name> -- /bin/sh

1. **Use ephemeral debug containers**:

kubectl debug -it <pod-name> --image=busybox

1. **Check node status**:

kubectl get nodes

kubectl describe node <node-name>

**Common Issues:**

* Missing ConfigMap/Secret
* Insufficient resources
* Incorrect image or tag
* Network policy restrictions
* Node taints or affinity rules

**11. What is a Kubernetes Job and how does it differ from a Deployment?**

**Answer:**  
A Job in Kubernetes is used to run **one-time or batch tasks** that terminate once completed successfully. It ensures that a pod runs to completion (success or failure) and can be retried if it fails.

A Deployment, by contrast, is used for **long-running, stateless applications** that must be kept running continuously.

**Use Cases for Job:**

* Database migrations
* Sending notification emails
* Data backups

**Types of Jobs:**

* **One-off Job**: Runs a pod to completion once.
* **Parallel Job**: Runs multiple pods in parallel.
* **Completions & Parallelism**: Controls number of successful completions and concurrent pods.

**Example YAML:**

apiVersion: batch/v1

kind: Job

metadata:

name: hello

spec:

template:

spec:

containers:

- name: hello

image: busybox

command: ["echo", "Hello World"]

restartPolicy: Never

**12. What are Init Containers in Kubernetes and why are they used?**

**Answer:**  
Init Containers are specialized containers that run **before the main application container** in a pod. They are used to perform setup tasks such as:

* Waiting for a service to become available
* Cloning a git repository
* Setting environment preconditions

**Features:**

* Run **sequentially** before app containers.
* Must complete **successfully** before main containers start.
* Support different images and tools than main containers.

**Example:**

initContainers:

- name: wait-for-db

image: busybox

command: ['sh', '-c', 'until nslookup mydb; do echo waiting; sleep 2; done']

**Benefits:**

* Separation of concerns.
* Improved modularity and debugging.
* Retry logic before main workload runs.

**13. What are taints and tolerations in Kubernetes?**

**Answer:**  
**Taints** are applied to nodes to prevent pods from being scheduled on them unless the pod has a matching **toleration**.

**Syntax for taint:**

kubectl taint nodes <node-name> key=value:NoSchedule

**Toleration in Pod:**

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

**Taint Effects:**

* NoSchedule: Pod will not be scheduled unless it tolerates the taint.
* PreferNoSchedule: Avoid scheduling if possible.
* NoExecute: Pod is evicted if already running.

**Use Cases:**

* Reserve nodes for specific workloads (e.g., GPU jobs).
* Ensure critical workloads get priority on dedicated nodes.

**14. Explain Kubernetes Pod affinity and anti-affinity with examples.**

**Answer:**  
**Affinity** allows you to constrain which nodes or pods your pod can be scheduled with.

* **Node Affinity**: Schedule pods based on node labels.
* **Pod Affinity**: Schedule pods **near** other pods (e.g., same zone).
* **Pod Anti-Affinity**: Schedule pods **away** from other pods (e.g., for high availability).

**Pod Affinity Example:**

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchLabels:

app: myapp

topologyKey: "kubernetes.io/hostname"

**Pod Anti-Affinity Example:**

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchLabels:

app: myapp

topologyKey: "kubernetes.io/hostname"

**Benefits:**

* High availability
* Resource optimization
* Fault isolation

**15. What are sidecar containers and how are they used in Kubernetes?**

**Answer:**  
A **sidecar container** is a helper container that runs in the same pod as the main application and provides supporting functionality.

**Common Use Cases:**

* Logging agents (e.g., Filebeat)
* Proxies (e.g., Envoy in Istio)
* Data synchronization tools
* Service mesh communication

**Example:**

containers:

- name: app

image: myapp

- name: sidecar

image: busybox

command: ["tail", "-f", "/dev/null"]

**Key Points:**

* Shares the same network namespace and volumes.
* Can start and stop independently of the main container.
* Promotes modularity and separation of concerns.

**16. What are Kubernetes Volumes and how do they differ from Docker volumes?**

**Answer:**  
In Kubernetes, a **Volume** is a directory accessible to containers in a pod that is preserved across container restarts.

**Differences from Docker Volumes:**

* Kubernetes volumes are tied to the pod's lifecycle, not the container.
* Kubernetes supports different volume types (e.g., hostPath, emptyDir, configMap, persistentVolumeClaim).

**Common Volume Types:**

* emptyDir: Temporary storage shared between containers.
* hostPath: Maps a file/directory from the host node.
* persistentVolumeClaim: For persistent storage (e.g., AWS EBS, NFS).

**Example:**

volumes:

- name: data

emptyDir: {}

**Benefits:**

* Share data between containers.
* Preserve logs or cache.
* Mount external storage (persistent).

**17. Explain Kubernetes PersistentVolumes (PV) and PersistentVolumeClaims (PVC).**

**Answer:**  
**PersistentVolume (PV)** is a piece of storage in the cluster that has been provisioned by an admin or dynamically via StorageClass.

**PersistentVolumeClaim (PVC)** is a request for storage by a user, specifying size and access mode.

**Workflow:**

1. Admin creates a PV or defines a StorageClass.
2. User creates a PVC.
3. Kubernetes binds a matching PV to the PVC.
4. Pod mounts the PVC.

**Example PVC:**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mypvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

**Access Modes:**

* ReadWriteOnce — one node read/write.
* ReadOnlyMany — multiple nodes read-only.
* ReadWriteMany — multiple nodes read/write.

**18. What is a Kubernetes ServiceAccount and when do you use it?**

**Answer:**  
A ServiceAccount provides an identity for pods to access the Kubernetes API or external services securely.

**Default Behavior:**

* Every pod is associated with a default service account in its namespace.
* Tokens are automatically mounted into pods via /var/run/secrets/....

**Use Cases:**

* Granting fine-grained API permissions via RBAC.
* Interacting with cloud provider APIs (e.g., via Workload Identity).

**Example:**

apiVersion: v1

kind: ServiceAccount

metadata:

name: read-only

**Mounting in Pod:**

spec:

serviceAccountName: read-only

**RBAC Binding Example:**

kind: RoleBinding

roleRef:

kind: Role

name: pod-reader

apiGroup: rbac.authorization.k8s.io

subjects:

- kind: ServiceAccount

name: read-only

**19. How do you perform a rolling restart of a deployment in Kubernetes?**

**Answer:**  
To trigger a rolling restart of a Deployment (without changing the image or YAML), use:

kubectl rollout restart deployment <deployment-name>

**Why It's Used:**

* To pick up new ConfigMap or Secret values.
* To refresh application state without downtime.
* To clear out memory leaks or stuck connections.

**Alternative Method:**  
Patch a dummy label to trigger the rollout:

kubectl patch deployment <deployment-name> -p \

'{"spec":{"template":{"metadata":{"annotations":{"restartedAt":"'$(date +%s)'"}}}}}'

**Check Progress:**

kubectl rollout status deployment <deployment-name>

**20. What is a Kubernetes Horizontal Pod Autoscaler (HPA) and how does it work?**

**Answer:**  
The Horizontal Pod Autoscaler (HPA) automatically scales the number of pods in a deployment, replication controller, or statefulset **based on CPU/memory usage or custom metrics**.

**How it Works:**

* Monitors pod metrics via the Metrics Server.
* If usage exceeds target thresholds, it increases replicas.
* If usage drops, it scales down.

**Configuration Example:**

kubectl autoscale deployment myapp --cpu-percent=50 --min=2 --max=10

**HPA YAML:**

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

spec:

minReplicas: 2

maxReplicas: 10

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 50

**Dependencies:**

* Metrics Server must be installed.
* Works with custom metrics (Prometheus Adapter) in v2.

**21. What is the difference between StatefulSet and Deployment in Kubernetes?**

**Answer:**  
Both StatefulSet and Deployment manage pods, but they are suited for different use cases.

| **Feature** | **Deployment** | **StatefulSet** |
| --- | --- | --- |
| Pod Identity | Pods are anonymous | Each pod gets a unique identity (pod-0, pod-1) |
| Storage | Ephemeral or shared | Persistent and unique storage per pod |
| Pod Order | No guaranteed order | Maintains startup and termination order |
| Use Case | Stateless apps | Stateful apps (databases, queues) |

**StatefulSet Features:**

* Stable network identities via DNS (pod-0.service.namespace.svc.cluster.local)
* PersistentVolumeClaims per replica
* Controlled scaling and updates

**Example Use Case:**  
Running **MySQL, Cassandra, Kafka** where each instance needs to retain its identity and data.

**22. How does Kubernetes handle Secrets and how can you access them in a pod?**

**Answer:**  
Secrets in Kubernetes store sensitive data such as passwords, OAuth tokens, and SSH keys in base64-encoded format.

**Creating a Secret:**

kubectl create secret generic db-secret --from-literal=username=admin --from-literal=password=pass123

**Accessing in Pod (Environment Variables):**

env:

- name: DB\_USER

valueFrom:

secretKeyRef:

name: db-secret

key: username

**Accessing as Volume:**

volumes:

- name: secret-volume

secret:

secretName: db-secret

**Security Best Practices:**

* Use RBAC to restrict access.
* Use external secret managers (e.g., HashiCorp Vault, AWS Secrets Manager).
* Enable encryption at rest for etcd.

**23. What is the difference between ConfigMap and Secret in Kubernetes?**

**Answer:**

| **Feature** | **ConfigMap** | **Secret** |
| --- | --- | --- |
| Purpose | Store non-sensitive configuration | Store sensitive data (passwords, tokens) |
| Encoding | Plain text | Base64 encoded |
| Access | Environment variables or volumes | Same as ConfigMap |
| Encryption | Not encrypted by default | Can be encrypted at rest in etcd |

**When to Use:**

* Use ConfigMap for environment-specific configurations (e.g., URLs, feature flags).
* Use Secret for credentials, keys, certificates.

**Note:** Base64 encoding is *not* encryption; use encryption at rest and RBAC.

**24. What are Kubernetes Admission Controllers?**

**Answer:**  
Admission Controllers are plugins that intercept requests to the Kubernetes API **after authentication and authorization but before persistence**.

**Two Types:**

* **Validating Admission Controllers**: Validate the request (e.g., resource quota).
* **Mutating Admission Controllers**: Modify the request (e.g., inject sidecar).

**Examples:**

* NamespaceLifecycle: Prevents deletion of system namespaces.
* LimitRanger: Enforces resource limits.
* MutatingAdmissionWebhook: Used by Istio, Linkerd to inject proxies.
* ValidatingAdmissionWebhook: Used for custom validations.

**Use Case:**  
Injecting a monitoring agent in all pods using a Mutating Admission Webhook.

**25. How do Kubernetes Probes (liveness, readiness, startup) work?**

**Answer:**  
**Probes** check the health and status of containers:

1. **Liveness Probe**:
   * Checks if the app is alive.
   * If it fails, the container is restarted.
2. **Readiness Probe**:
   * Checks if the app is ready to serve traffic.
   * If it fails, the pod is removed from Service endpoints.
3. **Startup Probe**:
   * Checks if the app has started correctly (for slow-starting apps).
   * Disables liveness/readiness checks until success.

**Example Liveness Probe:**

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 5

periodSeconds: 10

**Importance:**

* Prevents sending traffic to broken apps.
* Enables self-healing behavior.

**26. How does Kubernetes manage container resource limits?**

**Answer:**  
Resource limits are set using resources.requests and resources.limits in pod specs.

* **requests**: Minimum guaranteed resources.
* **limits**: Maximum allowed resources.

**Example:**

resources:

requests:

memory: "128Mi"

cpu: "500m"

limits:

memory: "256Mi"

cpu: "1"

**Impact:**

* Scheduler uses requests to place pods.
* Container exceeding limits can be throttled (CPU) or OOMKilled (memory).

**Best Practices:**

* Set both requests and limits to avoid over-provisioning or contention.
* Use LimitRange and ResourceQuota to enforce defaults and caps.

**27. What is the role of a Kubernetes DaemonSet?**

**Answer:**  
A DaemonSet ensures that a copy of a pod runs on **every node** (or a selected set of nodes) in the cluster.

**Use Cases:**

* Log collection agents (e.g., Fluentd)
* Node monitoring (e.g., Prometheus Node Exporter)
* Storage daemons (e.g., GlusterFS)

**Example:**

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: node-monitor

spec:

selector:

matchLabels:

app: monitor

template:

metadata:

labels:

app: monitor

spec:

containers:

- name: node-exporter

image: prom/node-exporter

**Key Benefits:**

* Automatic deployment on node join.
* Simplifies node-level service deployment.

**28. What is the difference between Horizontal and Vertical Pod Autoscaling?**

**Answer:**

| **Feature** | **Horizontal Pod Autoscaler (HPA)** | **Vertical Pod Autoscaler (VPA)** |
| --- | --- | --- |
| What it scales | Number of pods | CPU/memory requests and limits |
| Based on | CPU, memory, custom metrics | Historical and current resource usage |
| Common use case | Scale web apps | Optimize resource usage in batch jobs |
| Challenges | Needs metrics server | May restart pods to apply changes |

**Horizontal Example:**

kubectl autoscale deployment myapp --cpu-percent=70 --min=2 --max=5

**Vertical Example (VPA Custom Resource):**

apiVersion: autoscaling.k8s.io/v1

kind: VerticalPodAutoscaler

metadata:

name: myapp-vpa

spec:

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: myapp

updatePolicy:

updateMode: "Auto"

**29. How do you perform rolling updates and rollbacks in Kubernetes?**

**Answer:**  
**Rolling Update**:

* Ensures zero downtime by gradually replacing old pods with new ones.
* Default strategy in Deployment.

kubectl apply -f deployment.yaml

kubectl rollout status deployment myapp

**Rollback**:

* Reverts to the previous version.

kubectl rollout undo deployment myapp

**Check History:**

kubectl rollout history deployment myapp

**Custom Strategy:**

strategy:

type: RollingUpdate

rollingUpdate:

maxUnavailable: 1

maxSurge: 2

**Benefits:**

* Controlled and safe deployment.
* Easy rollback mechanism.

**30. What is Kubernetes Pod Disruption Budget (PDB)?**

**Answer:**  
A **PodDisruptionBudget** ensures that a minimum number or percentage of pods in a deployment/statefulset remain available during **voluntary disruptions**, such as:

* Node drain
* Rolling updates
* Maintenance tasks

**PDB Example:**

apiVersion: policy/v1

kind: PodDisruptionBudget

metadata:

name: myapp-pdb

spec:

minAvailable: 2

selector:

matchLabels:

app: myapp

**Key Parameters:**

* minAvailable: Minimum pods that must be running.
* maxUnavailable: Maximum pods that can be down.

**Note:**  
PDB does not apply to involuntary disruptions (e.g., hardware failure).

**31. What is the role of Kubernetes Network Policies?**

**Answer:**  
Network Policies control the communication between pods and services in a Kubernetes cluster, based on **IP addresses** or **labels**. These policies are used to enforce security rules, isolating applications and reducing the attack surface.

**Key Features:**

* Can restrict incoming and outgoing traffic to/from pods.
* Works by defining ingress (incoming) and egress (outgoing) rules.
* Can target specific pod selectors or namespaces.

**Example of a Network Policy:**

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-frontend

spec:

podSelector:

matchLabels:

app: frontend

ingress:

- from:

- podSelector:

matchLabels:

app: backend

ports:

- protocol: TCP

port: 80

**Use Cases:**

* Pod-to-pod isolation.
* Restricting external access to internal services.

**32. What are Kubernetes namespaces and how are they used?**

**Answer:**  
Namespaces are logical partitions within a Kubernetes cluster used to separate and organize resources. They allow you to create isolated environments for different applications, teams, or projects.

**Use Cases:**

* Multi-tenant clusters.
* Environments such as development, testing, production.
* Resource quotas and access controls for different teams.

**Example:**

kubectl create namespace dev

kubectl create -f pod.yaml --namespace=dev

**Access Control:**

* Use RBAC to control permissions within namespaces.

**33. What is a Kubernetes ReplicaSet and how does it differ from a Deployment?**

**Answer:**  
A ReplicaSet ensures that a specified number of pod replicas are running at all times. It is primarily used to maintain pod availability and redundancy.

**Differences with Deployment:**

* A **Deployment** is a higher-level controller that manages ReplicaSets and provides rolling updates, rollbacks, and more.
* A **ReplicaSet** is responsible for maintaining the desired pod count but does not manage deployments, upgrades, or rollbacks.

**Example ReplicaSet:**

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: nginx-replicaset

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:latest

**Recommendation:**

* Use **Deployment** for most use cases, as it automatically manages ReplicaSets for rolling updates.

**34. How does Kubernetes handle pod scheduling?**

**Answer:**  
Kubernetes uses the **Scheduler** to assign pods to nodes in the cluster based on resource availability, constraints, and other scheduling policies.

**Scheduling Steps:**

1. **Predicate Phase**: Filters nodes that don't meet pod requirements (e.g., resource requests, taints).
2. **Priority Phase**: Prioritizes nodes based on constraints like affinity and anti-affinity rules, resource usage, etc.
3. **Binding Phase**: Assigns the pod to the selected node.

**Types of Scheduling Constraints:**

* **Node Affinity**: Scheduling based on node labels.
* **Taints and Tolerations**: Restricting pod scheduling on specific nodes.
* **Pod Affinity/Anti-Affinity**: Scheduling based on other pods in the cluster.

**Example:**

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: "kubernetes.io/hostname"

operator: In

values:

- node1

**35. What is the Kubernetes API Server and its role?**

**Answer:**  
The **Kubernetes API Server** is the central control plane component that exposes the Kubernetes API and handles all requests to the cluster.

**Role:**

* Receives and validates API requests.
* Acts as the interface between clients (kubectl, controllers, etc.) and the cluster.
* Stores resource state in etcd.
* Communicates with other components (e.g., Scheduler, Controller Manager).

**Example Interaction:**

* When you run kubectl get pods, the API server validates the request and returns the list of pods.

**36. What is Kubernetes Ingress and how does it differ from a Service?**

**Answer:**  
**Ingress** is a set of rules that allow inbound connections to reach the services in a Kubernetes cluster, often managing HTTP/HTTPS traffic.

**Differences from Service:**

* **Service** provides a stable endpoint for accessing a set of pods, generally used for internal communication.
* **Ingress** manages external access, typically for HTTP/S traffic, by defining routing rules based on URL paths, hostnames, etc.

**Example of Ingress:**

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: my-ingress

spec:

rules:

- host: www.example.com

http:

paths:

- path: /foo

pathType: Prefix

backend:

service:

name: foo-service

port:

number: 80

**Benefits:**

* URL path-based routing.
* SSL termination.
* Load balancing and reverse proxy features.

**37. How does Kubernetes handle cluster upgrades?**

**Answer:**  
Kubernetes handles cluster upgrades in a **rolling** fashion to minimize downtime. The upgrade process includes:

1. **Control Plane Upgrades**:
   * Upgrade kube-apiserver, kube-controller-manager, kube-scheduler first.
   * Ensure backward compatibility with existing nodes.
2. **Node Upgrades**:
   * Upgrade nodes one at a time to avoid disrupting the entire cluster.
   * Use kubectl drain to evict pods and kubectl uncordon to bring nodes back online.

**Best Practices:**

* Test the upgrade in a non-production environment.
* Upgrade kubelet and kubectl to the matching version after upgrading the control plane.

**38. What is Kubernetes Horizontal Pod Autoscaler (HPA) and how is it configured?**

**Answer:**  
The **Horizontal Pod Autoscaler (HPA)** automatically adjusts the number of replicas of a pod based on observed metrics (e.g., CPU, memory, custom metrics).

**How it works:**

* HPA monitors the metrics (usually from the Metrics Server) and adjusts pod replicas to meet the target utilization.

**Configuration Example:**

kubectl autoscale deployment myapp --cpu-percent=50 --min=1 --max=10

**Custom Metrics:**  
To scale based on custom metrics, you need to use an adapter like the **Prometheus Adapter**.

**HPA YAML Example:**

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: myapp-hpa

spec:

minReplicas: 2

maxReplicas: 10

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 50

**39. What is Kubernetes ServiceAccount and how is it used for API access?**

**Answer:**  
A **ServiceAccount** provides an identity for pods to interact with the Kubernetes API.

**Key Features:**

* Service accounts are used for **API authentication**.
* They allow fine-grained **RBAC** policies to control access.
* A default service account is automatically created in each namespace.

**Usage Example:**

1. Create a service account:
2. kubectl create serviceaccount myapp-sa
3. Grant permissions using RBAC:
4. apiVersion: rbac.authorization.k8s.io/v1
5. kind: RoleBinding
6. metadata:
7. name: myapp-sa-binding
8. subjects:
9. - kind: ServiceAccount
10. name: myapp-sa
11. namespace: default
12. roleRef:
13. kind: Role
14. name: admin
15. apiGroup: rbac.authorization.k8s.io

**40. What is the difference between Kubernetes Deployment and StatefulSet?**

**Answer:**

| **Feature** | **Deployment** | **StatefulSet** |
| --- | --- | --- |
| Pod Identity | Pods are anonymous | Each pod has a unique identity (pod-0, pod-1) |
| Persistent Storage | Pods don't have stable storage | Each pod has its own persistent volume |
| Scaling | Easy scaling (add/remove pods) | Scaling pods is more controlled with stable identities |
| Use Case | Stateless applications | Stateful applications (e.g., databases) |

**Example Use Case for StatefulSet:**  
Stateful applications like **MySQL** and **Cassandra**, where each instance requires stable storage and network identity.

**41. What is Kubernetes Taints and Tolerations?**

**Answer:**  
**Taints and Tolerations** are used to prevent pods from being scheduled on inappropriate nodes.

* **Taint**: Applied to nodes, marking them as unsuitable for certain pods.
* **Toleration**: Applied to pods, allowing them to be scheduled on nodes with matching taints.

**Example of Taint:**

kubectl taint nodes node1 key=value:NoSchedule

**Example of Toleration in Pod:**

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

**Use Case:**

* Preventing non-critical pods from being scheduled on nodes used for critical workloads.

**42. How does Kubernetes handle high availability (HA) for control plane components?**

**Answer:**  
To achieve high availability for the Kubernetes control plane:

1. **Multiple API Servers**: Deploy multiple kube-apiserver instances behind a load balancer.
2. **Etcd Cluster**: Set up a distributed etcd cluster with an odd number of nodes (3, 5, etc.) to ensure quorum.
3. **Control Plane Pods**: Deploy multiple replicas of components like kube-scheduler and kube-controller-manager.

**Example:**

apiVersion: v1

kind: Service

metadata:

name: api-server

spec:

ports:

- port: 6443

selector:

app: kube-apiserver

**Best Practices:**

* Use an external load balancer to distribute traffic to API servers.
* Ensure proper backup and disaster recovery strategies for etcd.

**43. What are Kubernetes Affinity and Anti-Affinity rules?**

**Answer:**  
**Affinity and Anti-Affinity** rules are used to influence pod scheduling based on the presence or absence of other pods or node characteristics.

* **Pod Affinity**: Ensures pods are scheduled on nodes that already have certain pods.
* **Pod Anti-Affinity**: Ensures pods are **not** scheduled on nodes that have certain pods.

**Example of Pod Affinity:**

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchLabels:

app: myapp

topologyKey: "kubernetes.io/hostname"

**Example of Pod Anti-Affinity:**

affinity:

podAntiAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchLabels:

app: myapp

topologyKey: "kubernetes.io/hostname"

**44. What is the purpose of kubelet in Kubernetes?**

**Answer:**  
The **kubelet** is an agent that runs on each node in the cluster and ensures the containers in the pods are running as expected.

**Responsibilities:**

* Watches the API server for pod specifications and ensures the desired state is met on the node.
* Monitors containers and restarts them if necessary (e.g., if they crash).
* Reports node health to the API server.

**Example:**

kubectl get nodes

**Key Points:**

* Runs as a background process on each node.
* Communicates with the kube-apiserver.

**45. What is the Kubernetes Scheduler?**

**Answer:**  
The **Kubernetes Scheduler** is a control plane component that watches for newly created pods without nodes and assigns them to an appropriate node.

**Scheduling Process:**

1. **Predicate Phase**: Filters nodes based on requirements (e.g., resource requests, taints).
2. **Priority Phase**: Assigns a score to nodes, considering factors like affinity, resource utilization.
3. **Binding Phase**: Assigns the pod to a specific node.

**Example:**  
The scheduler will place a pod on a node with sufficient CPU and memory resources, without violating any node affinity or anti-affinity rules.

**46. What is a Kubernetes Service and how does it work?**

**Answer:**  
A **Service** in Kubernetes is an abstraction that defines a set of pods and provides a stable endpoint (IP or DNS) for accessing them.

**Types of Services:**

* **ClusterIP**: Default type, only accessible within the cluster.
* **NodePort**: Exposes the service on a static port across all nodes.
* **LoadBalancer**: Exposes the service externally using a cloud load balancer (if supported).
* **ExternalName**: Maps the service to an external DNS name.

**Example:**

apiVersion: v1

kind: Service

metadata:

name: my-service

spec:

selector:

app: myapp

ports:

- protocol: TCP

port: 80

targetPort: 8080

**Use Cases:**

* Load balancing across multiple pods.
* External exposure of internal services.

**47. What is Kubernetes' kubectl and what are some useful commands?**

**Answer:**  
kubectl is the command-line tool for interacting with the Kubernetes cluster. It communicates with the Kubernetes API server to perform actions on the cluster.

**Common Commands:**

* **Get Resources**: kubectl get pods, kubectl get services
* **Create Resources**: kubectl apply -f <file>.yaml
* **Describe Resources**: kubectl describe pod <pod-name>
* **Delete Resources**: kubectl delete pod <pod-name>
* **Logs**: kubectl logs <pod-name>
* **Exec**: kubectl exec -it <pod-name> -- /bin/bash

**Useful Flags:**

* -n <namespace>: Specify the namespace.
* -o wide: Show extra details.

**48. What are Kubernetes Persistent Volumes (PV) and Persistent Volume Claims (PVC)?**

**Answer:**  
**Persistent Volumes (PVs)** are resources in the cluster that provide durable storage, independent of the lifecycle of individual pods.

* A **Persistent Volume Claim (PVC)** is a request for storage, specifying size and access modes.

**Workflow:**

1. Define a PV (storage resource).
2. Create a PVC (a pod requesting storage).
3. Kubernetes binds the PVC to an appropriate PV.

**Example:**

apiVersion: v1

kind: PersistentVolume

metadata:

name: my-pv

spec:

capacity:

storage: 10Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Retain

hostPath:

path: "/mnt/data"

**Example of PVC:**

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-pvc

spec:

resources:

requests:

storage: 5Gi

**49. What are Kubernetes Endpoints and how are they related to Services?**

**Answer:**  
**Endpoints** are objects that represent the IP addresses of the pods that back a Kubernetes Service.

* When a Service is created, Kubernetes automatically creates Endpoints objects that match the pods selected by the Service's label selector.

**How They Work:**

* The Service proxy forwards requests to the endpoints associated with the service.
* When a pod is added or removed, the list of endpoints is updated.

**Example:**

kubectl get endpoints my-service

**50. What is the purpose of kubectl drain and kubectl cordon?**

**Answer:**  
These commands are used for **node maintenance**.

* **kubectl cordon**: Marks a node as unschedulable, preventing new pods from being scheduled on it.
* **kubectl drain**: Evicts all pods from the node, marking the node as unschedulable and preparing it for maintenance.

**Example:**

kubectl cordon node1

kubectl drain node1 --ignore-daemonsets

**Use Cases:**

* Draining a node before maintenance (e.g., upgrading hardware or OS).
* Temporarily isolating a node from the cluster.