**Kubernetes Cluster Scoped Resources**

A Kubernetes Cluster is the set of machines (nodes) that run your containerized applications.

It consists of:

* Control Plane (Master components) → API server, Scheduler, Controller Manager, etc. (manages the cluster).
* Worker Nodes → Run the actual application workloads (Pods). Together, they form a **single unit** where Kubernetes schedules, runs, and manages containers.

**What is Cluster-scoped**: A cluster-scoped resource exists at the cluster level and is not tied to any namespace. They apply to the entire cluster.

Example: **Nodes** → We don’t put a Node inside a namespace; it’s global to the cluster.

**Core Cluster Scoped Resources:**

**1.Namespaces:**

**What is a Namespace:** Namespaces in Kubernetes provide a mechanism for logically isolating resources within a single cluster. They function as virtual sub-clusters, allowing multiple teams, projects, or applications to share a physical Kubernetes cluster while maintaining separation and preventing naming conflicts.

**Why Namespaces?** :

* They help to organize resources in large clusters.
* They provide multitenancy, They are particularly useful in multi-tenant environments where different users or teams share a cluster. Namespaces enable each tenant to operate as if they have their own dedicated cluster, without interfering with others.
* Allow resource quotas per namespace.
* Simplify access control using RBAC(Role Based Access Control).

**Default Namespaces in Kubernetes:**

* default → Where resources go if no namespace is specified.
* kube-system → Critical Kubernetes system components (kube-dns, etc.).
* kube-public → Publicly readable data (like cluster info).
* kube-node-lease → Stores node heartbeat leases for faster failure detection.

**Some more Points:**

* Namespaces are cluster-scoped (you can’t create a namespace inside another).
* Resources like Pods, Services, ConfigMaps are namespace-scoped → they live inside namespaces.

**Namespaces commands:**

kubectl get namespaces : List namespaces.

kubectl create namespace adi: Creates a namespace.

**2.Nodes:**

**What is a Node?**

* A Node is a worker machine in a Kubernetes cluster.
* It can be a physical server or a virtual machine.
* Nodes are where your Pods (containers) actually run.
* There are two primary types of nodes in a Kubernetes cluster: Master Node and a Worker Node.
* 1. Master Node: These nodes host the core components of the Kubernetes control plane, which manages and controls the entire cluster.
* Key components on a control plane node include:Api server, etcd, controller manager and scheduler.
* 2.Worker Node: These nodes are where the actual containerized workloads (pods) are executed.
* Each worker node runs essential components that allow it to interact with the control plane and run containers: Kubelet, Kube Proxy, Container Runtime.

**2. Why is it used?**

Nodes are fundamental components in Kubernetes clusters because they provide the actual computing resources where containerized applications (encapsulated within Pods) are executed. Without nodes, a Kubernetes cluster would lack the infrastructure to run any workloads.

Kubernetes schedules Pods on Nodes depending on available resources.

Each Node runs critical components:

* **kubelet** → Communicates with the control plane, runs Pods.
* **kube-proxy** → Handles networking for Pods.
* **Container runtime** (Docker, containerd, etc.) → Runs the containers.

### **Key Points**

* Nodes are **cluster-scoped** (not tied to any namespace).
* The **control plane** automatically manages them.
* Nodes can be configured to control the pod scheduling like if a node fails, Kubernetes reschedules Pods on healthy Nodes.

**Commands:**

kubectl get nodes: Lists all nodes in a cluster.

kubectl describe node <node-name>: Detailed info about a node.

kubectl top nodes: Show resource usage (CPU/Memory) of nodes.

**3.PersistentVolumes (PVs):**

**What is a PersistentVolume (PV):**

A PersistentVolume (PV) is a piece of storage in the cluster. It’s provisioned by an admin (or dynamically via a StorageClass). Unlike a Pod’s temporary storage, PVs are independent of Pods and exist at the cluster level. Unlike a Pod’s temporary storage, PVs are independent of Pods and exist at the cluster level. It is a cluster-level resource, meaning its lifecycle is independent of any individual Pod that uses it. This ensures that data stored in a PV persists even if the Pod using it is deleted or recreated.

**Why it is Used:**

It is used to provide a **durable storage** for applications (so data isn’t lost if Pods restart or move).

Decouples storage from Pods:

* Pods use PersistentVolumeClaims (PVCs) to request storage.
* The PVC is then bound to a PV. A PersistentVolumeClaim (PVC) is a request for storage made by a user or application.

**Some Key Points:** PVs are cluster-scoped (not tied to namespaces). A PV can be backed by many storage types: NFS, AWS EBS, GCP PD, Azure Disk, local disk, etc. and the lifecycle is independent of Pods.

Access Modes:

* ReadWriteOnce → One node can read/write.
* ReadOnlyMany → Multiple nodes can read.
* ReadWriteMany → Multiple nodes can read/write.

**Commands:**

**kubectl get pv**: List all the PVs(Persistent Volumes).

**kubectl describe pv <pv-name>**: Describe a PV(Persistent Volume).

**kubectl delete pv <pv-name>**: Delete a PV(Persistent Volume).

**4.Storage Class:**

**What is a StorageClass:**

* A StorageClass defines the “blueprint” or template for how storage is provisioned in a cluster.
* It tells Kubernetes what type of storage to create (like SSD, HDD, cloud disk, etc.).
* When a PVC requests storage and doesn’t specify a PV, Kubernetes uses the StorageClass to dynamically create a PV.

StorageClasses enable the dynamic provisioning of PersistentVolumes (PVs). Instead of manually creating PVs, users can create PersistentVolumeClaims (PVCs) that reference a specific StorageClass. Kubernetes then automatically provisions a PV and the underlying storage resource based on the parameters defined in that StorageClass.

**Why is it used:**

* To enable dynamic provisioning of storage (on-demand).
* Developers don’t need to wait for admins to manually create PVs.
* Makes storage scalable and flexible.

Instead of manually creating Persistent Volumes for every storage request, StorageClasses allow Kubernetes to automatically provision the underlying storage and create the corresponding PV when a Persistent Volume Claim (PVC) is made.

A StorageClass allows cluster administrators to define various attributes and characteristics of different storage types, such as:

* Performance: e.g., SSD vs. HDD, IOPS limits.
* Quality of Service (QoS): Different tiers of performance or availability.
* Replication/Backup Policies: How data is protected or backed up.
* File System Type: e.g., ext4, xfs.
* Cost: Mapping to cheaper or more expensive storage options.

**Commands:**

**kubectl get storageclass:** List all storage classes.

**kubectl describe storageclass <sc-name>**: Describe a storage class.

**5.ClusterRole:**

### **What is a ClusterRole:**

* ClusterRole is a set of permissions (rules) in Kubernetes.
* Unlike a Role (which applies inside a namespace), a ClusterRole applies to the entire cluster.
* Defines what actions (verbs like get, list, create, delete) can be performed on which resources.

A cluster role is a set of RBAC (Role-Based Access Control) permissions in a Kubernetes environment that grants access to cluster-wide resources or resources across all namespaces, rather than being limited to a single namespace or throughout the whole cluster.

**Why is it used:**

To control access at the cluster level.

Useful for:

* Cluster-wide operations (e.g., list all nodes).
* Access to cluster-scoped resources (like Nodes, PVs).
* Reusable permissions across multiple namespaces.

ClusterRoles don’t grant permissions by themselves; they must be bound to users/groups/service accounts using ClusterRoleBindings (or RoleBindings).

**Commands:**

**kubectl get clusterrole:** List all ClusterRoles.

**kubectl describe clusterrole <name>:** Describe a specific ClusterRole.

**kubectl delete clusterrole <name>:** Delete a ClusterRole.

**6.ClusterRoleBinding:**

**What is a ClusterRoleBinding:**

A ClusterRoleBinding connects a ClusterRole to a user, group, or service account. This is how RBAC(Role-Based Access Control) permissions are applied across the whole cluster.

How RBAC Works: **Role / ClusterRole** → Defines a set of permissions (what actions are allowed).

* Role → Works inside a namespace.
* ClusterRole → Works across the entire cluster.

RoleBinding / ClusterRoleBinding → Connects a Role (or ClusterRole) to a user/group/service account.(like specific namespace or specific user)

* RoleBinding → Grants access in one namespace.
* ClusterRoleBinding → Grants access at the cluster level.

**Example**: If we want to only view Pods in the dev namespace. We create a Role with “get, list, watch” permissions for Pods. Then, create a RoleBinding attaching that Role to the developer’s account.

**Why it is used:**

* To grant cluster-wide permissions defined in a ClusterRole.
* Without a binding, a ClusterRole is just a definition and doesn’t give access to anyone.
* It ensures the right users/services have the right level of access.

**RoleBinding vs ClusterRoleBinding**:

* RoleBinding → Grants Role or ClusterRole permissions inside one namespace.
* ClusterRoleBinding → Grants ClusterRole permissions across the entire cluster.

**Commands:**

kubectl get clusterrolebinding:List all ClusterRoleBindings.

kubectl describe clusterrolebinding <name>: Describe a specific ClusterRoleBinding.

kubectl delete clusterrolebinding <name>: Delete a ClusterRoleBinding.

**Namespace Scoped Resources:**

**1.Pods:**

**What is a Pod:**

A Pod is the smallest deployable unit in Kubernetes. It represents one or more containers that run together on a Node.

Containers in a Pod share:

* **Networking** (same IP address, same port space).
* **Storage volumes** (if defined).

Pods within one namespace are logically separated from Pods in other namespaces. They cannot directly see or interact with resources in different namespaces unless specifically configured to do so, it provides isolation.

**Why it is used:**

Pods are used to run applications (containers) inside Kubernetes. Pods wrap around containers and give them:

* A network identity (IP).
* Storage volumes (via PV/PVC).
* Lifecycle management (restart, reschedule).

While containers within a pod share resources, pods themselves provide a level of isolation from other pods. Each pod is assigned its own unique IP address, enabling independent networking and resource allocation within the cluster.

Pods are ephemeral i.e if a Pod dies, Kubernetes can replace it (via ReplicaSet/Deployment) and Pods are always created inside a namespace.

**Commands**: kubectl get pods -n <namespace>: List Pods in a namespace.

kubectl describe pod <pod-name> -n <namespace>:Detailed info about a Pod.

kubectl delete pod <pod-name> -n <namespace>:Delete a Pod.

kubectl logs <pod-name> -n <namespace>:Get Pod logs.

kubectl exec -it <pod-name> -n <namespace> -- /bin/bash: Exec into a container inside a Pod.

**2.ReplicaSet:**

**What is a ReplicaSet:**

A ReplicaSet in k8s ensures that a specified number of identical Pods are always running. If a Pod fails or is deleted, the ReplicaSet automatically creates a new one. It acts like a self-healing mechanism for Pods.

**Why is it used:**

It is used to guarantee the availability and scalability of applications using pods. It also ensures workloads always have the desired number of replicas running. It also provides automatic replacement if Pods crash or nodes fail.

* ReplicaSets are namespace scoped.
* It is typically not used directly → Instead, we use Deployments, which manage ReplicaSets for you.
* Matches pods using labels (selector - matchLabels).
* Allows manual or automatic scaling.

**Commands:**

kubectl get rs -n <namespace>: List ReplicaSets in a namespace.

kubectl describe rs <rs-name> -n <namespace>: It describes a specific ReplicaSet.

kubectl scale rs <rs-name> --replicas=5 -n <namespace>: Scale a ReplicaSet to 5 replicas.

kubectl delete rs <rs-name> -n <namespace>: Delete a ReplicaSet.

**3.Deployments:**

**What is Deployments:**

A Deployment is a higher-level controller that manages ReplicaSets and Pods.It provides declarative updates for applications, meaning you just define the desired state, and Kubernetes will make it happen.Handles rolling updates, rollbacks, and scaling for developers.

**Why is it used?:**

* To easily manage applications with multiple replicas.
* Enables zero-downtime rolling updates of apps.
* Provides a way to rollback to a previous version if something goes wrong.
* Simplifies app management compared to managing ReplicaSets directly.

It is namespace scoped.Manages **ReplicaSets**, which in turn manage **Pods**.

It Supports:

* Rolling Updates → Gradual replacement of old Pods with new ones.
* Rollbacks → Revert to a previous working version.
* Scaling → Increase/decrease replicas easily.

**Commands:**

kubectl get deployments -n <namespace>:List Deployments in a namespace.

kubectl describe deployment <deployment-name> -n <namespace>: Describe a specific Deployment.

kubectl scale deployment <deployment-name> --replicas=5 -n <namespace>: Scale a Deployment to 5 replicas.

kubectl set image deployment/<deployment-name> <container-name>=nginx:latest -n <namespace>: Trigger a rolling update (by updating the image).

kubectl rollout undo deployment/<deployment-name> -n <namespace>: Rollback to previous version.

kubectl rollout status deployment/<deployment-name> -n <namespace>: Check rollout status.

**4.Service:**

**What is a Service:**

A Service exposes Pods to be reachable inside or outside the cluster. It provides a stable IP address and DNS name for Pods, even if Pods are restarted or rescheduled.

**Why is it used:**

Pods are ephemeral → their IPs can change so services ensures consistent access to Pods. Supports load balancing across multiple Pods.

So they are Namespace-scoped.

Types of Services:

* **ClusterIP** → default, accessible only within cluster.
* **NodePort** → accessible outside via Node’s IP and port.
* **LoadBalancer** → exposes Service externally via cloud LB.
* **ExternalName** → maps Service to external DNS name.

Commands:

kubectl get svc -n <namespace>

kubectl describe svc <service-name> -n <namespace>

kubectl delete svc <service-name> -n <namespace>

**5.Ingress:**

**What is an Ingress:**

An Ingress exposes HTTP and HTTPS routes from outside the cluster to Services inside. It allows URL-based routing and TLS termination.

**Why is it used:**

To manage external access to multiple services via a single IP.  
Supports path-based routing and host-based routing.

Requires an Ingress Controller (e.g., NGINX Ingress Controller) and supports features like SSL/TLS, load balancing, and rewrites.

**Commands:** kubectl get ingress -n <namespace>

kubectl describe ingress <ingress-name> -n <namespace>

kubectl delete ingress <ingress-name> -n <namespace>

**6. ConfigMap:**

**What is ConfigMap:**

Stores non-sensitive configuration data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line args, or files.

**Why it is Used:**

To decouple configuration from container images. Allows updating app configuration without rebuilding images. It can store multiple key-value pairs. It is often used with Deployments or Pods to inject configuration.

**Commands:** kubectl get configmap -n <namespace>

kubectl describe configmap <configmap-name> -n <namespace>

kubectl delete configmap <configmap-name> -n <namespace>

**7. Secrets:**

**What is Secrets:**

Stores **sensitive data** like passwords, tokens, and keys and the data is stored in **base64-encoded format**.

**Why it is Used:**

To protect sensitive info in Kubernetes. Pods can access Secrets via environment variables or volumes.

* Can be of type: Opaque, docker-registry, TLS, etc.
* Provides better security than hardcoding credentials in Pods.

**Commands:**

kubectl get secret -n <namespace>

kubectl describe secret <secret-name> -n <namespace>

kubectl delete secret <secret-name> -n <namespace>