**Cluster scoped Vs Namespace scoped**

**Namespace-Scoped Resources**

These are resources that live inside a specific namespace. A namespace is like a virtual cluster within your main Kubernetes cluster, used to partition resources. When you create a resource like a Pod, you must place it within a namespace (if you don't specify one, it goes into the default namespace). Names of these resources only need to be unique *within* their namespace.

**Cluster-Scoped Resources**

These are singular resources that have a global scope across the entire Kubernetes cluster. They are not tied to any single namespace. There can only be one resource with a given name in the whole cluster.

These resources are for managing the fundamental fabric and infrastructure of the cluster itself. They provide the overarching structure and capabilities that all namespaced resources rely on. They are not specific to any one application or team but are central to the cluster's operation.

**What is the Importance of Scoping?**

Scoping isn't just an arbitrary design choice; it's fundamental to how Kubernetes provides a robust, multi-tenant platform.

* **Isolation & Organization:** The primary goal is to isolate resources. Different teams, projects, or environments (e.g., development, staging, production) can operate on the same physical cluster without interfering with each other's applications, networking, or configurations.
* **Avoiding Naming Conflicts:** Team A can create a Deployment named my-app in the team-a namespace, and Team B can do the same in the team-b namespace without any conflict.
* **Access Control:** Kubernetes' Role-Based Access Control (RBAC) relies heavily on scope. You can grant a user permissions to manage Pods and Services **only within** the project-alpha namespace, preventing them from even seeing resources in other namespaces.
* **Resource Management:** You can set **ResourceQuotas** on a namespace to limit the total amount of CPU, memory, or storage that all resources within that namespace can consume. This is crucial for fairly sharing cluster resources.

**What is the Difference?**

| Feature | Namespace-Scoped Resources | Cluster-Scoped Resources |
| --- | --- | --- |
| **Scope** | Confined to one namespace. | Applies to the whole cluster. |
| **Purpose** | To run and manage specific applications. | To manage and configure the cluster's infrastructure. |
| **Naming** | Must be unique *within its namespace*. | Must be unique *across the entire cluster*. |
| **API Access** | API path includes the namespace: /api/v1/namespaces/{namespace}/pods | API path does not include a namespace: /api/v1/nodes |
| **Permissions** | Managed by **Roles** and **RoleBindings**. | Managed by **ClusterRoles** and **ClusterRoleBindings**. |

**Where to Use What?**

**Use Namespace-Scoped resources for everything related to a specific application.** This includes your application's containers (Pods), how it's scaled and updated (Deployments), its network endpoints (Services), its configuration (ConfigMaps), and its storage requests (PVCs). If you are a developer, 95% of the resources you interact with will be namespace-scoped.

**Use Cluster-Scoped resources only when you are defining the cluster's underlying fabric and global policies.** This is typically done by a cluster administrator or platform engineer. This includes adding new machines to the cluster (Nodes), defining types of available storage (StorageClasses), or setting up cluster-wide permissions (ClusterRoles).

**Namespace-Scoped Resources**

* **Pods:**
  + **What it is:** The most basic and smallest deployable unit in Kubernetes. It represents a single instance of a running process in your cluster and contains one or more containers (like Docker containers).
  + **Why it's namespaced:** Pods are the core building blocks of your application. They must be namespaced to keep one application's running processes isolated from another's.

# List all pods in the current namespace

kubectl get pods

# List all pods in a specific namespace

kubectl get pods -n <namespace-name>

# Get detailed information about a pod

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

# Delete a pod (it will likely be recreated by a controller like a Deployment)

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

# View logs from a pod

kubectl logs <pod-name> -n <namespace-name>

* **Deployments / ReplicaSets:**
  + **What it is:** A Deployment is a controller that manages the lifecycle of your Pods. It ensures a specified number of Pods (replicas) are running at all times and handles updates and rollbacks. A ReplicaSet is the underlying object that ensures the pod count.
  + **Why it's namespaced:** Deployments define the desired state for a *specific application*. The deployment for your user-facing API should be in a different namespace than the deployment for your internal data-processing tool.

# List all deployments in a namespace

kubectl get deployments -n <namespace-name>

# Get detailed information about a deployment

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

# Scale a deployment to a new number of replicas

kubectl scale deployment <deployment-name> --replicas=5 -n <namespace-name>

# Delete a deployment (this will also delete its pods)

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

* **Services:**
  + **What it is:** A Service provides a stable, single network endpoint (a consistent IP address and DNS name) to access a group of Pods. Since Pods can be created and destroyed, their IP addresses change, but the Service IP remains constant.
  + **Why it's namespaced:** It provides a network entry point for an application *within* a namespace. A Service named database in the prod namespace will correctly point to the production database Pods, not the ones in the dev namespace.

# List all services in a namespace

kubectl get services -n <namespace-name>

# Get detailed information about a service

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

# Delete a service

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

* **ConfigMaps & Secrets:**
  + **What they are:** ConfigMaps store non-sensitive configuration data (like environment variables or config files) as key-value pairs. Secrets do the same but are intended for sensitive data like passwords, API keys, or TLS certificates.
  + **Why they are namespaced:** Configuration and secrets are almost always specific to a single application. Isolating them by namespace is a critical security and organizational practice.

# List all configmaps in a namespace

kubectl get configmaps -n <namespace-name>

# List all secrets in a namespace

kubectl get secrets -n <namespace-name>

# Describe a secret (note: the data will be base64 encoded)

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

# Delete a configmap

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

* **PersistentVolumeClaims (PVCs):**
  + **What it is:** A request for storage made by an application. A developer creates a PVC to ask for a certain amount of storage (e.g., "I need 10 GiB of fast storage").
  + **Why it's namespaced:** It represents a specific application's *claim* on a piece of the cluster's storage pool.

# List all PersistentVolumeClaims in a namespace

kubectl get pvc -n <namespace-name>

# Get detailed information about a PVC

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

# Delete a PVC

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

**HorizontalPodAutoscaler (HPA)**

* **What it is:** An HPA automatically scales the number of Pods in a Deployment, ReplicaSet, or StatefulSet based on observed metrics like CPU utilization or custom metrics.
* **Why it's Namespace-Scoped:** An HPA's job is to manage the scale of a *specific application*. The scaling rules for the "user-api" in the production namespace (e.g., "scale up when CPU > 80%") are completely independent of the scaling rules for the "billing-service" in the finance namespace. Each application needs its own autoscaler defined within its own boundary.

# List all HPAs in a specific namespace

kubectl get hpa -n <namespace-name>

# View the detailed status and metrics of an HPA

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

# Create a new HPA for a deployment

kubectl autoscale deployment <deployment-name> --cpu-percent=80 --min=1 --max=10 -n <namespace-name>

**Role and RoleBinding**

* **What they are:** These are the namespaced versions of ClusterRole and ClusterRoleBinding. A **Role** defines permissions (e.g., "can get, list, and delete Pods"), and a **RoleBinding** grants those permissions to a user or group.
* **Why they're Namespace-Scoped:** This is the heart of multi-tenancy. You use a Role and RoleBinding to grant a developer permissions to manage resources *only within their team's specific namespace*. It prevents them from seeing or affecting any other team's applications, providing a secure, isolated boundary for their work.

# List all Roles in a namespace

kubectl get roles -n <namespace-name>

# List all RoleBindings in a namespace

kubectl get rolebindings -n <namespace-name>

# Describe a Role to see exactly what permissions it grants

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

# Describe a RoleBinding to see who has the permissions

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

**ServiceAccount**

* **What it is:** A ServiceAccount provides an identity for processes that run inside a Pod. This identity can then be granted permissions using Roles or ClusterRoles. It's essentially an "account" for a robot or application, rather than a human user.
* **Why it's Namespace-Scoped:** Each application should have its own unique identity for security reasons (the principle of least privilege). A ServiceAccount created in the monitoring namespace for Prometheus should only have the permissions it needs and is completely separate from the ServiceAccount used by a CI/CD pipeline in the dev-tools namespace.

# List all ServiceAccounts in a namespace

kubectl get serviceaccounts -n <namespace-name>

# Get detailed information about a ServiceAccount, including its tokens

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

# Create a new ServiceAccount

kubectl create serviceaccount <new-account-name> -n <namespace-name>

**Detailed Cluster-Scoped Resources**

* **Nodes:**
  + **What it is:** A Node is a worker machine in the cluster, which can be either a virtual machine or a physical server. Nodes are the machines where your Pods actually run.
  + **Why it's cluster-scoped:** Nodes are the fundamental compute capacity of the entire cluster. They form a global pool of resources available to all namespaces.

# List all nodes in the cluster

kubectl get nodes

# Get detailed information about a specific node

kubectl describe node <node-name>

# View node resource usage (requires metrics-server)

kubectl top node <node-name>

* **Namespaces:**
  + **What it is:** The resource used to create the virtual partitions themselves.
  + **Why it's cluster-scoped:** The partitioning mechanism itself must be global. You cannot create a namespace inside another namespace.

# List all namespaces

kubectl get namespaces

# Create a new namespace

kubectl create namespace <new-namespace-name>

# Get detailed information about a namespace

kubectl describe namespace <namespace-name>

# Delete a namespace (and ALL resources within it)

kubectl delete namespace <namespace-name>

* **PersistentVolumes (PVs):**
  + **What it is:** A PersistentVolume is a piece of storage in the cluster that has been provisioned by an administrator. It is a resource in the cluster, just like a Node is a resource.
  + **Why it's cluster-scoped:** PVs represent the *available pool of storage* for the entire cluster. A namespaced PVC then "claims" a PV from this global pool for its own use.

# List all PersistentVolumes

kubectl get pv

# Get detailed information about a specific PV

kubectl describe pv <pv-name>

# Delete a PV

kubectl delete pv <pv-name>

* **StorageClasses:**
  + **What it is:** A StorageClass provides a way for administrators to define different "classes" or "types" of storage they offer. For example, you might have classes named standard-hdd (slow, cheap) and premium-ssd (fast, expensive).
  + **Why it's cluster-scoped:** These storage definitions are available globally for any PVC in any namespace to request.

# List all available StorageClasses

kubectl get sc

# Get detailed information about a StorageClass

kubectl describe sc <storageclass-name>

* **ClusterRoles & ClusterRoleBindings:**
  + **What they are:** The cluster-level equivalent of Roles and RoleBindings. They are used to grant permissions in the RBAC system.
  + **Why they are cluster-scoped:** Their purpose is to grant rights to cluster-scoped resources (like Nodes) or to grant rights to namespaced resources *across all namespaces* (e.g., allowing a cluster admin to view Pods in *every* namespace).

# List all ClusterRoles

kubectl get clusterroles

# List all ClusterRoleBindings

kubectl get clusterrolebindings

# Describe a specific ClusterRole to see its permissions

kubectl describe clusterrole <clusterrole-name>

**CustomResourceDefinition (CRD)**

* **What it is:** A CRD is a powerful feature that lets you extend the Kubernetes API by adding your own custom object types. For example, a database operator might create a PostgresqlDB resource.
* **Why it's Cluster-Scoped:** When you add a new resource type to Kubernetes, you are fundamentally extending the API for the *entire cluster*. The definition of what a PostgresqlDB object is must be singular and globally understood so that all namespaces can create instances of that object. While the definition (CRD) is cluster-scoped, the custom objects you create using it (the PostgresqlDB instances) are typically **namespace-scoped**

# List all CustomResourceDefinitions in the cluster

kubectl get crd

# Describe a CRD to see its schema and definition

kubectl describe crd <crd-name.group>

**RuntimeClass**

* **What it is:** A RuntimeClass is a feature for selecting the container runtime configuration. This is used in clusters that have multiple types of runtimes, like the standard runc and a more sandboxed or high-performance runtime like gVisor or Kata Containers.
* **Why it's Cluster-Scoped:** The available container runtimes are a property of the cluster's underlying **Nodes**. They are a global, low-level capability that the cluster offers to all namespaces. Defining this at the cluster level creates a catalog of available runtimes that any Pod in any namespace can request.

# List all available RuntimeClasses

kubectl get runtimeclasses

# Describe a RuntimeClass to see its configuration

kubectl describe runtimeclass <runtimeclass-name>

**PriorityClass**

* **What it is:** A PriorityClass is an object that defines a priority level (e.g., "high-priority", "low-priority"). You can then assign this class to your Pods.
* **Why it's Cluster-Scoped:** Pod priority is a cluster-wide concept. When the cluster is running out of resources, the **Scheduler** (a cluster-level component) needs a global, consistent set of rules to decide which Pods are more important to keep running and which can be evicted (removed). This priority system must be the same across all namespaces to ensure fairness and prevent a low-priority Pod in one namespace from evicting a high-priority Pod in another.

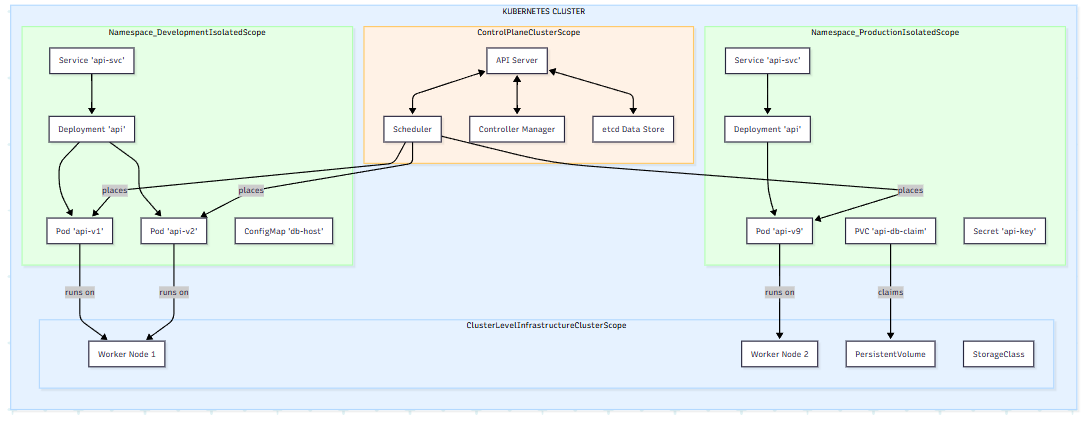
# List all PriorityClasses in the cluster

kubectl get priorityclasses

# View the details of a specific PriorityClass

kubectl describe priorityclass <priorityclass-name>

**Architecture –**

****