k8s-resources

What Are Kubernetes Resources?

In Kubernetes, **resources** are the building blocks used to deploy, manage, and scale applications. These include objects like **pods, services, deployments, config maps, secrets**, and more.

Each resource serves a specific role in defining the **desired state** of your application and the **infrastructure logic** needed to support it.

1. Namespace

A **Namespace** is a way to **divide a Kubernetes cluster into isolated environments**. Think of it as a separate workspace where you can group related resources like pods, services, and deployments.

- Isolated Project where you can create resources to related to your project.

✅ **Why we use Namespaces**:

- To **separate teams or projects** within the same cluster
- To apply **different access controls, quotas, or policies**
- To avoid **name conflicts** across environments (e.g., `dev`, `test`, `prod`)

📌 **Best Practices**:

- Use namespaces for logical separation of concerns
- Apply **resource limits and RBAC rules** per namespace
- Avoid using the default namespace for production workloads

example manifest file to create namespace

```
Kind: Namespace
apiVersion: v1
metadata:
    name: project
    labels:
        name: project
        environment: dev
```

kubectl apply -f <file.yml> # to create k8s resources kubectl delete -f <file.yml> # to delete k8s resources

2. Pod

A **Pod** is the **smallest deployable unit** in Kubernetes. It represents a single instance of a running process in a cluster.

✅ Key Features:

- A pod typically runs **one container**, but it can run multiple if needed
- All containers in a pod share the **same network IP and storage**
- Pods are **short-lived** and usually managed by controllers like Deployments

example manifest file to create pod

```
Kind: pod
apiVersion: v1
metadata:
    name: nginx
spec:
    containers:
    - name: nginx
      image: nginx
```

kubectl apply -f <pod.yml>

kubectl describe pod <pod-name> # to see pod info

3. Multi-Container Pod

A **multi-container pod** runs two or more containers in the same pod.These containers work together and can communicate over `localhost` .

✅ Why We Use Multi-Container Pods

- To create **sidecar containers** for logging, monitoring, or proxying
- To run **helper processes** alongside the main application
- To **share files and memory** via shared volumes

- To use **init containers** for startup tasks like waiting for dependencies, setting up config files, or initializing volumes before the main container starts
- ⚙️ **Workload separation** allows auxiliary tasks—such as log forwarding, data synchronization, or certificate renewal—to run in separate containers, keeping the **main container focused on its core responsibilities** (e.g., handling API requests)

🔹 *Example*: A sidecar like **Fluentd** can handle log shipping, offloading that task from the backend container, which improves performance and maintains separation of concerns.

Sidecar Container

In Kubernetes, a **sidecar container** runs alongside the main application container within the same pod to handle supporting tasks.

For example, rather than having the backend container push logs—which adds unnecessary load—we deploy a sidecar like **Fluentd** to collect and forward logs.

Fluentd then sends logs to **Elasticsearch**, our centralized logging solution. Since Elasticsearch is managed via **AWS**, we benefit from scalable and reliable log storage without burdening application containers.

This approach enhances **performance**, improves **observability**, and supports **separation of concerns** within our microservices architecture.

Init Container

An **init container** in Kubernetes runs **before** the main application container starts. It performs **preparation tasks** that must succeed before the app begins execution.

For instance, if the application depends on a service like a database, the init container can run a script to **wait for the database to become available** or to **initialize configuration files or shared volumes**.

By handling such setup logic outside the main container, we:

- Improve **startup reliability**
- Keep application code simpler
- Enforce a consistent **startup sequence**

📌 **Best Practices**:

- Use multi-container pods **only when containers must work together**
- Use **init containers** for setup tasks before main containers start

- Design containers to be loosely coupled and independently replaceable when possible

03.multi-containers.yml

```
Kind: pod
apiVersion: v1
metadata:
    name: multi-container
spec:
    containers:
    - name: nginx
      image: nginx
    - name: alma
      image: "almalinux:9"
      command: ["sleep", "2000"]
```

kubectl apply -f multi-containers.yml

kubectl exec -it <pod-name> -c <container-name> -- bash # Access a specific container in a pod

4. Labels

Labels are **key-value pairs** used to **organize and identify** Kubernetes resources.

> Labels are mandatory in Kubernetes as they help services identify which pods they belong to and which ones they should target.

✅ **Why we use Labels**:

- To group resources logically (e.g., `app=frontend` , `env=prod`)
- To **select resources** for services, deployments, and policies
- To filter and manage resources using `kubectl` or tools like Helm

📌 **Best Practices**:

- Define consistent label naming conventions (e.g., `app` , `tier` , `version`)
- Use labels to enable **rolling updates**, monitoring, and scaling
- Avoid putting sensitive or unique data in labels

sample manifest files, uploaded in below github repo https://github.com/xaravind/k8s-resources.git

5. Annotations

Annotations are also **key-value metadata**, but they store **non-identifying information** about Kubernetes objects.

✅ **Why we use Annotations**:

- To attach **descriptive or operational data** (e.g., build info, monitoring configs)
- To support **external tools and integrations** (e.g., Ingress controllers, CI/CD)
- To provide context or configuration **without affecting selection or grouping**

📌 **Best Practices**:

- Use annotations for **metadata that shouldn't affect behavior**
- Avoid storing sensitive data (use Secrets instead)
- Use standardized annotation keys where possible (e.g., `kubectl.kubernetes.io/last-applied-configuration`)

sample manifest files, uploaded in below github repo https://github.com/xaravind/k8s-resources.git

6. Environment Variables

Environment variables are used to **inject configuration values** into containers at runtime.

✅ **Why we use Environment Variables**:

- To configure applications **without modifying code or images**
- To provide settings like **database URLs, feature flags, or API keys**
- To make containers **portable across environments**

📌 **Best Practices**:

- Use them for simple values (e.g., strings, paths, ports)
- Store values in **ConfigMaps or Secrets** and reference them in pods
- Avoid hardcoding credentials—use Secrets for sensitive info

sample manifest file

```
Kind: pod
apiVersion: v1
metadata:
    name: labels
    labels:
      author: aravind
      project : k8s
spec:
    containers:
    - name: nginx
      image: nginx
```

once you create pod with env variables, log into pod check env

```
kubectl exec -it <pod-name> -- bash

root@env-test:/# env
practice=k8s
project=micro-services
practice=k8s
```

07. Resources

In Kubernetes, **resources** refer to the **compute limits and requests** (CPU and memory) assigned to containers within a pod.

> In Kubernetes, resource requests specify the minimum CPU and memory required for a pod to be scheduled on a node, while resource limits define the maximum CPU and memory the pod can use on that node.

✅ **Why we use them**:

- To **prevent a container from using too many resources** and affecting others on the same node
- To **schedule containers efficiently** based on available node capacity
- To **enforce stability and fairness** in multi-tenant environments

Types of Resource Settings:

1. **Requests**
    - Minimum amount of CPU or memory **guaranteed** to the container
    - Kubernetes uses this to decide **where to schedule** the pod

    - If resources are tight, a pod may not be scheduled unless its request can be met

2. **Limits**
    - The **maximum amount** of CPU or memory a container is allowed to use
    - If a container exceeds its memory limit, it will be **terminated (OOMKilled)**
    - CPU overuse can be **throttled**, not killed

How to determine appropriate values:

- **Start with baseline usage**: Monitor your app locally or in test environments
- Use Kubernetes tools like:
- `kubectl top pod` (CPU/memory usage)
- Metrics server, Prometheus, or Datadog
- Gradually tune **requests and limits** based on observed behavior
- Set conservative defaults, then adjust as needed to avoid underutilization or crashes

sample manifest file

```
kind: Pod
apiVersion: v1
metadata:
  name: project
  labels:
    name: project
    environment: dev
spec:
  containers:
  - name: app
    image: nginx
    resources:
      # soft limit
      requests:
        memory: "64Mi"
        cpu: "250m" # 1 cpu = 1000m
      # limits should be atleast same or more than requests i.e hard limit
      limits:
        memory: "128Mi"
        cpu: "500m"
```

08. ConfigMaps

A **ConfigMap** is a Kubernetes object used to **store configuration data** (key-value pairs) separately from application code. This allows you to **inject dynamic configuration** into your containers **without rebuilding images**.

sample manifest file configMap.yml

```
Kind: pod
apiVersion: v1
metadata:
    name: labels
    labels:
      author: aravind
      project : k8s
spec:
    containers:
    - name: nginx
      image: ngiapiVersion: v1
kind: ConfigMap
metadata:
  name: project
data:
  github: "https://github.com/xaravind/k8s-resources.git"
  db: "jdbc:mysql://mysql:3306/sample"
```

Pod-configMap.yml

```
apiVersion: v1
Kind: pod
metadata:
  name: pod-config
spec:
  containers:
  - name: app
    image: nginx
  envFrom: # to refer all values at once in configmaps
  - configMapKeyRef:
    name: project
  # env:
  # # to refer single values
  # - name: db
  #   valueFrom:
  #     configMapKeyRef:
  #       name: project # The ConfigMap name this value comes from.
  #       key: db #  The key to fetch.
```

✅ **Why we use ConfigMaps**:

- To **externalize app configuration** (e.g., URLs, feature flags, file paths)
- To keep containers **environment-agnostic**
- To **manage changes** to app settings without restarting or redeploying images
- To **reuse configurations** across multiple pods or deployments

Common ways to use ConfigMaps:

1. **As environment variables** in a container
2. **Mounted as files** inside a container (useful for config files)
3. Accessed programmatically via Kubernetes API (advanced use cases)

📌 **Best Practices**:

- Use ConfigMaps for **non-sensitive data** only (use Secrets for sensitive info)
- Combine with **Deployment strategies** to roll out config changes
- Keep your ConfigMaps under version control (e.g., as YAML files in Git)

kubectl get configmap # to list configmaps

kubectl describe configmap <configmap-name> # to show the details

09. Secrets

A **Secret** in Kubernetes is used to store **sensitive data** such as passwords, API keys, tokens, or certificates. Like ConfigMaps, Secrets decouple configuration from your application code — but with **added security**.

✅ **Why we use Secrets**:

- To **securely store and manage sensitive information**
- To avoid hardcoding secrets in container images or configuration files
- To **inject secrets into pods** via environment variables or mounted volumes
- To support **TLS, authentication, and encrypted credentials** management

Key Features:

- Secrets are **base64-encoded** in etcd (not encrypted by default, but can be with encryption-at-rest)

- They are only shared with pods that need them
- Kubernetes RBAC controls who can access Secrets

Common use cases:

- Database credentials
- SSH keys or TLS certificates
- Access tokens for external services (e.g., AWS, GitHub)

📌 **Best Practices**:

- Use Secrets **only for confidential data**
- Enable **encryption at rest** for etcd on your cluster
- Limit access with **RBAC policies**
- Avoid printing or logging secrets accidentally
- Use tools like **Sealed Secrets, HashiCorp Vault**, or **external secret stores** for advanced secret management

sample manifest files secrets.yml

```
apiVersion: v1
kind: Secret
metadata:
  name: pod-secret
data:
  username: "YWRtaW4K" # echo admin | base64
  password: "YWRtaW4xMjMK" #echo admin123 | base64
```

pod-secrets.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-secrets
spec:
  containers:
  - name: nginx
    image: nginx
    envFrom:
    - secretRef:
        name: pod-secret
```

10. Services

A **Service** is a stable way to expose a set of pods to other services or external users.

✅ **Why we use Services**:

- To give pods a **stable DNS name and IP** despite changing pod IPs
- To enable **load balancing** across multiple pods
- To control **how traffic reaches the application**

sample manifest files

pod-service.yml

```
apiVersion: v1
kind: pod
metadata:
  name: nginx
  labels:
    name: frontend
    environment: dev
spec:
  containers:
    - name: app-nginx
      image: nginx
```

kubectl apply -f pod-service.yml

service.yml

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
spec:
  selector:
    name: frontend
    environment: dev
  ports:
  - port: 80 # service port
    targetPort: 80 # target port
```

kubectl apply -f service.yml kubectl get service kubectl describe service nginx

10.1 NodePort

- Exposes the service on a **static port** on each node's IP
- Accessible from **outside the cluster** via `NodeIP:NodePort`
- ⚠️ Less flexible and secure for production
- Kubernetes opens a specific port (default range: **30000–32767**) on **all worker nodes**.
- You can then access your app using:

  ```
  http://<NodeIP>:<NodePort>
  ```

🔁 Flow:

```
Client --> NodeIP:NodePort --> Service --> Pod
```

Example:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30080
```

In this case:

- Traffic hits `http://<NodeIP>:30080`
- It forwards to the pod's port `8080`

10.2 ClusterIP

- Default service type
- Exposes the service **internally** within the cluster
- Ideal for **service-to-service** communication

10.3 LoadBalancer

- Provisions an **external load balancer** (cloud-dependent)
- Exposes service to **external traffic** using a **public IP**
- Ideal for **internet-facing applications**

lb.yml

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  type: NodePort
  selector:
    app: my-app
  ports:
    - port: 80
      targetPort: 8080
      nodePort: 30081
```

Flow for a `LoadBalancer` Service in Kubernetes:

```
Client
  ↓
External Load Balancer (Provisioned by Cloud Provider)
  ↓
NodePort (on Kubernetes Node)
  ↓
ClusterIP (Service inside the cluster)
  ↓
Pod (Application container)
```

🔍 Explanation:

- **External Load Balancer**: Created automatically when you use `type: LoadBalancer` on a supported cloud provider.
- **NodePort**: The load balancer forwards traffic to a NodePort on one of the cluster nodes.
- **ClusterIP**: The internal service that routes traffic to matching Pods.
- **Pod**: The actual container where your app is running.

🔚 End Section (Relationship between Pod, ReplicaSet, Deployment, and Service)

🔄 How Pod, ReplicaSet, Deployment, and Service Work Together

A typical application deployment in Kubernetes follows this flow:

1. **Pod**
   - The smallest unit that runs your container.
   - It can be manually created but is not recommended for production.

2. **ReplicaSet**
   - Ensures the desired number of pods are always running.
   - Automatically replaces failed pods.
   - Can be used directly but usually managed by a Deployment.

3. **Deployment**
   - Manages ReplicaSets and handles rollout/rollback strategies.
   - The recommended way to manage stateless apps in Kubernetes.

4. **Service**
   - Provides a stable endpoint to expose your pods (via selectors).
   - Ensures traffic is distributed across healthy pod replicas.

✅ Real-World Flow:
You define a **Deployment** → it creates and manages a **ReplicaSet** → which ensures multiple copies of a **Pod** → these pods are exposed using a **Service**.

This architecture ensures **resilience**, **scalability**, and **reliable communication** between components in your application.