Complete Kubernetes & Docker Guide

From Basics to Production

☸ Kubernetes — 8 Modules | Architecture, Storage, RBAC, Helm, EKS & more

🐳 Docker — 5 Modules | Images, Containers, Networking, Volumes, Compose

📝 Real YAML examples in every chapter

✔ Based on actual classroom notes and hands-on labs

By Dilip Kumar | 2026

Syllabus

Module 1: Introduction & Architecture

- Kubernetes Introduction
- Kubernetes Architecture
- Kubernetes Cluster (Self-Managed) Setup Using Kubeadm
- Kubernetes Namespace
- Kubernetes Objects Overview

Module 2: Core Kubernetes Resources

- POD
- Replication Controller
- Replica Set
- Daemon Set
- Deployment
  - Rolling Update
  - Recreate Strategy
- Stateful Set

Module 3: Storage & Configuration Management

- Service
- Persistent Volumes
- Persistent Volume Claim
- Dynamic Volumes
- ConfigMaps & Secrets

Module 4: Monitoring & Autoscaling

- Horizontal Pod Autoscaler (HPA) & Metrics Server
- Liveness & Readiness Probes

Module 5: Scheduling & Node Maintenance

- Node Selector
- Node Affinity
- Pod Affinity & Anti-Affinity
- Taints & Tolerations
- Cordon / Drain / Uncordon
- Scheduling Strategy Overview

Module 6: Security, Quotas, & Networking

- Resource Quota & Limit Range
- Network Policies
- Kubernetes RBAC (Role-Based Access Control)

Module 7: Cloud Kubernetes & Load Balancing

- EKS Kubernetes Cluster Setup
- Load Balancer Service
- Ingress Controller & Ingress Resource

Module 8: CI/CD & Helm

- Deploy Application Using Jenkins (CI/CD Integration)
- Helm: Package Manager for Kubernetes

DOCKER

Module 1: Docker Basics & Concepts

- Docker Introduction
- Containerization vs Virtualization
- Docker vs Virtual Machine
- Use Cases of Docker in Real Projects
- Before Docker and After Docker

Module 2: Installation & Architecture

- Installing Docker on Linux/Windows
- Docker Architecture: Docker Engine, CLI, Daemon, REST API
- Docker flow

Module 3: Docker Commands

- Basic Adhoc Commands (run, ps, exec, logs, etc.)
- Managing Containers and Images
- Working with Docker CLI

Module 4: Dockerfile & Image Creation

- What is a Dockerfile?
- Dockerfile Keywords (FROM, RUN, COPY, CMD, ENTRYPOINT, ENV, VOLUME, many more)
- Building Custom Docker Images
- Real-Time Dockerfile Examples

Module 5: Docker Objects

- Docker Images
- Docker Containers

- Docker Networks (bridge, host, overlay)
- Docker Volumes (bind mount, volume)

Module 6: Docker Registries

- What are Docker Registries?
- Using Docker Hub
- Using AWS ECR

Module 7: Docker Compose

- Introduction to docker-compose.yml
- Defining Multi-Container Applications
- Running Compose Projects (up, down, logs)

KUBERNETES

MODULE 1: Introduction & Architecture

Chapter 1: Kubernetes Introduction

Kubernetes is an open-source container orchestration platform used to automate the deployment, scaling, and management of containerized applications. It helps teams run applications reliably in production by managing containers efficiently.

K8s is the short name for Kubernetes (because there are 8 letters between K and s).

What is Container Orchestration?

Container orchestration is the process of automatically managing, organizing, and coordinating containers. It handles tasks such as deploying containers, scaling them based on load, and ensuring high availability. Kubernetes is one of the most popular container orchestration tools.

What Kubernetes Can Do

Using Kubernetes, we can perform:

- Pod deployment
- Auto scaling
- Resource allocation
- Load balancing
- Application self-healing

Main Responsibilities of Kubernetes

- Container deployment
- Scaling and downscaling applications
- Load balancing traffic across pods
- Managing application availability

Features of Kubernetes (K8s)

1. Auto Scheduling

When a new pod request is created, Kubernetes stores the request in etcd.The scheduler continuously monitors the cluster and automatically assigns pods to suitable nodes.

- If resource requirements are defined in the manifest file, the scheduler places the pod accordingly.
- If resources are insufficient, the pod remains in a Pending state until resources become available.

2. Self-Healing Capabilities

Kubernetes automatically detects and fixes failures:

- Restarts crashed pods
- Recreates failed containers
- Moves pods to healthy nodes if a node fails

This ensures high availability and reliability of applications.

3. Automatic Rollout and Rollback

Kubernetes supports automated deployments:

- New versions of applications can be rolled out smoothly
- If a new version causes issues, Kubernetes allows rollback to the previous stable version using a single command

4. Horizontal Scaling and Load Balancing

When application load increases, scaling is required to prevent downtime.

Vertical Scaling

- Increasing CPU or RAM of an existing pod
- Can cause application downtime

Horizontal Scaling

- Adding more pods or nodes
- No downtime
- Most commonly used and recommended approach

5. Load Balancing

Load balancing distributes traffic across multiple pods or servers:

- Prevents overloading a single server
- Improves application performance and availability

6. Service Discovery and Networking

- Each pod gets an internal IP address
- Kubernetes provides DNS names for services
- Enables easy pod-to-pod communication without hard-coding IPs

7. Storage Orchestration

Kubernetes supports different types of storage options:

1. EmptyDir

- Temporary storage
- Data is deleted when the pod is removed
- Suitable for temporary use only

2. HostPath

- Stores data on the node's filesystem
- Data persists after pod deletion
- Not reliable because pods may restart on different nodes

3. Persistent Volumes (EBS, etc.)

- Managed storage solutions (e.g., AWS EBS)
- Data remains consistent even if pods move between nodes
- Best option for production workloads

Conclusion

Kubernetes simplifies container management by providing automation, scalability, reliability, and high availability. It is widely used in modern cloud-native applications and is a core skill for DevOps engineers.

Chapter 2: Kubernetes Architecture

Kubernetes Architecture:

Kubernetes (K8s) is an open-source container orchestration platform used to automate the deployment, scaling, and management of containerized applications.

Kubernetes Architecture Overview

Kubernetes architecture is divided into two main parts:

- Control Plane
- Worker Nodes

Control Plane

The Control Plane (also called the Master Node) is responsible for managing the Kubernetes cluster. It makes global decisions about the cluster, such as scheduling workloads and maintaining the desired state.

The main components of the Control Plane are:

- kube-apiserver
- etcd
- kube-scheduler
- kube-controller-manager
- cloud-controller-manager

kube-apiserver

The kube-apiserver is the entry point to the Kubernetes cluster.

- All API requests (from users, CLI, or internal components) go through the API server.
- It validates requests and stores cluster state in etcd.

etcd

etcd is a distributed key-value store that acts as Kubernetes' database.

- Stores all cluster data (pods, services, configs, secrets, node state, etc.)
- Provides consistency and reliability for the cluster state.

kube-scheduler

The kube-scheduler decides which worker node a pod should run on.

- Checks available CPU, memory, and other resource requirements
- If no suitable node is available, the pod remains in Pending state

⚠ The scheduler does not start nodes — it only assigns pods to nodes.

kube-controller-manager

The controller manager runs controllers that continuously monitor the cluster state.

Examples:

- Node Controller (detects node failures)
- ReplicaSet Controller (maintains desired pod count)
- Deployment Controller (handles rolling updates)

If something goes wrong, controllers work to bring the cluster back to the desired state (self-healing).

cloud-controller-manager

The cloud controller manager integrates Kubernetes with cloud providers such as:

- AWS
- Azure
- GCP

It manages cloud-specific resources like:

- Load balancers
- Volumes
- Node lifecycle

Worker Nodes

Worker Nodes are where the actual applications run.

Each worker node contains the following components:

- kubelet
- kube-proxy
- Container Runtime (CRI)
- Pods

kubelet

The kubelet is the main agent running on each worker node.

- Communicates with the Control Plane
- Ensures that pods defined in the cluster are running
- Restarts containers if they fail
- Reports node and pod status back to the API server

kube-proxy

kube-proxy manages networking for pods and services.

- Maintains network rules
- Enables communication between pods
- Supports Service abstraction and load balancing inside the cluster

Container Runtime (CRI)

The Container Runtime Interface (CRI) is responsible for running containers.

Examples:

- containerd
- CRI-O
- Docker (via shim)

It pulls images, starts containers, stops them, and monitors their health.

Pods

A Pod is the smallest deployable unit in Kubernetes.

- A pod can contain one or more containers
- Containers in a pod share networking and storage

Chapter 3: Kubernetes Cluster Setup Using Kubeadm

To run applications effectively and securely, we need a Kubernetes cluster. A cluster provides the base infrastructure to deploy, manage, and scale containerized applications reliably.

Types of Kubernetes Clusters

1️⃣ Self-Managed Clusters

- Managed by you — you control the installation, configuration, and updates.
- Ideal for learning, small projects, or POCs.
- Examples:
  - Minikube: Great for small projects and local testing, but not suitable for production or large-scale workloads.
  - kubeadm: Used to practice Kubernetes and run small applications on self-managed nodes. Allows more flexibility than Minikube.

2️⃣ Cloud-Managed Clusters

- Managed by cloud providers like AWS, Azure, or GCP.
- The cloud provider handles the control plane, upgrades, and high availability.
- Your involvement is mostly with deploying applications and managing workloads.
- Examples:
  - EKS — AWS
  - AKS — Azure
  - GKE — Google Cloud

These clusters can be created using cloud consoles or CLI tools and can be secured using cloud-specific scripts and IAM policies.

Installing a Self-Managed Cluster Using kubeadm

Prerequisites

- Machines / VMs (minimum 2): 1 Master node + 1+ Worker nodes
- OS: Ubuntu 20.04 / 22.04 or CentOS 7/8
- Minimum 2 CPU and 2 GB RAM per node
- Network connectivity between all nodes
- Disable swap:

sudo swapoff -a

- Install Docker or a container runtime (containerd recommended)

Step 1: Install kubeadm, kubelet, and kubectl

Run on all nodes:

sudo apt-get update

sudo apt-get install -y apt-transport-https ca-certificates curl

curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -

cat <<EOF | sudo tee /etc/apt/sources.list.d/kubernetes.list

deb https://apt.kubernetes.io/ kubernetes-xenial main

EOF

sudo apt-get update

sudo apt-get install -y kubelet kubeadm kubectl

sudo apt-mark hold kubelet kubeadm kubectl

Step 2: Initialize the Master Node

sudo kubeadm init --pod-network-cidr=192.168.0.0/16

```
  --pod-network-cidr is required for most network plugins (e.g., Calico)
```

- Copy the join command for worker nodes

Step 3: Configure kubectl on Master

mkdir -p $HOME/.kube

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

sudo chown $(id -u):$(id -g) $HOME/.kube/config

Verify cluster:

kubectl get nodes

kubectl get pods --all-namespaces

Step 4: Install Pod Network Add-on

Example: Calico network

kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml

- Enables pods across nodes to communicate

Step 5: Join Worker Nodes

```
  On each worker node, run the join command from kubeadm init:
```

sudo kubeadm join <master-ip>:6443 --token <token> --discovery-token-ca-cert-hash sha256:<hash>

- Verify on master:

kubectl get nodes

```
  All nodes should show Ready.
```

Step 6: Deploy a Test Application

Example: Nginx

kubectl create deployment nginx --image=nginx

kubectl expose deployment nginx --port=80 --type=NodePort

kubectl get svc

```
Access via <node-ip>:<NodePort> in your browser
```

Optional: Cluster Maintenance Commands

- Drain a node:

kubectl drain <node-name> --ignore-daemonsets

- Uncordon a node:

kubectl uncordon <node-name>

cloud managed cluster AWS:

installation:

Kubernetes Cluster Setup on AWS (Cloud-Managed Clusters)

To run applications effectively and securely, we need a Kubernetes cluster. Clusters can be self-managed or cloud-managed. Let's start with cloud-managed clusters, which are ideal for production and large-scale projects.

Cloud-Managed Clusters

- Managed by cloud providers like AWS, Azure, or GCP
- The provider handles the control plane, upgrades, and high availability
- You only manage worker nodes and application deployments

Popular cloud-managed clusters:

Provider	Cluster Service
AWS	EKS (Elastic Kubernetes Service)
Azure	AKS (Azure Kubernetes Service)
GCP	GKE (Google Kubernetes Engine)

- Cloud-managed clusters can be created via cloud consoles or CLI tools

- Security and IAM policies can be enforced using provider scripts

AWS EKS Cluster Creation (Step by Step)

Prerequisites

- Active AWS account
- AWS CLI installed and configured:

aws configure

- kubectl installed
- eksctl installed (recommended for easy cluster creation)

Step 1: Create Cluster Using eksctl

eksctl create cluster \

--name my-eks-cluster \

--region us-east-1 \

--nodes 3 \

--node-type t3.medium

- Creates control plane, worker nodes, VPC, subnets, and security groups automatically
- Takes 10–15 minutes

Step 2: Update kubeconfig

aws eks --region us-east-1 update-kubeconfig --name my-eks-cluster

```
Allows kubectl to communicate with the cluster
```

Step 3: Verify Cluster

kubectl get nodes

kubectl get pods --all-namespaces

```
Nodes should show Ready
Core pods (like coredns) should be running
```

Step 4: Deploy Test Application

Example: Nginx

kubectl create deployment nginx --image=nginx

kubectl expose deployment nginx --port=80 --type=LoadBalancer

kubectl get svc

- Access the app via LoadBalancer external IP

Step 5: Optional Cleanup

eksctl delete cluster --name my-eks-cluster --region us-east-1

Chapter 4: Kubernetes Namespaces

Namespaces play a key role in Kubernetes for deploying applications, managing resources, and organizing services. They allow you to divide a cluster into virtual workspaces, making it easier to scale, manage, and isolate resources.

Why Namespaces Are Important

Imagine a company with hundreds of employees. Sending an email to everyone about a specific department's work can be chaotic. Instead, companies create departments (QA, Dev, Managers, etc.) to organize communication.

Similarly, in Kubernetes:

- A cluster may host hundreds of applications, services, and resources.
- Without organization, managing these resources becomes difficult.
- Namespaces help avoid conflicts and provide isolation.

Namespaces can help with:

- Resource isolation
- Access control (RBAC)
- Environment separation (Dev, QA, Prod)
- Resource quotas for limiting usage

Types of Namespaces in Kubernetes

Kubernetes comes with five main types of namespaces:

Namespace	Purpose
default	The default namespace for resources when none is specified. All standard deployments go here if no namespace is mentioned.
kube-system	Used by Kubernetes for system components and internal services (like kube-dns, kube-proxy).
kube-node-lease	Tracks node health; used for node heartbeat and leader election.
kube-public	Public namespace, readable by all users (even unauthenticated users).
custom namespaces	Created by users for organizing applications or environments. Can be deleted and managed by the user.

Example: Managing Namespaces

Creating a namespace

kubectl create ns prod

OR

kubectl create namespace prod

Listing namespaces

kubectl get ns

kubectl get namespaces

Deleting a namespace

kubectl delete ns prod

Switching namespace context

kubectl config set-context --current --namespace=prod

Describing a namespace

kubectl describe ns prod

Creating a Namespace Using YAML

apiVersion: v1

kind: Namespace

metadata:

name: prod

Apply the YAML file:

kubectl apply -f namespace.yaml

```
✔ Summary:Namespaces are essential for organizing, isolating, and managing
resources in Kubernetes. Use system namespaces (default, kube-system, etc.) for
internal operations, and create custom namespaces for your applications,
environments, or projects.
```

Chapter 5: Kubernetes Pods & Objects Overview

In Kubernetes, a Pod is the smallest deployable unit in the cluster. It represents one or more containers that run together, share resources, and act as an instance of your application.

Key Features of Pods

- Pods can contain one or more containers, created from Docker images, which are built from Dockerfiles.
- Each Pod gets a unique IP address to allow communication within the node and cluster.
- Application Pods run only on worker nodes.

```
System Pods (like kube-apiserver, coredns, etc.) run on the master node.
```

- By default, application Pods cannot run on the master node unless taints/tolerations are set.
- Pods are managed by kubelet on worker nodes, and kube-proxy handles network communication for Pods.

Pod Fundamentals

- Single Unit: A Pod is considered a single unit, even if it contains multiple containers.
- Shared Resources: Containers in a Pod share storage, network, and metadata.
- Ephemeral: Pods are temporary. Deleting a Pod removes all its containers; they do not self-heal unless managed by controllers like ReplicaSets or Deployments.

Types of Pods

- Single-Container Pod
- Contains one container.
- Ideal for simple applications and easier to monitor logs, status, and performance.
- Example: Web server Pod running Nginx.
- Multi-Container Pod
- Contains two or more containers that work together.
- Containers can share data, logs, or perform supporting tasks.
- Example: Main application + sidecar container for logging or metrics.

Creating Pods Using YAML

Example Pod YAML:

apiVersion: v1

kind: Pod

metadata:

name: eapp

```
  namespace: prod   # can be prod, dev, qa, staging, etc.
  labels:
    app: myapplication
    env: prod
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```

Explanation:

- apiVersion & kind: Kubernetes metadata. Check with:

kubectl api-resources

- metadata.name: Pod name, choose meaningful names.
- metadata.namespace: Namespace for resource isolation.
- labels: Key-value pairs to identify and select Pods.
- spec.containers: Defines container configuration, image, and ports.

Pod Commands

- Apply Pod YAML:

kubectl apply -f file.yaml

- Describe Pod:

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

- Delete Pod:

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

- List Pods in a Namespace:

kubectl get pods -n <namespace>

Summary

- Pods are the building blocks of Kubernetes.
- They can be single or multi-container, share storage/network, and run on worker nodes.
- Pods are ephemeral and require controllers like Deployments for self-healing, scaling, and management.

Chapter 6: Static Pods in Kubernetes

In Kubernetes, a Pod is the smallest deployable unit. Among different types of pods, Static Pods are a special category that plays an important role at the node level.

This blog explains what static pods are, how they work, their advantages, and when to use them.

What Is a Static Pod?

A Static Pod is a pod that is managed directly by the kubelet running on a node, instead of being managed by the Kubernetes API server.

```
Unlike regular pods created using kubectl, static pods are created from YAML
manifest files stored locally on the node. The kubelet continuously monitors these
files and ensures the pods are always running.
```

How Static Pods Work

- Static pod manifests are stored on the node, usually in the directory:

/etc/kubernetes/manifests

- The kubelet watches this directory
- When a YAML file is added:
  - kubelet automatically creates the pod
- If the pod crashes or is deleted:
  - kubelet recreates it automatically (self-healing)
- If the manifest file itself is deleted:
  - the pod is permanently removed

Static pods do not require an API request to be created, but Kubernetes still shows them as mirror pods in the cluster for visibility.

Why Static Pods Are Used

Static pods are commonly used for:

- Kubernetes control plane components
  - kube-apiserver
  - kube-scheduler
  - kube-controller-manager
  - etcd
- Node-level services
- Critical applications that must always run on a specific node

They are not typically used for general application deployment, which is better handled using Deployments or StatefulSets.

Key Features of Static Pods

Self-Healing Capability

If a static pod crashes or is deleted, the kubelet automatically recreates it as long as the manifest file exists.

No API Server Dependency

Static pods can run even if the API server is unavailable.

Node-Specific

Static pods always run on the node where the manifest file exists. Kubernetes scheduling does not apply to them.

Advantages of Static Pods

- No dependency on Kubernetes API server
- Managed directly by kubelet
- Automatic recovery (self-healing)
- Ideal for critical system-level components
- Simple and lightweight

Limitations of Static Pods

- Cannot be managed using Deployments or ReplicaSets
- No automatic scaling

```
Cannot be deleted permanently using kubectl
```

- Must be managed manually on each node

Example: Static Pod YAML

apiVersion: v1

kind: Pod

metadata:

name: app

namespace: prod

labels:

app: app

env: prod

spec:

```
containers:
  - name: app
    image: nginx:latest
    ports:
      - containerPort: 80
```

How to Run This as a Static Pod

```
Save the file as app.yaml
```

- Place it in the following directory on the node:

/etc/kubernetes/manifests/app.yaml

- The kubelet will automatically create and run the pod

To stop the pod, delete the YAML file from the directory.

Conclusion

Static pods are a powerful Kubernetes feature designed for node-level and critical workloads. They provide high reliability through self-healing and operate independently of the API server. While they are not suitable for most application deployments, they are essential for running Kubernetes core components and system services.

Chapter 7: Replication Controller

In Kubernetes, a Pod is the smallest deployable unit, but a pod by itself has limitations.If a pod crashes or gets deleted, Kubernetes does not recreate it automatically. To overcome this problem, Kubernetes provides a controller called ReplicationController.

What Is a ReplicationController?

A ReplicationController (RC) is a Kubernetes object that ensures a fixed number of pod replicas are running at all times.

If a pod is:

- Deleted
- Crashed
- Terminated unexpectedly

The ReplicationController automatically creates a new pod to maintain the desired number of replicas.

Why ReplicationController Is Needed

Drawbacks of a Single Pod

- Cannot maintain a fixed number of pods
- No self-healing
- Manual recreation required if pod fails

Solution: ReplicationController

- Maintains desired pod count
- Automatically recreates pods
- Improves application availability
- Avoids manual errors

How ReplicationController Works

- You define the number of replicas in the YAML file
- RC continuously watches the pods using labels and selectors
- If the running pod count is less than desired replicas:
  - RC creates new pods automatically

Example: ReplicationController YAML

apiVersion: v1

kind: ReplicationController

metadata:

```
  name: app
  namespace: prod
  labels:
    app: app
    env: prod
spec:
  replicas: 2
  selector:
    app: app
  template:
    metadata:
      labels:
        app: app
    spec:
      containers:
        - name: devs
          image: nginx:latest
          ports:
            - containerPort: 80
```

What This Does

- Runs 2 identical pods

```
Pods are identified using label app: app
```

- If one pod is deleted, a new one is created automatically

Exposing Pods Using a Service

To access the pods, we use a Service.

Service YAML

```
apiVersion: v1
kind: Service
metadata:
  name: app
```

```
  namespace: prod
spec:
  type: ClusterIP
  selector:
    app: app
  ports:
    - port: 8080
      targetPort: 80
```

Result

- One service
- Load balances traffic across 2 pods
- Pods can be replaced without affecting the service

Behavior Explanation

- When you apply the YAML:
  - 2 pods are created
  - 1 service routes traffic to both pods
- If you delete one pod:
  - ReplicationController automatically recreates it
- This provides self-healing and high availability

Limitations of ReplicationController

- Does not support advanced label selectors
- Cannot handle rolling updates efficiently
- Considered legacy

Because of these limitations, Kubernetes introduced ReplicaSet.

Conclusion

ReplicationController helps overcome the limitations of standalone pods by:

- Maintaining a fixed number of pods
- Automatically recreating failed pods
- Improving reliability and availability

Although it is mostly replaced by ReplicaSet and Deployment, understanding ReplicationController is important for Kubernetes fundamentals and interviews.

Chapter 8: Replica Set

ernetes

A ReplicaSet is a Kubernetes object used to overcome the limitations of ReplicationController.Like ReplicationController, a ReplicaSet ensures that a fixed number of pod replicas are running at any given time.

However, ReplicaSet supports modern Kubernetes features that are required to deploy applications reliably and efficiently.

Why ReplicaSet Was Introduced

Limitations of ReplicationController

- Supports only simple label selectors
- Cannot handle updated or complex labels
- Limited support for rolling updates
- Considered legacy
- Not used directly with Deployments

Because of these limitations, Kubernetes introduced ReplicaSet, which is now the recommended replacement.

What Is a ReplicaSet?

A ReplicaSet ensures that a specified number of identical pods are running at all times.

- If a pod is deleted or crashes → ReplicaSet creates a new pod
- If extra pods are created manually → ReplicaSet deletes them
- Ensures the cluster always matches the desired state

ReplicaSet and Deployment Relationship

In modern Kubernetes:

- We usually do not create ReplicaSets directly
- Instead, we create a Deployment
- The Deployment manages ReplicaSets in the background

👉 When you apply a Deployment YAML:

- Kubernetes automatically creates a ReplicaSet
- The ReplicaSet then creates and manages pods

Key Features of ReplicaSet

Fixed Number of Pods

ReplicaSet always maintains the number of replicas defined in the YAML file.

Advanced Label Selectors

ReplicaSet supports:

```
matchLabels
matchExpressions
```

These are not supported by ReplicationController.

Pod Adoption

ReplicaSet can adopt existing pods if:

- Pod labels match the selector
- The pods are not already owned by another controller

Important Difference from ReplicationController

- In ReplicationController, the selector can be omitted (auto-selected)
- In ReplicaSet, the selector is mandatory
- Selector must match the pod template labels

Example: ReplicaSet YAML

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: app

namespace: prod

labels:

app: app

env: prod

spec:

replicas: 2

selector:

matchLabels:

application: application

template:

metadata:

```
    labels:
      application: application
  spec:
    containers:
      - name: my-app
        image: nginx:latest
        ports:
          - containerPort: 80
```

Exposing ReplicaSet Pods Using a Service

```
apiVersion: v1
kind: Service
metadata:
  name: sap
  namespace: prod
spec:
  type: ClusterIP
  selector:
    application: application
  ports:
    - port: 8080
      targetPort: 80
```

What Happens When You Apply This YAML?

- ReplicaSet creates 2 pods
- Service selects pods using labels
- Traffic is load-balanced across the pods
- If one pod is deleted → ReplicaSet recreates it automatically

ReplicationController vs ReplicaSet

Feature	ReplicationController	ReplicaSet
Fixed pod count	✔	✔
Self-healing	✔	✔
matchLabels	✖	✔

matchExpressions	✗	✓
Selector required	✗	✓
Used by Deployment	✗	✓
Current usage	✗ (legacy)	✓

Conclusion

ReplicaSet is a modern and powerful replacement for ReplicationController.While both ensure a fixed number of pods, ReplicaSet adds:

- Advanced label selectors
- Better control
- Seamless integration with Deployments

☞ In real-world Kubernetes usage, Deployments + ReplicaSets are the standard approach

Chapter 9: Daemon Set

In Kubernetes, a DaemonSet is a powerful workload object used to ensure that a copy of a Pod runs on every node in your cluster. Unlike Deployments, DaemonSets do not use replicas, and the number of Pods is automatically managed based on the number of nodes.

In this blog, we'll explore DaemonSets, how they work, why you might use them, and how to expose them internally with a Service.

What is a DaemonSet?

A DaemonSet guarantees that:

- Exactly one Pod runs on each node
- When a new node is added, a Pod is automatically created
- When a node is removed, its Pod is automatically deleted

Because of this behavior, DaemonSets are ideal for node-level operations rather than standard application workloads.

Key Characteristics of DaemonSet

- ✕ No replica count required
- ✓ One Pod per node automatically
- ✓ Auto-handles node additions and removals
- ✕ Not a replacement for Deployment
- ✓ Used for infrastructure-level services

Real-World Use Cases

DaemonSets are commonly used for:

- Log collection agents (Fluentd, Filebeat)
- Monitoring agents (Node Exporter, Prometheus Node Exporter)
- Security agents
- Networking components (CNI plugins)

These workloads need to run on every node to collect logs, metrics, or system data.

Running Pods on Master / Control-Plane Nodes

By default, Kubernetes control-plane nodes are tainted to prevent regular workloads from running on them:

node-role.kubernetes.io/control-plane:NoSchedule

To schedule DaemonSet Pods on master nodes, we use tolerations in the Pod spec:

tolerations:

```
- key: "node-role.kubernetes.io/control-plane"
  operator: "Exists"
  effect: "NoSchedule"
```

Using hostNetwork and hostPID

- hostNetwork: truePod uses the node's network namespace, sharing the node's IP. Useful for monitoring, logging, or networking agents.
- hostPID: truePod can access node-level processes, which is essential for debugging, monitoring, or security agents.

⚠ These settings give elevated privileges, so they should be used carefully.

DaemonSet YAML Example

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: app
  namespace: prod
  labels:
    app: app
    env: prod
spec:
  selector:
    matchLabels:
      app: app
  template:
    metadata:
      labels:
        app: app
    spec:
      hostNetwork: true
      hostPID: true
      tolerations:
        - key: "node-role.kubernetes.io/control-plane"
```

```
      operator: "Exists"
      effect: "NoSchedule"
  containers:
   - name: apps
     image: nginx:latest
     ports:
       - containerPort: 80
```

```
This DaemonSet ensures that an nginx Pod runs on every node, including master
nodes.
```

Exposing DaemonSet with a ClusterIP Service

Sometimes, you want to access your DaemonSet Pods internally within the cluster. A ClusterIP Service can route traffic to the Pods managed by the DaemonSet.

apiVersion: v1

kind: Service

metadata:

name: app

namespace: prod

spec:

type: ClusterIP

selector:

app: app

ports:

- port: 8080

targetPort: 80

- The Service listens on port 8080 and forwards traffic to the DaemonSet Pods on port 80.

```
This allows other Pods in the cluster to access your DaemonSet via
app.prod.svc.cluster.local:8080.
For external access, you could use NodePort or LoadBalancer instead of ClusterIP.
```

Summary

- DaemonSet runs one Pod per node
- Automatically handles node addition and removal

- Used for cluster-level services, not application deployments
- Tolerations allow Pods to run on master nodes
- hostNetwork and hostPID provide node-level access
- Exposing via a Service allows internal communication with the DaemonSet Pods

Final Thoughts

DaemonSets are an essential part of Kubernetes operations. They are not meant for deploying applications but are crucial for monitoring, logging, security, and networking tasks that must run on every node. Combining DaemonSets with Services gives you both node-level coverage and internal accessibility for your cluster workloads.

Chapter 10: Deployment

In Kubernetes, a Deployment is one of the most commonly used workload objects in real-world applications. Deployments provide a declarative way to manage Pods, allowing you to scale, update, and rollback applications easily.

What is a Deployment?

A Deployment manages a set of Pods and ensures that:

- A fixed number of Pods (replicas) are always running
- You can perform rolling updates to deploy a new version
- If a new version causes issues, you can rollback to a previous version
- Provides self-healing, so crashed Pods are automatically replaced

Unlike a ReplicaSet, a Deployment adds features like rollout and rollback, making it the preferred choice for real-world applications.

Why Use Deployment?

- Quickly deploy new applications or updates
- Ensure a fixed number of Pods are always running
- Rollback safely if the current version misbehaves
- Declarative and automated management of Pods

Example use case:

```
Suppose you deploy version v1 of an application. Later, you add a patch and deploy
version v2. If v2 has errors or performance issues, the client can request a
rollback to v1. Deployment makes this safe and easy, which ReplicaSet alone cannot
do.
```

Deployment YAML Example

Here's a corrected and working Deployment YAML:

apiVersion: apps/v1

kind: Deployment

metadata:

name: app

namespace: prod

labels:

app: app

env: prod

spec:

```
 replicas: 2
 selector:
  matchLabels:
   app: app
 template:
  metadata:
   labels:
    app: app
  spec:
   containers:
    - name: app
      image: nginx:latest
      ports:
       - containerPort: 80
      resources:
       requests:
        cpu: "3"
        memory: "512Mi"
       limits:
        cpu: "4"
        memory: "600Mi"
      volumeMounts:
       - name: app
         mountPath: /mnt/apple
   volumes:
    - name: app
      nfs:
       server: <NFS_SERVER_IP>
       path: /mnt/apple
```

Exposing Deployment via Service

To allow internal communication within the cluster, we use a ClusterIP Service:

```
apiVersion: v1
kind: Service
metadata:
  name: app
  namespace: prod
spec:
  type: ClusterIP
  selector:
    app: app
  ports:
    - port: 8080
      targetPort: 80
```

Summary

- Deployment manages a fixed number of Pods (replicas)
- Supports rolling updates and rollbacks
- Automatically replaces failed Pods
- Preferred over ReplicaSet for real-world applications
- Combine with a Service to expose the Pods internally or externally
- Supports volumes for persistent storage

Final Thoughts

Deployments are the backbone of production workloads in Kubernetes. They allow developers and operators to manage application lifecycle efficiently, including updates and rollbacks, which is critical in a real-world environment.

Pairing Deployments with Services ensures that Pods are accessible within or outside the cluster, and using volumes allows persistent data storage for your applications.

Chapter 11: StatefulSet

In Kubernetes, applications are categorized as stateless and stateful. Based on this classification, Kubernetes provides different workload objects.For database applications, the most reliable and recommended object is StatefulSet.

```
This blog explains why StatefulSet is used, how it differs from Deployment, and
shows a real-time MySQL database example using orders-db.
```

Stateless vs Stateful Applications

Stateless Applications

Stateless applications do not store data and do not depend on previous state.

Characteristics:

- No database dependency
- Pod names change on restart
- Easy to scale
- Deployed using Deployment

Examples:

- Frontend applications
- APIs
- Microservices without databases

Stateful Applications

Stateful applications depend on persistent data and stable identity.

Characteristics:

- Data must not be lost
- Pod identity must remain the same
- Persistent storage required
- Deployed using StatefulSet

Examples:

- MySQL
- PostgreSQL
- MongoDB
- Redis (with persistence)

What is a StatefulSet?

A StatefulSet is a Kubernetes workload object used to deploy database pods/replicas securely.It ensures:

- Stable pod names
- Persistent storage
- Ordered pod startup and shutdown

Because of these features, StatefulSet is widely used for production database workloads.

Key Features of StatefulSet

1. Stable and Sequential Pod Names

StatefulSet pods have fixed and ordered names:

orders-db-0

orders-db-1

orders-db-2

If a pod is deleted or restarted:

- The same pod name is recreated
- Data remains consistent

In Deployments, pod names change every time, which is risky for databases.

2. Ordered Pod Creation and Deletion

StatefulSet pods:

- Start one by one
- Next pod starts only after the previous pod is running

Deployments start all pods at the same time.

This ordered behavior protects database consistency.

3. Persistent Storage Per Pod

Each StatefulSet pod gets:

- Its own PersistentVolumeClaim (PVC)
- Data survives pod restarts and rescheduling

4. Headless Service (Mandatory)

StatefulSet requires a Headless Service:

clusterIP: None

Why Headless Service?

- No load balancing
- Provides stable DNS records
- Example DNS:

orders-db-0.orders-db-headless.default.svc.cluster.local

Deployments do not require a headless service.

5. Slower Rollbacks

StatefulSet rollbacks are:

- Slower than Deployments
- Performed pod by pod
- Designed this way to avoid data corruption

Secure Credential Management

In real-time projects:

- Never hardcode credentials
- Use ConfigMaps for non-sensitive data
- Use Secrets for passwords and sensitive values

```
Real-Time Example: MySQL StatefulSet (orders-db)
```

ConfigMap

apiVersion: v1

kind: ConfigMap

metadata:

name: orders-db-config

data:

MYSQL_USER: "orders_user"

Secret

apiVersion: v1

kind: Secret

metadata:

name: orders-db-secret

type: Opaque

data:

MYSQL_PASSWORD: b3JkZXJzMTIz

(Password is Base64 encoded)

Headless Service

apiVersion: v1

kind: Service

metadata:

```
  name: orders-db-headless
spec:
  clusterIP: None
  selector:
    app: orders-db
  ports:
    - port: 3306
```

StatefulSet Definition

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: orders-db
spec:
  serviceName: orders-db-headless
  replicas: 1
  selector:
    matchLabels:
      app: orders-db
  template:
    metadata:
      labels:
        app: orders-db
    spec:
      containers:
        - name: orders-db
          image: mysql:8.0
          ports:
            - containerPort: 3306
          env:
            - name: MYSQL_ROOT_PASSWORD
              valueFrom:
```

```
            secretKeyRef:
              name: orders-db-secret
              key: MYSQL_PASSWORD
          - name: MYSQL_USER
            valueFrom:
              configMapKeyRef:
                name: orders-db-config
                key: MYSQL_USER
          - name: MYSQL_PASSWORD
            valueFrom:
              secretKeyRef:
                name: orders-db-secret
                key: MYSQL_PASSWORD
        volumeMounts:
          - name: orders-db-storage
            mountPath: /var/lib/mysql
  volumeClaimTemplates:
    - metadata:
        name: orders-db-storage
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
```

Why StatefulSet is Preferred for Databases

- Stable pod identity
- No data inconsistency
- Persistent storage per pod
- Ordered startup and shutdown
- Secure credential handling
- Production-ready database deployments

Deployment vs StatefulSet (Quick Comparison)

Feature	Deployment	StatefulSet
Pod names	Random	Fixed & sequential
Storage	Shared	Dedicated per pod
Startup order	Parallel	Ordered
Best for	Stateless apps	Databases
Headless service	Not required	Required

Conclusion

StatefulSet is a critical Kubernetes object for database workloads.Unlike Deployment, it guarantees stable pod identity, persistent storage, and controlled lifecycle, making it the best choice for real-time production databases.a

Chapter 12: StatefulSet vs Deployment

Kubernetes provides several objects to deploy applications, but StatefulSets and Deployments are the most common. Choosing the right one depends on whether your application is stateful or stateless.

What is a StatefulSet?

A StatefulSet is a Kubernetes object used to deploy stateful applications, like databases (MongoDB, MySQL, PostgreSQL).

Key Features of StatefulSet:

```
  Provides stable, unique identities for pods (e.g., mongo-0, mongo-1).
```

- Pods are started sequentially: Pod-1 starts only after Pod-0 is ready.
- Each pod retains its persistent storage even if deleted.
- Works with PersistentVolumeClaims (PVCs) to automatically create storage for pods.
- Slower than Deployments because pods start one by one.
- Not suitable for stateless applications.

Use Case: Databases, distributed stateful apps, or applications that need stable pod identities.

What is a Deployment?

A Deployment is a Kubernetes object used to deploy stateless applications, like web servers or APIs.

Key Features of Deployment:

- Pods are started all at once and are interchangeable.
- Pod identities are random; labels are used for identification.
- Fast deployment and scaling.
- Pod storage is usually ephemeral (unless using PVCs separately).
- Suitable for stateless applications.

Use Case: Web servers, APIs, microservices.

Key Differences Between StatefulSet and Deployment

Feature	StatefulSet	Deployment
Application type	Stateful	Stateless
Pod identity	Stable & unique	Random
Pod startup	Sequential	Parallel
Storage	Persistent (PVCs)	Usually ephemeral
Speed	Slower	Faster
Use case	Databases, stateful apps	Web servers, stateless apps

Important Notes About Services in StatefulSets

```
For StatefulSets, we usually use headless services (clusterIP: None) instead of
normal LoadBalancer services.
```

Why headless service?

- Provides stable DNS names for pods (important for databases and replica sets).
- No load balancing; each pod is addressed directly by name.
- Ensures internal cluster communication works correctly.

```
Why Services Matter for StatefulSetsFor StatefulSets, we usually create a headless
service (clusterIP: None) and specify it as serviceName in the StatefulSet.Why?
```

- Stable pod DNS names:

pod-0.mongodb.prod.svc.cluster.local

pod-1.mongodb.prod.svc.cluster.local

- Even if pods are deleted and recreated, they keep the same identity.
- Internal cluster communication:Essential for databases and clusters where pods need to talk to each other reliably.
- Pod discovery & replica sets:Allows other pods or apps to find pods predictably by name.

Example Headless Service:

apiVersion: v1

kind: Service

metadata:

name: mongodb

namespace: prod

spec:

clusterIP: None # Headless service

selector:

app: mongodb

ports:

- port: 27017

targetPort: 27017

protocol: TCP

💡 Best practice: For databases, use headless service unless you explicitly need external access.

Example: StatefulSet YAML for MongoDB

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: mongodb
 namespace: prod
spec:
 replicas: 1
 selector:
  matchLabels:
   app: mongodb
 serviceName: "mongodb"
 template:
  metadata:
   labels:
    app: mongodb
  spec:
   containers:
   - name: mongo
    image: mongo:8.0.9
    ports:
    - containerPort: 27017
    env:
    - name: MONGO_DB_HOSTNAME
     valueFrom:
      configMapKeyRef:
       name: config
       key: db_hostname
    - name: MONGO_DB_USERNAME
     valueFrom:
      configMapKeyRef:
```

```
            name: config
            key: db_username
        - name: MONGO_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: secrets
              key: db_password
        volumeMounts:
        - name: mongo-pv
          mountPath: /data/db
  volumeClaimTemplates:
  - metadata:
      name: mongo-pv
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: config
  namespace: prod
data:
  db_hostname: mongodb
  db_username: dev
---
apiVersion: v1
kind: Secret
metadata:
```

```
  name: secrets
  namespace: prod
type: Opaque
stringData:
  db_password: dev123
```

✅ This setup:

- Uses headless service for pod identity.
- Creates persistent storage automatically via PVCs.
- Uses ConfigMap and Secret for database connection variables.

Summary

- Use StatefulSet for stateful apps (databases) that need stable identities and persistent storage.
- Use Deployment for stateless apps that need fast, parallel scaling.

```
Headless services (clusterIP: None) are preferred for StatefulSets to ensure pods
can communicate
```

Chapter 13: Init Containers & Sidecar Containers

In Kubernetes, Pods can contain more than one container. Two important patterns used in real-world deployments are:

- Init Containers
- Sidecar Containers

Let's understand both in a simple way with examples.

◆ What is an Init Container?

An Init Container runs before the main application container starts.

It is mainly used to:

- Check conditions before starting the app
- Wait for a dependency (like database, API, or service)
- Perform setup tasks (create files, run migrations, configure environment)
- Validate configuration
- Fail early if something is wrong

☞ The main application container will NOT start until the init container successfully completes.

If the init container fails:

- The Pod will not move to running state
- Kubernetes will retry the init container
- The main app will never start until it succeeds

This helps us detect problems early before deploying the actual application.

✔ Example: Init Container YAML

apiVersion: v1

kind: Pod

metadata:

name: initcontainer

namespace: prod

spec:

initContainers:

- name: init-wait

image: nginx

command: ['sh', '-c', 'echo Init Task Starting; sleep 10; echo Init Task Done']

containers:

- name: main-app

image: nginx

command: ['sh', '-c', 'echo Main App Running; sleep 3600']

How It Works:

```
init-wait container runs first.
```

- It performs its task (sleep 10 seconds here).

```
After successful completion, the main-app container starts.
```

- If init fails → main container will NOT start.

◆ What is a Sidecar Container?

A Sidecar Container runs along with the main container inside the same Pod.

It is used to:

- Collect logs
- Monitor application behavior
- Sync data
- Handle proxy tasks
- Provide security features
- Perform backups

It is basically a multi-container Pod, where one container supports the main application.

Example Use Case: Log Collection

Suppose:

- One container runs your application.
- Another container collects logs and sends them to monitoring systems.

Both run together in the same Pod.

✔ Example: Pod with Sidecar Container

apiVersion: v1

kind: Pod

metadata:

name: sidecar-example

```
  namespace: prod
spec:
  volumes:
  - name: shared-logs
    emptyDir: {}
  containers:
  - name: main-app
    image: nginx
    command: ['sh', '-c', 'echo Main App Running; while true; do echo "Log entry" >> /logs/app.log; sleep 5; done']
    volumeMounts:
    - name: shared-logs
      mountPath: /logs
  - name: sidecar-log
    image: nginx
    command: ['sh', '-c', 'tail -f /logs/app.log']
    volumeMounts:
    - name: shared-logs
      mountPath: /logs
```

How It Works:

```
main-app writes logs to /logs/app.log.
sidecar-log reads the same log file using shared volume.
Both containers share the emptyDir volume.
```

- When Pod stops → both containers stop.

Difference Between Init Container and Sidecar Container

Feature	Init Container	Sidecar Container
When it runs	Before main container	Along with main container
Purpose	Setup / validation	Support main app
Runs continuously?	No	Yes

If fails	Main app won't start	Pod may restart depending on restart policy

Why We Use Them in Real Projects

Init Container

- Wait for database to be ready
- Run DB migrations
- Pull configuration from external systems
- Validate secrets or config maps

Sidecar Container

- Log collection
- Monitoring agents
- Service mesh proxy (like Istio)
- File synchronization
- Security scanning

Final Thoughts

- Init containers help prevent bugs early by ensuring all pre-conditions are met before the application starts.
- Sidecar containers enhance or support the application while it is running.
- Both patterns are powerful in Kubernetes production environments.

Using them properly makes applications:

- More reliable
- More modular
- Easier to monitor
- More production-ready

Chapter 14: Kubernetes Storage — emptyDir, hostPath & PersistentVolumes

When running applications in Kubernetes, managing data storage is a critical task. Unlike traditional servers, Pods in Kubernetes are ephemeral, meaning they can be created, destroyed, or moved across nodes. Without proper storage, important data like database files or configuration information can be lost.

In Kubernetes, there are three main storage types to consider: emptyDir, hostPath, and PersistentVolumes (PV) with PersistentVolumeClaims (PVC). Let's explore each.

1. emptyDir: Temporary Storage for Pods

emptyDir is the simplest form of storage in Kubernetes.

- Behavior: Data exists only while the Pod is running. Once the Pod is deleted or restarted, the data is erased.
- Use case: Ideal for local testing, caching, or temporary scratch space.
- Limitations: Not suitable for production or persistent data needs.

```
Example scenario: A developer testing a web application might use emptyDir for
session data or temporary files.
```

2. hostPath: Node-Specific Persistent Storage

hostPath allows a Pod to access a directory on the node's filesystem.

- Behavior: Data persists even if the Pod is deleted, as long as it stays on the same node.
- Limitations:
  - Data is not accessible if the Pod is scheduled on a different node.
  - Security risks, because Pods can access the host machine's filesystem.
  - Limited support in cloud environments like AWS or GCP.

Example scenario: Testing a local database on a single node, or storing logs temporarily on a specific host.

Drawback: In a multi-node cluster, if a Pod moves to another node, the previous data is lost, causing data inconsistency.

3. Persistent Volumes: Reliable, Long-Term Storage

For production-grade applications, Kubernetes provides PersistentVolumes (PV) and PersistentVolumeClaims (PVC).

- PersistentVolume (PV): Represents a piece of storage in the cluster (e.g., NFS, AWS EBS, or Azure Disk).

- PersistentVolumeClaim (PVC): A request for storage by a Pod.

Behavior:

- PVs allow data persistence across Pod restarts and node changes.
- PVCs claim storage from PVs, making it easier for Pods to access it.
- Supports shared storage between multiple Pods if the PV allows it (ReadWriteMany).

Use case: Databases (MongoDB, MySQL), file storage, or any application that requires reliable long-term data storage.

Access Modes in PersistentVolumes

Access Mode	Description
ReadWriteOnce (RWO)	Mountable as read-write by a single node
ReadWriteMany (RWX)	Mountable as read-write by many nodes
ReadOnlyMany (ROX)	Mountable as read-only by many nodes

Example: MongoDB with Persistent Storage

Here's an example of using NFS-backed PV and PVC with MongoDB:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongo-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: <NFS_SERVER_IP>
    path: /mnt/nfs_share
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pvc
```

```
  namespace: prod
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-app
  namespace: prod
  labels:
    app: mongo
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mongo
  template:
    metadata:
      labels:
        app: mongo
    spec:
      containers:
      - name: mongo
        image: mongo:8.0.9
        ports:
        - containerPort: 27017
        volumeMounts:
```

```
    - name: mongo-storage
      mountPath: /data/db
  volumes:
  - name: mongo-storage
    persistentVolumeClaim:
      claimName: mongo-pvc
```

✅ Key Points:

- PV connects to the NFS server for storage.
- PVC requests the storage for the Pod.

```
MongoDB stores its data in /data/db, which persists across Pod restarts or rescheduling.
```

Summary Table

Storage Type	Persistence	Multi-Node Support	Use Case
emptyDir	No	N/A	Testing, temporary storage
hostPath	Yes	No	Node-specific storage, local testing
PersistentVolume	Yes	Yes	Production-grade apps, databases, shared storage

Best Practice: Always use PV + PVC for production workloads to ensure data is safe, persistent, and accessible across nodes.

Chapter 15: ConfigMaps & Secrets

In Kubernetes, hardcoding sensitive information like usernames, passwords, or API keys directly into your application code or deployment manifests is a big security risk. If credentials are exposed, it can lead to hacking, data leaks, or compromise of your entire application or network.

To handle credentials securely, Kubernetes provides Secrets and ConfigMaps. These objects allow you to separate configuration and sensitive data from your application code.

Why Not Hardcode Credentials?

- Hardcoding credentials in code or manifests exposes them to anyone who can access the repository.
- If attackers get a password, they can potentially access databases, cloud services, or internal systems.
- Hardcoded credentials are difficult to rotate or update across multiple environments.

ConfigMaps vs Secrets

Feature	ConfigMap	Secret
Data Type	Non-sensitive configuration	Sensitive data (passwords, tokens, keys)
Use Case	Hostnames, usernames, API endpoints	Database passwords, API keys, TLS certificates
Encoding	Plain text	Base64 encoded
Security	Not encrypted	Encrypted at rest (in etcd)

Tip: If the data is sensitive, always use Secrets instead of ConfigMaps.

How it works

When you deploy an application, you can reference Secrets and ConfigMaps in your pod's environment variables or volumes. This way, your application can access credentials securely without storing them in the code.

- The input data (like a password) is stored encrypted in Kubernetes.
- When the pod starts, Kubernetes injects it into the container.
- The application reads it in plain text, but it was never hardcoded.

Think of it like sending a message securely:

- Sender encrypts the message.
- Message travels securely.
- Receiver decrypts it.

This ensures that sensitive data is safe in transit and at rest.

Example: ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: prod
data:
  db_hostname: mongodb
  db_username: devdb
```

- Stores non-sensitive data like database hostname or username.
- Can be shared across multiple pods.

Example: Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: app-secret
  namespace: prod
type: Opaque
stringData:
  db_password: dev123
```

- Stores sensitive data like passwords.
- Base64 encoded and encrypted at rest in etcd.
- Can be updated without changing your application code.

Using ConfigMaps and Secrets in Deployment

Here's how you can reference them in your pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app
  namespace: prod
spec:
```

```
 replicas: 2
 selector:
  matchLabels:
   app: myapp
 template:
  metadata:
   labels:
    app: myapp
  spec:
   containers:
    - name: myapp-container
      image: nginx:latest
      ports:
       - containerPort: 80
      env:
       - name: MONGO_DB_HOSTNAME
         valueFrom:
          configMapKeyRef:
            name: app-config
            key: db_hostname
       - name: MONGO_DB_USERNAME
         valueFrom:
          configMapKeyRef:
            name: app-config
            key: db_username
       - name: MONGO_DB_PASSWORD
         valueFrom:
          secretKeyRef:
            name: app-secret
            key: db_password
```

How It Works:

```
MONGO_DB_HOSTNAME and MONGO_DB_USERNAME come from the ConfigMap.
MONGO_DB_PASSWORD comes from the Secret.
```

- Your application can access these environment variables securely without ever hardcoding them.

Benefits of Using ConfigMaps and Secrets

- Improved security: Sensitive data is encrypted and not exposed in code.
- Centralized management: Update values in one place without redeploying code.
- Environment flexibility: Different ConfigMaps/Secrets can be used for dev, stage, prod.
- Compliance: Helps meet security standards and auditing requirements.

Key Takeaways

- Never hardcode credentials in Kubernetes manifests or code.
- Use Secrets for sensitive information and ConfigMaps for non-sensitive configuration.
- Reference them via environment variables or volumes in pods.
- Kubernetes will handle encryption and secure delivery to your application.

By following these best practices, you can ensure your Kubernetes applications are secure, scalable, and easy to manage.

Chapter 16: Horizontal Pod Autoscaler (HPA) & Metrics Server

In Kubernetes, scaling Pods is essential to ensure that your application can handle increased load without downtime. There are two types of Pod scaling:

- Vertical Pod Scaling (VPA)
- Horizontal Pod Scaling (HPA)

1️⃣ Vertical Pod Autoscaling (VPA)

Vertical scaling means increasing the resources (CPU or memory) of an existing Pod when it reaches its limits.

Example:

```
A Pod has 512Mi memory allocated
```

- Its usage reaches 80%

```
You manually increase memory to 1112Mi
```

512Mi + 600Mi = 1112Mi

Problems with vertical scaling:

- Requires Pod restart, which may cause downtime
- Not ideal for applications that need high availability

Because of these limitations, vertical scaling is less commonly used in production.

2️⃣ Horizontal Pod Autoscaling (HPA)

Horizontal scaling means adding more Pods instead of increasing resources of a single Pod. This ensures high availability and prevents downtime.

Example:

- A Deployment has 2 Pods
- CPU utilization reaches 30% or memory reaches 80%
- HPA automatically increases the number of Pods to 3 or 4

This approach is safer and more reliable than vertical scaling.

HPA YAML Example

Here's a corrected YAML for Horizontal Pod Autoscaler:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
 name: app
 namespace: prod
spec:
 scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: apps
 minReplicas: 2
 maxReplicas: 4
 metrics:
  - type: Resource
   resource:
    name: cpu
    target:
     type: Utilization
     averageUtilization: 30
  - type: Resource
   resource:
    name: memory
    target:
     type: Utilization
     averageUtilization: 80
```

How HPA Works

- HPA monitors CPU and memory usage of Pods in a Deployment

```
If metrics exceed the thresholds (CPU > 30% or Memory > 80%), HPA increases the
number of Pods up to maxReplicas
If usage drops below the thresholds, HPA reduces Pods to minReplicas
```

This ensures no downtime and keeps the application responsive.

Summary

Scaling Type	What it does	Pros	Cons
Vertical Pod Autoscaling	Increases resources of a Pod	Easy for single-Pod apps	Requires restart, downtime
Horizontal Pod Autoscaling	Adds more Pods to the Deployment	High availability, auto-scaling	Consumes more nodes/resources

Key takeaway:In production, horizontal scaling (HPA) is preferred for most applications, while vertical scaling is only used for single-Pod workloads that cannot be horizontally scaled.

Chapter 17: Liveness & Readiness Probes

Kubernetes Probes in Kubernetes

In Kubernetes, probes are used to check the health and status of containers running inside pods.The two most commonly used probes are:

- Liveness Probe
- Readiness Probe

(There is also a Startup Probe, but we'll keep this blog focused on the main two.)

1️⃣ Liveness Probe

What is a Liveness Probe?

A liveness probe is used to check whether a container is still alive and working.

If the liveness probe fails, Kubernetes assumes the container is stuck or unhealthy and:👉 restarts the container automatically

This helps recover applications that are running but not responding.

What liveness probe is used for

- Detects deadlocks
- Detects applications that are hung
- Restarts containers without manual intervention

⚠ Important:

- Liveness probe does NOT check pod startup
- It does NOT fix Pending pods
- It restarts containers, not the entire pod

Example scenario

If an application:

- starts successfully
- but later becomes unresponsive
- or stops serving requests

The liveness probe fails, and Kubernetes restarts the container automatically.

Liveness Probe YAML Example

livenessProbe:

httpGet:

path: /mavenwebapp

port: 8080

initialDelaySeconds: 30

periodSeconds: 10

failureThreshold: 3

What this means

```
initialDelaySeconds: 30
```

- Kubernetes waits 30 seconds after container starts before checking health

```
periodSeconds: 10
```

- Probe runs every 10 seconds

```
failureThreshold: 3
```

- If it fails 3 times in a row, container is restarted

2️⃣ Readiness Probe

What is a Readiness Probe?

A readiness probe is used to check whether a container is ready to accept traffic.

If the readiness probe fails:👉 Kubernetes stops sending traffic to the pod👉 The pod is NOT restarted

What readiness probe is used for

- Controls traffic routing
- Ensures traffic goes only to ready pods
- Useful during:
  - application startup
  - heavy processing
  - temporary downtime

⚠ Important:

- Readiness probe is NOT based on liveness probe
- Readiness probe does NOT restart pods
- It only adds or removes the pod from Service endpoints

Example scenario

- Pod is running
- Application is still initializing
- Readiness probe fails

👉 Pod stays running👉 Traffic is blocked until the probe passes

Readiness Probe YAML Example

```
readinessProbe:
  httpGet:
    path: /mavenwebapp
    port: 8080
  initialDelaySeconds: 30
  periodSeconds: 10
  failureThreshold: 3
```

What this means

- After 30 seconds, Kubernetes starts checking readiness
- Probe runs every 10 seconds
- If it fails 3 times, traffic is stopped
- When it passes again, traffic is restored automatically

Key Differences

Feature	Liveness Probe	Readiness Probe
Purpose	Restart unhealthy containers	Control traffic
Restarts container	✔ Yes	✖ No
Stops traffic	✖ No	✔ Yes
Used for startup	✖ No	✔ Yes
Affects Service endpoints	✖ No	✔ Yes

Conclusion

Liveness and readiness probes are critical for building self-healing and reliable Kubernetes applications.Liveness probes ensure containers are restarted when they become unhealthy, while readiness probes ensure traffic is sent only to pods that are ready to serve requests.

Using both probes correctly improves application stability, availability, and user experience.

Chapter 18: Node Selector & Node Affinity

In Kubernetes, the scheduler decides which node a pod should run on. By default, pods can be scheduled on any node that has sufficient resources, but sometimes we need more control. That's where Node Selector and Node Affinity come in.

Node Selector

Node Selector is the simplest way to schedule pods onto specific nodes.

How It Works:

```
Each node can have labels (key-value pairs) like app=frontend.
In the pod YAML, you specify nodeSelector with the same label.
```

- The scheduler will place the pod only on nodes matching that label.
- If the label doesn't exist, the pod won't be scheduled.

Commands to Set Node Labels:

Get node names

kubectl get nodes

Label a node

kubectl label nodes <node-id> -n prod app=app

Remove a label

kubectl label nodes <node-id> app-

Example Pod YAML with Node Selector:

apiVersion: v1

kind: Pod

metadata:

name: sai

namespace: prod

labels:

app: app

env: prod

spec:

```
 nodeSelector:
  app: app
 containers:
 - name: sam
  image: nginx:latest
  ports:
  - containerPort: 80
```

Key Points:

- Node Selector is simple but strict: pods either match the label or they won't run.
- Typically used by infra teams, but DevOps may also use it in smaller companies.

Node Affinity

Node Affinity is a more flexible and advanced version of Node Selector.

- Lets you schedule pods to preferred or required nodes.
- Supports hard rules (must schedule) and soft rules (preferred, but optional).
- Useful for scheduling pods to specific regions or zones in a cluster.

Types of Node Affinity

1. Hard Rule (Required)

Pods must be scheduled on nodes that match the specified labels.

```
apiVersion: v1
kind: Pod
metadata:
 name: apppod
spec:
 affinity:
  nodeAffinity:
   requiredDuringSchedulingIgnoredDuringExecution:
    nodeSelectorTerms:
    - matchExpressions:
      - key: region
        operator: In
        values:
```

```
        - ap-south-1
       - key: zone
        operator: In
        values:
        - ap-south-1b
 containers:
 - name: nginx
  image: nginx
```

2. Soft Rule (Preferred)

Pods prefer certain nodes, but the scheduler can place them elsewhere if needed.

```
apiVersion: v1
kind: Pod
metadata:
 name: apppon
spec:
 affinity:
  nodeAffinity:
   preferredDuringSchedulingIgnoredDuringExecution:
   - weight: 100
    preference:
     matchExpressions:
     - key: region
      operator: In
      values:
      - ap-south-1
     - key: zone
      operator: In
      values:
      - ap-south-1a
 containers:
 - name: nginx
```

image: nginx

Labeling Nodes for Node Affinity

Before using node affinity, label your nodes:

kubectl label nodes <node-id> region=ap-south-1 zone=ap-south-1c

kubectl label nodes <node-id> region=ap-south-1 zone=ap-south-1a

Key Differences: Node Selector vs Node Affinity

Feature	Node Selector	Node Affinity
Flexibility	Strict match	Can be required or preferred
Syntax	Simple key-value	More advanced expressions (In, NotIn, Exists, etc.)
Use case	Basic scheduling	Advanced scheduling (zones, regions, preferences)

Summary

- Node Selector: Simple way to schedule pods to specific labeled nodes.
- Node Affinity: Advanced, flexible scheduling, supports soft/hard rules and multi-node constraints.
- Labeling nodes is a prerequisite for both approaches.

Chapter 19: Pod Affinity & Anti-Affinity

In Kubernetes, controlling **where your pods run** is crucial for performance, availability, and reliability.

Two powerful scheduling features help you achieve this:

- Pod Affinity
- Pod Anti-Affinity

Let's break them down in simple terms with practical examples.

Why Do We Need Affinity Rules?

By default, Kubernetes schedules pods automatically based on resource availability.

But in real-world production environments, you may want to:

- Run certain pods close together
- Separate certain pods across nodes
- Improve performance
- Improve high availability
- Reduce network latency

That's where affinity and anti-affinity come in.

What is Pod Affinity?

Pod Affinity tells Kubernetes:

☞ "Schedule this pod on a node where specific other pods are already running."

✔ Example Use Case

- Run backend pods close to database pods to reduce network latency.
- Keep microservices that communicate frequently on the same node.

Pod Affinity Example (YAML)

apiVersion: v1

kind: Pod

metadata:

name: frontend-pod

spec:

affinity:

podAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

- labelSelector:

matchExpressions:

- key: app

operator: In

values:

- backend

topologyKey: "kubernetes.io/hostname"

containers:

- name: nginx

image: nginx

What This Does

- The pod will only be scheduled on a node

```
Where another pod with label app=backend is running
On the same node (kubernetes.io/hostname)
```

What is Pod Anti-Affinity?

Pod Anti-Affinity tells Kubernetes:

👉 "Do NOT schedule this pod on a node where certain other pods are running."

✔ Example Use Case

- Spread replicas across multiple nodes for high availability.
- Avoid placing two replicas of the same app on the same node.

Pod Anti-Affinity Example (YAML)

apiVersion: apps/v1

kind: Deployment

metadata:

name: web-app

spec:

replicas: 3

selector:

```
  matchLabels:
    app: web
template:
  metadata:
    labels:
      app: web
  spec:
    affinity:
      podAntiAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
                - key: app
                  operator: In
                  values:
                    - web
            topologyKey: "kubernetes.io/hostname"
    containers:
      - name: nginx
        image: nginx
```

What This Does

```
Ensures replicas of app=web
```

- Do NOT run on the same node
- Improves availability

Required vs Preferred Rules

There are two types of affinity rules:

1️⃣ requiredDuringSchedulingIgnoredDuringExecution

- Hard rule
- Pod will NOT schedule if condition is not met

2️⃣ preferredDuringSchedulingIgnoredDuringExecution

- Soft rule
- Scheduler tries to follow it
- But may ignore if necessary

Example of Preferred Rule

```
preferredDuringSchedulingIgnoredDuringExecution:
 - weight: 100
   podAffinityTerm:
    labelSelector:
      matchLabels:
        app: backend
    topologyKey: "kubernetes.io/hostname"
```

Understanding topologyKey

```
topologyKey defines the level at which rules apply.
```

Common values:

Topology Key	Meaning
kubernetes.io/hostname	Same node
topology.kubernetes.io/zone	Same availability zone
topology.kubernetes.io/region	Same region

Real Production Example

Scenario: High Availability Web App

You have:

- 3 replicas of a web application
- 3 worker nodes

With Pod Anti-Affinity:

- Each replica runs on a different node
- If one node crashes → Only 1 pod is affected
- App remains available

Without Anti-Affinity:

- All 3 replicas might run on the same node
- Node failure = full outage

Important Considerations

- Works based on labels
- Can slow scheduling in large clusters
- Hard rules may cause pods to stay Pending
- Always test before production

Pod Affinity vs Pod Anti-Affinity

Feature	Pod Affinity	Pod Anti-Affinity
Purpose	Keep pods together	Spread pods apart
Improves	Performance	High availability
Use case	Microservices communication	Replica distribution

Final Thoughts

Pod Affinity and Pod Anti-Affinity give you advanced control over pod placement.

Use:

- Affinity for performance optimization
- Anti-Affinity for reliability and high availability

Mastering these concepts is essential for production-grade Kubernetes deployments.

Chapter 20: Taints & Tolerations

Introduction

In a Kubernetes cluster, not all nodes are equal. Some nodes may be:

- Reserved for high-memory workloads
- GPU-enabled for AI/ML jobs
- Dedicated to system services or monitoring

Kubernetes provides a powerful mechanism called Taints and Tolerations to control which pods can run on which nodes. This ensures pods are scheduled only on appropriate nodes.

What is a Taint?

A taint is applied to a node to repel pods that do not explicitly tolerate it.

Think of a taint as a "Keep Out" sign on a node.

Taint Structure

key=value:effect

Effect Types:

Effect	Description
NoSchedule	Pods that do not tolerate the taint cannot be scheduled on the node
PreferNoSchedule	Scheduler tries to avoid the node but can still schedule if necessary
NoExecute	Pods that do not tolerate will be evicted if already running, and new pods are prevented from scheduling

Example: Adding a Taint

kubectl taint nodes worker-node-1 dedicated=gpu:NoSchedule

Explanation:

```
Node worker-node-1 is tainted
```

- Only pods with a matching toleration can be scheduled here
- All other pods will be blocked

What is a Toleration?

A toleration is applied to a pod to allow it to be scheduled on nodes with matching taints.

Think of a toleration as a “Permission Pass” allowing the pod to enter a tainted node.

Pod YAML Example with Toleration

apiVersion: v1

kind: Pod

metadata:

name: gpu-pod

spec:

tolerations:

- key: "dedicated"

operator: "Equal"

value: "gpu"

effect: "NoSchedule"

containers:

- name: gpu-app

image: nvidia/cuda

Explanation:

```
This pod can now run on nodes tainted with dedicated=gpu:NoSchedule
```

- Pods without this toleration will not schedule on the node

Key Points About Taints and Tolerations

- Taints are applied to nodes to repel pods.
- Tolerations are applied to pods to allow them to be scheduled on tainted nodes.

```
Pods without matching tolerations cannot run on a node with a NoSchedule taint.
```

- Taints and tolerations do not control network traffic, they only affect pod scheduling.

Real-World Use Cases

- GPU Nodes – Only GPU workloads can run on GPU nodes.
- High-Memory Nodes – Critical workloads get dedicated memory.
- Control-Plane Nodes – System pods are isolated from regular workloads using taints.

Example:

kubectl taint nodes control-plane node-role.kubernetes.io/control-plane=:NoSchedule

Only system pods with tolerations can run on control-plane nodes.

Taints vs Node Affinity

Feature	Applies To	Purpose
Taints	Node	Repel pods from nodes
Tolerations	Pod	Allow pods to schedule on tainted nodes
Node Affinity	Pod	Prefer or require nodes with certain labels

💡 Difference: Node affinity attracts pods to nodes with labels; taints repel pods unless tolerated.

Best Practices

- Use taints for nodes dedicated to special workloads
- Use tolerations only on pods that need to run on tainted nodes
- Avoid over-tainting to prevent scheduler starvation
- Combine with node affinity for more precise scheduling

Summary

- Taints = “Keep out” signs on nodes
- Tolerations = Pods’ permission to ignore taints

```
Effect types control scheduling behavior (NoSchedule, PreferNoSchedule, NoExecute)
```

- Core use case: isolate nodes for specific workloads like GPU, high-memory, or system pods
- Example Workflow
  - Taint a GPU node:

kubectl taint nodes gpu-node dedicated=gpu:NoSchedule

- Add a toleration to a GPU workload pod:

tolerations:

```
  - key: "dedicated"
    operator: "Equal"
    value: "gpu"
    effect: "NoSchedule"
```

- The pod now schedules only on the GPU node. Other pods are blocked.

Chapter 21: Cordon, Drain & Uncordon

When managing a Kubernetes cluster, there are times when you need to take a node out of service — maybe for maintenance, upgrades, or troubleshooting.

Kubernetes provides three important commands for this:

- Cordon
- Uncordon
- Drain

Let's understand what each one does and when to use them.

What is Cordon in Kubernetes?

Cordon marks a node as **unschedulable**, meaning:

- ✅ Existing pods continue running
- ❌ No new pods will be scheduled on that node

Command:

kubectl cordon <node-name>

Example:

kubectl cordon worker-node-1

When to Use Cordon?

- Before performing maintenance
- Before upgrading the node
- When troubleshooting node issues
- When you want to gradually move workloads

After cordoning, you can verify:

kubectl get nodes

```
You'll see SchedulingDisabled next to the node.
```

What is Uncordon?

Uncordon makes a node schedulable again.

Command:

kubectl uncordon <node-name>

Example:

kubectl uncordon worker-node-1

Now Kubernetes can start placing new pods on this node again.

What is Drain in Kubernetes?

Drain safely removes all running pods from a node and evicts them.

It does two things:

- Cordon the node
- Evict the pods (except DaemonSets and mirror/static pods)

Command:

kubectl drain <node-name> --ignore-daemonsets --delete-emptydir-data

Example:

kubectl drain worker-node-1 --ignore-daemonsets --delete-emptydir-data

Important Flags:

Flag	Purpose
--ignore-daemonsets	Required because DaemonSet pods cannot be deleted normally
--delete-emptydir-data	Deletes pods using emptyDir volumes
--force	Forces deletion (use carefully)

Real-World Maintenance Workflow

Here's a common production workflow:

Step 1: Cordon the node

kubectl cordon worker-node-1

Step 2: Drain the node

kubectl drain worker-node-1 --ignore-daemonsets --delete-emptydir-data

Step 3: Perform maintenance

- OS patching
- Security updates
- Kernel upgrade
- Hardware fix

Step 4: Bring node back

kubectl uncordon worker-node-1

Things to Be Careful About

- Make sure your workloads use ReplicaSets or Deployments

- Ensure proper Pod Disruption Budgets (PDB)
- Never drain all nodes at once
- Always check cluster health before and after

Key Differences

Feature	Cordon	Drain	Uncordon
Stops new pods	✔	✔	✖
Removes existing pods	✖	✔	✖
Used for maintenance	✔	✔	✖ (used after maintenance)

Final Thoughts

Understanding **cordon, drain, and uncordon** is essential for any Kubernetes administrator. These commands help ensure zero downtime maintenance and safe workload migration.

Chapter 22: Reclaim Policies & Network Policies

Kubernetes provides mechanisms to manage persistent data and control network traffic in a cluster. Two important concepts are PersistentVolume reclaim policies and NetworkPolicies.

This blog explains both topics clearly, with examples.

1️⃣ PersistentVolume Reclaim Policies

When a PersistentVolumeClaim (PVC) or pod is deleted in Kubernetes, what happens to the data? This is controlled by the PersistentVolumeReclaimPolicy.

Types of Reclaim Policies

Kubernetes supports three types of reclaim policies:

Policy	Description	Usage
Retain	Keeps the data even after PVC or pod deletion	Default policy; widely used in production
Delete	Deletes the data along with PVC or pod	Can lead to data loss; used only when cleanup is intended
Recycle / Repolicy	Deprecated	Not recommended; do not use

How It Works

- Retain: Safely stores data even if the pod or PVC is deleted.
- Delete: Removes all associated data when PVC or pod is deleted.
- Deprecated Recycle/Repolicy: Old method; data will be deleted.

YAML Example: PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: app-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
```

nfs:

server: <NFS_SERVER_IP>

path: /mnt/nfs_area

persistentVolumeReclaimPolicy: Retain

```
The persistentVolumeReclaimPolicy can be set to Retain, Delete, or Recycle
(deprecated).
```

2 Network Policies in Kubernetes

Network policies allow you to control inbound and outbound traffic to pods. They provide security and traffic isolation within a cluster.

Kubernetes supports two types of network traffic rules:

- Ingress: Controls incoming traffic to a pod
- Egress: Controls outgoing traffic from a pod

Ingress Example

Allow inbound traffic to a MongoDB pod only from specific pods:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: app-ingress

namespace: prod

spec:

podSelector:

matchLabels:

app: mongodb

policyTypes:

- Ingress

ingress:

- from:

- podSelector:

matchLabels:

app: app

- podSelector:

matchLabels:

app: app

ports:

- protocol: TCP

port: 27017

Explanation:

```
podSelector and matchLabels identify which pods are allowed to send traffic.
ports specify which ports are open for communication.
policyTypes: Ingress ensures it only applies to inbound traffic.
```

Egress Example

Allow outbound traffic from a pod to a specific IP range:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: app-egress

namespace: prod

spec:

podSelector:

matchLabels:

app: app

policyTypes:

- Egress

egress:

- to:

- ipBlock:

cidr: 10.1.0.0/36

ports:

- protocol: TCP

port: 80

Explanation:

```
policyTypes: Egress applies the rule to outgoing traffic.
Only allows communication to the specified IP range (10.1.0.0/36) on TCP port 80.
```

Key Takeaways

PersistentVolume Reclaim Policies

- Retain: Keep data safe after PVC/pod deletion.
- Delete: Remove data when PVC/pod is deleted.
- Recycle/Repolicy: Deprecated, avoid using.

Network Policies

- Ingress: Control inbound traffic to pods.
- Egress: Control outbound traffic from pods.
- Use pod selectors, labels, ports, and IP blocks to define rules

Chapter 23: Kubernetes RBAC

Understanding RBAC in Kubernetes: Role-Based Access Control

In Kubernetes, RBAC stands for Role-Based Access Control. It is a critical feature that allows you to manage and restrict access to cluster resources in a secure and organized way.

RBAC helps ensure that users and applications only have the permissions they need, which prevents unauthorized access and improves security.

Why RBAC Matters

When running applications in Kubernetes, you often need to:

- View pod status and logs
- Deploy applications
- Access services or tools

Without RBAC, you cannot control who can do these actions, leaving your cluster vulnerable to misuse or hacking.

With RBAC, you can:

- Assign permissions securely
- Restrict access at namespace or cluster level
- Provide the least privilege required for users

RBAC Components in Kubernetes

RBAC in Kubernetes has four main components:

Component	Level	Purpose
Role	Namespace	Defines permissions within a namespace
RoleBinding	Namespace	Grants permissions defined in a Role to a user or group
ClusterRole	Cluster	Defines permissions at the cluster level
ClusterRoleBinding	Cluster	Grants cluster-level permissions to a user or group

1 Role

- Namespace-level object
- Sets permissions for resources in a specific namespace

- Does not grant access by itself

Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: app-role
  namespace: prod
rules:
  - apiGroups: [""]
    resources: ["pods", "services"]
    verbs: ["get", "list", "watch"]
```

2 RoleBinding

- Namespace-level object
- Grants permissions defined in a Role to users, groups, or service accounts

Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: app-rolebinding
  namespace: prod
subjects:
  - kind: User
    name: sai
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: app-role
  apiGroup: rbac.authorization.k8s.io
```

3 ClusterRole

- Cluster-level object
- Sets permissions across the entire cluster
- Useful for cluster-wide resources or admin tasks

Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: fullaccess-cr
rules:
  - apiGroups: [""]
    resources: ["*"]
    verbs: ["*"]
```

4 ClusterRoleBinding

- Cluster-level object
- Grants cluster-wide permissions defined in a ClusterRole to users, groups, or service accounts

Example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: fullaccess-binding
subjects:
  - kind: User
    name: sai
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: fullaccess-cr
  apiGroup: rbac.authorization.k8s.io
```

Key Takeaways

- Roles and RoleBindings are namespace-level, controlling access within a specific namespace.
- ClusterRoles and ClusterRoleBindings are cluster-level, controlling access across the entire cluster.
- RBAC allows you to secure your cluster, enforce least privilege, and prevent unauthorized actions.
- Always follow the principle of least privilege: give only the access a user needs.

Chapter 24: Ingress Controller & Ingress Resource

In Kubernetes, managing traffic to multiple applications efficiently is critical, especially when running many microservices. Ingress Controllers provide a solution by allowing multiple application routes to be handled through a single load balancer, instead of creating one load balancer per service.

Why Use an Ingress Controller?

If you create a LoadBalancer service for every application in AWS or another cloud, it can quickly become:

- Expensive (each load balancer costs money)
- Hard to manage
- Limited by cloud provider quotas

An Ingress Controller solves this by acting as a traffic router. You define rules in Ingress resources, and the controller applies them to route traffic to the correct service.

Traffic Flow Example:

User Request → Cloud LoadBalancer → Ingress Controller Pod → Service → Application Pods

This allows multiple applications to share one external IP, saving costs and simplifying DNS management.

Installing NGINX Ingress Controller in EKS

You can install NGINX Ingress using Helm:

Step 1 — Add Helm Repository

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

helm repo update

Step 2 — Install NGINX Ingress

helm install nginx-ingress ingress-nginx/ingress-nginx \

--namespace ingress-nginx --create-namespace \

--set controller.service.type=LoadBalancer

Step 3 — Verify Installation

kubectl get svc -n ingress-nginx

Example Ingress YAML

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress
  namespace: prod
  annotations:
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  ingressClassName: nginx
  rules:
    - http:
        paths:
          - path: /v1(/|$)(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: assvcv1-lb
                port:
                  number: 80
          - path: /v2(/|$)(.*)
            pathType: ImplementationSpecific
            backend:
              service:
                name: assvcv2-lb
                port:
                  number: 80
```

Explanation:

- Ingress Resource: Kubernetes YAML object with routing rules.
- Ingress Controller: Actual software (like NGINX) that applies those rules and handles traffic.

Why Not Use LoadBalancer for Every Service?

Using a LoadBalancer service for each app creates challenges:

Problem	What Actually Happens
High Cost	Each Service = 1 Load Balancer → monthly cost explodes with many microservices
Slow Provisioning	Each LB takes 2–5 minutes to create → slows down CI/CD
AWS LB Limits	Accounts have load balancer quotas → new services can fail
One IP per Service	Every app gets separate public IP → DNS management is harder
Not Smart Routing	No path-based or hostname-based routing → Ingress required

💰 Cost Example:10 microservices × 1 Load Balancer each = $180–$250/month just for load balancers.

Summary

- Ingress Controller: Routes multiple apps through a single load balancer using rules defined in Ingress resources.
- Cost-saving: Avoids creating multiple cloud load balancers.
- Routing: Supports host- and path-based routing for HTTP/S traffic.
- Limitations: Ingress has some constraints; Gateway API addresses these with more flexibility and extensibility.

Chapter 25: Kubernetes Cluster Upgrade

Introduction

Kubernetes (K8s) is constantly evolving, and upgrading your cluster to a newer version is essential for:

- Security patches
- New features
- Performance improvements
- Bug fixes

For example, if your cluster is running v1.33 and you want to upgrade to v1.35, you need to follow a safe upgrade process to ensure no downtime and no data loss.

Step 1: Check Current Cluster Version

Before upgrading, verify your cluster version:

kubectl version --short

- This will show the client version and server version
- Confirm that your cluster is currently running v1.33

Step 2: Verify Running Pods

Before upgrading:

- List all running pods in all namespaces:

kubectl get pods --all-namespaces

- Ensure you know which workloads are running
- Note down critical pods (stateful sets, deployments, databases)

This helps prevent accidental data loss during the upgrade.

Step 3: Understand How Upgrade Works

When upgrading Kubernetes in AWS (or any managed platform):

- Nodes are upgraded one by one
- Existing pods are safely moved to new nodes automatically
- Pod IDs may change, but data is preserved
- Managed services like AWS EKS handle node rotation and scheduling automatically

Key Points:

- No downtime for running applications
- StatefulSets, Deployments, and ReplicaSets are handled safely
- The process may take hours or even days depending on cluster size

Step 4: Example Upgrade Scenario

Current Cluster

- Version: v1.33
- Running Pods:

Pod Name	Description
pod-1a	Frontend application
pod-abc	Backend service
pod-cds	Database
pod-fesd	Cache service
pod-afsv	Monitoring agent

- Nodes:

Node ID	Application
node1	My application pods
node2	My application pods

Upgrade Process

- Start cluster upgrade to v1.35
- Nodes are upgraded one by one
  - Pods are safely evicted and rescheduled on other nodes
  - StatefulSets are upgraded pod by pod
- After node upgrade, new pods start on upgraded nodes
- Applications continue running without downtime

Advantages of This Approach

- Zero downtime: Applications remain available during upgrade
- Data safety: Persistent data is preserved during pod rescheduling
- Controlled upgrade: Nodes and pods are upgraded sequentially

Step 5: Post-Upgrade Verification

After the upgrade:

- Verify cluster version:

kubectl version --short

- Confirm all nodes are upgraded to v1.35:

kubectl get nodes

- Check all pods are running:

kubectl get pods --all-namespaces

- Test critical workloads to ensure functionality

Notes for Large vs. Small Teams

- In large companies, cluster upgrades are usually handled by an infrastructure team
- In startups, DevOps engineers often perform the upgrade themselves
- Upgrades may take hours or days, depending on cluster size and complexity

Summary

Upgrading a Kubernetes cluster is a critical but manageable process:

- Always check current version and running pods
- Upgrade nodes sequentially to ensure safety
- Pod rescheduling ensures no downtime
- Always verify all workloads after the upgrade

By following this approach, you can safely move from v1.33 to v1.35 (or any other version) while keeping your applications running smoothly.

Chapter 26: Cluster Autoscaler on AWS EKS

Introduction

In a production Kubernetes cluster, ensuring that your workloads always have sufficient resources is critical.

Sometimes, a node may crash or become overloaded. Without additional nodes, pods running on that node may fail. To solve this, Kubernetes provides the Cluster Autoscaler, which automatically adds or removes nodes based on workload demand.

Using AWS EKS (Elastic Kubernetes Service), the Cluster Autoscaler can safely manage node scaling while keeping your applications running with zero downtime.

What is Cluster Autoscaler?

Cluster Autoscaler (CA) automatically adjusts the size of your Kubernetes cluster:

- Scale up: Adds nodes when there are pending pods that cannot be scheduled due to resource constraints.
- Scale down: Removes underutilized nodes to save costs.

Key Benefits:

- Ensures pods are always scheduled
- Prevents node resource exhaustion
- Automatically manages cluster capacity without manual intervention

Prerequisites

- EKS Cluster running on AWS
- kubectl configured for your cluster
- IAM permissions for the autoscaler
- Service account and RBAC configured for Cluster Autoscaler

Step 1: Create IAM Policy for Cluster Autoscaler

The Cluster Autoscaler requires specific permissions to manage EC2 instances and Auto Scaling groups.

- Go to AWS → IAM → Policies → Create Policy
- Select JSON and paste the following:

```
{
 "Version": "2012-10-17",
 "Statement": [
  {
   "Effect": "Allow",
```

```
    "Action": [
      "autoscaling:DescribeAutoScalingGroups",
      "autoscaling:DescribeAutoScalingInstances",
      "autoscaling:DescribeLaunchConfigurations",
      "autoscaling:DescribeScalingActivities",
      "ec2:DescribeImages",
      "ec2:DescribeInstanceTypes",
      "ec2:DescribeLaunchTemplateVersions",
      "ec2:GetInstanceTypesFromInstanceRequirements",
      "eks:DescribeNodegroup"
    ],
    "Resource": ["*"]
  },
  {
    "Effect": "Allow",
    "Action": [
      "autoscaling:SetDesiredCapacity",
      "autoscaling:TerminateInstanceInAutoScalingGroup"
    ],
    "Resource": ["*"]
  }
 ]
}
```

- Name the policy ClusterAutoScalerPolicy and create it.

Step 2: Attach IAM Policy to EKS Node Role

- Go to AWS → EKS → Cluster → Compute → Node IAM Role
- Attach ClusterAutoScalerPolicy to the role
- This allows the autoscaler to manage nodes in the cluster

Step 3: Deploy Cluster Autoscaler

Cluster Autoscaler runs as a deployment in the kube-system namespace with a service account and RBAC permissions.

Service Account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
```

ClusterRole

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-autoscaler
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
rules:
- apiGroups: [""]
  resources: ["events", "endpoints"]
  verbs: ["create", "patch"]
- apiGroups: [""]
  resources: ["pods/eviction"]
  verbs: ["create"]
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["watch", "list", "get", "update"]
- apiGroups: ["apps"]
  resources: ["statefulsets", "replicasets", "daemonsets"]
  verbs: ["watch", "list", "get"]
- apiGroups: ["batch", "extensions"]
```

```
  resources: ["jobs"]
  verbs: ["get", "list", "watch", "patch"]
```

ClusterRoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-autoscaler
  labels:
    k8s-addon: cluster-autoscaler.addons.k8s.io
    k8s-app: cluster-autoscaler
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-autoscaler
subjects:
- kind: ServiceAccount
  name: cluster-autoscaler
  namespace: kube-system
```

Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cluster-autoscaler
  namespace: kube-system
  labels:
    app: cluster-autoscaler
spec:
  replicas: 1
  selector:
    matchLabels:
      app: cluster-autoscaler
```

```
  template:
    metadata:
      labels:
        app: cluster-autoscaler
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "8085"
    spec:
      serviceAccountName: cluster-autoscaler
      containers:
      - name: cluster-autoscaler
        image: registry.k8s.io/autoscaling/cluster-autoscaler:v1.26.2
        imagePullPolicy: Always
        resources:
          limits:
            cpu: 100m
            memory: 600Mi
          requests:
            cpu: 100m
            memory: 600Mi
        command:
        - ./cluster-autoscaler
        - --v=4
        - --stderrthreshold=info
        - --cloud-provider=aws
        - --skip-nodes-with-local-storage=false
        - --expander=least-waste
        - --node-group-auto-discovery=asg:tag=k8s.io/cluster-autoscaler/enabled,k8s.io/cluster-
autoscaler/<YOUR_CLUSTER_NAME>
        securityContext:
          allowPrivilegeEscalation: false
```

```
        readOnlyRootFilesystem: true
      volumeMounts:
      - name: ssl-certs
        mountPath: /etc/ssl/certs/ca-certificates.crt
        readOnly: true
    volumes:
    - name: ssl-certs
      hostPath:
        path: /etc/ssl/certs/ca-bundle.crt
```

```
Replace <YOUR_CLUSTER_NAME> with your actual EKS cluster name.
```

How Cluster Autoscaler Works

- Detects pending pods: If a pod cannot be scheduled due to insufficient resources, CA adds a new node.
- Scales down underutilized nodes: If nodes are mostly empty, CA removes them to save cost.
- Automatic rescheduling: Pods from terminated nodes are safely moved to other nodes.
- Zero downtime: Applications continue running while the cluster adjusts automatically.

Summary

Cluster Autoscaler in AWS EKS is an essential tool to:

- Automatically scale worker nodes based on demand
- Safely move pods during node failures
- Reduce manual operations and costs
- Ensure high availability for workloads

With proper IAM policies, RBAC, and deployment, your EKS cluster can self-manage capacity efficiently.

Chapter 27: Helm Charts in Kubernetes

Helm charts are widely used in real-time Kubernetes environments to deploy applications securely and efficiently. One of the biggest advantages of Helm is reusability—we can deploy the same application multiple times by simply changing parameters instead of rewriting manifests.

In production environments, Helm charts help reduce manual errors, standardize deployments, and simplify application management.

What is a Helm Chart?

A Helm chart is a collection of YAML files that define, install, and manage a Kubernetes application. It allows us to template Kubernetes manifests and customize deployments using a single configuration file.

By using Helm charts:

- We avoid repetitive YAML writing
- We reduce bugs and misconfigurations
- We can deploy the same application across multiple environments (dev, QA, prod)

Helm Chart Structure

A typical Helm chart contains the following important files:

- Chart.yaml
- values.yaml
- templates/ (folder containing Kubernetes manifests like Deployment, Service, PVC, etc.)

1. Chart.yaml

```
The Chart.yaml file defines the metadata of the application.
Example Chart.yaml
```

apiVersion: v2

name: my-application

description: "My application"

type: application

version: 1.0.0

appVersion: 1.0.0

Explanation:

```
apiVersion: v2We use v2 because Helm 3 supports and requires API version v2.
```

- nameName of the Helm chart (used during deployment).
- descriptionShort explanation of the application.
- type: applicationSpecifies this chart deploys an application (not a library chart).
- versionHelm chart version (used for chart upgrades).

```
appVersionApplication version.Version format: MAJOR.MINOR.PATCH
```

- Major: breaking changes
- Minor: feature updates
- Patch: bug fixes

This file is mainly used to define application metadata.

2. values.yaml

```
The values.yaml file is the heart of the Helm chart.Here, we define all
configurable values used by the templates.
Example values.yaml
```

replicas: 2

image:

repository: dilipdara/image1

tag: 1.0.0

pullPolicy: IfNotPresent

service:

type: ClusterIP

port: 8080

resources:

requests:

cpu: "3"

memory: "512Mi"

limits:

cpu: "3"

memory: "600Mi"

volumeMount:

enabled: true

```
  name: mynews
  path: /data/db
volumes:
  name: persistentvolumeclaim
  nfs:
    server: <IP_ADDRESS>
    path: /mnt/nfs_share
env:
  enabled: true
  username: ""
secretsKeyRef:
  db_password
```

```
  What we define in values.yaml:
```

- Replicas
- Image details
- Service type and ports
- Resource requests and limits
- Volumes and volume mounts
- Environment variables
- Secrets
- ConfigMaps
- Probes
- HPA
- PV / PVC

Instead of hardcoding values in manifests, we centralize configuration in this file.

3. templates directory

```
  The templates/ directory contains Kubernetes manifest templates such as:
```

- Deployment
- Service
- ConfigMap
- Secret
- PV / PVC
- HPA

```
These templates use values from values.yaml.
```

Example: Deployment Template

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: {{ .Values.name }}
 namespace: prod
 labels:
  app: app
  env: prod
spec:
 replicas: {{ .Values.replicas }}
 selector:
  matchLabels:
   app: app
 template:
  metadata:
   labels:
    app: app
  spec:
   containers:
    - name: {{ .Values.name }}
      image: {{ .Values.image.repository }}:{{ .Values.image.tag }}
      ports:
       - containerPort: {{ .Values.service.port }}
```

Example: Service Template

```
apiVersion: v1
kind: Service
metadata:
```

```
  name: {{ .Values.name }}
  namespace: prod
spec:
  type: {{ .Values.service.type }}
  selector:
    app: app
  ports:
    - port: {{ .Values.service.port }}
```

We can also add:

- Environment variables
- Volumes
- Volume mounts
- Resources
- Secrets
- PV / PVC

```
All values come from values.yaml, making the templates clean and reusable.
```

Why Helm Charts Are Used in Real Time

- Reduces bugs and manual mistakes
- Reusable across multiple projects

```
Easy configuration changes using values.yaml
```

- Simplifies Kubernetes deployments
- Supports versioning and rollback

By changing only the values file, we can deploy the same Helm chart to multiple environments or projects.

Conclusion

```
Helm charts are a best practice for Kubernetes deployments.They help teams maintain
consistency, reduce errors, and deploy applications faster.By separating
configuration (values.yaml) from logic (templates/), Helm makes Kubernetes
manageable at scale.
```

Chapter 28: Canary Deployment

Introduction

When deploying applications in Kubernetes, one of the biggest concerns is avoiding downtime during updates.

There are several deployment strategies:

- Recreate Deployment – Shuts down old pods and starts new pods. This causes downtime.
- Rolling Update Deployment – Gradually replaces old pods with new pods, avoiding downtime.
- Canary Deployment – Gradually shifts traffic from old version pods to new version pods, minimizing risk and downtime.

This blog focuses on Canary Deployment and how it works in production.

What is Canary Deployment?

Canary Deployment is a strategy where a new version of the application is deployed gradually alongside the old version.

- A small percentage of traffic is initially routed to the new version.
- The rest of the traffic continues to flow to the old version.
- If no issues are detected, traffic to the new version is gradually increased until it fully replaces the old version.

Think of it as testing the new version with a few users first, like a “canary in a coal mine.”

How Canary Deployment Works

Example Scenario:

- Old version: 10 pods running
- New version: 10 pods ready to be deployed

Traffic distribution can be controlled in steps:

Old Version	New Version
90%	10%
80%	20%
70%	30%
60%	40%
50%	50%
40%	60%
30%	70%
20%	80%
10%	90%

0%	100%

- The new version gradually receives more traffic
- Old version receives less traffic until it is completely replaced
- No downtime occurs, and user experience is uninterrupted

Benefits of Canary Deployment

- Zero downtime – Users continue using the old version while the new version is being tested.
- Reduced risk – Any issues in the new version affect only a small portion of users.
- Safe rollbacks – If errors are detected, traffic can be shifted back to the old version immediately.
- Gradual load handling – Avoids sudden spikes that can destabilize the system.

Example: Kubernetes Deployment YAML for Canary

Here's a simple illustration of how you can implement a canary deployment using two deployments:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: my-app-old
spec:
 replicas: 10
 selector:
  matchLabels:
   app: my-app
   version: v1
 template:
  metadata:
   labels:
    app: my-app
    version: v1
  spec:
   containers:
   - name: my-app
    image: my-app:v1
```

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-new
spec:
  replicas: 1   # Start with 10% of traffic
  selector:
    matchLabels:
      app: my-app
      version: v2
  template:
    metadata:
      labels:
        app: my-app
        version: v2
    spec:
      containers:
      - name: my-app
        image: my-app:v2
```

- Start with 1 pod (10% traffic) for the new version
- Gradually increase the replicas to shift more traffic to the new version
- Eventually, scale old version down to 0

Summary

Canary Deployment is a safe, progressive rollout strategy that ensures:

- No downtime during updates
- Gradual traffic shift from old to new version
- Quick rollback if issues occur
- Smooth user experience even during patches

This method is widely used in production environments to minimize risk while deploying new features or bug fixes.

Chapter 29: Blue-Green Deployment

Introduction

In production environments, application downtime is unacceptable.

Kubernetes offers deployment strategies to update applications safely, and Blue-Green Deployment is one of the most effective methods for zero-downtime releases.

This strategy allows you to switch traffic between two identical environments — Blue (current version) and Green (new version) — instantly and safely.

What is Blue-Green Deployment?

Blue-Green Deployment is a release strategy where:

- The Blue environment represents the current live version of the application.
- The Green environment is a new version deployed alongside the Blue environment.
- Traffic is switched from Blue to Green only after testing and verification.

Key Advantages:

- No downtime during deployment
- Immediate rollback if issues occur
- Fully isolated new version for testing

How It Works

Example Scenario:

- Current live version (Blue) has 10 pods running
- New version (Green) also deploys 10 pods

Step-by-step process:

- Deploy the Green environment alongside Blue.
- Test Green internally without affecting users.
- Switch traffic from Blue to Green (using service selector or Ingress routing).
- Monitor Green for any issues.
- Keep Blue idle for rollback if needed.

Traffic Flow Visualization:

Step	Blue (Old Version)	Green (New Version)
Before switch	100%	0%
After switch	0%	100%

Blue-Green Deployment in Kubernetes

Kubernetes handles traffic routing using Services. You can update the Service selector to point from Blue pods to Green pods instantly.

Step 1: Deploy Blue (Current Version)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-blue
spec:
  replicas: 10
  selector:
    matchLabels:
      app: my-app
      version: blue
  template:
    metadata:
      labels:
        app: my-app
        version: blue
    spec:
      containers:
      - name: my-app
        image: my-app:v1
---
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
    version: blue
```

```
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

- Service routes all traffic to Blue pods

Step 2: Deploy Green (New Version)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-green
spec:
  replicas: 10
  selector:
    matchLabels:
      app: my-app
      version: green
  template:
    metadata:
      labels:
        app: my-app
        version: green
    spec:
      containers:
      - name: my-app
        image: my-app:v2
```

- Green runs in parallel with Blue
- No user traffic goes to Green yet

Step 3: Switch Traffic from Blue to Green

Update the Service selector:

```
kubectl patch service my-app-service -p '{"spec":{"selector":{"app":"my-app","version":"green"}}}'
```

- Traffic is instantly routed to Green pods

- Blue pods are idle but available for rollback

Step 4: Optional Rollback

If any issues occur with Green:

kubectl patch service my-app-service -p '{"spec":{"selector":{"app":"my-app","version":"blue"}}}'

- Traffic is instantly switched back to Blue
- Minimal impact on users

Advantages of Blue-Green Deployment

- Zero downtime – Users are never affected during deployment.
- Quick rollback – Switch back to previous version instantly.
- Safe testing – New version is fully isolated from live traffic.
- Simplified operations – Traffic routing is handled at the Service level.

Blue-Green vs Canary Deployment

Feature	Blue-Green	Canary
Traffic Routing	100% switch from old to new	Gradual traffic shift
Risk	Low	Very low (gradual)
Rollback	Instant	Gradual (adjust traffic back)
Use Case	Major version updates	Minor patches, new features

Summary

Blue-Green Deployment in Kubernetes provides a safe and reliable way to release new versions:

- Deploy new version in parallel
- Switch traffic instantly after testing
- Rollback if necessary
- Achieve zero downtime during deployment

It is an essential strategy for production-grade applications where availability and reliability are critical.

Chapter 30: Deploying Application with Database — YAML Guide

In Kubernetes, YAML files are used to define objects like Pods, Deployments, Services, ConfigMaps, Secrets, PersistentVolumes (PV), and PersistentVolumeClaims (PVC). Writing these YAMLs requires understanding key concepts like metadata, labels, selectors, containers, images, ports, namespaces, volumes, and resource requests.

Key Concepts to Know Before Writing YAMLs

- apiVersion & kind: Specify the Kubernetes API version and resource type (e.g., Deployment, Service, ConfigMap).
- metadata: Defines the resource name, namespace, and labels.
- labels & selectors: Labels categorize objects; selectors match labels to connect resources (e.g., Services to Pods).
- spec: Defines the desired state, such as replicas, container details, ports, volumes, etc.
- namespace: Logical partitioning of cluster resources.
- resources: CPU and memory requests/limits for containers.
- volumes and volumeMounts: Manage persistent storage.
- ConfigMap and Secrets: Store configuration data and sensitive info (e.g., passwords) without hardcoding.

Example: Deploying an Application and MongoDB with YAML

Below is a complete example of the YAML files needed to deploy a sample application and a MongoDB database with secure configuration and persistent storage.

1. Deployment for Application (e.g., Nginx-based app)

```
apiVersion: apps/v1

kind: Deployment

metadata:

 name: application

 namespace: prod

 labels:

  app: app

  env: prod

  service: ingress

spec:

 replicas: 2
```

```
selector:
  matchLabels:
    app: app
template:
  metadata:
    labels:
      app: app
  spec:
    containers:
    - name: application
      image: nginx:latest
      ports:
      - containerPort: 80
      env:
      - name: MONGO_DB_HOSTNAME
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: db_hostname
      - name: MONGO_DB_USERNAME
        valueFrom:
          configMapKeyRef:
            name: app-config
            key: db_username
      - name: MONGO_DB_PASSWORD
        valueFrom:
          secretKeyRef:
            name: app-secrets
            key: db_password
      resources:
        requests:
```

```
        cpu: "200m"
        memory: "512Mi"
      limits:
        cpu: "500m"
        memory: "1Gi"
    volumeMounts:
    - name: app-storage
      mountPath: /data/app
  volumes:
  - name: app-storage
    persistentVolumeClaim:
      claimName: app-pvc
```

Explanation:

- This deployment runs 2 replicas of an Nginx container.
- Environment variables for MongoDB credentials are injected via ConfigMap and Secret to avoid hardcoding sensitive data.
- CPU and memory requests and limits ensure proper resource management.
- Volume mount connects a PVC for persistent storage.

2. Service to Expose Application

```
apiVersion: v1
kind: Service
metadata:
  name: application-service
  namespace: prod
spec:
  type: NodePort
  selector:
    app: app
  ports:
  - port: 80
    targetPort: 80
    nodePort: 30002
```

Explanation:

- This Service exposes the application on port 80 internally and node port 30002 externally.

```
Selector matches pods labeled app: app.
```

- Type NodePort opens access outside the cluster on the specified port.

3. StatefulSet for MongoDB (for stable network identity & storage)

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
 name: applidb
 namespace: prod
spec:
 replicas: 2
 selector:
  matchLabels:
   app: app
 serviceName: "applidb-service"
 template:
  metadata:
   labels:
    app: app
  spec:
   containers:
   - name: mongo
    image: mongo:8.0.9-noble
    ports:
    - containerPort: 27017
    env:
    - name: MONGO_INITDB_ROOT_USERNAME
     valueFrom:
      configMapKeyRef:
```

```
            name: app-config
            key: db_username
        - name: MONGO_INITDB_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: app-secrets
              key: db_password
        volumeMounts:
        - name: mongo-data
          mountPath: /data/db
  volumeClaimTemplates:
  - metadata:
      name: mongo-data
    spec:
      accessModes: ["ReadWriteOnce"]
      resources:
        requests:
          storage: 1Gi
```

Explanation:

- StatefulSet is used for MongoDB for stable pod identities and persistent volumes.
- Environment variables for DB username and password come from ConfigMap and Secret.
- VolumeClaimTemplates dynamically create PVCs for data persistence.

4. ConfigMap for DB Config

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
  namespace: prod
data:
  db_hostname: mongodb.prod.svc.cluster.local
  db_username: devdb
```

Explanation:

- Stores non-sensitive configuration like DB hostname and username.
- Can be updated without redeploying containers.

5. Secret for DB Password

apiVersion: v1

kind: Secret

metadata:

name: app-secrets

namespace: prod

type: Opaque

stringData:

db_password: dev123

Explanation:

- Stores sensitive data securely (passwords, tokens).

```
Use stringData to input plain text which Kubernetes encrypts internally.
```

6. PersistentVolume (PV)

apiVersion: v1

kind: PersistentVolume

metadata:

name: app-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteMany

nfs:

server: <NFS_SERVER_IP>

path: /mnt/nfs_share

Explanation:

- Defines a physical storage resource (NFS in this example).
- Capacity and access modes specify storage size and read/write permissions.

7. PersistentVolumeClaim (PVC)

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: app-pvc
  namespace: prod
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
  storageClassName: ""
```

Explanation:

- PVC requests storage from PV based on size and access mode.
- Bound to PV to provide storage for pods.

Summary

- Start with defining Deployment or StatefulSet with proper container specs.
- Use Services to expose pods inside or outside the cluster.
- Store config data in ConfigMaps and sensitive data in Secrets to avoid hardcoding.
- Use PersistentVolumes and PersistentVolumeClaims for data persistence.
- Use labels and selectors consistently for proper linking between resources

DOCKER

MODULE 1: Docker Basics & Concepts

Chapter 31: Docker Introduction & Virtual Machines

What is Docker?

Docker is an open-source containerization platform that allows developers to package an application along with all its dependencies into a single unit called a container.

By using containers, we can securely deploy applications without worrying about version conflicts or compatibility issues. Docker makes it easy to build, run, and manage applications consistently across different environments.

Unlike traditional virtual machines, Docker containers share the host system's kernel. This makes container images lightweight, faster, and more efficient.

Before Docker: Virtual Machines

Before Docker became popular, organizations mainly used Virtual Machines (VMs).

A virtual machine is software that behaves like a separate physical computer. Each VM:

- Runs its own operating system
- Has its own allocated CPU, RAM, and storage
- Runs inside a hypervisor such as VMware or Oracle VM VirtualBox

Problems with Virtual Machines

- Heavyweight ImagesEach VM contains a full operating system, which increases image size.
- High Resource UsageSince every VM has its own OS, RAM, and storage allocation, resource usage becomes high.
- Slow Startup TimeVMs take longer to boot compared to containers.
- Version and Configuration IssuesApplications often behave differently across development, QA, and production environments.

Example: Memory Usage in Virtual Machines

Imagine a system with 24 GB RAM.

We run 3 VMs:

- VM1: Allocated 8 GB → Uses 3 GB (5 GB unused)
- VM2: Allocated 8 GB → Uses 7 GB (1 GB unused)

- VM3: Allocated 8 GB → Uses 6 GB (2 GB unused)

In total, 8 GB is unused, but this memory cannot be easily shared between VMs because each VM has fixed allocation. Even if one VM needs more memory, it cannot automatically use unused memory from another VM.

This makes VMs inefficient in resource utilization.

Containers vs Virtual Machines

With Docker containers:

- Containers share the host operating system kernel.
- No need for a separate OS for each instance.
- Images are lightweight.
- Faster startup and deployment.
- Better resource utilization.

If we run 3 containers with similar usage, unused resources can be efficiently utilized by other containers because they share the host OS.

Why Choose Docker?

1. Faster Development and Deployment

Developers can build images once and deploy them anywhere.

2. Environment Consistency

The same Docker image can run in:

- Development
- QA
- Pre-production
- Production

No configuration differences. No version issues.

3. Lightweight and Fast

Since containers share the host kernel, they start quickly and consume fewer resources.

4. Isolation

Each container runs as an isolated instance with its own environment.

5. CI/CD Friendly

Docker integrates easily with CI/CD pipelines, making automation simple and efficient.

6. Unique Networking

Each container can have its own networking configuration and IP address.

Before Docker vs After Docker

Before Docker

- Developers write code on local machines.
- Manual deployment to QA/Prod.
- Environment configuration differences.
- Version conflicts.
- Heavy VM images.
- Slow startup.
- High resource usage.

After Docker

- Developers define dependencies in a Dockerfile.
- Build a Docker image once.
- Deploy the same image everywhere.
- Lightweight and fast execution.
- No version conflicts.
- Easier isolation.
- Better CI/CD integration.
- Efficient resource usage.

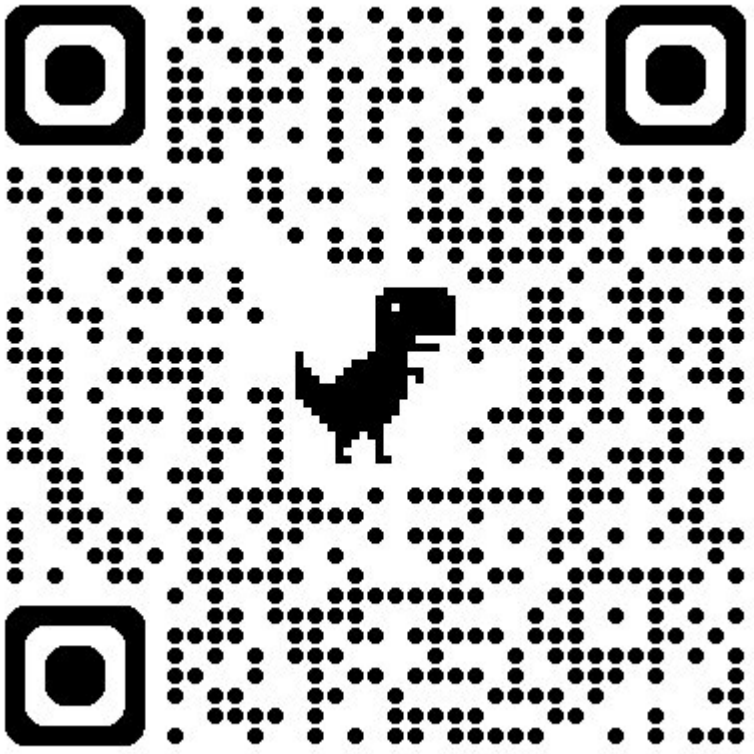
Conclusion

Docker has transformed the way applications are built and deployed. Compared to virtual machines, containers are:

- Faster
- Lightweight
- More efficient
- Easier to manage
- More scalable

That's why most modern applications today use Docker and containerization technologies

Chapter 32: Docker Architecture — Client, Host & Registry



Docker is an open-source containerization platform that allows us to package an application and all its dependencies into a single unit called a container.

To understand how Docker works internally—how it builds images, creates containers, and pushes images—we need to understand Docker Architecture.

Docker architecture mainly consists of three components:

- Docker Client
- Docker Host
- Docker Registry

1 Docker Client

The Docker Client is the primary interface through which users interact with Docker.

It is typically a Command Line Interface (CLI) where we run Docker commands.

For example:

```
docker ps → Lists all running containers
docker images → Lists available images
docker run → Creates and runs a container
```

How It Works

When you run a command like:

docker ps

The Docker client:

- Sends an API request
- Communicates with the Docker Host
- Waits for the response
- Displays the output in the terminal

The client itself does not execute tasks. It simply sends requests to the Docker daemon running on the Docker Host.

2 Docker Host

The Docker Host is the core component of Docker architecture. This is where actual execution happens.

The Docker Host contains:

- Docker Daemon
- Docker Images
- Docker Containers

Docker Daemon

The Docker Daemon (dockerd) is the heart and brain of Docker.

Its responsibilities:

- Listens for API requests from the Docker Client
- Manages images, containers, networks, and volumes
- Pulls images from the registry if not available locally
- Executes commands and sends responses back to the client

When the Docker Client sends a request:

- The daemon checks if the required image exists locally.
- If not found, it connects to a remote registry.
- Pulls the image.
- Creates and runs the container.

- Sends the result back to the client.

The daemon continuously monitors API requests and processes them automatically.

3 Docker Images

A Docker Image is a blueprint for a container.

Key points:

- Images are immutable (cannot be changed after creation).
- Built using a Dockerfile.
- Contain application code, dependencies, libraries, and configurations.

Once an image is built, it can be used to create multiple containers.

4 Docker Containers

A Docker Container is a running instance of a Docker image.

Characteristics:

- Containers are mutable (can change during runtime).
- Multiple containers can be created from a single image.
- Lightweight and fast.
- Designed to be temporary (ephemeral), although they can persist data using volumes.

5 Docker Registry

A Docker Registry is used to store and distribute Docker images.

There are two types of registries:

Local Registry

- Runs within your environment.
- Can be private or public.

Remote Registry

- Hosted externally.
- Used to share images across teams and environments.
- Requires authentication for private repositories.

Common examples:

- Docker Hub (public registry)
- Amazon Elastic Container Registry (private AWS registry)

When an image is not available locally, Docker pulls it from a remote registry.

Complete Flow of Docker Architecture

Let's understand the full workflow:

- Developer writes a Dockerfile.

```
Docker Client runs docker build.
```

- Docker Daemon builds an image.
- Image is stored locally on the Docker Host.
- Image can be pushed to a Docker Registry.

```
When docker run is executed:
```

- The daemon checks for the image.
- Pulls it if not available.
- Creates and starts a container.

Simple Architecture Diagram (Conceptual)

```
Docker Client⬇Docker Daemon (inside Docker Host)⬇Images & Containers⬇Docker
Registry (if image not found locally)
```

Conclusion

Docker architecture is simple but powerful.

- Docker Client → Sends commands
- Docker Host → Executes everything
- Docker Registry → Stores and distributes images

Understanding this architecture helps you:

- Debug issues easily
- Design better CI/CD pipelines
- Optimize deployments
- Manage containers efficiently

That’s the complete overview of Docker architecture and how images and containers are created internally

Chapter 33: Docker Images & Containers — Complete Guide

Docker is an open-source containerization platform that allows us to package applications and their dependencies into lightweight, portable units called containers.

To understand Docker properly, we must clearly understand two important components:

- Docker Image
- Docker Container

Docker Image

What is a Docker Image?

A Docker Image is a blueprint or template used to create containers.

It contains:

- Application code
- Runtime
- Libraries
- Dependencies
- Configuration files
- Required services

Images are built using a Dockerfile.

Key Characteristics of Docker Images

- Images are immutable (cannot be modified once built).
- Images are read-only.
- Multiple containers can be created from a single image.
- The same image can be deployed in:
  - Development
  - QA
  - Staging
  - Pre-production
  - Production
- Lightweight and fast to deploy.

Because the image remains the same across environments, it eliminates version and configuration issues.

Docker Image Commands

1️⃣ Build a Docker Image

docker build -t username/image1:latest .

Explanation:

```
docker build → Builds the image
-t → Tags the image
username/image1 → Repository name (Docker Hub username + image name)
:latest → Tag/version
. → Current directory (where Dockerfile exists)
```

2️⃣ List Docker Images

docker images

or

docker image ls

Displays:

- Repository
- Tag
- Image ID
- Created time
- Size

3️⃣ Inspect an Image

docker inspect image_name_or_id

Provides detailed metadata about the image.

4️⃣ View Image History

docker history image_name_or_id

Shows layer-by-layer build history.

5️⃣ Remove an Image

docker rmi image_id

Force remove:

docker rmi -f image_id

Dangling Images

```
Dangling images are untagged images (shown as <none>). They usually appear when:
```

- A new image build replaces an old tag.
- A build fails midway.

To list dangling images:

docker images -f "dangling=true"

To remove dangling images:

docker image prune

To remove all unused images:

docker image prune -a

Dangling images consume disk space, so regular cleanup is recommended.

Docker Container

What is a Docker Container?

A Docker Container is a running instance of a Docker image.

Key points:

- Created from an image
- Lightweight and fast
- Isolated environment
- Mutable (can change during runtime)
- Multiple containers can run from the same image

In container orchestration systems like Kubernetes, containers run inside a Pod. A Pod can contain one or more containers.

Docker Container Commands

1️⃣ Create and Run a Container

docker run -d --name new_container --network bridge username/image1:latest

Explanation:

```
docker run → Creates and starts container
-d → Run in detached mode
--name → Assign container name
--network bridge → Use default bridge network
username/image1:latest → Image used
```

2️⃣ Create Container Without Starting (Optional)

docker create --name new_container username/image1:latest

Start it later:

docker start container_id_or_name

3 List Running Containers

docker ps

4 List All Containers (Running + Stopped)

docker ps -a

5 Stop a Container

docker stop container_id

6 Remove a Container

docker rm container_id

Force remove:

docker rm -f container_id

7 Remove All Stopped Containers

docker container prune

8 Inspect a Container

docker inspect container_id

9 View Container Logs

docker logs container_id

10 View Container History (Image Layers)

docker history image_name

```
(Note: docker history works on images, not containers directly.)
```

s Image vs Container

Docker Image	Docker Container
Blueprint	Running instance
Immutable	Mutable
Read-only	Writable layer added
Used to create containers	Created from image
Stored in registry	Runs on Docker Host

Best Practices

```
Always tag images properly (v1, v2, etc.)
```

- Remove unused images regularly.

```
Use docker image prune and docker container prune for cleanup.
```

- Keep images minimal for better performance.
- Use official base images when possible.

Conclusion

Docker Images and Containers are the foundation of containerization.

- container mutable
- Images → Blueprint (immutable)
- its all about the above 2

Chapter 34: Dockerfile — Complete Flow from Image to Container

Docker is an open-source containerization tool that allows us to package an application along with all its dependencies into a single unit called a container.

Docker ensures that applications run consistently across all environments — Development, QA, Staging, Pre-Production, and Production.

To understand Docker clearly, we must understand the Docker Flow:

- Dockerfile
- Docker Image
- Docker Container

1️⃣ Dockerfile

What is a Dockerfile?

A Dockerfile is a text file that contains a set of instructions used to build a Docker image.

It defines:

- Base image
- Application dependencies
- Required packages
- Environment variables
- Execution commands
- Ports to expose

Once the Dockerfile is written, we use it to build a Docker image.

Common Dockerfile Instructions

Below are the most important Dockerfile commands:

1️⃣ FROM

Sets the base image.

FROM maven:3.9.6-eclipse-temurin-17

```
Every Dockerfile must start with FROM.
```

It defines the base environment required to build or run the application.

2 ARG

Defines build-time variables.

ARG BASE_IMAGE=maven:3.9.6-eclipse-temurin-17

FROM ${BASE_IMAGE}

```
ARG helps reduce repetition and makes Dockerfiles reusable.
```

3 WORKDIR

Sets the working directory inside the container.

WORKDIR /app

All subsequent commands will run inside this directory.

4 COPY

Copies files from the local system into the container.

COPY pom.xml .

COPY src ./src

```
Best practice: Use COPY instead of ADD unless you specifically need extra features.
```

5 ADD

Similar to COPY but can:

- Extract tar files automatically
- Download files from URLs

Example:

ADD app.tar.gz /app

Use it only when needed.

6 RUN

Executes commands during image build.

RUN mvn clean package

```
Each RUN creates a new layer in the image.
```

7 ENV

Sets environment variables.

ENV APP_HOME=/app

Used to define runtime variables.

8️⃣ EXPOSE

Specifies which port the application uses.

EXPOSE 8080

This is documentation for the container — it does not automatically publish the port.

9️⃣ USER

Runs the container with a non-root user (recommended for security).

RUN useradd -m appuser

USER appuser

Running containers as root is not recommended.

🔟 ENTRYPOINT

Defines the main command that always runs.

ENTRYPOINT ["catalina.sh"]

It cannot be easily overridden.

1️⃣1️⃣ CMD

Provides default arguments for ENTRYPOINT or runs a default command.

CMD ["run"]

Unlike ENTRYPOINT, CMD can be overridden at runtime.

1️⃣2️⃣ LABEL / MAINTAINER

Adds metadata.

LABEL author="Sai"

```
(Note: MAINTAINER is deprecated. Use LABEL instead.)
```

Multi-Stage Docker Build (Best Practice)

Instead of using multiple Dockerfiles, we use multi-stage builds.

Example:

Build Stage

FROM maven:3.9.6-eclipse-temurin-17 AS builder

WORKDIR /app

COPY pom.xml .

COPY src ./src

RUN mvn clean package

Runtime Stage

FROM tomcat:10-jdk17

COPY --from=builder /app/target/app.war /usr/local/tomcat/webapps/

EXPOSE 8080

This reduces image size and makes it production-ready.

2 Docker Image

What is a Docker Image?

A Docker Image is a blueprint or template used to create containers.

It contains:

- Application code
- Runtime
- Libraries
- Dependencies
- Configuration files

Images are built using:

docker build -t username/app:1.0 .

Key Characteristics of Docker Images

- Immutable (cannot be modified once built)
- Read-only
- Lightweight
- Portable
- Multiple containers can be created from one image
- Same image can be deployed across all environments

Because the image remains the same everywhere, version conflicts are eliminated.

3 Docker Container

What is a Docker Container?

A Docker Container is a running instance of a Docker image.

It is:

- Lightweight

- Fast
- Isolated
- Mutable during runtime

Create and run a container:

docker run -d -p 8080:8080 --name myapp username/app:1.0

List running containers:

docker ps

Stop a container:

docker stop myapp

Remove a container:

docker rm myapp

Containers in Kubernetes

In orchestration platforms like Kubernetes, containers run inside a Pod.

A Pod can contain:

- One container
- Or multiple tightly coupled containers

Complete Docker Flow

1 Write Dockerfile2 Build Docker Image3 Push image to registry (optional)4 Run container from image

Flow:

Dockerfile → Docker Image → Docker Container

Why Docker Flow Is Powerful

- Faster development cycle
- Environment consistency
- Lightweight deployment
- Easy scaling
- Secure isolation
- CI/CD friendly

Conclusion

Docker simplifies modern application deployment.

- Dockerfile → Defines how to build
- Docker Image → Blueprint
- Docker Container → Running instance

Understanding this complete flow helps you build production-ready, scalable applications efficiently.

Chapter 35: Docker Networking & Docker Volumes

Docker is one of the most popular containerization platforms used to build, ship, and run applications. But running containers alone is not enough in real-world environments.

Two critical concepts make Docker powerful in production:

- Container-to-container communication (Networking)
- Persistent data storage (Volumes)

In this blog, we will deeply understand both concepts in a simple and practical way.

Part 1: Docker Networking (Container-to-Container Communication)

Why Do Containers Need Networking?

In real-world applications, we don't run just one container.

For example:

- One container for frontend
- One container for backend
- One container for database

All these containers must communicate with each other.

Docker provides networking features to enable this communication.

How Docker Networking Works

When a container is created:

- Docker automatically assigns it a unique IP address.
- That IP address is used for communication inside the Docker network.
- Each container gets its own network namespace.

Using Docker networks, containers can talk to each other securely and efficiently.

Types of Docker Networks

Docker provides four types of networks:

- Bridge
- Host
- None

- Custom (User-defined Bridge)

Let's understand each in detail.

1️⃣ Bridge Network (Default Network)

The bridge network is Docker's default network.

If you do not specify a network while creating a container, Docker automatically attaches it to the bridge network.

Features:

- Each container gets a private internal IP address.
- Containers communicate using IP addresses.
- IP address is dynamically assigned.
- If the container restarts, IP may change.
- DNS name-based communication is limited in default bridge.

Problem:

If a container restarts and its IP changes, other containers cannot communicate using the old IP. This makes it less suitable for production systems.

2️⃣ Custom Network (User-Defined Bridge) – Recommended

A custom network is created manually by the user.

Example:

docker network create mynetwork

When containers are attached to this network:

docker run --network mynetwork nginx

Advantages:

- Containers can communicate using container names (DNS resolution works).
- Better isolation.
- More stable communication.
- Suitable for production.

Example:

If you have:

```
Container name: backend
Container name: database
```

You can connect using:

database:3306

Instead of using IP address.

This is why custom networks are recommended in real-time applications.

3 Host Network

In host networking:

- Container uses the host machine's IP address.
- No separate internal IP is created.
- Container shares host's network stack.

Advantages:

- High performance
- No network translation

Disadvantages:

- No isolation
- Port conflicts possible
- Less secure

Used only in special performance-based scenarios.

4 None Network

In this network:

- No IP address
- No internet access
- No communication with other containers

Used for highly restricted or secure environments.

Part 2: Docker Volumes (Persistent Storage)

Why Do We Need Volumes?

Containers are temporary (ephemeral).

If you:

- Stop the container
- Delete the container
- Container crashes

All data inside the container is lost.

This is a major problem for:

- Databases

- Log files
- Uploaded files
- Application data

To solve this, Docker provides volumes.

What is a Docker Volume?

A Docker volume is a storage mechanism managed by Docker that allows data to persist outside the container lifecycle.

Even if the container is deleted, the data remains safe.

Types of Docker Storage

There are three main types:

- Bind Mount
- Named Volume (Docker Volume)
- Anonymous Volume

1️⃣ Bind Mount

A bind mount connects a host directory directly to a container directory.

Example:

docker run -v /home/user/data:/app/data nginx

How It Works:

```
/home/user/data → Host path
/app/data → Container path
```

Data is stored on the host.

Advantages:

- Simple
- Useful in development

Disadvantages:

- Security risks
- Direct host access
- Not portable
- Not ideal for production

2️⃣ Named Volume (Recommended)

Named volumes are created and managed by Docker.

Create volume:

docker volume create myvolume

Use in container:

docker run -v myvolume:/app/data nginx

Advantages:

- Managed by Docker
- More secure
- Easy backup
- Portable
- Production ready
- Can be shared between multiple containers

Docker stores volumes internally in:

/var/lib/docker/volumes/

We should not manually modify this location.

3 Anonymous Volume

Created automatically when no name is given.

Example:

docker run -v /app/data nginx

Docker generates a random name.

Disadvantage:

- Hard to manage
- Difficult to reuse
- Not recommended in production

Sharing Data Between Containers

Docker volumes allow multiple containers to use the same storage.

Example:

docker run -v myvolume:/data app1

docker run -v myvolume:/data app2

Both containers can read and write to the same volume.

Used in:

- Microservices
- Logging systems
- Backup services
- Data processing pipelines

Volume Drivers (Advanced Concept)

Docker supports different volume drivers.

Drivers allow storing data:

- Locally
- On NFS
- On cloud storage like AWS EBS
- On remote servers

In cloud environments, remote volumes help in:

- High availability
- Scalability
- Data safety

Important Docker Volume Commands

Create volume:

docker volume create myvolume

List volumes:

docker volume ls

Inspect volume:

docker volume inspect myvolume

Remove volume:

docker volume rm myvolume

Remove unused volumes:

docker volume prune

Docker Networking + Volumes (Real-World Example)

Imagine a 3-tier application:

- Frontend container
- Backend container

- MySQL database container

Best practice:

- Use custom network for communication.
- Use named volume for database storage.

If MySQL container crashes:

- Volume keeps data safe.
- Backend reconnects using container name.
- Application continues working.

This is how Docker is used in real production systems.

Best Practices

✔ Use custom networks in production

✔ Use named volumes instead of bind mounts

✔ Never store important data inside containers without volumes

✔ Use remote storage in cloud environments

✔ Regularly backup important volumes

Final Conclusion

Docker Networking ensures containers communicate properly.

Docker Volumes ensure data is persistent and secure.

Together, they form the foundation of reliable, scalable, and production-ready containerized applications.

If you understand these two concepts clearly, you are ready to build real-world Docker-based systems.

DOCKER BEST PRACTICEs:

1. Use Official Images

Always pull images from **Docker Hub Official Repositories** or trusted sources.
Example:

- nginx

- mysql
- redis

Official images are secure and maintained.

✔ 2. Use Lightweight Images

Prefer smaller base images like:

- alpine
- distroless
- slim versions

Example:

FROM node:18-alpine

Smaller image = faster pull + less attack surface.

✔ 3. Regularly Update Images

Keep base images updated to avoid vulnerabilities.

docker pull nginx:latest

Rebuild your images frequently.

✔ 4. Remove Unused Resources

Clean up regularly:

```
docker system prune -a
docker image prune
docker container prune
docker volume prune
```

Prevents disk from filling.

✔ 5. Free Up Space

Check disk usage:

```
docker system df
```

Remove dangling images:

```
docker images -f "dangling=true"
```

✔ 6. Use Specific Tags (Avoid latest)

✕ Bad:

```
FROM nginx:latest
```

✔ Good:

```
FROM nginx:1.25.3
```

Prevents unexpected breaking changes.

✔ 7. Use Multi-Stage Builds

Reduces image size drastically.

```
# Build stage
FROM node:18 AS builder
WORKDIR /app
```

```
COPY package*.json .
RUN npm install
COPY . .

# Production stage
FROM node:18-alpine
WORKDIR /app
COPY --from=builder /app .
CMD ["node", "app.js"]
```

✔ 8. Use .dockerignore

Prevent unnecessary files from being copied.

Example .dockerignore:

```
node_modules
.git
.env
*.log
```

Improves build speed.

✔ 9. Run as Non-Root User

Security best practice.

```
RUN adduser -D appuser
USER appuser
```

Avoid running container as root.

✔ 10. Use Environment Variables Properly

Do not hardcode secrets.

✕ Wrong:

```
ENV DB_PASSWORD=123456
```

✓ Use:

- .env file
- Docker secrets
- Kubernetes secrets

✓ 11. Use Health Checks

Add health checks to monitor container.

```
HEALTHCHECK CMD curl --fail http://localhost:3000 || exit 1
```

✓ 12. One Process Per Container

Follow single responsibility principle.

Good:

- One container for app
- One for DB
- One for cache

✓ 13. Use Volumes for Persistent Data

Never store database data inside container layer.

docker run -v mydata:/var/lib/mysql mysql

✔ 14. Limit Container Resources

Prevent one container from consuming everything.

docker run -m 512m --cpus="1.0" nginx

✔ 15. Scan Images for Vulnerabilities

Use:

- docker scan
- Trivy
- Clair

Security is very important in production.

sss