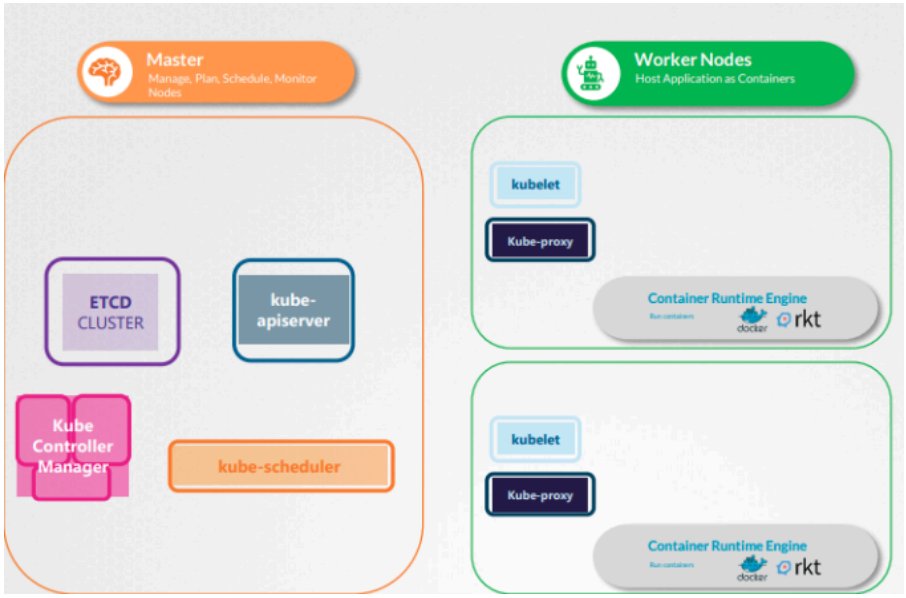
CKA NOTES

Overall Architecture

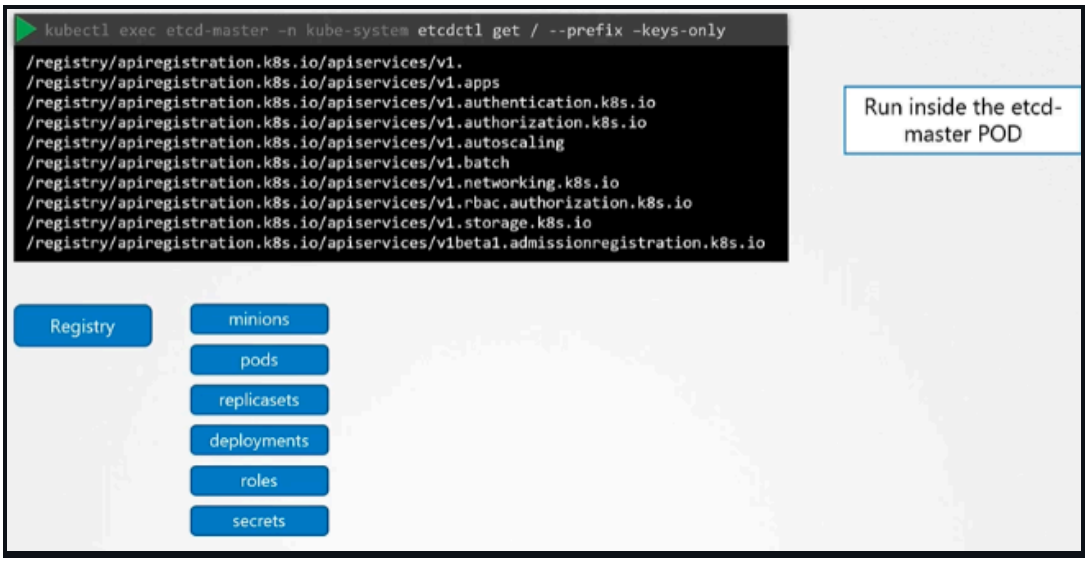


ETCD

- Distributed reliable key value store that is simple, secure fast
- The ETCD Datastore stores information regarding the cluster such as Nodes, PODS, Configs, Secrets, Accounts, Roles, Bindings and Others.
- Every information you see when you run the kubectl get command is from the ETCD Server.

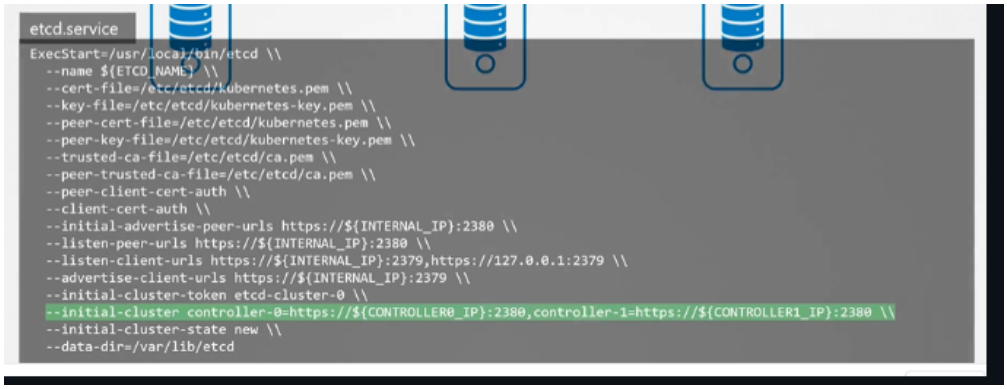
Setup

- Manually: deploy etcd by downloading binaries
    - $ wget -q --https-only "https://github.com/etcd-io/etcd/releases/download/v3.3.11/etcd-v3.3.11-linux-amd64.tar.gz"
- Kubeadm: then it will deploy the etcd server for you as a pod in kybe-system namespace
    - $ kubectl get pods -n kube-system
- To list all keys stored by kubernetes
    - $ kubectl exec etcd-master -n kube-system -- sh -c "ETCDCTL_API=3 etcdctl --cert=/etc/kubernetes/pki/etcd/server.crt --key=/etc/kubernetes/pki/etcd/server.key --cacert=/etc/kubernetes/pki/etcd/ca.crt get / --prefix --keys-only"
- Kubernetes Stores data in a specific directory structure, the root directory is the registry and under that you have various kubernetes constructs such as minions, nodes, pods



ETCD in a highly available environment (HA)

- You will have multiple masternodes in your cluster that will have multiple etcd instances spread across the master nodes
- Make sure instances of etcd know each other by setting the right parameters in etcd.service config.
    - The –inital-clustser option where you need to specify the different instances of the etcd service



Containerd vs Docker

- Originally docker was the dominant container tool, k8 built to orchestrate docker, being tightly coupled. K8 did not support any other container solutions initially.
- Kubernetes became popular as a **container orchestrator**, but needed to support **multiple container runtimes** (not just Docker).
- To enable this, Kubernetes introduced the **Container Runtime Interface (CRI)**, allowing any runtime to integrate with Kubernetes if it followed standards.
- **OCI (Open Container Initiative)** defines:
    - **Image spec** → how container images must be built

    - **Runtime spec** → how container runtimes must behave.

- **Docker was built before CRI existed** and **did not follow CRI**, so Kubernetes created **dockershim**, a temporary compatibility layer.
- Docker is **not just a runtime** — it includes:
    - Docker CLI
    - Docker API
    - Build tools
    - Volumes, security tools
    - **containerd** (daemon managing containers)
    - **runc** (low-level runtime)

- **Containerd**:
    - Is fully **CRI-compatible**.
    - Can run independently from Docker.
    - Became the preferred Kubernetes runtime.

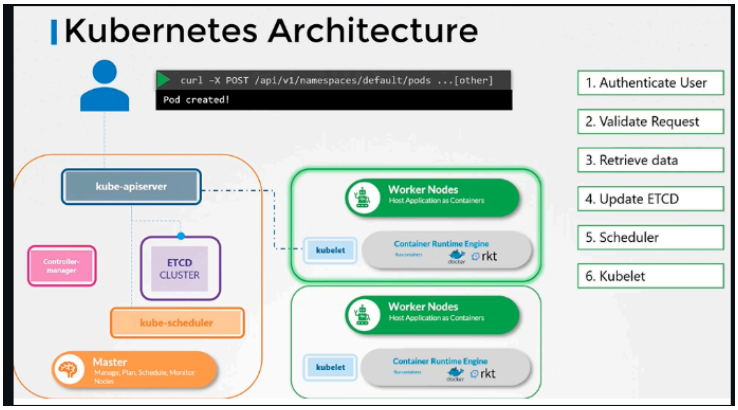
- Maintaining **dockershim** became unnecessary and complicated, so:Docker Engine is no longer a supported Kubernetes runtime.
- **Docker images still work** because:
    - Docker already followed **OCI image spec**.
    - Images remain compatible with containerd and other runtimes.
- ***Note: a container runtime is the low-level engine that actually launches the container process (is software that builds, starts, stops, and manages containers.)***

Crictl

- CLI tool for interacting with CRI-compatible runtimes (containerd, CRI-O, etc.).
- Developed by the Kubernetes community.
- Used mainly for debugging Kubernetes nodes and inspecting runtimes.
- Not intended for normal container management (kubelet would delete containers you create).
- Has Docker-like commands: crictl pS, crictl images, crictl exec, crictl log, critcl stats, critcl version, ceiricl inspect

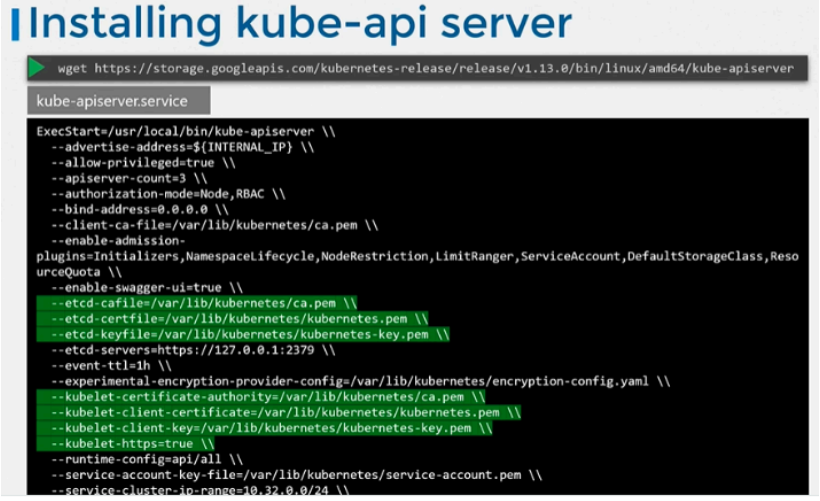
Kube-api server

- Is responsible for **authenticating**, **validating** requests, **retrieving and updating data** in etcd key-value store.
- kube-apiserver is the **only** component that interacts with etcd datastore
    - Other components like scheduler, controller manager and kubelet uses the api-server to update the cluster in their respective areas



1. **Pod Creation Flow**
    a. Client sends request: A user (or controller) sends a POST /api/v1/pods request to the API server.
    b. Authentication: API server verifies the identity of the caller (certs, tokens, etc.).
    c. Authorization & Admission Control:
        i. Checks if the user is allowed to create the resource.
        ii. Admission webhooks/validators mutate or validate the Pod spec.
    d. Update the etcd: API server writes the Pod object to etcd. After validation, the API server stores the Pod in the etcd datastore as “Pending”.
    e. The scheduler notices the unscheduled Pod. it watches the API server for Pods that: have no assigned node,are in Pending state.
    f. Scheduler selects a node Based on resource availability, constraints, affinity, etc and updates the Pod object with .spec.nodeName.
    g. Kubelet sees the Pod assigned to its node
        i. Each kubelet watches the API server for Pods assigned to its node.
        ii. The kubelet pulls required images, creates containers, sets up networking, volumes, etc.
    h. Kubelet reports Pod status back. The Pod transitions from Pending → ContainerCreating → Running
    i. Kubelet continuously updates status back to the API server, which writes it to etcd.

Kube api installation



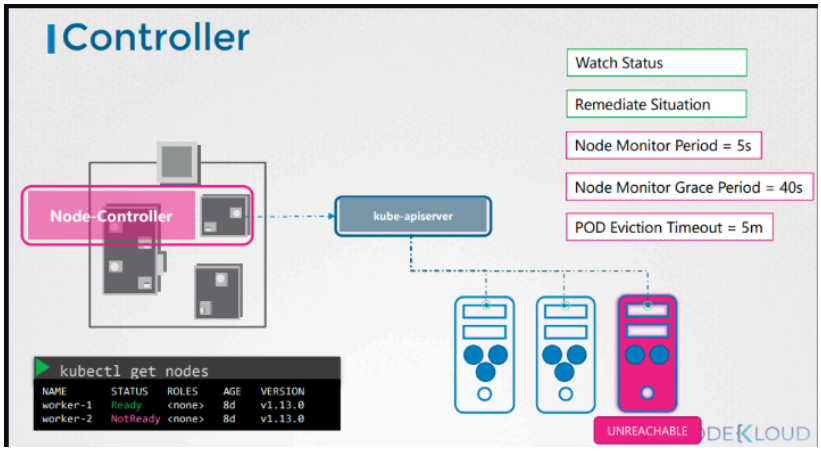
- To see the api-server options:
    - in the pod definition file located at /etc/kubernetes/manifests/kube-apiserver.yaml
    - In a Non-kubeadm setup, you can inspect the options by viewing the kube-apiserver.service (as shown above)
        - $ cat /etc/systemd/system/kube-apiserver.service
- $ ps -aux | grep kube-apiserver, to see the running processes

Kube control manager

- manages various controllers in Kubernetes
- A controller is a process that continuously monitors the state of components within the system and works towards bringing the whole system to the desired functioning state

Node Controller

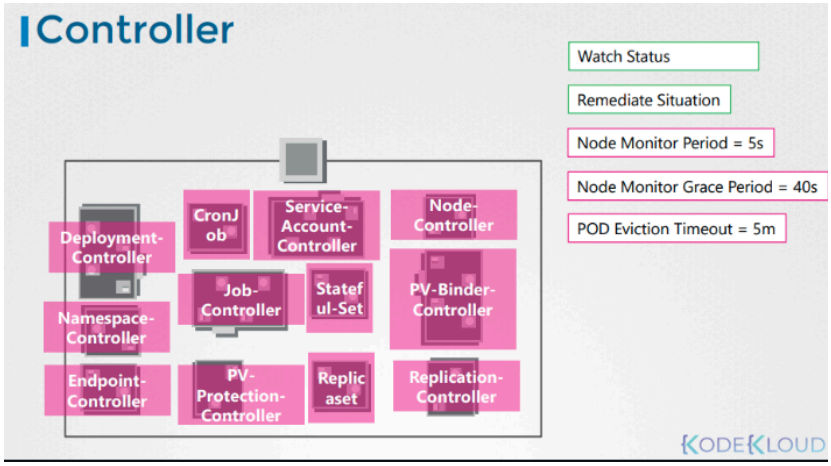
- Responsible for monitoring the state of the nodes and taking necessary actions to keep the application running



Replication Controller

- It is responsible for monitoring the status of replica sets and ensuring that the desired number of pods are available at all times within the set.

Other controllers:



Install

- When you install kube-controller-manager the different controllers will get installed as well.
- manual: $ wget https://storage.googleapis.com/kubernetes-release/release/v1.13.0/bin/linux/amd64/kube-controller-manager
- By default all controllers are enabled, but you can choose to enable specific ones from kube-controller-manager.service
    - $ cat /etc/systemd/system/kube-controller-manager.service
- Ikubeadm: You can see the options within the pod located at /etc/kubernetes/manifests/kube-controller-manager.yaml
-
- Can also use the ps -aux | grep kube-controller-manager to see the running process

Kube Scheduler

- Responsible for deciding which pod goes on which node. It doesn't actually place the pod on the nodes, that 's the job of the kubelet
- **Why do we need a kube-scheduler?**
    - Assigns Pods to nodes based on resource availability (CPU, RAM, etc.).
    - Optimises cluster utilisation by balancing workloads across nodes.
    - Respects constraints like affinity/anti-affinity, taints, tolerations, node selectors.
    - Ensures reliability by avoiding overloaded or unsuitable nodes.
    - Automates placement, so users don't manually choose nodes for every Pod.

Installation

- Manually using wget, Then extract it and run it as a service

- Using kubeadm: $ kubectl get pods -n kube-system, (options) $ cat /etc/kubernetes/manifests/kube-scheduler.yaml. Can also grep the process aswell

Kubelet

- Will create the pods on the nodes, the scheduler only decides which pods go where
- Kubelet workflow:
    - Watches the API server for Pods assigned to its node.
    - Downloads container images required by those Pods.
    - Starts containers using the node's container runtime (containerd/CRI-O).
    - Sets up networking and storage (CNI plugins, volumes, mounts).
    - Monitors container health and restarts them if needed.
    - Reports Pod and node status back to the API server.
    - Enforces desired state by ensuring the actual containers match the Pod spec.

Kube proxy

- In a Kubernetes cluster, every Pod can communicate with every other Pod because Kubernetes requires a flat, cluster-wide network. This is provided by installing a CNI Pod network plugin (like Calico, Flannel, Cilium).
- kube-proxy runs on every node and manages how network traffic is routed to Services. It sets up the necessary iptables, IPVS, or nftables rules so that Pods can reach Services reliably.

CNI Pod Network Plugin

- CNI = Container Network Interface, a standard for configuring container networking.
- A CNI plugin provides the actual network connectivity for Pods in a Kubernetes cluster.
- It ensures that:
    - Every Pod gets its own IP address.
    - Pods can communicate with each other across nodes.
    - Network policies (like restrictions or security rules) are enforced.

- Examples of CNI plugins: Calico, Flannel, Cilium, Weave Net.
- Without a CNI plugin, Pods cannot communicate outside their own node.

What is a Pod, Node, and Cluster in Kubernetes?

- Pod: The smallest deployable unit in Kubernetes. It wraps one or more containers and provides shared networking and storage.
- Node: A worker machine (VM or physical) where Pods run. Each node has the kubelet, kube-proxy, and container runtime.
- Cluster: A set of nodes managed by the Kubernetes control plane.

Deploying a Container as a Pod

- Kubernetes does not run containers directly on worker nodes; instead, it schedules Pods, which then run containers.

- To create an Nginx Pod: kubectl run nginx --image=nginx
- A ReplicaSet ensures a specified number of identical Pods are running at all times.
- If a Pod fails or is deleted, the ReplicaSet automatically creates a new one to maintain the desired count.
- Commonly managed indirectly through Deployments (which create and manage ReplicaSets for you).

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myapp-replicaset
  labels:
    app: myapp
    type: front-end
spec:
 template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
     containers:
     - name: nginx-container
       image: nginx
 replicas: 3
 selector:
   matchLabels:
    type: front-end
```

This means:

- The ReplicaSet will look for all Pods that have the label type: front-end.
- Any Pod that matches these labels is considered part of this ReplicaSet.
  The ReplicaSet will ensure that exactly replicas: 3 of those Pods exist at all times.
- Create: Kubectl create -f (file name flag) replicaset-definition.yaml
- Get all: Kubectl get replicaset -n <namespace>

Labels & Selectors — What's the Deal?

Labels

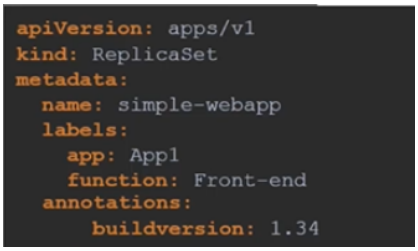
- Key–value pairs attached to Kubernetes objects (Pods, Services, Deployments, etc.).
- Used to organize, group, and identify objects, attach metadata
- Example:
    - Labels:
        - app: nginx
        - tier: frontend

Selectors

- Queries used by other resources to find and match objects based on labels.
- Example: A Service uses a selector to know which Pods to route traffic to.

    - Example query:
    - Kubeclt get pods - - selector app=App1

- While labels and selectors are used to group objects, annotations are used to record other details for informative purpose



Difference Between kubectl create, apply, replace, scale?

kubectl create

- Creates a new resource from a manifest or command.
- Fails if the resource already exists.
- Good for one-off object creation.

kubectl apply

- Declarative management.
- Creates the resource if missing, **or updates it to match your file.**
- Safest for day-to-day DevOps because it merges with existing configuration.

kubectl replace

- Imperative update.
- Replaces the existing resource entirely with what's in the file.
- Deletes fields not present in the new manifest.
- Fails if resource does not exist.

kubectl scale

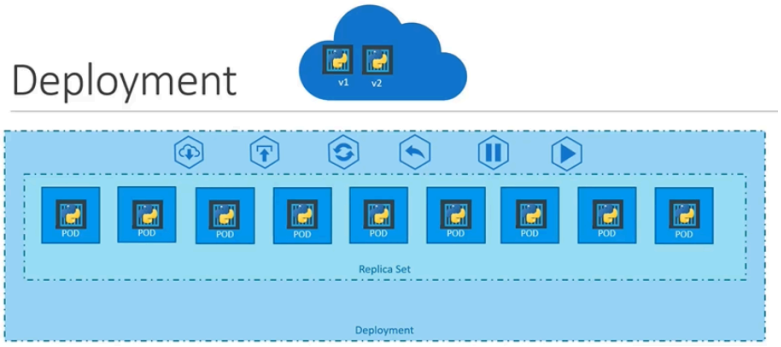
- Updates only the replica count of scalable controllers (ReplicaSet, Deployment).
- Does not modify other configuration

Ways to scale replicasets

1. update the number of replicas in the replicaset-definition.yaml definition file
2. use kubectl scale command.$ kubectl scale --replicas=6 -f replicaset-definition.yaml
3. third way is to use kubectl scale command with type and name
    a. kubectl scale --replicas=6 replicaset myapp-replicaset

Deployment

- A Deployment (a kubernetes object) is a higher-level controller (management layer) that manages ReplicaSets, which then manage your Pods.
- It provides:
    - Rolling updates (zero-downtime upgrades of Pods)
    - Rollbacks (undo to previous version)
    - Declarative spec management (you describe the desired state, K8s handles it)
- In short:Deployment = versioned, rolling-update manager for Pods.



Example of deployment definition:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
    type: front-end
spec:
 template:
    metadata:
      name: myapp-pod
      labels:
        app: myapp
        type: front-end
    spec:
     containers:
     - name: nginx-container
       image: nginx
 replicas: 3
 selector:
   matchLabels:
    type: front-end
```

- Note the outer spec is deployment level, Wherea the inside spec spec describes the Pod.
- Create kubeclt create -f deployment definition
- Kubectl get deployment -n namespace

Namespace

- A Namespace is a virtual cluster inside your Kubernetes cluster Used to isolate, organize, and separate resources.
- Useful for:
    - Multi-team environments (e.g., dev, test, prod)
    - Avoiding name conflicts (Pod named “api” can exist in multiple namespaces)
    - Applying resource quotas or RBAC rules per group.
- In short: Namespace = logical isolation for resources within a cluster.
- namespaces isolate Kubernetes objects, not nodes. They group and control resources logically but all Pods still run on the same cluster unless extra scheduling rules are applied.
- By default pods are created in the **default namespace/**
- To switch to a particular namespace permanently run the below command.
    - **$ kubectl config set-context $(kubectl config current-context) --namespace=dev.** This command finds the current context by name and then sets the namespace field to dev
- To view pods in all namespaces
    - $ kubectl get pods --all-namespaces
- To create a pod in a different namespaces use
    - $ kubectl create -f pod-definition.yaml --namespace=dev
    - Alternatively can move the –namespace flag definition into the pod definition file

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  namespace: dev
  labels:
     app: myapp
     type: front-end
```

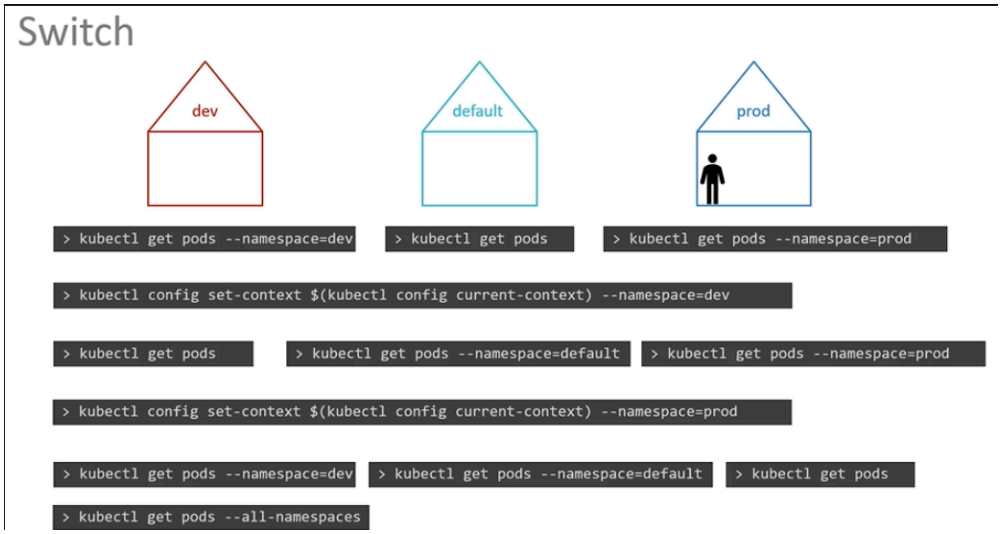
- To create a **new namespace**. Create a namespace definition as shown below and then run kubectl create

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

```
$ kubectl create -f namespace-dev.yaml
```

Another way to create namespace

- $ kubectl create namespace dev
- An illustration of switihcing between namespaces



- To limit resources on a namespace, create a resource quota. To create one start with ResourceQuota definition file:

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
  namespace: dev
spec:
  hard:
    pods: "10"
    requests.cpu: "4"
    requests.memory: 5Gi
    limits.cpu: "10"
    limits.memory: 10Gi
```

Service

- A Service provides a stable network endpoint to access one or more Pods. enables communication between various components within and outside of the application.
- Since Pods are ephemeral (IP changes when recreated), a Service gives:
    - A consistent cluster IP
    - Load-balancing across matching Pods (using label selectors)
    - Types include: ClusterIP (default): internal-only access
- Ty[pes of services include, clusterip, nodeport, loadbalancer LoadBalancer: cloud provider LB (AWS ELB, GCP LB, etc.)

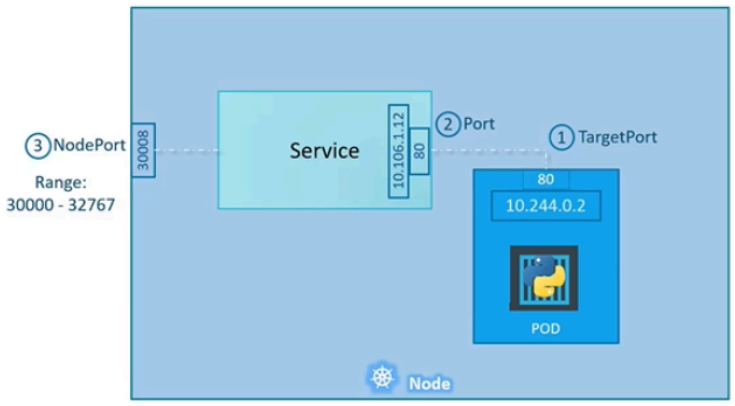
ClusterIP

- Purpose: Internal-only access within the cluster.
- creates a Virtual IP inside the cluster to enable communication between different services such as a set of frontend servers to a set of backend servers.

- How it works:
    - Assigns a virtual IP to the Service.
    - Any Pod in the cluster can use this IP to reach the Service.
    - Uses label selectors to forward traffic to matching Pods.
- Use case: Microservices communication inside the cluster.
- Example: **kubectl expose pod nginx --port=80 --target-port=80 --type=ClusterIP.** This creates an internal Service (`ClusterIP`) that load-balances traffic on port 80 to port 80 of the selected nginx Pods.
-
    - How traffic flows:

```
php-template

<External Client>
       |
       ▼
<Node IP>:<nodePort>   ← NodePort opens this port
       |
       ▼ kube-proxy
<Service Port>          ← ClusterIP of the service inside cluster
       |
       ▼
<Pod TargetPort>        ← Actual port on the Pod container
```

NodePort




```
service-definition.yml
apiVersion: v1
kind: Service
metadata:
    name: myapp-service

spec:
    type: NodePort
    ports:
     - targetPort: 80
      *port: 80
       nodePort: 30008
```

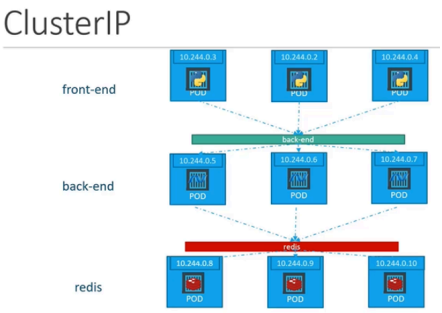
In Kubernetes, a Service acts as a stable access point for Pods, and its ports work together to route traffic: the targetPort is the port on the Pod container where the application actually listens,

the port is the internal Service port that other Pods in the cluster use to reach the Service, and the ClusterIP is the virtual IP that binds to this port to provide a stable internal address. The kube-proxy on each node handles routing, forwarding traffic from the ClusterIP or the nodePort—which exposes the Service on a specific port on every node for external access—through the Service port to the Pod's targetPort. Together, these components ensure that both internal and external clients can reliably reach the correct Pod regardless of Pod IPs or scaling changes.

What is a right way to establish connectivity between these services or tiers

- A kubernetes service can help us group the pods together and provide a single interface to access the pod in a group
- Key Concepts
    - Pods are ephemeral
    - Yes, Pods can be created, deleted, or rescheduled at any time. Their IP addresses are not guaranteed to stay the same.Because of this, directly communicating with Pod IPs is unreliable.

    - Services provide stability
    - A Service (like ClusterIP) provides a stable endpoint (virtual IP or DNS name) for a group of Pods.
    - The Service uses label selectors to know which Pods belong to it.
    - When traffic is sent to the Service, it automatically load-balances across all matching Pods



- so assume we have the definitions:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  selector:
    app: redis
  ports:
    - port: 6379
      targetPort: 6379
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    app: backend
  ports:
    - port: 8080
      targetPort: 8080
```

How communication works

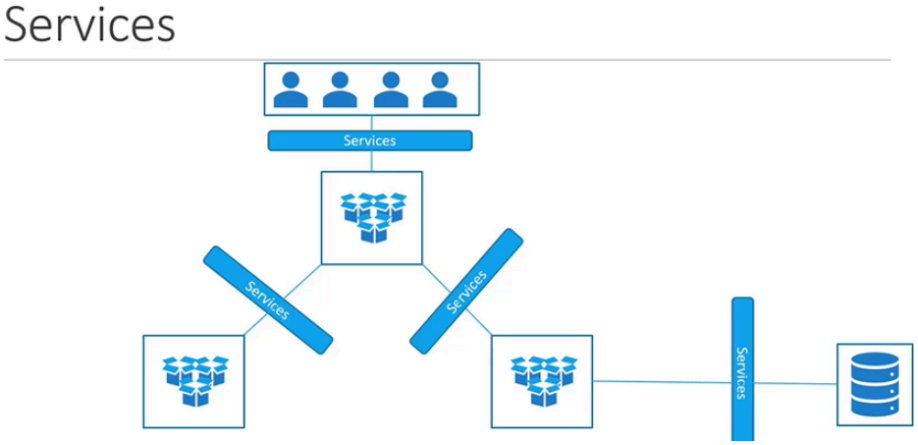
1. Backend wants to talk to Redis
    - Instead of looking up individual Pod IPs, the backend Pods connect to the Service name: redis:6379.
    - Kubernetes DNS resolves redis to the Service's ClusterIP.

2. Service load-balances
    - The Redis Service knows all Pods labeled app=redis.
    - kube-proxy forwards the traffic from the Service's ClusterIP to one of the Redis Pods' targetPort 6379.
    - If a Redis Pod dies, it's automatically removed from the pool, and traffic is sent to the remaining Pods.

3. Pod-to-Pod communication is stable
    - Backend Pods don't need to know Pod IPs — they just use the Service DNS.
    - Service ensures reliable communication despite Pod ephemeral nature.

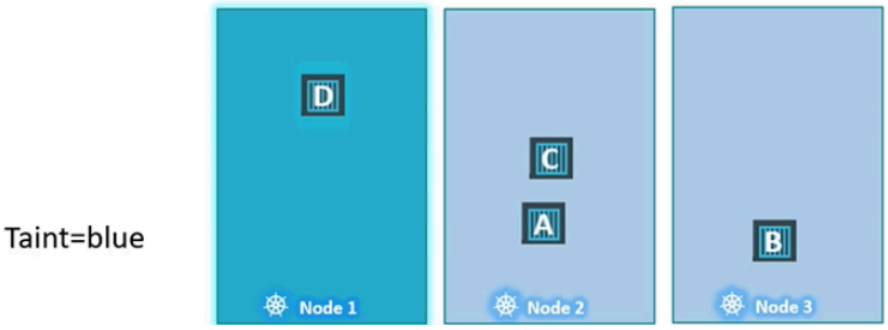
- Get all services: $ kubectl get services
- When you define a NodePort Service (or any Service) with a selector, you are telling Kubernetes: "Forward traffic arriving at this Service (via ClusterIP or NodePort) to all Pods that match these labels."
- The Service doesn't care if multiple Pods use the same port (like port 80) — in fact, that's expected.
- The Service will load-balance traffic across all matching Pods, forwarding to their targetPort.

Note: A NodePort is not attached to an existing ClusterIP Service; it is a Service type that automatically creates its own ClusterIP, and you must either change the Service type or create a new Service. A NodePort Service includes its own ClusterIP; to expose an existing ClusterIP Service, you change its type to NodePort or create a new NodePort Service pointing to the same Pods—NodePorts cannot be shared across Services.



Taints and tolerations

- focuses on pod to node relationships and how you can restrict what pods are places on what nodes
- Only pods which are tolerant to the particular taint on a node will get scheduled ono that node
- Taints and Tolerations do not tell the pod to go to a particular node. Instead, they tell the node to only accept pods with certain tolerations.



Taint command:

- Kubectl taint nodes <node-name> key=value:taint-effect
- l**e: kubectl taint nodes node1 app=blue:NoSchedule**
- The key-value:tain-effect is considered the tain part of the command
    - You choose what rule you're putting on that node → via key=value:effect.
    - Pods choose whether they can tolerate that rule → via tolerations
    - Taint-effect defines what would happen to the pods if they do not tolerate the taint.
    - 3 possible taint effects NoSchedule,, prefereNoScehdule, noExecute

1. NoSchedule
    - New pods that don't tolerate the taint will NOT be scheduled on the node.
    - Existing pods already on the node are not affected.

2. PreferNoSchedule
    - Kubernetes tries to avoid scheduling pods that don't tolerate the taint,
    - but it's only a preference — pods may still be scheduled if needed.

3. NoExecute
    - New pods without toleration: will NOT be scheduled on that node.
    - Existing pods without toleration: are evicted (removed) from the node.

Tolerations

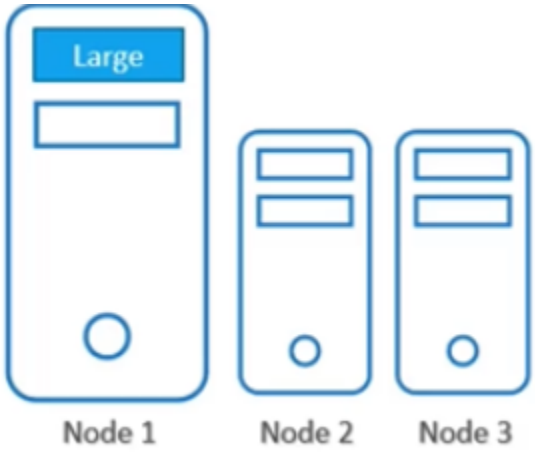
- are added to pods by adding a tolerations section in pod definition

```
pod-definition.yml
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: nginx-container
    image: nginx

  tolerations:
  - key:"app"
    operator:"Equal"
    value:" blue"
    effect:" NoSchedule"
```

- To see taint on node run the command kubectl describe node <node-name> | grep Taint

Node Selector

- simplest ways to control where your pods run in Kubernetes.
- Add property nodeselector to the spec and specify the label (corresponding to the label of the node you want to run it on)




```
pod-definition.yml
apiVersion:
kind: Pod
metadata:
 name: myapp-pod
spec:
  containers:
  - name: data-processor
    image: data-processor

  nodeSelector:
    size: Large
```

- Kubectl label node <node-name> label_key=label_value (to label the node)
- This approach is limited assuming needs are more complex, therefore need to use node affinity and antiaffinity instead

Node affinity

- Node affinity is a scheduling rule that tells Kubernetes which nodes a pod prefers or requires, based on node labels.

Node Affinity Types

1. requiredDuringSchedulingIgnoredDuringExecution (MOST COMMON)
   - Hard rule: Pod must be scheduled on nodes matching the affinity.
   - IgnoredDuringExecution: If the node later loses the label, the pod stays running.

2. preferredDuringSchedulingIgnoredDuringExecution
   - Soft rule: Scheduler will prefer matching nodes but can pick others.
   - IgnoredDuringExecution: Pod won't be moved if node labels change.

🔮 Planned but NOT implemented types
(Kubernetes has placeholders but they are not active types today.)
3. requiredDuringSchedulingRequiredDuringExecution
   - Hard requirement for both scheduling and runtime.
   - If node loses the label → pod must be evicted.
   - Not implemented as of today.

4. preferredDuringSchedulingRequiredDuringExecution
   - Soft during scheduling, but once running the pod must remain on a node with the label. If node loses the la
   - Not implemented yet.

Template of defining this within the pods:

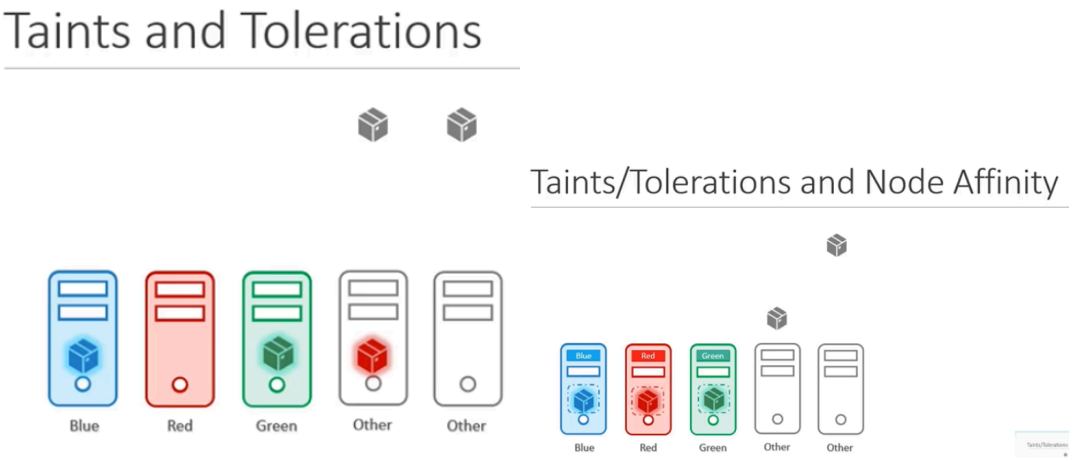
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example-pod
spec:
  affinity:
    nodeAffinity:

      # Hard requirement: must match
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: <label-key>
            operator: In
            values:
            - <label-value>

      # Soft preference: scheduler prefers this but can choose others
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: <label-key>
            operator: In
            values:
            - <label-value>
```

Taints and tolerations vs node affinity

- Taints and Tolerations **do not guarantee** that the pods will only prefer these nodes; in this case, the red pods may end up on one of the other nodes that do not have a taint or toleration set. (*Note it only says what pods to accept not weather to prefer one node over the other)*

- This means Taints and tolerations do not make pods prefer a node — they only restrict nodes.

In summary: Taints and tolerations repel pods from nodes: a taint marks a node as restricted, and only pods with matching tolerations are allowed to run there. Node affinity does the opposite—it attracts pods to nodes based on labels. Affinity can be a hard requirement or a soft preference, guiding the scheduler toward suitable nodes, while taints enforce that certain pods must not schedule on a node unless they explicitly opt in. In short, taints/tolerations push pods away; node affinity pulls pods toward specific nodes.



- Anti-affinity is the opposite of affinity: It tells Kubernetes to avoid placing certain pods together, usually to increase reliability or spread workloads.
- There are two kinds:
    - 1. Pod Anti-Affinity: Controls which other pods this pod should avoid being co-located with. Example use case: “Do not place two replicas of the same app on the same node.”

Resource Limits

- let us take a look at 3 node kubernetes cluster, Each node has a set of CPU, Memory and Disk resources available.
- If there is no sufficient resources available on any of the nodes, kubernetes holds scheduling the pod. You will see the pod in pending state. If you look at the events, you will see the reason is insufficient CPU.

```
NAME        READY   STATUS    RESTARTS   AGE
Nginx       0/1     Pending   0          7m

Events:
  Reason                  Message
  ------                  -------
  FailedScheduling  No nodes are available that match all of the following predicates:: Insufficient cpu (3).
```

Resource Requirements

- By default, K8s assume that a pod or container within a pod requires 0.5 CPU and 256Mi of memory. This is known as the Resource Request for a container.

- If application within the pod requires more than the default resources, you need to set them in the pod definition file
- By default, k8s sets resource limits to 1 CPU and 512Mi of memory

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
  labels:
    name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080
    resources:
      requests:
        memory: "1Gi"
        cpu: 1
      limits:
        memory: "2Gi"
        cpu: 2
```

- requests: The amount of resources the pod gua

yaml Copy code

```yaml
podAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchLabels:
          app: myapp
      topologyKey: "kubernetes.io/hostname"
```

grantees to get. Kubernetes uses this for scheduling—it ensures the node has at least this much available.

- limits: The maximum resources the pod is allowed to use.
    - The pod is scheduled on a node with ≥1Gi free memory.
    - At runtime, the pod can use more than 1Gi if available, up to 2Gi.
    - If the pod exceeds 2Gi memory limit, the kernel (via cgroups) will terminate (OOMKill) the pod.

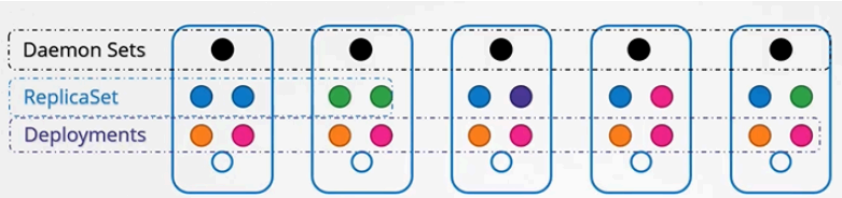
Deamonsets

- Ensures that exactly one copy of a pod runs on each node (or a subset of nodes based on labels).
- Unlike a ReplicaSet, which maintains a fixed number of replicas anywhere in the cluster, a DaemonSet ties pods to nodes.
- Common use cases:
    - Running log collectors on every node.
    - Running monitoring agents or network proxies per node.

- Ensuring node-level services are always present.

Behavior:

- New nodes joining the cluster automatically get the DaemonSet pod.
- Deleting a node removes its pod.
- You can restrict it to certain nodes using node selectors or affinities.



DaemonSet use cases:

- kube-proxy: Runs on every node to handle network routing and service proxying.
- Logging agents (e.g., Fluentd, Logstash): Collect logs from all nodes for central storage or analysis.
- Monitoring agents (e.g., Prometheus Node Exporter): Gather node-level metrics from every node.
- Networking plugins / CNI agents: Ensure network setup and connectivity on each node.

Key idea: Any service that must run on every node (or a subset) uses a DaemonSet

daemon-set-definition.yaml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: monitoring-daemon
spec:
  selector:
    matchLabels:
      app: monitoring-agent
  template:
    metadata:
      labels:
        app: monitoring-agent
    spec:
      containers:
      - name: monitoring-agent
        image: monitoring-agent
```

replicaset-definition.yaml

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: monitoring-daemon
spec:
  selector:
    matchLabels:
      app: monitoring-agent
  template:
    metadata:
      labels:
        app: monitoring-agent
    spec:
      containers:
      - name: monitoring-agent
        image: monitoring-agent
```

- Kubectl get daemonsets
- Kubectl describe daemonsets monitortong-daemon

Static Pods

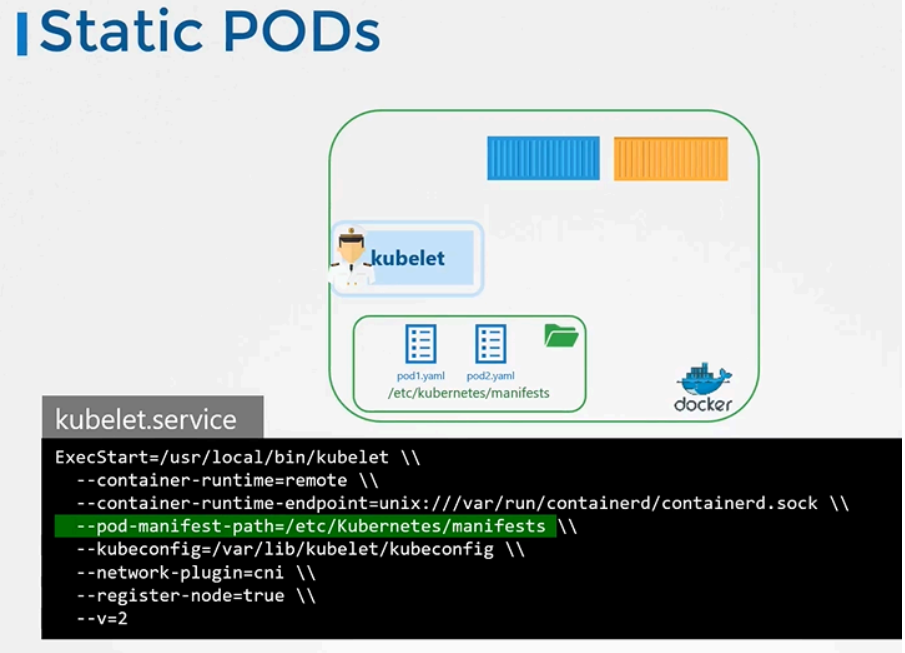
- are pods managed directly by the kubelet on a node, not by the API server or controllers. They run independently of the Kubernetes control plane.

Key Points

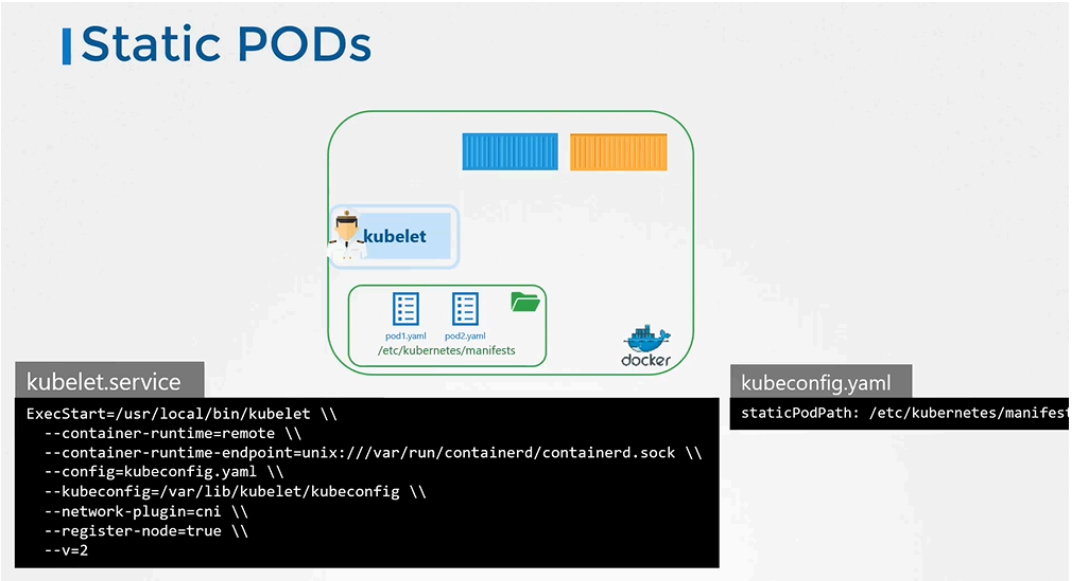
- Created from local files:
  You place a Pod manifest YAML file in the kubelet's staticPodPath directory (e.g., /etc/kubernetes/manifests).
- Kubelet manages them, not the scheduler:
  - Kubelet creates, restarts, and ensures the static pod is always running.
  - Scheduler never chooses a node because the pod must run on that node.

- Mirror Pods appear in the API:
  - The kubelet creates a mirror pod in the API server so you can "see" the pod via kubectl, but you cannot edit/delete it from the API.

You are "mirroring" the pod's object metadata and spec into the API so it can be observed with kubectl. You are NOT mirroring: runtime state, containers, logs, or lifecycle decisions.
  - Only editing/removing the file on disk changes the pod.
- Use cases
  - Bootstrapping the control plane (e.g., kube-apiserver, etcd, controller-manager).
  - Critical node-level services.



- Another way to configure the static pod, instead of specifying the option in the kublet.service, you could provide a path to another config file using the config option, and then defined the directory path as staticPodPath in the file

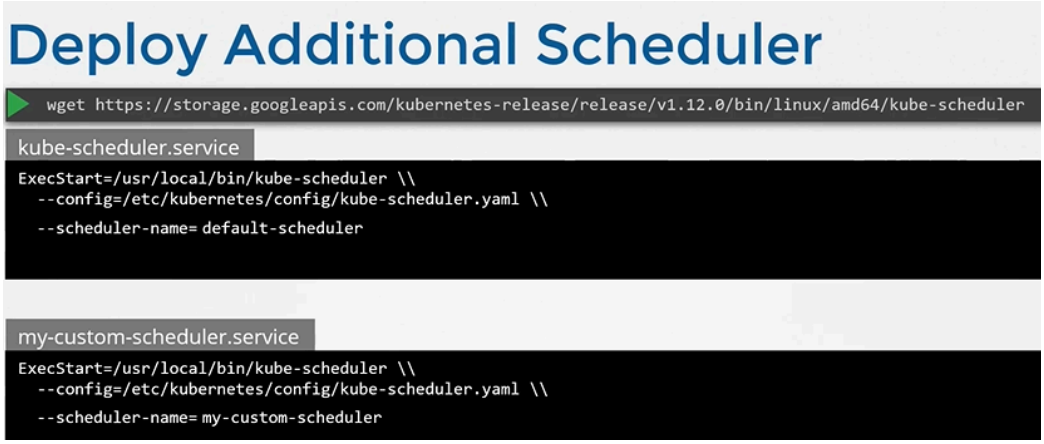


- To view static pods use command docker ps, because static pods are managed at pod level
- Kubelet can create both kinds of pods- static and the ones from the api server at the same time

Static PODs	DaemonSets
Created by the Kubelet	Created by Kube-API server (DaemonSet Controller)
Deploy Control Plane components as Static Pods	Deploy Monitoring Agents, Logging Agents on nodes
Ignored by the Kube-Scheduler	

Multiple schedulers

- Your kubernetes cluster can schedule multiple schedulers at the same time.



Deploy Additional Scheduler - kubeadm

/etc/kubernetes/manifests/kube-scheduler.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kube-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    image: k8s.gcr.io/kube-scheduler-amd64:v1.11.3
    name: kube-scheduler
```

my-custom-scheduler.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-custom-scheduler
  namespace: kube-system
spec:
  containers:
  - command:
    - kube-scheduler
    - --address=127.0.0.1
    - --kubeconfig=/etc/kubernetes/scheduler.conf
    - --leader-elect=true
    - --scheduler-name=my-custom-scheduler
    - --lock-object-name=my-custom-scheduler


    image: k8s.gcr.io/kube-scheduler-amd64:v1.11.3
    name: kube-scheduler
```

- Create the scheduler: kubectl create -f …yaml
- Kubectl get pods -n kube-system

To use the custom scheduler, add a new section called schedulerName and specify the name of the new scheduler

Use Custom Scheduler

```
kubectl get pods --namespace=kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
coredns-78fcdf6894-bk4ml	1/1	Running	0	1h
coredns-78fcdf6894-ppr6m	1/1	Running	0	1h
etcd-master	1/1	Running	0	1h
kube-apiserver-master	1/1	Running	0	1h
kube-controller-manager-master	1/1	Running	0	1h
kube-proxy-dgbgv	1/1	Running	0	1h
kube-proxy-fptbr	1/1	Running	0	1h
kube-scheduler-master	1/1	Running	0	1h
my-custom-scheduler	1/1	Running	0	9s
weave-net-4tfpt	2/2	Running	1	1h
weave-net-6j6zs	2/2	Running	1	1h

pod-definition.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  -  image: nginx
     name: nginx

   schedulerName: my-custom-scheduler
```

```
kubectl create -f pod-definition.yaml
```

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	0/1	Pending	0	6s

```
kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	6s

- When you add a custom scheduler to Kubernetes, it does not “interact” with the default scheduler. Instead, they run independently and in parallel, each responsible only for pods that explicitly target them.

- Here’s the clean breakdown: How Multiple Schedulers Work in Kubernetes
    - 1. Each scheduler is just a process watching for unscheduled pods

    - Both the default scheduler and your custom scheduler watch the API server for pods that don't yet have spec.nodeName set.
    - But each scheduler only acts on pods whose schedulerName matches theirs.
    - 2. Pod decides which scheduler handles it. You specify a scheduler per pod:
    - Spec:
        - schedulerName: my-custom-scheduler
        - Pods with this name → scheduled by your custom scheduler.
        - Pods without it → handled by the default scheduler.
        - They don't fight over pods.

What prevents conflicts?
Atomic binding

- When a scheduler chooses a node, it calls: POST /bind
- This sets spec.nodeName.
- This operation is atomic:
    - Whichever scheduler binds the pod first wins.
    - Other schedulers immediately see the pod is already bound and stop trying.

Leader-election inside each scheduler

- Each scheduler (default or custom) can run multiple replicas but only one becomes active.

What do they NOT do?

- They do not coordinate decisions with each other.
- They do not share queues or load-balance scheduling.
- They do not hand pods off to each other.
- **They are completely decoupled.**

Commands:

- To view events: kubectl get events
- Scheduler logs: kubectl logs <scheduler_name> -n <namespace>

Configuring Kubernetes Schedulers
(need to watch the video on this not enough information on the slides)

Monitoring

Metrics Server
What it is:

- A lightweight cluster-wide aggregator of CPU and memory resource usage.
- Powers kubectl top, Horizontal Pod Autoscaler (HPA), and Vertical Pod Autoscaler (VPA).

What it is NOT:

- It is not a full monitoring solution (that's Prometheus).

- It does not store metrics long-term (only in-memory and short-lived).

How Metrics Server Gets Metrics (Flow)

1. cAdvisor (built into kubelet) collects raw metrics:
    - Per-pod CPU usage
    - Memory working set
    - Container-level stats
    - Kubelet exposes these metrics at:
        i. /metrics/resource
        ii. /stats/summary

2. Metrics Server scrapes the kubelets periodically (default ~15 seconds).
3. Metrics Server aggregates the data in-memory and exposes it via the Kubernetes API as metrics.k8s.io.

Where metrics are stored

In-memory only

- Metrics Server does not write to disk or long-term storage.
- It keeps just enough recent samples (seconds) to respond to HPA and kubectl top.This makes it lightweight but not historical.

cAdvisor (Container Advisor)

- Runs inside the kubelet process (not a separate container anymore).
- Collects container/resource stats using:
    - cgroupS
    - kernel interfaces
    - container runtime
- Produces CPU usage, memory working set, filesystem, and network stats.

This is the source of truth.

- $ git clone https://github.com/kubernetes-incubator/metrics-server.git
- $ kubectl create -f metric-server/deploy/1.8+/, $ kubectl top node, $ kubectl top pod

Managing Application Logs

- At the container level (using docker)
    - Docker run -d image
    - Docker logs -f <container name> (-f a.k.a –follow, tells docker to stream logs in real time
- In k8s,
    - $ kubectl logs -f event-simulator-pod
    - Or if there are multiple containers running in the pod and want to view logs for a specific container
        - $ kubectl logs -f <pod-name> <container-name>
        - $ kubectl logs -f even-simulator-pod event-simulator

Rolling Updates and Rollback

- To make an update to an resource, either edit the yaml, save it and run the apply command, or alternatively explicitly make references using set command
- $ kubectl set image deployment/myapp-deployment nginx=nginx:1.9.1
- kubectl set image <resource>/<name> <containerName>=<newImage>

$ kubectl **rollout status** deployment/myapp-deployment

- Shows the progress of the rolling update.
- Waits until new pods are available.
- Tells you if rollout is complete or stuck.

$ kubectl **rollout history** deployment/myapp-deployment

- Shows revision history of previous updates.
- Useful for debugging rollbacks.

$ kubectl **rollout undo** deployment/myapp-deployment

- rolls the Deployment back to the previous revision.
- Triggered if a rollout fails or you want to revert a change.

Deployment strategies

Recreate Strategy

- Delete all existing pods first, then create new pods.
- Causes downtime because no pods are serving traffic during the replacement.
- When used:
    - When the new version is not compatible with the old version.
    - When you must ensure only one version runs at a time.

2. RollingUpdate Strategy (Default)

- Gradually replace old pods with new pods.
- Ensures zero downtime.
- Controls rollout speed with:
    - maxUnavailable (how many old pods can be down)
    - maxSurge (how many extra new pods can be created)
    - Default rollout: Usually 1 pod down, 1 extra allowed.
- Note: Undo command also uses same deployment strategy specified in the spec. So if rolling, undo will do it by pod basis

```yaml
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxUnavailable: 1
    maxSurge: 1
```

Commands and Arguments in Docker

- Containers run **ONE main process**, not a full OS like a VM.
- Containers are designed to start one program (e.g., nginx, python app, database), when that program exits → container exits.

CMD and ENTRYPOINT define what process runs when the container starts.

- If you pass a command to docker run, **it overrides CMD (but not ENTRYPOINT).**
- Note: ENTRYPOINT = **the main program**
- CMD = the **default argument**s to that program. So for example sleep is the program being run., and CMD 5 is overridable parameter

NOTE: **Why containers "don't host an OS," but you still need a base image.**
The base image (Ubuntu/alpine) is not a full OS — it's just the filesystem + userland tools, no kernel, no init system, no boot. That is why we say: "A container does not host an OS."
But yes, you still need a base image (like Ubuntu) because your application needs its libraries and filesystem, not because it needs its own kernel.

Commands and Arguments in Kubernetes

- Anything that is appended to the docker run command will go into the args property of the pod definition file in the form of an array.
- The command field corresponds to the entrypoint instruction in the Dockerfile so to summarize there are 2 fields that correspond to 2 instructions in the Dockerfile.

Configure Environment Variables In Applications

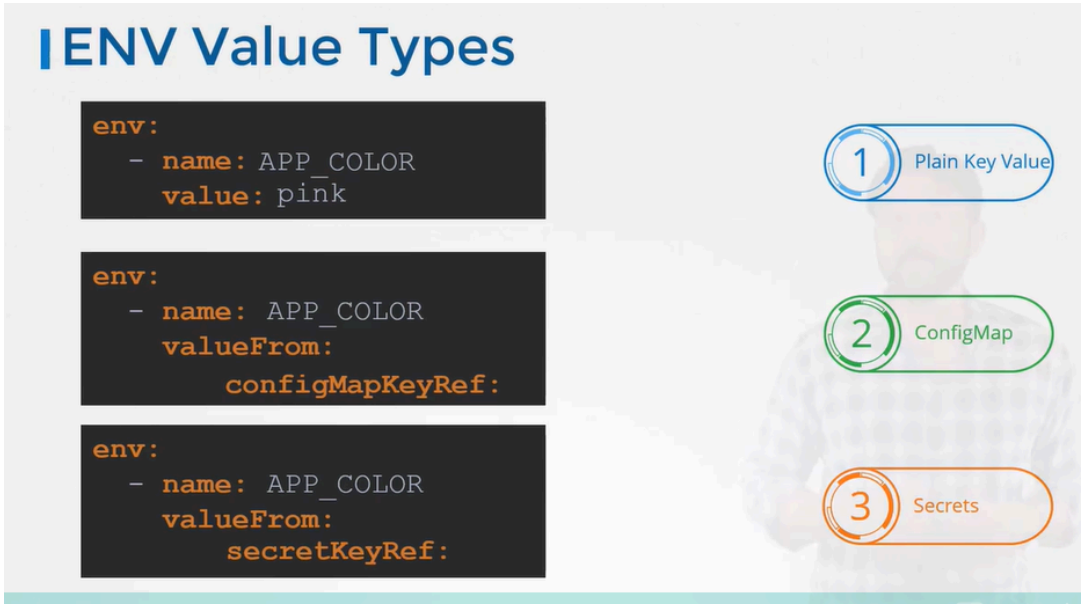
- In docker: use -e flag
    - $ docker run -e APP_COLOR=pink simple-webapp-color
- In k8, set an env property in pod definition file

```
docker run  -e APP_COLOR=pink simple-webapp-color
```

pod-definition.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: simple-webapp-color
spec:
  containers:
  - name: simple-webapp-color
    image: simple-webapp-color
    ports:
      - containerPort: 8080

    env:
      - name: APP_COLOR
        value: pink
```

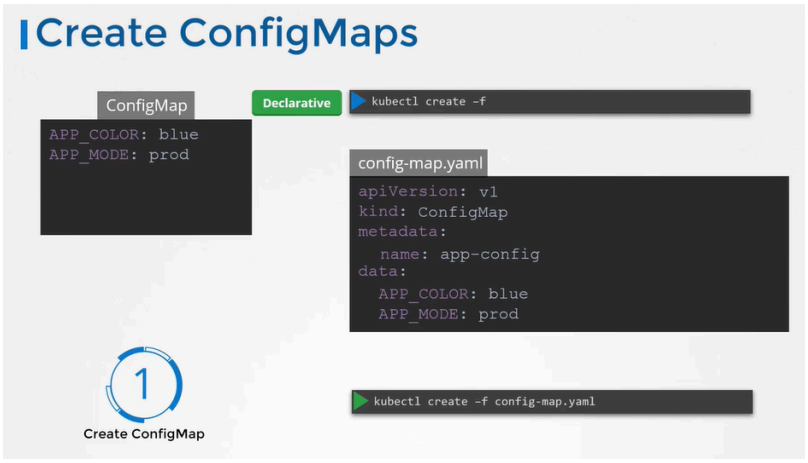
- Alternatively use configmaps/secrests



Configmaps

- There are 2 phases involved in configuring ConfigMaps.
    - First, create the configMaps
    - Second, inject them into the pod.
- 2 ways of creating a configmap.
    - The Imperative way
        - $ kubectl create configmap app-config --from-literal=APP_COLOR=blue --from-literal=APP_MODE=prod

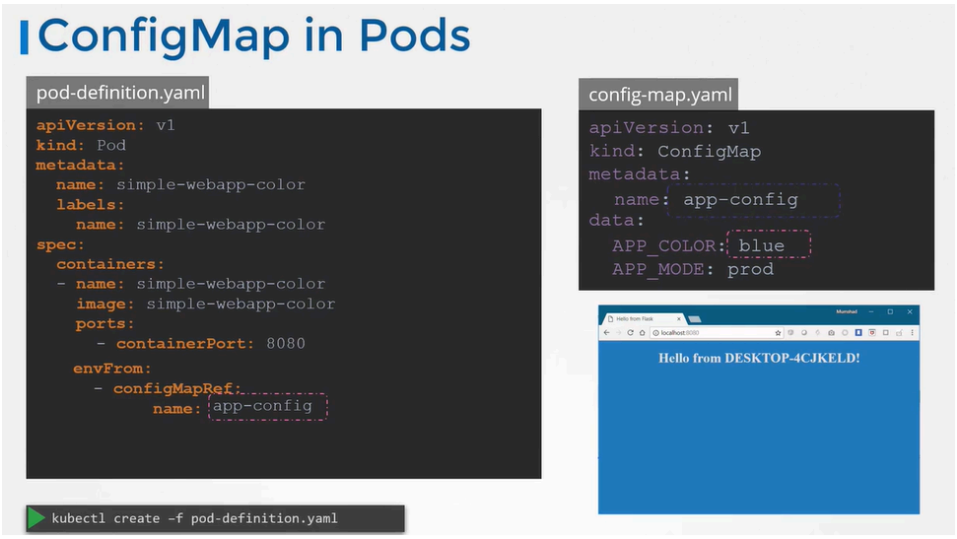
        - $ kubectl create configmap app-config --from-file=app_config.properties (Another way)

- declarative way
    - Create a config map definition file and run the 'kubectl create` command to deploy it. => kubectl create -f config-map.yaml
    - 

To view config Maps

- $ kubectl get configmaps (or) $ kubectl get cm
- $ kubectl describe configmaps

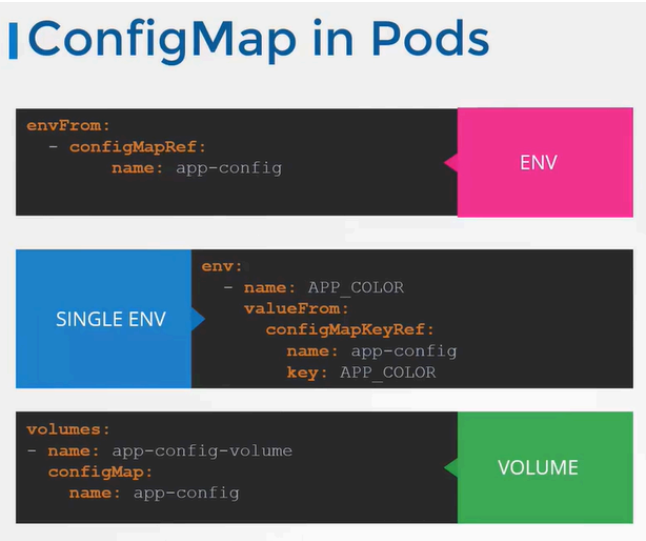
To inject:



- envFrom when loading all envars from configmap not env

there are other ways to inject configuration variables into pod

- You can inject it as a Single Environment Variable
- You can inject it as a file in a Volume



Why inject a ConfigMap via a volume

1. Decouple configuration from container images
    - Containers can remain immutable.
    - Config can change without rebuilding the image.

2. Dynamic updates
    - When you mount a ConfigMap as a volume, updates to the ConfigMap can propagate to the pod (depending on kubelet sync).
    - Allows apps to pick up new config without redeploying the container.

3. File-based applications
    - Some applications expect config as files, not environment variables.
    - Mounting a ConfigMap as a volume creates files inside the container corresponding to each key.

4. Large or structured config
    - Environment variables are limited in size and flat.
    - Volumes can hold structured config (e.g., multiple files, JSON, YAML).

Example

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-config
data:
  app.conf: |
    key1=value1
    key2=value2
---
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: app
    image: myapp:latest
    volumeMounts:
    - name: config
      mountPath: /etc/myapp
  volumes:
  - name: config
    configMap:
      name: my-config
```

The `app.conf` file becomes available at `/etc/myapp/app.conf` inside the container.
The app can read it like a normal file.

Secrets

1. Why Secrets exist
   - ConfigMaps store data in plain text, so passwords or API keys should not go there.
   - Secrets store sensitive information in base64-encoded format (not encrypted by default, just safer than plain text)
   -
2. Creating Secrets
   - **Imperative (CLI)**
     kubectl create secret generic app-secret \
       --from-literal=DB_Host=mysql \
       --from-literal=DB_User=root \
       --from-literal=DB_Password=paswrd

   - kubectl create secret generic app-secret --from-file=app_secret.properties

Declarative (YAML)

1. Base64-encode values:
   a. echo -n "mysql" | base64 # → bX1zcWw=
   b. echo -n "root" | base64 # → cm9vdA==

2. **Create secret.yaml:**
   apiVersion: v1
   kind: Secret
   metadata:
     name: app-secret
   data:
     DB_Host: bX1zcWw=
     DB_User: cm9vdA==
     DB_Password: cGFzd3Jk

3. Apply:kubectl create -f secret.yaml

3. Viewing and decoding secrets
   kubectl get secret app-secret -o yaml
   echo "bX1zcWw=" | base64 --decode # → mysql

4. Using Secrets in Pods
- Inject as environment variables

  envFrom:
  - secretRef:
      name: app-secret

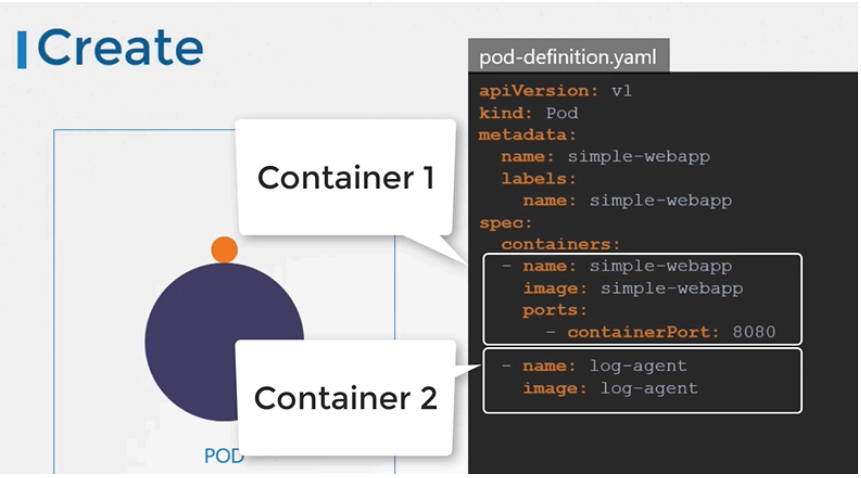
Inject as volume (files)
- Each key becomes a file with the value inside.
- Mounted in volumeMounts.

5. Security considerations
- Base64 is not encryption; anyone with access can decode.
- Secrets are “safer” than plain text mainly because:
  - They reduce accidental exposure.
  - Kubelet stores them in tmpfs, not disk.
  - Only sent to nodes that need them.
  - Deleted automatically when pods are removed.

- Best practices:
  - Don’t check secrets into Git.
  - Enable encryption at rest in etcd.
  - Consider tools like Vault or Helm Secrets for stronger security.

Multi-Container Pods

- To create a new multi-container pod, add the new container information to the pod definition file.



1. Sidecar Pattern

- Purpose: Adds functionality to the main container without modifying it.
- Characteristics:
    - Runs in the same pod as the main container.
    - Share network namespace and volumes, so they can easily communicate.

- Examples:
    - Logging: a container tails logs and sends them to a central system.
    - Monitoring: collects metrics or health info from the main container.
    - Configuration updater: automatically pulls config changes for the main container.
- Key takeaway: Sidecar is supportive, it extends capabilities.

2. Adapter Pattern

- Purpose: Transforms or standardizes data so the main container can work with it.
- Characteristics:
    - Runs alongside the main container.
    - Converts formats, protocols, or APIs to what the main container expects.
- Examples:
    - Converts logs into a different format before sending them to a logging system.
    - Aggregates metrics or events into a single format.
- Key takeaway: Adapter is like a translator—main container doesn't change, adapter does the conversion.

3. Ambassador Pattern

- Purpose: Acts as a proxy between the pod and external systems.
- Characteristics:
    - The main container doesn't handle external communication directly.
    - Ambassador handles routing, retries, TLS, or caching.
- Examples:
    - A proxy to a database outside the cluster.
    - Part of a service mesh (Envoy) to handle requests in/out of the pod.

    - Key takeaway: Ambassador is a gateway—it handles all external interactions for the pod.

Visual analogy:

- Main container: your app (e.g., web server)
- Sidecar: your personal assistant inside the office helping you with tasks
- Adapter: a translator in the office converting foreign documents for you
- Ambassador: the receptionist handling all interactions with the outside world

In a multi-container pod, each container is expected to run a process that stays alive as long as the POD's lifecycle. For example in the multi-container pod that we talked about earlier that has a web application and logging agent, both the containers are expected to stay alive at all times. The process running in the log agent container is expected to stay alive as long as the web application is running. **If any of them fails, the POD restarts.**

But at times you may want to run a process that runs to completion in a container. For example a process that pulls a code or binary from a repository that will be used by the main web application. That is a task that will be run only one time when the pod is first created. Or a process that waits for an external service or database to be up before the actual application starts. **That's where initContainers comes in.**

An initContainer is configured in a pod like all other containers, except that it is specified inside a initContainers section, like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox
    command: ['sh', '-c', 'git clone <some-repository-that-will-be-used-by-application> ;']
```

When a POD is first created the initContainer is run, and the process in the initContainer must run to a completion before the real container hosting the application starts. You can configure multiple such initContainers as well, like how we did for multi-containers pod. In that case, each init container is run one at a time in sequential order.

If any of the initContainers fail to complete, Kubernetes restarts the Pod repeatedly until the Init Container succeeds.

```
initContainers:
- name: init-myservice
  image: busybox:1.28
  command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
- name: init-mydb
  image: busybox:1.28
  command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

- Kubernetes supports self-healing applications through **ReplicaSets** and Replication Controllers. The replication controller helps in ensuring that a POD is re-created automatically when the application within the POD crashes. It helps in ensuring enough replicas of the application are running at all times

Cluster maintenance

OS upgrades:

- If a node stays down for >5 minutes, Kubernetes terminates its pods and reschedules them elsewhere.
- Before maintenance, you can safely move workloads off a node using: **kubectl drain node-1**
    - This evicts pods and cordons the node (marks it unschedulable).
- After the node comes back, it remains unschedulable until you explicitly run: **kubectl uncordon node-1**
- You can also manually mark a node as unschedulable without evicting pods using
- kubectl cordon node-1 (This does not move or terminate existing pods.)

Software versions

Is it mandatory for all of the kubernetes components to have the same versions?

- No, The components can be at different release versions.
- At any time, kubernetes supports only up to the recent 3 minor versions
- The recommended approach is to upgrade one minor version at a time.

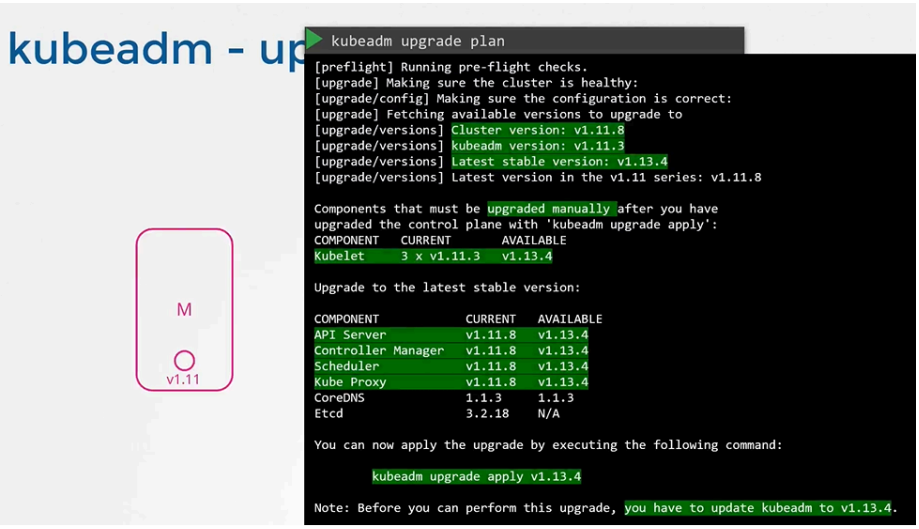
Upgrading a cluster

- different strategies that are available to upgrade the worker nodes

- One is to upgrade all at once. But then your pods will be down and users will not be able to access the applications.
- Second is to upgrade one node at a time
- Third would be to add new nodes to a cluster
- kubeadm has an upgrade command that helps in upgrading clusters.

$ kubeadm upgrade plan

-



- Upgrade kubeadm from v1.11 to v1.12
- $ apt-get upgrade -y kubeadm=1.12.0-00
- Upgrade the cluster
- $ kubeadm upgrade apply v1.12.0

Backup & Restore in Kubernetes

What to Backup

- Resource Configurations
    - Imperative commands exist, but declarative YAML files are preferred.
    - Keep YAML manifests in version control (GitHub).

You can export live configs:

- kubectl get all --all-namespaces -o yaml > all-resources.yaml

Note:

- -o yaml = format the output as YAML. it does NOT create a YAML file or directory. It just prints the resource in YAML to stdout.
- Common formats: -o yaml, json, wide name
- but this captures only some resource types. For full cluster backups, tools like Velero (formerly Ark) automate resource + volume backups.

- ETCD Database: ETCD stores all cluster state. I**nstead of backing up manifests, you can back up ETCD itself.**

ETCD Backup
Use etcdctl with proper certs:

ETCDCTL_API=3(this is the version number) etcdctl snapshot save snapshot.db \
--endpoints=https://127.0.0.1:2379
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
--cert=/etc/kubernetes/pki/etcd/etcd-server.crt \
--key=/etc/kubernetes/pki/etcd/etcd-server.key\
(*note `etcdctl` is just a client. To take a snapshot, it must connect to a running etcd server and ask it to create the snapshot.)*

- Check snapshot: etcdctl snapshot status snapshot.db

ETCD Restore

1. Stop the API server: service kube-apiserver stop
2. Restore the snapshot: etcdctl snapshot restore snapshot.db
3. Update etcd service config to point to restored data.

Reload systemd & restart etcd

4. systemctl daemon-reload, service etcd restart
5. Start the API server: service kube-apiserver start

Inspect cluster health:

6. etcdctl endpoint status,etcdctl endpoint health
7. To find the endpoint, as etcd is a static pod, check def under dir /etc/kubernetes/manifests/kube-apiserver.yaml amd look for the --etcd-servers flag

Security

Kubernetes security focuses on two key decisions:

- Who can access the cluster? → Authentication
- What are they allowed to do? → Authorization

Securing the Cluster

- **Secure Hosts:** Ensure the underlying nodes (VMs/servers) are hardened and secured before running Kubernetes.

- **TLS Encryption:** All communication between Kubernetes components (API server, etcd, scheduler, controller manager, kubelet, kube-proxy) is encrypted using TLS certificates.

- **Authentication**: All requests go through the kube-apiserver. Kubernetes does not manage end-user app authentication (that's handled by the apps themselves).

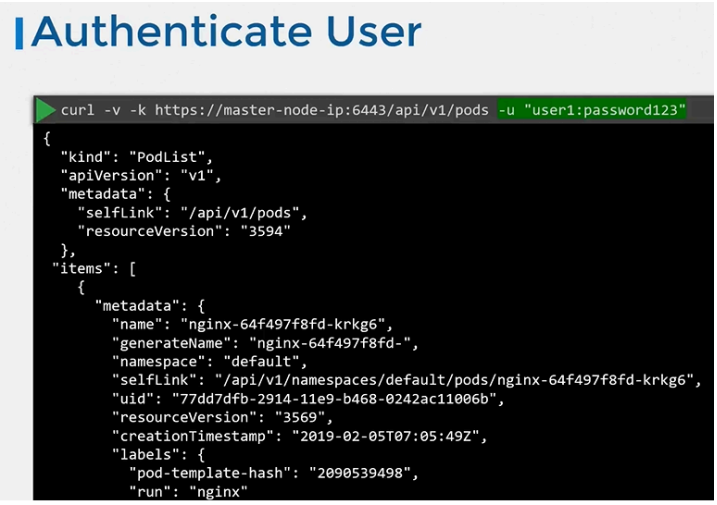
Types of Accounts
There are two types of users:

1. Humans – administrators and developers
2. Robots – services, applications, or processes needing cluster access

Authentication Mechanisms
Kubernetes supports multiple authentication methods, including:

- Basic authentication (username/password)
    - Configured via the kube-apiserver configuration (e.g., kube-apiserver.yaml when using kubeadm)
    - Credentials are stored in a CSV file
    - Users can be assigned to groups using extra CSV columns
- Certificates
- Identity services

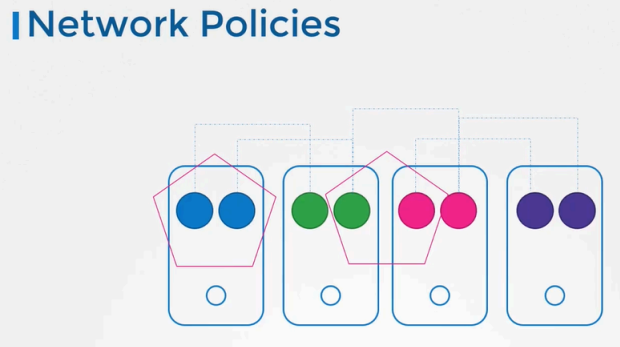
Example (Basic Auth API Call):
To authenticate using the basic credentials while accessing the API server specify the username and password in a curl command.
curl -k (or –insecure flag) https://<master-node-ip>:6443/api/v1/pods -u "user1:password123"



- v (verbose), tells curl to print detailed information about what its doing while handling the request
- K (insecure), tells curl to skip ssl/tls certification verification
- Basic auth is mainly for learning/testing and is not recommended for production.

Network Policies

- Control communication between pods/applications inside the cluster
- Define which workloads can talk to each other



TLS and Certificates

- TLS exists to ensure that **communication between two systems is encrypted**, authenticated, and protected from tampering.
- It achieves this by combining encryption keys, certificates, and trusted authorities into a structured process.
- **A certificate is not an encryption key**. A certificate is a **signed document** that proves identity. It contains a public key and identity information, and it is digitally signed by a Certificate Authority (CA). The certificate's purpose is to allow others to trust that a s**pecific public key truly belongs to a specific server or client**.

- Encryption itself is performed by keys, not certificates. TLS uses both asymmetric and symmetric encryption because each solves a different problem. Asymmetric encryption is used to **prove identity and securely exchange secrets**. Symmetric encryption is used for fast, ongoing data encryption **once trust is established.**

Workflow

- When a client connects to a server using TLS, the **server first sends its certificate** to the client. This certificate includes the server's public key and identity details and is signed by a CA.
- The client verifies the certificate by checking the CA's digital signature using the **CA's public key**, confirming the certificate has not expired, and ensuring the certificate identity matches the server it intended to contact. If any of these checks fail, the connection is rejected.
- Once the server's identity is trusted, the client generates a r**andom symmetric session key**. This session key will be used to encrypt all actual data. To send this session key securely, the client encrypts it using the **server's public key from the certificate.** Only the server can decrypt this encrypted session key because only the server has the matching private key. The server never sends its private key to anyone.

- After both sides have the same session key, all further communication is encrypted using symmetric encryption. Asymmetric encryption is no longer used after the handshake because it is slower and unnecessary once a shared secret exists.

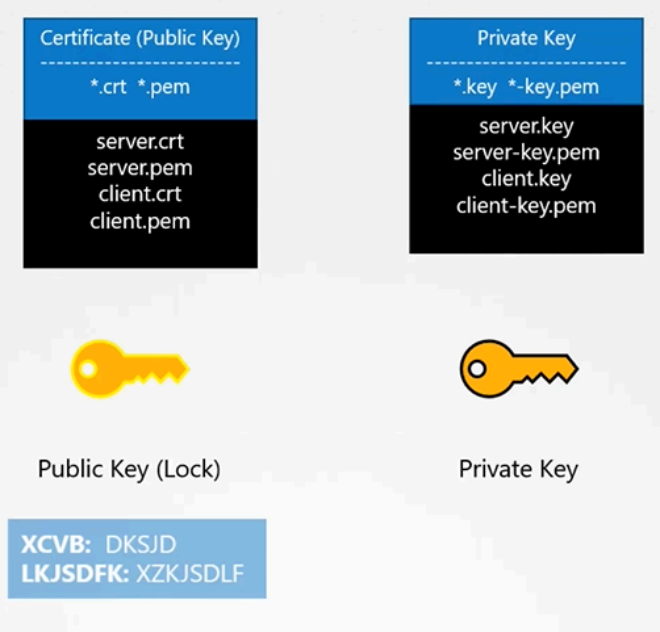
TLS in Kubernetes — How This Applies

- Kubernetes uses TLS everywhere because it is a distributed system with many **independent components communicating** over the network.
- Every internal connection must be encrypted and authenticated to prevent **impersonation** and **data leaks**.
- Kubernetes uses **mutual TLS (mTLS),** which means both the server and the client present certificates and verify each other's identities. This prevents unauthorized components from talking to the cluster.
- Each Kubernetes cluster has its own Certificate Authority. This CA signs all certificates used by Kubernetes components. Trust inside the cluster is established by trusting this CA.
- When a Kubernetes client such as kubectl or the controller manager connects to the kube-apiserver, it presents a client certificate. The API server verifies this certificate using the cluster CA, extracts the identity information from the certificate, and treats that identity as the authenticated user. Authorization mechanisms such as RBAC then decide what that user is allowed to do.
- Server certificates are used by components prove the identity of the service and enable encrypted communication. The certificates must include correct DNS names and IP addresses; otherwise, TLS verification fails.
- Certificates and keys are generated during cluster setup, usually stored under /etc/kubernetes/pki. If certificates expire, are misconfigured, or are signed by the wrong CA, Kubernetes components will fail to communicate, and the cluster may become partially or fully unusable.
-

Workflow

- In mutual TLS (mTLS), both the client and the server authenticate each other using certificates, but certificates are never used to carry encryption keys.
- When an mTLS connection starts, the server first sends its certificate to the client. This certificate contains the server's public key and identity information and is signed by a trusted Certificate Authority. The client verifies this certificate by checking the CA's signature, the certificate's validity period, and that the identity matches the server it intended to reach.
- After this, the server requests a client certificate. The client sends its certificate, which contains the client's public key and identity information and is also signed by the same CA. The server verifies the client certificate in the same way, using the CA to establish trust.
- Once both sides have verified each other's identity, the client generates a random symmetric session key. This session key is not placed inside a certificate. Instead, the client encrypts the session key using the server's public key from the server certificate and sends it to the server as part of the TLS handshake.

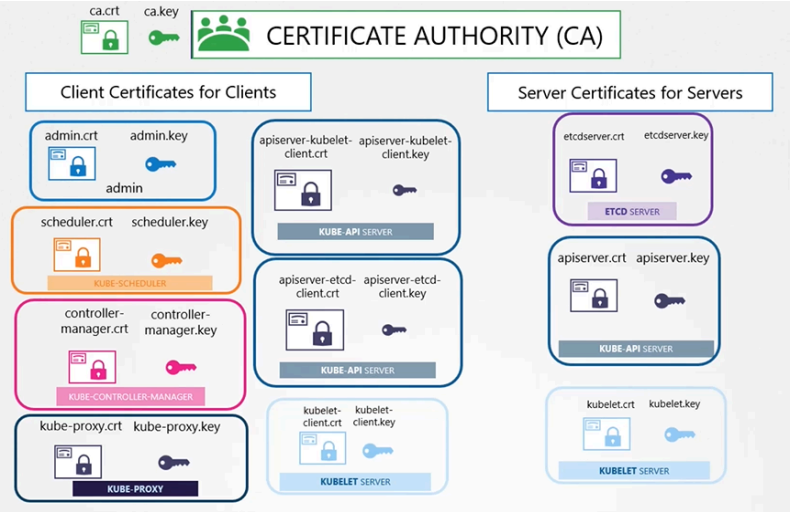
- The server then decrypts the session key using its private key, **which has never left the server.** At this point, both the client and the server possess the same symmetric session key.
- From this moment onward, all communication is encrypted using symmetric encryption with the shared session key. Certificates are no longer used for encryption; their role was only to establish identity and trust during the handshake.



Important Kubernetes-specific clarification

1. Each connection has its own session key.
   That means:
   - kubectl → API server → one session key
   - kubelet → API server → different session key
   - controller → API server → different session key

2. No session keys are shared across components or reused long-term.



Big picture first (so commands make sense)
Creating TLS certs always follows the same logic:

1. Generate a private key (secret, never shared)
2. Create a CSR (Certificate Signing Request = “please sign this public key for this identity”)
3. Sign the CSR with a CA → get a certificate

That’s it. Every command you see fits into one of those steps.

1. Generate CA private key

openssl genrsa -out ca.key 2048

What this does

- **genrsa** Generates an RSA private key
  - -out ca.key → save the private key to ca.key
- Key size is 2048 bits (standard secure size)

2. Generate CA CSR

openssl req -new -key ca.key -subj "/CN=KUBERNETES-CA" -out ca.csr

What this does

- Creates a CSR for the CA itself

- req → certificate request operations
- -new → create a new CSR
- -key ca.key → use this existing private key (ca.key) to derive public key. (priv key signs the csr)
- -subj "/CN=KUBERNETES-CA" → identity info
  - CN (Common Name) = name of this CA
- -out ca.csr → output CSR file
- **A CSR contains the public key + identity, not the private key.**

3. Self-sign the CA certificate

openssl x509 -req -in ca.csr -signkey ca.key -out ca.crt

What this does

- Turns the CSR into a certificate, Signs it with its own private key → self-signed CA
- x509 → X.509 certificate operations
- -req → input is a CSR
- -in ca.csr → CSR file
- -signkey ca.key → sign using this private key
- -out ca.crt → output certificate

4. Generate admin private key (same workflow for admin)

openssl genrsa -out admin.key 2048

- Creates admin's private key
- Never shared

5. Generate admin CSR (basic)

openssl req -new -key admin.key -subj "/CN=kube-admin" -out admin.csr

- CN=kube-admin → this becomes the username in Kubernetes

6. Sign admin CSR with CA

openssl x509 -req -in admin.csr -CA ca.crt -CAkey ca.key -out admin.crt

7. Admin certificate with privileges

openssl req -new -key admin.key -subj "/CN=kube-admin/O=system:masters" -out admin.csr

Critical Kubernetes detail

- CN → username
- O (Organization) → group

system:masters is a predefined Kubernetes admin group. This is how certificates map to RBAC identities.

- Controller manager, scheduler, kube-proxy, kubelet-as-client all follow the same exact process: Generate key, Generate CSR, Sign with CA. Only the CN and O fields change.

Server Certificates: servers must prove their identity to clients. ie

- ETCD server certificate. ETCD is a server, so:
    - Needs a server certificate
    - Must include correct SANs (DNS/IPs). SANs are mandatory; CN alone is not enough.

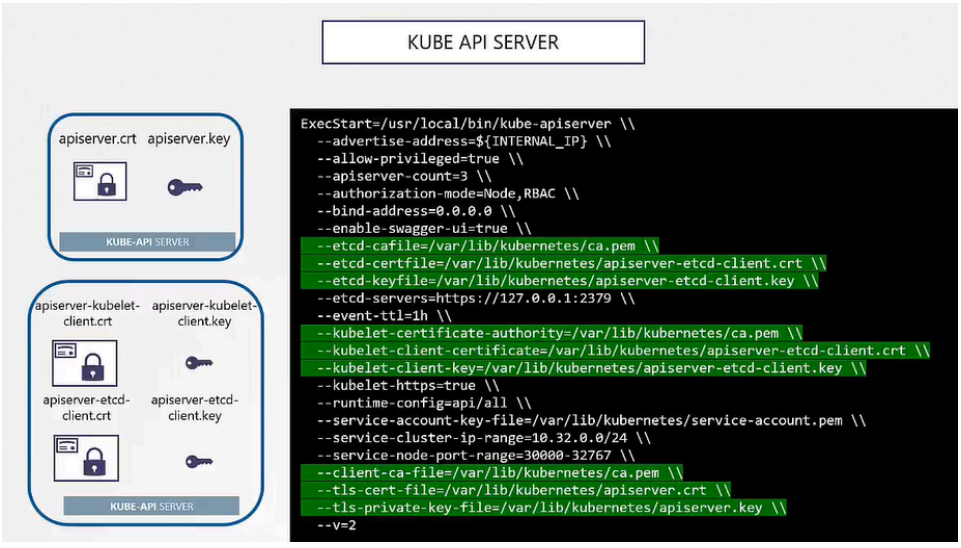
- kube-apiserver certificate This certificate is critical and must include:
    - Cluster IP, Service IP, DNS names, Load balancer IPs (if any)

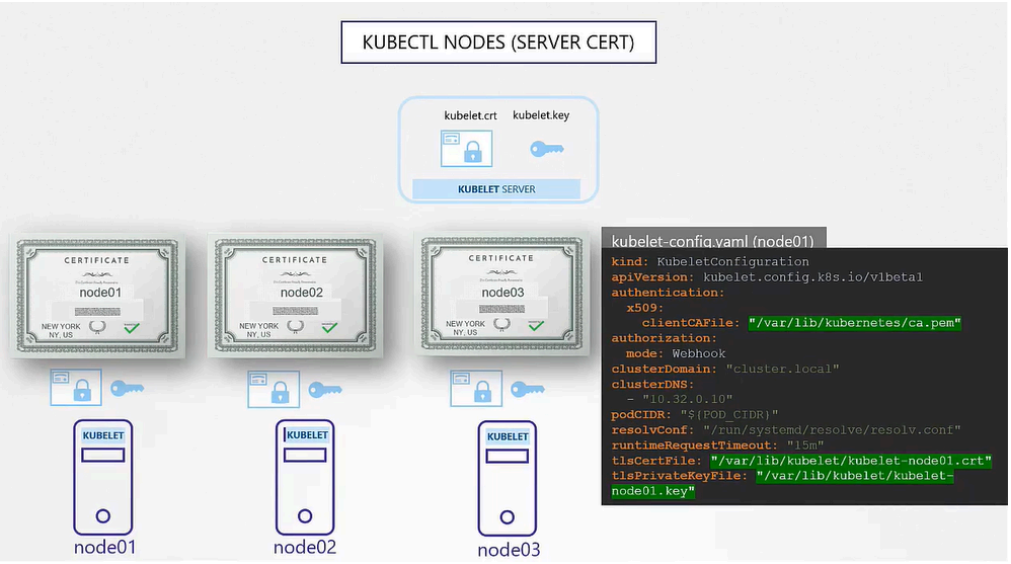
- kubelet certificates (this is where confusion usually is)
    - Kubelet has two identities: Kubelet as a server
        - API server connects to kubelet
        - Needs a server certificate. Used for: exec, logs, port-forward
    - Kubelet as a client
        - Kubelet connects to API server
        - Needs a client certificate

- Used to: register node, update node status, watch pods
- These are separate certificates.

x509 authentication in kube-apiserver config
When you see something like: --client-ca-file=/etc/kubernetes/pki/ca.crt

- This tells the API server: “Trust any client certificate that is signed by this CA.”
- The API server then Verifies the cert, Extracts CN and O, Uses them for authentication and RBAC
- No passwords, no tokens — pure x509-based auth.





Viewing Certificate Details in Kubernetes

- In a Kubernetes cluster, certificates are stored on control-plane nodes, usually under /etc/kubernetes/pki.
- When TLS issues occur, the first thing you want to check is what a certificate actually contains, rather than guessing.
- The following command is used to inspect a certificate: **openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout**
    - This command does not modify anything. It simply decodes and displays the certificate in a human-readable format.
        - -text expands all certificate fields so you can read them clearly.
        - -noout prevents OpenSSL from printing the raw encoded certificate at the end.
        -

When you run this command, you typically check: Issuer: which CA signed the certificate, Subject (CN and O fields), Validity: start and expiry dates (very common failure point), Subject Alternative Names (SANs): DNS names and IPs the certificate is valid for. Key usage / Extended key usage: whether the cert is allowed for client auth, server auth, or both

Why certificate inspection matters

Many Kubernetes TLS failures come from:

- Expired certificates
- Missing or incorrect SANs
- Certificates signed by the wrong CA
- Client certs missing correct CN or O fields
- **All of these problems are visible directly in the certificate output.**

Inspecting Server Logs (TLS Troubleshooting)

- Certificates alone don't always tell the full story. When TLS fails, components log exact reasons for the failure. Where you look depends on how Kubernetes was installed.

Hardware / systemd-based setup

- If a component like etcd runs as a system service, you inspect logs using journalctl: **journalctl -u etcd.service -l**
    - journalctl reads system logs managed by systemd.
        - -u etcd.service filters logs to the etcd service only.
        - -l prevents log lines from being truncated.

kubeadm-based setup (static pods)

- In kubeadm clusters, control-plane components run as pods, not system services. You inspect logs using kubectl: **kubectl logs etcd-master**
- This fetches logs directly from the etcd pod running on the control-plane node. The same applies to: kube-apiserver, kube-controller-manager, kube-scheduler

Docker-based setups

- If Kubernetes components are running as Docker containers (older or custom setups), you inspect them manually:

    docker ps -a
    docker logs <container-id>

How this ties back to TLS

When TLS breaks in Kubernetes:

1. Inspect the certificate → check identity, SANs, expiry, CA
2. Inspect logs → find the exact reason TLS negotiation failed
3. Match logs to cert details → fix the root cause

Most TLS problems are configuration or lifecycle issues, not "mystery bugs."

"The Hard Way"

```
cat /etc/systemd/system/kube-apiserver.service
[Service]
ExecStart=/usr/local/bin/kube-apiserver \\
  --advertise-address=172.17.0.32 \\
  --allow-privileged=true \\
  --apiserver-count=3 \\
  --authorization-mode=Node,RBAC \\
  --bind-address=0.0.0.0 \\
  --client-ca-file=/var/lib/kubernetes/ca.pem \\
  --enable-swagger-ui=true \\
  --etcd-cafile=/var/lib/kubernetes/ca.pem \\
  --etcd-certfile=/var/lib/kubernetes/kubernetes.pem \\
  --etcd-keyfile=/var/lib/kubernetes/kubernetes-key.pem \\
  --event-ttl=1h \\
  --kubelet-certificate-authority=/var/lib/kubernetes/ca.pem \\
  --kubelet-client-key=/var/lib/kubernetes/kubernetes-key.pem \\
  --kubelet-https=true \\
  --service-node-port-range=30000-32767 \\
  --tls-cert-file=/var/lib/kubernetes/kubernetes.pem \\
  --tls-private-key-file=/var/lib/kubernetes/kubernetes-key.pem
  --v=2
```

kubeadm

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.32
    - --allow-privileged=true
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --disable-admission-plugins=PersistentVolumeLabel
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.cr
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-k
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-c
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Host
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-cli
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-clie
    - --requestheader-allowed-names=front-proxy-client
```

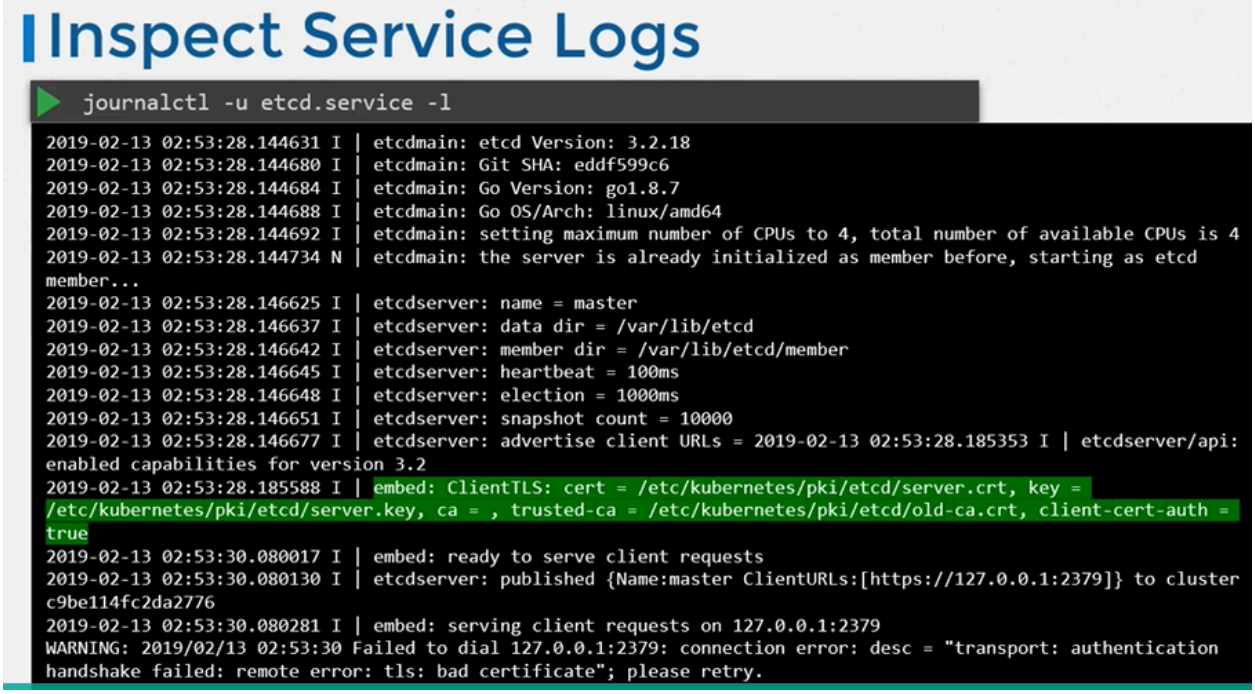

```
cat /etc/kubernetes/manifests/kube-apiserver.yaml
spec:
  containers:
  - command:
    - kube-apiserver
    - --authorization-mode=Node,RBAC
    - --advertise-address=172.17.0.32
    - --allow-privileged=true
    - --client-ca-file=/etc/kubernetes/pki/ca.crt
    - --disable-admission-plugins=PersistentVolumeLabel
    - --enable-admission-plugins=NodeRestriction
    - --enable-bootstrap-token-auth=true
    - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
    - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
    - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key
    - --etcd-servers=https://127.0.0.1:2379
    - --insecure-port=0
    - --kubelet-client-certificate=/etc/kubernetes/pki/apiserver-kubelet-client.crt
    - --kubelet-client-key=/etc/kubernetes/pki/apiserver-kubelet-client.key
    - --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
    - --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
    - --proxy-client-key-file=/etc/kubernetes/pki/front-proxy-client.key
    - --secure-port=6443
    - --service-account-key-file=/etc/kubernetes/pki/sa.pub
    - --service-cluster-ip-range=10.96.0.0/12
    - --tls-cert-file=/etc/kubernetes/pki/apiserver.crt
    - --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
```

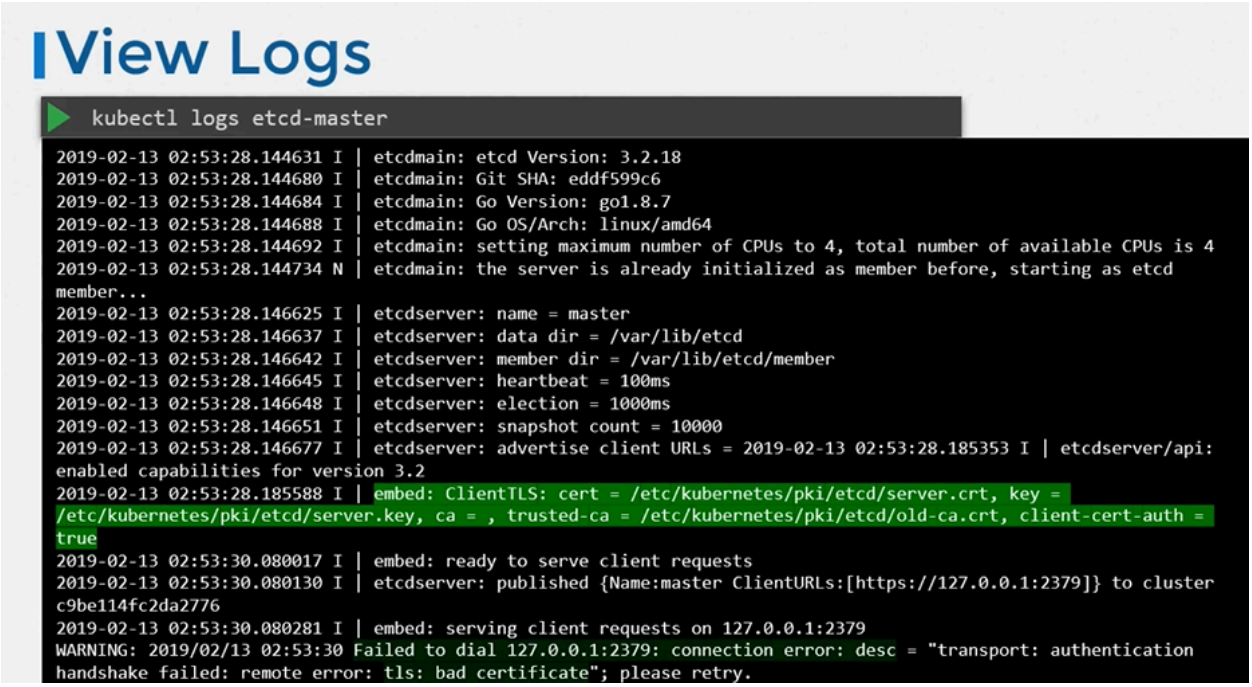
/etc/kubernetes/pki/apiserver.crt

```
openssl x509 -in /etc/kubernetes/pki/apiserver.crt -text -noout
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 3147495682089747350 (0x2bae26a58f090396)
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: CN=kubernetes
        Validity
            Not Before: Feb 11 05:39:19 2019 GMT
            Not After : Feb 11 05:39:20 2020 GMT
        Subject: CN=kube-apiserver
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:d9:69:38:80:68:3b:b7:2e:9e:25:00:e8:fd:01:

                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Key Usage: critical
                Digital Signature, Key Encipherment
            X509v3 Extended Key Usage:
                TLS Web Server Authentication
            X509v3 Subject Alternative Name:
                DNS:master, DNS:kubernetes, DNS:kubernetes.default,
DNS:kubernetes.default.svc, DNS:kubernetes.default.svc.cluster.local, IP
Address:10.96.0.1, IP Address:172.17.0.27
```

kubeadm

Certificate Path	CN Name	ALT Names	Organization	Issuer	Expiration
/etc/kubernetes/pki/apiserver.crt /etc/kubernetes/pki/apiserver.key	kube-apiserver	DNS:master DNS:kubernetes DNS:kubernetes.default DNS:kubernetes.default.svc IP Address:10.96.0.1 IP Address:172.17.0.27		kubernetes	Feb 11 05:39:20 2020
/etc/kubernetes/pki/ca.crt	kubernetes			kubernetes	Feb 8 05:39:19 2029
/etc/kubernetes/pki/apiserver-kubelet-client.crt /etc/kubernetes/pki/apiserver-kubelet-client.key	kube-apiserver-kubelet-client		system:masters	kubernetes	Feb 11 05:39:20 2020
/etc/kubernetes/pki/apiserver-etcd-client.crt /etc/kubernetes/pki/apiserver-etcd-client.key	kube-apiserver-etcd-client		system:masters	self	Feb 11 05:39:22 2020
/etc/kubernetes/pki/etcd/ca.crt	kubernetes			kubernetes	Feb 8 05:39:21 2017





Certificate api

The **Certificate API** allows Kubernetes to **manage certificate signing requests (CSRs)** in a standardized and secure way instead of manually signing certificates using OpenSSL**. mTLS automatically uses certificates to secure connections; the Certificate API controls how those certificates are issued and approved**

- **In** summary: The cluster CA is created at bootstrap, but certificate issuance is gated by approval; mTLS only begins after a certificate exists.
    - Cert issuance policy: Kubernetes must decide **Who is allowed to receive a certificate.**
    - For users: Requires CSR approval, which can either be Manual or policy-based
    - For internal components: Automatically approved

- A Certificate Authority (CA) consists of: CA private key, CA certificate
- Anyone with access to the CA private key can sign certificates that are trusted by the Kubernetes cluster.

- In Kubernetes:
    - The CA is usually generated during cluster setup
    - It is used to sign certificates for:kube-apiserver, kubelets, users, other components

Kubernetes Certificate API

- Kubernetes provides a built-in CertificateSigningRequest (CSR) API
- Instead of manually signing certificates:
    - submit CSRs to Kubernetes
    - An administrator (or automated controller) approves them
    - Kubernetes signs them using the cluster CA

Workflow:

- Step 1: User Generates a Private Key. **openssl genrsa -out jane.key 2048**
- Step 2: User Creates a CSR. **openssl req -new -key jane.key -subj "/CN=jane" -out jane.csr**
- Step 3: Sends the request to the administrator who creates a CSR object, with kind as "CertificateSigningRequest" and a encoded "jane.csr"

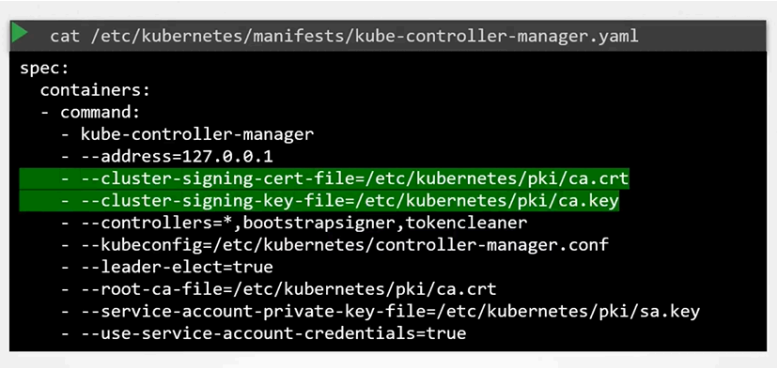
```
apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: jane
spec:
  groups:
  - system:authenticated
  usages:
  - digital signature
  - key encipherment
  - server auth
  request:
    <certificate-goes-here>
```

$ cat jane.csr |base64 $ kubectl create -f jane.yaml

- **S**tep 4: View Pending CSRs. **$ kubectl get csr**
- Step 5: approve CSRs. $ kubectl certificate approve jane
- Step 6: Retrieve the Signed Certificate kubectl get csr jane -o yaml
    - to decode it: $ echo "<certificate>" |base64 --decode

All the certificate related operations are carried out by the **controller manager.**

- If anyone has to sign the certificates they need the CA Servers, root certificate and private key. The controller manager configuration has two options where you can specify this.



    - To clarify The administrator approves certificate requests as a trust decision, while the controller manager automatically signs approved CSRs using the cluster CA. It Watches CertificateSigningRequest objects, Detects when a CSR is approved, Uses the cluster CA private key, Signs the certificate, Writes it back into the CSR object

Kubeconfig

What is kubeconfig?

I**t is a client-side configuration file.** A kubeconfig file tells kubectl:

- Which cluster to talk to
- Which user credentials to use
- Which namespace & cluster combination to use by default

<u>How kubectl Authenticates (Without kubeconfig)</u>

A client can directly call the Kubernetes API using: Client certificate, Private key, API server endpoint, CA certificate

- Example (conceptual): **curl --cert user.crt --key user.key https://api-server:6443**

kubectl does the same thing, but more conveniently.

- Instead of passing: Certificate paths, Keys, Server URLs on every command, kubeconfig stores them once in a file.
    - Example: kubectl get pods --kubeconfig=config

<u>kubeconfig File Structure</u>

A kubeconfig file has three main sections:

1. **Clusters**. Defines where the cluster is: API server address, CA certificate

```
clusters:
- name: production
  cluster:
    server: https://api.prod:6443
    certificate-authority: ca.crt
```

2. **Users**. Defines who you are: Client certificate, Client private key, token-based auth

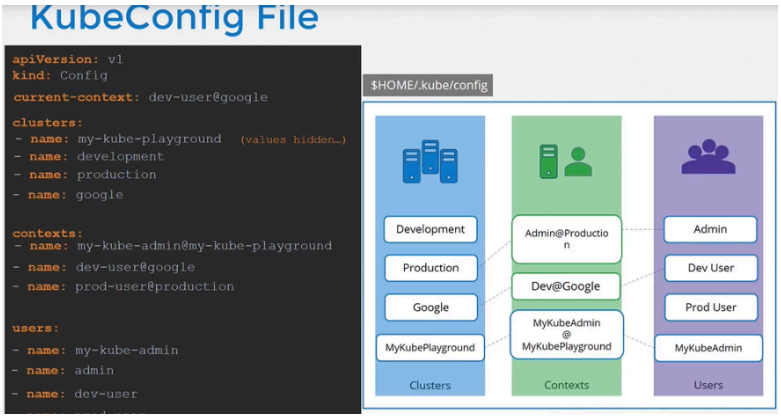
```
users:
- name: prod-user
  user:
    client-certificate: user.crt
    client-key: user.key
```

3. **Contexts:** Binds user + cluster + namespace together.

```
contexts:
- name: prod-user@production
  context:
    cluster: production
    user: prod-user
    namespace: default
```

How kubectl Chooses What to Use

- A context selects: Cluster, User, Namespace
- Only one context is active at a time
-

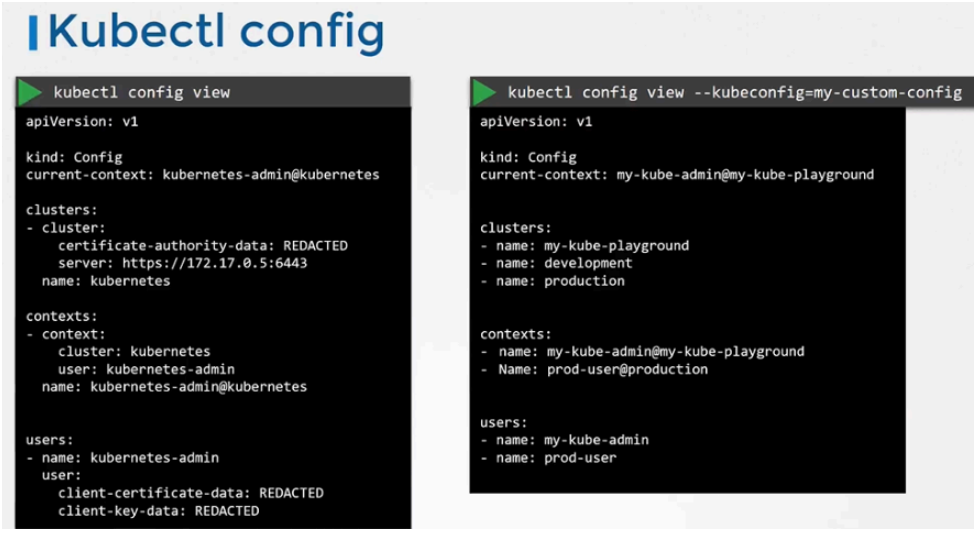


Viewing kubeconfig

- View the currently used kubeconfig: **kubectl config view**
- Specify a different config file: **kubectl config view --kubeconfig=my-custom-config**
- Default location: ~/.kube/config

Changing Contexts

- Switch current context: kubectl config use-context <context-name>
- Example: kubectl config use-context prod-user@production



Namespaces in kubeconfig

- Namespace is stored in the context
- Avoids typing -n <namespace> every time

- Override namespace per command: kubectl get pods **-n kube-system**
- Kubectl config -h (for help)

Certificates in kubeconfig

- Certificates can be: Referenced by file path or Embedded as base64 data
- Example (embedded):

```
client-certificate-data: <base64-cert>
client-key-data: <base64-key>
```

Used commonly in: Managed clusters, Portable kubeconfig files

In Summary:

- kubeconfig is a client configuration, not a cluster resource
- It does not create access, it uses existing credentials
- Context = cluster + user + namespace
- kubectl always operates in the current context
- A kubeconfig file defines how a client authenticates to and interacts with a Kubernetes cluster by mapping users, clusters, and contexts.

API Groups

Storage

Filesystem vs Directory

- Filesystem: A filesystem is a self-contained hierarchy (tree) that defines how files and directories are stored and managed on a storage device (or virtual device). Examples: ext4, NTFS, tmpfs.
- Directory: A directory is an object/entry inside a filesystem. It is simply a mapping from names to other objects (files, directories, symlinks, etc.).

How Mounting Works

- The OS has one global directory tree (e.g., starting at / on Unix).
- Mounting a filesystem attaches a different filesystem at an existing directory path (the mount point).
- When mounted:
    - The directory's original contents are hidden, not deleted.
    - Accessing that path now shows the root of the mounted filesystem.
    - When unmounted:

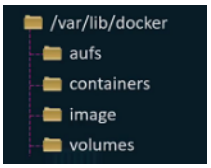
        - The mounted filesystem is detached.
        - The original directory contents become visible again.

Key Mental Model

- A directory is a location, a filesystem is a tree, and mounting swaps which tree you see at that location.
- Directories live inside filesystems; filesystems are mounted onto directories, temporarily replacing what you see at that path without destroying the underlying data.

Storage in Docker

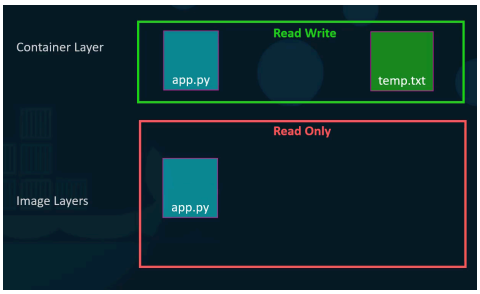
- When you install Docker on a system, it creates this directory structures at /var/lib/docker where Docker stores all its data by default



- files related to containers are stored under the containers directory and the files related to images are stored under the image directory. Any volumes created by the Docker containers are created under the volumes directory.

Layered Architecture

- When docker builds images, it builds these in a layered architecture. Each line of instruction in the Docker file creates a new (read-only) layer in the Docker image with just the changes from the previous layer.
- During builds, Docker reuses cached layers if instructions haven't changed
- If only source code or the entry point changes, only the affected layers are rebuilt
- This makes image builds faster and more storage-efficient
- Image layers are immutable (read-only) after build and shared by all containers using the image
- Running a container adds a writable container layer on top of the image layers
- The writable layer stores logs, temporary files, and runtime modifications
- Files in image layers cannot be edited directly
- When a file from an image layer is modified, Docker uses copy-on-write:
    - the file is copied into the writable layer
    - all changes apply only to this copy



- The original image remains unchanged unless rebuilt
- Destroying a container removes the writable layer and all its data
- Image layers persist and can be reused by future containers

Docker Volumes (Volume Mounts)

- A Docker volume is storage managed by Docker
- Created explicitly with:
    - **docker volume create data_volume**
    - **Docker volume ls**
- Stored by default under:
    - **/var/lib/docker/volumes/**
- Volumes can also be auto-created by Docker if they don't exist at container start
- Mounted into a container using:
    - -v volume_name:/container/path
    - or --mount type=volume

- Volume data persists even if the container is destroyed
- Volumes are independent of container lifecycle
- Best suited for application data, databases, and production use

Bind Mounts

- A bind mount maps an existing directory or file on the host directly into the container
- The host path is chosen and managed by the user
- Mounted using:
    - -v /host/path:/container/path or --mount type=bind
    - **$ mkdir -p /data/mysql**
    - **$ docker run --mount type=bind,source=/data/mysql,target=/var/lib/mysql mysql**

- Data is stored outside Docker's volume directory
- Changes are immediately reflected on both host and container
- Commonly used for development, config files, or when data already exists on the host

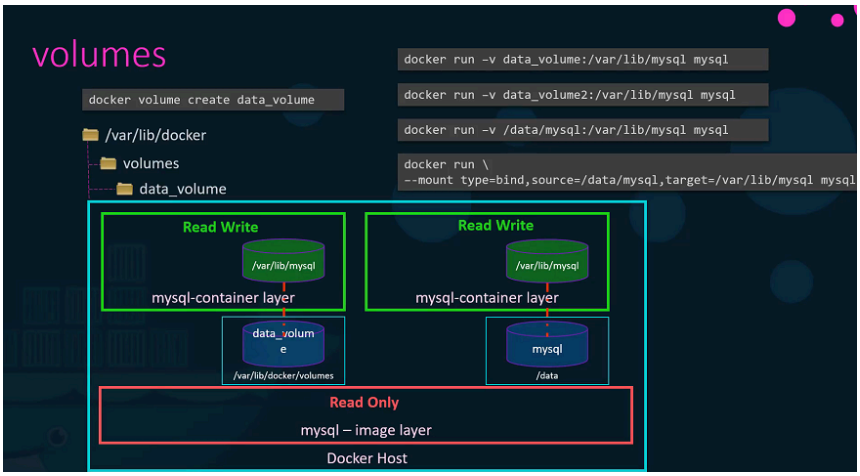
Key Distinction: Volume vs Bind Mount

- Both map host storage into the container filesystem (functionally similar at runtime)
- Volume mount:
    - Docker-managed
    - Stored in Docker's internal directory
    - Portable and safer (permissions, isolation)
- Bind mount:
    - User-managed
    - Uses arbitrary host paths

    - Tightly coupled to host filesystem structure

Storage Drivers

- Docker storage drivers handle: Layered image architecture, Read-only image layers, Writable container layer, Copy-on-write behavior
    - common Storage Drivers: AUFS, ZFS, BTRFS, Device Mapper, Overlay, Overlay2
    - The Selection of the storage drivers depends on the underlying OS. Docker will choose the best storage driver available automatically based on the operating system.
- Mounts (volumes/binds) bypass the container's writable layer, enabling persistence



Volume Driver Plugins in Docker

- As weve seen, docker uses diff drivers for different storage responsibilities
- Storage drivers: Manage image layers, container writable layers, and copy-on-write but **Do not manage persistent volumes**

Volume Drivers: volumes are managed by volume driver plugins, not storage drivers

- The default volume driver is local
- The local driver: Creates volumes on the Docker host
    - Stores data under: /var/lib/docker/volumes/

Third-Party Volume Driver Plugins

- Docker supports external volume drivers for persistent storage beyond the host
- These drivers provision and manage storage on external systems, such as:
    - AWS EBS (e.g. RexRay), Azure File Storage, Google Persistent Disks, DigitalOcean Block Storage, Portworx
- Useful for: Cloud deployments, Stateful applications, Data durability beyond a single host

Using a Specific Volume Driver

- You can specify a volume driver when running a container, for example Example (AWS EBS via RexRay):
    - docker run --volume-driver rexray/ebs --mount src=ebs-vol,target=/var/lib/mysql mysql

- Docker:
    - Requests the driver to provision or attach the volume
    - Mounts it into the container

- When the container stops:
    - The volume remains intact on the external storage provider

Container Storage Interface (CSI)

- The container storage interface was developed to support multiple storage solutions. With CSI, you can now write your own drivers for your own storage to work with Kubernetes. Portworx, Amazon EBS, Azure Disk, GlusterFS etc.

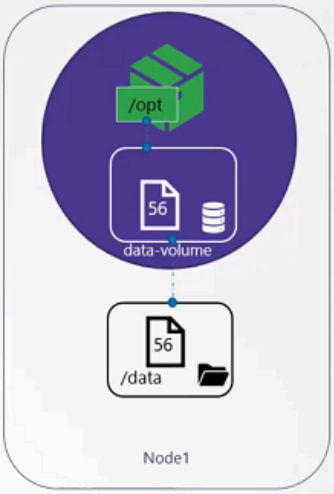
- CSI is not a Kubernetes specific standard. It is meant to be a universal standard and if implemented, allows any container orchestration tool to work with any storage vendor with a supported plugin. Kubernetes, Cloud Foundry and Mesos are onboard with CSI.
- It defines a set of RPCs or remote procedure calls that will be called by the container orchestrator. These must be implemented by the storage drivers.
- (focuses on volume drivers)

Persisting Data in Kubernetes

- Pods are also ephemeral: Data inside the Pod is lost when the Pod is deleted unless a volume is used. For example:
    - Assume create a pod that generates random numbers and writes data to file at /opt/num.out. To mount to host on /data
    - We create a volume for that. In this case I specify a path **/data** on the host. Files are stored in the directory data on my **node**. We use the **volumeMounts** field in

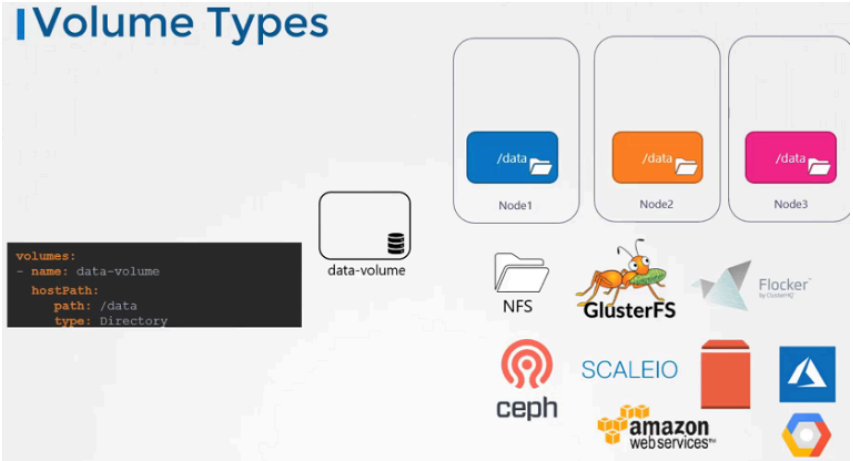
each container to mount the data-volume to the directory /opt within the container. The random number will now be written to **/opt mount** inside the container, which happens to be on the data-volume which is in fact /data directory on the host. When the pod gets deleted, the file with the random number still lives on the host.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: random-number-generator
spec:
  containers:
  - image: alpine
    name: alpine
    command: ["/bin/sh","-c"]
    args: ["shuf -i 0-100 -n 1 >> /opt/number.out;"]
    volumeMounts:
    - mountPath: /opt
      name: data-volume

  volumes:
  - name: data-volume
    hostPath:
      path: /data
      type: Directory
```



Volume Storage Options

- In the volumes, hostPath volume type is fine with the single node**. It's not recommended for use with the multi node cluster.**
    - hostPath is local to a node; even if directories exist on other nodes, the data is not shared or synchronized, making it unsafe for multi-node workloads.
- In the Kubernetes, it supports several types of standard storage solutions such as NFS, GlusterFS, CephFS or public cloud solutions like AWS EBS, Azure Disk or Google's Persistent Disk.



Persistent Volumes (PVs) in Kubernetes

- Problem PVs solve:
    - Without PVs, users must configure storage for every Pod individually
    - Any changes require updating all Pod definitions

- Persistent Volume (PV):
    - A cluster-wide pool of storage pre-configured by an administrator
    - Decouples storage provisioning from Pod deployment
    - Users can request storage via Persistent Volume Claims (PVCs) without knowing underlying details

- Key characteristics of PVs:
    - Cluster-scoped — available to all Pods in the cluster
    - Can have different storage backends: hostPath, NFS, cloud volumes, etc.
    - Access modes define how the volume can be mounted:

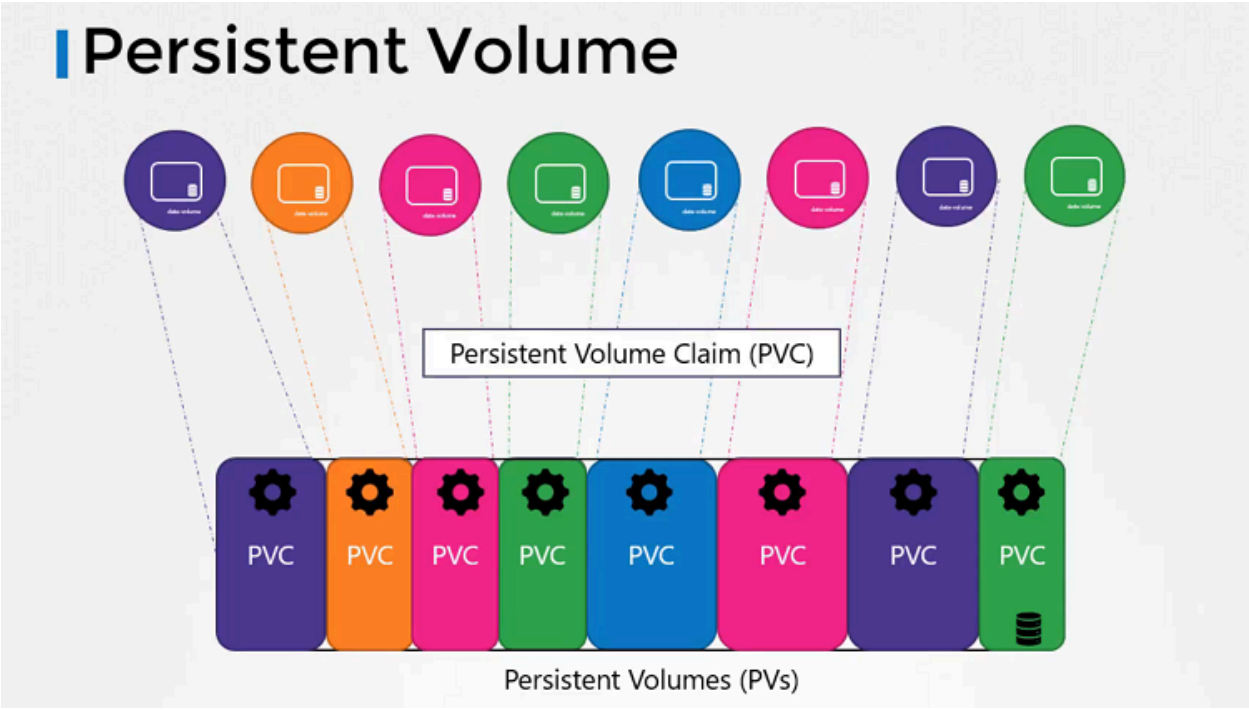
ReadWriteOnce (RWO)
- Mounted read-write by a single node at a time
- Multiple Pods on the same node can read/write
- Pods on other nodes cannot mount the volume simultaneously

ReadOnlyMany (ROX)
- Mounted read-only by multiple nodes
- Multiple Pods across nodes can read the data
- Writes are not allowed

ReadWriteMany (RWX)
- Mounted read-write by multiple nodes
- Multiple Pods across nodes can read and write simultaneously
- Requires shared storage (e.g., NFS, GlusterFS, cloud file storage)

Lifecycle commands:
- Create PV: kubectl create -f pv-definition.yaml
- List PVs: kubectl get pv
- Delete PV: kubectl delete pv pv-vol1

Key benefits:
- Simplifies storage management for large clusters
- Separates storage admin duties from application deployment
- Makes storage reusable, persistent, and centrally managed




```
pv-definition.yaml

kind: PersistentVolume
apiVersion: v1
metadata:
  name: pv-vol1
spec:
  accessModes: [ "ReadWriteOnce" ]
  capacity:
   storage: 1Gi
  hostPath:
   path: /tmp/data
```

```
$ kubectl create -f pv-definition.yaml
persistentvolume/pv-vol1 created

$ kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
pv-vol1   1Gi        RWO            Retain           Available                                    3min

$ kubectl delete pv pv-vol1
persistentvolume "pv-vol1" deleted
```

Persistent Volume Claims (PVCs) in Kubernetes

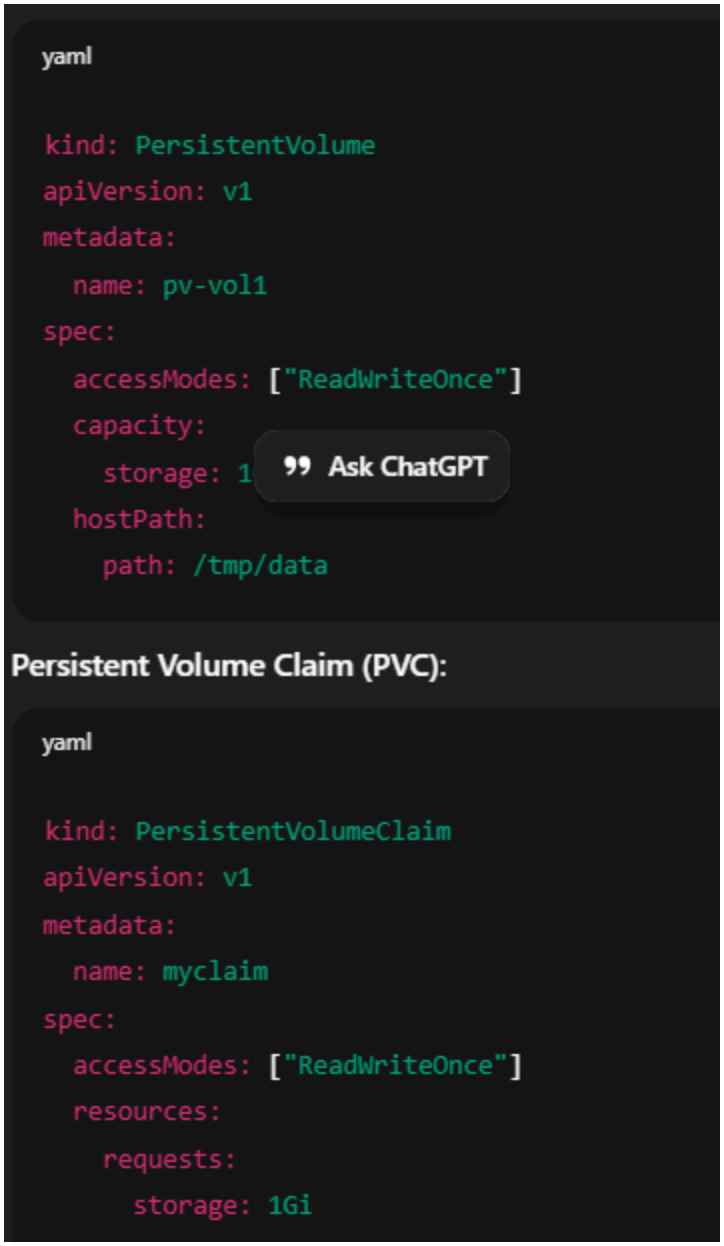
- Purpose:
    - PVCs let users request storage from cluster-wide Persistent Volumes (PVs) without knowing the underlying storage details

    - Separates user storage requests from administrator-managed PVs

- PVC vs PV:
    - PV: cluster-scoped, created and managed by admins, represents actual storage
    - PVC: namespace-scoped, created by users, requests storage from PVs

- Binding process:
    - When a PVC is created, Kubernetes matches it to a PV based on:
        - Access modes (RWO, ROX, RWX)
        - Requested capacity
        - Storage class (if used)

    - If no matching PV exists → PVC status is Pending
    - Once matched → PVC status becomes Bound



sing PVCs in Pods

- Pods use PVCs as volumes:
    - PVC must exist in the same namespace as the Pod
    - Kubernetes resolves the PVC → mounts the backing PV → makes it available to the Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: web
  volumes:
    - name: web
      persistentVolumeClaim:
        claimName: myclaim
```

Why Access Mode Appears in Pod/VolumeMount

1. PVC enforces access mode
    - When you create a PVC, you request storage with a certain access mode (RWO, ROX, RWX)
    - Kubernetes binds a matching PV that satisfies this access mode

2. Pod inherits the access mode via the PVC
    - When a Pod mounts the PVC:
        - Kubernetes ensures the Pod cannot exceed the access restrictions of the underlying PV
        - Example:
            - PV is ReadOnlyMany → Pod can only read, attempts to write will fail
            - PV is ReadWriteOnce → Pod can read/write if it's on the same node as the PV

3. Access mode in the Pod spec is a declarative check
    - You don't re-specify the mode in the Pod manually — it comes from the PVC you reference
    - Kubernetes uses it to enforce correct mounting and prevent access violations

Q: How do Pods get data from a read-only Persistent Volume (ROX PV) if they cannot write to it?

A: The data must exist before the Pods mount it. It can be:

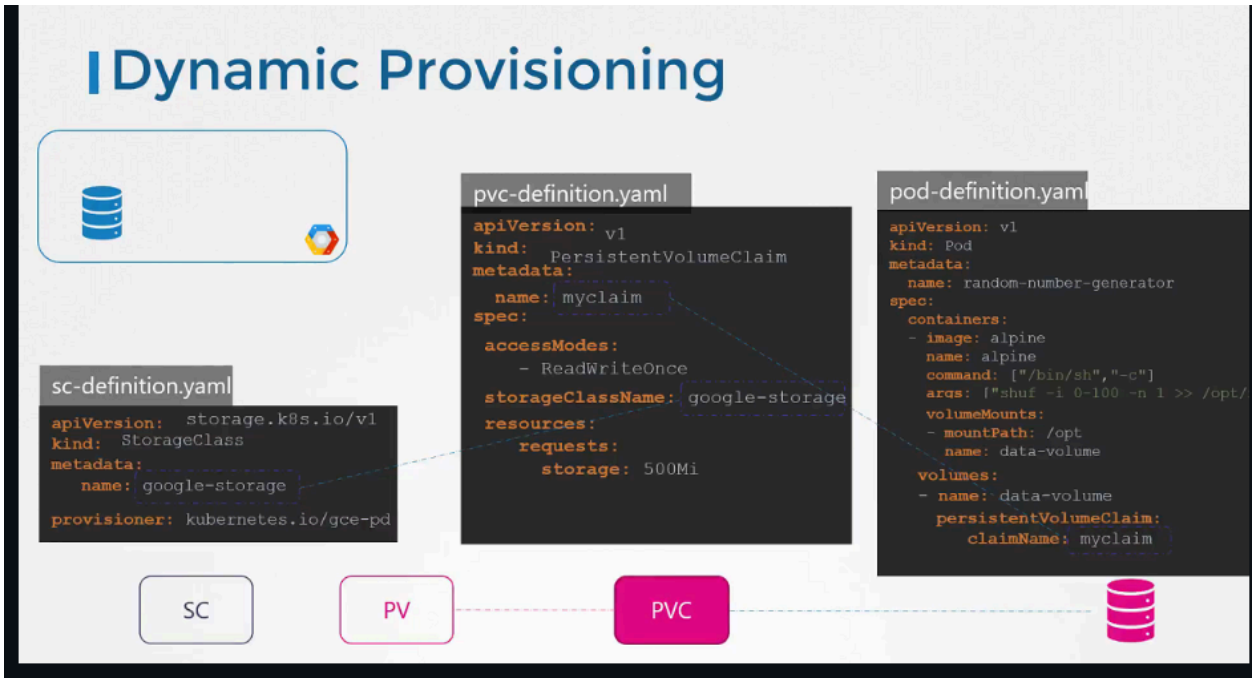
- Pre-populated by the cluster administrator on the underlying storage
- Populated via an init container before the main container runs
- Stored on external or cloud storage that already contains the data
  Pods then mount the ROX PV read-only, allowing multiple Pods to safely access the same data without modifying it.

Storage Class and Dynamic Provisioning

- Problem with static provisioning:
  - PVs must be created manually for each Pod before use
  - Tedious for cloud-backed storage (AWS EBS, GCP PD, Azure Disk) were for example . We need to first create disk in the Google Cloud as an example. This is done manually each tim ewen we define in the pod definition file
  - 

  - Every PVC references a pre-created PV
- Dynamic provisioning with Storage Class:

- Automates PV creation when a PVC requests storage
- No need to manually create PVs beforehand
- PVC specifies the storageClassName to trigger automatic provisioning




```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   name: google-storage
provisioner: kubernetes.io/gce-pd
```

Storage Class

sc-definition.yaml

```yaml
apiVersion:  storage.k8s.io/v1
kind:  StorageClass
metadata:
   name: google-storage
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard [ pd-standard | pd-ssd ]
  replication-type: none [ none | regional-pd ]
```

Volume Plugin
AWSElasticBlockStore
AzureFile
AzureDisk
CephFS
Cinder
FC
FlexVolume
Flocker
GCEPersistentDisk
Glusterfs
iSCSI
Quobyte
NFS
RBD
VsphereVolume
PortworxVolume
ScaleIO
StorageOS
Local