Resource request and limit:

To set resource limits for a Deployment in Kubernetes, you need to specify the resource requests and limits in the pod template within the Deployment's YAML configuration. Resource limits define the maximum amount of CPU and memory that each pod within the Deployment can consume. Resource requests are used to reserve resources for the pod.

        resources:

          requests:

            cpu: "0.5"   # Requests 0.5 CPU cores

            memory: "256Mi"  # Requests 256 megabytes of memory

          limits:

            cpu: "1"   # Limits CPU to 1 CPU core

            memory: "512Mi"  # Limits memory to 512 megabytes

* The **requests** section specifies the minimum resources that the pod requests. These resources are used by the Kubernetes scheduler to find a suitable node for the pod.
* The **limits** section sets the maximum resources that the pod is allowed to consume. If a container in the pod exceeds these limits, it may be terminated or throttled, depending on the resource type.

Autoscaling in Kubernetes:

Horizontal pod autoscaler (HPA):

Vertical Pod Autoscaling(VPA)

StatefulSet

A StatefulSet is a type of workload in Kubernetes that is used to manage stateful applications. It provides guarantees about the ordering and uniqueness of pods and is particularly well-suited for stateful applications like databases and distributed systems. Here are some key features and concepts related to StatefulSets:

**Pod Ordering:** StatefulSets maintain a stable, unique identifier for each pod it manages, which is based on an index starting from 0. This ensures that pods are created, updated, and deleted in a predictable and ordinal manner.

apiVersion: apps/v1

kind: StatefulSet

metadata:

  name: web

spec:

  serviceName: "nginx"   # Headless service for network identity

  replicas: 3

  selector:

    matchLabels:

      app: nginx

  template:

    metadata:

      labels:

        app: nginx

    spec:

      containers:

      - name: nginx

        image: nginx

**DaemonSet**

A DaemonSet is used to ensure that a specific pod runs on every node in the cluster, regardless of node attributes. It is typically used for running system-level services, infrastructure components, or background tasks on all nodes.

Scope:

A DaemonSet is a type of controller that operates at the cluster level. It automatically manages the deployment of a pod to every node in the cluster, ensuring that one instance runs on each node without requiring explicit node attributes.

Usecase: DaemonSet: DaemonSets are used for deploying background or system-level services that need to exist on every node. Common use cases include log collectors, monitoring agents, security agents, networking components, and other infrastructure services.

Placement: A DaemonSet guarantees that one pod runs on every node in the cluster. It is not concerned with node attributes but ensures that the pod is deployed everywhere, regardless of the nodes' attributes.

apiVersion: apps/v1

kind: DaemonSet

metadata:

  name: example-daemonset

spec:

  selector:

    matchLabels:

      app: example-app

  template:

    metadata:

      labels:

        app: example-app

    spec:

      containers:

      - name: example-container

        image: example-image:latest

**Node Affinity**:

**Purpose**: Node Affinity is used to influence pod placement by specifying preferences or requirements for scheduling pods on nodes based on node attributes or characteristics.

**Matching** **Nodes**:

**Preferred Node Affinity**: In the case of preferred node affinity, the scheduler tries to place pods on nodes that match the specified criteria but is not restricted to only those nodes. If no matching nodes are available, pods can still be scheduled elsewhere.

**Required Node Affinity**: Required node affinity enforces that pods are scheduled only on nodes that meet the specified criteria. If no matching nodes are available, the pod remains unscheduled.

**Examples:**

You might use Node Affinity to ensure that pods are scheduled on nodes with specific hardware capabilities or in specific availability zones.

**Node Anti-Affinity:**

Purpose: Node Anti-Affinity is used to influence pod placement by specifying preferences or requirements for avoiding scheduling pods on nodes with specific attributes or characteristics.

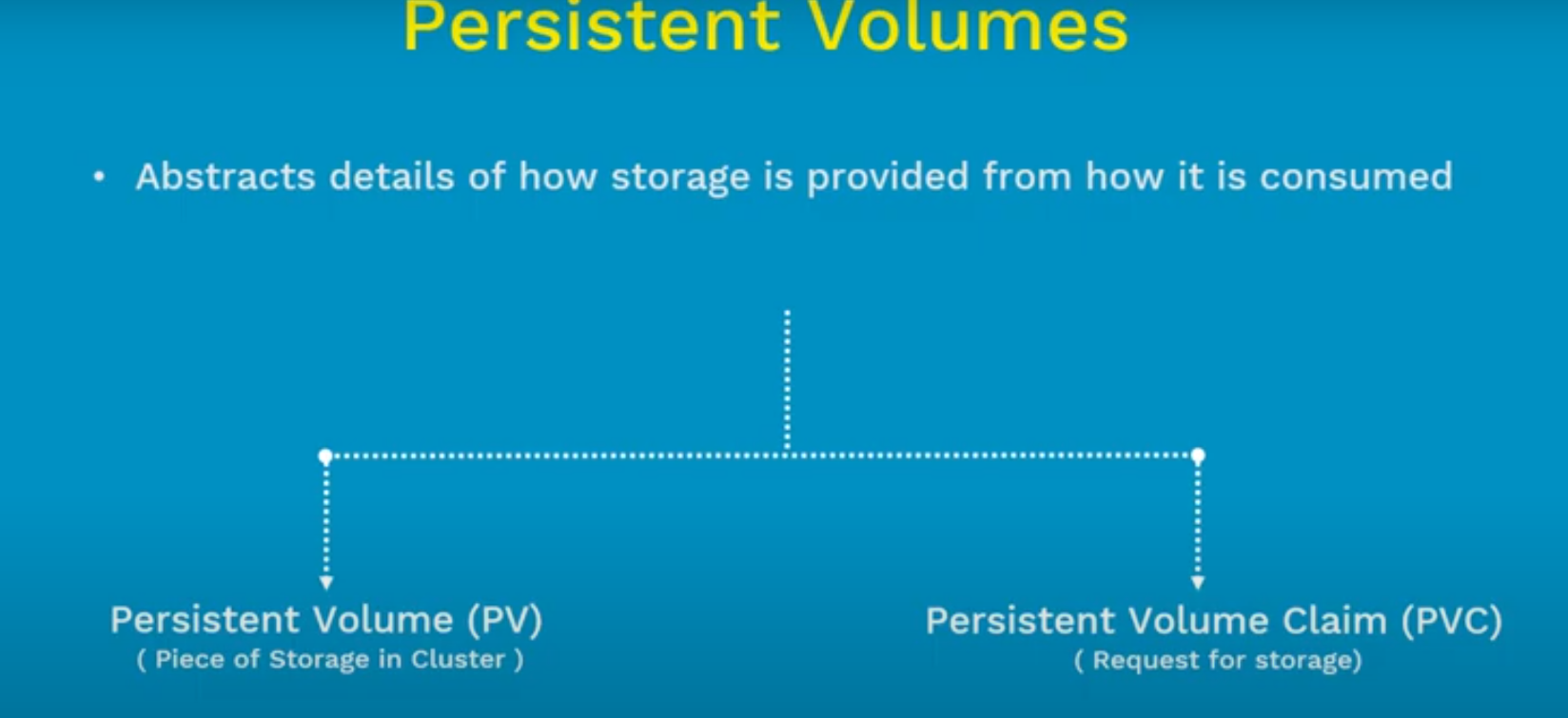
**Matching Nodes:**

Node Anti-Affinity rules specify nodes to be avoided, based on labels or other attributes. The scheduler tries to prevent scheduling pods on nodes that match the criteria.

**Examples:**

You might use Node Anti-Affinity to ensure that pods from the same application or service are spread across nodes to enhance fault tolerance.

PV AND PVC:



Config map and secrets:

In Kubernetes, both ConfigMaps and Secrets are resources used for configuring and managing application settings, but they differ in terms of their use cases, data sensitivity, and storage.

ConfigMap:

Use Case: ConfigMaps are used for storing non-sensitive configuration data, such as environment variables, command-line arguments, or configuration files.

Data Format:

ConfigMap data is typically stored as key-value pairs.

Visibility:

ConfigMap data is visible in clear text within the cluster, and it is not designed for storing sensitive information.

Examples of Use:

Storing database connection strings.

Defining environment variables for applications.

Access:

ConfigMaps are mounted as volumes or exposed as environment variables inside pods.

**Secret**:

Use Case: Secrets are specifically designed for managing sensitive information, such as passwords, API keys, or certificates.

Data Format:

Secret data is also stored as key-value pairs, but the data is base64-encoded by default, providing a level of obfuscation.

Visibility:

Secrets are intended to store sensitive information, and their data is encoded, but they are not encrypted. While base64 encoding provides a layer of encoding, it is not secure against determined attackers.

Examples of Use:

Storing database passwords.

Managing API keys for external services.

Storing TLS certificates.

Access:

Secrets can be mounted as volumes or exposed as environment variables inside pods, similar to ConfigMaps.

Key Differences:

Security:

ConfigMaps are designed for non-sensitive configuration data and are visible in clear text.

Secrets are specifically designed for sensitive data and are base64-encoded for a minimal level of obfuscation.

Data Sensitivity:

ConfigMaps are suitable for non-sensitive information.Secrets are explicitly meant for sensitive data that requires additional protection.

Encoding:

ConfigMap data is stored in plain text.

Secret data is base64-encoded by default.

Use Cases:

ConfigMaps for general configuration settings.

Secrets for sensitive information.

When using ConfigMaps or Secrets, it's important to consider the nature of the data being stored. For highly sensitive information, additional security measures, such as encryption or third-party secret management tools, may be necessary. Always follow security best practices and consider the sensitivity of the data when choosing between ConfigMaps and Secrets.

Liveness Probe & Readiness Probe:

In Kubernetes, "readiness" refers to the state of a container or a Pod indicating whether it is ready to serve traffic or handle requests. It is a crucial concept for ensuring the reliability and stability of applications running in a Kubernetes cluster.

Kubernetes introduces the idea of readiness probes, which are a type of health check performed by the Kubernetes system to determine if a container is ready to handle requests. These probes help Kubernetes manage the lifecycle of Pods and ensure that only healthy Pods receive traffic.

There are three types of probes in Kubernetes:

**Liveness Probe**: Checks whether the container is running properly.

**Readiness Probe:**  Checks whether the container is ready to serve traffic.

**Startup Probe**: Introduced in Kubernetes 1.16, it checks whether the container has started successfully.

**Readiness Probe:**

* It is used to indicate when a container is ready to start accepting traffic.
* If a container fails the readiness probe, Kubernetes considers it not ready, and it won't receive traffic through the associated service.
* This helps prevent sending requests to containers that are still initializing or experiencing issues.
* The readiness probe is defined in the Pod specification and can be configured to execute various types of checks, such as HTTP endpoint checks, TCP socket checks, or command execution checks.

Example of a Pod specification with a readiness probe:

apiVersion: v1

kind: Pod

metadata:

  name: mypod

spec:

  containers:

  - name: mycontainer

    image: myimage

    ports:

    - containerPort: 80

    readinessProbe:

      httpGet:

        path: /healthz

        port: 8080

      initialDelaySeconds: 10

      periodSeconds: 5

In this example, the readiness probe checks the /healthz HTTP endpoint on port 8080, and it starts 10 seconds after the container starts. The probe is repeated every 5 seconds. If the probe fails, Kubernetes considers the container not ready, and it won't receive traffic until it passes the probe.

Taint and toleration:

n Kubernetes, taints and tolerations are mechanisms used to control which nodes can run particular Pods. They are used to influence the scheduling decisions made by the Kubernetes control plane.

Taints:

* Definition: A taint is a property applied to a node that repels (or "taints") the node. It specifies that certain conditions must be met for a Pod to be scheduled on that node.
* Purpose: Taints are typically used to repel Pods from nodes that have specific characteristics, such as being reserved for certain workloads or having particular hardware requirements.
* Syntax: Taints have a key, value, and effect. The key and value are arbitrary strings, and the effect can be one of NoSchedule, PreferNoSchedule, or NoExecute.
* Example: To add a taint to a node, you might use the following command:
* kubectl taint nodes <node-name> key=value:effect

Toleration:

Tolerations:

* Definition: A toleration is a specification added to a Pod that allows the Pod to be scheduled on nodes with specific taints. It essentially makes a Pod "tolerate" nodes with certain taints.
* Purpose: Tolerations are used to make sure that specific Pods, despite the presence of taints on nodes, can still be scheduled on those nodes.
* Syntax: A toleration is specified in the Pod's YAML definition and includes key, value, effect, and optional tolerationSeconds (which indicates the effect duration).
* Example: To add a toleration to a Pod, you might use the following in the Pod's YAML:
* tolerations:
* - key: "key"
* operator: "Equal"
* value: "value"
* effect: "NoSchedule"

In summary, taints are applied to nodes, indicating conditions under which Pods should avoid those nodes. Tolerations, on the other hand, are specified in the Pod's configuration to allow the Pod to be scheduled on nodes with specific taints. Taints and tolerations work together to influence where Pods are scheduled in a Kubernetes cluster, providing a flexible way to handle node affinity and anti-affinity.