K8’S

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.31/

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.31/#deployment-v1-apps

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It enables developers to run their applications in a cluster of machines (nodes), ensuring high availability, load balancing, and efficient resource utilization. Here's a breakdown of the key components of Kubernetes:

**1. API Server**

The API Server is the central management component of Kubernetes. It exposes the Kubernetes API, which is used by the control plane, components, and external users (like `kubectl`) to interact with the cluster. It processes RESTful requests and ensures that the data is stored in etcd and is updated to reflect the desired cluster state.

**2. Kubelet**

Kubelet is an agent that runs on each worker node. It communicates with the API server to manage and ensure that containers in pods are running as expected. It also reports the node's status and monitors the health of pods, restarting them when necessary.

**3. Worker Node**

A Worker Node is a machine (virtual or physical) where application containers are deployed. It contains all the necessary services to run the containerized applications. Each worker node includes components like Kubelet, Kube Proxy, and a Container Runtime.

**4. Node**

A Node refers to an individual machine in the Kubernetes cluster, which can be either a Worker Node or a Master Node. A Kubernetes cluster consists of multiple nodes that share the workload.

**5. Master Node (Control Plane)**

The Master Node is the brain of the Kubernetes cluster. It manages the entire cluster by controlling the worker nodes and orchestrating tasks like scheduling, maintaining cluster health, and scaling applications. Key components on the master node include:

- API Server

- Controller Manager

- Scheduler

- etcd

- Kube Controller Manager

**6. etcd**

`etcd` is a distributed key-value store used by Kubernetes to store all the configuration data, cluster state, and metadata. It ensures the cluster state is always consistent across all components. It acts as the single source of truth for the cluster.

**7. Kube Controller Manager**

This component runs multiple controllers that manage different aspects of the cluster's lifecycle, such as:

- Node Controller: Manages node availability.

- Replication Controller: Ensures that the correct number of pods are running.

- Endpoints Controller: Manages service discovery by associating services with endpoints.

- Service Account & Token Controllers: Handle authentication and authorization.

**8. Kube Proxy**

Kube Proxy runs on each worker node and handles networking. It routes traffic to the appropriate container or pod, ensuring network rules and load balancing are applied to the services. It facilitates communication between services and pods within and outside the cluster.

**9. Container Runtime**

The container runtime is the software responsible for running containers. Kubernetes supports different runtimes, such as:

- Docker

- containerd

- CRI-O

The container runtime pulls images from container registries, launches containers, and manages their lifecycle.

**10. Pod**

A Pod is the smallest and simplest unit in the Kubernetes object model. It represents a single instance of a running process in the cluster. A pod can contain one or more containers, and all containers in a pod share the same network and storage resources.

**Minikube and its Behavior**

1. What is Minikube?

- Minikube is a tool that allows you to run Kubernetes clusters locally on your machine. It's ideal for testing and development environments because it sets up a single-node Kubernetes cluster on your local machine.

- Minikube supports multiple drivers (also known as hypervisors) such as Docker, Hyper-V, VirtualBox, and more to manage virtual machines and containers.

2. Minikube Driver Selection:

- By default, Minikube selects the Docker driver if Docker is installed on your machine. This happens even if other drivers like Hyper-V, SSH, or VirtualBox are available.

- If you're using Minikube on a local server, Docker is required to run Kubernetes. If you're using an online server like an EC2 instance, you need to install Docker there as well for Minikube to work.

- You can specify a different driver using the `--driver` flag when starting Minikube. For example, to use VirtualBox:

minikube start --driver=virtualbox

3. Running Minikube with Admin Privileges:

- Minikube often needs to be run with administrator or elevated privileges (such as `system2` or `sudo` on Linux) because it needs to access system-level resources to start and manage containers or VMs.

### Package Managers: Choco, Winget, and Scoop

1. Choco (Chocolatey):

- Chocolatey is a package manager for Windows that allows you to install software from the command line. It simplifies installing and updating software on Windows systems.

- Example of installing software with Chocolatey:

```bash

choco install minikube

2. Winget:

- Winget (Windows Package Manager) is another command-line package manager for Windows, provided by Microsoft. It is similar to Chocolatey but is integrated with Windows.

- Example of installing software with Winget:

```bash

winget install Minikube

3. Scoop:

- Scoop is another package manager for Windows that focuses on simplicity and less administrative overhead (it doesn't always require admin rights to install packages).

- Example of installing software with Scoop:

```bash

scoop install minikube

Each of these package managers provides a streamlined way to install, update, and manage software on Windows systems, making it easier to automate software management.

To begin using Kubernetes, you need to install two essential tools:

1. **Minikube**: A tool to run Kubernetes locally.
2. **kubectl**: The command-line tool for interacting with Kubernetes clusters.

Getting Started with Kubernetes (K8s): Minikube and kubectl Setup

To begin using Kubernetes, you need to install two essential tools:

1. Minikube: A tool to run Kubernetes locally.

2. kubectl: The command-line tool for interacting with Kubernetes clusters.

Deploying an NGINX Application in Kubernetes

We'll use an NGINX application from Docker Hub as an example for this deployment.

**Step 1: Create a Pod**

Pods are the basic units in Kubernetes that run containers.

Command: kubectl create deployment my-nginx --image=nginx:latest

This creates a deployment for the NGINX container.

Commands to verify the deployment:

- View Deployments: kubectl get deployments

- View Pods: kubectl get pods

- Access the Kubernetes Dashboard: minikube dashboard

**Step 2: Expose the Application**

The NGINX application listens on port 80. To access it locally, you need to expose it via a service.

Command: kubectl expose deployment my-nginx --port=80 --type=LoadBalancer

**Check the services and access the application:**

- View Services: kubectl get services

- Open the service: minikube service my-nginx

Deploying a Node.js Web Application in Kubernetes

To deploy a custom Node.js web app, follow these steps:

**Step 1: Create a Dockerfile**

Create a `Dockerfile` for the Node.js application. This file defines how to build your app's Docker image.

Dockerfile contents:

```Dockerfile

FROM node:20

WORKDIR /myapp

COPY . .

RUN npm install

EXPOSE 3000

CMD ["npm", "start"]

**Step 2: Build and Push the Docker Image**

1. Build the Docker image:

docker build -t <dockerhub-username>/<repository-name>:<version> .

Example: docker build -t vaibhawpandeydev082/my-containers:01 .

2. Check the Docker image:

docker images

3. Log in to Docker Hub (if not already):

docker login

4. Push the image to Docker Hub:

docker push <dockerhub-username>/<repository-name>:<version>

**Step 3: Deploy the Node.js App in Kubernetes**

1. Start Minikube if it's not running:

minikube status

minikube start

2. Create a deployment in Kubernetes using the custom Docker image:

kubectl create deployment my-nodeapp --image=<dockerhub-username>/<repository-name>:<version>

3. Expose the application to run on port 3000:

kubectl expose deployment my-nodeapp --port=3000 --type=LoadBalancer

4. View the services and access the app:

kubectl get services

minikube service my-nodeapp

Additional Commands

- View Minikube dashboard:

minikube dashboard

- View logs for a pod:

kubectl logs <pod-name>

- Delete a deployment:

kubectl delete deployment <deployment-name>

Updating the Application in Kubernetes (K8s Rollout)

If there are changes in the application (e.g., code updates), follow these steps to update the deployment:

1. Build a new Docker image with the updated version:

docker build -t <dockerhub-username>/<repository-name>:<updated-version> .

2. Push the updated image to Docker Hub:

docker push <dockerhub-username>/<repository-name>:<updated-version>

1. Update the deployment with the new image version in Kubernetes.

Update Without Any Downtime (Rolling Update in Kubernetes)

To update a deployment without downtime in Kubernetes, use the following command:

kubectl set image deployment <deployment\_name> <container\_name>=<new\_image\_name>:<new\_version>

- Find the container name: Navigate to the Kubernetes dashboard, go to Pods, and check the container name.

Key Advantage of Kubernetes:

Kubernetes will terminate the old image only after the new image is live and running, ensuring zero downtime during the update process.

### Roll Back (Revert Deployment)

In case you need to revert to a previous version or undo a failed update, you can perform a rollback.

1. Check rollout status to verify if the update was successful:

kubectl rollout status deployment <deployment\_name>

2. Get pods to monitor the running containers:

kubectl get pods

3. Roll back to the previous version:

kubectl rollout undo deployment <deployment\_name>

### Image Pull Backoff Issue

If you're trying to set an image version (e.g., `version 6`) that is not available in Docker Hub, you might encounter an "ImagePullBackOff" error. This indicates Kubernetes is unable to pull the specified image.

**### Steps for in-case Website Shutdown in Kubernetes**

**1. Check Pod/Deployment Status**

Use these commands to check for issues:

kubectl get pods

kubectl get deployments

**2. Inspect Pod Logs**

Check logs for errors causing crashes:

kubectl logs <pod\_name>

kubectl logs <pod\_name> -c <container\_name>

**3. Check Events**

View events to find warnings or errors:

kubectl get events

**4. Restart Pods**

Try restarting Pods to fix temporary issues:

kubectl rollout restart deployment <deployment\_name>

**5. Check Resource Limits**

Verify CPU/memory usage isn't causing problems:

kubectl describe pod <pod\_name>

**6. Check Probes**

Ensure your liveness and readiness probes are correct:

```yaml

livenessProbe: httpGet: path: /health port: 80

readinessProbe: httpGet: path: /ready port: 80

**7. Rollback Deployment**

Revert to a previous version if a new update fails:

kubectl rollout undo deployment <deployment\_name>

**8. Scale Deployment**

Adjust replicas to handle traffic surges:

kubectl scale deployment <deployment\_name> --replicas=<number>

**9. Check Cluster Health**

Make sure nodes are in `Ready` state:

kubectl get nodes

**10. Check Image Pull Issues**

Inspect image pull failures:

kubectl describe pod <pod\_name>

**11. Fix Code Issues**

If the issue is in the code, fix and redeploy:

kubectl set image deployment <deployment\_name> <container\_name>=<new\_image>

**### Why Scaling is Important in Kubernetes**

**1. Handle Traffic Spikes**

Scaling adds pods during high traffic and reduces them during low demand, improving resource use.

**2. Ensure High Availability**

Multiple pods provide redundancy, preventing downtime in case of failure.

**3. Optimize Performance**

Load balancing across pods improves performance and resource usage.

**4. Cost Efficiency**

Scale down to avoid overpaying for unused resources, scale up when needed.

**5. Improve Resilience**

Scaling across nodes makes your app more resilient to failures.

**6. Zero Downtime Updates**

Rolling updates maintain service availability during deployments.

**7. Meet SLAs**

Maintain performance to meet Service Level Agreements (SLAs) and enhance the user experience.

**8. Adaptability**

Kubernetes autoscaling adjusts resources dynamically as demand changes, ensuring long-term flexibility.

To handle multiple containers running in a single pod, you can define multiple containers within the same `containers` section of the pod specification. These containers can communicate with each other using the `localhost` network as they share the same network namespace, and they can also share storage volumes if needed.

Here’s an example of a Kubernetes deployment that runs both an NGINX and a Node.js application in a single pod:

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-nodejs-deployment # Name of the deployment

labels:

app: nginx-nodejs # Labels for this deployment

spec:

replicas: 2 # Number of pods to run

selector:

matchLabels:

app: nginx-nodejs # Select pods matching this label

template:

metadata:

labels:

app: nginx-nodejs # Assign labels to the pod template

spec:

containers:

- name: nginx # First container: NGINX

image: nginx:1.14.2 # Image version for NGINX

ports:

- containerPort: 80 # Exposing port 80 for the NGINX container

- name: nodejs # Second container: Node.js

image: your-dockerhub-username/your-nodejs-app-image:latest # Docker image for your Node.js app

ports:

- containerPort: 3000 # Exposing port 3000 for the Node.js container

**Key Points on Handling Multiple Containers in a Pod:**

1. Shared Networking:

All containers in the same pod share the same network namespace, meaning they can communicate with each other using `localhost`. For example, the NGINX container can proxy requests to the Node.js app using `localhost:3000`.

2. Shared Storage (if needed):

You can define shared volumes within the pod to allow the containers to share file storage. Example:

```yaml

spec:

volumes:

- name: shared-data # Defining a shared volume

emptyDir: {}

containers:

- name: nginx

image: nginx:1.14.2

volumeMounts:

- name: shared-data # Mounting the shared volume

mountPath: /usr/share/nginx/html

- name: nodejs

image: your-dockerhub-username/your-nodejs-app-image:latest

volumeMounts:

- name: shared-data # Mounting the shared volume in the Node.js container

mountPath: /usr/src/app

3. Container Lifecycle Management:

Kubernetes manages the containers' lifecycle in the pod as a unit. If any container fails, Kubernetes restarts the whole pod. It is important to consider the dependencies between the containers inside a pod and ensure they are designed to work together properly.

4. Logs and Monitoring:

Each container generates logs separately. You can view logs for each container in the pod by specifying the container name:

**kubectl logs <pod-name> -c nginx**

**kubectl logs <pod-name> -c nodejs**

5. Inter-Container Communication:

Inside the pod, containers can communicate with each other via `localhost` and their respective container ports. For example:

- The NGINX container can route traffic to `localhost:3000` where the Node.js container is running.

By running multiple containers in the same pod, you ensure they share resources, network, and storage (if required) while keeping them isolated from other pods. However, pods are designed to run tightly coupled containers that need to operate together.

Running multiple containers in the same pod can offer benefits like resource sharing and easier communication between containers, but it also has some notable disadvantages:

### 1. Tight Coupling:

- Issue: Containers in the same pod are tightly coupled and share the same lifecycle, meaning they must be deployed, scaled, and managed together.

- Impact: If one container fails, the whole pod will restart, impacting all the containers inside it, even if the other containers are functioning properly. This makes it harder to isolate issues related to specific containers.

### 2. Scaling Limitations:

- Issue: Pods scale as a unit.

- Impact: If one container in the pod requires more replicas, the whole pod must scale. This is inefficient because it forces unnecessary replication of containers that don't need scaling, wasting resources.

### 3. Resource Contention:

- Issue: Containers within a pod share resources like CPU and memory.

- Impact: A single container consuming excessive resources can affect the performance of other containers in the pod. Kubernetes manages resources at the pod level, not at the container level, leading to potential resource contention.

### 4. Security Risks:

- Issue: Containers in a pod share the same network and volume space.

- Impact: If one container is compromised (e.g., a vulnerability in one of the containers), it can potentially affect other containers in the pod, increasing security risks. The shared environment makes it harder to isolate containers from each other.

### 5. Complexity in Logs and Monitoring:

- Issue: All containers in the pod share the same logs and monitoring settings.

- Impact: Managing logs and monitoring multiple containers in a single pod can become complicated, as it requires distinguishing logs and metrics between containers. It may also lead to issues in troubleshooting.

### 6. Lifecycle Management Complexity:

- Issue: All containers in a pod have the same lifecycle.

- Impact: If containers have different lifecycle requirements (e.g., one needs to be updated more frequently than the other), it can be problematic. You would have to redeploy or restart the whole pod to update one container, even if the others do not need changes.

### 7. Limited Use Cases:

- Issue: Multi-container pods are designed for specific use cases like sidecar patterns (e.g., logging agents, proxies).

- Impact: In most applications, running multiple containers in a single pod is unnecessary and might complicate the architecture. For standard workloads, it is often more efficient to use separate pods for each container.

### 8. Deployment and Maintenance Complexity:

- Issue: Managing configuration, scaling, and deployment for multiple containers in the same pod can become complicated.

- Impact: Updates or configuration changes for one container in the pod require redeploying the entire pod, making maintenance and updates more challenging. This can lead to increased downtime and operational overhead.

### Summary of Disadvantages:

1. Tight Coupling (Shared lifecycle)

2. Limited scaling flexibility

3. Resource contention within the pod

4. Increased security risks

5. Complexity in logging and monitoring

6. Challenges in lifecycle management

7. Limited to specific use cases (e.g., sidecar containers)

8. Increased complexity in deployment and maintenance

For most applications, it's better to run separate containers in different pods, unless the containers are tightly coupled and need to share resources directly (e.g., sidecars like loggers or reverse proxies).

### Project Overview:

- Node.js app: A simple application that connects to a MongoDB database.

- MongoDB: A database that the Node.js app interacts with.

### **Step 1: Create Deployment and Service for MongoDB**

#### `mongodb-deployment.yaml`

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongodb-deployment

labels:

app: mongodb

spec:

replicas: 1

selector:

matchLabels:

app: mongodb

template:

metadata:

labels:

app: mongodb

spec:

containers:

- name: mongodb

image: mongo:4.4

ports:

- containerPort: 27017

env:

- name: MONGO\_INITDB\_ROOT\_USERNAME

value: root

- name: MONGO\_INITDB\_ROOT\_PASSWORD

value: password

---

apiVersion: v1

kind: Service

metadata:

name: mongodb-service

spec:

selector:

app: mongodb

ports:

- protocol: TCP

port: 27017

targetPort: 27017

type: ClusterIP

### Step 2: Create Deployment and Service for Node.js App

#### `nodejs-deployment.yaml`

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: nodejs-deployment

labels:

app: nodejs

spec:

replicas: 1

selector:

matchLabels:

app: nodejs

template:

metadata:

labels:

app: nodejs

spec:

containers:

- name: nodejs

image: your-dockerhub-username/nodejs-app:latest # Replace with your image

ports:

- containerPort: 3000

env:

- name: MONGO\_URL

value: mongodb-service:27017 # Reference the MongoDB service

---

apiVersion: v1

kind: Service

metadata:

name: nodejs-service

spec:

selector:

app: nodejs

ports:

- protocol: TCP

port: 3000

targetPort: 3000

type: LoadBalancer # Expose the Node.js app externally

### Step 3: Deploy the Application

1. Apply the MongoDB deployment and service:

**kubectl apply -f mongodb-deployment.yaml**

2. Apply the Node.js deployment and service:

**kubectl apply -f nodejs-deployment.yaml**

### Potential Configuration Issues and Solutions

#### 1. Issue: MongoDB Connection Failure

Symptom: The Node.js application can't connect to the MongoDB database.

Cause:

- The `MONGO\_URL` environment variable in the Node.js deployment might be incorrect, or the MongoDB service is not properly configured.

Solution:

- Verify that the MongoDB service name (`mongodb-service`) matches the one specified in the Node.js environment variable (`MONGO\_URL`).

- Use the command `kubectl get services` to check the service name and make sure it's correctly set.

#### 2. Issue: Node.js Service Not Accessible Externally

Symptom: You can't access the Node.js app using its external IP.

Cause:

- The Node.js service might not be properly exposing the app or the service type is incorrect (e.g., ClusterIP instead of LoadBalancer).

Solution:

- Ensure that the service type for the Node.js app is `LoadBalancer` (which exposes it externally).

- Verify the external IP with:

kubectl get services

- If using Minikube or a local Kubernetes setup, use:

minikube service nodejs-service

#### 3. Issue: Resource Limitations

Symptom: The application crashes, or one of the pods keeps getting restarted.

Cause:

- The pods may not have enough resources (CPU, memory).

Solution:

- Set resource limits in the pod specifications. For example, you can define resource requests and limits in the container section:

```yaml

resources:

requests:

memory: "256Mi"

cpu: "500m"

limits:

memory: "512Mi"

cpu: "1000m"

This ensures the pod requests the necessary resources and prevents resource contention.

#### 4. Issue: Pods Stuck in Pending State

Symptom: Either the Node.js or MongoDB pod is stuck in `Pending` state.

Cause:

- There might not be enough resources available on the cluster, or the PVC (Persistent Volume Claim) for MongoDB is not configured if storage is required.

Solution:

- Check available resources using:

kubectl describe nodes

- If persistent storage is needed for MongoDB, define a PersistentVolume and PersistentVolumeClaim in the MongoDB deployment file.

#### 5. Issue: Pod Networking

Symptom: Node.js cannot communicate with MongoDB, even though both are deployed.

Cause:

- Network policies, or issues with DNS in the cluster, can prevent communication between pods.

Solution:

- Ensure Kubernetes DNS is functioning correctly, and confirm that both the Node.js app and MongoDB can resolve each other's service names using DNS. You can test DNS resolution by running a shell in a pod:

kubectl exec -it <pod-name> -- nslookup mongodb-service

#### 6. Issue: Logs Not Providing Enough Information

Symptom: Debugging issues is hard because logs from the containers are not helpful.

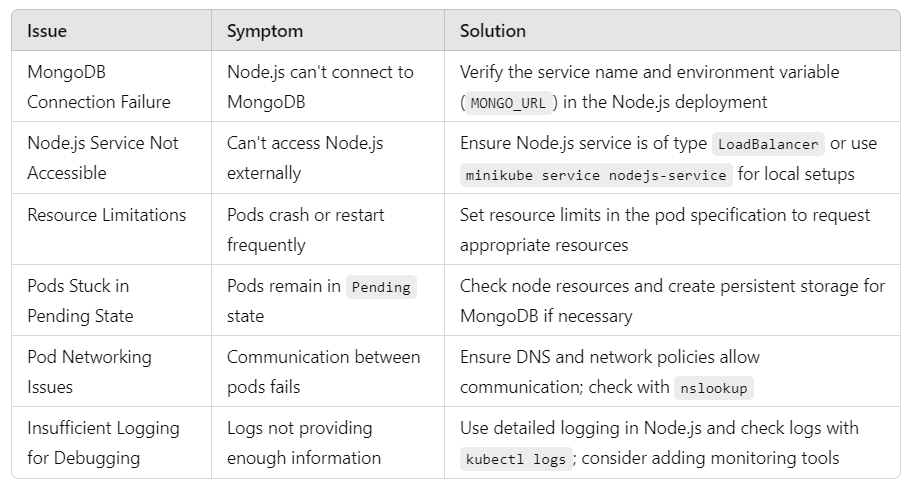
Cause:

- Default logging may not provide enough information.

Solution:

- Use `kubectl logs <pod-name>` to check logs for both MongoDB and Node.js pods. Ensure that both apps have appropriate logging mechanisms (e.g., use libraries like `morgan` or `winston` in Node.js for detailed logs).

- Alternatively, you can use monitoring and logging tools like Prometheus and ELK Stack to capture logs and metrics effectively.

****

In Kubernetes, data management for pods becomes crucial when you need to handle stateful applications or persist data beyond the lifecycle of a pod. By default, Kubernetes pods are ephemeral, meaning when a pod is destroyed, any data generated inside it is also lost. To overcome this, Kubernetes provides a mechanism called Persistent Volumes (PV).

### What is a Persistent Volume (PV)?

A Persistent Volume (PV) in Kubernetes is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using a storage class. Persistent Volumes allow you to store data independently from a pod’s lifecycle, making it available even when a pod is destroyed or restarted.

Persistent Volumes provide persistent storage for applications and decouple the storage from the pod. The Persistent Volume Claim (PVC) is the way applications request the storage they need.

### Key Concepts in Kubernetes Persistent Storage

1. Persistent Volume (PV):

- A PV is a resource in the cluster that represents storage.

- It is provisioned and managed by Kubernetes, but it is independent of the lifecycle of a pod.

- PVs can come from various types of storage backends (local storage, NFS, cloud providers like AWS EBS, GCEPersistentDisk, etc.).

2. Persistent Volume Claim (PVC):

- A PVC is a request for storage by a pod. It specifies the size and access mode of the storage.

- The PVC is bound to a specific Persistent Volume that matches the requested criteria.

3. Storage Class:

- A StorageClass defines the types of storage (e.g., SSD, HDD, provisioner type) that can be dynamically created for a PVC.

- If a PVC does not match an existing PV, Kubernetes can dynamically create a PV using a StorageClass.

4. Volume:

- A Volume is simply storage mounted in a pod. It could be temporary (e.g., `emptyDir`) or persistent (using a PV).

- Volumes can store data that survives container restarts within the same pod.

### Use Case of Persistent Volume

Imagine you are running a MongoDB database in Kubernetes. MongoDB requires persistent storage to store its database files, which need to survive pod restarts or failures.

**### Example: MongoDB Using Persistent Volumes**

**#### 1. Define a Persistent Volume (PV)**

```yaml

apiVersion: v1

kind: PersistentVolume

metadata:

name: mongodb-pv

spec:

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Retain

hostPath:

path: "/data/mongodb"

```

- capacity: The size of the storage.

- accessModes:

- `ReadWriteOnce`: The volume can be mounted as read-write by a single node.

- persistentVolumeReclaimPolicy: Defines what happens to the PV when the PVC is deleted.

- `Retain`: The PV is retained, even if the PVC is deleted.

- hostPath: This refers to a local directory on the host (useful for local development). In production, you might use network or cloud-based storage (e.g., AWS EBS, NFS, etc.).

**#### 2. Define a Persistent Volume Claim (PVC)**

```yaml

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: mongodb-pvc

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 1Gi

- The Persistent Volume Claim requests 1GB of storage with read-write access for MongoDB.

**#### 3. Define the MongoDB Deployment Using the PVC**

```yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: mongodb-deployment

spec:

replicas: 1

selector:

matchLabels:

app: mongodb

template:

metadata:

labels:

app: mongodb

spec:

containers:

- name: mongodb

image: mongo:4.4

ports:

- containerPort: 27017

volumeMounts:

- name: mongodb-storage

mountPath: /data/db

volumes:

- name: mongodb-storage

persistentVolumeClaim:

claimName: mongodb-pvc

- volumeMounts: Mount the volume inside the pod at `/data/db` (which is where MongoDB stores its data).

- volumes: Connect the Persistent Volume Claim (PVC) to the pod.

#### 4. Apply the Files

To deploy the MongoDB application with persistence, apply the following YAML files:

kubectl apply -f mongodb-pv.yaml

kubectl apply -f mongodb-pvc.yaml

kubectl apply -f mongodb-deployment.yaml

### Issues and Solutions in Persistent Volumes

1. Issue: PV Not Binding to PVC

- Symptom: The PVC remains in a `Pending` state, and the pod cannot start.

- Cause: The PV does not match the storage requirements specified in the PVC (e.g., wrong access modes or insufficient storage).

- Solution: Verify the PV’s storage capacity, access modes, and reclaim policies match the PVC’s request.

kubectl get pv

kubectl describe pvc <pvc-name>

2. Issue: Data Loss on Pod Restart

- Symptom: Data is lost when the pod is restarted, despite using persistent volumes.

- Cause: The volume was not properly mounted or defined in the pod's specification.

- Solution: Ensure the `volumeMounts` in the pod specification are correctly defined and the PVC is bound to a valid PV.

3. Issue: Stale PV After PVC Deletion

- Symptom: After deleting the PVC, the associated PV remains in a `Released` state, and cannot be used by other PVCs.

- Cause: The PV is set to Retain instead of Delete.

- Solution: Manually clean up the PV or use the appropriate reclaim policy (`Retain`, `Delete`, or `Recycle`) based on your needs.

kubectl delete pv <pv-name>

4. Issue: Insufficient Storage Space

- Symptom: The application fails to write to the volume due to insufficient storage.

- Cause: The PVC has requested less storage than required, or the storage provider has run out of space.

- Solution: Request more storage in the PVC or ensure the storage backend has enough capacity.

### Summary of Kubernetes Persistent Volumes

- Persistent Volume (PV): A piece of storage in the cluster provided by an admin or dynamically provisioned.

- Persistent Volume Claim (PVC): A request for storage by a pod. The PVC is used to bind the application to the PV.

- Volumes: Storage mounted into the pod, allowing data to persist beyond the pod’s lifecycle.

Persistent Volumes in Kubernetes are essential for handling stateful applications where data needs to be preserved across pod restarts and failures.