K8S volumes

10 July 2025 23:16

📦 Title: EmptyDir Volume in Kubernetes – Practical Documentation

🎯 Objective:

To demonstrate how Kubernetes emptyDir volumes work by sharing data between two containers in the same pod.

📦 What is emptyDir?

- **emptyDir** is a Kubernetes volume type.
- It creates a temporary directory that is **shared among all containers** in a pod.
- The directory is created when the pod starts and **exists only for the lifetime of the pod**.
- Data stored in it is deleted when the pod is removed.

🛠️ Your YAML File: emptydir2.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: m-empty-dir
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: mydata
          mountPath: /usr/share/nginx/html
- name: alpine-writer
      image: alpine
      command: ["/bin/sh", "-c"]
      args:
        - while true; do
            echo "Hello from Alpine at $(date)" > /data/index.html;
            sleep 5;
          done
      volumeMounts:
        - name: mydata
          mountPath: /data
volumes:
    - name: mydata
      emptyDir: {}
```

🔍 Explanation of the YAML

Section	Description
kind: Pod	Creates a single pod with two containers
nginx container	Web server that serves HTML from /usr/share/nginx/html
alpine-writer	A small Alpine container that writes a message to /data/index.html every 5

	seconds
volumeMounts	Both containers mount the same volume (mydata) at different paths
emptyDir volume	Shared temporary storage between containers in this pod

How It Works

1. When the pod starts, Kubernetes creates an **empty directory** in the node's filesystem.
2. This volume is **mounted**:
   - At /usr/share/nginx/html inside the nginx container
   - At /data inside the alpine-writer container
3. The alpine-writer container writes to /data/index.html every 5 seconds.
4. The nginx container automatically serves /usr/share/nginx/html/index.html — which is the same file — to any browser/client.
5. This demonstrates **data sharing between containers** using emptyDir.

Steps to Apply and Verify

1. Apply the Pod

kubectl apply -f emptydir2.yml

2. Check Pod Status

kubectl get pods
You should see:

```
NAME         READY   STATUS    RESTARTS   AGE
m-empty-dir  2/2     Running   0          Xs
```

3. Exec Into Nginx Container

kubectl exec -it m-empty-dir -c nginx -- /bin/sh
Then inside the container:
cat /usr/share/nginx/html/index.html
You should see output like:
Hello from Alpine at Thu Jul 10 18:45:32 UTC 2025
Exit:
exit

4. (Optional) Port-forward and View in Browser

If you want to open in a browser:
kubectl port-forward pod/m-empty-dir 8080:80
Then visit:
http://localhost:8080
The message from the Alpine container will be visible and updating every few seconds.

Key Points About emptyDir

Feature	Detail
Scope	Pod-level only (shared between containers in the same pod)
Lifetime	Lives only as long as the pod lives
Use Cases	Sharing logs, temporary files, cache data between containers
Storage Type	Backed by the node’s disk (RAM if medium: Memory is set)

Bonus: Variants of emptyDir

You can optionally use memory-backed storage:

```
volumes:
  - name: mydata
    emptyDir:
```

medium: Memory
This stores data in RAM — faster but erased on pod restart.

📝 Conclusion

You successfully:

- Created a pod with two containers
- Shared data between them using emptyDir
- Verified the data flow live from one container to another
- Learned how to inspect and test Kubernetes volume usage

📓 Title: Using InitContainer with emptyDir Volume to Clone a Git Repo in Kubernetes

🎯 Objective:

To demonstrate how to use an **initContainer** to clone a GitHub repository into a shared volume (emptyDir), and then serve its contents using **Nginx** in the main container.

🛠️ Your YAML File: init-repo.yml

```
apiVersion: v1
kind: Pod
metadata:
  name: gitpod
spec:
  volumes:
  - name: git-vol
    emptyDir: {}
initContainers:
  - name: git-clone
    image: alpine/git
    command:
      - git
      - clone
      - --branch
      - main
      - https://github.com/rajputganesh217/docker-3-tier.git
      - /git-data
    volumeMounts:
    - name: git-vol
      mountPath: /git-data
containers:
  - name: web
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: git-vol
```

mountPath: /usr/share/nginx/html

🔍 Explanation

Part	Description
initContainers.git-clone	A one-time container that runs **before** the main container, clones a GitHub repo using alpine/git.
volumes.emptyDir	A temporary shared volume (git-vol) created for the pod's lifecycle.
volumeMounts	Both containers mount the same volume, so the cloned repo becomes available to the Nginx container.
nginx container	Serves the contents of the cloned repo through /usr/share/nginx/html.

☑ Step-by-Step Execution

1. Apply the YAML

kubectl apply -f init-repo.yml

2. Check Pod Status

kubectl get pods
Expected output:
NAME READY STATUS RESTARTS AGE
gitpod 1/1 Running 0 Xs
If stuck in Init: state, check logs (explained below).

3. Verify Git Clone

View the output of the Git clone step:
kubectl logs gitpod -c git-clone
Expected output:
Cloning into '/git-data'...
If this works, it means the code was successfully cloned from GitHub into the volume.

4. Check Contents Inside Nginx Container

kubectl exec -it gitpod -- /bin/sh
Then run:
ls /usr/share/nginx/html
cat /usr/share/nginx/html/index.html
You should see the contents of the GitHub repo.

5. (Optional) Expose Pod via Port Forwarding

kubectl port-forward pod/gitpod 8080:80
Open in browser:
http://localhost:8080
You'll see the website loaded from the GitHub content served by Nginx.

📌 Key Concepts & Notes

Concept	Description
initContainer	Runs once before the main container starts. Ideal for setup tasks like downloading code or configs.
emptyDir	Temporary shared storage, exists only for the lifetime of the pod.
alpine/git	Lightweight image with Git installed, used to clone repo.

Why not gitRepo:	gitRepo volume is deprecated and not reliable in modern Kubernetes. This method is more flexible and supported.

Use Cases

- Serve static sites directly from GitHub in dev environments.
- Pull startup config/scripts from a Git repo.
- Separate concerns (Git logic in initContainer, serving logic in app container).

What Happens If You Update the GitHub Repo?

Changes **won't auto-sync**.

To pull the latest code:

kubectl delete pod gitpod

kubectl apply -f init-repo.yml

This triggers the initContainer again, pulling the updated repo.

Summary

- You cloned a GitHub repo into a shared volume.
- Used initContainer to separate Git logic from app logic.
- Verified the result by checking the volume contents and serving it via Nginx.
- Applied a clean and modern approach (preferred over gitRepo volume).

To use a **private GitHub repository** in Kubernetes using an initContainer, you need to **authenticate**. GitHub supports access using:

- Personal Access Token (PAT)
- SSH (not recommended for containers without SSH setup)

We'll use **HTTPS + Personal Access Token**, which is the most practical in Kubernetes.

Step 1: Create a GitHub Personal Access Token (PAT)

1. Go to https://github.com/settings/tokens
2. Click **"Generate new token (classic)"**
3. Give it scopes like:
    - repo (for private repo access)
4. Copy the token and **store it safely**.

Step 2: Create Kubernetes Secret

Assume your repo URL is:

https://github.com/username/private-repo.git

Create a **basic-auth string** like this:

echo -n "git:<YOUR_PAT_TOKEN>" | base64

Then create a secret YAML (git-cred-secret.yml):

```
apiVersion: v1
kind: Secret
metadata:
  name: git-credentials
type: Opaque
data:
  username: Z2l0  # base64 for 'git'
  password: <BASE64_ENCODED_PAT>
```

Apply it:

kubectl apply -f git-cred-secret.yml

🛠️ Step 3: Use It in a Pod

Here's a gitrepo-private.yml example that uses the secret:

```
apiVersion: v1
kind: Pod
metadata:
  name: gitpod-private
spec:
  volumes:
  - name: git-vol
    emptyDir: {}
initContainers:
  - name: git-clone
    image: alpine/git
    env:
    - name: GIT_USERNAME
      valueFrom:
        secretKeyRef:
          name: git-credentials
          key: username
    - name: GIT_PASSWORD
      valueFrom:
        secretKeyRef:
          name: git-credentials
          key: password
    command:
    - /bin/sh
    - -c
    - |
      git clone https://${GIT_USERNAME}:
${GIT_PASSWORD}@github.com/username/private-repo.git /git-data
    volumeMounts:
    - name: git-vol
      mountPath: /git-data
containers:
  - name: web
    image: nginx
    ports:
    - containerPort: 80
    volumeMounts:
    - name: git-vol
      mountPath: /usr/share/nginx/html
```

🔍 Verify

After applying:

```
kubectl apply -f gitrepo-private.yml
kubectl get pods
kubectl logs gitpod-private -c git-clone
kubectl exec -it gitpod-private -- ls /usr/share/nginx/html
```

📄 Title: Using hostPath Volume in Kubernetes with ReplicaSet

🎯 Objective:

To demonstrate how to use a Kubernetes **hostPath** volume to mount a directory from the **worker node's file system** into containers of a ReplicaSet.

📦 What is hostPath?

- hostPath is a **Kubernetes volume type** that mounts a **directory or file from the node's local filesystem** directly into a pod.
- Useful for accessing files that exist on the **Kubernetes worker node**.
- Should be used with caution in production due to:
  - Node dependency (data is tied to a specific node).
  - Potential security implications.

🛠️ Your YAML File: hostpath.yml

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: myrset
spec:
 replicas: 3
 selector:
   matchLabels:
     app: facebook
 template:
   metadata:
     labels:
       app: facebook
   spec:
     containers:
       - name: mycont
         image: nginx
         ports:
           - containerPort: 80
         volumeMounts:
           - name: mydata
             mountPath: /usr/share/nginx/html
     volumes:
       - name: mydata
         hostPath:
           path: /home/ubuntu/myvolume
           type: Directory
```

🔍 Explanation

Component	Description
ReplicaSet	Ensures 3 identical pods are always running
volumeMounts	Mounts the volume into the container at /usr/share/nginx/html

hostPath	Points to the directory /home/ubuntu/myvolume on the worker node
type: Directory	Ensures the given path must exist; otherwise the pod will fail to start

🧱 Node Preparation (Before Applying YAML)

On the **worker node**, create the hostPath directory:
sudo mkdir -p /home/ubuntu/myvolume
echo "Hello from hostPath volume" | sudo tee /home/ubuntu/myvolume/index.html
This file will be served by Nginx running in each pod.

☑ Steps to Apply and Verify

1. Apply the ReplicaSet

kubectl apply -f hostpath.yml

2. Check ReplicaSet and Pods

kubectl get rs
kubectl get pods -l app=facebook
Expected:

```
NAME     READY   STATUS    RESTARTS   AGE
<3 pods>  1/1     Running   0          Xs
```

3. Verify Volume Mount

Pick one pod and exec into it:
kubectl exec -it <pod-name> -- /bin/sh
Then inside:
cat /usr/share/nginx/html/index.html
You should see:
Hello from hostPath volume
This confirms the Nginx container is serving the file from the host node's /home/ubuntu/myvolume.

🌐 Optional: Access in Browser

Create a NodePort service:

service.yml

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

Apply:
kubectl apply -f service.yml
Access via browser:
http://<your-node-ip>:30080
You will see the content of index.html from hostPath volume.

📌 Key Notes About hostPath

Feature	Description
Lifespan	Exists independently of pod — tied to node
Data Sharing	Useful for logs, data import/export, or config sharing from host
Node Dependency	Pod must be scheduled on the node where path exists
type: Directory	Ensures the path must already exist; or use DirectoryOrCreate to create it

⚠ Warnings for Production Use

- hostPath makes your pods dependent on specific nodes.
- Use only for **testing, prototyping**, or **single-node clusters**.
- For durable, multi-node safe volumes use **PersistentVolume + StorageClass** (EBS, NFS, etc.).

Summary

You successfully:

- Created a hostPath-based ReplicaSet
- Mounted a host node directory into all pods
- Verified that Nginx is serving data from the node file system

Title: Mounting and Using NFS (Amazon EFS) Volume in Kubernetes Pod

Objective:

To mount an **Amazon EFS (Elastic File System)** volume inside a Kubernetes pod using the NFS protocol and verify data sharing with an NGINX container.

What is NFS / Amazon EFS?

Feature	Description
NFS	A network file system protocol that allows multiple clients to access shared directories over a network.
Amazon EFS	AWS-managed scalable NFS storage that works with EC2, containers, and Lambda.
Kubernetes Use	Commonly used for **persistent shared storage** between pods or across nodes.

Prerequisites

Before setting up NFS on Kubernetes, you need:

☑ 1. An EFS File System in AWS

- Created in the **same region** and **VPC** as your Kubernetes nodes.
- DNS name: fs-xxxxxxxx.efs.<region>.amazonaws.com

☑ 2. Mount Targets

- One for **each availability zone** where your worker nodes are running.

- Ensure they are **in the same security group** as your worker nodes.

☑ 3. Security Group Rules

In your EFS SG, allow inbound traffic from your worker nodes:

Type	Protocol	Port Range	Source
NFS	TCP	2049	Security group of worker nodes (or 0.0.0.0/0 for test)

🖥 Worker Node Setup

🔧 Install NFS Client (One-time per node)

```
sudo apt update
sudo apt install -y nfs-common
```

📁 Mount Manually to Test

```
sudo mkdir mydata
sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=
600,retrans=2,noresvport \
fs-0afb3d3e269b89e37.efs.ap-south-1.amazonaws.com:/ mydata
cd mydata
sudo mkdir test
cd test
echo "<h1>Hello from EFS</h1>" | sudo tee index.html
```

☑ This confirms that EFS is mounted and writable.

📄 Your Pod Configuration (nfs.yml)

```
apiVersion: v1
kind: Pod
metadata:
 name: nfspod
spec:
 containers:
  - name: mycont1
    image: nginx
    ports:
     - containerPort: 80
    volumeMounts:
     - name: mydata
       mountPath: /usr/share/nginx/html
 volumes:
  - name: mydata
    nfs:
     server: fs-0afb3d3e269b89e37.efs.ap-south-1.amazonaws.com
     path: /test
```

🔍 Explanation

Component	Description
kind: Pod	Creates a single pod running NGINX
volumeMounts	Mounts the EFS /test folder into the container
nfs.server	EFS DNS endpoint
nfs.path	Path inside EFS (/test in this case)

☑ Steps to Apply and Verify

1. Apply the Pod
kubectl apply -f nfs.yml
2. Verify Pod Status
kubectl get pods
Expected:
NAME READY STATUS RESTARTS AGE
nfspod 1/1 Running 0 Xs

3. Check Mounted Data Inside Pod
kubectl exec -it nfspod -- /bin/sh
Inside the container:
cd /usr/share/nginx/html
ls
cat index.html
Expected output:
<h1>Hello from EFS</h1>
You can even write new files if permissions allow.

4. (Optional) Expose via Port Forwarding
kubectl port-forward pod/nfspod 8080:80
Visit:
http://localhost:8080
You will see the index.html page from the EFS volume.

📌 Key Points about NFS / EFS in Kubernetes

Feature	Description
Shared Storage	Multiple pods across nodes can read/write
Persistent	Data remains even after pods are deleted
Scalability	EFS automatically scales with usage
Port Required	**2049/TCP** must be open between node and EFS

⚠️ Tips & Best Practices
- Use a **PersistentVolume (PV)** and **PersistentVolumeClaim (PVC)** in production instead of direct pod-level NFS.
- Monitor **latency-sensitive workloads** — NFS is slower than local storage.
- Be careful with **file permissions**, especially with non-root containers.

📝 Summary
You successfully:
- Created and tested an EFS-backed NFS volume
- Mounted it into a Kubernetes pod
- Verified data sharing and serving with NGINX
- Understood SG, ports, and worker setup required

📓 Detailed Documentation: NFS with Kubernetes PV, PVC, and Pod Mounting
🧱 Objective

To successfully demonstrate end-to-end implementation of shared storage using:

1. Setup of Network File System (NFS) on AWS using Amazon EFS.
2. Mounting the EFS on a Kubernetes worker node.
3. Creating and configuring a Persistent Volume (PV) that connects to the EFS.
4. Creating a Persistent Volume Claim (PVC) to consume the PV.
5. Using the PVC inside a pod to make the storage accessible to applications.
6. Verifying file creation from inside the pod is reflected on the worker node.

Step-by-Step Setup

1. NFS Setup (EFS on AWS)

What:

Amazon Elastic File System (EFS) is a fully-managed NFS file system that can be mounted across multiple AWS resources such as EC2 instances and Kubernetes pods.

Why:

EFS provides a scalable and elastic file system with support for multiple readers and writers. It is perfect for shared storage scenarios in Kubernetes such as shared logs, media content, or application configuration files.

How:

- Navigate to **AWS Console > EFS > Create File System**.
- Choose your **VPC** and enable **automatic mount targets** in each Availability Zone.
- Assign a security group to the EFS that allows NFS access.

Security Group:

- Add an **inbound rule**:
  - Type: NFS
  - Protocol: TCP
  - Port: 2049
  - Source: The security group of your Kubernetes worker nodes or CIDR block.

2. Mount EFS on Worker Node

Create mount directory:

```
sudo mkdir -p /mydata
```

Mount EFS:

```
sudo mount -t nfs4 -o nfsvers=4.1,rsize=1048576,wsize=1048576,hard,timeo=
600,retrans=2,noresvport \
fs-<id>.efs.<region>.amazonaws.com:/ /mydata
```

Verify mount:

```
df -hT | grep mydata
```

Create Subdirectory:

```
sudo mkdir /mydata/test
```

This test folder is referenced by the Kubernetes PV and is where data will be stored.

3. Create Kubernetes Persistent Volume (PV)

Example PV YAML:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv
  labels:
    app: dev
spec:
  capacity:
    storage: 20Gi
```

```
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  nfs:
    path: /test
    server: fs-<id>.efs.<region>.amazonaws.com
```

🛠 **Apply:**

```
kubectl apply -f pv.yml
kubectl get pv
```

Ensure the PV shows STATUS: Available.

📄 4. Create PVC (Persistent Volume Claim)

🔧 **Example PVC YAML:**

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
  labels:
    app: dev
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  selector:
    matchLabels:
      app: dev
```

🛠 **Apply:**

```
kubectl apply -f pvc.yml
kubectl get pvc
```

Ensure PVC status becomes Bound.

🚀 5. Create Pod Using PVC

🔧 **Example Pod YAML:**

```
apiVersion: v1
kind: Pod
metadata:
  name: mypvcpod
spec:
  containers:
    - name: mycont
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: myvolume
          mountPath: "/mnt/data"
  volumes:
    - name: myvolume
      persistentVolumeClaim:
        claimName: mypvc
```

🛠 **Apply and Test:**

```
kubectl apply -f pvcpod.yml
kubectl get pods
```

kubectl exec -it mypvcpod -- bash
cd /mnt/data
echo "Hello from pod" > hello.txt
echo "This is second file" > fast.txt

📝 6. Verify From Worker Node

❗ **NFS Directory Caching Behavior:**

If you're inside /mydata/test and don't see files created in the pod:
cd
ls /mydata/test # This shows the correct file list
But:
cd /mydata/test
ls # May show nothing
This is due to **NFS client directory caching**. Always refresh directory view or re-enter the directory to resolve it.

☑️ **Expected Output:**

ubuntu@ip-172-31-36-111:~$ ls /mydata/test
fast.txt hello.txt

🚨 Common Issues & Troubleshooting

❌ df -hT | grep mydata shows nothing

- **Cause**: Mount point does not exist or EFS not mounted.
- **Fix**: Create the mount directory and re-run the mount command.

❌ ls inside /mydata/test shows nothing but ls /mydata/test works

- **Cause**: NFS client directory cache not refreshed
- **Fix**:
  - Exit and re-enter the directory:
    cd /
    cd /mydata/test
    ls
  - Use ls -la to force NFS to refresh directory listing.

❌ Pod volume not mounted

- **Fix**:
  - Use kubectl describe pod <name>
  - Check Events for mount errors such as permission denied or no such file or directory

❌ Files not syncing between pod and worker

- **Fix**:
  - Ensure the EFS server address in PV matches the one mounted manually
  - Validate EFS Security Group rules (NFS access allowed)

📋 Helpful Commands

📌 PV, PVC, and Pods:

kubectl get pv
kubectl get pvc
kubectl get pods

🧹 Delete Resources:

kubectl delete pod mypvcpod
kubectl delete pvc mypvc

kubectl delete pv mypv

🔍 Debugging:

kubectl describe pod mypvcpod
kubectl exec -it mypvcpod -- bash

📂 Check EFS Mount on Node:

df -hT | grep mydata
ls /mydata/test

📌 Notes

- EFS is ideal for workloads needing ReadWriteMany (RWX) access.
- Directory cache issues are common in NFS; re-navigating helps.
- Always use the correct EFS endpoint in PV configuration.
- Ensure firewall/security groups allow NFS traffic (port 2049).

⌨ Conclusion

You have successfully:

- Provisioned and mounted an NFS file system (EFS)
- Set up a Persistent Volume (PV) and Persistent Volume Claim (PVC)
- Used the PVC in a Kubernetes pod
- Verified persistent and shared data across environments

This setup ensures scalable, reusable, and shared storage in Kubernetes environments.
Ideal for apps storing logs, media, configs, or shared cache.
Let me know if you'd like a PDF version or a visual architecture diagram!

Kubernetes Persistent Volume Setup Using hostPath

🧮 Objective

To implement and verify a hostPath-based Persistent Volume (PV), Persistent Volume Claim (PVC), and mount it into an NGINX pod on Kubernetes for shared storage between the host and the container.

🔧 Use Case

Use hostPath to mount a local directory on a Kubernetes **worker node** into a pod. This is useful for:

- Testing local persistent storage
- Debugging and educational purposes
- Single-node clusters
  Note: hostPath volumes are not suitable for production or multi-node clusters.

🔑 Prerequisites

On Worker Node:

1. Ensure Kubernetes is running and node is Ready.
2. Create a host path directory:

sudo mkdir -p /home/ubuntu/myhost
sudo chmod -R 777 /home/ubuntu/myhost

1. Add a test file:

echo "Hello from Worker Node" | sudo tee /home/ubuntu/myhost/about.html

Step-by-Step Implementation

Step 1: Define Persistent Volume (PV)

File: pvhost.yml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mypv1
  labels:
    app: dev
spec:
  capacity:
    storage: 256Mi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  hostPath:
    path: /home/ubuntu/myhost
    type: Directory
```

Commands:

```
kubectl apply -f pvhost.yml
kubectl get pv
```

Expected Output:

```
NAME    CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS     AGE
mypv1   256Mi     RWO           Recycle         Available  ...
```

Step 2: Define Persistent Volume Claim (PVC)

File: pvchost.yml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc1
  labels:
    app: dev
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 256Mi
  selector:
    matchLabels:
      app: dev
```

Commands:

```
kubectl apply -f pvchost.yml
kubectl get pvc
```

Expected Output:

```
NAME    STATUS   VOLUME   CAPACITY   ACCESS MODES
mypvc1  Bound    mypv1    256Mi      RWO
```

Step 3: Create Pod Using PVC

File: pvchostpod.yml

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: mypvchostpod
spec:
  containers:
    - name: mycont
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: myvolume
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: myvolume
      persistentVolumeClaim:
        claimName: mypvc1
```

Commands:

```
kubectl apply -f pvchostpod.yml
kubectl get pods
```

Expected Output:

```
NAME           READY   STATUS    RESTARTS   AGE
mypvchostpod   1/1     Running   0          ...
```

Verification

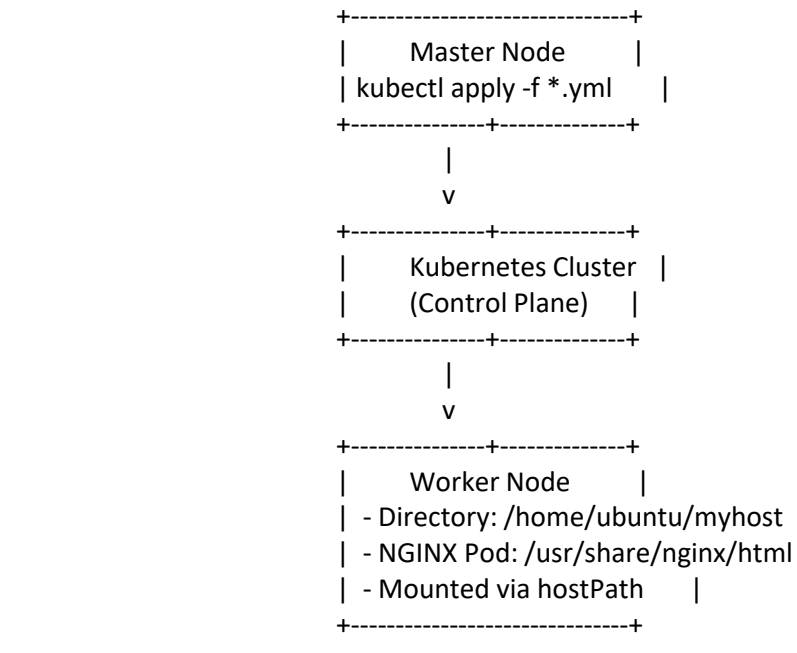
Inside Pod:

```
kubectl exec -it mypvchostpod -- bash
cd /usr/share/nginx/html
ls
cat about.html
```

Expected Output:

```
about.html
Hello from Worker Node
```

Architecture Diagram



Conclusion

You have successfully:

- Created a hostPath-based PV and PVC
- Mounted it into a running NGINX pod
- Verified file sharing between host and container
  This setup is ideal for quick local testing or development scenarios.