This shows how to configure a Pod to use a PersistentVolumeClaim for storage. Here is a summary of the process:

1. A cluster administrator creates a PersistentVolume that is backed by physical storage. The administrator does not associate the volume with any Pod.
2. A cluster user creates a PersistentVolumeClaim, which gets automatically bound to a suitable PersistentVolume.
3. The user creates a Pod that uses the PersistentVolumeClaim as storage.

* You need to have a Kubernetes cluster that has only one Node, and the kubectl command-line tool must be configured to communicate with your cluster. If you do not already have a single-node cluster, you can create one by using [Minikube](https://github.com/rootsongjc/kubernetes.github.io/blob/master/docs/getting-started-guides/minikube).
* Familiarize yourself with the material in [Persistent Volumes](https://github.com/rootsongjc/kubernetes.github.io/blob/master/docs/concepts/storage/persistent-volumes).

**Create an index.html file on your Node**

Open a shell to the Node in your cluster. How you open a shell depends on how you set up your cluster. For example, if you are using Minikube, you can open a shell to your Node by entering minikube ssh.

In your shell, create a /tmp/data directory:

mkdir /tmp/data

In the /tmp/data directory, create an index.html file:

echo 'Hello from Kubernetes storage' > /tmp/data/index.html

**Create a PersistentVolume**

In this exercise, you create a *hostPath* PersistentVolume. Kubernetes supports hostPath for development and testing on a single-node cluster. A hostPath PersistentVolume uses a file or directory on the Node to emulate network-attached storage.

In a production cluster, you would not use hostPath. Instead a cluster administrator would provision a network resource like a Google Compute Engine persistent disk, an NFS share, or an Amazon Elastic Block Store volume. Cluster administrators can also use [StorageClasses](https://github.com/rootsongjc/kubernetes.github.io/blob/master/docs/resources-reference/%7B%7Bpage.version%7D%7D/" \l "storageclass-v1-storage) to set up [dynamic provisioning](http://blog.kubernetes.io/2016/10/dynamic-provisioning-and-storage-in-kubernetes.html).

Here is the configuration file for the hostPath PersistentVolume:

The configuration file specifies that the volume is at /tmp/data on the the cluster's Node. The configuration also specifies a size of 10 gibibytes and an access mode of ReadWriteOnce, which means the volume can be mounted as read-write by a single Node. It defines the [StorageClass name](https://github.com/rootsongjc/kubernetes.github.io/blob/master/docs/concepts/storage/persistent-volumes/" \l "class) manual for the PersistentVolume, which will be used to bind PersistentVolumeClaim requests to this PersistentVolume.

Create the PersistentVolume:

kubectl create -f https://raw.githubusercontent.com/sureshsubbaiah/kubernetes-examples/master/volumes/persistentVolume/task-pv-volume.yaml

View information about the PersistentVolume:

kubectl get pv task-pv-volume

The output shows that the PersistentVolume has a STATUS of Available. This means it has not yet been bound to a PersistentVolumeClaim.

NAME CAPACITY ACCESSMODES RECLAIMPOLICY STATUS CLAIM STORAGECLASS REASON AGE

task-pv-volume 10Gi RWO Retain Available manual 4s

**Create a PersistentVolumeClaim**

The next step is to create a PersistentVolumeClaim. Pods use PersistentVolumeClaims to request physical storage. In this exercise, you create a PersistentVolumeClaim that requests a volume of at least three gibibytes that can provide read-write access for at least one Node.

Here is the configuration file for the PersistentVolumeClaim:

Create the PersistentVolumeClaim:

kubectl create -f https://raw.githubusercontent.com/sureshsubbaiah/kubernetes-examples/master/volumes/persistentVolume/task-pv-claim.yaml

After you create the PersistentVolumeClaim, the Kubernetes control plane looks for a PersistentVolume that satisfies the claim's requirements. If the control plane finds a suitable PersistentVolume with the same StorageClass, it binds the claim to the volume.

Look again at the PersistentVolume:

kubectl get pv task-pv-volume

Now the output shows a STATUS of Bound.

NAME CAPACITY ACCESSMODES RECLAIMPOLICY STATUS CLAIM STORAGECLASS REASON AGE

task-pv-volume 10Gi RWO Retain Bound default/task-pv-claim manual 2m

Look at the PersistentVolumeClaim:

kubectl get pvc task-pv-claim

The output shows that the PersistentVolumeClaim is bound to your PersistentVolume, task-pv-volume.

NAME STATUS VOLUME CAPACITY ACCESSMODES STORAGECLASS AGE

task-pv-claim Bound task-pv-volume 10Gi RWO manual 30s

**Create a Pod**

The next step is to create a Pod that uses your PersistentVolumeClaim as a volume.

Here is the configuration file for the Pod:

Notice that the Pod's configuration file specifies a PersistentVolumeClaim, but it does not specify a PersistentVolume. From the Pod's point of view, the claim is a volume.

Create the Pod:

kubectl create -f https://raw.githubusercontent.com/sureshsubbaiah/kubernetes-examples/master/volumes/persistentVolume/task-pv-pod.yaml

Verify that the Container in the Pod is running;

kubectl get pod task-pv-pod

Get a shell to the Container running in your Pod:

kubectl exec -it task-pv-pod -- /bin/bash

In your shell, verify that nginx is serving the index.html file from the hostPath volume:

root@task-pv-pod:/# apt-get update

root@task-pv-pod:/# apt-get install curl

root@task-pv-pod:/# curl localhost

The output shows the text that you wrote to the index.html file on the hostPath volume:

Hello from Kubernetes storage

**Access control**

Storage configured with a group ID (GID) allows writing only by Pods using the same GID. Mismatched or missing GIDs cause permission denied errors. To reduce the need for coordination with users, an administrator can annotate a PersistentVolume with a GID. Then the GID is automatically added to any Pod that uses the PersistentVolume.

Use the pv.beta.kubernetes.io/gid annotation as follows:

kind: PersistentVolume

apiVersion: v1

metadata:

name: pv1

annotations:

pv.beta.kubernetes.io/gid: "1234"

When a Pod consumes a PersistentVolume that has a GID annotation, the annotated GID is applied to all Containers in the Pod in the same way that GIDs specified in the Pod’s security context are. Every GID, whether it originates from a PersistentVolume annotation or the Pod’s specification, is applied to the first process run in each Container.

**Note**: When a Pod consumes a PersistentVolume, the GIDs associated with the PersistentVolume are not present on the Pod resource itself. {: .note}