Storing and Reading Data on Disk

Overview

This chapter introduces the concept of using Volumes to store or read data from the containers running inside pods. By the end of this chapter, you will be able to create Volumes to temporarily store data in a pod independent of a container's life cycle, as well as share the data among different containers inside the same pod. You will also learn how to use **PersistentVolumes** (**PVs**) to store data on your cluster independent of the pod life cycle. We will also cover how to create **PersistentVolumeClaims** (**PVCs**) to dynamically provision volumes and use them inside a pod.

Introduction

In previous chapters, we created Deployments to create multiple replicas of our application and exposed our application using Services. However, we have not yet properly explored how Kubernetes facilitates applications to store and read data, which is the subject of this chapter.

In practice, most applications interact with data in some way. It's possible that we may have an application that needs to read data from a file. Similarly, our application may need to write some data locally in order for other parts of the application, or different applications, to read it. For example, if we have a container running our main application that produces some logs locally, we would want to have a sidecar container (which is a second container running inside the pod along with the main application container) that can run inside the same pod to read and process the local logs produced by the main application. However, to enable this, we need to find a way to share the storage among different containers in the same pod.

Let's say we are training a machine learning model in a pod. During the intermediate stages of the model training, we would need to store some data locally on a disk. Similarly, the end result -- the trained model -- will need to be stored on a disk, such that it can be retrieved later even once the pod terminates. For this use case, we need some way of allocating some storage to the pod such that the data written in that storage exists even beyond the life cycle of the pod.

Similarly, we may have some data that needs to be written or read by multiple replicas of the same application. This data should also persist when some of such pod replicas crash and/or restart. For example, if we have an e-commerce website, we may want to store the user data, as well as inventory records, in a database. This data will need to be persisted across pod restarts as well as Deployment updates or rollbacks.

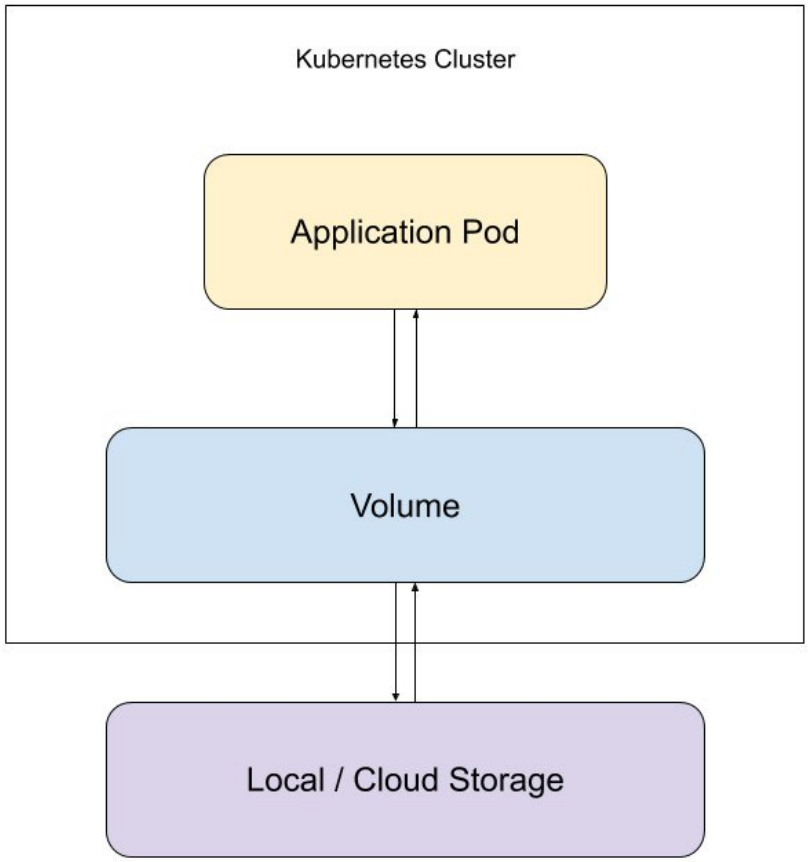
To serve these purposes, Kubernetes provides an abstraction called Volume. A **PersistentVolume** (**PV**) is the most common type of Volume that you will encounter. In this chapter, we will cover this, as well as many other types of Volumes. We will learn how to use them and provision them on-demand.

Volumes

Let's say we have a pod that stores some data locally on a disk. Now, if the container that's storing the data crashes and is restarted, the data will be lost. The new container will start with an empty disk space allocated. Thus, we cannot rely on containers themselves even for the temporary storage of data.

We may also have a case where one container in a pod stores some data that needs to be accessed by other containers in the same pod as well.

The Kubernetes Volume abstraction solves both of these problems. Here's a diagram showing Volumes and their interaction with physical storage and the application:



As you can see from this diagram, a Volume is exposed to the applications as an abstraction, which eventually stores the data on any type of physical storage that you may be using.

The lifetime of a Kubernetes Volume is the same as that of the pod that uses it. In other words, even if the containers within a pod restart, the same Volume will be used by the new container as well. Hence, the data isn't lost across container restarts. However, once a pod terminates or is restarted, the Volume ceases to exist, and the data is lost. To solve this problem, we can use PVs, which we will cover later in this chapter.

How to Use Volumes

A Volume is defined in the pod spec. Here's an example of a pod configuration with Volumes:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-emptydir-volume
spec:
  restartPolicy: Never
  containers:
  - image: ubuntu
    name: ubuntu-container
    volumeMounts:
    - mountPath: /data
      name: data-volume
```

```
  volumes:
  - name: data-volume
    emptyDir: {}
```

As we can see in the preceding configuration, to define a Volume, a pod configuration needs to set two fields:

- The `.spec.volumes` field defines what Volumes this pod is planning to use.
- The `.spec.containers.volumeMounts` defines where to mount those Volumes in individual containers. This will be defined separately for all the containers.

Defining Volumes

In the preceding example, the `.spec.volumes` field has two fields that define the configuration of a Volume:

- `name` : This is the name of the Volume by which it will be referred to in the containers' `volumeMounts` fields when it will be mounted. It has to be a valid DNS name. The name of the Volume must be unique within a single pod.
- `emptyDir` : This varies based on the type of the Volume being used (which, in the case of the preceding example, is `emptyDir`). This defines the actual configuration of the Volume. We will go through the types of Volumes in the next section with some examples.

Mounting Volumes

Each container needs to specify `volumeMounts` separately to mount the volume. In the preceding example, you can see that the `.spec.containers[*].volumeMounts` configuration has the following fields:

- `name` : This is the name of the Volume that needs to be mounted for this container.
- `mountPath` : This is the path inside the container where the Volume should be mounted. Each container can mount the same Volume on different paths.

Other than these, there are two other notable fields that we can set:

- `subPath` : This is an optional field that contains the path from the Volume that needs to be mounted on the container. By default, the volume is mounted from its root directory. This field can be used to mount only a sub-directory in the volume and not the entire volume. For example, if you're using the same Volume for multiple users, it's useful to mount a sub-path on the containers, rather than the root directory of the Volume.
- `readonly` : This is an optional flag that determines whether the mounted volume will be read-only or not. By default, the volumes are mounted with read-write access.

Types of Volumes

As mentioned earlier, Kubernetes supports several types of Volumes and the availability of most of them depends on the cloud provider that you use. AWS, Azure, and Google Cloud all have different types of Volumes supported.

Let's take a look at some common types of Volumes in detail.

emptyDir

An `emptyDir` Volume refers to an empty directory that's created when a pod is assigned to a node. It only exists as long as the pod does. All the containers running inside the pod have the ability to write and read files from this directory. The same `emptyDir` Volume can be mounted on different paths for different containers.

Here's an example of pod configuration using the `emptyDir` Volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-emptydir-volume
spec:
  restartPolicy: Never
  containers:
  - image: ubuntu
    name: ubuntu-container
    volumeMounts:
    - mountPath: /data
      name: data-volume
  volumes:
  - name: data-volume
    emptyDir: {}
```

In this example, `{}` indicates that the `emptyDir` Volume will be defined in the default manner. By default, the `emptyDir` Volumes are stored on the disk or SSD, depending on the environment. We can change it to use RAM instead by setting the `.emptyDir.medium` field to `Memory`.

Thus, we can modify the `volumes` section of the preceding pod configuration to use the `emptyDir` Volume backed by memory, as follows:

```
  volumes:
  - name: data-volume
    emptyDir:
      medium: Memory
```

This informs Kubernetes to use a RAM-based filesystem (tmpfs) to store the Volume. Even though tmpfs is very fast compared to data on a disk, there are a couple of downsides to using in-memory Volume. First, the tmpfs storage is cleared on the system reboot of the node on which the pod is running. Second, the data stored in a memory-based Volume counts against the memory limits of the container. Hence, we need to be careful while using memory-based Volumes.

We can also specify the size limit of the storage to be used in the `emptyDir` Volume by setting the `.volumes.emptyDir.sizeLimit` field. This size limit applies to both disk-based and memory-based `emptyDir` Volumes. In the case of memory-based Volumes, the maximum usage allowed will be either the `sizeLimit` field value or the sum of memory limits on all containers in the pod -- whichever is lower.

Use Cases

Some of the use cases for `emptyDir` Volumes are as follows:

- Temporary scratch space for computations requiring a lot of space, such as on-disk merge sort
- Storage required for storing checkpoints for a long computation, such as training machine learning models where the progress needs to be saved to recover from crashes

hostPath

A `hostPath` Volume is used to mount a file or a directory from the host node's filesystem to a pod.

Here's an example of pod configuration using the `hostPath` Volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-hostpath-volume
spec:
  restartPolicy: Never
  containers:
  - image: ubuntu
    name: ubuntu-container
    volumeMounts:
    - mountPath: /data
      name: data-volume
  volumes:
  - name: data-volume
    hostPath:
      path: /tmp
      type: Directory
```

In this example, the `/home/user/data` directory from the host node will be mounted on the `/data` path on the container. Let's look at the two fields under `hostPath`:

- `path`: This is the path of the directory or the file that will be mounted on the containers mounting this Volume. It can also be a symlink (symbolic link) to a directory or a file, the address of a UNIX socket, or a character or block device, depending on the `type` field.
- `type`: This is an optional field that allows us to specify the type of the Volume. If this field is specified, certain checks will be performed before mounting the `hostPath` Volume.

The `type` field supports the following values:

- `""` (an empty string): This is the default value implying that no checks will be performed before mounting the `hostPath` Volume. If the path specified doesn't exist on the node, the pod will still be created without verifying the existence of the path. Hence, the pod will keep crashing indefinitely because of this error.
- `DirectoryOrCreate`: This implies that the directory path specified may or may not already exist on the host node. If it doesn't exist, an empty directory is created.
- `Directory`: This implies that a directory must exist on the host node at the path specified. If the directory doesn't exist at the path specified, there will be a `FailedMount` error while creating the pod, indicating that the `hostPath` type check has failed.
- `FileOrCreate`: This implies that the file path specified may or may not already exist on the host node. If it doesn't exist, an empty file is created.
- `File`: This implies that a file must exist on the host node at the path specified.
- `Socket`: This implies that a UNIX socket must exist at the path specified.
- `CharDevice`: This implies that a character device must exist at the path specified.
- `BlockDevice`: This implies that a block device must exist at the path specified.

Use Cases

In most cases, your application won't need a `hostPath` Volume. However, there are some niche use cases where the `hostPath` Volume may be particularly useful. Some of these use cases for the `hostPath` Volume are as follows:

- Allowing pods to be created only if a particular host path exists on the host node before running the pod. For example, a pod may require some Secrets or credentials to be present in a file on the host before it can

run.

- Running a container that needs access to Docker internals. We can do that by setting `hostPath` to `/var/lib/docker`.

  Note

  In addition to the two types of Volumes covered here, Kubernetes supports many more, some of which are specific to certain cloud platforms. You can find more information about them at https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes.

In the previous sections, we learned about Volumes and how to use their different types. In the following exercises, we will put these concepts into action and use Volumes with pods.

Exercise 9.01: Creating a Pod with an emptyDir Volume

In this exercise, we will create a basic pod with an `emptyDir` Volume. We will also simulate data being written manually, and then make sure that the data stored in the Volume is kept across container restarts:

1. Create a file called `pod-with-emptydir-volume.yaml` with the following content:

   ```
   apiVersion: v1
   kind: Pod
   metadata:
     name: pod-with-emptydir-volume
   spec:
     containers:
     - image: nginx
       name: nginx-container
       volumeMounts:
       - mountPath: /mounted-data
         name: data-volume
     volumes:
     - name: data-volume
       emptyDir: {}
   ```

   In this pod configuration, we have used an `emptyDir` Volume mounted at the `/mounted-data` directory.

2. Run the following command to create the pod using the preceding configuration:

   ```
   kubectl create -f pod-with-emptydir-volume.yaml
   ```

   You should see the following response:

   ```
   pod/pod-with-emptydir-volume created
   ```

3. Run the following command to confirm that the pod was created and is ready:

   ```
   kubectl get pod pod-with-emptydir-volume
   ```

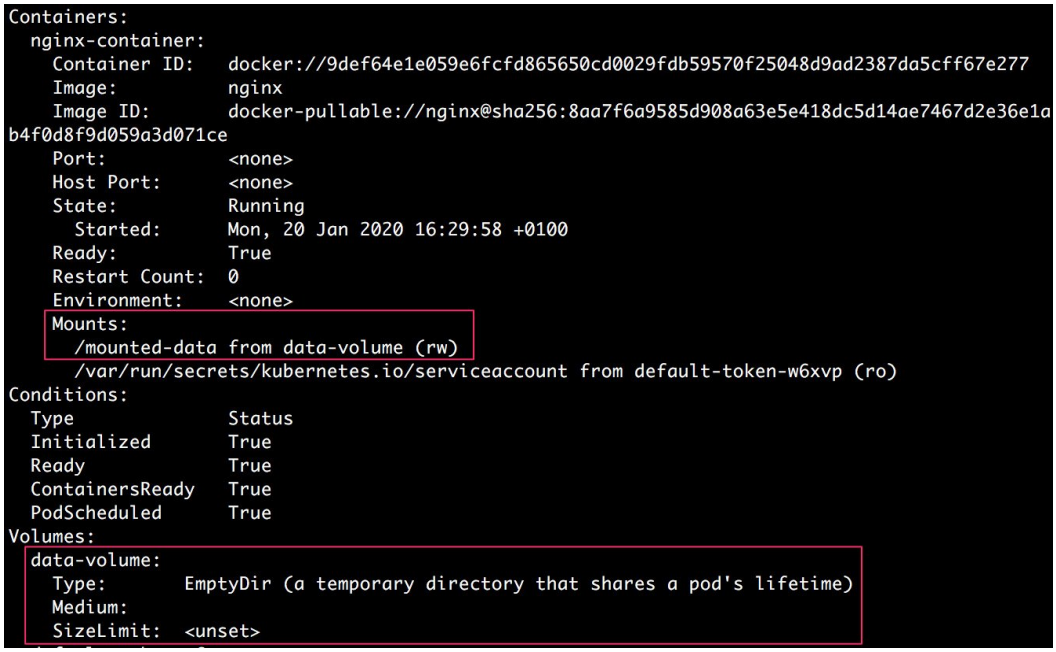
   You should see the following response:

```
NAME                        READY   STATUS    RESTARTS   AGE
pod-with-emptydir-volume    1/1     Running   0          20s
```

4. Run the following command to describe the pod so that we can verify that the correct Volume was mounted on this pod:

```
kubectl describe pod pod-with-emptydir-volume
```

This will give a long output. Look for the following section in the terminal output:



As highlighted in the preceding image, the `emptyDir` Volume named `data-volume` was created and it was mounted on `nginx-container` at the `/mounted-data` path. We can see that the Volume has been mounted in `rw` mode, which stands for read-write.

Now that we have verified that the pod was created with the correct Volume configured, we will manually write some data to this path. In practice, this writing will be done by your application code.

5. Now, we will use the `kubectl exec` command to run the Bash shell inside the pod:

```
kubectl exec pod-with-emptydir-volume -it /bin/bash
```

You should see the following on your terminal screen:

```
root@pod-with-emptydir-volume:/#
```

This will now allow you to run commands via an SSH connection on the Bash shell running in the `nginx-container`. Note that we are running as a root user.

Note

If you had a sidecar container running in the pod (or any number of multiple containers in a pod), then you can control where the `kubectl exec` command will execute by adding the `-c` parameter to specify the container, as you will see in the next exercise.

6. Run the following command to check the content of the root directory of the pod:

```
ls
```

   You should see an output similar to this one:

```
bin   dev  home  lib64  mnt          opt   root  sbin   sys   usr
boot  etc  lib   media  mounted-data proc  run   srv    tmp   var
```

   Notice that there's a directory called `mounted-data`.

7. Run the following commands to go to the `mounted-data` directory and check its content:

```
cd mounted-data
ls
```

   You should see a blank output, as follows:

```
root@pod-with-emptydir-volume:/mounted-data#
```

   This output indicates that the `mounted-data` directory is empty as expected because we don't have any code running inside the pod that would write to this path.

8. Run the following command to create a simple text file inside the `mounted-data` directory:

```
echo "Manually stored data" > manual-data.txt
```

9. Now, run the `ls` command again to check the content of the directory:

```
ls
```

   You should see the following output:

```
manual-data.txt
```

   Thus, we have created a new file with some content in the mounted volume directory. Now, our aim will be to verify that this data will still exist if the container is restarted.

10. In order to restart the container, we will kill the `nginx` process, which will trigger a restart. Run the following commands to install the procps package so that we can use the `ps` command to find out the process ID (PID) of the process that we want to kill. First, update the package lists:

```
sudo apt-get update
```

    You should see an output similar to the following:

```
Get:2 http://deb.debian.org/debian buster InRelease [122 kB]
Get:3 http://deb.debian.org/debian buster-updates InRelease [49.3 kB]
Get:1 http://security-cdn.debian.org/debian-security buster/updates InRelease [65.4 kB]
Get:4 http://deb.debian.org/debian buster/main amd64 Packages [7908 kB]
Get:5 http://security-cdn.debian.org/debian-security buster/updates/main amd64 Packages [171 kB]
Get:6 http://deb.debian.org/debian buster-updates/main amd64 Packages [5792 B]
Fetched 8321 kB in 4s (2319 kB/s)
Reading package lists... Done
```

```
Our package lists are up to date and we are now ready to install
procps.
```

11. Use the following command to install procps:

    ```
    sudo apt-get install procps
    ```

    Enter *Y* when prompted to confirm the installation, and then the installation will proceed with an output similar to the following:

    ```
    Reading package lists... Done
    Building dependency tree
    Reading state information... Done
    The following additional packages will be installed:
      libgpm2 libncurses6 libprocps7 psmisc
    Suggested packages:
      gpm
    The following NEW packages will be installed:
      libgpm2 libncurses6 libprocps7 procps psmisc
    0 upgraded, 5 newly installed, 0 to remove and 0 not upgraded.
    Need to get 584 kB of archives.
    After this operation, 1931 kB of additional disk space will be used.
    Do you want to continue? [Y/n]
    Get:1 http://deb.debian.org/debian buster/main amd64 libncurses6 amd64 6.1+20181013-2+deb10u2 [10
    2 kB]
    ```

12. Now, run the following command to check the list of processes running on the container:

    ```
    ps aux
    ```

    You should see the following output:

    ```
    USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
    root         1  0.0  0.2  10632  5020 ?        Ss   15:45   0:00 nginx: master process nginx -g d
    nginx        6  0.0  0.1  11088  2560 ?        S    15:45   0:00 nginx: worker process
    root         7  0.0  0.1   3988  3216 pts/0    Ss   15:46   0:00 /bin/bash
    root       338  0.0  0.1   7640  2684 pts/0    R+   15:55   0:00 ps aux
    ```

```
In the output, we can see that among several other processes, the
`nginx` master process is running with a `PID`
of `1`.
```

13. Run the following command to kill the `nginx` master process:

    ```
    kill 1
    ```

    You should see the following response:

    ```
    root@pod-with-emptydir-volume:/mounted-data# command terminated with exit code 137
    ```

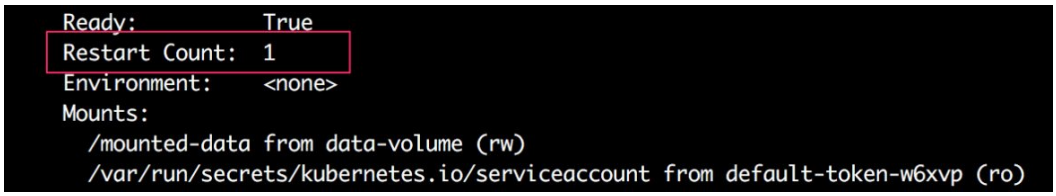
```
The output shows that the terminal exited the Bash session on the
pod. This is because the container was killed. The `137`
```

```
exit code indicates that the session was killed by manual
intervention.
```

14. Run the following command to get the status of the pod:

    ```
    kubectl describe pod pod-with-emptydir-volume
    ```

    Observe the following section in the output that you get:


```
You will see that there\'s now a `Restart Count` field for
`nginx-container` that has a value of `1`. That
means that the container was restarted after we killed it. Please
note that restarting a container doesn\'t trigger a restart of a
pod. Hence, we should expect the data stored in the Volume to still
exist. Let\'s verify that in the next step.
```

15. Let's run Bash inside the pod again and go to the `/mounted-data` directory:

    ```
    kubectl exec pod-with-emptydir-volume -it /bin/bash
    cd mounted-data
    ```

    You will see the following output:

    ```
    root@pod-with-emptydir-volume:/# cd mounted data/
    ```

16. Run the following command to check the contents of `/mounted-data` directory:

    ```
    ls
    ```

    You will see the following output:

    ```
    manual-data.txt
    ```

    This output indicates that the file we created before killing the container still exists in the Volume.

17. Run the following command to verify the contents of the file we created in the Volume:

    ```
    cat manual-data.txt
    ```

    You will see the following output:

    ```
    Manually stored data
    ```

    This output indicates that the data we stored in the Volume stays intact even when the container gets restarted.

18. Run the following command to delete the pod:

```
kubectl delete pod pod-with-emptydir-volume
```

You will see the following output confirming that the pod has been deleted:

```
pod "pod-with-emptydir-volume" deleted
```

In this exercise, we created a pod with the `emptyDir` Volume, checked that the pod was created with an empty directory mounted at the correct path inside the container, and verified that we can write the data inside that directory and that the data stays intact across the container restarts as long as the pod is still running.

Now, let's move to a scenario that lets us observe some more uses for Volumes. Let's consider a scenario where we have an application pod that runs a total of three containers. We can assume that two of the three containers are serving traffic and they dump the logs into a shared file. The third container acts as a sidecar monitoring container that reads the logs from the file and dumps them into an external log storage system where the logs can be preserved for further analysis and alerting. Let's consider this scenario in the next exercise and understand how we can utilize an `emptyDir` Volume shared between the three containers of a pod.

Exercise 9.02: Creating a Pod with an emptyDir Volume Shared by Three Containers

In this exercise, we will show some more uses of the `emptyDir` Volume and share it among three containers in the same pod. Each container will mount the same volume at a different local path:

1. Create a file called `shared-emptydir-volume.yaml` with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: shared-emptydir-volume
spec:
  containers:
  - image: ubuntu
    name: container-1
    command: ['/bin/bash', '-ec', 'sleep 3600']
    volumeMounts:
    - mountPath: /mounted-data-1
      name: data-volume
  - image: ubuntu
    name: container-2
    command: ['/bin/bash', '-ec', 'sleep 3600']
    volumeMounts:
    - mountPath: /mounted-data-2
      name: data-volume
  - image: ubuntu
    name: container-3
    command: ['/bin/bash', '-ec', 'sleep 3600']
    volumeMounts:
    - mountPath: /mounted-data-3
      name: data-volume
  volumes:
```

```
  - name: data-volume
    emptyDir: {}
```

In this configuration, we have defined an `emptyDir` Volume named `data-volume`, which is being mounted on three containers at different paths.

Note that each of the containers has been configured to run a command on startup that makes them sleep for 1 hour. This is intended to keep the `ubuntu` container running so that we can perform the following operations on the containers. By default, an `ubuntu` container is configured to run whatever command is specified and exit upon completion.

2. Run the following command to create the pod with the preceding configuration:

```
kubectl create -f shared-emptydir-volume.yaml
```

You will see the following output:

```
pod/shared-emptydir-volume created
```

3. Run the following command to check the status of the pod:

```
kubectl get pod shared-emptydir-volume
```

You will see the following output:

```
NAME                     READY   STATUS    RESTARTS   AGE
shared-emptydir-volume   3/3     Running   0          13s
```

This output indicates that all three containers inside this pod are running.

4. Next, we will run the following command to run Bash in the first container:

```
kubectl exec shared-emptydir-volume -c container-1 -it -- /bin/bash
```

Here, the `-c` flag is used to specify the container that we want to run Bash in. You will see the following in the terminal:

```
root@shared-emptydir-volume:/#
```

5. Run the following command to check the content of the root directory on the container:

```
ls
```

You will see the following output:

```
bin   dev  home  lib32  libx32  mnt            opt   root  sbin  sys  usr
boot  etc  lib   lib64  media   mounted-data-1  proc  run   srv   tmp  var
```

```
We can see that the `mounted-data-1` directory has been
created on the container. Also, you can see the list of directories
you would see in a typical Ubuntu root directory, in addition to the
`mounted-data-1` directory that we created.
```

6. Now, we will go to the `mounted-data-1` directory and create a simple text file with some text in it:

```
cd mounted-data-1
echo 'Data written on container-1' > data-1.txt
```

7. Run the following command to verify that the file has been stored:

```
ls
```

You will see the following output:

```
data-1.txt
```

8. Run the following command to exit `container-1` and go back to your host terminal:

```
exit
```

9. Now, let's run Bash inside the second container, which is named `container-2`:

```
kubectl exec shared-emptydir-volume -c container-2 -it -- /bin/bash
```

You will see the following in your terminal:

```
root@shared-emptydir-volume:/#
```

10. Run the following command to locate the mounted directory in the root directory on the container:

```
ls
```

You will see the following output:


```
Note the directory called `mounted-data-2`, which is the
mount point for our Volume inside `container-2`.
```

11. Run the following command to check the content of the `mounted-data-2` directory:

```
cd mounted-data-2
ls
```

You will see the following output:

```
data-1.txt
```

This output indicates that there's already a file called `data-1.txt`, which we created in `container-1` earlier.

12. Let's verify that it's the same file that we created in earlier steps. Run the following command to check the content of this file:

```
cat data-1.txt
```

You will see the following output:

```
Data written on container-1
```

This output verifies that this is the same file that we created in earlier steps of this exercise.

13. Run the following command to write a new file called `data-2.txt` into this directory:

```
echo 'Data written on container-2' > data-2.txt
```

14. Now, let's confirm that the file has been created:

```
ls
```

You should see the following output:

```
data-1.txt   data-2.txt
```

As you can see in this screenshot, the new file has been created and there are now two files -- `data-1.txt` and `data-2.txt` -- in the mounted directory.

15. Run the following command to exit the Bash session on this container:

```
exit
```

16. Now, let's run Bash inside `container-3`:

```
kubectl exec shared-emptydir-volume -c container-3 -it -- /bin/bash
```

You will see the following in your terminal:

```
root@shared-empty-dir-volume:/#
```

17. Go to the `/mounted-data-3` directory and check its content:

```
cd mounted-data-3
ls
```

You will see the following output:

```
data-1.txt   data-2.txt
```

This output shows that this container can see the two files -- `data-1.txt` and `data-2.txt` -- that we created in earlier steps from `container-1` and `container-2`, respectively.

18. Run the following command to verify the content of the first file, `data-1.txt`:

```
cat data-1.txt
```

You should see the following output:

```
Data written on container-1
```

19. Run the following commands to verify the content of the second file, `data-2.txt`:

    ```
    cat data-2.txt
    ```

    You should see the following output:

    ```
    Data written on container-2
    ```

    The output of the last two commands proves that the data written by any container on the mounted volume is accessible by other containers for reading. Next, we will verify that other containers have write access to the data written by a particular container.

20. Run the following command to overwrite the content of the `data-2.txt` file:

    ```
    echo 'Data updated on container 3' > data-2.txt
    ```

21. Next, let's exit `container-3`:

    ```
    exit
    ```

22. Run the following command to run Bash inside `container-1` again:

    ```
    kubectl exec shared-emptydir-volume -c container-1 -it -- /bin/bash
    ```

    You should see the following in your terminal:

    ```
    root@shared-emptydir-volume:/#
    ```

23. Run the following command to check the content of the `data-2.txt` file:

    ```
    cat mounted-data-1/data-2.txt
    ```

    You should see the following output:

    ```
    Data updated on container 3
    ```

    This output indicates that the data overwritten by `container-3` becomes available for other containers to read as well.

24. Run the following command to come out of the SSH session inside `container-3`:

    ```
    exit
    ```

25. Run the following command to delete the pod:

    ```
    kubectl delete pod shared-emptydir-volume
    ```

    You should see the following output, indicating that the pod has been deleted:

    ```
    pod "shared-emptydir-volume" deleted
    ```

In this exercise, we learned how to use Volumes and verified that the same Volume can be mounted at different paths in different containers. We also saw that the containers using the same Volume can read or write (or overwrite) content of the Volume.

Persistent Volumes

The Volumes we have seen so far have the limitation that their life cycle depends on the life cycle of pods. Volumes such as emptyDir or hostPath get deleted when the pod using them is deleted or gets restarted. For example, if we use Volumes to store user data and inventory records for our e-commerce website, the data will be deleted when the application pod restarts. Hence, Volumes are not suited to store data that you want to persist.

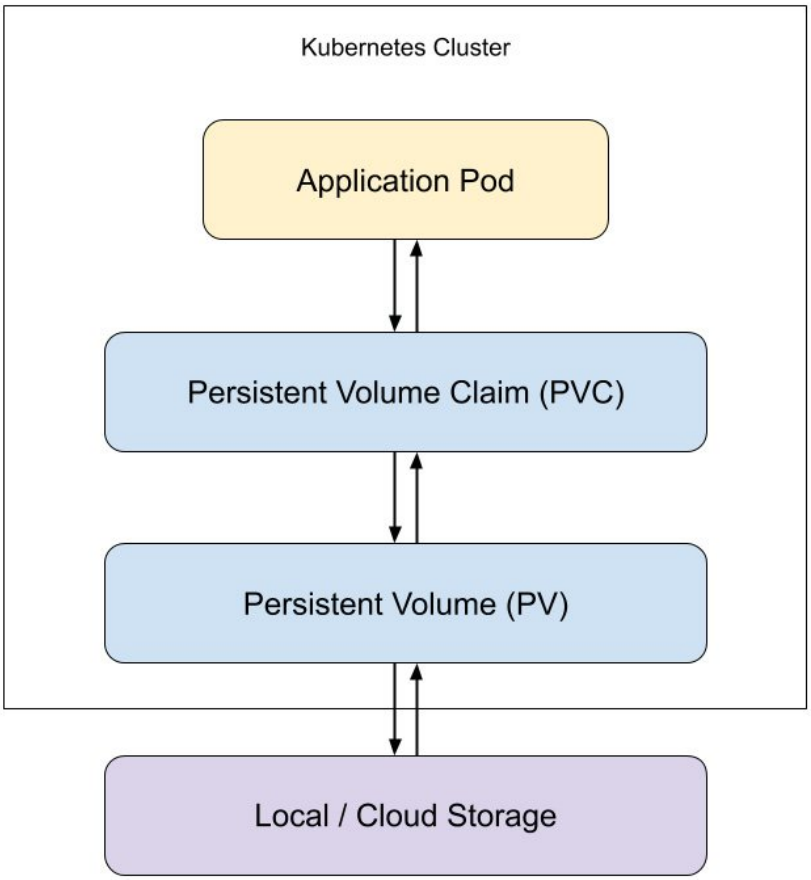
To solve this problem, Kubernetes supports persistent storage in the form of a **Persistent Volume** (**PV**). A PV is a Kubernetes object that represents a block of storage in the cluster. It can either be provisioned beforehand by the cluster administrators or be dynamically provisioned. A PV can be considered a cluster resource just like a node and, hence, it is not scoped to a single namespace. These Volumes work similarly to the Volumes we have seen in previous sections. The life cycle of a PV doesn't depend on the life cycle of any pod that uses the PV. From the pod's perspective, however, there's no difference between using a normal Volume and a PV.

In order to use a PV, a **PersistentVolumeClaim** (**PVC**) needs to be created. A PVC is a request for storage by a user or a pod. A PVC can request a specific size of storage and specific access modes. A PVC is effectively an abstract way of accessing the various storage resources by users. PVCs are scoped by namespaces, so pods can only access the PVCs created within the same namespace.

Note

At any time, a PV can be bound to one PVC only.

Here's a diagram showing how an application interacts with a PV and PVC:



As you can see in this diagram, Kubernetes uses a combination of PV and PVC to make storage available to your applications. A PVC is basically a request to provide a PV that meets certain criteria.

This is a notable variation from what we saw in the previous exercises, where we created Volumes directly in the pod definitions. This separation of the request (PVC) and the actual storage abstraction (PV) allows an application developer to not worry about the specifics and the statuses of all the different PVs present on the cluster; they can simply create a PVC with the application requirements and then use it in the pod. This kind of loose binding also allows the entire system to be resilient and remain stable in the case of pod restarts.

Similar to Volumes, Kubernetes supports several types of PVs. Some of them may be specific to your cloud platform. You can find a list of the different supported types at this link: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes

PersistentVolume Configuration

Here's an example of PV configuration:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: example-persistent-volume
spec:
  storageClassName: standard
  capacity:
    storage: 10Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Retain
  nfs:
    server: 172.10.1.1
    path: /tmp/pv
```

As usual, the PV object also has the three fields that we have already seen: `apiVersion`, `kind`, and `metadata`. Since this is an `nfs` type of PV, we have the `nfs` section in the configuration. Let's go through some important fields in the PV `spec` section one by one.

storageClassName

Each PV belongs to a certain storage class. We define the name of the storage class that the PV is associated with using the `storageClassName` field. A StorageClass is a Kubernetes object that provides a way for administrators to describe the different types or profiles of storages they support. In the preceding example, `standard` is just an example of a storage class.

Different storage classes allow you to allocate different types of storage based on performance and capacity to different applications based on the specific needs of the application. Each cluster administrator can configure their own storage classes. Each storage class can have its own provisioners, backup policies, or reclamation policies determined by administrators. A provisioner is a system that determines how to provision a PV of a particular type. Kubernetes supports a set of internal provisioners as well as external ones that can be implemented by users. The details about how to use or create a provisioner are, however, beyond the scope of this book.

A PV belonging to a certain storage class can only be bound to a PVC requesting that particular class. Note that this is an optional field. Any PV without the storage class field will only be available to PVCs that do not request a specific storage class.

capacity

This field denotes the storage capacity of the PV. We can set this field in a similar way as we would define constraints used by memory and CPU limit fields in a pod spec. In the preceding example spec, we have set the capacity to 10 GiB.

volumeMode

The `volumeMode` field denotes how we want the storage to be used. It can have two possible values: `Filesystem` (default) and `Block`. We can set the `volumeMode` field to `Block` in order to use the raw block device as storage, or `Filesystem` to use a traditional filesystem on the persistent volume.

accessModes

The access mode for a PV represents the capabilities allowed for a mounted Volume. A Volume can be mounted using only one of the supported access modes at a time. There are three possible access modes:

- `ReadWriteOnce` (`RWO`): Mounted as read-write by a single node only
- `ReadOnlyMany` (`ROX`): Mounted as read-only by many nodes
- `ReadWriteMany` (`RWX`): Mounted as read-write by many nodes

Note that not all the types of volumes support all the access modes. Please check the reference for the allowed access modes for the specific type of volume you are using.

persistentVolumeReclaimPolicy

Once a user is done with a volume, they can delete their PVC, and that allows the PV resource to be reclaimed. The reclaim policy field denotes the policy that will be used to allow a PV to be claimed after its release. A PV being *released* implies that the PV is no longer associated with the PVC since that PVC is deleted. Then, the PV is available for any other PVCs to use, or in other words, *reclaim*. Whether a PV can be reused or not depends on the reclaim policy. There can be three possible values for this field:

- `Retain`: This reclaim policy indicates that the data stored in the PV is kept in storage even after the PV has been released. The administrator will need to delete the data in storage manually. In this policy, the PV is marked as `Released` instead of `Available`. Thus, a `Released` PV may not necessarily be empty.
- `Recycle`: Using this reclaim policy means that once the PV is released, the data on the volume is deleted using a basic `rm -rf` command. This marks the PV as `Available` and hence ready to be claimed again. Using dynamic provisioning is a better alternative to using this reclaim policy. We will discuss the dynamic provisioning in the next section.
- `Delete`: Using this reclaim policy means that once the PV is released, both the PV as well as the data stored in the underlying storage will be deleted.

  Note

  Various cloud environments have different default values for reclaim policies. So, make sure you check the default value of the reclaim policy for the cloud environment you're using to avoid the accidental deletion of data in PVs.

PV Status

At any moment of its life cycle, a PV can have one of the following statuses:

- `Available`: This indicates that the PV is available to be claimed.

- `Bound`: This indicates that the PV has been bound to a PVC.
- `Released`: This indicates that the PVC bound to this resource has been deleted; however, it's yet to be reclaimed by some other PVC.
- `Failed`: This indicates that there was a failure during reclamation.

Now that we have taken a look at the various aspects of the PV, let's take a look at the PVC.

PersistentVolumeClaim Configuration

Here's an example of PVC configuration:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-persistent-volume-claim
spec:
  storageClassName: standard
  resources:
    requests:
      storage: 500Mi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  selector:
    matchLabels:
      environment: "prod"
```

Again, as usual, the PVC object also has three fields that we have already seen: `apiVersion`, `kind`, and `metadata`. Let's go through some important fields in the PVC `spec` section one by one.

storageClassName

A PVC can request a particular class of storage by specifying the `storageClassName` field. Only the PVs of the specified storage class can be bound to such a PVC.

If the `storageClassName` field is set to an empty string (`""`), these PVCs will only be bound to PVs that have no storage class set.

On the other hand, if the `storageClassName` field in the PVC is not set, then it depends on whether `DefaultStorageClass` has been enabled by the administrator. If a default storage class is set for the cluster, the PVCs with no `storageClassName` field set will be bound to PVs with that default storage class. Otherwise, PVCs with no `storageClassName` field set will only be bound to PVs that have no storage class set.

resources

Just as we learned that pods can make specific resource requests, PVCs can also request resources in a similar manner by specifying the `requests` and `limits` fields, which are optional. Only the PVs satisfying the resource requests can be bound to a PVC.

volumeMode

PVCs follow the same convention as PVs to indicate the use of storage as a filesystem or a raw block device. A PVC can only be bound to a PV that has the same Volume mode as the one specified in the PVC configuration.

accessMode

A PVC should specify the access mode that it needs, and a PV is assigned as per the availability based on that access mode.

selectors

Similar to pod selectors in Services, PVCs can use the `matchLabels` and/or `matchExpressions` fields to specify the criteria of volumes that can satisfy a particular claim. Only the PVs whose labels satisfy the conditions specified in the `selectors` field are considered for a claim. When both of these fields are used together as selectors, the conditions specified by the two fields are combined using an AND operation.

How to Use Persistent Volumes

In order to use a PV, we have the following three steps: provisioning the volume, binding it to a claim (PVC), and using the claim as a volume on a pod. Let's go through these steps in detail.

Step 1 -- Provisioning the Volume

A Volume can be provisioned in two ways -- statically and dynamically:

- **Static**: In static provisioning, the cluster administrator has to provision several PVs beforehand, and only then are they available to PVCs as available resources.
- **Dynamic**: If you are using dynamic provisioning, the administrator doesn't need to provision all the PVs beforehand. In this kind of provisioning, the cluster will dynamically provision the PV for the PVC based on the storage class requested. Thus, as the applications or microservices demand more storage, Kubernetes can automatically take care of it and expand the cloud infrastructure as needed.

  We will go through dynamic provisioning in more detail in a later section.

Step 2 -- Binding the Volume to a Claim

In this step, a PVC is to be created with the requested storage limits, a certain access mode, and a specific storage class. Whenever a new PVC is created, the Kubernetes controller will search for a PV matching its criteria. If a PV matching all of the PVC criteria is found, it will bind the claim to the PV. Each PV can be bound to only one PVC at a time.

Step 3 -- Using the Claim

Once the PV has been provisioned and bound to a PVC, the PV can be used by the pod as a Volume. Next, when a pod uses a PVC as a Volume, Kubernetes will take the PV bound to that PVC and mount that PV for the pod.

Here's an example of pod configuration using a PVC as a Volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-pvc-as-volume
spec:
  containers:
  - image: nginx
    name: nginx-application
    volumeMounts:
    - mountPath: /data/application
```

```
      name: example-storage
  volumes:
  - name: example-storage
    persistentVolumeClaim:
      claimName: example-claim
```

In this example, we assume that we have a PVC named `example-claim` that has already been bound to `PersistentVolume`. The pod configuration specifies `persistentVolumeClaim` as the type of the Volume and specifies the name of the claim to be used. Kubernetes will then find the actual PV bound to this claim and mount it on `/data/application` inside the container.

Note

The pod and the PVC have to be in the same namespace for this to work. This is because Kubernetes will look for the claim inside the pod's namespace only, and if the PVC isn't found, the pod will not be scheduled. In this case, the pod will be stuck in a `Pending` state until deleted.

Now, let's put these concepts into action by creating a pod that uses PV in the following exercise.

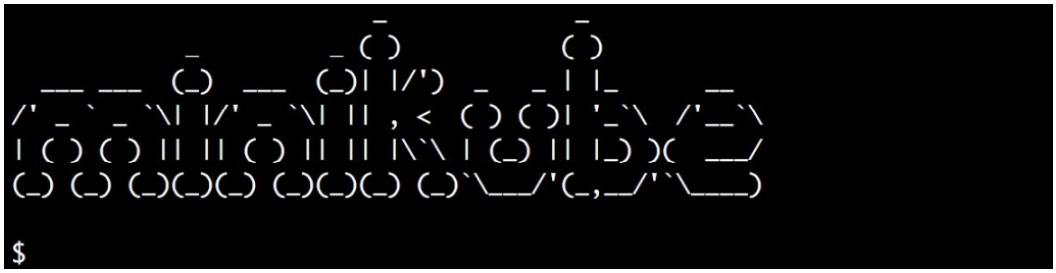
Exercise 9.03: Creating a Pod That Uses PersistentVolume for Storage

In this exercise, we will first provision the PV pretending that the cluster administrator does it in advance. Next, assuming the role of a developer, we will create a PVC that is bound to the PV. After that, we will create a pod that will use this claim as a Volume mounted on one of the containers:

1. First of all, we will access the host node via SSH. In the case of Minikube, we can do so by using the following command:

   ```
   minikube ssh
   ```

   You should see an output similar to this one:

2. Run the following command to create a directory named `data` inside the `/mnt` directory:

   ```
   sudo mkdir /mnt/data
   ```

3. Run the following command to create a file called `data.txt` inside the `/mnt/data` directory:

   ```
   sudo bash -ec 'echo "Data written on host node" > /mnt/data/data.txt'
   ```

   This command should create a file, `data.txt`, with the `Data written on host node` content. We will use the content of this file to verify at a later stage that we can successfully mount this directory on a container using a PV and a PVC.

4. Run the following command to exit the host node:

```
exit
```

That will bring us back to the local machine terminal where we can run `kubectl` commands.

5. Create a file called `pv-hostpath.yaml` with the following content:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-hostpath
spec:
  storageClassName: local-pv
  capacity:
    storage: 500Mi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

In this PV configuration, we have used the `local-pv` storage class. The Volume will be hosted at the `/mnt/data` path on the host node. The size of the volume will be `500Mi` and the access mode will be `ReadWriteOnce`.

6. Run the following command to create the PV using the preceding configuration:

```
kubectl create -f pv-hostpath.yaml
```

You should see the following output:

```
persistentvolume/pv-hostpath created
```

7. Run the following command to check the status of the PV we just created:

```
kubectl get pv pv-hostpath
```

As you can see in this command, `pv` is an accepted shortened name for `PersistentVolume`. You should see the following output:

```
NAME          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS   REASON   AGE
pv-hostpath   500Mi      RWO            Retain           Available           local-pv               113s
```

In the preceding output, we can see that the Volume was created with the required configuration and that its status is `Available`.

8. Create a file called `pvc-local.yaml` with the following content:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-local
spec:
```

```
  storageClassName: local-pv
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

In this configuration, we have a claim that requests a Volume with the `local-pv` storage class, the `ReadWriteOnce` access mode and a storage size of `100Mi`.

9. Run the following command to create this PVC:

```
kubectl create -f pvc-local.yaml
```

You should see the following output:

```
persistentvolumeclaim/pvc-local created
```

Once we create this PVC, Kubernetes will search for a matching PV to satisfy this claim.

10. Run the following command to check the status of this PVC:

```
kubectl get pvc pvc-local
```

You should see the following output:

```
NAME        STATUS   VOLUME       CAPACITY   ACCESS MODES   STORAGECLASS   AGE
pvc-local   Bound    pv-hostpath   500Mi      RWO            local-pv       103s
```

```
As we can see in this output, the PVC has been created with the
required configuration and has been immediately bound to the
existing PV named `pv-hostpath` that we created in earlier
steps of this exercise.
```

11. Next, we can create a pod that will use this PVC as a Volume. Create a file called `pod-local-pvc.yaml` with the following content:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-local-pvc
spec:
  restartPolicy: Never
  containers:
  - image: ubuntu
    name: ubuntu-container
    command: ['/bin/bash', '-ec', 'cat /data/application/data.txt']
    volumeMounts:
    - mountPath: /data/application
      name: local-volume
  volumes:
```

```
  - name: local-volume
    persistentVolumeClaim:
      claimName: pvc-local
```

The pod will use a PVC named `pvc-local` as a Volume and mount it at the `/data/application` path in the container. Also, we have a container that will run the `cat /data/application/data.txt` command on startup. This is just a simplified example where we will showcase that the data we wrote in the PV directory on the host node initially is now available to this pod.

12. Run the following command to create this pod:

```
kubectl create -f pod-local-pvc.yaml
```

You should see the following output:

```
pod/pod-local-pvc created
```

This output indicates that the pod was created successfully.

13. Run the following command to check the status of the pod we just created:

```
kubectl get pod pod-local-pvc
```

You should see the following output:

```
NAME             READY     STATUS      RESTARTS    AGE
pod-local-pvc    0/1       Completed   1           7s
```

In this output, we can see that the pod has run to completion since we didn't add any sleep commands this time.

14. Run the following command to check the logs. We expect to see the output of the `cat /data/application/data.txt` command in the logs:

```
kubectl logs pod-local-pvc
```

You should see the following output:

```
Data written on host node
```

This output clearly indicates that this pod has access to the file that we created at `/mnt/data/data.txt`. This file is a part of the directory mounted at `/data/application` in the container.

15. Now, let's clean up the resources created in this exercise. Use the following command to delete the pod:

```
kubectl delete pod pod-local-pvc
```

You should see the following output, indicating that the pod has been deleted:

```
pod "pod-local-pvc" deleted
```

16. Use this command to delete the PVC:

```
kubectl delete pvc pvc-local
```

You should see the following output, indicating that the PVC has been deleted:

```
persistentvolumeclaim "pvc-local" deleted
```

Note that if we try to delete the PV before the PVC is deleted, the PV will be stuck in the `Terminating` phase and will wait for it to be released by the PVC. Hence, we need to first delete the PVC bound to the PV before the PV can be deleted.

17. Now that our PVC has been deleted, we can safely delete the PV by running the following command:

```
kubectl delete pv pv-hostpath
```

You should see the following output, indicating that the PV has been deleted:

```
persistentvolume "pv-hostpath" deleted
```

In this exercise, we learned how to provision PVs, create claims to use these volumes, and then use those PVCs as volumes inside pods.

Dynamic Provisioning

In previous sections of this chapter, we saw that the cluster administrator needs to provision PVs for us before we can use them as storage for our application. To solve this problem, Kubernetes supports dynamic volume provisioning as well. Dynamic volume provisioning enables the creation of storage volumes on-demand. This eliminates the need for administrators to create volumes before creating any PVCs. The volume is provisioned only when there's a claim requesting it.

In order to enable dynamic provisioning, the administrator needs to create one or more storage classes that users can use in their claims to make use of dynamic provisioning. These `StorageClass` objects need to specify what provisioner will be used along with its parameters. The provisioner depends on the environment. Every cloud provider supports different provisioners, so make sure you check with your cloud provider if you happen to create this kind of storage class in your cluster.

Here's an example of the configuration for creating a new `StorageClass` on the AWS platform:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: example-storage-class
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4
```

In this configuration, the `kubernetes.io/aws-ebs` provisioner is used -- EBS stands for Elastic Block Store and is only available on AWS. This provisioner takes various parameters, including `type`, which we can use to specify what kind of disk we want to use for this storage class. Please check the AWS docs to find out more about the various parameters we can use and their possible values. The provisioner and the parameters required will change based on what cloud provider you use.

Once a storage class is created by the cluster administrator, users can create a PVC, requesting storage with that storage class name set in the `storageClassName` field. Kubernetes will then automatically provision the storage volume, create a PV object with that storage class satisfying the claim, and bind it to the claim:

Here's an example of the configuration for a PVC using the storage class we defined previously:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: example-pvc
spec:
  storageClassName: example-storage-class
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

As we can see, the configuration of the PVC stays the same, except that now, we have to use a storage class that has already been created by the cluster administrator for us.

Once the claim has been bound to an automatically created Volume, we can create pods using that PVC as a Volume, as we saw in the previous section. Once the claim is deleted, the Volume is automatically deleted.

Activity 9.01: Creating a Pod That Uses a Dynamically Provisioned PersistentVolume

Consider that you are a cluster administrator, at first, and are required to create a custom storage class that will enable the developers using your cluster to provision PVs dynamically. To create a storage class on a minikube cluster, you can use the `k8s.io/minikube-hostpath` provisioner without any extra parameters, similar to what we showed in the `StorageClass` example in the *Dynamic Provisioning* section.

Next, acting as a developer or a cluster user, claim a PV with a storage request of 100Mi and mount it on the containers inside the pod created using the following specifications:

1. The pod should have two containers.
2. Both the containers should mount the same PV locally.
3. The first container should write some data into the PV and the second container should read and print out the data written by the first container.

For simplicity, consider writing a simple string to a file in the PV from the first container. For the second container, add a bit of wait time so that the second container does not start reading data until it is fully written. Then, the latter container should read and print out the content of the file written by the first container.

Note

Ideally, you would want to create this deployment to be in a different namespace to keep it separate from the rest of the stuff that you created during these exercises. So, feel free to create a namespace and create all the objects in this activity in that namespace.

The high-level steps to perform this activity are as follows:

1. Create a namespace for this activity.
2. Write the appropriate configuration for the storage class using the given information, and create the `StorageClass` object.

3. Write the appropriate configuration for the PVC using the storage class created in the previous step. Create the PVC using this configuration.
4. Verify that the claim was bound to an automatically created PV of the same storage class that we created in *step 2*.
5. Write the appropriate configuration for the pod using the given information and the PVC from the previous step as a Volume. Create the pod using this configuration.
6. Verify that one of the containers can read the content of the file written to PV by another container.

You should be able to check the logs of the second container and verify that the data written by the first container in the PV can be read by the second container, as shown in the following output:

```
Data written by container-1
```

Note

The solution to this activity can be found at the following address: `Activity_Solutions\Solution_Final.pdf`.

Summary

As we mentioned in the introduction, most applications need to store or retrieve data for a lot of different reasons. In this chapter, we saw that Kubernetes provides various ways of provisioning storage for not just storing the state of an application, but also for the long-term storage of data.

We have covered ways to use storage for our application running inside pods. We saw how we can use the different types of Volumes to share temporary data among containers running in the same pod. We also learned how to persist data across pod restarts. We learned how to manually provision PVs to create PVCs to bind to those Volumes, as well as how to create pods that can use these claims as Volumes mounted on their containers. Next, we learned how to request storage dynamically using only the PVCs with pre-created storage classes. We also learned about the life cycle of these volumes with respect to that of the pods.

In the next chapter, we will extend these concepts further and learn how to store application configurations and secrets.