Application Storage Basics

If a pod in OpenShift needs reliable storage, for example, to host a database, we would need to supply a**persistent** volume to the pod. This type of storage outlives the container, i.e. it persists when the pod dies. It typically comes from an external storage system.

> Doing these exercises requires that you already have deployed the application featured in the Application Management Basics exercises. You should do those exercises now before continuing.

The `mapit` application currently doesn't leverage any persistent storage. If the pod dies, so does all the content inside the container.

We will talk about this concept in more detail later. But let's imagine for a moment, the `mapit` application needs persistent storage available under the `/app-storage` directory inside the container.

> The directories that make up the container's internal filesystem are a blend of the read-only layers of the container image and the top-most writable layer that is added as soon as a container instance is started from the image. The writable layer is disposed of once the container is deleted which can happen regularly in a dynamic container orchestration environment.

You should be in the `app-management` project from the previous lab. To make sure, run the following command:

```
oc project app-management
```

Adding Persistent Volume Claims

Here's how you would instruct OpenShift to create a `PersistentVolume` object, which represents external, persistent storage, and have it **mounted** inside the container's filesystem:

```
oc set volume deploy/mapit --add --name=mapit-storage -t pvc --claim-mode=ReadWriteOnce --claim-size=1Gi --claim-name=mapit-
storage --mount-path=/app-storage
```

The output looks like this:

```
deployment.apps/mapit volume updated
```

In the first step a **PersistentVolumeClaim** was created. This object represents a request for storage of a certain kind, with a certain capacity from the user to OpenShift.

Next the `Deployment` of `mapit` is updated to reference this storage and make it available under the `/app-storage` directory inside the pod.

You can see the new `Deployment` like this:

```
oc get deploy mapit
```

The output will show that a new deployment was created by inspecting the`AGE` column.

```
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
mapit   1/1     1            1           14m
```

Likely, depending when you ran the command, you may or may not see that the new pod is still being spawned:

```
oc get pod
```

```
NAME             READY     STATUS              RESTARTS   AGE
mapit-3-ntd9w    1/1       Running             0          9m
mapit-4-d872b    0/1       ContainerCreating   0          5s
mapit-4-deploy   1/1       Running             0          10s
```

Take a look at the `Deployment` now:

```
oc describe deploy mapit
```

You will see there is now both `Mounts` and `Volumes` details about the new storage:

```
...
    Mounts:
      /app-storage from mapit-storage (rw)
  Volumes:
   mapit-storage:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  mapit-storage
    ReadOnly:   false
...
```

But what is happening under the covers?

Storage Classes

When OpenShift 4 was first installed, a dynamic storage provider for AWS EBS was configured. You can see this `StorageClass` with the following:

```
oc get storageclass
```

And you will see something like:

```
NAME            PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ALLOWVOLUMEEXPANSION   AGE
gp2 (default)   kubernetes.io/aws-ebs   Delete          WaitForFirstConsumer   true                   47m
gp2-csi         ebs.csi.aws.com         Delete          WaitForFirstConsumer   true                   47m
```

Any time a request for a volume is made (`PersistentVolumeClaim`) that doesn't specify a `StorageClass`, the default will be used. In this case, the default is an EBS provisioner that will create an EBS GP2 volume of the requested size (in our example, 1Gi).

> You'll note that there is also a `gp2-csi`. This implements the CSI interface, which stands for "Container Storage Interface". This specification enables storage vendors to develop their plugins once, and have it work across various container orchestration systems.

Persistent Volume (Claims)

The command you ran earlier referenced a `claim`. Storage in a Kubernetes environment uses a system of volume claims and volumes. A user makes a `PersistentVolumeClaim` and Kubernetes tries to find a `PersistentVolume` that matches. In the case where a volume does not already exist, if there is a dynamic provisioner that satisfies the claim, a `PersistentVolume` is created.

Execute the following:

```
oc get persistentvolume
```

You will see something like the following:

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS   REASON   AGE
pvc-4397c6be-9f53-490e-960d-c1b77de6000c   1Gi        RWO            Delete           Bound    app-management/mapit-storage
gp2                     12m
```

This is the volume that was created as a result of your earlier claim. Note that the volume is**bound** to the claim that exists in the `app-management` project.

Now, execute:

```
oc get persistentvolumeclaim -n app-management
```

You will see something like:

```
NAME            STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mapit-storage   Bound    pvc-4397c6be-9f53-490e-960d-c1b77de6000c   1Gi        RWO            gp2            14m
```

Testing Persistent Storage

Log on to the pod using the remote-shell capability of the `oc` client:

```
oc rsh $(oc get pods -l deployment=mapit -o name)
```

Being in the container's shell session, list the content of the root directory from the perspective of the container's namespace:

```
ls -ahl /
```

You will see a directory there called `/app-storage`:

```
total 20K
drwxr-xr-x.   1 root  root         81 Apr 12 19:11 .
drwxr-xr-x.   1 root  root         81 Apr 12 19:11 ..
-rw-r--r--.   1 root  root        16K Dec 14  2016 anaconda-post.log
drwxrwsr-x.   3 root  1000570000 4.0K Apr 12 19:10 app-storage (1)
lrwxrwxrwx.   1 root  root          7 Dec 14  2016 bin -> usr/bin
drwxrwxrwx.   1 jboss root         45 Aug  4  2017 deployments
drwxr-xr-x.   5 root  root        360 Apr 12 19:11 dev
drwxr-xr-x.   1 root  root         93 Jan 18  2017 etc
drwxr-xr-x.   2 root  root          6 Nov  5  2016 home
lrwxrwxrwx.   1 root  root          7 Dec 14  2016 lib -> usr/lib
lrwxrwxrwx.   1 root  root          9 Dec 14  2016 lib64 -> usr/lib64
drwx------.   2 root  root          6 Dec 14  2016 lost+found
drwxr-xr-x.   2 root  root          6 Nov  5  2016 media
drwxr-xr-x.   2 root  root          6 Nov  5  2016 mnt
drwxr-xr-x.   1 root  root         19 Jan 18  2017 opt
dr-xr-xr-x. 183 root  root          0 Apr 12 19:11 proc
dr-xr-x---.   2 root  root        114 Dec 14  2016 root
drwxr-xr-x.   1 root  root         21 Apr 12 19:11 run
lrwxrwxrwx.   1 root  root          8 Dec 14  2016 sbin -> usr/sbin
drwxr-xr-x.   2 root  root          6 Nov  5  2016 srv
dr-xr-xr-x.  13 root  root          0 Apr 10 14:34 sys
drwxrwxrwt.   1 root  root         92 Apr 12 19:11 tmp
drwxr-xr-x.   1 root  root         69 Dec 16  2016 usr
drwxr-xr-x.   1 root  root         41 Dec 14  2016 var
```

1. This is where the persistent storage appears inside the container

Amazon EBS volumes are read-write-once. In other words, because they are block storage, they may only be attached to one EC2 instance at a time, which means that only one container can use an EBS-based `PersistentVolume` at a time. In other words: read-write-once.

Execute the following inside the remote shell session:

```
echo "Hello World from OpenShift" > /app-storage/hello.txt
exit
```

Then, make sure your file is present:

```
oc rsh $(oc get pods -l deployment=mapit -o name) cat /app-storage/hello.txt
```

Now, to verify that persistent storage really works, delete your pod:

```
oc delete pods -l deployment=mapit && oc get pod
```

After some time, your new pod will be ready and running. Once it’s running, check the file:

```
oc rsh $(oc get pods -l deployment=mapit -o name) cat /app-storage/hello.txt
```

It’s still there. In fact, the new pod may not even be running on the same node as the old pod, which means that, under the covers, Kubernetes and OpenShift automatically attached the real, external storage to the right place at the right time.

If you needed read-write-many storage, file-based storage solutions can provide it. OpenShift Container Storage is a hyperconverged storage solution that can run inside OpenShift and provide file, block and even object storage by turning locally attached storage devices into storage pools and then creating volumes out of them.