Container-Native Storage Management

8

In this lab we are going to provide a view 'under the hood' of OpenShift `PersistentVolumes` provided by CNS. For this purpose we will examine volumes leveraged by example applications using different volume access modes.

A Simple CNS Use Case

We are going to deploy a sample application that ships with OpenShift which creates a PVC as part of the deployment. Log on to the system as `fancyuser1`, using the password `openshift` and create a project with the name `my-database-app`.

Create/Deploy the Application

```
oc login -u fancyuser1 -p openshift
oc new-project my-database-app
```

The example applications ships in form of ready-to-use resource templates. Enter the following command to look at the template for a sample Ruby on Rails application with a PostgreSQL database:

```
oc get template/rails-pgsql-persistent -n openshift -o yaml
```

This template creates a Rails Application instance which mimics a very basic weblog. The articles and comments are saved in a PostgreSQL database which runs in another pod. As part of the resource template a PVC is issued (line 194) to supply the PostgreSQL pod with persistent storage with the mount point `/var/lib/pgsql/data` (line 275). This will request a **PersistentVolume** in `RWO` mode. Storage provided in this mode can only be mounted by a single pod at a time.

If you want to see more about the parameters or other details of this template, you can execute the following:

```
oc describe template rails-pgsql-persistent -n openshift
```

The storage size parameter in the template is called `VOLUME_CAPACITY`. The `new-app` command will actually handle processing and interpreting a **Template** into the appropriate OpenShift objects. We will specify that we want *5Gi* of storage as part of deploying a new app from the template as follows:

```
oc new-app rails-pgsql-persistent -p VOLUME_CAPACITY=5Gi
```

The `new-app` command will automatically check for templates in the special `openshift` namespace. In fact, `new-app` tries to do quite a lot of interesting automagic things, including code introspection when pointed at code repositories. It is a developer's good friend.

You will then see something like the following:

```
--> Deploying template "openshift/rails-pgsql-persistent" to project my-

     Rails + PostgreSQL (Persistent)
     ---------
     An example Rails application with a PostgreSQL database. For more i

     The following service(s) have been created in your project: rails-p

     For more information about using this template, including OpenShift


     * With parameters:
        * Name=rails-pgsql-persistent
        * Namespace=openshift
        * Memory Limit=512Mi
        * Memory Limit (PostgreSQL)=512Mi
        * Volume Capacity=5Gi
        * Git Repository URL=https://github.com/openshift/rails-ex.git
        * Git Reference=
        * Context Directory=
        * Application Hostname=
        * GitHub Webhook Secret=yGhTIuuUjH7JHClrCtYYbY2FdtT0RF5oxA77tGWC
        * Secret Key=8phdjyreu8vaai84ffmvyw18vc3awvgje1c4mw42uplrcvf0dbc
        * Application Username=openshift
        * Application Password=secret
        * Rails Environment=production
        * Database Service Name=postgresql
        * Database Username=userP8B # generated
        * Database Password=USrJhqh6 # generated
        * Database Name=root
        * Maximum Database Connections=100
        * Shared Buffer Amount=12MB
        * Custom RubyGems Mirror URL=

--> Creating resources ...
    secret "rails-pgsql-persistent" created
    service "rails-pgsql-persistent" created
    route "rails-pgsql-persistent" created
```

```
    imagestream "rails-pgsql-persistent" created
    buildconfig "rails-pgsql-persistent" created
    deploymentconfig "rails-pgsql-persistent" created
    persistentvolumeclaim "postgresql" created
    service "postgresql" created
    deploymentconfig "postgresql" created
--> Success
    Build scheduled, use 'oc logs -f bc/rails-pgsql-persistent' to track
    Run 'oc status' to view your app.
```

Go back to the OpenShift web console:

https://openshift.193183262213.aws.testdrive.openshift.com/console

Make sure you are logged in as *fancyuser1* and find your newly created project `my-database-app`. You can now follow the deployment process. Alternatively, watch the containers deploy from the command line like this:

```
oc get pods -w
```

You will see something like:

```
NAME                              READY     STATUS              RESTARTS
postgresql-1-deploy               0/1       ContainerCreating   0
rails-pgsql-persistent-1-build    0/1       ContainerCreating   0
NAME                   READY     STATUS    RESTARTS   AGE
postgresql-1-deploy    1/1       Running   0          14s
postgresql-1-81gnm     0/1       Pending   0          0s
postgresql-1-81gnm     0/1       Pending   0          0s
rails-pgsql-persistent-1-build   1/1       Running   0         19s
```

```
postgresql-1-81gnm   0/1       Pending   0         15s
postgresql-1-81gnm   0/1       ContainerCreating   0         16s
postgresql-1-81gnm   0/1       Running   0         47s
postgresql-1-81gnm   1/1       Running   0         4m
postgresql-1-deploy   0/1       Completed   0         4m
postgresql-1-deploy   0/1       Terminating   0         4m
postgresql-1-deploy   0/1       Terminating   0         4m
rails-pgsql-persistent-1-deploy   0/1       Pending   0         0s
rails-pgsql-persistent-1-deploy   0/1       Pending   0         0s
rails-pgsql-persistent-1-deploy   0/1       ContainerCreating   0
rails-pgsql-persistent-1-build   0/1       Completed   0         11m
rails-pgsql-persistent-1-deploy   1/1       Running   0         6s
rails-pgsql-persistent-1-hook-pre   0/1       Pending   0         0s
rails-pgsql-persistent-1-hook-pre   0/1       Pending   0         0s
rails-pgsql-persistent-1-hook-pre   0/1       ContainerCreating   0
rails-pgsql-persistent-1-hook-pre   1/1       Running   0         6s
rails-pgsql-persistent-1-hook-pre   0/1       Completed   0         15s
rails-pgsql-persistent-1-dkj7w   0/1       Pending   0         0s
rails-pgsql-persistent-1-dkj7w   0/1       Pending   0         0s
rails-pgsql-persistent-1-dkj7w   0/1       ContainerCreating   0
rails-pgsql-persistent-1-dkj7w   0/1       Running   0         1m
rails-pgsql-persistent-1-dkj7w   1/1       Running   0         1m
rails-pgsql-persistent-1-deploy   0/1       Completed   0         1m
rails-pgsql-persistent-1-deploy   0/1       Terminating   0         1m
rails-pgsql-persistent-1-deploy   0/1       Terminating   0         1m
rails-pgsql-persistent-1-hook-pre   0/1       Terminating   0         1m
rails-pgsql-persistent-1-hook-pre   0/1       Terminating   0         1m
```

Exit out of the watch mode with `Ctrl` + `c` when you see the *-deploy and *-hook-pre pods terminating.

It may take up to 5 minutes for the deployment to complete, and you might not see *exactly* the same output, depending on when you first start watching (`-w`) the **Pod** list.

You should now also see a PVC that has been issued and now being in the *Bound* state.

```
oc get pvc
```

You will see something like:

```
NAME         STATUS    VOLUME                                     CAPACI
postgresql   Bound     pvc-9bb84d88-4ac6-11e7-b56f-2cc2602a6dc8   5Gi
```

This PVC has been automatically fulfilled by CNS because the `cns-gold` **StorageClass** was set up as the system-wide default in lab module "Deploying Container-native Storage"

Try the Application

Now go ahead and try out the application. The overview page in the OpenShift UI will tell you the **Route** which has been deployed as well. Otherwise get it on the CLI like this:

```
oc get route
```

You will see something like:

```
NAME                     HOST/PORT
rails-pgsql-persistent   rails-pgsql-persistent-my-database-app.apps.193
```

Following this output, point your browser to:

http://rails-pgsql-persistent-my-database-app.apps.193183262213.aws.testdrive.openshift.com/articles

The username/password to create articles and comments is by default 'openshift'/'secret'.

You should be able to successfully create articles and comments. When they are saved they are actually saved in the PostgreSQL database which stores it's table spaces on a GlusterFS volume provided by CNS.

This application's template included a **Route** object definition, which is why the **Service** was automatically exposed. This is a good practice.

Explore the Underlying CNS Artifacts

Now let's take a look at how this was deployed on the GlusterFS side. First you need to acquire necessary permissions:

```
oc login -u system:admin
```

Select the example project of the user `fancyuser1` if not already/still selected:

```
oc project my-database-app
```

Look at the PVC to determine the PV:

```
oc get pvc
```

You will see something like:

```
NAME         STATUS    VOLUME                                     CAPACI
postgresql   Bound     pvc-9bb84d88-4ac6-11e7-b56f-2cc2602a6dc8   5Gi
```

Your PV name will be different as it's dynamically generated.

Look at the details of the PV bound to the PVC, in this case `pvc-9bb84d88-4ac6-11e7-b56f-2cc2602a6dc8`:

```
oc describe pv/pvc-9bb84d88-4ac6-11e7-b56f-2cc2602a6dc8
```

You will see something like:

```
Name:           pvc-9bb84d88-4ac6-11e7-b56f-2cc2602a6dc8 (1)
Labels:         <none>
StorageClass:   cns-gold
Status:         Bound
Claim:          my-database-app/postgresql
Reclaim Policy: Delete
Access Modes:   RWO
Capacity:       5Gi
```

```
Message:
Source:
    Type:               Glusterfs (a Glusterfs mount on the host that sh
    EndpointsName:      glusterfs-dynamic-postgresql
    Path:               vol_e8fe7f46fedf7af7628feda0dcbf2f60 (2)
    ReadOnly:           false
No events.
```

(1) The unique name of this PV in the system OpenShift refers to

(2) The unique volume name backing the PV known to GlusterFS

Note the GlusterFS volume name, in this case **vol_e8fe7f46fedf7af7628feda0dcbf2f60**.

Now let's switch to the namespace we used for CNS deployment:

```
oc project container-native-storage
```

Look at the GlusterFS pods running and pick one (which one is not important):

```
oc get pods -o wide
```

You will see something like:

```
NAME              READY     STATUS    RESTARTS   AGE       IP
glusterfs-37vn8   1/1       Running   0          3m       10.0.1.6
```

```
glusterfs-cq68l   1/1       Running   0          3m        10.0.3.137
glusterfs-m9fvl   1/1       Running   0          3m        10.0.4.68
heketi-1-cd032    1/1       Running   0          1m        10.0.4.68
```
(2)

Remember the IP address of the pod you select, for example: **10.0.1.6** of pod **glusterfs-37vn8**.
Log on to the selected GlusterFS pod with a remote terminal session like so:

```
oc rsh glusterfs-37vn8
```

You have now access to this container's namespace which has the GlusterFS CLI utilities installed.
Let's use them to list all known volumes:

```
sh-4.2# gluster volume list
```

You will see something like:

```
heketidbstorage (1)
vol_e8fe7f46fedf7af7628feda0dcbf2f60 (2)
vol_5e1cd71070734a3b02f58d822f89486a
vol_f2e8fda1d42a41efabbb4d4a3b4a5659
```

(1) A special volume dedicated to heketi's internal database.

2 The volume backing the PV of the PostgreSQL database deployed earlier.

Query GlusterFS about the topology of this volume:

```
sh-4.2# gluster volume info vol_e8fe7f46fedf7af7628feda0dcbf2f60
```

You will see something like:

```
Volume Name: vol_e8fe7f46fedf7af7628feda0dcbf2f60
Type: Replicate
Volume ID: c2bedd16-8b0d-432c-b9eb-4ab1274826dd
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 3 = 3
Transport-type: tcp
Bricks:
Brick1: 10.0.3.137:/var/lib/heketi/mounts/vg_63b05bee6695ee5a63ad95bfbce
Brick2: 10.0.1.6:/var/lib/heketi/mounts/vg_0246fd563709384a3cbc3f3bbeeb8
Brick3: 10.0.4.68:/var/lib/heketi/mounts/vg_5a8c767e65feef7455b58d01c693
Options Reconfigured:
transport.address-family: inet
performance.readdir-ahead: on
nfs.disable: on
```

1 According to the output of `oc get pods -o wide` this is the container we are logged on to.

Identify the right brick by looking at the host IP of the GlusterFS pod you have just logged on to. `oc get pods -o wide` will give you this information. The host's IP will be noted next to one of the bricks.

GlusterFS created this volume as a 3-way replica set across all GlusterFS pods, in therefore across all your OpenShift App nodes running CNS. + Each pod/node exposes it's local storage via the GlusterFS protocol. This local storage is known as a **brick** in GlusterFS and is usually backed by a local SAS disk or NVMe device. The brick is simply a directory on a block device formatted with XFS and thus made available to GlusterFS.

You can even look at this yourself, by listing the files in the brick directory. Select the brick's directory (the path starting with `/var/lib/heketi/…`) marked in the output above:

```
sh-4.2# ls -ahl /var/lib/heketi/mounts/vg_0246fd563709384a3cbc3f3bbeeb87
```

You will see something like:

```
total 16K
drwxrwsr-x.   5 root       2001   57 Jun  6 14:44 .
drwxr-xr-x.   3 root       root   19 Jun  6 14:44 ..
drw---S---. 263 root       2001 8.0K Jun  6 14:46 .glusterfs
drwxr-sr-x.   3 root       2001   25 Jun  6 14:44 .trashcan
drwx------.  20 1000080000 2001 8.0K Jun  6 14:46 userdata
```

Then, try looking at the data folder:

```
sh-4.2# ls -ahl /var/lib/heketi/mounts/vg_0246fd563709384a3cbc3f3bbeeb87
```

You will see something like:

```
total 68K
drwx------. 20 1000080000 2001 8.0K Jun  6 14:46 .
drwxrwsr-x.  5 root       2001   57 Jun  6 14:44 ..
-rw-------.  2 1000080000 root    4 Jun  6 14:44 PG_VERSION
drwx------.  6 1000080000 root   54 Jun  6 14:46 base
drwx------.  2 1000080000 root 8.0K Jun  6 14:47 global
drwx------.  2 1000080000 root   18 Jun  6 14:44 pg_clog
drwx------.  2 1000080000 root    6 Jun  6 14:44 pg_commit_ts
drwx------.  2 1000080000 root    6 Jun  6 14:44 pg_dynshmem
-rw-------.  2 1000080000 root 4.6K Jun  6 14:46 pg_hba.conf
-rw-------.  2 1000080000 root 1.6K Jun  6 14:44 pg_ident.conf
drwx------.  2 1000080000 root   32 Jun  6 14:46 pg_log
drwx------.  4 1000080000 root   39 Jun  6 14:44 pg_logical
drwx------.  4 1000080000 root   36 Jun  6 14:44 pg_multixact
drwx------.  2 1000080000 root   18 Jun  6 14:46 pg_notify
drwx------.  2 1000080000 root    6 Jun  6 14:44 pg_replslot
drwx------.  2 1000080000 root    6 Jun  6 14:44 pg_serial
drwx------.  2 1000080000 root    6 Jun  6 14:44 pg_snapshots
drwx------.  2 1000080000 root    6 Jun  6 14:46 pg_stat
drwx------.  2 1000080000 root   84 Jun  6 15:16 pg_stat_tmp
drwx------.  2 1000080000 root   18 Jun  6 14:44 pg_subtrans
drwx------.  2 1000080000 root    6 Jun  6 14:44 pg_tblspc
drwx------.  2 1000080000 root    6 Jun  6 14:44 pg_twophase
drwx------.  3 1000080000 root   60 Jun  6 14:44 pg_xlog
-rw-------.  2 1000080000 root   88 Jun  6 14:44 postgresql.auto.conf
-rw-------.  2 1000080000 root  21K Jun  6 14:46 postgresql.conf
-rw-------.  2 1000080000 root   46 Jun  6 14:46 postmaster.opts
-rw-------.  2 1000080000 root   89 Jun  6 14:46 postmaster.pid
```

The exact path name will be different in your environment as it has been automatically generated.

You are looking at the PostgreSQL internal data file structure from the perspective of the GlusterFS server side. It's a normal local filesystem here.

Clients, like the OpenShift nodes and their application pods talk to this storage with the GlusterFS protocol. Which abstracts the 3-way replication behind a single FUSE mount point. + When a pod starts that mounts storage from a PV backed by GlusterFS, OpenShift will mount the GlusterFS volume on the right app node and then *bind-mount* this directory to the right pod. + This is happening transparently to the application inside the pod and looks like a normal local filesystem.

You may now exit your remote session to the GlusterFS pod.

```
sh-4.2# exit
```

Providing Shared Storage With CNS

So far only very few options, like the basic NFS support, existed to provide a **PersistentVolume** to more than one container at once. The access mode used for this is `ReadWriteMany`. Traditional block-based storage solutions are not able to do this.

With CNS this capability is now available to all OpenShift deployments, no matter where they are deployed. To illustrate the benefit of this, we will deploy a PHP application, a file uploader that has multiple front-end instances sharing a common storage repository.+ To highlight the difference this makes to non-shared storage we will first run this application without a PV.

Deploy a New Application

First log back in as `fancyuser1` using the password `openshift` and create a new project:

```
oc login -u fancyuser1 -p openshift
oc new-project my-shared-storage
```

Next deploy the example application:

```
oc new-app openshift/php:7.0~https://github.com/christianh814/openshift-
```

You will see something like:

```
--> Found image a1ebebb (6 weeks old) in image stream "openshift/php" ur

    Apache 2.4 with PHP 7.0
    -----------------------
    Platform for building and running PHP 7.0 applications

    Tags: builder, php, php70, rh-php70

    * A source build using source code from https://github.com/christiar
      * The resulting image will be pushed to image stream "file-uploade
      * Use 'start-build' to trigger a new build
    * This image will be deployed in deployment config "file-uploader"
    * Port 8080/tcp will be load balanced by service "file-uploader"
      * Other containers can access this service through the hostname "f

--> Creating resources ...
    imagestream "file-uploader" created
    buildconfig "file-uploader" created
    deploymentconfig "file-uploader" created
```

```
    service "file-uploader" created
--> Success
    Build scheduled, use 'oc logs -f bc/file-uploader' to track its prog
    Run 'oc status' to view your app.
```

Watch and wait for the application to be deployed:

```
oc logs -f bc/file-uploader
```

You will see something like:

```
Cloning "https://github.com/christianh814/openshift-php-upload-demo" ...
        Commit: 7508da63d78b4abc8d03eac480ae930beec5d29d (Update index.h
        Author: Christian Hernandez <christianh814@users.noreply.github.
        Date:   Thu Mar 23 09:59:38 2017 -0700
---> Installing application source...
Pushing image 172.30.120.134:5000/my-shared-storage/file-uploader:latest
Pushed 0/5 layers, 2% complete
Pushed 1/5 layers, 20% complete
Pushed 2/5 layers, 40% complete
Push successful
```

You should `Ctrl` + `c` out of the tail mode once you see *Push successful*.

When the build is completed ensure the pods are running:

```
oc get pods
```

You will see something like:

```
NAME                              READY     STATUS      RESTARTS   AGE
file-uploader-1-build             0/1       Completed   0          2m
file-uploader-1-k2v0d             1/1       Running     0          1m
...
```

Note the name of the single pod currently running the app, in the example above `file-uploader-1-k2v0d`. The container called `file-uploader-1-build` is the builder container and is not relevant for us. A service has been created for our app but not exposed externally via a **Route** yet. Let's fix this:

```
oc expose svc/file-uploader
```

Check the **Route** that has been created:

```
oc get route
```

You will see something like:

```
NAME                     HOST/PORT
file-uploader            file-uploader-my-shared-storage.apps.1931832622
...
```

This use of the `new-app` command directly asked for application code to be built and did not involve a template. This is why a **Route** needs to be created by hand.

Point your browser the the URL advertised by the route (http://file-uploader-my-shared-storage.apps.193183262213.aws.testdrive.openshift.com)

The application simply lists all file previously uploaded and offers the ability to upload new ones as well as download the existing data. Right now there is nothing.

Select an arbitrary from your local system and upload it to the app.

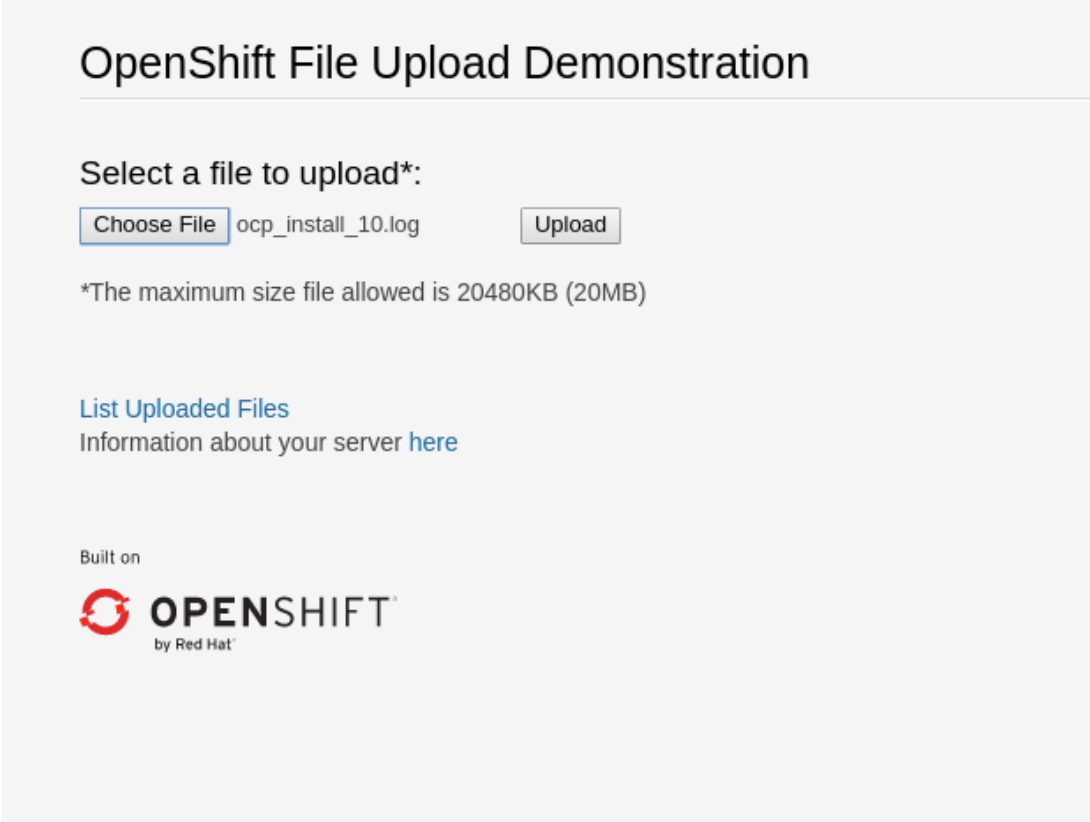

Figure 1. A simple PHP-based file upload tool

After uploading a file validate it has been stored locally in the container by following the link *List uploaded files* in the browser or logging into it via a remote session (using the name noted earlier):

```
oc rsh file-uploader-1-k2v0d
```

Once inside:

```
sh-4.2$ cd uploaded
sh-4.2$ pwd
/opt/app-root/src/uploaded
sh-4.2$ ls -lh
```

You should see something like:

```
total 16K
-rw-r--r--. 1 1000080000 root 16K May 26 09:32 cns-deploy-4.0.0-15.el7rh
```

The exact name of the **Pod** will be different in your environment.

The app should also list the file in the overview:

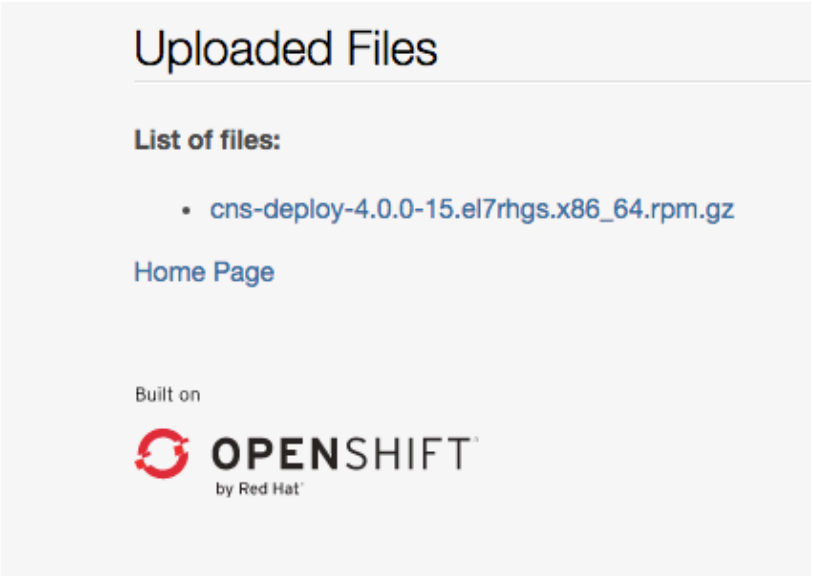

Figure 2. The file has been uploaded and can be downloaded again

This pod currently does not use any persistent storage. It stores the file locally.

Never attempt to store persistent data in a **Pod**. **Pods** and their containers are ephemeral by definition, and any stored data will be lost as soon as the **Pod** terminates.

Let’s see when this become a problem. Exit out of the container shell:

```
sh-4.2$ exit
```

Let’s scale the deployment to 3 instances of the app:

```
oc scale dc/file-uploader --replicas=3
```

Eventually, the additional **Pods** get spawned:

```
oc get pods
```

You will see something like:

```
NAME                          READY     STATUS      RESTARTS   AGE
file-uploader-1-3cgh1         1/1       Running     0          20s
file-uploader-1-3hckj         1/1       Running     0          20s
file-uploader-1-build         0/1       Completed   0          4m
file-uploader-1-k2v0d         1/1       Running     0          3m
...
```

The pod names will be different in your environment since they are automatically generated.

When you log on to one of the new instances (one of the ones with an age much smaller than the others) you will see they have no data:

```
oc rsh file-uploader-1-3cgh1
```

And then:

```
sh-4.2$ cd uploaded
sh-4.2$ pwd
/opt/app-root/src/uploaded
sh-4.2$ ls -hl
```

You'll notice that the file is gone:

```
total 0
```

Similarly, other users of the app will sometimes see your uploaded files and sometimes not - whenever the load balancing service in OpenShift points to the **Pod** that has the file stored locally. You can simulate this with another instance of your browser in "Incognito mode" pointing to your app.

The app is of course not usable like this. We can fix this by providing shared storage to this app.

You can create a **PersistentVolumeClaim** and attach it into an application with the `oc volume` command. Execute the following

```
oc volume dc/file-uploader --add --name=my-shared-storage \
-t pvc --claim-mode=ReadWriteMany --claim-size=5Gi \
--claim-name=my-shared-storage --mount-path=/opt/app-root/src/u
```

This command will:

- create a **PersistentVolumeClaim**
- update the **DeploymentConfig** to include a `volume` definition
- update the **DeploymentConfig** to attach a `volumemount` into the specified `mount-path`
- cause a new deployment of the application **Pods**

For more information on what `oc volume` is capable of, look at its help output with `oc volume -h`. Now, let's look at the result of adding the volume:

```
oc get pvc
```

You will see something like:

```
NAME                STATUS    VOLUME
my-shared-storage   Bound     pvc-62aa4dfe-4ad2-11e7-b56f-2cc2602a6dc8
...
```

Notice the `ACCESSMODE` being set to **RWX** (short for `ReadWriteMany`, equivalent to "shared storage"). Without this `ACCESSMODE`, OpenShift will not attempt to attach multiple **Pods** to the same **PersistentVolume**. If you attempt to scale up deployments that are using `ReadWriteOnce` storage, they will actually fail.

The app is now re-deploying (in a rolling fashion) with the new settings - all pods will mount the volume identified by the PVC under `/opt/app-root/src/upload`.

You can watch the redeployment like this:

```
oc logs dc/file-uploader -f
```

And you may see:

```
--> Scaling up file-uploader-2 from 0 to 3, scaling down file-uploader-1
    Scaling file-uploader-2 up to 1
    Scaling file-uploader-1 down to 2
    Scaling file-uploader-2 up to 2
    Scaling file-uploader-1 down to 1
    Scaling file-uploader-2 up to 3
    Scaling file-uploader-1 down to 0
--> Success
```

The logs for the **DeploymentConfig** will only be available / look like the above if you attempt to view them **during** the deployment. Once the deployment has completed, you will see something very different:

```
-> Cgroups memory limit is set, using HTTPD_MAX_REQUEST_WORKERS=68
AH00558: httpd: Could not reliably determine the server's fully qualified domain n
AH00558: httpd: Could not reliably determine the server's fully qualified domain n
[Sat Sep 02 20:15:14.848118 2017] [auth_digest:notice] [pid 1] AH01757: generating
[Sat Sep 02 20:15:14.852187 2017] [http2:warn] [pid 1] AH02951: mod_ssl does not s
[Sat Sep 02 20:15:14.853049 2017] [lbmethod_heartbeat:notice] [pid 1] AH02282: No
[Sat Sep 02 20:15:15.175812 2017] [mpm_prefork:notice] [pid 1] AH00163: Apache/2.4
[Sat Sep 02 20:15:15.175844 2017] [core:notice] [pid 1] AH00094: Command line: 'ht
```

Don’t worry. Nothing is wrong. There is actually a `deployer` **Pod** that handles the deployment, and it disappears once the deployment is complete. You are seeing the logs for this deployer in the above example (`Scaling up…`).

The new config `file-uploader-2` will have 3 pods all sharing the same storage.

```
oc get pods
```

You will see something like:

```
NAME                        READY     STATUS      RESTARTS   AGE
file-uploader-1-build       0/1       Completed   0          18m
file-uploader-2-jd22b       1/1       Running     0          1m
file-uploader-2-kw9lq       1/1       Running     0          2m
file-uploader-2-xbz24       1/1       Running     0          1m
...
```

Try it out in your application: upload new files and watch them being visible from within all application pods. In the browser the application behaves seamlessly as it circles through the pods between browser sessions (if you use an incognito session or a different browser).

```
oc rsh file-uploader-2-jd22b
sh-4.2$ ls -lh uploaded
total 16K
-rw-r--r--. 1 1000080000 root 16K May 26 10:21 cns-deploy-4.0.0-15.el7rh
sh-4.2$ exit
exit
oc rsh file-uploader-2-kw9lq
```

```
sh-4.2$ ls -lh uploaded
-rw-r--r--. 1 1000080000 root 16K May 26 10:21 cns-deploy-4.0.0-15.el7rh
sh-4.2$ exit
exit
oc rsh file-uploader-2-xbz24
sh-4.2$ ls -lh uploaded
-rw-r--r--. 1 1000080000 root 16K May 26 10:21 cns-deploy-4.0.0-15.el7rh
sh-4.2$ exit
```

That's it. You have successfully provided shared storage to pods throughout the entire system, therefore avoiding the need for data to be replicated at the application level to each pod.

With CNS this is available wherever OpenShift is deployed with no external dependency.

Increasing Storage Capacity in CNS

Once deployed there are two way in which to increase the storage capacity offered by CNS. Either by adding additional nodes with storage to OpenShift cluster or by adding additional storage devices to the existing nodes running CNS.

Adding nodes to CNS

The pre-requisite of adding nodes to the CNS setup is that these nodes have been added to the OpenShift cluster before. That is, increasing the storage capacity of CNS this way is a two-step process:

1. Extend the OpenShift cluster with additional nodes
2. Add the newly added nodes to the CNS setup

Fortunately both steps are easy thanks to automation. In the preceeding "Infrastructure Management Module" you have already added a second set of 3 nodes to the OpenShift cluster. + These have an additional storage device available, so we will use those.

For the second step, adding these new nodes to the CNS setup, you generally have two options:

A. add the new nodes to the existing CNS storage cluster, provisioned in the module "Deploying Container-native Storage"

B. add the new nodes to a new, independent CNS storage cluster, still managed by the single heketi API service

Option A is the straigt-forward choice when you just need more storage space. For this you can start with a single additional node. Use option B when you need a net-new, independent storage cluster for the sake of tenant isolation, different geographical region or exposing different storage tiers as separate clusters. For this, you need at least 3 new nodes. In this exercise we will implement Option B.

The following action require elevated privileges in OpenShift. Login as cluster admin and change to the CNS namespace:

```
oc login -u system:admin
oc project container-native-storage
```

First, identify the newly added nodes - the easiest way is to look at their uptime:

```
oc get nodes
```

You will see something like:

```
NAME                                STATUS                     AGE
node01.internal.aws.testdrive.openshift.com   Ready
node04.internal.aws.testdrive.openshift.com   Ready
master.internal.aws.testdrive.openshift.com   Ready,SchedulingDisabled
node02.internal.aws.testdrive.openshift.com   Ready
infra.internal.aws.testdrive.openshift.com   Ready
node05.internal.aws.testdrive.openshift.com   Ready
node03.internal.aws.testdrive.openshift.com   Ready
node06.internal.aws.testdrive.openshift.com     Ready
1
```

1 The nodes added in the previous lab

Now we need to make sure, that these new systems have the right firewall ports opened. For simplicity, we will just re-execute the `configure-firewall.yaml` from the "Deploying Container-native Storage" module against these new systems.

First uncomment the additional nodes entries already prepared in the ansible inventory file `/etc/ansible/hosts` (they are prefixed with `#addcns_`).
You need `sudo` privileges to do that:

```
sudo vim /etc/ansible/hosts
```

Remove the comment to enable the new host group like so:

/etc/ansible/hosts

```
[...]

[cns]
node01.193183262213.aws.testdrive.openshift.com
node02.193183262213.aws.testdrive.openshift.com
node03.193183262213.aws.testdrive.openshift.com
node04.193183262213.aws.testdrive.openshift.com
node05.193183262213.aws.testdrive.openshift.com
node06.193183262213.aws.testdrive.openshift.com

[...]
```

Then execute the `configure-firewall.yaml` playbook again:

```
ansible-playbook /opt/lab/support/configure-firewall.yaml
```

Next, add the following label to these nodes in order have the **DaemonSet** that CNS is based upon schedule new GlusterFS pods on them:

```
oc get daemonset
```

You will see something like:

```
NAME        DESIRED   CURRENT   READY     NODE-SELECTOR           AGE
glusterfs   3         3         3         storagenode=glusterfs   3h
```

1 The label definition the **DaemonSet** uses to select the nodes which run a GlusterFS pod.

```
oc label node/node04.internal.aws.testdrive.openshift.com storagenode=gl
oc label node/node05.internal.aws.testdrive.openshift.com storagenode=gl
oc label node/node06.internal.aws.testdrive.openshift.com storagenode=gl
```

The **DaemonSet** will detect that new nodes have these labels, and GlusterFS **Pods** will be launched on the newly labeled nodes. Wait for these **Pods** to be in `Ready` state:

```
oc get pods -o wide
```

You will see something like:

```
NAME              READY     STATUS    RESTARTS   AGE       IP
glusterfs-3gjc5   1/1       Running   0          1m        10.0.4.161
glusterfs-37vn8   1/1       Running   0          3h        10.0.1.6
glusterfs-ng00k   1/1       Running   0          1m        10.0.1.182
glusterfs-cq68l   1/1       Running   0          3m        10.0.3.137
glusterfs-zkvfl   1/1       Running   0          1m        10.0.3.132
glusterfs-m9fvl   1/1       Running   0          3m        10.0.4.68
heketi-1-cd032    1/1       Running   0          1m        10.0.4.68
```

1 The newly spawned GlusterFS pods.

It may take up to 120 seconds for the GlusterFS **Pods** to be up and in *Ready* state.

The new **Pods** run GlusterFS uninitialized. That is, they have not formed a cluster among themselves yet. This is triggered via heketi.

heketi initializes vanilla GlusterFS **Pods** as part of loading the topology file. Like during the cns-deploy phase in the "Deploying Container-native Storage" module it can read an additional cluster structure from the JSON file. This has already been prepared suitable for your environment in the `/opt/lab/support/topology-extended.json`. It contains the original 3 nodes we started with, and then newly added nodes.

Initialize the heketi-cli with environment variables like so:

```
export HEKETI_CLI_SERVER=http://heketi-container-native-storage.apps.193
export HEKETI_CLI_USER=admin
export HEKETI_CLI_KEY=myS3cr3tpassw0rd
```

This avoids repetitive command switches with heketi-cli. Use the heketi client to load the new topology. Make sure you are currently in `/opt/lab/support`:

```
heketi-cli topology load --json=/opt/lab/support/topology-extended.json
```

And you will see something like:

```
        Found node node01.internal.aws.testdrive.openshift.com on cluste
                Found device /dev/xvdd
```

```
        Found node node02.internal.aws.testdrive.openshift.com on cluste
                Found device /dev/xvdd
        Found node node03.internal.aws.testdrive.openshift.com on cluste
                Found device /dev/xvdd
        Creating node node04.internal.aws.testdrive.openshift.com ... ID
                Adding device /dev/xvdd ... OK
        Creating node node05.internal.aws.testdrive.openshift.com ... ID
                Adding device /dev/xvdd ... OK
        Creating node node06.internal.aws.testdrive.openshift.com ... ID
                Adding device /dev/xvdd ... OK
```

With this you've successfully initialized a second CNS storage cluster that is managed by heketi. You can query heketi for the new topology:

```
heketi-cli topology info
```

You will see something like:

```
Cluster Id: ca777ae0285ef6d8cd7237c862bd591c (1)

    Volumes:

    Nodes:

        Node Id: caaed3927e424b22b1a89d261f7617ad
        State: online
        Cluster Id: ca777ae0285ef6d8cd7237c862bd591c
        Zone: 3
        Management Hostname: node06.internal.aws.testdrive.openshift.com
        Storage Hostname: node06.internal.aws.testdrive.openshift.com
        Devices:
                Id:b65fee8350c2b4cad4fd68535aba05b7   Name:/dev/xvdd
```

```
                        Bricks:

          Node Id: 33e0045354db4be29b18728cbe817605
          State: online
          Cluster Id: ca777ae0285ef6d8cd7237c862bd591c
          Zone: 1
          Management Hostname: node04.internal.aws.testdrive.openshift.com
          Storage Hostname: 10.0.1.182
          Devices:
                  Id:b75d8e52e6978675d599111d50e46969   Name:/dev/xvdd
                          Bricks:

          Node Id: d8443e7ee8314c0c9fb4d8274a370bbd
          State: online
          Cluster Id: ca777ae0285ef6d8cd7237c862bd591c
          Zone: 2
          Management Hostname: node05.internal.aws.testdrive.openshift.com
          Storage Hostname: 10.0.3.132
          Devices:
                  Id:4330fb2333c5dfb9add3e3ea00ec82a6   Name:/dev/xvdd
                          Bricks:

      Cluster Id: ec7a9c8be8327a54839236791bf7ba24

          Volumes:
...
```

❶ The internal ID of the new cluster managed by heketi

The cluster ID will be different for you since it's automatically generated.

To use this cluster specifically, you can create a separate **StorageClass** for it in OpenShift. PVCs issued against it, will only be served from this particular CNS

storage cluster. For this purpose, note it's internal heketi ID - in the example above **ca777ae0285ef6d8cd7237c862bd591c**.

There is a file on your system `/opt/lab/support/second-cns-storageclass.yaml`. Open it with your favorite editor:

/opt/lab/support/second-cns-storageclass.yaml

```
apiVersion: storage.k8s.io/v1beta1
kind: StorageClass
metadata:
  name: cns-silver
provisioner: kubernetes.io/glusterfs
parameters:
  resturl: "http://heketi-container-native-storage.apps.1931832(
  restauthenabled: "true"
  restuser: "admin"
  volumetype: "replicate:3"
  clusterid: "INSERT-CLUSTER-ID-HERE" (1)
  secretNamespace: "default"
  secretName: "cns-secret"
```

(1) The heketi internal ID of the new cluster is used to specifically direct requests to it. **Replace it with the ID of your cluster!**

After you have correctly replaced your new cluster ID, create the `StorageClass`:

```
oc create -f /opt/lab/support/second-cns-storageclass.yaml
```

There is a **PersistentVolumeClaim** definition file that already has been placed in `/opt/lab/support` for you:

/opt/lab/support/cns-silver-pvc.yaml

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-container-storage-silver
  annotations:
    volume.beta.kubernetes.io/storage-class: cns-silver
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

You can create it with the following:

```
oc create -f /opt/lab/support/cns-silver-pvc.yaml
```

This PVC will now be fulfilled by the *cns-silver* **StorageClass** which specifically directs the requests to the second cluster specified by its UUID in the `clusterid` parameter of the **StorageClass**.

You can see that this claim is automatically bound:

```
oc get pvc
```

You will see something like:

```
NAME                          STATUS    VOLUME
my-container-storage-silver   Bound     pvc-5efde23a-901e-11e7-bebd-12ea
```

Other CNS Maintenance Activities

In addition to extending your CNS cluster with additional storage nodes, you may also want to perform other maintenance activities. For example, if you have added more block devices to one of your CNS nodes, you may simply wish to add additional devices to the cluster. Or, if you have degraded physical devices that need to be replaced, maintained, or eliminated, you may wish to remove devices from a cluster.

Adding Additional Devices to a CNS Cluster

Instead of adding a net-new cluster you can also add additional devices to an existing cluster. The process is very similar to adding new nodes - loading a modified topology JSON file via the heketi client.

To illustrate an alternative we are going to use `heketi-cli` tool directly.

The nodes of the second cluster, have an additional, unused storage device `/dev/xvde`. To add them we need to know their node IDs. + With the environment variables for `heketi-cli` still set run:

```
heketi-cli node list | grep ca777ae0285ef6d8cd7237c862bd591c
Id:33e0045354db4be29b18728cbe817605     Cluster:ca777ae0285ef6d8cd7237c8
Id:d8443e7ee8314c0c9fb4d8274a370bbd     Cluster:ca777ae0285ef6d8cd7237c8
Id:caaed3927e424b22b1a89d261f7617ad     Cluster:ca777ae0285ef6d8cd7237c8
```

You will need to replace the `grep` with your unique cluster ID. This is the cluster ID of the second / new CNS cluster that you just created previously, and used when creating the new `cns-silver` **StorageClass**.

For each node in the output (eg: `33e0045354db4be29b18728cbe817605`, `d8443e7ee8314c0c9fb4d8274a370bbd`, and `caaed3927e424b22b1a89d261f7617ad`), go ahhead and `device add` the additional block device:

```
heketi-cli device add --name=/dev/xvde --node=33e0045354db4be29b18728cbe
Device added successfully

heketi-cli device add --name=/dev/xvde --node=d8443e7ee8314c0c9fb4d8274a
Device added successfully

heketi-cli device add --name=/dev/xvde --node=caaed3927e424b22b1a89d261f
Device added successfully
```

The node UUIDs will be different for you since they are automatically generated.

You can now verify the presence of these new devices by running:

```
heketi-cli topology info
```

You should see a `/dev/xvde` device present for each of the nodes in the `cns-silver` cluster.

Replacing Failed Disks and Nodes

Despite CNS' capability to continue operating transparently to the client in the face of failing disks and nodes, you soon might want to replace such components to move out of a degraded state.

For this exercise, let's assume the device `/dev/xvdd` of your node node04.internal.aws.testdrive.openshift.com failed and you need to replace it. You can do that as long as there is enough spare capacity somewhere else in the cluster, preferrable but not necessarily in the same failure domain (as specifed in the topology).

The first step is to, again, determine the CNS node's internal UUID in heketi's database:

```
heketi-cli topology info | grep -B4 node04.internal.aws.testdrive.opensh
```

You will see something like:

```
        Node Id: 33e0045354db4be29b18728cbe817605
        State: online
        Cluster Id: ca777ae0285ef6d8cd7237c862bd591c
        Zone: 1
        Management Hostname: node04.internal.aws.testdrive.openshift.com
```

Second, determine the device's UUID by querying the node (indicated above by `Node Id`):

```
heketi-cli node info 33e0045354db4be29b18728cbe817605
```

You will see something like:

```
Node Id: 33e0045354db4be29b18728cbe817605
State: online
Cluster Id: ca777ae0285ef6d8cd7237c862bd591c
Zone: 1
Management Hostname: node04.internal.aws.testdrive.openshift.com
Storage Hostname: 10.0.1.182
Devices:
Id:01c94798bf6b1af87974573b420c4dff   Name:/dev/xvdd           State:onl
Id:da91a2f1c9f62d9916831de18cc09952   Name:/dev/xvde           State:onl
```

Notice the UUID of the device `/dev/xvdd` as shown.

The device ID, as well as all other UUIDs in heketi commands are automatically generated and different in your environment. Please be aware when copy & pasting.

Third, mark the device as offline to stop heketi from further attempts to allocate space from it:

```
heketi-cli device disable 01c94798bf6b1af87974573b420c4dff
```

You will see something like:

```
Device 01c94798bf6b1af87974573b420c4dff is now offline
```

The device is now offline but it's still part of replicated volumes. To remove it and trigger a self-healing operation in the background issue:

```
heketi-cli device remove 01c94798bf6b1af87974573b420c4dff
```

You will see something like:

```
Device 01c94798bf6b1af87974573b420c4dff is now removed
```

This command can take a bit longer as it will go through the topology and search for the next available device on the same node, in the same failure domain and in the rest of the cluster (in that order) and trigger a brick-replacement operation. This way data is re-replicated to another health storage device and the 3-way replicated storage volume moves out of degraded state.

The device is still lurking around in *failed* state. To finally get rid of it issue:

```
heketi-cli device delete 01c94798bf6b1af87974573b420c4dff
```

You will see something like:

```
Device 01c94798bf6b1af87974573b420c4dff deleted
```

Only devices that are not used by other Gluster volumes can be deleted. If that's not the case `heketi-cli` will tell you about it. In this case you need to issue a `remove` operation before.

You can now check that the device is gone from the topology by running:

```
heketi-cli topology info
```

Node deletion is also possible and is basically comprised of:

1. successful execution of the `remove` operation on all devices of the node
2. running `heketi-cli node delete <node_id>` on the node in question

Running the OpenShift Registry with CNS

The Registry in OpenShift is a critical component. As it is the default destination for all container builds in the cluster, and is the source for deploying applications built inside the cluster, being unavailable is a big problem.

The internal registry runs as one or more **Pods** inside the OpenShift environment. By default the registry uses local ephemeral storage in its **Pod**. This means that any restarts or re-deployments or outages would cause all of the built/pushed container images to be lost. Also, only having one registry instance

and/or one infrastructure node could cause temporary outages. So, adding storage and scaling up the registry is a good idea.

Your cluster only has one infrastructure node. In practice, you would want a minimum of two to achieve high-availability for all infrastructure services.

Adding CNS to the Registry

Adding storage to the registry is as easy as it was for our file-uploader application. Simply make the registry **Pods** use a PVC in access mode **RWX** based on CNS. This way, a highly-available scale-out registry can be provided without external dependencies on NFS or Cloud Provider storage.

The following method will be disruptive. All data stored in the registry so far will be lost (the Rails and PHP app images). Migration scenarios exist but are beyond the scope of this lab, but normally you would configure persistent storage for the registry before starting to really use your cluster.

Make sure you are logged in as `system:admin` in the `default` namespace:

```
oc login -u system:admin -n default
```

Just like with the file uploader example, you can simply add a volume (and have its **PersistentVolumeClaim** created automatically) with the `oc volume` command. Execute the following:

```
oc volume dc/docker-registry --add --name=registry-storage -t pvc \
--claim-mode=ReadWriteMany --claim-size=10Gi \
```

```
--claim-name=registry-storage --overwrite
```

The registry will now redeploy.

The registry is preconfigured with a volume called `registry-storage` that is using the `emptyDir` storage type. The above command `--overwrite` the existing volume with our new PVC. More information can be found in the volumes documentation.

In a future release of OpenShift, you will be able to configure Container Native Storage as part of the OpenShift installation directly, including automatically using CNS for the storage for the registry, fully supported.

Observe the registry deployment get updated:

```
oc get pod -w
```

Remember to `Ctrl` + `C` when you are done watching the **Pods** redeploy.

After a couple of seconds a new deployment of the registry should be available. Verify a new version of the registry's **DeploymentConfig** is running:

```
oc get dc/docker-registry
```

You will see something like:

```
NAME              REVISION   DESIRED   CURRENT   TRIGGERED BY
docker-registry   2          1         1         config
```

Now your OpenShift Registry is using persistent storage provided by CNS. Since this is shared storage this also allows you to scale out the registry pods.

You can scale the registry like this:

```
oc scale dc/docker-registry --replicas=3
```

After a short while you should see 3 healthy registry pods in the default **Project**:

```
oc get pods
```

And you should see something like:

```
NAME                       READY     STATUS    RESTARTS   AGE
docker-registry-2-5rszg    1/1       Running   0          1m
docker-registry-2-7s3tm    1/1       Running   0          14s
docker-registry-2-g3l70    1/1       Running   0          14s
registry-console-1-b47jt   1/1       Running   0          6h
router-1-hs9wp             1/1       Running   0          6h
```

Go to previous module | Go to next module