+++

1) After cluster deployment we tried to deploy php and Jenkins application

It failed because of images not available to cluster

So we have added images to open shift clusert(process yestarday screen shot)

We have added php and Jenkins image to open shift cluster then we are able to deploy

Jenkins also we are able to deploy only jenkins emphermal

But we are not able to deploy jenkins which has volume and I don’t see where we have to provide PVname when doing via GUI

2)After deploying application the route has to create a default with wild card name but it not taking that.

3)We have integrated open shift cluster with LDAP

How can we verify all the users is LDAP exist in Openshift

OpenShift Container## Environment Overview

You will be interacting with an OpenShift 3.11 cluster that is running on Amazon Web Services. During the lab you will also install OpenShift Container Storage 3.11. The complete environment consists of the following systems:

* 1 master node
* 1 infrastructure node
* 6 "application" nodes
  + 3 will run workload and the initial Container Native Storage instances
  + 3 will be added to the cluster later
* 1 server running Red Hat Identity Management (IdM, for LDAP authentication)

| *Table 1. Lab Environment Overview* | |
| --- | --- |
| **Role** | **Internal FQDN** |
| Master Node | master.internal.aws.testdrive.openshift.com |
| Infrastructure Node | infra.internal.aws.testdrive.openshift.com |
| Application Node #1 | node01.internal.aws.testdrive.openshift.com |
| Application Node #2 | node02.internal.aws.testdrive.openshift.com |
| Application Node #3 | node03.internal.aws.testdrive.openshift.com |
| Application Node #4 | node04.internal.aws.testdrive.openshift.com |
| Application Node #5 | node05.internal.aws.testdrive.openshift.com |
| Application Node #6 | node06.internal.aws.testdrive.openshift.com |
| IdM Server | idm.internal.aws.testdrive.openshift.com |

All addresses are internal to the lab environment. The only system you publicly access via SSH and the browser is the OpenShift Master node:

| *Table 2. Public Lab Access* | |
| --- | --- |
| **Role** | **Public FQDN** |
| Master Node | master.004325440631.aws.testdrive.openshift.com |

You will be installing OpenShift Container Platform v3.11 using the advanced installation method, which involves executing various Ansible playbooks. You will also install Container Native Storage v3.11.

Note that references to product documentation will be specifically pointing to the 3.11 versions, but newer software and documentation versions may be available.

How to get support

In case of technical or functional problems please reach out to [openshift-ops-testdrive@redhat.com](mailto:openshift-ops-testdrive@redhat.com). This is a ticket based system where you will be able to reach the authors and supporters of this Test Drive Experience.

Conventions

You will see various code and command blocks throughout these exercises. Some of the command blocks can be copy/pasted directly. Others will require modification of the command before execution. If you see a command block with a red border (see below), the command will require slight modification.

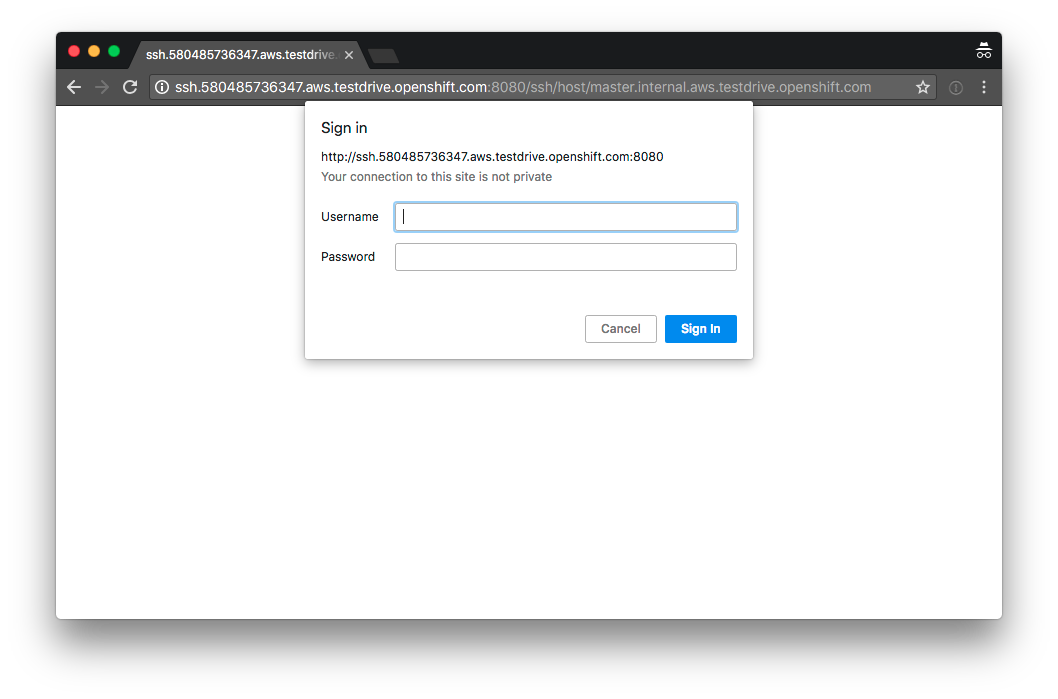
some command to modify

Most command blocks support auto highlighting with a click. If you hover over the command block above and click, it should automatically highlight all the text to make for easier copying.

Logging in

Most of the exercises in this lab will be facilitated using the OpenShift command line client on the master node. For convenient access to the master’s command line we provide a web-based SSH console: <http://ssh.004325440631.aws.testdrive.openshift.com:8080/ssh/host/master.internal.aws.testdrive.openshift.com>

Use the user name cloud-user and the password qwikLABS-L64-28668 when prompted.



*Figure 1. SSH Console Login*

If you prefer to use an SSH client on your system you can do that too, using the same credentials:

ssh -l cloud-user master.004325440631.aws.testdrive.openshift.com

Once you are logged in you end up on the OpenShift Master Node:

[cloud-user@master ~]$

The cloud-user account has password-less sudo privileges and SSH login on all systems using internal addressing from the table above.

Installation and Verification

The primary method of installing OpenShift Container Platform is based on Ansible playbooks. These playbooks ship as part of the product in the openshift-ansible package.

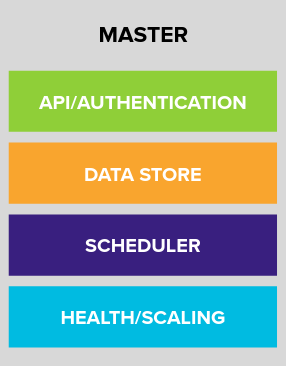
This method has, in the past, been referred to as the advanced installation method and it involves Ansible directly running the installation playbooks. The advanced installer supports many configuration and customization options. It also covers installation of supporting infrastructure like OpenShift Container storage, logging and metrics components.

Your environment comes with a preinstalled cluster that has been deployed using the installer’s configuration file (/etc/ansible/hosts) when you started the lab.

For more information on installing OpenShift Container Platform, please refer to the [installation section](https://docs.openshift.com/container-platform/3.11/install/index.html)of the product documentation.

|  |  |
| --- | --- |
|  | At this point you should be logged in as cloud-user on the OpenShift Master node via SSH. |

Master Components



*Figure 1. OpenShift Master’s 4 main responsibilities.*

API/Authentication

The Kubernetes API server validates and configures the resources that make up a Kubernetes cluster.

Common things that interact with the Kubernetes API server are:

* OpenShift Web Console
* OpenShift oc command line tool
* OpenShift Node
* Kubernetes Controllers

All interactions with the API server are secured using TLS. In addition, all API calls must be authenticated (the user is who they say they are) and authorized (the user has rights to make the requested API calls).

Data Store

The OpenShift Data Store (etcd) stores the persistent master state while other components watch etcd for changes to bring themselves into the desired state. etcd can be optionally configured for high availability, typically deployed with 2n+1 peer services.

|  |  |
| --- | --- |
|  | etcd stores the cluster’s state. It is not used to store user application data. |

Scheduler

The pod scheduler is responsible for determining placement of new pods onto nodes within the cluster.

The scheduler is very flexible and can take the physical topology of the cluster into account (racks, datacenters, etc).

Health / Scaling

Each pod can register both liveness and readiness probes.

Liveness probes tell the system if the pod is healthy or not. If the pod is not healthy, it can be restarted automatically.

Readiness probes tell the system when the pod is ready to take traffic. This, for example, can be used by the cluster to know when to put a pod into the load balancer.

For more information on the OpenShift Master’s areas of responsibility, please refer to the [infrastructure components section](https://docs.openshift.com/container-platform/3.11/architecture/infrastructure_components/kubernetes_infrastructure.html) of the product documentation.

Examining the provided Ansible inventory file

First, let’s examine the provided installer configuration file.

Look at the file

Use cat, less, or an editor to look at the /etc/ansible/hosts file:

cat /etc/ansible/hosts

General settings and other variable information is defined on lines within the [OSEv3:vars] section. There are also various host groups defined for things like **Masters** and **Nodes**.

For more details on how OpenShift uses Ansible for its installation, please refer to the [Configuring Your Inventory File](https://docs.openshift.com/container-platform/3.11/install/configuring_inventory_file.html).

The top-level playbook in/usr/share/ansible/openshift-ansible/playbooks/deploy\_cluster.yml triggers the installation of the cluster and all of it’s components. It’s idempotent, which means you can execute this playbook multiple times without harm. This also allows you to deploy additional components after the initial install by simply modifying the configuration in /etc/ansible/hosts and re-run the installer.

In addition to this, there are playbooks that only deploy a specific component or service, which makes them faster to execute.

|  |  |
| --- | --- |
|  | A typical multi-host installation like this might normally take around an hour depending on the speed of your internet connection. Disconnected installation options are also available. Prerequisites and other information is all covered in the documentation. |

Verifying the Installation

Let’s do some basic tests with your installation. As an administrator, most of your interaction with OpenShift will be from the command line. The oc program is a command line interface that talks to the OpenShift API.

During the OpenShift installation, the root system account on the master host is configured to use a special OpenShift "super administrator" account. Because of this, it is vitally important that you protect access to the root system account, or remove this preconfigured config. Otherwise, anyone who can sudo on the master has super user privileges on the entire cluster.

Login on the master

Additionally, your Linux system account on the master, cloud-user, is preconfigured to access this OpenShift "super administrator" without a password. Type the following command to login as the internal super-user on OpenShift:

oc login -u system:admin

You will see that you got logged in to a project called 'default'. More on projects later.

Logged into "https://master.internal.aws.testdrive.openshift.com:443" as "system:admin" using existing credentials.

You have access to the following projects and can switch between them with 'oc project <projectname>':

\* default

kube-public

kube-system

management-infra

openshift

openshift-console

openshift-infra

openshift-logging

openshift-metrics

openshift-monitoring

openshift-node

openshift-sdn

openshift-web-console

storage

Using project "default".

Look at the Nodes

Execute the following command to see a list of the **Nodes** that OpenShift knows about:

oc get nodes

The output should look something like the following:

NAME STATUS ROLES AGE VERSION

infra.internal.aws.testdrive.openshift.com Ready infra 1m v1.11.0+d4cacc0

master.internal.aws.testdrive.openshift.com Ready master 1m v1.11.0+d4cacc0

node01.internal.aws.testdrive.openshift.com Ready compute 1m v1.11.0+d4cacc0

node02.internal.aws.testdrive.openshift.com Ready compute 1m v1.11.0+d4cacc0

node03.internal.aws.testdrive.openshift.com Ready compute 1m v1.11.0+d4cacc0

All of the systems listed in the [nodes] group in the /etc/ansible/hosts file should be listed here. 1 Infrastructure Node, 1 Master and 3 Worker nodes.

The OpenShift **Master** is also a **Node** because it needs to participate in the software defined network (SDN). The **Infra** node will only run workloads related to supporting OpenShift infrastructure.

Check the Web Console

OpenShift provides a web console for users, developers and application operators to interact with the environment. There aren’t many cluster administrative functions to perform through the web console. Some OpenShift components (like the internal image registry) run on top of the OpenShift environment, and you can see these things. However, we have not yet explored authentication topics, so you have no cluster administrator "human" accounts yet.

Point your browser to <https://openshift.004325440631.aws.testdrive.openshift.com/console> to verify that the web console is available and responding. You can login using the user teamuser1 with password openshift. You are not required to do anything in the web console at this point.

|  |  |
| --- | --- |
|  | You will receive a self-signed certificate error in your browser. When OpenShift is installed, by default, a CA and SSL certificates are generated for all inter-component communication within OpenShift, including the web console. It is possible to provide your own SSL certificates during the installation, and more information can be found in the [custom certificates](https://docs.openshift.com/container-platform/3.11/install_config/certificate_customization.html#ansible-configuring-custom-certificates) section of the installation documentation. |

Verify the Storage cluster

In your environment Red Hat OpenShift Container Storage was installed as part of OpenShift. It will serve robust and persistent storage to both business applications as well as OpenShift infrastructure. It is based on Red Hat Gluster Storage, running in containers on OpenShift nodes and an additional API server called heketi that enables the API integration with OpenShift.

We will now use a command line client on the **master** to talk via this server to the container storage cluster. It’s password protected, so let’s export a couple of environment variables first to configure the client:

export HEKETI\_CLI\_SERVER=http://heketi-storage-storage.apps.004325440631.aws.testdrive.openshift.com

export HEKETI\_CLI\_USER=admin

export HEKETI\_CLI\_KEY=myS3cr3tpassw0rd

Then use the CLI tool heketi-cli to query heketi about all the storage clusters it knows about:

heketi-cli cluster list

heketi will list all known clusters with internal UUIDs:

Clusters:

ec7a9c8be8327a54839236791bf7ba24 [file][block]

|  |  |
| --- | --- |
|  | This is the internal UUID of the OCS cluster |
|  | | The cluster UUID will be different for you since it’s automatically generated. |

To get more detailed information about the topology of your OCS cluster (i.e. nodes, devices and volumes heketi has discovered) run the following command (output abbreviated):

heketi-cli topology info

You will get a lengthy output that describes the GlusterFS cluster topology as it is known by heketi:

Cluster Id: ec7a9c8be8327a54839236791bf7ba24

File: true

Block: true

Volumes

Name: heketidbstorage

Size: 2

Id: 272c8d37828c62c4002a19027abd2feb

Cluster Id: ec7a9c8be8327a54839236791bf7ba24

Mount: 10.0.1.114:heketidbstorage

Mount Options: backup-volfile-servers=10.0.3.106,10.0.3.106

Durability Type: replicate

Replica: 3

Snapshot: Disabled

Nodes:

Node Id: 099b016da11a623bef37de9b85aaa2d7

State: online

Cluster Id: ec7a9c8be8327a54839236791bf7ba24

Zone: 3

Management Hostname: node03.internal.aws.testdrive.openshift.com

Storage Hostname: node03.internal.aws.testdrive.openshift.com

Devices:

Id:e64fac664861c14bd75e3116f805b8fc Name:/dev/xvdd State:online Size (GiB):49 Used (GiB):0 Free (GiB):49

Bricks:

[...]

Node Id: 43336d05323e6003be6740dbb7477bd6

State: online

Cluster Id: ec7a9c8be8327a54839236791bf7ba24

Zone: 1

Management Hostname: node01.internal.aws.testdrive.openshift.com

Storage Hostname: 10.0.1.114

Devices:

Id:11a148d8065f6a6220f89c2912d00d13 Name:/dev/xvdd State:online Size (GiB):49 Used (GiB):0 Free (GiB):49

Bricks:

[...]

Node Id: 6c738028f642e37b2368eca88f8c626c

State: online

Cluster Id: ec7a9c8be8327a54839236791bf7ba24

Zone: 2

Management Hostname: node02.internal.aws.testdrive.openshift.com

Storage Hostname: 10.0.3.106

Devices:

Id:cf7c0dfb258f07be25ac9cd4c4d2e6ae Name:/dev/xvdd State:online Size (GiB):49 Used (GiB):0 Free (GiB):49

Bricks:

[...]

|  |  |
| --- | --- |
|  | An internal GlusterFS volume that is automatically generated by the setup routine to hold the heketi database. |

This output tells you that Red Hat OpenShift Container Storage currently consists of a single cluster, which consists of 3 nodes, each with a single block device /dev/xvdd of 50GiB in size. The GlusterFS layer will turn these 3 devices/hosts into a single, flat storage pool from which OpenShift will be able to carve out either distinct filesystem volumes or block devices that serve as persistent storage for containers.

Application Management Basics

In this module, you will deploy a sample application using the oc tool and learn about some of the core concepts, fundamental objects, and basics of application management on OpenShift Container Platform.

|  |  |
| --- | --- |
|  | You will want to have an SSH session opened to the master server for these lab exercises. |

Core OpenShift Concepts

As a future administrator of OpenShift, it is important to understand several core building blocks as it relates to applications. Understanding these building blocks will help you better see the big picture of application management on the platform.

Projects

A **Project** is a "bucket" of sorts. It’s a meta construct where all of a user’s resources live. From an administrative perspective, each **Project** can be thought of like a tenant. **Projects** may have multiple users who can access them, and users may be able to access multiple **Projects**.

For this exercise, first create a **Project** to hold our resources:

oc new-project app-management

Deploy a Sample Application

The new-app command provides a very simple way to tell OpenShift to run things. You simply provide it with one of a wide array of inputs, and it figures out what to do. Users will commonly use this command to get OpenShift to launch existing images, to create builds of source code and ultimately deploy them, to instantiate templates, and so on.

You will now launch a specific image that exists on Dockerhub

oc new-app docker.io/siamaksade/mapit

The output will look like:

--> Found Docker image 9eca6ec (11 days old) from docker.io for "docker.io/siamaksade/mapit"

\* An image stream will be created as "mapit:latest" that will track this image

\* This image will be deployed in deployment config "mapit"

\* Ports 8080/tcp, 8778/tcp, 9779/tcp will be load balanced by service "mapit"

\* Other containers can access this service through the hostname "mapit"

--> Creating resources ...

imagestream "mapit" created

deploymentconfig "mapit" created

service "mapit" created

--> Success

Application is not exposed. You can expose services to the outside world by executing one or more of the commands below:

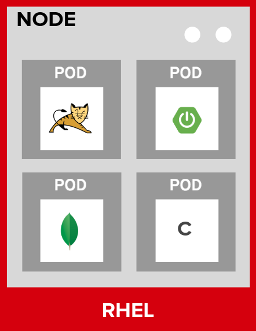
'oc expose svc/mapit'

Run 'oc status' to view your app.

You can see that OpenShift automatically created several resources as the output of this command. We will take some time to explore the resources that were created.

For more information on the capabilities of new-app, take a look at its help message by running oc new-app -h.

Pods



*Figure 1. OpenShift Pods*

Pods are one or more containers deployed together on host. A pod is the smallest compute unit you can define, deploy and manage. Each pod is allocated its own internal IP address on the SDN and will own the entire port range. The containers within pods can share local storage space and networking resources.

Pods are treated as **static** objects by OpenShift, i.e., one cannot change the pod definition while running.

You can get a list of pods:

oc get pods

And you will see something like the following:

NAME READY STATUS RESTARTS AGE

mapit-1-6lczv 1/1 Running 0 3m

|  |  |
| --- | --- |
|  | Pod names are dynamically generated as part of the deployment process, which you will learn about shortly. Your name will be slightly different. |

The describe command will give you more information on the details of a pod. In the case of the pod name above:

oc describe pod mapit-1-6lczv

And you will see output similar to the following:

Name: mapit-1-r2sjj

Namespace: app-management

Priority: 0

PriorityClassName: <none>

Node: node02.internal.aws.testdrive.openshift.com/10.0.3.98

Start Time: Wed, 24 Oct 2018 17:17:38 +0000

Labels: app=mapit

deployment=mapit-1

deploymentconfig=mapit

Annotations: openshift.io/deployment-config.latest-version=1

openshift.io/deployment-config.name=mapit

openshift.io/deployment.name=mapit-1

openshift.io/generated-by=OpenShiftNewApp

openshift.io/scc=restricted

Status: Running

IP: 10.129.0.2

Controlled By: ReplicationController/mapit-1

Containers:

mapit:

Container ID: docker://b00a80b47411665930b5f2f08b6df76d49f40869a599500af99d073aac730654

Image: docker.io/siamaksade/mapit@sha256:338a3031df6354e3adc3ba7d559ae22a0f2c79eade68aa72447f821cc7b8370c

Image ID: docker-pullable://docker.io/siamaksade/mapit@sha256:338a3031df6354e3adc3ba7d559ae22a0f2c79eade68aa72447f821cc7b8370c

Ports: 8080/TCP, 8778/TCP, 9779/TCP

Host Ports: 0/TCP, 0/TCP, 0/TCP

State: Running

Started: Wed, 24 Oct 2018 17:17:42 +0000

Ready: True

Restart Count: 0

Environment: <none>

Mounts:

/var/run/secrets/kubernetes.io/serviceaccount from default-token-nh85v (ro)

Conditions:

Type Status

Initialized True

Ready True

ContainersReady True

PodScheduled True

Volumes:

default-token-nh85v:

Type: Secret (a volume populated by a Secret)

SecretName: default-token-nh85v

Optional: false

QoS Class: BestEffort

Node-Selectors: node-role.kubernetes.io/compute=true

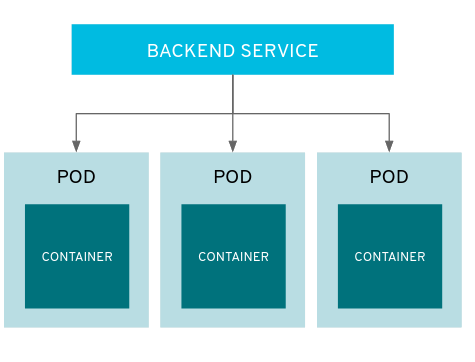
Tolerations: <none>

Events:

...

This is a more detailed description of the pod that is running. You can see what node the pod is running on, the internal IP address of the pod, various labels, and other information about what is going on.

Services



*Figure 2. OpenShift Service*

**Services** provide a convenient abstraction layer inside OpenShift to find a group of like **Pods**. They also act as an internal proxy/load balancer between those **Pods** and anything else that needs to access them from inside the OpenShift environment. For example, if you needed more mapit instances to handle the load, you could spin up more **Pods**. OpenShift automatically maps them as endpoints to the **Service**, and the incoming requests would not notice anything different except that the **Service** was now doing a better job handling the requests.

When you asked OpenShift to run the image, it automatically created a **Service** for you. Remember that services are an internal construct. They are not available to the "outside world", or anything that is outside the OpenShift environment. That’s OK, as you will learn later.

The way that a **Service** maps to a set of **Pods** is via a system of **Labels** and **Selectors**. **Services** are assigned a fixed IP address and many ports and protocols can be mapped.

There is a lot more information about [Services](https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/pods_and_services.html#services), including the YAML format to make one by hand, in the official documentation.

The new-app command used earlier caused a service to be created. You can see the current list of services in a project with:

oc get services

You will see something like the following:

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

mapit ClusterIP 172.30.48.204 <none> 8080/TCP,8778/TCP,9779/TCP 3m

|  |  |
| --- | --- |
|  | Service IP addresses are dynamically assigned on creation and are immutable. The IP of a service will never change, and the IP is reserved until the service is deleted. Your service IP will likely be different. |

Just like with pods, you can describe services, too. In fact, you can describe most objects in OpenShift:

oc describe service mapit

You will see something like the following:

Name: mapit

Namespace: app-management

Labels: app=mapit

Annotations: openshift.io/generated-by=OpenShiftNewApp

Selector: app=mapit,deploymentconfig=mapit

Type: ClusterIP

IP: 172.30.48.204

Port: 8080-tcp 8080/TCP

TargetPort: 8080/TCP

Endpoints: 10.129.0.2:8080

Port: 8778-tcp 8778/TCP

TargetPort: 8778/TCP

Endpoints: 10.129.0.2:8778

Port: 9779-tcp 9779/TCP

TargetPort: 9779/TCP

Endpoints: 10.129.0.2:9779

Session Affinity: None

Events: <none>

Information about all objects (their definition, their state, and so forth) is stored in the etcd datastore. etcd stores data as key/value pairs, and all of this data can be represented as serializable data objects (JSON, YAML).

Take a look at the YAML output for the service:

oc get service mapit -o yaml

You will see something like the following:

apiVersion: v1

kind: Service

metadata:

annotations:

openshift.io/generated-by: OpenShiftNewApp

creationTimestamp: 2018-10-24T17:17:35Z

labels:

app: mapit

name: mapit

namespace: app-management

resourceVersion: "19394"

selfLink: /api/v1/namespaces/app-management/services/mapit

uid: b2e86550-d7b0-11e8-bb81-02f850fb3182

spec:

clusterIP: 172.30.48.204

ports:

- name: 8080-tcp

port: 8080

protocol: TCP

targetPort: 8080

- name: 8778-tcp

port: 8778

protocol: TCP

targetPort: 8778

- name: 9779-tcp

port: 9779

protocol: TCP

targetPort: 9779

selector:

app: mapit

deploymentconfig: mapit

sessionAffinity: None

type: ClusterIP

status:

loadBalancer: {}

Take note of the selector stanza. Remember it.

It is also of interest to view the YAML of the **Pod** to understand how OpenShift wires components together. Go back and find the name of your mapit **Pod**, and then execute the following:

oc get pod mapit-1-6lczv -o yaml

Under the metadata section you should see the following:

labels:

app: mapit

deployment: mapit-1

deploymentconfig: mapit

name: mapit-1-6lczv

* The **Service** has selector stanza that refers to app: mapit and deploymentconfig: mapit.
* The **Pod** has multiple **Labels**:
  + deploymentconfig: mapit
  + app: mapit
  + deployment: mapit-1

**Labels** are just key/value pairs. Any **Pod** in this **Project** that has a **Label** that matches the **Selector** will be associated with the **Service**. If you look at the describe output again, you will see that there is one endpoint for the service: the existing mapit **Pod**.

The default behavior of new-app is to create just one instance of the item requested. We will see how to modify/adjust this in a moment, but there are a few more concepts to learn first.

Background: Deployment Configurations and Replication Controllers

While **Services** provide routing and load balancing for **Pods**, which may go in and out of existence, **ReplicationControllers** (RC) are used to specify and then ensure the desired number of **Pods** (replicas) are in existence. For example, if you always want an application to be scaled to 3 **Pods** (instances), a**ReplicationController** is needed. Without an RC, any **Pods** that are killed or somehow die/exit are not automatically restarted. **ReplicationControllers** are how OpenShift "self heals".

A **DeploymentConfiguration** (DC) defines how something in OpenShift should be deployed. From the [deployments documentation](https://docs.openshift.com/container-platform/3.11/architecture/core_concepts/deployments.html):

Building on replication controllers, OpenShift adds expanded support for the

software development and deployment lifecycle with the concept of deployments.

In the simplest case, a deployment just creates a new replication controller and

lets it start up pods. However, OpenShift deployments also provide the ability

to transition from an existing deployment of an image to a new one and also

define hooks to be run before or after creating the replication controller.

In almost all cases, you will end up using the **Pod**, **Service**, **ReplicationController** and **DeploymentConfiguration** resources together. And, in almost all of those cases, OpenShift will create all of them for you.

There are some edge cases where you might want some **Pods** and an **RC** without a **DC** or a **Service**, and others, but these are advanced topics not covered in these exercises.

Exploring Deployment-related Objects

Now that we know the background of what a **ReplicatonController** and **DeploymentConfig** are, we can explore how they work and are related. Take a look at the **DeploymentConfig** (DC) that was created for you when you told OpenShift to stand up the mapit image:

oc get dc

You will see something like the following:

NAME REVISION DESIRED CURRENT TRIGGERED BY

mapit 1 1 1 config,image(mapit:latest)

To get more details, we can look into the **ReplicationController** (**RC**).

Take a look at the **ReplicationController** (RC) that was created for you when you told OpenShift to stand up the mapit image:

oc get rc

You will see something like the following:

NAME DESIRED CURRENT READY AGE

mapit-1 1 1 1 4h

This lets us know that, right now, we expect one **Pod** to be deployed (Desired), and we have one **Pod**actually deployed (Current). By changing the desired number, we can tell OpenShift that we want more or less **Pods**.

Scaling the Application

Let’s scale our mapit "application" up to 2 instances. We can do this with the scale command.

oc scale --replicas=2 dc/mapit

To verify that we changed the number of replicas, issue the following command:

oc get rc

You will see something like the following:

NAME DESIRED CURRENT READY AGE

mapit-1 2 2 1 4h

You can see that we now have 2 replicas. Let’s verify the number of pods with the oc get podscommand:

oc get pods

You will see something like the following:

NAME READY STATUS RESTARTS AGE

mapit-1-6lczv 1/1 Running 0 4h

mapit-1-rq6t6 1/1 Running 0 1m

And lastly, let’s verify that the **Service** that we learned about in the previous lab accurately reflects two endpoints:

oc describe svc mapit

You will see something like the following:

Name: mapit

Namespace: app-management

Labels: app=mapit

Annotations: openshift.io/generated-by=OpenShiftNewApp

Selector: app=mapit,deploymentconfig=mapit

Type: ClusterIP

IP: 172.30.48.204

Port: 8080-tcp 8080/TCP

TargetPort: 8080/TCP

Endpoints: 10.129.0.2:8080,10.130.0.3:8080

Port: 8778-tcp 8778/TCP

TargetPort: 8778/TCP

Endpoints: 10.129.0.2:8778,10.130.0.3:8778

Port: 9779-tcp 9779/TCP

TargetPort: 9779/TCP

Endpoints: 10.129.0.2:9779,10.130.0.3:9779

Session Affinity: None

Events: <none>

Another way to look at a **Service**'s endpoints is with the following:

oc get endpoints mapit

And you will see something like the following:

NAME ENDPOINTS AGE

mapit 10.128.2.3:9779,10.129.0.3:9779,10.128.2.3:8080 + 3 more... 4h

Your IP addresses will likely be different, as each pod receives a unique IP within the OpenShift environment. The endpoint list is a quick way to see how many pods are behind a service.

Overall, that’s how simple it is to scale an application (**Pods** in a **Service**). Application scaling can happen extremely quickly because OpenShift is just launching new instances of an existing image, especially if that image is already cached on the node.

One last thing to note is that there are actually several ports defined on this **Service**. Earlier we said that a pod gets a single IP and has control of the entire port space on that IP. While something running inside the **Pod** may listen on multiple ports (single container using multiple ports, individual containers using individual ports, a mix), a **Service** can actually proxy/map ports to different places.

For example, a **Service** could listen on port 80 (for legacy reasons) but the **Pod** could be listening on port 8080, 8888, or anything else.

In this mapit case, the image we ran has several EXPOSE statements in the Dockerfile, so OpenShift automatically created ports on the service and mapped them into the **Pods**.

Application "Self Healing"

Because OpenShift’s **RCs** are constantly monitoring to see that the desired number of **Pods** actually is running, you might also expect that OpenShift will "fix" the situation if it is ever not right. You would be correct!

Since we have two **Pods** running right now, let’s see what happens if we "accidentally" kill one. Run the oc get pods command again, and choose a **Pod** name. Then, do the following:

oc delete pod mapit-1-6lczv && oc get pods

And you will see something like the following:

pod "mapit-1-lhqgq" deleted

NAME READY STATUS RESTARTS AGE

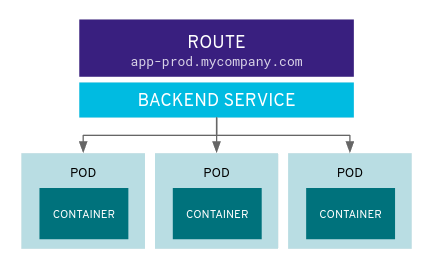
mapit-1-7dw5t 1/1 Running 0 3m

mapit-1-rgnht 0/1 ContainerCreating 0 2s

Did you notice anything? There’s a new container already being created.

Also, the ContainerCreating **Pod** has a different name. That’s because OpenShift almost immediately detected that the current state (1 **Pod**, because one was deleted) didn’t match the desired state (2 **Pods**), and it fixed it by scheduling another **Pod**.

Background: Routes



*Figure 3. OpenShift Route*

While **Services** provide internal abstraction and load balancing within an OpenShift environment, sometimes clients (users, systems, devices, etc.) **outside** of OpenShift need to access an application. The way that external clients are able to access applications running in OpenShift is through the OpenShift routing layer. And the data object behind that is a **Route**.

The default OpenShift router (HAProxy) uses the HTTP header of the incoming request to determine where to proxy the connection. You can optionally define security, such as TLS, for the **Route**. If you want your **Services** (and by extension, your **Pods**) to be accessible to the outside world, then you need to create a **Route**.

Do you remember setting up the router? You probably don’t. That’s because the installer settings created a router for you! The router lives in the default **Project**, and you can see information about it with the following command:

oc describe dc router -n default

Creating a Route

Creating a **Route** is a pretty straight-forward process. You simply expose the **Service** via the command line. If you remember from earlier, your **Service** name is mapit. With the **Service** name, creating a **Route** is a simple one-command task:

oc expose service mapit

You will see:

route.route.openshift.io/mapit exposed

Verify the **Route** was created with the following command:

oc get route

You will see something like:

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

mapit mapit-app-management.apps.004325440631.aws.testdrive.openshift.com mapit 8080-tcp None

If you take a look at the HOST/PORT column, you’ll see a familiar looking FQDN. The default behavior of OpenShift is to expose services on a formulaic hostname:

{SERVICENAME}-{PROJECTNAME}.{ROUTINGSUBDOMAIN}

How does this work? Firstly, the ROUTINGSUBDOMAIN can be configured at install time. We did this for you. In the /etc/ansible/hosts file you will find the following line:

openshift\_master\_default\_subdomain=apps.004325440631.aws.testdrive.openshift.com

There is also a wildcard DNS entry that points \*.apps... to the host where the router lives. OpenShift concatenates the **Service** name, **Project** name, and the routing subdomain to create this FQDN/URL.

You can visit this URL using your browser, or using curl, or any other tool. It should be accessible from anywhere on the internet.

The **Route** is associated with the **Service**, and the router automatically proxies connections directly to the **Pod**. The router itself runs as a **Pod**. It bridges the "real" internet to the SDN.

If you take a step back to examine everything you’ve done so far, in three commands you deployed an application, scaled it, and made it accessible to the outside world:

oc new-app docker.io/siamaksade/mapit

oc scale --replicas=2 dc/mapit

oc expose service mapit

Scale Down

Before we continue, go ahead and scale your application down to a single instance:

oc scale --replicas=1 dc/mapit

Application Probes

OpenShift provides rudimentary capabilities around checking the liveness and/or readiness of application instances. If the basic checks are insufficient, OpenShift also allows you to run a command inside the **Pod**/container in order to perform the check. That command could be a complicated script that uses any language already installed inside the container image.

There are two types of application probes that can be defined:

**Liveness Probe**

A liveness probe checks if the container in which it is configured is still running. If the liveness probe fails, the container is killed, which will be subjected to its restart policy.

**Readiness Probe**

A readiness probe determines if a container is ready to service requests. If the readiness probe fails, the endpoint’s controller ensures the container has its IP address removed from the endpoints of all services that should match it. A readiness probe can be used to signal to the endpoint’s controller that even though a container is running, it should not receive any traffic.

More information on probing applications is available in the [Application Health](https://docs.openshift.com/container-platform/latest/dev_guide/application_health.html) section of the documentation.

Add Probes to the Application

The oc set command can be used to perform several different functions, one of which is creating and/or modifying probes. The mapit application exposes an endpoint which we can check to see if it is alive and ready to respond. You can test it using curl:

curl mapit-app-management.apps.004325440631.aws.testdrive.openshift.com/health

You will get some JSON as a response:

{"status":"UP","diskSpace":{"status":"UP","total":10724835328,"free":10257825792,"threshold":10485760}}

We can ask OpenShift to probe this endpoint for liveness with the following command:

oc set probe dc/mapit --liveness --get-url=http://:8080/health --initial-delay-seconds=30

You can then see that this probe is defined in the oc describe output:

oc describe dc mapit

You will see a section like:

...

Containers:

mapit:

Image: docker.io/siamaksade/mapit@sha256:338a3031df6354e3adc3ba7d559ae22a0f2c79eade68aa72447f821cc7b8370c

Ports: 8080/TCP, 8778/TCP, 9779/TCP

Liveness: http-get http://:8080/health delay=30s timeout=1s period=10s #success=1 #failure=3

Volume Mounts: <none>

Environment Variables: <none>

No volumes.

...

Similarly, you can set a readiness probe in the same manner:

oc set probe dc/mapit --readiness --get-url=http://:8080/health --initial-delay-seconds=30

Add Storage to the Application

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

|  |  |
| --- | --- |
|  | The directories that make up the containers 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 happens regularly in a container orchestration environment. |

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.

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

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 dc/mapit --add --name=mapit-storage -t pvc --claim-mode=ReadWriteMany --claim-size=1Gi --claim-name=mapit-storage --mount-path=/app-storage

The output looks like this:

deploymentconfig.apps.openshift.io/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 DeploymentConfig of mapitis updated to reference this storage and make it available under the /app-storage directory inside the pod.

You can see the new DeploymentConfig like this:

oc get dc mapit

The output will show that a new revision was created as part of the update with storage.

NAME REVISION DESIRED CURRENT TRIGGERED BY

mapit 4 1 1 config,image(mapit:latest)

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

We will look at how this storage was provisioned automatically in the background using *Red Hat OpenShift Container Storage* later. You will also learn how to request storage as part of a template.

Suffice it to say, a 1GiB GlusterFS volume has been created and made available to the pod.

Log on to the new pod (**your pod names will be different**) using the remote-shell capability of the occlient:

oc rsh mapit-4-d872b

**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 an additional directory there under /app-storage

total 36K

drwxr-xr-x. 19 root root 4.0K Apr 9 11:00 .

drwxr-xr-x. 19 root root 4.0K Apr 9 11:00 ..

-rwxr-xr-x. 1 root root 0 Apr 9 11:00 .dockerenv

-rw-r--r--. 1 root root 16K Dec 14 2016 anaconda-post.log

drwxrwsr-x. 4 root 2000 4.0K Apr 9 11:05 app-storage

lrwxrwxrwx. 1 root root 7 Dec 14 2016 bin -> usr/bin

drwxrwxrwx. 2 jboss root 137 Aug 4 2017 deployments

drwxr-xr-x. 5 root root 360 Apr 9 11:00 dev

drwxr-xr-x. 52 root root 4.0K Apr 9 11:00 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. 4 root root 61 Jan 18 2017 opt

dr-xr-xr-x. 299 root root 0 Apr 9 11:00 proc

dr-xr-x---. 2 root root 114 Dec 14 2016 root

drwxr-xr-x. 11 root root 145 Apr 9 11:00 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 9 09:14 sys

drwxrwxrwt. 10 root root 241 Apr 9 11:00 tmp

drwxr-xr-x. 13 root root 155 Dec 16 2016 usr

drwxr-xr-x. 18 root root 238 Dec 14 2016 var

|  |  |
| --- | --- |
|  | This is where the persistent storage appears inside the container |

One of the interesting aspects of persistent storage from GlusterFS is that it is actually "shared" as indicated by the claim mode **ReadWriteMany**. This means that multiple containers can read and write to the same storage location concurrently.

Let’s try this. First write a file to the persistent, shared storage and then exit the remote shell session.

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

exit

Now, let’s scale your deployment to two pods:

oc scale --replicas=2 dc/mapit

After some time, ensure both are in the Running state:

oc get pods

NAME READY STATUS RESTARTS AGE

mapit-4-ljjmf 1/1 Running 0 24m

mapit-4-d872b 1/1 Running 0 25m

Read the text file from the other pod using the cat command appended directly to the oc rsh call:

oc rsh mapit-4-ljjmf cat /app-storage/hello.txt

You should see the content of the file from **the other pod**:

Hello World from OpenShift

This illustrates how to provide persistent storage, that is independent from the pod lifecycle and can optionally be shared by multiple pods at the same time.

[Go to previous module](http://support.004325440631.aws.testdrive.openshift.com/index.html#/workshop/ocp-admin-testdrive/module/installation)

Project Request Template, Quotas, Limits

In the Application Management Basics lab, you dealt with the fundamental building blocks of OpenShift, all contained within a **Project**.

Out of the box, OpenShift does not restrict the quantity of objects or amount of resources (eg: CPU, memory) that they can consume within a **Project**. Further, users may create an unlimited number of **Projects**. In a world with no constraints, or in a POC-type environment, that would be fine. But reality calls for a little restraint.

|  |  |
| --- | --- |
|  | Before continuing, make sure you are logged in with the built-in system administrator account in OpenShift.  oc login -u system:admin |

Background: Project Request Template

When users use the new-project command, what is actually happening behind the scenes is a project request flow. Much like it sounds, there is a **Template** that is processed during the project request.

View the Default Project Request Template

Execute the following command to view the built-in default **Project Request Template**:

oc adm create-bootstrap-project-template

If you dig into the JSON output of this command, you will notice that there are various parameters defined at the end. Take a look at the help output for the new-project command:

oc new-project -h

Do you see how there is a --display-name directive on new-project and how there is a PROJECT\_DISPLAYNAME parameter?

The new-project workflow involves the user providing information to fulfill the project request. OpenShift decides if this request should be allowed (for example, are users allowed to create **Projects**? Does this user have too many **Projects**?) and, if so, processes the **Template**.

If you look at the objects defined in the **Template**, you will notice that there’s no mention of quota or limits. You’ll change that now.

|  |  |
| --- | --- |
|  | **Templates** are a powerful tool that enable you to easily create reusable sets of OpenShift objects with powerful parameterization. They can be used to quickly deploy more complicated and related components into OpenShift, and can be a useful part of your organization’s software development lifecycle (SDLC). More information can be found in the [template documentation](https://docs.openshift.com/container-platform/3.11/dev_guide/templates.html). We will not go into more detail on **Templates** in these exercises. |

Modify the Project Request Template

You won’t actually have to make template changes in this lab — we’ve made them for you already. Use cat, less, or your favorite editor to view the modified **Project Request Template**:

cat /opt/lab/support/project\_request\_template.yaml

Take note that there are two new sections added: **ResourceQuota** and **LimitRange**.

Background: ResourceQuota

The [quota documentation](https://docs.openshift.com/container-platform/3.11/admin_guide/quota.html) provides a great description of what **ResourceQuota** is about:

A resource quota, defined by a ResourceQuota object, provides constraints that

limit aggregate resource consumption per project. It can limit the quantity of

objects that can be created in a project by type, as well as the total amount of

compute resources and storage that may be consumed by resources in that

project.

In our case, we are setting a specific set of quota for CPU, memory, storage, volume claims, and **Pods**. Take a look at the ResourceQuota section from the project\_request\_template.yaml file:

- apiVersion: v1

kind: ResourceQuota

metadata:

name: ${PROJECT\_NAME}-quota

spec:

hard:

pods: 10

requests.cpu: 4000m

requests.memory: 8Gi

resourcequotas: 1

requests.storage: 50Gi

persistentvolumeclaims: 5

glusterfs-storage.storageclass.storage.k8s.io/requests.storage: 25Gi

glusterfs-registry-block.storageclass.storage.k8s.io/persistentvolumeclaims: 0

|  |  |
| --- | --- |
|  | While only one quota can be defined in a **Project**, it still needs a unique name/id. |
|  | The total number of pods in a non-terminal state that can exist in the project. |
|  | CPUs are measured in "milicores". More information on how Kubernetes/OpenShift calculates cores can be found in the [upstream documentation](https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/). |
|  | There is a system of both limits and requests that we will discuss more when we get to the **LimitRange** object. |
|  | Across all persistent volume claims in a project, the sum of storage requested cannot exceed this value. |
|  | The total number of persistent volume claims in a project. |
|  | This setting limits the amount of storage that can be provisioned using the glusterfs-storage **StorageClass**. |
|  | This setting limits the number of **PersistentVolumeClaims** for a **StorageClass** called glusterfs-registry-block. A value of 0 means that creating **PersistentVolumeClaims** from this storage class is not allowed in this project. |

For more details on the available quota options, refer back to the [quota documentation](https://docs.openshift.com/container-platform/3.11/admin_guide/quota.html).

Background: LimitRange

The [limit range documentation](https://docs.openshift.com/container-platform/3.11/admin_guide/limits.html) provides some good background:

A limit range, defined by a LimitRange object, enumerates compute resource

constraints in a project at the pod, container, image, image stream, and

persistent volume claim level, and specifies the amount of resources that a pod,

container, image, image stream, or persistent volume claim can consume.

While the quota sets an upper bound on the total resource consumption within a project, the LimitRange generally applies to individual resources. For example, setting how much CPU an individual **Pod** or container can use.

Take a look at the sample LimitRange definition that we have provided in theproject\_request\_template.yaml file:

- apiVersion: v1

kind: LimitRange

metadata:

name: ${PROJECT\_NAME}-limits

creationTimestamp: null

spec:

limits:

-

type: Container

max:

cpu: 4000m

memory: 1024Mi

min:

cpu: 10m

memory: 5Mi

default:

cpu: 4000m

memory: 1024Mi

defaultRequest:

cpu: 100m

memory: 512Mi

The difference between requests and default limits is important, and is covered in the [limit range documentation](https://docs.openshift.com/container-platform/3.11/admin_guide/limits.html). But, generally speaking:

|  |  |
| --- | --- |
|  | max is the highest value that may be specified for limits and requests |
|  | min is the lowest value that may be specified for limits and requests |
|  | default is the maximum amount (limit) that the container may consume, when nothing is specified |
|  | defaultRequest is the minimum amount that the container may consume, when nothing is specified |

In addition to these topics, there are things like **Quality of Service Tiers** as well as a **Limit** : **Request**ratio. There is additionally more information in the [compute resources](https://docs.openshift.com/container-platform/3.11/dev_guide/compute_resources.html) section of the documentation.

For the sake of brevity, suffice it to say that there is a complex and powerful system of Quality of Service and resource management in OpenShift. Understanding the types of workloads that will be run in your cluster will be important to coming up with sensible values for all of these settings.

The settings we provide for you in these examples generally restrict projects to:

* A total CPU quota of 4 cores (4000m) where
  + Individual containers
    - must use 4 cores or less
    - cannot be defined with less than 10 milicores
    - will default to a request of 100 milicores (if not specified)
    - may burst up to a limit of 4 cores (if not specified)
* A total memory usage of 8 Gibibyte (8192 Megabytes) where
  + Individual containers
    - must use 1 Gi or less
    - cannot be defined with less than 5 Mi
    - will default to a request of 512 Mi
    - may burst up to a limit of 1024 Mi
* Total storage claims of 25 Gi or less
* A total number of 5 volume claims
* 10 or less **Pods**

In combination with quota, you can create very fine-grained controls, even across projects, for how users are allowed to request and utilize OpenShift’s various resources.

|  |  |
| --- | --- |
|  | Remember that quotas and limits are applied at the **Project** level. **Users** may have access to multiple **Projects**, but quotas and limits do not apply directly to **Users**. If you want to apply one quota across multiple **Projects**, then you should look at the [multi-project quota](https://docs.openshift.com/container-platform/3.11/admin_guide/multiproject_quota.html) documentation. We will not cover multi-project quota in these exercises. |

Installing the Project Request Template

OK, with this background in place, let’s go ahead and actually tell OpenShift to use this new **Project Request Template**.

Create the Template

As we discussed earlier, a **Template** is just another type of OpenShift object. The oc command provides a create function that will take YAML/JSON as input and simply instantiate the objects provided.

Go ahead and execute the following:

oc create -f /opt/lab/support/project\_request\_template.yaml -n default

This will create the **Template** object in the default **Project**. You can now see the **Templates** in the default project with the following:

oc get template -n default

You will see something like the following:

NAME DESCRIPTION PARAMETERS OBJECTS

project-request 5 (5 blank) 7

Edit the master-config.yaml

This handy command will replace the default configuration (a blank entry of '') with the desired setting of default/project-request:

sudo sed -i "s/projectRequestTemplate: ''/projectRequestTemplate: 'default\/project-request'/" /etc/origin/master/master-config.yaml

Restart the Master

Since you have made a configuration change to the master, you will need to restart its service. You can do so with the following command with sudo privileges:

sudo /usr/local/bin/master-restart api

sudo /usr/local/bin/master-restart controllers

|  |  |
| --- | --- |
|  | In OpenShift 3.11 there were some underlying changes to how the master software is run. Part of this change included running the control plane itself in containers. This involved a switch from a more traditional systemd-type service to using a script to restart the various components. |

Test the Project Request Template

At this point you have reconfigured the master to use the **Project Request Template** (a special kind of **Template**) called project-request that is located in the default **Project**. Now it is time to observe this change in action.

Create a New Project

When creating a new project, you should see that a **Quota** and a **LimitRange** are created with it. First, create a new project called template-test:

oc new-project template-test

Then, use describe to look at some of this **Project’s** details:

oc describe project template-test

The output will look something like:

Name: template-test

Created: 7 seconds ago

Labels: <none>

Annotations: openshift.io/description=

openshift.io/display-name=

openshift.io/requester=system:admin

openshift.io/sa.scc.mcs=s0:c10,c0

openshift.io/sa.scc.supplemental-groups=1000090000/10000

openshift.io/sa.scc.uid-range=1000090000/10000

Display Name: <none>

Description: <none>

Status: Active

Node Selector: <none>

Quota:

Name: template-test-quota

Resource Used Hard

-------- ---- ----

glusterfs-registry-block.storageclass.storage.k8s.io/persistentvolumeclaims 0 0

glusterfs-storage.storageclass.storage.k8s.io/requests.storage 0 25Gi

persistentvolumeclaims 0 5

pods 0 10

requests.cpu 0 4

requests.memory 0 8Gi

requests.storage 0 50Gi

resourcequotas 0 1

Resource limits:

Name: template-test-limits

Type Resource Min Max Default Limit Limit/Request

---- -------- --- --- --- ----- -------------

Container memory 5Mi 1Gi 1Gi 1Gi -

Container cpu 10m 4 4 4 -

You can also see that the **Quota** and **LimitRange** objects were created:

oc get quota -n template-test

You will see:

NAME CREATED AT

template-test-quota 2018-10-24T19:19:40Z

And:

oc get limitrange -n template-test

You will see:

NAME CREATED AT

template-test-limits 2018-10-24T19:19:40Z

Configure Using the Installer

While it is possible to configure the **ProjectRequestTemplate** using the advanced installer, you are still responsible for installing the template into the **Project** that you specify. For example, we could have specified the following:

osm\_project\_request\_template='default/project-request'

But, until a **Template** was created in the default **Project** called project-request, user **Project**creation would have failed due to a lack of the template. So, beware.

Clean Up

If you wish, you can deploy the application from the Application Management Basics lab again inside this template-test project to observe how the **Quota** and **LimitRange** are applied. If you do, be sure to look at the JSON/YAML output (oc get …​ -o yaml) for things like the **DeploymentConfig** and the **Pod**.

Before you continue, you may wish to delete the **Project** you just created:

oc delete project template-test

Configuring External Authentication Providers

OpenShift supports a number of different authentication providers, and you can find the complete list in the [authentication documentation](https://docs.openshift.com/container-platform/3.11/install_config/configuring_authentication.html). One of the most commonly used authentication providers is LDAP, whether provided by Microsoft Active Directory or by other sources.

OpenShift can perform user authentication against an LDAP server, and can also configure group membership and certain RBAC attributes based on LDAP group membership.

Background: LDAP Structure

In this environment we are providing LDAP with a Red Hat Identity Management server running on Red Hat Enterprise Linux 7 running in a VM. IDM is configured with the following user groups:

* ose-user: Users with OpenShift access
  + Any users who should be able to log-in to OpenShift must be members of this group
  + All of the below mentioned users are in this group
* ose-normal-dev: Normal OpenShift users
  + Regular users of OpenShift without special permissions
  + Contains: normaluser1, teamuser1, teamuser2
* ose-fancy-dev: Fancy OpenShift users
  + Users of OpenShift that are granted some special privileges
  + Contains: fancyuser1, fancyuser2
* ose-teamed-app: Teamed app users
  + A group of users that will have access to the same OpenShift **Project**
  + Contains: teamuser1, teamuser2

Examine the master configuration

The installation of OpenShift already included setting up LDAP authentication in the inventory file. As the root user on the master host, examine the master config with cat, less, or your favorite editor:

sudo cat /etc/origin/master/master-config.yaml

Look for one of the oauthConfig sections that looks like the following:

oauthConfig:

assetPublicURL: https://master.004325440631.aws.testdrive.openshift.com/console/

grantConfig:

method: auto

identityProviders:

- challenge: true

login: true

mappingMethod: claim

name: idm

provider:

apiVersion: v1

attributes:

email:

- mail

id:

- dn

name:

- cn

preferredUsername:

- uid

bindDN: uid=admin,cn=users,cn=accounts,dc=unset,dc=ocp-admin,dc=aws,dc=openshifttestdrive,dc=com

bindPassword: ldapadmin

ca: /etc/origin/master/idm\_ldap\_ca.crt

insecure: false

kind: LDAPPasswordIdentityProvider

url: ldap://idm.internal.aws.testdrive.openshift.com/cn=users,cn=accounts,dc=unset,dc=ocp-admin,dc=aws,dc=openshifttestdrive,dc=com?uid?sub?(memberOf=cn=ose-user,cn=groups,cn=accounts,dc=unset,dc=ocp-admin,dc=aws,dc=openshifttestdrive,dc=com)

...

Some notable fields under identityProviders::

|  |  |
| --- | --- |
|  | mappingMethod: claim: This section has to do with how usernames are assigned within an OpenShift cluster when multiple providers are configured. See the [Identity provider Parameters](https://docs.openshift.com/container-platform/3.11/install_config/configuring_authentication.html#identity-providers_parameters) section for more information. |
|  | name: The unique ID of the identity provider. It is possible to have multiple authentication providers in an OpenShift environment, and OpenShift is able to distinguish between them. |
|  | attributes: This section defines the LDAP fields to iterate over and assign to the fields in the OpenShift user’s "account". If any attributes are not found / not populated when searching through the list, the entire authentication fails. In this case we are creating an identity that is associated with the LDAP dn, an email address from the LDAP mail, a name from the LDAP cn, and a username from the LDAP uid. |
|  | bindDN: When searching LDAP, bind to the server as this user. |
|  | bindPassword: The password to use when binding for searching. |
|  | ca: The CA certificate to use for validating the SSL certificate of the LDAP server. |
|  | url: Identifies the LDAP server and the search to perform. |

For more information on the specific details of LDAP authentication in OpenShift you can refer to the[LDAP provider documentation](https://docs.openshift.com/container-platform/3.11/install_config/configuring_authentication.html#LDAPPasswordIdentityProvider).

|  |  |
| --- | --- |
|  | You can find the installer configuration line that sets up LDAP authentication by looking in /etc/ansible/hosts at the line that begins withopenshift\_master\_identity\_providers= |

Syncing LDAP Groups to OpenShift Groups

In OpenShift, groups can be used to manage users and control permissions for multiple users at once. There is a section in the documentation on how to [sync groups with LDAP](https://docs.openshift.com/container-platform/3.11/install_config/syncing_groups_with_ldap.html). Syncing groups involves running a program called groupsync when logged into OpenShift as a user with cluster-adminprivileges, and using a configuration file that tells OpenShift what to do with the users it finds in the various groups.

We have provided a groupsync configuration file for you already. Using cat, less, or your favorite editor, look at the following file:

cat /opt/lab/support/groupsync.yaml

Without going into too much detail (you can look at the documentation), the groupsync config file does the following:

* searches LDAP using the specified bind user and password
* queries for any LDAP groups whose name begins with ose-
* creates OpenShift groups with a name from the cn of the LDAP group
* finds the members of the LDAP group and then puts them into the created OpenShift group
* uses the dn and uid as the UID and name attributes, respectively, in OpenShift

Make sure you are logged in as the root user:

sudo -i

Then, make sure you are logged in as a cluster administrator:

oc login -u system:admin

And then execute the groupsync:

oc adm groups sync --sync-config=/opt/lab/support/groupsync.yaml --confirm

You will see output like the following:

group/ose-user

group/ose-normal-dev

group/ose-fancy-dev

group/ose-teamed-app

What you are seeing is the **Group** objects that have been created by the groupsync command. If you are curious about the --confirm flag, check the output of the help with oc adm groups sync -h.

If you want to see the **Groups** that were created, execute the following:

oc get groups

You will see output like the following:

NAME USERS

ose-fancy-dev fancyuser1, fancyuser2

ose-normal-dev normaluser1, teamuser1, teamuser2

ose-teamed-app teamuser1, teamuser2

ose-user normaluser1, fancyuser1, fancyuser2, teamuser1, teamuser2

Take a look at a specific group in YAML:

oc get group ose-fancy-dev -o yaml

The YAML looks like:

apiVersion: v1

kind: Group

metadata:

annotations:

openshift.io/ldap.sync-time: 2017-08-29T14:29:33Z

openshift.io/ldap.uid: cn=ose-fancy-dev,cn=groups,cn=accounts,dc=auth,dc=internal,dc=aws,dc=testdrive,dc=openshift,dc=com

openshift.io/ldap.url: idm.internal.aws.testdrive.openshift.com:389

creationTimestamp: 2017-08-29T14:29:33Z

labels:

openshift.io/ldap.host: idm.internal.aws.testdrive.openshift.com

name: ose-fancy-dev

resourceVersion: "5242"

selfLink: /oapi/v1/groups/ose-fancy-dev

uid: 7a353ec7-8cc6-11e7-b355-0ee4d6c98466

users:

- fancyuser1

- fancyuser2

OpenShift has automatically associated some LDAP metadata with the **Group**, and has listed the users who are in the group.

What happens if you list the **Users**?

oc get user

If you logged into the web console as teamuser1 previously, you will see:

NAME UID FULL NAME IDENTITIES

teamuser1 ace42f02-4bcf-11e8-98be-0a8fee356162 OpenShift User idm:uid=teamuser1,cn=users,cn=accounts,dc=auth,dc=internal,dc=aws,dc=testdrive,dc=openshift,dc=com

Or, if you did not login with the UI before, you will get:

No resources found.

Why would there be no **Users** found? They are clearly listed in the **Group** definition.

**Users** are not actually created until the first time they try to log in. What you are seeing in the **Group**definition is simply a placeholder telling OpenShift that, if it encounters a **User** with that specific ID, that it should be associated with the **Group**.

Change Group Policy

In your environment, there is a special group of super developers called *ose-fancy-dev* who should have special cluster-reader privileges. This is a role that allows a user to view administrative-level information about the cluster. For example, they can see the list of all **Projects** in the cluster.

Change the policy for the ose-fancy-dev **Group**:

oc adm policy add-cluster-role-to-group cluster-reader ose-fancy-dev

|  |  |
| --- | --- |
|  | If you are interested in the different roles that come with OpenShift, you can learn more about them in the [role-based access control (RBAC)](https://docs.openshift.com/container-platform/3.11/admin_guide/manage_rbac.html#admin-guide-manage-rbac) documentation. |

Examine cluster-reader policy

Go ahead and login as a regular user:

oc login -u normaluser1 -p openshift

Then, try to list **Projects**:

oc get projects

You will see:

No resources found.

Now, login as a member of ose-fancy-dev:

oc login -u fancyuser1 -p openshift

And then perform the same oc get projects and you will now see the list of all of the projects in the cluster:

NAME DISPLAY NAME STATUS

app-management Active

default Active

kube-public Active

kube-system Active

management-infra Active

openshift Active

openshift-infra Active

openshift-logging Active

openshift-metrics Active

openshift-node Active

openshift-sdn Active

openshift-web-console Active

storage Active

You should now be starting to understand how RBAC in OpenShift Container Platform can work.

Create Projects for Collaboration

Make sure you login as the cluster administrator:

oc login -u system:admin

Then, create several **Projects** for people to collaborate:

oc adm new-project app-dev --display-name="Application Development"

oc adm new-project app-test --display-name="Application Testing"

oc adm new-project app-prod --display-name="Application Production"

You have now created several **Projects** that represent a typical Software Development Lifecycle setup. Next, you will configure **Groups** to grant collaborative access to these projects.

|  |  |
| --- | --- |
|  | Creating projects with oc adm new-project does **not** use the project request process or the project request template. These projects will not have quotas or limitranges applied by default. A cluster administrator can "impersonate" other users, so there are several options if you wanted these projects to get quotas/limit ranges:   1. use --as to specify impersonating a regular user with oc new-project 2. use oc process and provide values for the project request template, piping into create (eg: oc process …​ | oc create -f -). This will create all of the objects in the project request template, which would include the quota and limit range. 3. manually create/define the quota and limit ranges after creating the projects.   For these exercises it is not important to have quotas or limit ranges on these projects. |

Map Groups to Projects

As you saw earlier, there are several roles within OpenShift that are preconfigured. When it comes to **Projects**, you similarly can grant view, edit, or administrative access. Let’s give our ose-teamed-appusers access to edit the development and testing projects:

oc adm policy add-role-to-group edit ose-teamed-app -n app-dev

oc adm policy add-role-to-group edit ose-teamed-app -n app-test

And then give them access to view production:

oc adm policy add-role-to-group view ose-teamed-app -n app-prod

Now, give the ose-fancy-dev group edit access to the production project:

oc adm policy add-role-to-group edit ose-fancy-dev -n app-prod

Examine Group Access

Log in as normaluser1 and see what **Projects** you can see:

oc login -u normaluser1 -p openshift

oc get projects

You should get:

No resources found.

Then, try teamuser1 from the ose-teamed-app group:

oc login -u teamuser1 -p openshift

oc get projects

You should get:

NAME DISPLAY NAME STATUS

app-dev Application Development Active

app-prod Application Production Active

app-test Application Testing Active

You did not grant the team users edit access to the production project. Go ahead and try to create something in the production project as teamuser1:

oc project app-prod

oc new-app docker.io/siamaksade/mapit

You will see that it will not work:

error: can't lookup images: imagestreamimports.image.openshift.io is forbidden: User "teamuser1" cannot create imagestreamimports.image.openshift.io in the namespace "app-prod": User "teamuser1" cannot create imagestreamimports.image.openshift.io in project "app-prod"

error: local file access failed with: stat docker.io/siamaksade/mapit: no such file or directory

error: unable to locate any images in image streams, templates loaded in accessible projects, template files, local docker images with name "docker.io/siamaksade/mapit"

Argument 'docker.io/siamaksade/mapit' was classified as an image, image~source, or loaded template reference.

The 'oc new-app' command will match arguments to the following types:

1. Images tagged into image streams in the current project or the 'openshift' project

- if you don't specify a tag, we'll add ':latest'

2. Images in the Docker Hub, on remote registries, or on the local Docker engine

3. Templates in the current project or the 'openshift' project

4. Git repository URLs or local paths that point to Git repositories

--allow-missing-images can be used to point to an image that does not exist yet.

See 'oc new-app -h' for examples.

This failure is exactly what we wanted to see.

Prometheus

Now that you have a user with cluster-reader privileges (one that can see many administrative aspects of the cluster), we can take a look at Prometheus and attempt to log-in to it.

Login as a the user with cluster-reader privileges:

oc login -u fancyuser1 -p openshift

Find the prometheus Route with the following command:

oc get route prometheus-k8s -n openshift-monitoring

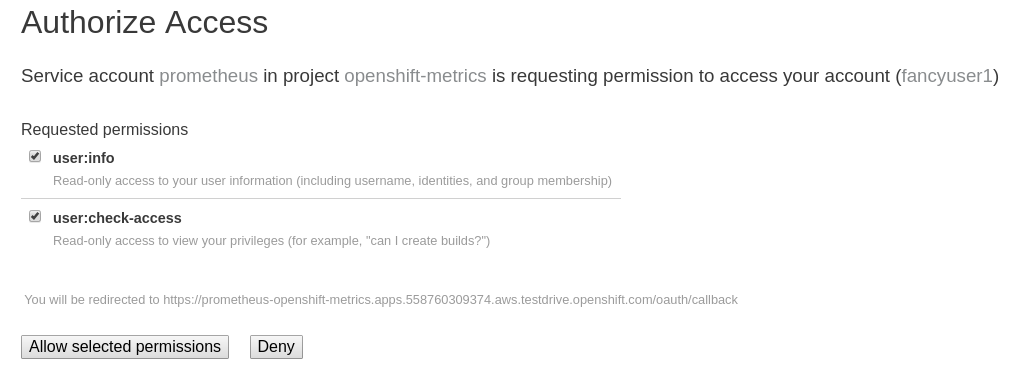
You will see something like the following:

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

prometheus-k8s prometheus-k8s-openshift-monitoring.apps.004325440631.aws.testdrive.openshift.com prometheus-k8s web reencrypt None

The installer configured a Route for prometheus by default. Go ahead and click the [Prometheus link](https://prometheus-k8s-openshift-monitoring.apps.004325440631.aws.testdrive.openshift.com/) to open it in your browser. You’ll be greeted with a login screen. Use the fancyuser1 user that you gave cluster-reader privileges to earlier. More specifically, the ose-fancy-dev group has cluster-reader permissions, and fancyuser1 is a member. Remember that the password for all of these users is openshift. You will probably get a certificate error because of the self-signed certificate. Make sure to accept it.

Once you log-in, the first time you will be presented with an auth proxy permissions acknowledgement:



*Figure 1. Auth Proxy Acceptance.*

There is actually an OAuth proxy that sits in the flow between you and the Prometheus container. This proxy is used to validate your AuthenticatioN (AuthN) as well as authorize (AuthZ) what is allowed to happen. Here you are explicitly authorizing the permissions from your fancyuser1 account to be used as part of accessing Prometheus. Hit *Allow selected permissions*.

At this point you are viewing Prometheus. If you look at Status and then Targets you can see some interesting information about the current state of the cluster.

## Infrastructure Management, Metrics and Logging

In this lab you will explore various aspects of managing cluster infrastructure. This includes extending the OpenShift cluster and installation of the Logging and Metrics components, all automated by the installer. It also includes some maintenance of nodes, as well as manipulating the multitenant network.

|  |  |
| --- | --- |
|  | It is recommended that you sudo -i to root before performing these exercises. |

### Quick Background on Nodes, Groups and Selectors

Log in as cluster administrator:

oc login -u system:admin

Verify your cluster currently consists of 5 nodes. Remember that the Master is also a node, and that we have a host dedicated to running OpenShift’s infrastructure.

oc get nodes

You should see:

NAME STATUS ROLES AGE VERSION

infra.internal.aws.testdrive.openshift.com Ready infra 9m v1.11.0+d4cacc0

master.internal.aws.testdrive.openshift.com Ready master 15m v1.11.0+d4cacc0

node01.internal.aws.testdrive.openshift.com Ready compute 9m v1.11.0+d4cacc0

node02.internal.aws.testdrive.openshift.com Ready compute 9m v1.11.0+d4cacc0

node03.internal.aws.testdrive.openshift.com Ready compute 9m v1.11.0+d4cacc0

In the App Management lab we talked a little bit about Selectors as they are used with Services to find Pods. But there is also a concept of a NodeSelector. It is used to tell a Pod which Node it should run on. During the installation process, when you don’t specify anything, OpenShift will configure a default NodeSelector. You can find it with the following command:

grep defaultNodeSelector /etc/origin/master/master-config.yaml

You’ll see:

defaultNodeSelector: node-role.kubernetes.io/compute=true

This looks like the key=value syntax we are used to seeing when we have talked previously about Labels. Execute the following:

oc get node --show-labels

You’ll see something like:

NAME STATUS ROLES AGE VERSION LABELS

infra.internal.aws.testdrive.openshift.com Ready infra 14m v1.11.0+d4cacc0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=infra.internal.aws.testdrive.openshift.com,node-role.kubernetes.io/infra=true

master.internal.aws.testdrive.openshift.com Ready master 20m v1.11.0+d4cacc0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/hostname=master.internal.aws.testdrive.openshift.com,node-role.kubernetes.io/master=true

node01.internal.aws.testdrive.openshift.com Ready compute 14m v1.11.0+d4cacc0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,glusterfs=storage-host,kubernetes.io/hostname=node01.internal.aws.testdrive.openshift.com,node-role.kubernetes.io/compute=true

node02.internal.aws.testdrive.openshift.com Ready compute 14m v1.11.0+d4cacc0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,glusterfs=storage-host,kubernetes.io/hostname=node02.internal.aws.testdrive.openshift.com,node-role.kubernetes.io/compute=true

node03.internal.aws.testdrive.openshift.com Ready compute 14m v1.11.0+d4cacc0 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,glusterfs=storage-host,kubernetes.io/hostname=node03.internal.aws.testdrive.openshift.com,node-role.kubernetes.io/compute=true

|  |  |
| --- | --- |
|  | The above has line wraps in it, especially at larger font sizes. To get slightly reduced output you can also use the following command:  oc get node --show-labels | awk '{ print $1, $6 }' | column -t |

Every node has a Label for its role, as well as some extra labels that specify other details about the host. Where do these Labels come from? Execute the following:

grep openshift\_node\_group\_name /etc/ansible/hosts

You will see that each node has a node group name specified. The OpenShift installer has several node group definitions built-in for things like compute nodes (the generic hosts where all your apps will run), masters, infra nodes, and etc. If you want to define additional types, specify additional kubelet arguments, add additional labels to hosts, or similar things, you will need to define custom node groups. This is quite trivial, and you can find out how to do it in the [documentation for node groups](https://docs.openshift.com/container-platform/latest/install/configuring_inventory_file.html#configuring-inventory-defining-node-group-and-host-mappings).

Thinking back to the previous paragraphs, since the default NodeSelector isnode-role.kubernetes.io/compute=true, we can expect that pods without a NodeSelectorspecified will only end up on these compute hosts.

### Extending the Cluster

Extending the cluster is easy. Simply add a new set of hosts to an Ansible group called new\_nodes in the openshift-ansible installer’s inventory. Then, run the scaleup playbook.

#### Configure the Installer

Your environment already has 3 additional nodes provisioned, but you have not used them so far. They are already configured in the inventory file, but commented out with a #scaleup\_ prefix.

To see the lines run:

grep '#scaleup\_' /etc/ansible/hosts

Remove the #scaleup\_ comment prefix by running the below sed command:

sudo sed -i 's/#scaleup\_//g' /etc/ansible/hosts

When finished, your inventory file should look like the following:

*/etc/ansible/hosts*

**[OSEv3:children]**

masters

nodes

etcd

glusterfs

new\_nodes

...

**[new\_nodes]**

node04.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node04.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node04.004325440631.aws.testdrive.openshift.com

node05.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node05.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node05.004325440631.aws.testdrive.openshift.com

node06.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node06.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node06.004325440631.aws.testdrive.openshift.com

...

Now that these hosts are properly defined (uncommented), you can use Ansible to verify that they are, in fact, online:

ansible new\_nodes -m ping

You will see:

node05.internal.aws.testdrive.openshift.com | SUCCESS => {

"changed": false,

"ping": "pong"

}

node04.internal.aws.testdrive.openshift.com | SUCCESS => {

"changed": false,

"ping": "pong"

}

node06.internal.aws.testdrive.openshift.com | SUCCESS => {

"changed": false,

"ping": "pong"

}

Much like when you installed OpenShift originally, these new hosts have all of the [prerequisites](https://docs.openshift.com/container-platform/3.11/install_config/install/prerequisites.html) already taken care of.

#### Run the Playbook to Extend the Cluster

To extend your cluster run the following playbook:

ansible-playbook /usr/share/ansible/openshift-ansible/playbooks/openshift-node/scaleup.yml

The playbook takes 1-2 minutes to complete. When done, you can verify that there are now 6 computenodes:

oc get nodes -l node-role.kubernetes.io/compute=true

You will see:

NAME STATUS ROLES AGE VERSION

node01.internal.aws.testdrive.openshift.com Ready compute 1h v1.11.0+d4cacc0

node02.internal.aws.testdrive.openshift.com Ready compute 1h v1.11.0+d4cacc0

node03.internal.aws.testdrive.openshift.com Ready compute 1h v1.11.0+d4cacc0

node04.internal.aws.testdrive.openshift.com Ready compute 18m v1.11.0+d4cacc0

node05.internal.aws.testdrive.openshift.com Ready compute 18m v1.11.0+d4cacc0

node06.internal.aws.testdrive.openshift.com Ready compute 18m v1.11.0+d4cacc0

|  |  |
| --- | --- |
|  | When deploying a highly-available multi-master OpenShift environment, it is also possible to add new master nodes. There is a similar playbook to run. For more information on multi-master and HA setups, please refer to the [documentation](https://docs.openshift.com/container-platform/3.11/architecture/infrastructure_components/kubernetes_infrastructure.html#high-availability-masters). |

After the scaleup succeeds you need to remove the new\_nodes entry from [osev3:children]. You also need to remove the '[new\_nodes]' section to add the new nodes to the regular [nodes] section of the inventory file.

Check the two lines that got added to enable the scaleup operation:

grep new\_nodes /etc/ansible/hosts

You will see:

new\_nodes

[new\_nodes]

Remove [new\_nodes] to add new nodes to the [nodes] section in the inventory file.

sudo sed -i '/^\[new\_nodes/d' /etc/ansible/hosts

Remove new\_nodes from [osev3:children] section of the inventory file.

sudo sed -i '/^new\_nodes/d' /etc/ansible/hosts

Your modified inventory file should now look like this:

[OSEv3:children]

masters

nodes

etcd

glusterfs

#ocsinfra\_glusterfs\_registry

...

[nodes]

master.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-master' openshift\_hostname=master.internal.aws.testdrive.openshift.com openshift\_public\_hostname=master.004325440631.aws.testdrive.openshift.com

infra.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-infra' openshift\_hostname=infra.internal.aws.testdrive.openshift.com

node01.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node01.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node01.004325440631.aws.testdrive.openshift.com

node02.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node02.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node02.004325440631.aws.testdrive.openshift.com

node03.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node03.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node03.004325440631.aws.testdrive.openshift.com

node04.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node04.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node04.004325440631.aws.testdrive.openshift.com

node05.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node05.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node05.004325440631.aws.testdrive.openshift.com

node06.internal.aws.testdrive.openshift.com openshift\_node\_group\_name='node-config-compute' openshift\_hostname=node06.internal.aws.testdrive.openshift.com openshift\_public\_hostname=node06.004325440631.aws.testdrive.openshift.com

...

### OpenShift Container Storage for OpenShift Infrastructure

OpenShift infrastructure, like the Registry, Logging and Metrics (introduced in the following paragraphs in this module) have a requirement for reliable storage.  
OpenShift Container Storage can be configured directly in the installer, to provide a separate storage pool just for those workloads. This is a good practice in order to separate failure domains.

By definition you need a separate set of hosts for this - which you have just made available in the previous paragraph.

#### Configure the Installer

Several directives for a second, infrastructure-centric OCS cluster are in the /etc/ansible/hosts file. They have been prepared but commented out using the #ocsinfra\_ prefix.

To see the lines run:

grep '#ocsinfra\_' /etc/ansible/hosts

Remove the #ocsinfra\_ comment prefix by running the below sed command:

sudo sed -i 's/#ocsinfra\_//g' /etc/ansible/hosts

When finished, your inventory file should look like the following:

*/etc/ansible/hosts*

**[OSEv3:children]**

masters

nodes

etcd

glusterfs

glusterfs\_registry

...

**[OSEv3:vars]**

...

openshift\_storage\_glusterfs\_registry\_namespace=infra-storage

openshift\_storage\_glusterfs\_registry\_storageclass=true

openshift\_storage\_glusterfs\_registry\_block\_deploy=true

openshift\_storage\_glusterfs\_registry\_block\_storageclass=true

openshift\_storage\_glusterfs\_registry\_block\_host\_vol\_create=true

openshift\_storage\_glusterfs\_registry\_block\_host\_vol\_size=30

...

**[glusterfs\_registry]**

node04.internal.aws.testdrive.openshift.com glusterfs\_ip=10.0.1.214 glusterfs\_zone=1 glusterfs\_devices='**[ "/dev/xvdd" ]**'

node05.internal.aws.testdrive.openshift.com glusterfs\_ip=10.0.3.230 glusterfs\_zone=2 glusterfs\_devices='**[ "/dev/xvdd" ]**'

node06.internal.aws.testdrive.openshift.com glusterfs\_ip=10.0.3.230 glusterfs\_zone=3 glusterfs\_devices='**[ "/dev/xvdd" ]**'

...

|  |  |
| --- | --- |
|  | Deploys a resource of the OCS cluster for infrastructure in a separate namespace |
|  | Creates a StorageClass for the OCS infra cluster |
|  | Enables support for block storage - the supported storage option for Logging and Metrics |
|  | Creates a StorageClass for the block storage service in the OCS infra cluster |
|  | Automatically create block-hosting volumes (see OCS module for further explanations) |
|  | Allocate a total of 30GiB for block storage based volumes from the OCS infra cluster |
|  | An additional group of hosts which form the OCS infra cluster |
|  | Each line is a node with a device list, consumed by OCS |

#### Install the OCS cluster for OpenShift infrastructure

To illustrate what becomes available with this step, first look at the StorageClass definitions in the system as of now:

oc get sc

There is only a single StorageClass defined, the default OCS cluster that shipped with this installation:

NAME PROVISIONER AGE

glusterfs-storage (default) kubernetes.io/glusterfs 10m

Don’t worry about the concept of the StorageClass - we will explain it in more detail later.

With all required lines uncommented you can start the deployment of the second OCS cluster, dedicated to OpenShift infrastructure workloads:

ansible-playbook /usr/share/ansible/openshift-ansible/playbooks/openshift-glusterfs/config.yml

This playbook takes about 1-2 minutes to execute and will install an entirely independent OCS cluster, including a separate heketi management stack. Additional StorageClasses will be set up to make this storage usable.

Verify by using the oc get sc command that two new StorageClasses are available:

NAME PROVISIONER AGE

glusterfs-registry kubernetes.io/glusterfs 42s

glusterfs-registry-block gluster.org/glusterblock 24s

glusterfs-storage (default) kubernetes.io/glusterfs 15m

|  |  |
| --- | --- |
|  | The StorageClass representing shared file storage from the OCS infra cluster |
|  | The StorageClass representing block storage from the OCS infra cluster |

The block storage service (identified by the gluster.org/glusterblock provisioner) provided by this second OCS cluster will be explained in the following module. For now, it’s only important to know that for OCS serving storage Logging and Metrics, the OCS block storage service is the **only** supported option. The regular file storage service of OCS (identified by the kubernetes.io/glusterfsprovisioner) is **not supported** for Logging and Metrics.

The target use case for these additional storage services is to provide robust, persistent storage for the Registry, Logging and Metrics service - the latter 2 we will set up now.

### OpenShift Metrics

Metrics in OpenShift refers to the continuous collection of performance and utilization data of pods in the cluster. It allows for centralized monitoring in the OpenShift UI and automated horizontal scaling of pods based on utilization.

The metrics implementation is based on [Hawkular](http://www.hawkular.org/), a metrics collection system running on OpenShift persisting data in a Cassandra database.

In your environment metrics is not yet deployed. Configuration is done by customizing the Ansible inventory file /etc/ansible/hosts and deployment is facilitated by running a specific playbook that is part of the openshift-ansible installer. You could have chosen to install the metrics solution when the cluster was initially installed.

#### Configure the Installer

The lines to configure OpenShift Metrics are already configured in the inventory file but commented out with a #metrics\_ prefix.

To see the lines run:

grep '#metrics\_' /etc/ansible/hosts

Remove the #metrics\_ comment prefix by running the below sed command:

sudo sed -i 's/#metrics\_//g' /etc/ansible/hosts

The OpenShift installer variable openshift\_metrics\_install\_metrics=false tells the installer to **not** install the metrics solution when it runs. Remove that line by running the below sed command:

sudo sed -i '/openshift\_metrics\_install\_metrics=false/d' /etc/ansible/hosts

When finished, your inventory file should look like the following:

*/etc/ansible/hosts*

...

**[OSEv3:vars]**

...

openshift\_metrics\_install\_metrics=true

openshift\_metrics\_cassandra\_storage\_type=dynamic

openshift\_metrics\_storage\_volume\_size=10Gi

openshift\_metrics\_hawkular\_hostname=metrics.apps.004325440631.aws.testdrive.openshift.com

openshift\_metrics\_cassanda\_pvc\_storage\_class\_name= glusterfs-registry-block

...

|  |  |
| --- | --- |
|  | Instruct the installer to actually deploy the Metrics service |
|  | Cassandra, part of the Metrics service, will get dynamically provisioned storage |
|  | The resulting PersistentVolumeClaim will be of 10Gi in size |
|  | The name of the StorageClass to use for the PersistentVolumeClaim, makes it use block storage from OCS |
|  | The Metrics frontend (hawkular) will be reachable under this domain. |

#### Install Metrics

There is a specific playbook included with the installer that will handle metrics. It can be run like so:

ansible-playbook /usr/share/ansible/openshift-ansible/playbooks/openshift-metrics/config.yml

This will deploy the metric collection and visualization stack on OpenShift. All resources will be stood up in the openshift-infra **Project**. As part of the deployment, persistent storage will automatically be used for storing the metrics information. It will take roughly 2 minutes to complete.

Once the installation playbook has completed, you can then verify that the metrics components are running in the openshift-infra **Project**:

oc login -u system:admin -n openshift-infra

oc get pods -o wide

It might take a while but after some time you will see something like:

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

hawkular-cassandra-1-gmqv8 1/1 Running 0 4m 10.129.0.19 infra.internal.aws.testdrive.openshift.com <none>

hawkular-metrics-schema-llf7v 0/1 Completed 0 4m 10.129.2.4 infra.internal.aws.testdrive.openshift.com <none>

hawkular-metrics-sv5mb 1/1 Running 0 4m 10.129.0.17 infra.internal.aws.testdrive.openshift.com <none>

heapster-z9lgv 1/1 Running 0 4m 10.129.0.18 infra.internal.aws.testdrive.openshift.com <none>

You will also see the storage for Cassandra being automatically provisioned from the OCS block storage service if you query the PersistentVolumeClaim objects in this project using oc get pvc:

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

metrics-cassandra-1 Bound pvc-e289ba7c-6af6-11e8-af61-02cea7838d26 10Gi RWO glusterfs-registry-block 3m

|  |  |
| --- | --- |
|  | In this lab environment it can take up to 2-3 minutes after the metrics playbook finishes for the metrics stack to finish initialization and for all pods to reach the Ready state. |

In the NODE column you will notice that the **Pods** for Metrics are all located on the infra node. This is because we have set the NodeSelector for all of the metrics components using the following bits of the Ansible hosts file:

openshift\_metrics\_hawkular\_nodeselector={"node-role.kubernetes.io/infra": "true"}

openshift\_metrics\_cassandra\_nodeselector={"node-role.kubernetes.io/infra": "true"}

openshift\_metrics\_heapster\_nodeselector={"node-role.kubernetes.io/infra": "true"}

#### Explore the Metrics UI

If you don’t have it open, return to the OpenShift web console:

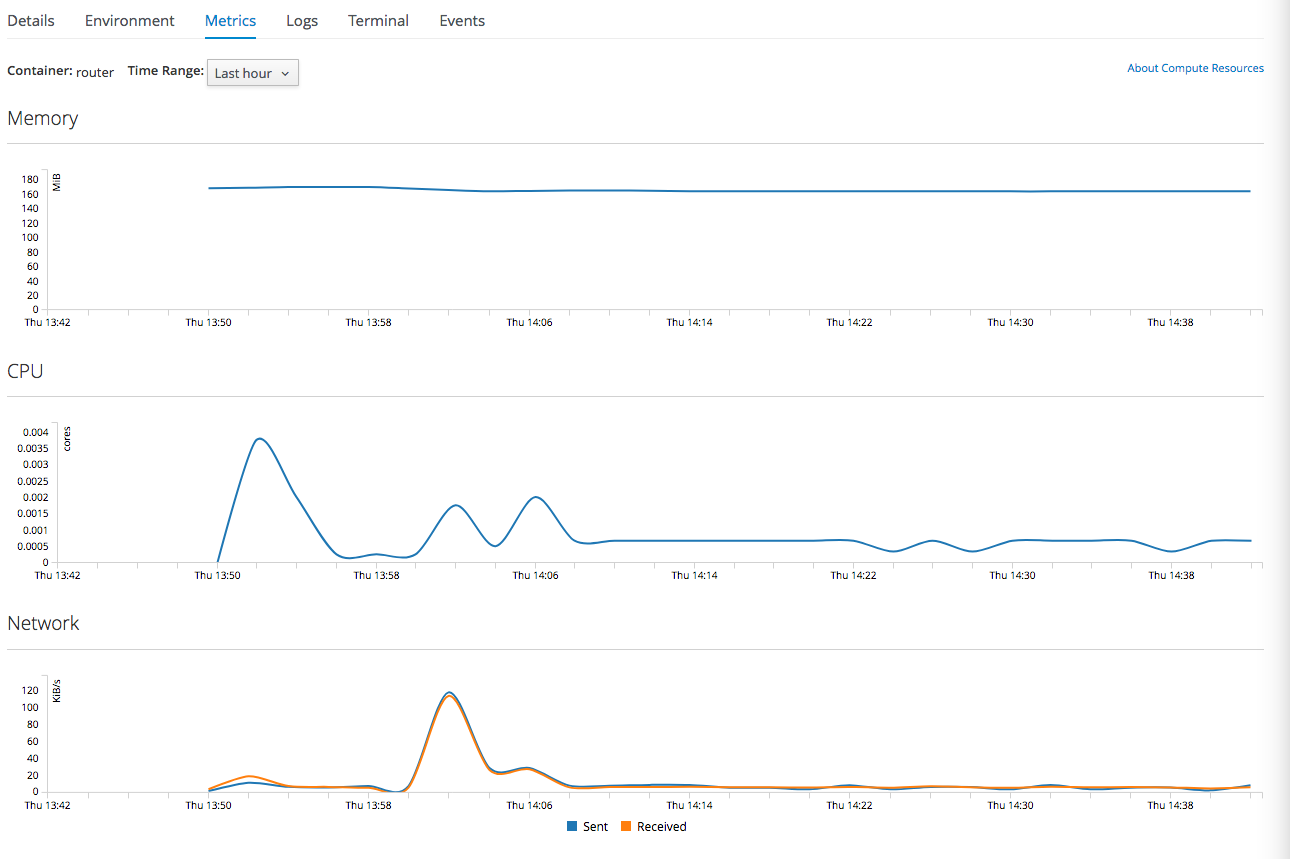
[**https://openshift.004325440631.aws.testdrive.openshift.com/console**](https://openshift.004325440631.aws.testdrive.openshift.com/console)

You will want to be sure you are logged in as fancyuser1 with the password openshift, who is a cluster-reader and can see interesting **Projects**.

Click on the default project.

|  |  |
| --- | --- |
|  | At this point the OpenShift UI will display an error message, stating that the metrics URL could not be reached:  openshift metrics url error  This is because OpenShift generated a self-signed certificate for the Hawkular API. Go ahead and click the metrics URL <https://metrics.apps.004325440631.aws.testdrive.openshift.com/> to access Hawkular and accept the untrusted certificate. Then, return to the OpenShift web console and refresh the page, and the metrics should begin to display.  When working properly and the resource view is expanded, it looks like this:  openshift metrics overview  *Figure 1. The OpenShift UI will show history metrics for applications* |

In the context of a specific **Pod**, the Metrics tab in the UI will show CPU, memory and network throughput for this particular **Pod** with a configurable time-range. Also optionally a donut chart next to a resource appears if the pod was given a consumption limit on this resource (e.g. RAM).



If you want to see interesting metrics, explore the **Project** for metrics itself, openshift-infra.

### OpenShift Logging

Equally important to performance metrics is collecting and aggregating logs from the environments and the application pods it is running. OpenShift ships with an elastic log aggregation solution: **EFK**. **E**lasticSearch, **F**luentd and **K**ibana form a configuration where logs from all nodes and applications are consolidated (Fluentd) in a central place (ElasticSearch) on top of which rich queries can be made from a single UI (Kibana). Administrators can see and search through all logs. Application owners and developers can allow access to logs that belong to their projects. Like metrics the EFK stack runs on top of OpenShift.

#### Configuring the Inventory

The lines to configure OpenShift Logging are already configured in the inventory file but commented out with a #logging\_ prefix.

To see the lines run:

grep '#logging\_' /etc/ansible/hosts

Remove the #logging\_ comment prefix by running the below sed command:

sudo sed -i 's/#logging\_//g' /etc/ansible/hosts

The OpenShift installer variable openshift\_logging\_install\_logging=false tells the installer to **not** install the logging solution when it runs. Remove that line by running the below sed command:

sudo sed -i '/openshift\_logging\_install\_logging=false/d' /etc/ansible/hosts

When finished, your inventory file should look like the following:

*/etc/ansible/hosts*

...

**[OSEv3:vars]**

...

openshift\_logging\_install\_logging=true

openshift\_logging\_es\_pvc\_dynamic=true

openshift\_logging\_es\_pvc\_size=10Gi

openshift\_logging\_es\_pvc\_storage\_class\_name=glusterfs-registry-block

openshift\_logging\_es\_memory\_limit=2G

openshift\_logging\_kibana\_hostname=kibana.apps.004325440631.aws.testdrive.openshift.com

openshift\_logging\_curator\_nodeselector={"node-role.kubernetes.io/infra": "true"}

openshift\_logging\_kibana\_nodeselector={"node-role.kubernetes.io/infra": "true"}

openshift\_logging\_es\_nodeselector={"node-role.kubernetes.io/infra": "true"}

...

|  |  |
| --- | --- |
|  | Trigger the installation of the Logging service |
|  | ElasticSearch, part of the Logging service, will request persistent storage for Logging via a claim toward StorageClass |
|  | The resulting PersistentVolumeClaim will be of 10Gi in size |
|  | The name of the StorageClass to use for the PersistentVolumeClaim |
|  | Limit the required memory for the ElasticSearch pods to 2GB (refer to the [official docs](https://docs.openshift.com/container-platform/3.11/install_config/aggregate_logging_sizing.html) for guidance in production environment) |
|  | The FQDN under which the Logging frontend UI (Kibana) will be available |

#### Install Logging

With these settings in place execute the openshift-logging Ansible playbook that ships as part of the openshift-ansible installer:

ansible-playbook /usr/share/ansible/openshift-ansible/playbooks/openshift-logging/config.yml

Once the installation finishes (roughly 4 minutes), log in as the cluster administrator, using theopenshift-logging **Project**:

oc login -u system:admin -n openshift-logging

Verify the logging stack components are up and running:

oc get pods -o wide

You will see something like:

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

logging-es-data-master-55lp74ix-1-jms4g 2/2 Running 0 1m 10.129.0.24 infra.internal.aws.testdrive.openshift.com <none>

logging-fluentd-2pc7j 1/1 Running 0 1m 10.128.2.4 node02.internal.aws.testdrive.openshift.com <none>

logging-fluentd-6pl9r 1/1 Running 0 1m 10.131.2.4 node05.internal.aws.testdrive.openshift.com <none>

logging-fluentd-7nd2l 1/1 Running 0 1m 10.131.0.4 node01.internal.aws.testdrive.openshift.com <none>

logging-fluentd-gvkbv 1/1 Running 0 1m 10.130.0.6 node03.internal.aws.testdrive.openshift.com <none>

logging-fluentd-ptqvs 1/1 Running 0 1m 10.129.2.5 node04.internal.aws.testdrive.openshift.com <none>

logging-fluentd-qb42p 1/1 Running 0 1m 10.130.2.6 node06.internal.aws.testdrive.openshift.com <none>

logging-fluentd-tdczj 1/1 Running 0 1m 10.128.0.6 master.internal.aws.testdrive.openshift.com <none>

logging-fluentd-tn9ww 1/1 Running 0 1m 10.129.0.22 infra.internal.aws.testdrive.openshift.com <none>

logging-kibana-1-b54pv 2/2 Running 0 2m 10.129.0.21 infra.internal.aws.testdrive.openshift.com <none>

The Fluentd **Pods** are deployed as part of a **DaemonSet**, which is a mechanism to ensure that specific **Pods** run on specific **Nodes** in the cluster at all times:

oc get daemonset

You will see something like:

NAME DESIRED CURRENT READY UP-TO-DATE AVAILABLE NODE SELECTOR AGE

logging-fluentd 8 8 8 8 8 logging-infra-fluentd=true 3m

You will also see the storage for ElasticSearch being automatically provisioned from the OCS block storage service if you query the PersistentVolumeClaim objects in this project

oc get pvc

And you will see something like:

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

logging-es-0 Bound pvc-8188d8dd-6af7-11e8-af61-02cea7838d26 10Gi RWO glusterfs-registry-block 3m

|  |  |
| --- | --- |
|  | Much like with the Metrics solution, we defined the appropriate NodeSelector in the Logging configuration to ensure that the Logging components only landed on the infra node. That being said, the DaemonSet ensures FluentD runs on **all** nodes. Otherwise we would not capture all of the container logs. |

To reach the Kibana user interface, first determine its public access URL by querying the **Route** that got set up to expose Kibana’s **Service**:

oc get route/logging-kibana

You will see something like:

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

logging-kibana kibana.apps.apps.004325440631.aws.testdrive.openshift.com logging-kibana <all> reencrypt/Redirect None

You can click the link ( [https://kibana.apps.004325440631.aws.testdrive.openshift.com](https://kibana.apps.004325440631.aws.testdrive.openshift.com/) ) to open the Kibana interface. There is a special authentication proxy that is configured as part of the EFK installation that results in Kibana requiring OpenShift credentials for access. You should login to Kibana as the fancyuser1 user with password openshift to be able to see all of the cluster’s logs. Kibana utilizes the same RBAC underpinning OpenShift to ensure that users can only see the logs they should have access to.

|  |  |
| --- | --- |
|  | The block-storage service of OCS (also referred to as gluster-block, introduced in the next chapter) is **only** supported for Logging and Metrics as of this release. This is about to change in the near future as we qualify more workloads. |

### OpenShift Multitenant Networking

OpenShift has a software defined network (SDN) inside the platform that is based on Open vSwitch. This SDN is used to provide connectivity between application components inside of the OpenShift environment. It comes with default network ranges pre-configured, although you can make changes to these should they conflict with your existing infrastructure, or for whatever other reason you may have.

When you installed OpenShift, there was an option set in the installer’s configuration to enable the multitenant network plugin:

os\_sdn\_network\_plugin\_name=redhat/openshift-ovs-multitenant

The OpenShift Multitenant SDN plug-in enables a true isolated multi-tenant network infrastructure inside OpenShift’s software defined network. While you have seen projects isolate resources through OpenShift’s RBAC, the multitenant SDN plugin isolates projects using separate virtual network IDs within Open vSwitch.

The multitenant network plugin was introduced in OpenShift 3.1, and more information about it and its configuration can be found in the [networking documentation](https://docs.openshift.com/container-platform/3.11/architecture/networking/sdn.html). Additionally, other vendors are working with the upstream Kubernetes community to implement their own SDN plugins, and several of these are supported by the vendors for use with OpenShift. These plugin implementations make use of appc/CNI, which is outside the scope of this lab.

#### Execute the Creation Script

Only users with cluster administration privileges can manipulate **Project** networks. First, make sure you are logged in as the cluster administrator:

oc login -u system:admin

Then, execute a script that we have prepared for you. It will create two **Projects** and then deploy a **DeploymentConfig** with a **Pod** for you:

bash /opt/lab/support/net-proj.sh

#### Examine Network Namespaces

Two **Projects** were created for you, netproj-a and netproj-b. Execute the following command to see the network namespaces:

oc get netnamespaces

You will see something like the following:

**NAME** **NETID** **EGRESS** **IPS**

**app-management** 10765501 []

**default** 0 []

**infra-storage** 12129484 []

**kube-public** 3885278 []

**kube-system** 7128412 []

**management-infra** 15734027 []

**netproj-a** 11910878 []

**netproj-b** 6478895 []

...

Note that each project has its own network namespace with a unique ID. The default project is a special exception. Its network ID is 0. This network is a global network. It is joined (not isolated) to all other networks in the SDN by default. If you remember from earlier exercises, the OpenShift router and the image registry are both in the default project. This means that **Pods** in all other projects can access them. That’s good, because the router needs to be able to proxy traffic to the **Pods** to make them accessible from outside of OpenShift.

#### Test Connectivity

Now that you have some networks and pods, you will need to find the IP address of the pod in the netproj-b **Project**. Make sure that the pod is Running in the netproj-b namespace:

oc get pod -n netproj-b

Once it’s running, the following command will show you the IP address:

bash /opt/lab/support/podbip.sh

The output will simply be the IP address of the pod in the netproj-b project. The everyday way to do this would be with a combination of the get and describe verbs. Feel free to do the following to verify what the script did:

oc get pod -n netproj-b

oc describe pod ose-1-f0deb

Make sure to substitute the correct pod name in the describe command.

describe will show you a lot of information about the pod, including its IP address on the software defined network. Either way, make note of the IP address you found above. It will look something like 10.1.4.12.

Export the IP address of your pod into a shell variable like so:

export POD\_B\_IP=10.1.4.12

Make sure to use the correct IP address that you saw earlier in the command output.

The OpenShift command-line tool and the web console provide mechanisms to execute commands inside **Pods** running in the environment. This is a useful feature for both developers as well as for cluster and application operators/administrators. You will use that feature in order to test network connectivity between the two **Pods** you created.

Get the name of the **Pod** running in the netproj-a **Project**:

oc get pods -n netproj-a

Then, export the **Pod** ID as a shell variable:

export POD\_A\_NAME=ose-1-q9mt5

Be sure to use the name that you saw in the output of your command.

Now, go ahead and exec a ping command inside **Pod** A, trying to reach **Pod** B:

oc exec -n netproj-a $POD\_A\_NAME -- ping -c1 -W1 $POD\_B\_IP

Your ping output should look like the following:

PING 10.129.0.10 (10.129.0.10) 56(84) bytes of data.

--- 10.129.0.10 ping statistics ---

1 packets transmitted, 0 received, 100% packet loss, time 0ms

You will see 100% packet loss (your ping command sends 1 packet, waits 1 second, and gets no response). This is because the networks are not connected to one another. Now simply execute the following:

ping -c1 -W1 $POD\_B\_IP

You will see a successful ping. This is because the master (the system you are on) is also a node attached to the SDN. At the host level you are able to reach across all networks, virtual or otherwise. This is important to keep in mind when you consider the overall network-level security of your cluster. Someone logged in to an OpenShift host can "see" and touch everything on the SDN.

#### Join the Networks

Now it’s time to join the networks. Execute the following:

oc get netnamespace

Take note of the network IDs for netproj-a and netproj-b. Then:

oc adm pod-network join-projects netproj-a --to=netproj-b

And then look at the network IDs again:

oc get netnamespace

You should see that the network IDs of the two projects are now the same.

#### Retest Connectivity

Now that the projects are joined, your ping between the pods should work. Execute the original pingtest again:

oc exec -n netproj-a $POD\_A\_NAME -- ping -c1 -W1 $POD\_B\_IP

This time, your packet should reach its destination:

PING 10.129.0.10 (10.129.0.10) 56(84) bytes of data.

64 bytes from 10.129.0.10: icmp\_seq=1 ttl=64 time=1.07 ms

--- 10.129.0.10 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time 0ms

rtt min/avg/max/mdev = 1.075/1.075/1.075/0.000 ms

#### Isolate Projects

Now, go ahead and isolate (unjoin) the projects, and then run your ping again:

oc adm pod-network isolate-projects netproj-a

oc exec -n netproj-a $POD\_A\_NAME -- ping -c1 -W1 $POD\_B\_IP

You should see that your ping fails again.

Network multitenancy is a bit of a blunt tool. You can either give total access between two projects, or completely restrict access. Don’t fret, though. If you need finer-grained control of inter-**Pod** and **Service**communication, there is a Tech Preview network implementation called NetworkPolicy. You can learn more about it in the [product documentation](https://docs.openshift.com/container-platform/3.11/admin_guide/managing_networking.html#admin-guide-networking-networkpolicy).

### Node Maintenance

It is possible to put any node of the OpenShift environment into maintenance by marking it as non-schedulable followed by a drain of all pods on the node.

These operations require elevated privileges. Ensure you are logged in as cluster admin:

oc login -u system:admin

You will see by now that there are pods running on almost all of your nodes:

oc get pods --all-namespaces -o wide

Sometimes you might need to perform maintenance on a host. Let’s take a look at the **Pods** that are on node02:

oc adm manage-node --list-pods node02.internal.aws.testdrive.openshift.com

Firstly, we probably want to ensure that no new workload can be put on this host. Mark node node02.internal.aws.testdrive.openshift.com as non-schedulable to prevent the schedulers in the system to place any new workloads on it:

oc adm manage-node node02.internal.aws.testdrive.openshift.com --schedulable=false

The output of the command will show that the node is now not schedulable:

NAME STATUS ROLES AGE VERSION

node02.internal.aws.testdrive.openshift.com Ready,SchedulingDisabled compute 1h v1.11.0+d4cacc0

Marking the node as non-schedulable did not impact the pods it is running. List those pods:

oc adm manage-node node02.internal.aws.testdrive.openshift.com --list-pods

Other than a **Pod** for Container Native Storage and a Fluentd instance (there is one on every node), there may or may not be other **Pods** running on this node.

The next step is to drain the **Pods** to other nodes in the cluster.

|  |  |
| --- | --- |
|  | **Pods** running on the node as part of a DaemonSet like those associated to Logging or OCS will **not** be drained. They will not be accessible anymore through OpenShift, but will continue to run as containers on the nodes until the local OpenShift services are stopped and/or the node is shutdown. This is not a problem since software like OCS or the OpenShift Metrics stack is designed to handle such situations transparently. |

Start the drain process like this:

oc adm drain node02.internal.aws.testdrive.openshift.com --ignore-daemonsets

After a few moments, all of the **Pods**, except those for Fluentd, Container Native Storage, and Prometheus previously running on node02.internal.aws.testdrive.openshift.com should have terminated and been launched elsewhere.

oc adm manage-node node02.internal.aws.testdrive.openshift.com --list-pods

The node node02.internal.aws.testdrive.openshift.com is now ready for an administrator to start maintenance operations. If those include a reboot of the system or upgrading OpenShift components, the **Pods** associated with OCS and logging will come back up automatically.

Now that our maintenance is complete, the node is still non-schedulable. Let’s fix that:

oc adm manage-node node02.internal.aws.testdrive.openshift.com --schedulable=true

Now the node will be able to have workload scheduled on it again:

NAME STATUS ROLES AGE VERSION

node02.internal.aws.testdrive.openshift.com Ready compute 1h v1.11.0+d4cacc0

### Running the OpenShift Registry with OCS

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 three to achieve high-availability for all infrastructure services. |

#### Adding OCS 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 OCS. 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 set volume command. Execute the following:

oc set volume dc/docker-registry --add --name=registry-storage -t pvc \

--claim-mode=ReadWriteMany --claim-size=5Gi \

--claim-name=registry-storage --claim-class=glusterfs-registry --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 will --overwrite the existing volume with our new PVC. More information can be found in the [volumes documentation](https://docs.openshift.com/container-platform/3.11/dev_guide/volumes.html). |
|  | It is also possible to use openshift-ansible to deploy the registry |

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

Wait until you see the following state:

NAME REVISION DESIRED CURRENT TRIGGERED BY

docker-registry 2 1 1 config

Now your OpenShift Registry is using persistent storage provided by OCS. 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

Check the registry’s DeploymentConfig to verify it indeeds mounts a PersistentVolume to the /registry directory which is where the registry stores all container images:

oc describe dc docker-registry

This should show:

Name: docker-registry

Namespace: default

Created: 2 hours ago

Labels: docker-registry=default

Annotations: <none>

Latest Version: 2

Selector: docker-registry=default

Replicas: 3

Triggers: Config

Strategy: Rolling

Template:

Pod Template:

Labels: docker-registry=default

Service Account: registry

Containers:

registry:

Image: support.internal.aws.testdrive.openshift.com:5000/openshift3/ose-docker-registry:v3.11.16

Port: 5000/TCP

Requests:

cpu: 100m

memory: 256Mi

Liveness: http-get https://:5000/healthz delay=10s timeout=5s period=10s #success=1 #failure=3

Readiness: http-get https://:5000/healthz delay=0s timeout=5s period=10s #success=1 #failure=3

Environment:

REGISTRY\_HTTP\_ADDR: :5000

REGISTRY\_HTTP\_NET: tcp

REGISTRY\_HTTP\_SECRET: g4fMc23QUZLFhRmtu7m7mCah5bhefi3h2sBPbjgJvdw=

REGISTRY\_MIDDLEWARE\_REPOSITORY\_OPENSHIFT\_ENFORCEQUOTA: false

REGISTRY\_OPENSHIFT\_SERVER\_ADDR: docker-registry.default.svc:5000

REGISTRY\_HTTP\_TLS\_KEY: /etc/secrets/registry.key

REGISTRY\_HTTP\_TLS\_CERTIFICATE: /etc/secrets/registry.crt

Mounts:

/etc/secrets from registry-certificates (rw)

/registry from registry-storage (rw)

Volumes:

registry-certificates:

Type: Secret (a volume populated by a Secret)

SecretName: registry-certificates

Optional: false

registry-storage:

Type: PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)

ClaimName: registry-storage

ReadOnly: false

Deployment #2 (latest):

Name: docker-registry-2

Created: 48 seconds ago

Status: Complete

Replicas: 3 current / 3 desired

Selector: deployment=docker-registry-2,deploymentconfig=docker-registry,docker-registry=default

Labels: docker-registry=default,openshift.io/deployment-config.name=docker-registry

Pods Status: 3 Running / 0 Waiting / 0 Succeeded / 0 Failed

Deployment #1:

Created: 2 hours ago

Status: Complete

Replicas: 0 current / 0 desired

Events:

Type Reason Age From Message

---- ------ ---- ---- -------

Normal DeploymentCreated 48s deploymentconfig-controller Created new replication controller "docker-registry-2" for version 2

Normal ReplicationControllerScaled 29s deploymentconfig-controller Scaled replication controller "docker-registry-2" from 1 to 3

|  |  |
| --- | --- |
|  | The /registry directory in the pod namespace will be a mountpoint for a PersistentVolume called registry-storage |
|  | The definition for the volume registry-storage |
|  | The volume will be of the type PersistentVolume which is referenced to a PersistentVolumeClaim |
|  | The name of the PersistentVolumeClaim which this volume references |

OpenShift Container Storage Concepts

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

How OpenShift Container Storage runs

Make sure you are logged on as the super user in the storage:

oc login -u system:admin -n storage

OpenShift Container Storage is GlusterFS running in containers, specifically in pods managed by OpenShift. We have looked at the pods making up the storage cluster already in the introduction chapter. Go ahead and switch to the storage project:

oc project storage

Then, take a look at the storage **Pods**:

oc get pods -o wide

Which yields:

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

glusterfs-storage-7qzsm 1/1 Running 0 2h 10.0.3.252 node01.internal.aws.testdrive.openshift.com <1> <none>

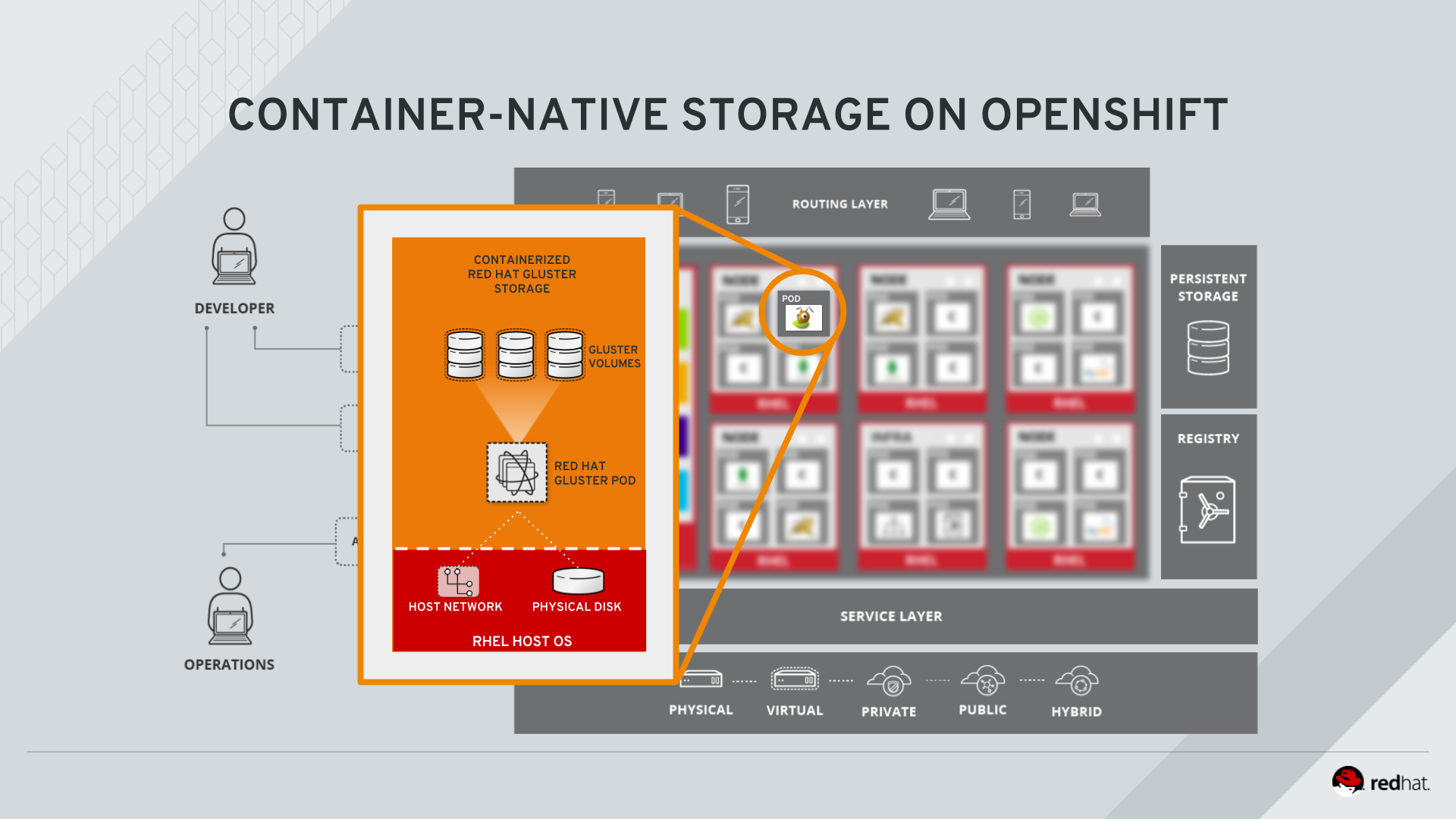
glusterfs-storage-7rds5 1/1 Running 0 2h 10.0.1.238 node02.internal.aws.testdrive.openshift.com <1> <none>

glusterfs-storage-x7chr 1/1 Running 0 2h 10.0.4.221 node03.internal.aws.testdrive.openshift.com <1> <none>

heketi-storage-1-bxqr2 1/1 Running 0 2h 10.131.0.6 infra.internal.aws.testdrive.openshift.com <2> <none>

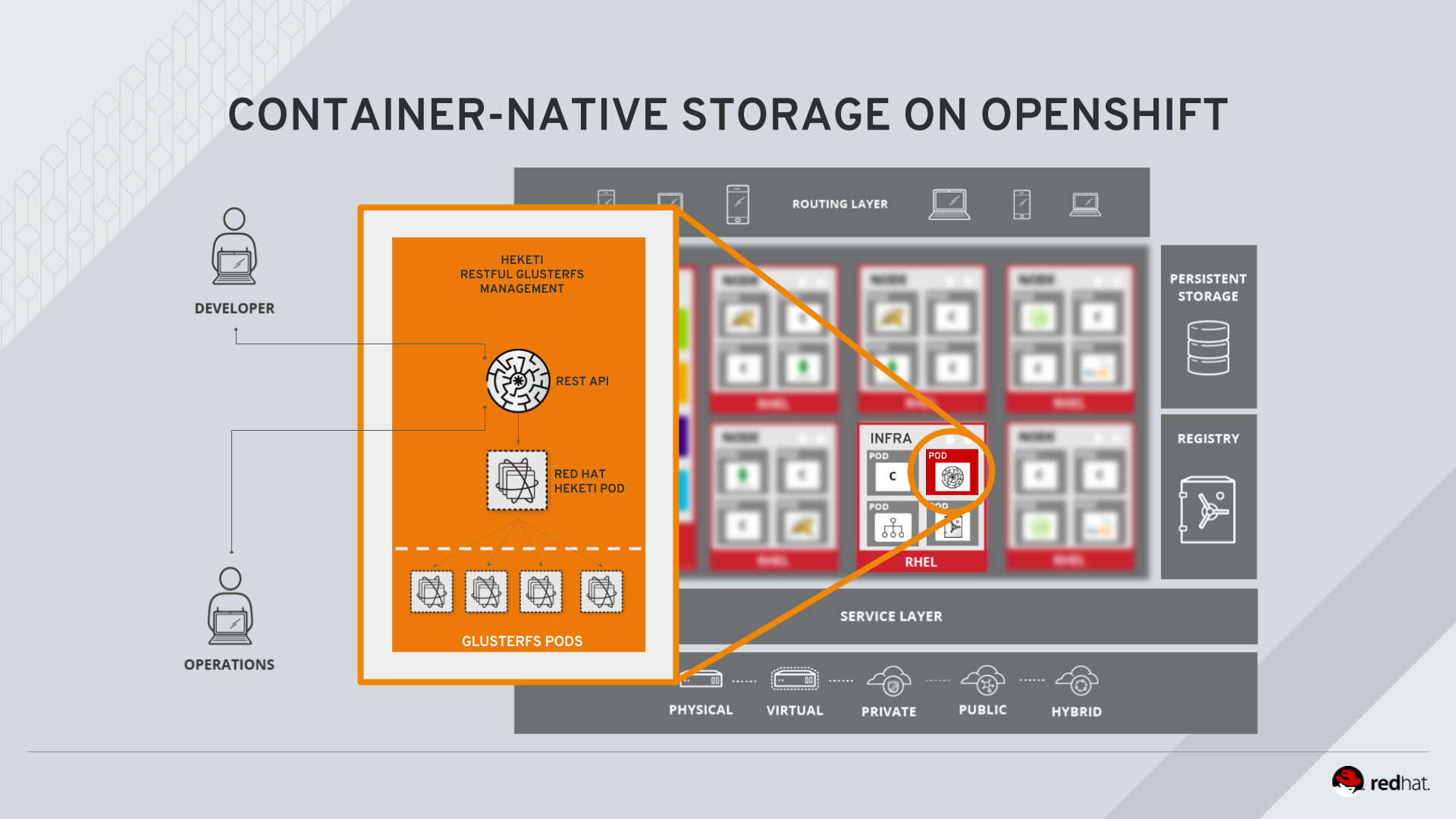
|  |  |
| --- | --- |
|  | OCS **Pods**, with each of the designated nodes running exactly one. |
|  | heketi API frontend pod |
|  | | The exact **pod** names will be different in your environment, since they are auto-generated. Also the heketi **pod** might run on any node. |

The OCS **Pods** use the host’s network and block devices to run the software-defined storage system. See schematic below for a visualization.



*Figure 1. GlusterFS pods in OCS in detail.*

heketi is a component that exposes an API to the storage system for OpenShift. This allows OpenShift to dynamically allocate storage from OCS in a programmatic fashion. See below for a visualization. Note that for simplicity, in our example heketi runs on the OpenShift application nodes, not on the infrastructure node.



*Figure 2. heketi pod running in OCS*

Examine heketi

To expose heketi’s API outside of OpenShift for administrators (for monitoring and maintenance), a **Service** named *heketi-storage* and a **Route** has been set up:

oc get service,route

You will see something like:

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE

svc/heketi-db-storage-endpoints ClusterIP 172.30.228.77 <none> 1/TCP 2h

svc/heketi-storage ClusterIP 172.30.54.191 <none> 8080/TCP 2h

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

route.route.openshift.io/heketi-storage heketi-storage-storage.apps.004325440631.aws.testdrive.openshift.com heketi-storage <all> None

You may verify external availability of this API and heketi being alive with a trivial health check:

curl -w "\n" http://heketi-storage-storage.apps.004325440631.aws.testdrive.openshift.com/hello

This should return:

Hello from Heketi

This how the heketi API is made available to both external clients, like heketi-cli which we examined in the introduction. But mainly it is leveraged by OpenShift to provision storage dynamically. Let’s look at this use case.

A Simple OCS 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 application ships in the 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

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 created in the YAML. Run the following command to grep the relavant part:

oc get template/rails-pgsql-persistent -n openshift -o yaml | grep PersistentVolumeClaim -A8

This shows the basic structure of a PersistentVolumeClaim:

kind: PersistentVolumeClaim

metadata:

name: ${DATABASE\_SERVICE\_NAME}

spec:

accessModes:

- ReadWriteOnce

resources:

requests:

storage: ${VOLUME\_CAPACITY}

This will request a **PersistentVolume** in RWO mode. Storage provided in this mode can only be mounted by a single pod at a time. For a database that is usually what you want. The requested capacity under spec.resources.requests.storage is coming in via a parameter when the template is parsed. This is how storage is *requested*.

Using persistent storage is done via a PersistentVolume provided in response to this PersistentVolumeClaim. A PersistentVolume is a representation of some physical storage capacity provisioned by the backing storage system. It will supply the PostgreSQL pod with persistent storage on the mount point /var/lib/pgsql/data.

You can see this when inspecting how the pod is described as part of the DeploymentConfig:

oc get template/rails-pgsql-persistent -n openshift -o yaml | grep mountPath -B58 -A5

Will show:

- apiVersion: v1

kind: DeploymentConfig

metadata:

annotations:

description: Defines how to deploy the database

template.alpha.openshift.io/wait-for-ready: "true"

name: ${DATABASE\_SERVICE\_NAME}

spec:

replicas: 1

selector:

name: ${DATABASE\_SERVICE\_NAME}

strategy:

type: Recreate

template:

metadata:

labels:

name: ${DATABASE\_SERVICE\_NAME}

name: ${DATABASE\_SERVICE\_NAME}

spec:

containers:

- env:

- name: POSTGRESQL\_USER

valueFrom:

secretKeyRef:

key: database-user

name: ${NAME}

- name: POSTGRESQL\_PASSWORD

valueFrom:

secretKeyRef:

key: database-password

name: ${NAME}

- name: POSTGRESQL\_DATABASE

value: ${DATABASE\_NAME}

- name: POSTGRESQL\_MAX\_CONNECTIONS

value: ${POSTGRESQL\_MAX\_CONNECTIONS}

- name: POSTGRESQL\_SHARED\_BUFFERS

value: ${POSTGRESQL\_SHARED\_BUFFERS}

image: ' '

livenessProbe:

initialDelaySeconds: 30

tcpSocket:

port: 5432

timeoutSeconds: 1

name: postgresql

ports:

- containerPort: 5432

readinessProbe:

exec:

command:

- /bin/sh

- -i

- -c

- psql -h 127.0.0.1 -U ${POSTGRESQL\_USER} -q -d ${POSTGRESQL\_DATABASE}

-c 'SELECT 1'

initialDelaySeconds: 5

timeoutSeconds: 1

resources:

limits:

memory: ${MEMORY\_POSTGRESQL\_LIMIT}

volumeMounts:

- mountPath: /var/lib/pgsql/data

name: ${DATABASE\_SERVICE\_NAME}-data

volumes:

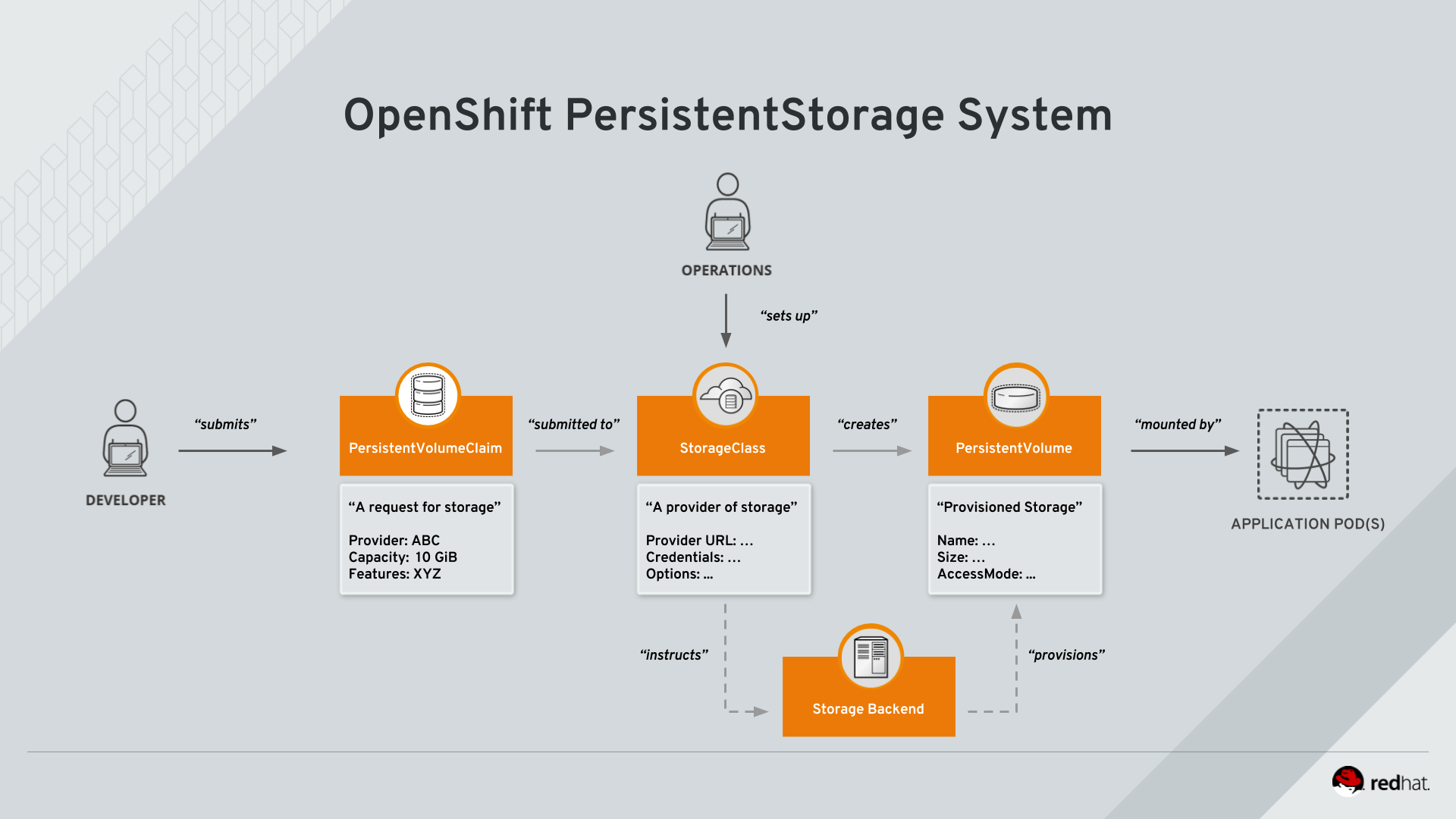
- name: ${DATABASE\_SERVICE\_NAME}-data

persistentVolumeClaim:

claimName: ${DATABASE\_SERVICE\_NAME}

|  |  |
| --- | --- |
|  | The mount path where the persistent storage should appear inside the container |
|  | The name of the volume known by the container |
|  | The PersistentVolumeClaim from which this volume should come from |
|  | | In the above snippet you see there are even more parameters in this template. 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 following diagram sums up how storage get’s provisioned in OpenShift and depicts the relationship of PersistentVolumes, PersistentVolumeClaims and StorageClasses:



*Figure 3. OpenShift Persistent Volume Framework*

Let’s try it out. The storage size parameter in the template is called VOLUME\_CAPACITY. The new-app command will again 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-apptries 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-database-app [2/1622]

Rails + PostgreSQL

---------

An example Rails application with a PostgreSQL database. For more information about using this template, including OpenShift considerations, see https://github.com/openshift/rails-ex/blob/master/README.md.

The following service(s) have been created in your project: rails-pgsql-persistent, postgresql.

For more information about using this template, including OpenShift considerations, see https://github.com/openshift/rails-ex/blob/master/README.md.

\* 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=pIXDthfeGR7PHxxbASEjCM7jQ0hAJ8Ph8HTIttvl # generated

\* Secret Key=ij54gqv7w04habvy6dn2sninbbdgmlicwnsvpfwa1gdn6of2rrxgo211njqaekqlhg1503xdnvo2oc7h3dk7dd3cmk7h8mvnmijikovjw5jnl2w2pnfrukkwx0sq0uj # generated

\* Application Username=openshift

\* Application Password=secret

\* Rails Environment=production

\* Database Service Name=postgresql

\* Database Username=userAFJ # generated

\* Database Password=pn6A2x3B # 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.route.openshift.io "rails-pgsql-persistent" created

imagestream.image.openshift.io "rails-pgsql-persistent" created

buildconfig.build.openshift.io "rails-pgsql-persistent" created

deploymentconfig.apps.openshift.io "rails-pgsql-persistent" created

persistentvolumeclaim "postgresql" created

service "postgresql" created

deploymentconfig.apps.openshift.io "postgresql" created

--> Success

Access your application via route 'rails-pgsql-persistent-my-database-app.apps.790442527540.aws.testdrive.openshift.com'

Build scheduled, use 'oc logs -f bc/rails-pgsql-persistent' to track its progress.

Run 'oc status' to view your app.

Go back to the OpenShift web console:

[**https://openshift.004325440631.aws.testdrive.openshift.com/console**](https://openshift.004325440631.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 here. The deployment is complete when both the PostgreSQL pod and the Ruby application pod have one healthy instance (rings are dark, solid blue).

|  |  |
| --- | --- |
|  | It may take up to 5 minutes for the deployment to complete. |

On the CLI, you should now see a PVC that has been issued and has a status of *Bound*. state.

oc get pvc

You will see something like:

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

postgresql Bound pvc-6de8449e-3f34-11e8-87ea-0298f449cc4c 5Gi RWO glusterfs-storage 4m

Alternatively, in the web console, check the **"Storage"** menu.

|  |  |
| --- | --- |
|  | This PVC has been automatically fulfilled by OCS because the glusterfs-storage **StorageClass** was set up as the system-wide default as part of the installation. The responsible parameter in the inventory file was: openshift\_storage\_glusterfs\_storageclass\_default=true |

Try the Application

Now go ahead and try out the application. The overview page in the OpenShift web console 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 PATH SERVICES PORT TERMINATION WILDCARD

rails-pgsql-persistent rails-pgsql-persistent-my-database-app.apps.004325440631.aws.testdrive.openshift.com rails-pgsql-persistent <all> None

Following this output, point your browser to:

[**http://rails-pgsql-persistent-my-database-app.apps.004325440631.aws.testdrive.openshift.com/articles**](http://rails-pgsql-persistent-my-database-app.apps.004325440631.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 its table spaces on a GlusterFS volume provided by OCS.

|  |  |
| --- | --- |
|  | This application’s template included a **Route** object definition, which is why the **Service** was automatically exposed. This is a good practice. Note how the actual application is hosted under the **/articles** path of the URL. |

Explore the underlying OCS 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 the PVC in a BOUND state and the name of the PV it has been bound to in the VOLUME column:

NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE

postgresql Bound pvc-6de8449e-3f34-11e8-87ea-0298f449cc4c 5Gi RWO glusterfs-storage 144m

|  |  |
| --- | --- |
|  | Your PV name will be different as it’s dynamically generated. A lot of the following things contain dynamically generated names. **Use the supplied bash shortcuts to easy copying and pasting.** |

Here’s a little bash shortcut to store the name of the PVC in a Bash environment variable:

export PGSQL\_PV\_NAME=$(oc get pvc/postgresql -o jsonpath="{.spec.volumeName}" -n my-database-app)

echo $PGSQL\_PV\_NAME

Look at the details of the PV bound to the PVC, in this case pvc-6de8449e-3f34-11e8-87ea-0298f449cc4c (your’s will be different, use the bash variable):

oc describe pv $PGSQL\_PV\_NAME

You will see something like:

Name: pvc-6de8449e-3f34-11e8-87ea-0298f449cc4c

Labels: <none>

Annotations: Description=Gluster-Internal: Dynamically provisioned PV

gluster.kubernetes.io/heketi-volume-id=7da624d82941c50d704dd01b366c5806

gluster.org/type=file

kubernetes.io/createdby=heketi-dynamic-provisioner

pv.beta.kubernetes.io/gid=2001

pv.kubernetes.io/bound-by-controller=yes

pv.kubernetes.io/provisioned-by=kubernetes.io/glusterfs

volume.beta.kubernetes.io/mount-options=auto\_unmount

Finalizers: [kubernetes.io/pv-protection]

StorageClass: glusterfs-storage

Status: Bound

Claim: my-database-app/postgresql

Reclaim Policy: Delete

Access Modes: RWO

Capacity: 5Gi

Node Affinity: <none>

Message:

Source:

Type: Glusterfs (a Glusterfs mount on the host that shares a pod's lifetime)

EndpointsName: glusterfs-dynamic-postgresql

Path: vol\_e8fe7f46fedf7af7628feda0dcbf2f60

ReadOnly: false

Events: <none>

|  |  |
| --- | --- |
|  | The unique name of this PV in the system OpenShift refers to |
|  | The unique volume name backing the PV known to GlusterFS |

Note the GlusterFS volume name, in this case **vol\_e8fe7f46fedf7af7628feda0dcbf2f60**. The following is another Bash shortcut to store the name of the GlusterFS volume backing the PersistentVolume:

export PGSQL\_GLUSTER\_VOLUME=$(oc get pv $PGSQL\_PV\_NAME -o jsonpath='{.spec.glusterfs.path}')

echo $PGSQL\_GLUSTER\_VOLUME

Now let’s switch to the namespace we used for OCS deployment:

oc project storage

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

oc get pods -o wide -l glusterfs=storage-pod

You will see something like:

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

glusterfs-storage-7qzsm 1/1 Running 0 2h 10.0.3.252 node01.internal.aws.testdrive.openshift.com <none>

glusterfs-storage-7rds5 1/1 Running 0 2h 10.0.1.238 node02.internal.aws.testdrive.openshift.com <none>

glusterfs-storage-x7chr 1/1 Running 0 2h 10.0.4.221 node03.internal.aws.testdrive.openshift.com <none>

We are now going to select the first pod (which one doesn’t really matter) and, store it’s IP address in above example that is: **10.0.1.114** of pod **glusterfs-storage-37vn8**.

Again, for easy copying and pasting, here are some Bash shortcuts:

export FIRST\_GLUSTER\_POD=$(oc get pods -o jsonpath='{.items[0].metadata.name}' -l glusterfs=storage-pod)

export FIRST\_GLUSTER\_IP=$(oc get pods -o jsonpath='{.items[0].status.podIP}' -l glusterfs=storage-pod)

echo $FIRST\_GLUSTER\_POD

echo $FIRST\_GLUSTER\_IP

We will again use the oc rsh facility to log on to the selected GlusterFS pod which has the GlusterFS CLI utilities installed. This time we will use the non-interactive mode which immediately drops out after executing the supplied command.

Query GlusterFS from inside the first GlusterFS pod for all known volumes:

oc rsh $FIRST\_GLUSTER\_POD gluster volume list

You will immediately drop back out to your shell and you will see something like:

heketidbstorage

vol\_e8fe7f46fedf7af7628feda0dcbf2f60

vol\_5e1cd71070734a3b02f58d822f89486a

vol\_f2e8fda1d42a41efabbb4d4a3b4a5659

|  |  |
| --- | --- |
|  | A special volume dedicated to heketi’s internal database. |
|  | The volume backing the PV of the PostgreSQL database we asked you to remember. |

Query GlusterFS about the topology of this volume:

oc rsh $FIRST\_GLUSTER\_POD gluster volume info $PGSQL\_GLUSTER\_VOLUME

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: 10.0.3.106:/var/lib/heketi/mounts/vg\_63b05bee6695ee5a63ad95bfbce43bf7/brick\_aa28de668c8c21192df55956a822bd3c/brick

Brick2: 10.0.1.114:/var/lib/heketi/mounts/vg\_0246fd563709384a3cbc3f3bbeeb87a9/brick\_684a01f8993f241a92db02b117e0b912/brick

Brick3: 10.0.4.79:/var/lib/heketi/mounts/vg\_5a8c767e65feef7455b58d01c6936b83/brick\_25972cf5ed7ea81c947c62443ccb308c/brick

Options Reconfigured:

transport.address-family: inet

nfs.disable: on

performance.client-io-threads: off

cluster.brick-multiplex: on

|  |  |
| --- | --- |
|  | 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, and therefore across all your OpenShift App nodes running OCS. Data written to such a replica volume is replicated 3 times to all **bricks**. **Bricks** are local storage in GlusterFS nodes, usually backed by a local SAS \*disk or NVMe device. Each node exposes its local storage via the GlusterFS \*protocol. The brick itself is simply a directory on a block device formatted \*with XFS. Hence you can look with a simple ls command and see how the data \*is actually stored in each brick.

For easy copying and pasting, here’s another bash shortcut to extract the brick directory path of our PostgreSQL volume from the fist GlusterFS pod in the list:

export PGSQL\_GLUSTER\_BRICK=$(echo -n $(oc rsh $FIRST\_GLUSTER\_POD gluster vol info $PGSQL\_GLUSTER\_VOLUME | grep $FIRST\_GLUSTER\_IP) | cut -d ':' -f 3 | tr -d $'\r' )

echo $PGSQL\_GLUSTER\_BRICK

You can look at the brick directory of the first GlusterFS pod and see how GlusterFS stores the files from the clients in a brick:

oc rsh $FIRST\_GLUSTER\_POD ls -ahl $PGSQL\_GLUSTER\_BRICK

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

Dig a bit deeper, try looking at the userdata folder:

oc rsh $FIRST\_GLUSTER\_POD ls -ahl $PGSQL\_GLUSTER\_BRICK/userdata

You will see the PostgreSQL database folder structure:

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

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 set of replicated brick storage via the GlusterFS protocol. Which abstracts the 3-way replication behind a single FUSE mount point - this is called a volume in GlusterFS. 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.

Providing Scalable, Shared Storage With OCS

Historically very few options, like basic NFS support, existed to provide a **PersistentVolume** to more than one container at a time. The access mode used **for this in OpenShift is ReadWriteMany. Traditional block-based storage solutions are not able to provide \*PersistentVolumes** with this access mode.

Also, once provisioned, most storage cannot easily be resized.

With OCS these capabilities are now available to all OpenShift deployments, no matter where they are deployed. To illustrate the benefit of this, we will deploy a PHP file uploader application that has multiple front-end instances sharing a common storage repository.

Deploy the File Uploader 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 PHP application called file-uploader:

oc new-app openshift/php:7.1~https://github.com/christianh814/openshift-php-upload-demo --name=file-uploader

You will see something like:

--> Found image 691930e (5 weeks old) in image stream "openshift/php" under tag "7.1" for "openshift/php:7.1"

Apache 2.4 with PHP 7.1

-----------------------

PHP 7.1 available as container is a base platform for building and running various PHP 7.1 applications and frameworks. PHP is an HTML-embedded scripting language. PHP attempts to make it easy for developers to write dynamically generated web pages. PHP also offers built-in database integration for several commercial and non-commercial database management systems, so writing a database-enabled webpage with PHP is fairly simple. The most common use of PHP coding is probably as a replacement for CGI scripts.

Tags: builder, php, php71, rh-php71

\* A source build using source code from https://github.com/christianh814/openshift-php-upload-demo will be created

\* The resulting image will be pushed to image stream tag "file-uploader:latest"

\* Use 'start-build' to trigger a new build

\* This image will be deployed in deployment config "file-uploader"

\* Ports 8080/tcp, 8443/tcp will be load balanced by service "file-uploader"

\* Other containers can access this service through the hostname "file-uploader"

--> Creating resources ...

imagestream.image.openshift.io "file-uploader" created

buildconfig.build.openshift.io "file-uploader" created

deploymentconfig.apps.openshift.io "file-uploader" created

service "file-uploader" created

--> Success

Build scheduled, use 'oc logs -f bc/file-uploader' to track its progress.

Application is not exposed. You can expose services to the outside world by executing one or more of the commands below:

'oc expose svc/file-uploader'

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.html)

Author: Christian Hernandez <christianh814@users.noreply.github.com>

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

The command prompt returns out of the tail mode once you see *Push successful*.

|  |  |
| --- | --- |
|  | This use of the new-app command directly asked for application code to be built and did not involve a template. That’s why it only created a **single Pod** deployment with a **Service** and no **Route**. |

Let’s make our application production ready by exposing it via a Route and scale to 3 instances for high availability:

oc expose svc/file-uploader

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

Now, check the **Route** that has been created:

oc get route

You will see something like:

NAME HOST/PORT PATH SERVICES PORT TERMINATION WILDCARD

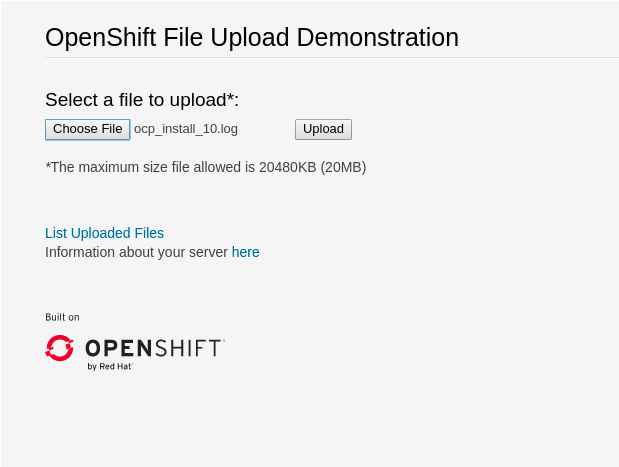
file-uploader file-uploader-my-shared-storage.apps.004325440631.aws.testdrive.openshift.com file-uploader 8080-tcp None

...

Point your browser to the web application using the URL advertised by the route ([http://file-uploader-my-shared-storage.apps.004325440631.aws.testdrive.openshift.com](http://file-uploader-my-shared-storage.apps.004325440631.aws.testdrive.openshift.com/))

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

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



*Figure 4. A simple PHP-based file upload tool*

Once done click ***List uploaded files*** to see the list of all currently uploaded files.

Do you see it? Don’t worry if you don’t.

Change back to the command line and look at the running pods.

oc get pods -l app=file-uploader

You will see 3 pods running:

NAME READY STATUS RESTARTS AGE

file-uploader-1-5hhqb 1/1 Running 0 6m

file-uploader-1-trkxr 1/1 Running 0 6m

file-uploader-1-vqszb 1/1 Running 0 7m

Now let’s look back at where this file got stored inside the pods. Again use the oc rsh utility via a scriptlet to execute an ls command on the upload directory that the PHP code uses to store the files:

**for** pod **in** $(oc get pod -l app=file-uploader --no-headers | awk '{print $1}'); **do** echo $pod; oc rsh $pod ls -hl uploaded; **done**

You will see that only one of the pods has the uploaded file

file-uploader-1-5hhqb

total 0

file-uploader-1-trkxr

total 352K

-rw-r--r--. 1 1000380000 root 352K Oct 29 16:00 firefly-episode-list.txt

file-uploader-1-vqszb

total 0

Why is that? These pods currently do not use any persistent storage. They store the file locally in the container root file system. That means the application cannot effectively be scaled since the pods do not share data and every client would see different uploaded files. To verify this, try accessing the URL with a second *Icognito* browser session.

|  |  |
| --- | --- |
|  | Never attempt to store persistent data in a **Pod** that has no persistent volume associated with it. **Pods** and their containers are ephemeral by definition, and any stored data will be lost as soon as the **Pod** terminates for whatever reason. |

The app is of course not useful 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 set volume command. Execute the following

oc set volume dc/file-uploader --add --name=my-shared-storage \

-t pvc --claim-mode=ReadWriteMany --claim-size=1Gi \

--claim-name=my-shared-storage --mount-path=/opt/app-root/src/uploaded

Like with the mapit application in "*Application Management Basics*" chapter, 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 set volume is capable of, look at its help output with oc set volume -h. Now, let’s look at the result of adding the volume:

oc get pvc

You will see something like:

NAME STATUS VOLUME CAPACITY ACCESSMODES AGE

my-shared-storage Bound pvc-62aa4dfe-4ad2-11e7-b56f-2cc2602a6dc8 1Gi RWX 22s

...

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** reliably. If you attempt to scale up deployments that are using ReadWriteOnce storage, they will actually all become co-located on the same node.

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

Check you have a new set of pods:

oc get pods -l app=file-uploader

You will see something like:

NAME READY STATUS RESTARTS AGE

file-uploader-2-4h7bx 1/1 Running 0 2m

file-uploader-2-gqbsn 1/1 Running 0 2m

file-uploader-2-pkmpj 1/1 Running 0 2m

Try it out in your file uploader web application using your browser. Upload new files and see that they are visible from within all application pods.

|  |  |
| --- | --- |
|  | Where is my previously uploaded file?  Since the pod redeployed the file has been lost with the previous container’s root filesystem going away as part of the configuration update. One more reason to provide persistent storage! |

Once done, return to the command line and look at the contents of pods:

**for** pod **in** $(oc get pod -l app=file-uploader --no-headers | awk '{print $1}'); **do** echo $pod; oc rsh $pod ls -hl uploaded; **done**

You will see that now all of the pods have the uploaded file:

file-uploader-2-4h7bx

total 352K

-rw-r--r--. 1 1000380000 2002 352K Oct 29 16:10 firefly-episode-list.txt

file-uploader-2-gqbsn

total 352K

-rw-r--r--. 1 1000380000 2002 352K Oct 29 16:10 firefly-episode-list.txt

file-uploader-2-pkmpj

total 352K

-rw-r--r--. 1 1000380000 2002 352K Oct 29 16:10 firefly-episode-list.txt

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 OCS this is available wherever OpenShift is deployed without external dependencies like NFS.

Increasing volume capacity

However, what happens when the volume is full?

Let’s try it. Run the following command to fill up the currently 1GiB of free space in the persistent volume. Since it’s shared, you can use any the 3 file-uploader pods:

oc rsh $(oc get pod -l app=file-uploader --no-headers | head -n1 | awk '{print $1}') dd **if**=/dev/zero of=uploaded/bigfile bs=100M count=1000

The result after some time is:

dd: error writing 'uploaded/bigfile': No space left on device

dd: closing output file 'uploaded/bigfile': No space left on device

command terminated with exit code 1

Oops. The file system seems to have a problem. Let’s check it:

oc rsh $(oc get pod -l app=file-uploader --no-headers | head -n1 | awk '{print $1}') df -h /opt/app-root/src/uploaded

Clearly the file system is full:

Filesystem Size Used Avail Use% Mounted on

10.0.1.36:vol\_6320cd6974d8573f49f85a5d7255a7f2 1019M 1019M 0 100% /opt/app-root/src/uploaded

If you were to try uploading another file via the web application it would fail with something along the lines:

[...]

failed to open stream: No space left on device in /opt/app-root/src/upload.php on line 26

[...]

First the StorageClass glusterfs-storage needs to be modified to include allowVolumeExpansion: true. To add this new parameter the following process is used.

|  |  |
| --- | --- |
|  | It is required that the feature-gates: (below) is added to the /etc/origin/master/master-config.yaml and the master services restarted before modifying the StorageClass glusterfs-storage.  kubernetesMasterConfig:  apiServerArguments:  feature-gates:  - ExpandPersistentVolumes=true |

oc get sc glusterfs-storage -o yaml > glusterfs-storage.yaml

And then add the new parameter to the glusterfs-storage-new.yaml file.

sed '/volumeBindingMode: Immediate/a allowVolumeExpansion: true' glusterfs-storage.yaml > glusterfs-storage-new.yaml

Now to modify this StorageClass the current glusterfs-storage needs to be deleted and the new glusterfs-storage-new.yaml used to create glusterfs-storage that containes the necessary parameter allowVolumeExpansion: true.

oc login -u system:admin

oc delete sc glusterfs-storage

oc create -f glusterfs-storage-new.yaml

Now do the following to validate the StorageClass is modified.

oc get sc glusterfs-storage -o yaml

You will see something like below.

allowVolumeExpansion: true

apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

annotations:

storageclass.kubernetes.io/is-default-class: "true"

creationTimestamp: 2019-04-22T19:33:05Z

name: glusterfs-storage

...

Also verify using this command:

oc describe sc glusterfs-storage

You can see AllowVolumeExpansion: True in this output as well.

Name: glusterfs-storage

IsDefaultClass: Yes

Annotations: storageclass.kubernetes.io/is-default-class=true

Provisioner: kubernetes.io/glusterfs

Parameters: resturl=http://heketi-storage.storage.svc:8080,restuser=admin,secretName=heketi-storage-admin-secret,secretNamespace=storage

AllowVolumeExpansion: True

MountOptions: <none>

ReclaimPolicy: Delete

VolumeBindingMode: Immediate

Events: <none>

After the StorageClass is modified to allow PersistentVolume expansion, the volume size can be increased by the user or owner of the app, even without administrator intervention.

|  |  |
| --- | --- |
|  | If you are unfamiliar with the vi editor, please run the following command before continuing:  export EDITOR=nano |

Use the oc edit command to edit the PersistentVolumeClaim that we used to generate the PersistentVolume:

oc edit pvc my-shared-storage

You end up in a vi session editing the PVC object properties in YAML. Go to line that says storage: 1Gi below spec → resources → requests and increase to 5Gi like shown below:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

annotations:

pv.kubernetes.io/bind-completed: "yes"

pv.kubernetes.io/bound-by-controller: "yes"

volume.beta.kubernetes.io/storage-provisioner: kubernetes.io/glusterfs

creationTimestamp: 2018-04-18T10:17:24Z

name: my-shared-storage

namespace: my-shared-storage

resourceVersion: "41960"

selfLink: /api/v1/namespaces/my-shared-storage/persistentvolumeclaims/my-shared-storage

uid: b0544244-42f1-11e8-8f68-02f9630bd644

spec:

accessModes:

- ReadWriteMany

resources:

requests:

storage: 5Gi

storageClassName: glusterfs-storage

volumeName: pvc-b0544244-42f1-11e8-8f68-02f9630bd644

status:

accessModes:

- ReadWriteMany

capacity:

storage: 1Gi

phase: Bound

|  |  |
| --- | --- |
|  | Set this to **5Gi** |

Exit out of vi mode with the :wq command.

|  |  |
| --- | --- |
|  | Upon writing the file the oc edit command will update the PersistentVolumeClaim definition in OpenShift. This way of ad-hoc editing works with many objects in OpenShift. |

Give it a couple of seconds and then check the filesystem again:

oc rsh $(oc get pod -l app=file-uploader --no-headers | head -n1 | awk '{print $1}') df -h /opt/app-root/src/uploaded

The situation should look much better now:

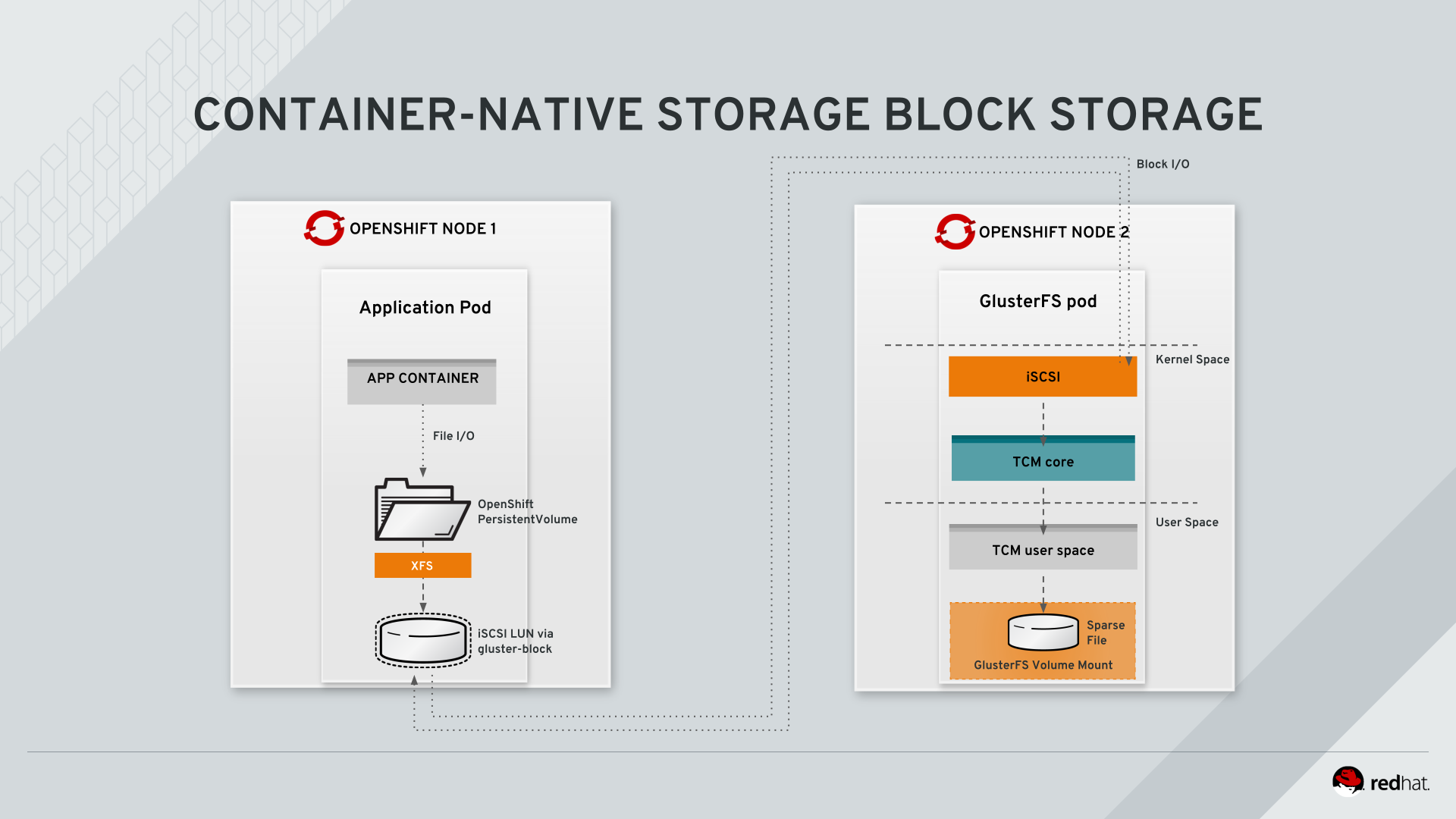
Filesystem Size Used Avail Use% Mounted on

10.0.1.36:vol\_6320cd6974d8573f49f85a5d7255a7f2 5.0G 1.1G 4.0G 21% /opt/app-root/src/uploaded

Providing block storage with OCS

OpenShift Container Storage also contains a block storage persona. At the very end of every **Pod** accessing a PersistentVolume is a filesystem directory bind-mounted to the container’s filesystem namespace. In the case of GlusterFS it’s the GlusterFS filesystem, a POSIX compatible, replicated shared network filesystem. As of today, OpenShift doesn’t support provisioning a block device directly into a **Pod**. All block storage supported by OpenShift eventually gets formatted with a filesystem (like XFS), and is then bind-mounted into the container’s filesystem namespace.

When we speak of block storage in OCS, we are talking about an iSCSI LUN getting provisioned as part of a PersistentVolumeClaimagainst the block-based StorageClass of OCS. This iSCSI LUN is generated from the LIO stack running in the OCS pods. It is backed by a sparse file which is hosted on an internal GlusterFS volume. This subsystem is called gluster-block. See below graphic for a representation:



*Figure 5. gluster-block IO flow in OCS*

Why is this beneficial? Some applications, like OpenShift Logging and Metrics services facilitate operations which are cheap on a local filesystem like XFS but expensive on distributed filesystem like GlusterFS.

With gluster-block you get the advantage of resilient, scalable storage without the overhead on filesystem operations like locking and byte-range locking.

OpenShift Metrics and Logging issue a lot of these operations, and hence **gluster-block is currently the only supported backend in OCS for those services**.

gluster-block was deployed in the previous chapter (*Infrastructure Management Basics*) and used to supply storage to Cassandra as part of the Metrics service and to ElasticSearch as part of the Logging service.

If you look on the host running any of those service, you will see that there are iSCSI sessions open.

For example, pick the host running the ElasticSearch pod:

oc get pod -l component=es -n openshift-logging -o wide

You will see the IP and the hostname of the host the pod is running on. In this example the pod is running on node05.internal.aws.testdrive.openshift.com.

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE

logging-es-data-master-uud3jzgn-1-tkxsd 2/2 Running 0 1h 10.131.0.24 infra.internal.aws.testdrive.openshift.com <none>

|  |  |
| --- | --- |
|  | Above you see one of the examples where a **Pod** actually contains two containers. The ElasticSearch pod contains an additional proxy service, living in its own container but running with the actual ElasticSearch service on the same host. |
|  | To SSH to hosts in the environment you need to do so from the cloud-user account. The root account does not have the SSH keys and root SSH is disabled in this environment. |

Sign on to this host (use the host shown in the last command) from the master using SSH and run the iscsiadm utility to display running iSCSI sessions:

ssh infra.internal.aws.testdrive.openshift.com sudo iscsiadm -m session

Answer "**yes**" to the SSH security prompt. You should see output similar to the below:

tcp: [1] 10.0.4.80:3260,1 iqn.2016-12.org.gluster-block:f625464b-42b7-4251-8dce-ae78f1bdb17d (non-flash)

tcp: [2] 10.0.1.105:3260,2 iqn.2016-12.org.gluster-block:f625464b-42b7-4251-8dce-ae78f1bdb17d (non-flash)

tcp: [3] 10.0.3.126:3260,3 iqn.2016-12.org.gluster-block:f625464b-42b7-4251-8dce-ae78f1bdb17d (non-flash)

tcp: [4] 10.0.4.80:3260,1 iqn.2016-12.org.gluster-block:236d8c72-3229-4863-9576-3e59055336ec (non-flash)

tcp: [5] 10.0.1.105:3260,2 iqn.2016-12.org.gluster-block:236d8c72-3229-4863-9576-3e59055336ec (non-flash)

tcp: [6] 10.0.3.126:3260,3 iqn.2016-12.org.gluster-block:236d8c72-3229-4863-9576-3e59055336ec (non-flash)

The IPs and LUN IDs are going to be different for you, but essentially you see 3 iSCSI sessions open to the same LUN (identified by the UUID after iqn.2016-12.org.gluster-block). There are 3 sessions because every OCS pod of the second OCS cluster for Infrastructure runs the Linux iSCSI target stack (TCMU) and each session represents an independent IO path to the same LUN, thus achieving high availability and path-based failover.

Like all block storage supplied to OpenShift, it gets formatted with XFS which you can see if you look at mounts on the host running ElasticSearch:

ssh infra.internal.aws.testdrive.openshift.com mount | grep iscsi

You will see something similar to this:

/dev/mapper/mpatha on /var/lib/origin/openshift.local.volumes/plugins/kubernetes.io/iscsi/iface-default/10.0.4.80:3260-iqn.2016-12.org.gluster-block:f625464b-42b7-4251-8dce-ae78f1bdb17d-lun-0 type xfs (rw,relatime,seclabel,attr2,inode64,noquota)

/dev/mapper/mpatha on /var/lib/origin/openshift.local.volumes/pods/7c8f4b23-db86-11e8-882a-0e4710a83b18/volumes/kubernetes.io~iscsi/pvc-756ad53d-db86-11e8-882a-0e4710a83b18 type xfs (rw,relatime,seclabel,attr2,inode64,noquota)

/dev/mapper/mpathb on /var/lib/origin/openshift.local.volumes/plugins/kubernetes.io/iscsi/iface-default/10.0.4.80:3260-iqn.2016-12.org.gluster-block:236d8c72-3229-4863-9576-3e59055336ec-lun-0 type xfs (rw,relatime,seclabel,attr2,inode64,noquota)

/dev/mapper/mpathb on /var/lib/origin/openshift.local.volumes/pods/4e4cc348-db87-11e8-882a-0e4710a83b18/volumes/kubernetes.io~iscsi/pvc-20fccede-db87-11e8-882a-0e4710a83b18 type xfs (rw,relatime,seclabel,attr2,inode64,noquota)

As you can see the devicemapper mulipath devices are how the iSCSI LUN ended up (in this case) on infra.internal.aws.testdrive.openshift.com.

To serve a block device from OCS, a special external provisioner is used. You can see its **Pod** in the namespace that the second OCS cluster was deployed to:

oc get pod -n infra-storage -l glusterfs=block-registry-provisioner-pod

You should see something like:

NAME READY STATUS RESTARTS AGE

glusterblock-registry-provisioner-dc-1-vsgpg 1/1 Running 0 21m

This component contains the additional logic to carve out block devices from OCS.

You will also find evidence of the different provisioning mechanism if you look at the StorageClass:

oc get sc

Shows the 3 currently defined StorageClasses in the system:

NAME PROVISIONER AGE

glusterfs-registry kubernetes.io/glusterfs 58m

glusterfs-registry-block gluster.org/glusterblock 58m

glusterfs-storage (default) kubernetes.io/glusterfs 1h

|  |  |
| --- | --- |
|  | The provisioner does not start with kubernetes.io which indicates it’s an external provisioner (shipping as an additional component, not as part of OpenShift or Kubernetes) |

Finally, the block device is reflected as a specific type of volume, a blockvolume in heketi.

Run the following command to ask heketi about all block volumes currently present using the heketi-cli tool:

heketi-cli --server http://heketi-registry-infra-storage.apps.004325440631.aws.testdrive.openshift.com --user=admin --secret myS3cr3tpassw0rd blockvolume list

There should be two, one for Logging and one for Metrics:

Id:a2ceeabc91d453a30e197da764fca8c9 Cluster:f68d7554542bab9d0fdeb683d66d951a Name:blockvol\_a2ceeabc91d453a30e197da764fca8c9

Id:a67906a197ad0c750a90c793452f83c7 Cluster:f68d7554542bab9d0fdeb683d66d951a Name:blockvol\_a67906a197ad0c750a90c793452f83c7

Using heketi-cli you could also provision new block volumes or even create new internal GlusterFS volumes to host block volumes. However this is rarely necessary, since this, at time of writing (2018), is only meant to be in place for Logging and Metrics and provisioning is handled automatically.

OCS Operations

Options to increase Storage Capacity in OCS

At some point the overall OCS cluster capacity may need to be expanded. There are a couple of ways to increase the storage capacity offered by OCS.

1. add a second, separate OCS cluster with its own management stack (heketi) (like you did in the *Infrastructure Management*module )
2. add a second, separate OCS cluster with its own management stack (as described in the [documentation](https://access.redhat.com/documentation/en-us/red_hat_openshift_container_storage/3.11/html-single/operations_guide/index#idm140385886778944))
3. add additional nodes to an existing OCS cluster (as described in the [documentation](https://access.redhat.com/documentation/en-us/red_hat_openshift_container_storage/3.11/html-single/operations_guide/index#sect_Adding_New_Nodes))
4. add additional devices to existing nodes

Option 1) is automated using openshift-ansible

Option 2) is an option you likely want to take when you have nodes with different media types (SSD vs. HDD) and you want to offer quality of service.

Option 3) allows you to easily expand the cluster capacity in-place. In this lab we however have no nodes left to add, so we will illustrate Option 4).

Adding Additional Devices to a OCS Cluster

To perform management operations we’ll use the heketi-cli tool. It manages several entities that make up OCS, that is: clusters, nodes, volumes and devices.

For each entity there are several create/add, update, delete commands available. For initial cluster setup heketi-cli also offers batch processing via a JSON file.

In the following we will manually add devices from node04, node05 and node06, which form the OCS cluster for OpenShift infrastructure.

Like in the *Installation* module, we first set up some Bash environment variables to configure our heketi-cli client to talk to the second OCS cluster. This time we take a shortcut by programmatically determining the URL to heketi and the password by querying the heketi pod:

export HEKETI\_POD=$(oc get pods -l glusterfs=heketi-registry-pod -o jsonpath='{.items[0].metadata.name}' -n infra-storage)

export HEKETI\_CLI\_SERVER=http://$(oc get route -l glusterfs=heketi-registry-route -o jsonpath='{.items[0].spec.host}' -n infra-storage)

export HEKETI\_CLI\_USER=admin

export HEKETI\_CLI\_KEY=$(oc get pod/$HEKETI\_POD -o jsonpath='{.spec.containers[0].env[?(@.name=="HEKETI\_ADMIN\_KEY")].value}' -n infra-storage)

We can now query heketi about the nodes in this cluster:

heketi-cli node list

And you will see something like:

Id:33e0045354db4be29b18728cbe817605 Cluster:ca777ae0285ef6d8cd7237c862bd591c

Id:d8443e7ee8314c0c9fb4d8274a370bbd Cluster:ca777ae0285ef6d8cd7237c862bd591c

Id:caaed3927e424b22b1a89d261f7617ad Cluster:ca777ae0285ef6d8cd7237c862bd591c

The UUIDs of the nodes will be different for you. We however need them to tell heketi from which nodes to add a device. To avoid repetitive copying and pasting here is another Bash short cut to parse above output in a Bash variable:

Run the following command to store the heketi-internal ID of the OCS cluster (there is only one for this heketi instance) in a bash variable:

export CNS\_INFRA\_CLUSTER=$(heketi-cli cluster list --json | jq -r '.clusters[0]')

echo $CNS\_INFRA\_CLUSTER

Then get a list of the nodes of this cluster into a Bash variable:

export NODES=$(heketi-cli cluster info $CNS\_INFRA\_CLUSTER --json | jq -r '.nodes[]')

export NODE\_LIST=($NODES)

echo $NODES

To illustrate the before and after effect, first inspect the output of:

heketi-cli topology info

You should see that every node currently has a single device: /dev/xvdd.

These nodes of the second OCS cluster, have an additional, unused storage device /dev/xvde. For each node now go ahead and make heketi aware of this device using the device add directive

heketi-cli device add --name=/dev/xvde --node=${NODE\_LIST[0]}

heketi-cli device add --name=/dev/xvde --node=${NODE\_LIST[1]}

heketi-cli device add --name=/dev/xvde --node=${NODE\_LIST[2]}

Each command should return with the message Device added successfully.

Check heketi-cli topology info again to verify the presence of the new devices.

That’s it - the devices are now available to heketi and will be considered the next time OCS serves a volume request. Adding devices and nodes are online operations, meaning they are non-disruptive and can be run in production without downtime.

Replacing Failed Disks and Nodes

When a device fails, OCS transparently continues operations with the remaining replicas. You will need to replace such components to move out of a degraded state and get to 3 replicas again, either using other devices free capacity in the same node or in different nodes.

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).

|  |  |
| --- | --- |
|  | OCS is aware of failure domains in your infrastructure. These could be racks in a data center or availability zones in public cloud environments. The zones are identified by distinct values in the zone parameter of each node. Nodes with the same value for zone are considered part of the same failure domain. OCS will try to do its best (but not enforce it) to replicate and rebalance data across 3 different failure domains at all times. |

The first step is to determine the OCS node’s internal UUID in heketi’s database. You can do that manually:

heketi-cli topology info | grep -B4 node04.internal.aws.testdrive.openshift.com

…​and see something like:

Node Id: 33e0045354db4be29b18728cbe817605

State: online

Cluster Id: ca777ae0285ef6d8cd7237c862bd591c

Zone: 1

Management Hostname: node04.internal.aws.testdrive.openshift.com

Or you can do it programmatically, for easy copying and pasting, by asking heketi and parsing its JSON output using jq:

NODE\_4\_ID=$(heketi-cli topology info --json | jq -r ".clusters[] | select(.id==\"$CNS\_INFRA\_CLUSTER\") | .nodes[] | select(.hostnames.manage[0] == \"node04.internal.aws.testdrive.openshift.com\") | .id")

echo $NODE\_4\_ID

This should yield, like above 33e0045354db4be29b18728cbe817605

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

Again, you could do this manually by looking at heketi information about the node:

heketi-cli node info $NODE\_4\_ID

And then you will see:

Node Id: 33e0045354db4be29b18728cbe817605

State: online

Cluster Id: 119ea7f96ce132f15a04c28de9978018

Zone: 1

Management Hostname: node04.internal.aws.testdrive.openshift.com

Storage Hostname: 10.0.1.214

Devices:

Id:0b32d5e57f2047485e42e6288405ad7f Name:/dev/xvde State:online Size (GiB):49 Used (GiB):0 Free (GiB):49

Id:4fb2ae473d5ee451906d5489abfc653e Name:/dev/xvdd State:online Size (GiB):49 Used (GiB):42 Free (GiB):7

Or again, for easy copying and pasting, you can do it the smart way and retrieve the device ID of /dev/xvdd programmatically from the JSON output using jq:

export FAILED\_DEVICE\_ID=$(heketi-cli node info $NODE\_4\_ID --json | jq -r '.devices[] | select(.name=="/dev/xvdd") | .id')

echo $FAILED\_DEVICE\_ID

You should get the UUID of /dev/xvdd from this command, in this example 4fb2ae473d5ee451906d5489abfc653e.

With the UUID we can first mark the device as offline to stop heketi from further attempts to allocate space from it:

heketi-cli device disable $FAILED\_DEVICE\_ID

You will see something like:

Device 4fb2ae473d5ee451906d5489abfc653e 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 $FAILED\_DEVICE\_ID

You will see something like:

Device 4fb2ae473d5ee451906d5489abfc653e is now removed

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

This is an online operation and can absolutely be run in production.

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

heketi-cli device delete $FAILED\_DEVICE\_ID

You will see something like:

Device 4fb2ae473d5ee451906d5489abfc653e deleted

|  |  |
| --- | --- |
|  | Only devices that are not currently used by other Gluster volumes can be deleted. If that’s not the case, heketi-cliwill tell you about it. Devices that are in use always need to have remove performed first. |

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

[Go to previous module](http://support.004325440631.aws.testdrive.openshift.com/index.html#/workshop/ocp-admin-testdrive/module/infra-mgmt-basics)

## Skipping Modules

While the Test Drive is designed to be an experience that is linear in nature, it is possible to skip modules. Skipping modules requires that you perform certain automation tasks that will actually "do" the exercise for you.

For example, if you wanted to skip user and group configuration, later labs still rely on this being completed, so you must still make sure the steps have been executed.

We provide automation of each module using Ansible playbooks that are located in the /opt/lab/code/tests folder.

| *Table 1. The Automation Playbooks* | |
| --- | --- |
| **Module** | **Playbook** |
| 2. Installation and Verification | run-module-2-installation.yaml |
| 3. Application Management Basics | run-module-3-app-mgmt-basics.yaml |
| 4. Project Request Template, Quotas, Limits | run-module-4-template-quota-limits.yaml |
| 5. Using External Authentication Providers (LDAP) | run-module-5-ldap-groupsync.yaml |
| 6. Infrastructure Management Basics | run-module-6-infra-mgmt.yaml |
| 7. OpenShift Container Storage | run-module-7-cns.yaml |

Executing the skip module playbooks is trivial. They are self-contained Ansible playbooks that are run from the /opt/lab/code/testsdirectory.

There is a constraint that if you want to skip across multiple modules, **you need to skip all preceding modules in order**. For example, if you wanted to skip to Module 5 (that is, skipping modules 2-4 as well), you would need to run the following:

cd /opt/lab/code/tests/

ansible-playbook run-module-2-installation.yaml

ansible-playbook run-module-3-app-mgmt-basics.yaml

ansible-playbook run-module-4-template-quota-limits.yaml

|  |  |
| --- | --- |
|  | Do not run playbooks without following above constraint of running them in order. |
|  | Do not run the playbooks twice. They are not quite idempotent yet. |