1. Environment Overview

2. Installation and Verification

3. Deploying Container Native Storage

4. Application Management Basics

5. Project Template, Quota, and Limits

6. Using External Authentication Providers (LDAP)

7. Infrastructure Management Basics

8. Container-native Storage Management

Application Management Basics

4

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:

9. Skipping modules

```
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 "d

    * An image stream will be created as "mapit:latest" that will 
    * This image will be deployed in deployment config "mapit"
    * Ports 8080/tcp, 8778/tcp, 9779/tcp will be load balanced by 
      * Other containers can access this service through the hostn

--> Creating resources ...
    imagestream "mapit" created
    deploymentconfig "mapit" created
    service "mapit" created
--> Success
    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

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-1-6lczv
Namespace:              app-management
Security Policy:        restricted
Node:                   node02.internal.aws.testdrive.openshift.co
Start Time:             Thu, 17 Aug 2017 13:41:00 +0000
Labels:                 app=mapit
                        deployment=mapit-1
                        deploymentconfig=mapit
Status:                 Running
IP:                     10.129.0.3
Controllers:            ReplicationController/mapit-1
Containers:
  mapit:
    Container ID:       docker://7eb42d5d95b38f7804e38f05cd423314c
    Image:              docker.io/siamaksade/mapit@sha256:338a3031
    Image ID:           docker-pullable://docker.io/siamaksade/map
    Ports:              8080/TCP, 8778/TCP, 9779/TCP
    State:              Running
```

```
      Started:         Thu, 17 Aug 2017 13:41:27 +0000
    Ready:             True
    Restart Count:     0
    Volume Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-t
    Environment Variables:      <none>
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  default-token-kthcr:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-kthcr
...
```

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

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, 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      CLUSTER-IP     EXTERNAL-IP   PORT(S)
mapit     172.30.3.117   <none>        8080/TCP,8778/TCP,9779/TCP
```

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
Selector:               app=mapit,deploymentconfig=mapit
Type:                   ClusterIP
IP:                     172.30.3.117
Port:                   8080-tcp        8080/TCP
Endpoints:              10.129.0.3:8080
Port:                   8778-tcp        8778/TCP
Endpoints:              10.129.0.3:8778
Port:                   9779-tcp        9779/TCP
Endpoints:              10.129.0.3:9779
Session Affinity:       None
No events.
```

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: 2017-08-17T13:40:51Z
  labels:
    app: mapit
  name: mapit
  namespace: app-management
  resourceVersion: "1492"
  selfLink: /api/v1/namespaces/app-management/services/mapit
  uid: af2cb9cd-8351-11e7-afdc-0a128c2d4cfe
spec:
  clusterIP: 172.30.3.117
  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:

```
Building on replication controllers, OpenShift adds expanded suppo
software development and deployment lifecycle with the concept of 
In the simplest case, a deployment just creates a new replication 
lets it start up pods. However, OpenShift deployments also provide
```

```
to transition from an existing deployment of an image to a new one
define hooks to be run before or after creating the replication co
```

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          0          4h
```

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

```
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
Selector:               app=mapit,deploymentconfig=mapit
Type:                   ClusterIP
IP:                     172.30.3.117
Port:                   8080-tcp        8080/TCP
Endpoints:              10.128.2.3:8080,10.129.0.3:8080
Port:                   8778-tcp        8778/TCP
Endpoints:              10.128.2.3:8778,10.129.0.3:8778
Port:                   9779-tcp        9779/TCP
Endpoints:              10.128.2.3:9779,10.129.0.3:9779
Session Affinity:       None
No events.
```

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
mapit     10.128.2.3:9779,10.129.0.3:9779,10.128.2.3:8080 + 3 more
```

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-1-6lczv" deleted
NAME            READY     STATUS              RESTARTS   AGE
mapit-1-6lczv   1/1       Terminating         0          4h
mapit-1-qtdks   0/1       ContainerCreating   0          0s
mapit-1-rq6t6   1/1       Running             0          6m
```

Did you notice anything? There is a container being terminated (the one we deleted), and there's a new container already being created.

Also, the names of the **Pods** are slightly changed. That's because OpenShift almost immediately detected that the current state (1 **Pod**) didn't match the desired state (2 **Pods**), and it fixed it by scheduling another **Pod**.

Background: Routes

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, 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 something 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 "mapit" exposed
```

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

```
oc get route
```

You will see something like:

```
NAME      HOST/PORT
mapit     mapit-app-management.apps.647073518612.aws.testdrive.ope
```

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.647073518612.aws.t
```

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 endpoints 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 endpoints 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 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.647073518612.aws.testdrive.openshif
```

You will get some JSON as a response:


```
{"status":"UP","diskSpace":{"status":"UP","total":10724835
```


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 --i
```

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::
    Ports:                        8080/TCP, 8778/TCP, 9779/TCP
    Liveness:                     http-get http://:8080/health delay=
    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 --i
```

Go to previous module

Go to next module