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.

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**. Technically speaking, a user doesn’t own the resources, the **Project** does. Deleting the user doesn’t affect any of the created resources.

For this exercise, first create a **Project** to hold some 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 quay.io/thoraxe/mapit
```

The output will look like:

```
--> Found container image 7ce7ade (3 years old) from quay.io for "quay.io/thoraxe/mapit"

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

--> Creating resources ...
    imagestream.image.openshift.io "mapit" created
    deployment.apps "mapit" created
    service "mapit" created
--> Success
    Application is not exposed. You can expose services to the outside world by executin
g one or more of the commands below:
     'oc expose service/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 a host. A pod is the smallest compute unit you can define, deploy and manage in OpenShift. Each pod is allocated its own internal IP address on the SDN and owns 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-764c5bf8b8-l49z7    1/1     Running   0          2m53s
```

NOTE 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 -l deployment=mapit
```

NOTE The `-l deployment=mapit` selects the pod that is related to the `Deployment` which will be discussed later.

And you will see output similar to the following:

```
Name:         mapit-764c5bf8b8-l49z7
Namespace:    app-management
Priority:     0
Node:         ip-10-0-128-29.us-west-2.compute.internal/10.0.128.29
Start Time:   Tue, 10 Nov 2020 21:01:09 +0000
Labels:       deployment=mapit
              pod-template-hash=764c5bf8b8
Annotations:  k8s.v1.cni.cncf.io/network-status:
                [{
                    "name": "",
                    "interface": "eth0",
                    "ips": [
                        "10.129.2.99"
                    ],
                    "default": true,
                    "dns": {}
                }]
              k8s.v1.cni.cncf.io/networks-status:
                [{
                    "name": "",
                    "interface": "eth0",
                    "ips": [
                        "10.129.2.99"
                    ],
                    "default": true,
                    "dns": {}
                }]
              openshift.io/generated-by: OpenShiftNewApp
              openshift.io/scc: restricted
Status:       Running
IP:           10.129.2.99
IPs:
  IP:           10.129.2.99
Controlled By:  ReplicaSet/mapit-764c5bf8b8
Containers:
  mapit:
    Container ID:   cri-o://fb708e659c19c6aaf8211bf7e3029f8adc8cf14959bcaefa5c7e6df17d37
feaf
    Image:          quay.io/thoraxe/mapit@sha256:8c7e0349b6a016e3436416f3c54debda4594ba0
9fd34b8a0dee0c4497102590d
    Image ID:       quay.io/thoraxe/mapit@sha256:8c7e0349b6a016e3436416f3c54debda4594ba0
9fd34b8a0dee0c4497102590d
    Ports:          9779/TCP, 8080/TCP, 8778/TCP
    Host Ports:     0/TCP, 0/TCP, 0/TCP
    State:          Running
      Started:      Tue, 10 Nov 2020 21:01:29 +0000
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-v7fpq (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-v7fpq:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-v7fpq
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  <none>
Tolerations:     node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
                 node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type    Reason          Age    From               Message
  ----    ------          ----   ----               -------
  Normal  Scheduled       6m50s  default-scheduler  Successfully assigned app-management
/mapit-764c5bf8b8-l49z7 to ip-10-0-128-29.us-west-2.compute.internal
  Normal  AddedInterface  6m48s  multus             Add eth0 [10.129.2.99/23]
  Normal  Pulling         6m48s  kubelet            Pulling image "quay.io/thoraxe/mapit
@sha256:8c7e0349b6a016e3436416f3c54debda4594ba09fd34b8a0dee0c4497102590d"
  Normal  Pulled          6m31s  kubelet            Successfully pulled image "quay.io/t
horaxe/mapit@sha256:8c7e0349b6a016e3436416f3c54debda4594ba09fd34b8a0dee0c4497102590d" in
 16.762028989s
  Normal  Created         6m31s  kubelet            Created container mapit
  Normal  Started         6m31s  kubelet            Started container mapit
```

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, the `new-app` command 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.

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   172.30.167.160   <none>        8080/TCP,8778/TCP,9779/TCP   26
```

NOTE 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
                   app.kubernetes.io/component=mapit
                   app.kubernetes.io/instance=mapit
Annotations:       openshift.io/generated-by: OpenShiftNewApp
Selector:          deployment=mapit
Type:              ClusterIP
IP:                172.30.167.160
Port:              8080-tcp  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.129.2.99:8080
Port:              8778-tcp  8778/TCP
TargetPort:        8778/TCP
Endpoints:         10.129.2.99:8778
Port:              9779-tcp  9779/TCP
TargetPort:        9779/TCP
Endpoints:         10.129.2.99:9779
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: "2020-11-10T21:01:09Z"
  labels:
    app: mapit
    app.kubernetes.io/component: mapit
    app.kubernetes.io/instance: mapit
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          .: {}
          f:openshift.io/generated-by: {}
        f:labels:
          .: {}
          f:app: {}
          f:app.kubernetes.io/component: {}
          f:app.kubernetes.io/instance: {}
      f:spec:
        f:ports:
          .: {}
          k:{"port":8080,"protocol":"TCP"}:
            .: {}
            f:name: {}
            f:port: {}
            f:protocol: {}
            f:targetPort: {}
          k:{"port":8778,"protocol":"TCP"}:
            .: {}
            f:name: {}
            f:port: {}
            f:protocol: {}
            f:targetPort: {}
          k:{"port":9779,"protocol":"TCP"}:
            .: {}
            f:name: {}
            f:port: {}
            f:protocol: {}
            f:targetPort: {}
        f:selector:
          .: {}
          f:deployment: {}
        f:sessionAffinity: {}
        f:type: {}
    manager: oc
    operation: Update
    time: "2020-11-10T21:01:09Z"
  name: mapit
  namespace: app-management
  resourceVersion: "106194"
  selfLink: /api/v1/namespaces/app-management/services/mapit
  uid: 921c2e2c-a53e-4f83-8e76-9df962069314
spec:
  clusterIP: 172.30.167.160
  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:
    deployment: 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 -l deployment=mapit -o jsonpath='{.items[*].metadata.labels}' | jq -r
```

NOTE The `-o jsonpath` selects a specific field. In this case we are asking for the `labels` section in the manifest.

The output should look something like this:

```
{
  "deployment": "mapit",
  "pod-template-hash": "764c5bf8b8"
}
```

- The **Service** has a `selector` stanza that refers to `deployment: mapit`.
- The **Pod** has multiple **Labels**:
    - `deployment: mapit`
    - `pod-template-hash: 764c5bf8b8`

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 Replica Sets

While **Services** provide routing and load balancing for **Pods**, which may go in and out of existence, **ReplicaSets** (RS) 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 **ReplicaSet** is needed. Without an RS, any **Pods** that are killed or somehow die/exit are not automatically restarted. **ReplicaSets** are how OpenShift "self heals".

A **Deployments** (deploy) defines how something in OpenShift should be deployed. From the deployments documentation:

```
Deployments describe the desired state of a particular component of an
application as a Pod template. Deployments create ReplicaSets, which
orchestrate Pod lifecycles.
```

In almost all cases, you will end up using the **Pod**, **Service**, **ReplicaSet** and **Deployment** 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 **RS** without a **Deployments** or a **Service**, and others, but these are advanced topics not covered in these exercises.

NOTE Earlier versions of OpenShift used something called a **DeploymentConfig**. While still a valid deployment mechanism, moving forward the **Deployment** will be what will be created with `oc new-app`. See the official documentation for more details.

Exploring Deployment-related Objects

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

```
oc get deploy
```

You will see something like the following:

```
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
mapit   1/1     1            1           76m
```

To get more details, we can look into the **ReplicaSet** (**RS**).

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

```
oc get rs
```

You will see something like the following:

```
NAME               DESIRED   CURRENT   READY   AGE
mapit-764c5bf8b8   1         1         1       77m
```

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 deploy/mapit
```

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

```
oc get rs
```

You will see something like the following:

```
NAME               DESIRED   CURRENT   READY   AGE
mapit-764c5bf8b8   2         2         2       79m
mapit-8695cb9c67   0         0         0       79m
```

NOTE The "older" version was kept. This is to we can "rollback" to a previous version of the application.

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-764c5bf8b8-b4vpn   1/1     Running   0          112s
mapit-764c5bf8b8-l49z7   1/1     Running   0          81m
```

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
                   app.kubernetes.io/component=mapit
                   app.kubernetes.io/instance=mapit
Annotations:       openshift.io/generated-by: OpenShiftNewApp
Selector:          deployment=mapit
Type:              ClusterIP
IP:                172.30.167.160
Port:              8080-tcp  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.128.2.19:8080,10.129.2.99:8080
Port:              8778-tcp  8778/TCP
TargetPort:        8778/TCP
Endpoints:         10.128.2.19:8778,10.129.2.99:8778
Port:              9779-tcp  9779/TCP
TargetPort:        9779/TCP
Endpoints:         10.128.2.19:9779,10.129.2.99:9779
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   10.128.2.19:8080,10.129.2.99:8080,10.128.2.19:9779 + 3 more...   81m
```

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

Now that we have two **Pods** running right now, let’s see what happens when we delete them. Frist, run the `oc get pods` command, and make note of the **Pod** names:

```
oc get pods
```

You will see something like the following:

```
NAME                     READY   STATUS    RESTARTS   AGE
mapit-764c5bf8b8-lxnvw   1/1     Running   0          2m28s
mapit-764c5bf8b8-rscss   1/1     Running   0          2m54s
```

Now, delete the pods that belog to the **Deployment** `mapit`:

```
oc delete pods -l deployment=mapit
```

Run the `oc get pods` command once again:

```
oc get pods
```

Did you notice anything? There are new containers already running!

The **Pods** has a different name. That's because OpenShift almost immediately detected that the current state (0 **Pods**, because they were deleted) didn't match the desired state (2 **Pods**), and it fixed it by scheduling the **Pods**.

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 installation deployed an Operator for the router, and the operator created a router for you! The router lives in the `openshift-ingress` **Project**, and you can see information about it with the following command:

```
oc describe deployment router-default -n openshift-ingress
```

You will explore the Operator for the router more in a subsequent exercise.

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.{{ ROUTE_SUBDOMAIN }}           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}`

In the subsequent router Operator labs we'll explore this and other configuration options.

While the router configuration specifies the domain(s) that the router should listen for, something still needs to get requests for those domains to the Router in the first place. There is 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 quay.io/thoraxe/mapit
oc scale --replicas=2 deploy/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 deploy/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 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.{{ ROUTE_SUBDOMAIN }}/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 deploy/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 deploy mapit
```

You will see a section like:

```
...
  Containers:
   mapit:
    Image:        quay.io/thoraxe/mapit@sha256:8c7e0349b6a016e3436416f3c54debda
4594ba09fd34b8a0dee0c4497102590d
    Ports:        9779/TCP, 8080/TCP, 8778/TCP
    Host Ports:   0/TCP, 0/TCP, 0/TCP
    Liveness:     http-get http://:8080/health delay=30s timeout=1s period=10s
#success=1 #failure=3
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
...
```

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

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

Examining Deployments and ReplicaSets

Each change to the **Deployment** is counted as a *configuration* change, which *triggers* a new *deployment*. The **Deployment** in in charge of which **ReplicaSet** to deploy. The *newest* is always deployed.

Execute the following:

```
oc get deployments
```

You should see something like:

```
NAME    READY   UP-TO-DATE   AVAILABLE   AGE
mapit   1/1     1            1           131m
```

You made two material configuration changes (plus a scale), after the initial deployment, thus you are now on the fourth revision of the **Deployment**.

Execute the following:

```
oc get replicasets
```

You should see something like:

```
NAME               DESIRED   CURRENT   READY   AGE
mapit-5f695ff4b8   1         1         1       4m19s
mapit-668f69cdd5   0         0         0       6m18s
mapit-764c5bf8b8   0         0         0       133m
mapit-8695cb9c67   0         0         0       133m
```

Each time a new deployment is triggered, the deployer pod creates a new**ReplicaSet** which then is responsible for ensuring that pods exist. Notice that the old RSs have a desired scale of zero, and the most recent RS has a desired scale of 1.

If you `oc describe` each of these RSs you will see how earlier versions have no probes, and the latest running RS has the new probes.

Last updated 2020-12-08 08:51:23 +0100