Agility 2018 Hands-on Lab Guide

Contents:

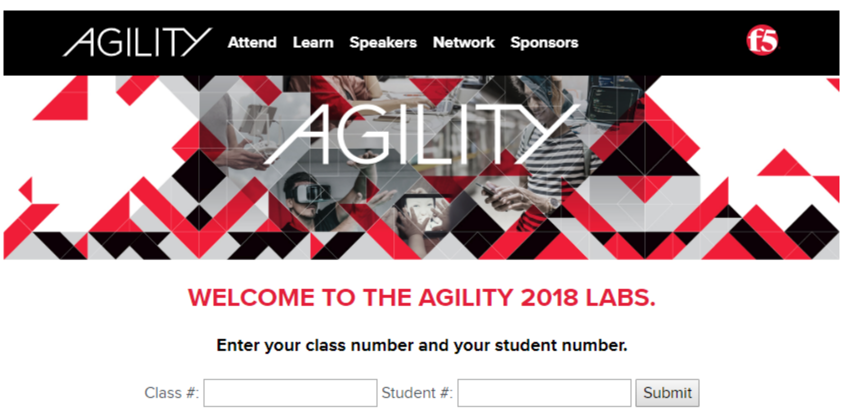
1 Getting Started

Important:

- **The instructor will provide the necessary details to connect to lab environment.**
- Please follow the instructions provided to start your lab and access your jump host.
- All work for this lab will be performed exclusively from the Windows jumpbox via RDP.
- No installation or interaction with your local system is required.

Attention:

- The lab is based on Ravello blueprint `Agility 2018-Containers (Exp Nov 9th, 2018)-vtog-2.2.2`
- To access the lab environment follow this link http://training.f5agility.com
- Once you have established remote access to your lab jumpbox, launch Google Chrome and mRemoteNG (both have shortcuts on the desktop and taskbar).



Tip: For MAC user, it is recommended to use Microsoft Remote Desktop. You may not be able to access your jumpbox otherwise. It is available in the App store (FREE).

Tip: The default keyboard mapping is set to english. If you need to change it, follow these steps:

1. Click on the start menu button and type ‘language’ in the search field
2. Click on ‘Change keyboards or other input methods’ in the search list
3. Click on ‘Change keyboards. . . ’
4. Click ‘Add. . . ’
5. Select the language you want for your keyboard mapping and click ‘OK’
6. Change the ‘Default input language’ in the drop down list to the language added in the previous step
7. Click ‘Apply’ –> Click ‘OK’ –> Click ‘OK’

2 Class 1: Introduction to Docker

This introductory class covers the following topics:

2.1 Module 1: Introduction to Docker

2.1.1 Introduction to Docker

Docker and containers have been a growing buzzword in recent years. As companies started asking for integration with F5, F5 PD resources have been building BIG-IP integration with a BIG-IP controller (more later in the labs) with a container. Via some configs.yaml files (more later in the labs), you can automate F5 into dynamic services being deployed within your organization as well for both on-prem and cloud locations.

To the question what Docker is, and for you reading that haven't researched what Docker is, Docker is a company that figured out to simplify some old linux services into an extremely easy and quick way to deploy smaller images than entire guest images as we have been doing for the past 10-15 years on hypervisor systems.

Let us step back for a moment and look at the context of technologies as they apply to I.T. history. While some products only last moments, others seem to endure forever (COBOL for example – there are companies still using it today). Some of you reading this will be new to the world of IT, while others have seen the progression from mainframes, mini, physical servers, Hypervisors, and as of late docker/containers/microservices, and serverless. Docker is one of companies' technology that might not be the end state of IT, but just like COBOL, this docker technology has the power stay around for a very long time. In much of the same way that VMWare and other hypervisors over the past dozen or so years have transformed most businesses physical servers into a world of virtual servers saving cost, floor space, enabling easier management, ability to support snapshots and many other technologies only dreamed of decades ago.

In a way, containers are doing what hypervisors did to physical servers. Docker essential development deploying containers via a simplification of old features of Unix (going back to Sun Solaris or FreeBSD from early 2000's with zones and jail to separate users, file system views, and processes). By delivering this in a container to run specific code i.e. Tomcat, PHP, or WordPress for example. As containers removes the need to support the Guest OS, this has immediate benefits: running a single file/container with all the software/code embedded within that "image". Containers are typically much smaller, faster, and easier to swap in/out as needed with code upgrades. A decent laptop can spin up a dozen TomCat Apache servers in about a second with embedded HTML code for your site, or within a few seconds have pulled down new html code. Lastly, one can update the container image with new HTML code, save the new container. All while saving over a traditional OS and Tomcat install anywhere from 5X to 25X(or more) less memory and disk requirements.

For today labs at Agility, all these labs will run in the cloud, due to the number of guests needed to host a few different management platforms for containers (RedHat Openshift, Kubernetes (K8s), and Mesos/Marathon). Next page we will install Docker and run a small container for a “hello world”.

Side note for your own work after today: Windows versus Linux you are in luck (mostly), containers are cross platform or “agnostic” of OS that containers run on. If you decide to install Docker on Linux on your own (as in next page) you install only the Docker Engine and management tools. You don’t need to create a virtual machine or virtual networks, because Docker via it’s containers will handle the setup for you.

For Windows: having another hypervisor can cause conflicts. During Docker installation, Docker creates a Linux-based virtual machine called MobyLinuxVM. The Docker application connects to this machine, so that you can create your container with the necessary apparatus for operation. This installation also configures a subnet for the virtual machine to communicate with the local network / NAT for your containers to use in the application. All of these steps occur behind the scenes and, as the user, you don’t really have to worry about them. Still, the fact that Docker on Windows runs a virtual machine in the background is a major difference between Docker on Windows and Docker on Linux.

See also:

For more information please come back and visit any of these links below:

https://www.docker.com

https://www.infoworld.com/article/3204171/linux/what-is-docker-linux-containers-explained.html

https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/

Next, we’re going to install Docker and learn some of the basic commands. We’ll do this on a few Ubuntu servers (Kubernetes VM’s in the lab).

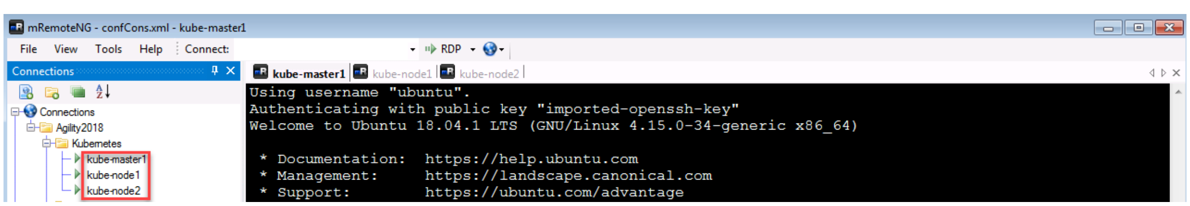
Lab 1.1 Install Docker

Important: The following commands need to be **run on all three nodes** unless otherwise specified.

---

1. From the jumpbox open **mRemoteNG** and start a session to each of the following servers. The sessions are pre-configured to connect with the default user “ubuntu”.

   - kube-master1
   - kube-node1
   - kube-node2

2. Once connected via CLI(SSH) to **ALL** three nodes as user *ubuntu* (it’s the user already setup in the MremoteNG settings), let’s elevate to root:

```
su -

#When prompted for password enter "default" without the quotes
```

Your prompt should change to root@ at the start of the line :

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ su -
Password:
root@kube-master1:~#
```

3. Then, to ensure the OS is up to date, run the following command

```
apt update && apt upgrade -y
```

Note: This can take a few seconds to several minute depending on demand to download the latest updates for the OS.

4. Add the docker repo

```
curl \-fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add \-

add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu
↪$(lsb_release -cs) stable"
```

5. Install the docker packages

```
apt update && apt install docker-ce -y
```

6. Verify docker is up and running

```
docker run --rm hello-world
```

If everything is working properly you should see the following message

```
root@kube-master1:~# docker run --rm hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

root@kube-master1:~#
```

Hint: If you are not a linux/unix person - don't worry. What happened above is how linux installs and updates software. This is ALL the ugly (under the cover steps to install apps, and in this case Docker on a Linux host. Please ask questions as to what really happened, but this is how with linux on ubuntu (and many other linux flavors) installs applications. Linux uses a term called “package manager”, and there are many: like PIP, YUM, APT, DPKG, RPM, PACMAN, etc. usually one is more favored by the flavor of linux (i.e. debian, ubuntu, redhat, gentoo, OpenSuse, etc.), but at the end of the day they all pretty much do the same thing, download and keep applications updated.

Lab 1.2 Run a Container on Docker

See also:

This is only a very basic introduction to docker. For everything else see Docker Documentation

1. Continuing from where we left off on the jumpbox go back to the **kube-master1** session.
2. Now that docker is up and running and confirmed to be working lets deploy the latest Apache web server.

   Note: The `docker run` command will first look for a local cache of the container **httpd**, and upon comparing that copy to the latest instance, decide to either download an update or use the local copy. Since there is no local copy, docker will download the container **httpd** to your local cache. This can take a few seconds (or longer), depending on container size and your bandwidth. Docker will chunk this into parts called a pull.

   `--rm` “tells docker to remove the container after stopping”

   `--name` “give the container a memorable name”

   `-d` “tells docker to run detached. Without this the container would run in foreground and stop upon exit”

   `-it` “this allows for interactive process, like shell, used together in order to allocate a tty for the container process”

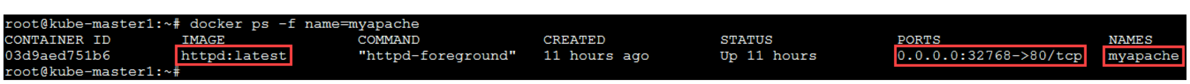
   `-P` “tells docker to auto assign any required ephemeral port and map it to the container”

```
docker run --rm --name "myapache" -d -it -P httpd:latest
```

3. If everything is working properly you should see your container running.

   Note: `-f` “lets us filter on key:pair”

```
docker ps -f name=myapache
```

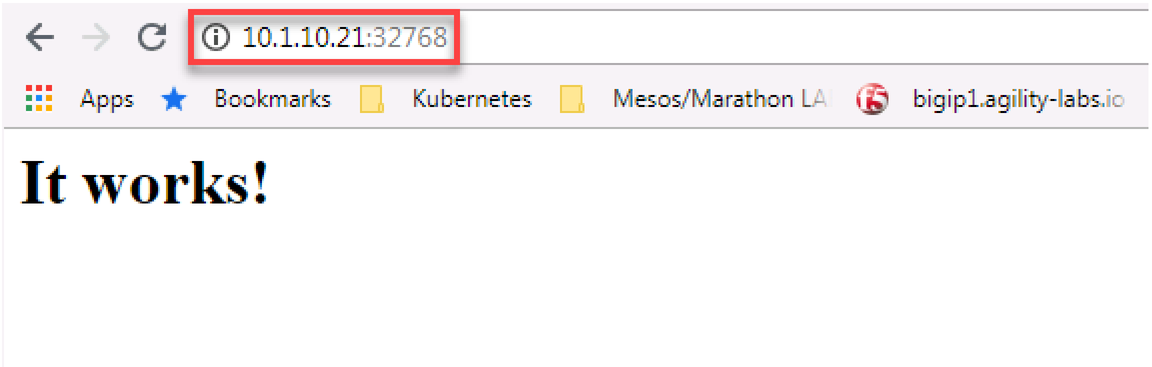


   Note: The “PORTS” section shows the container mapping. In this case the nodes local IP and port 32768 are mapped to the container. We can use this info to connect to the container in the next step.

   ---

4. The httpd container “myapache, is running on kube-master1 (10.1.10.21) and port 32768. To test, connect to the webserver via chrome.

```
http://ip:port
```



Attention: That's it, you installed docker, downloaded a container, ran the Hello World container, ran a web server container, and accessed your web server container via the browser.

Expected time to complete: **15 minutes**

2.2 Lab Setup

We will leverage the kubernetes VM's to configure the Docker environment.

Hostname	IP-ADDR	Credentials
jumpbox	10.1.1.250	user/Student!Agility!
bigip1	10.1.1.245 10.1.10.60	admin/admin root/default
kube-master1	10.1.10.21	ubuntu/ubuntu root/default
kube-node1	10.1.10.22	ubuntu/ubuntu root/default
kube-node2	10.1.10.23	ubuntu/ubuntu root/default

3 Class 2: Introduction to Kubernetes

This introductory class covers the following topics:

3.1 Module 1: Introduction to Kubernetes

The purpose of this module is to give you a basic understanding of kubernetes concepts and components

3.1.1 Kubernetes Overview

Kubernetes has a lot of documentation available at this location: Kubernetes docs

On this page, we will try to provide all the relevant information to deploy successfully a cluster (Master + nodes)

Overview

Extract from: Kubernetes Cluster Intro

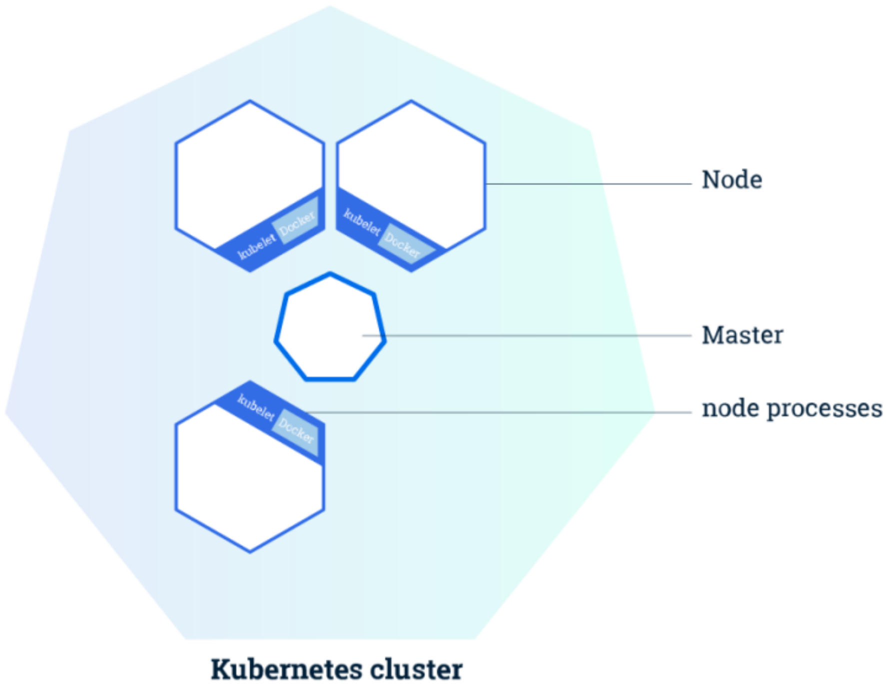
Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit. The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines. To make use of this new model of deployment, applications need to be packaged in a way that decouples them from individual hosts: they need to be containerized.

Containerized applications are more flexible and available than in past deployment models, where applications were installed directly onto specific machines as packages deeply integrated into the host. Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way. Kubernetes is an open-source platform and is production-ready.

A Kubernetes cluster consists of two types of resources:

- The *Master* coordinates the cluster
- *Nodes* are the workers that run applications

Cluster Diagram



The Master is responsible for managing the cluster. The master coordinates all activity in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.

A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster. Each *node* has a *Kubelet*, which is an agent for managing the node and communicating with the Kubernetes master. The node should also have tools for handling container operations, such as Docker or rkt. A ubernetes cluster that handles production traffic should have a minimum of three nodes.

Masters manage the cluster and the *nodes* are used to host the running applications.

When you deploy applications on Kubernetes, you tell the *master* to start the application containers. The *master* schedules the containers to run on the cluster's nodes. **The nodes communicate with the master using the Kubernetes API**, which the *master* exposes. End users can also use the Kubernetes API directly to interact with the cluster.

Kubernetes concepts

Extract from Kubernetes concepts

Cluster: Kubernetes Cluster A cluster is a set of physical or virtual machines and other infrastructure resources used by Kubernetes to run your applications.

Namespace: Kubernetes Namespace Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces. Namespaces are intended for use in environments with many users spread across multiple teams, or projects. For clusters with a few to tens of users, you should not need to create or think about namespaces at all. Start using namespaces when you need the features they provide. Namespaces provide a scope for names. Names of resources need to be unique within a namespace, but not across namespaces. Namespaces are a way to divide cluster resources between multiple uses

Node: Kubernetes Node A node is a physical or virtual machine running Kubernetes, onto which *pods* can be scheduled. It was previously known as *Minion*

Pod: Kubernetes Pods A pod is a co-located group of containers and volumes. The applications in a *pod* all use the same network namespace (same IP and port space), and can thus *find* each other and communicate using **localhost**. Because of this, applications in a pod must coordinate their usage of ports. Each *pod* has an IP address in a flat shared networking space that has full communication with other physical computers and pods across the network. In addition to defining the application containers that run in the *pod*, the *pod* specifies a set of shared storage volumes. Volumes enable data to survive container restarts and to be shared among the applications within the pod.

Label: Kubernetes Label and Selector A label is a key/value pair that is attached to a resource, such as a *pod*, to convey a user-defined identifying attribute. Labels can be used to organize and to select subsets of resources.

Selector: Kubernetes Label and Selector A selector is an expression that matches labels in order to identify related resources, such as which *pods* are targeted by a load-balanced service.

deployments: Kubernetes deployments A Deployment provides declarative updates for Pods and Replica Sets (the next-generation Replication Controller). You only need to describe the desired state in a Deployment object, and the Deployment controller will change the actual state to the desired state at a controlled rate for you. You can define Deployments to create new resources, or replace existing ones by new ones. A typical use case is:

- Create a Deployment to bring up a Replica Set and Pods.
- Check the status of a Deployment to see if it succeeds or not.
- Later, update that Deployment to recreate the Pods (for example, to use a new image).
- Rollback to an earlier Deployment revision if the current Deployment isn't stable.
- Pause and resume a Deployment

ConfigMap: Kubernetes ConfigMap Any applications require configuration via some combination of config files, command line arguments, and environment variables. These configuration artifacts should be decoupled from image content in order to keep containerized applications portable. The ConfigMap API resource provides mechanisms to inject containers with configuration data while keeping containers agnostic of Kubernetes. ConfigMap can be used to store fine-grained information like individual properties or coarse-grained information like entire config files or JSON blobs

Replication Controller: Kubernetes replication controller A replication controller ensures that a specified number of *pod* replicas are running at any one time. It both allows for easy scaling of replicated systems and handles re-creation of a *pod* when the machine it is on reboots or otherwise fails.

Service: Kubernetes Services A service defines a set of *pods* and a means by which to access them, such as single stable IP address and corresponding DNS name. Kubernetes *pods* are mortal. They are born and they die, and they are **not resurrected**. Replication Controllers in particular create and destroy *pods* dynamically (e.g. when scaling up or down or when doing rolling updates). While each *pod* gets its own IP address, even those IP addresses cannot be relied upon to be stable over time. This leads to a problem: if some set of *pods* (let's call them backends) provides functionality to other *pods* (let's call them frontends) inside the Kubernetes cluster, how do those frontends find out and keep track of which backends are in that set? Enter Services. A Kubernetes *service* is an abstraction which defines a logical set of *pods* and a policy by which to access them - sometimes called a micro-service. The set of *pods* targeted by a *service* is (usually) determined by a *label selector*

Volume: Kuebernetes volume A volume is a directory, possibly with some data in it, which is accessible to a Container as part of its filesystem. Kubernetes volumes build upon Docker Volumes, adding provisioning of the volume directory and/or device.

3.1.2 Kubernetes Networking Overview

This is an extract from Networking in Kubernetes

Summary

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. We give every pod its own IP address so you do not need to explicitly create links between pods and you almost never need to deal with mapping container ports to host ports. This creates a clean, backwards-compatible model where pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration

Docker Model

Before discussing the Kubernetes approach to networking, it is worthwhile to review the "normal" way that networking works with Docker.

By default, Docker uses host-private networking. It creates a virtual bridge, called docker0 by default, and allocates a subnet from one of the private address blocks defined in RFC1918 for that bridge. For each container that Docker creates, it allocates a virtual ethernet device (called veth) which is attached to the bridge. The veth is mapped to appear as eth0 in the container, using Linux namespaces. The in-container eth0 interface is given an IP address from the bridge's address range. The result is that Docker containers can talk to other containers only if they are on the same machine (and thus the same virtual bridge). Containers on different machines can not reach each other - in fact they may end up with the exact same network ranges and IP addresses. In order for Docker containers to communicate across nodes, they must be allocated ports on the machine's own IP address, which are then forwarded or proxied to the containers. This obviously means that containers must either coordinate which ports they use very carefully or else be allocated ports dynamically.

Kubernetes Model

Coordinating ports across multiple containers is very difficult to do at scale and exposes users to cluster-level issues outside of their control. Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):

- All containers can communicate with all other containers without NAT
- All nodes can communicate with all containers (and vice-versa) without NAT
- The IP that a container sees itself as is the same IP that others see it as

- What this means in practice is that you can not just take two computers running Docker and expect Kubernetes to work. You must ensure that the fundamental requirements are met.

Kubernetes applies IP addresses at the *Pod* scope - containers within a Pod share their network namespaces - including their IP address. This means that containers within a Pod can all reach each other's ports on **localhost**. This does imply that containers within a Pod must coordinate port usage, but this is no different than processes in a VM. We call this the *IP-per-pod* model. This is implemented in Docker as a *pod container* which holds the network namespace open while "app containers" (the things the user specified) join that namespace with Docker's **–net=container:<id>** function

How to achieve this

There are a number of ways that this network model can be implemented. Here is a list of possible options:

- Contiv Netplugin
- Flannel
- Open vSwitch
- Calico
- Romana
- Weave Net
- L2 networks and linux bridging

Important: For this lab, we will use **Flannel**.

3.1.3 Kubernetes Services Overview

Refer to Kubernetes services for more information

A Kubernetes *service* is an abstraction which defines a logical set of *pods* and a policy by which to access them. The set of *pods* targeted by a *service* is (usually) determined by a *label selector*.

As an example, consider an image-processing backend which is running with 3 replicas. Those replicas are fungible - frontends do not care which backend they use. While the actual *pods* that compose the backend set may change, the frontend clients should not need to be aware of that or keep track of the list of backends themselves. The *service* abstraction enables this decoupling.

For Kubernetes-native applications, Kubernetes offers a simple *Endpoints API* that is updated whenever the set of *pods* in a *service* changes. For non-native applications, Kubernetes offers a virtual-IP-based bridge to services* which redirects to the backend *pods*.

Defining a service

A *service* in Kubernetes is a REST object, similar to a *pod*. Like all of the REST objects, a *service* definition can be *POSTed* to the *apiserver* to create a new instance. For example, suppose you have a set of *pods* that each expose port 9376 and carry a *label* "app=MyApp".

```
{
   "kind": "Service",
   "apiVersion": "v1",
   "metadata": {
```

```
      "name": "my-service"
   },
   "spec": {
      "selector": {
         "app": "MyApp"
      },
      "ports": [
         {
            "protocol": "TCP",
            "port": 80,
            "targetPort": 9376
         }
      ]
   }
}
```

This specification will create a new *service* object named “my-service” which targets TCP port 9376 on any *pod* with the “app=MyApp” *label*.

This *service* will also be assigned an IP address (sometimes called the *cluster IP*), which is used by the *service proxies* . The *service’s selector* will be evaluated continuously and the results will be POSTed to an *Endpoints* object also named “my-service”.

If the service is not a native kubernetes app, then you can do a service definition without the *selector* field. In such a case you’ll have to specify yourself the *endpoints*

```
{
   "kind": "Service",
   "apiVersion": "v1",
   "metadata": {
      "name": "my-service"
   },
   "spec": {
      "ports": [
         {
            "protocol": "TCP",
            "port": 80,
            "targetPort": 9376
         }
      ]
   }
}

{
   "kind": "Endpoints",
   "apiVersion": "v1",
   "metadata": {
      "name": "my-service"
   },
   "subsets": [
      {
         "addresses": [
            { "ip": "1.2.3.4" }
         ],
         "ports": [
            { "port": 9376 }
         ]
      }
```

```
    ]
}
```

Note: A *service* can map an incoming port to any *targetPort*. By default the *targetPort* will be set to the same value as the *port* field. In the example above, the port for the service is 80 (HTTP) and will redirect traffic to port 9376 on the Pods

---

You can specify multiple ports if needed (like HTTP/HTTPS for an app)

Kubernetes *service* supports TCP (default) and UDP.

Publishing services - service types

For some parts of your application (e.g. frontends) you may want to expose a *Service* onto an external (outside of your cluster, maybe public internet) IP address, other services should be visible only from inside of the cluster.

Kubernetes ServiceTypes allow you to specify what kind of *service* you want. **The default and base type is *ClusterIP*, which exposes a *service* to connection from inside the cluster**. NodePort and LoadBalancer are two types that expose services to external traffic.

Valid values for the ServiceType field are:

- **ExternalName**: map the *service* to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up. This requires version 1.7 or higher of kube-dns.
- **ClusterIP**: use a cluster-internal IP only - this is the default and is discussed above. Choosing this value means that you want this *service* to be reachable only from inside of the *cluster*.
- **NodePort**: on top of having a cluster-internal IP, expose the *service* on a port on each node of the cluster (the same port on each *node*). You'll be able to contact the service on any <NodeIP>:NodePort address. If you set the type field to "NodePort", the Kubernetes master will allocate a port from a flag-configured range **(default: 30000-32767)**, and each Node will proxy that port (the same port number on every Node) into your *Service*. That port will be reported in your Service's spec.ports[*].nodePort field. If you want a specific port number, you can specify a value in the nodePort field, and the system will allocate you that port or else the API transaction will fail (i.e. you need to take care about possible port collisions yourself). **The value you specify must be in the configured range for node ports**.
- **LoadBalancer**: on top of having a cluster-internal IP and exposing service on a NodePort also, ask the cloud provider for a load balancer which forwards to the Service exposed as a <NodeIP>:NodePort for each Node

Service type: LoadBalancer

On cloud providers which support external load balancers, setting the type field to "LoadBalancer" will provision a load balancer for your *Service*. The actual creation of the load balancer happens asynchronously, and information about the provisioned balancer will be published in the Service's status.loadBalancer field. For example:

```
{
   "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
```

```
        "name": "my-service"
    },
    "spec": {
        "selector": {
            "app": "MyApp"
        },
        "ports": [
            {
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376,
                "nodePort": 30061
            }
        ],
        "clusterIP": "10.0.171.239",
        "loadBalancerIP": "78.11.24.19",
        "type": "LoadBalancer"
    },
    "status": {
        "loadBalancer": {
            "ingress": [
                {
                    "ip": "146.148.47.155"
                }
            ]
        }
    }
}
```

Traffic from the external load balancer will be directed at the backend *Pods*, though exactly how that works depends on the cloud provider (AWS, GCE, ...). Some cloud providers allow the loadBalancerIP to be specified. In those cases, the load-balancer will be created with the user-specified loadBalancerIP. If the loadBalancerIP field is not specified, an ephemeral IP will be assigned to the loadBalancer. If the loadBalancerIP is specified, but the cloud provider does not support the feature, the field will be ignored

Service proxies

Every node in a Kubernetes cluster runs a *kube-proxy*. *kube-proxy* is responsible for implementing a form of virtual IP for *Services*

Since Kubernetes 1.2, the iptables proxy is the default behavior (another implementation of kube-proxy is the userspace implementation)

In this mode, *kube-proxy* watches the Kubernetes *master* for the addition and removal of *Service* and *Endpoints* objects. For each*Service*, it installs iptables rules which capture traffic to the *Service*'s *cluster IP* (which is virtual) and *Port* and redirects that traffic to one of the *Service*'s backend sets. For each *Endpoints* object, it installs iptables rules which select a backend *Pod*.

By default, the choice of backend is random. Client-IP based session affinity can be selected by setting **service.spec.sessionAffinity** to "ClientIP" (the default is "None").

As with the userspace proxy, the net result is that any traffic bound for the *Service*'s IP:Port is proxied to an appropriate backend without the clients knowing anything about Kubernetes or *Services* or *Pods*. This should be faster and more reliable than the userspace proxy. However, unlike the userspace proxier, the iptables proxier cannot automatically retry another *Pod* if the one it initially selects does not respond, so it depends on having working *readiness probes*. A readiness probe gives you the capability to monitor the status of a *pod* via health-checks

Service discovery

The recommended way to implement Service discovery with Kubernetes is the same as with Mesos: DNS

when building a cluster, you can add *add-on* to it. One of the available *add-on* is a DNS Server.

The DNS server watches the Kubernetes API for new *Services* and creates a set of DNS records for each. If DNS has been enabled throughout the cluster then all *Pods* should be able to do name resolution of Services automatically.

For example, if you have a *Service* called "my-service" in Kubernetes Namespace "my-ns" a DNS record for "my-service.my-ns" is created. *Pods* which exist in the "my-ns" Namespace should be able to find it by simply doing a name lookup for "my-service". *Pods* which exist in other Namespaces must qualify the name as "my-service.my-ns". The result of these name lookups is the *cluster IP*.

Kubernetes also supports DNS SRV (service) records for named ports. If the "my-service.my-ns" *Service* has a port named "http" with protocol TCP, you can do a DNS SRV query for "_http._tcp.my-service.my-ns" to discover the port number for "http"

3.2 Module 2: Build a Kubernetes Cluster

In this module, we will build a 3 node cluster (1x master and 2x nodes) utilizing Ubuntu server images.

As a reminder, in this module, our cluster setup is:

Hostname	IP-ADDR	Role
kube-master1	10.1.10.21	Master
kube-node1	10.1.10.22	Node
kube-node2	10.1.10.23	Node

3.2.1 Lab 2.1 - Prep Ubuntu

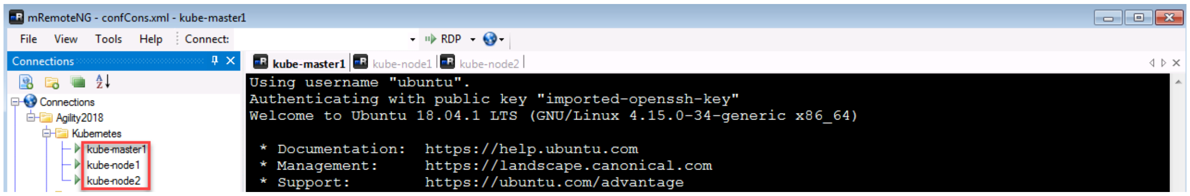
Note: This installation will utilize Ubuntu v18.04 (Bionic)

Important: The following commands need to be run on all three nodes unless otherwise specified.

1. From the jumpbox open **mRemoteNG** and start a session to each of the following servers. The sessions are pre-configured to connect with the default user "ubuntu".

    - kube-master1
    - kube-node1
    - kube-node2

    Tip: These sessions should be running from the previous Docker lab.



2. If not already done from the previous Docker lab elevate to "root"

```
su -

#When prompted for password enter "default" without the quotes
```

Your prompt should change to root@ at the start of the line :

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ su -
Password:
root@kube-master1:~#
```

3. For your convenience we've already added the host IP & names to /etc/hosts. Verify the file

```
cat /etc/hosts
```

The file should look like this:

```
root@kube-master1:~# cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       ubuntu.localdomain      ubuntu

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

10.1.10.21 kube-master1
10.1.10.22 kube-node1
10.1.10.23 kube-node2
root@kube-master1:~#
```

If entries are not there add them to the bottom of the file be editing "/etc/hosts" with 'vim'

```
vim /etc/hosts

#cut and paste the following lines to /etc/hosts

10.1.10.21    kube-master1
10.1.10.22    kube-node1
10.1.10.23    kube-node2
```

4. The linux swap file needs to be disabled, this is not the case by default. Again for your convenience we disabled swap. Verify the setting

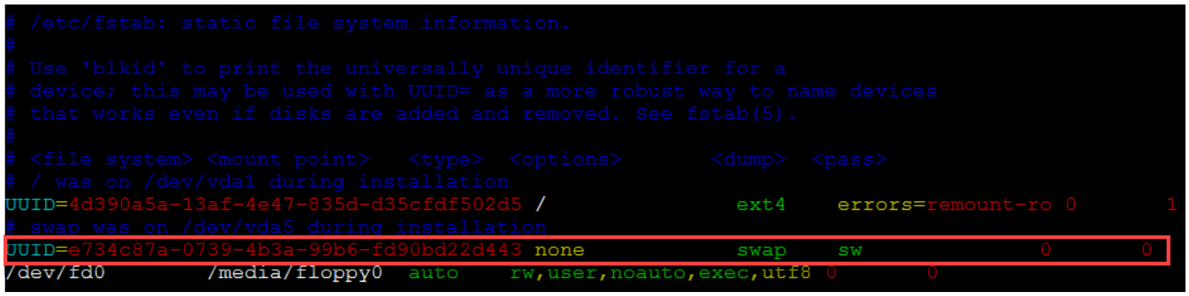
Important: Running a swap file is incompatible with Kubernetes. Lets use the linux top command, which allows users to monitor processes and system resource usage

```
top
```

```
top - 17:34:21 up 11:36,  1 user,  load average: 1.32, 1.12, 1.17
Tasks: 118 total,   2 running,  85 sleeping,   0 stopped,   0 zombie
%Cpu(s):  5.7 us, 13.5 sy,  0.0 ni, 78.8 id,  0.0 wa,  0.0 hi,  0.7 si,  1.3 st
KiB Mem :  4039472 total,   910760 free,   659572 used,  2469140 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  3088872 avail Mem
```

If you see a number other than "0" you need to run the following commands (press 'q' to quit top)

```
swapoff -a

vim /etc/fstab

#rem out the highlighted line below by adding "#" to the beginning of the line, write and save the file by typing ":wq"
```



```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump>  <pass>
# / was on /dev/vda1 during installation
UUID=4d390a5a-13af-4e47-835d-d35cfdf502d5 /               ext4    errors=remount-ro 0       1
# swap was on /dev/vda5 during installation
UUID=e734c87a-0739-4b3a-99b6-fd90bd22d443 none            swap    sw              0       0
/dev/fd0        /media/floppy0  auto    rw,user,noauto,exec,utf8 0       0
```

5. Ensure the OS is up to date, run the following command

Tip: You can skip this step if it was done in the previous Docker lab.

```
apt update && apt upgrade -y

#This can take a few seconds to several minute depending on demand to download the latest updates for the OS.
```

6. Install docker-ce

Attention: This was done earlier in Class 1 / Module1 / Lab 1.1: Install Docker . If skipped go back and install Docker by clicking the link.

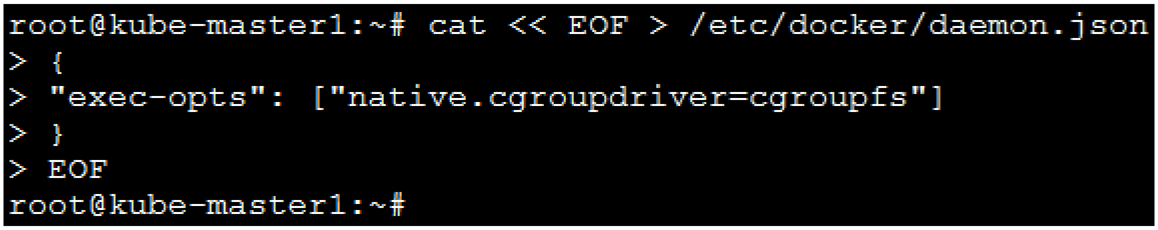
7. Configure docker to use the correct cgroupdriver

Important: The cgroupdrive for docker and kubernetes have to match. In this lab "cgroupfs" is the correct driver.

Note: This next part can be a bit tricky - just copy/paste the 5 lines below exactly as they are and paste via buffer to the CLI (and press return when done)

```
cat << EOF > /etc/docker/daemon.json
{
"exec-opts": ["native.cgroupdriver=cgroupfs"]
}
EOF
```

It should look something like this image below:



```
root@kube-master1:~# cat << EOF > /etc/docker/daemon.json
> {
> "exec-opts": ["native.cgroupdriver=cgroupfs"]
> }
> EOF
root@kube-master1:~#
```

8. Add the kubernetes repo

```
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
```

9. Install the kubernetes packages

```
apt update && apt install kubelet kubeadm kubectl -y
```

Limitations

See also:

For a full list of the limitations go here: kubeadm limitations

---

Important: The cluster created has a single master, with a single etcd database running on it. This means that if the master fails, your cluster loses its configuration data and will need to be recreated from scratch.

3.2.2 Lab 2.2 - Setup the Master

The master is the system where the "control plane" components run, including etcd (the cluster database) and the API server (which the kubectl CLI communicates with). All of these components run in pods started by kubelet (which is why we had to setup docker first even on the master node)

Important: The following commands need to be run on the **master** only unless otherwise specified.

---

1. Switch back to the ssh session connected to kube-master1

   Tip: This session should be running from the previous if lab. If not simply open **mRemoteNG** and connect via the saved session.

2. Initialize kubernetes

```
kubeadm init --apiserver-advertise-address=10.1.10.21 --pod-network-cidr=10.244.0.
↪0/16
```

Note:

- The IP address used to advertise the master. 10.1.10.0/24 is the network for our control plane. if you don't specify the –apiserver-advertise-address argument, kubeadm will pick the first interface with a default gateway (because it needs internet access).
- 10.244.0.0/16 is the default network used by flannel. We'll setup flannel in a later step.
- Be patient this step takes a few minutes. The initialization is successful if you see "Your Kubernetes master has initialized successfully!".

```
Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular user:

  mkdir -p $HOME/.kube
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
  sudo chown $(id -u):$(id -g) $HOME/.kube/config

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/

You can now join any number of machines by running the following on each node
as root:

  kubeadm join 10.1.10.21:6443 --token o3wtpq.8wnefwyt44xgz63o --discovery-token-ca-cert-hash sha256:d61b81374b20262c400eba37905c2fe91a559bb15f87fab80f9b1e74c885aa58

root@kube-master1:~#
```

Important:

- Be sure to save the highlighted output from this command to notepad. You'll need this information to add your worker nodes and configure user administration.
- The "kubeadm join" command is run on the nodes to register themselves with the master. Keep the secret safe since anyone with this token can add an authenticated node to your cluster. This is used for mutual auth between the master and the nodes.

---

3. Configure kubernetes administration. At this point you should be logged in as root. The following will update both root and ubuntu user accounts for kubernetes administration.

```
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
exit
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
cd $HOME
```

4. Verify kubernetes is up and running. You can monitor the services are running by using the following command.

```
kubectl get pods --all-namespaces
```

You'll need to run this several times until you see several containers "Running" It should look like the following:

```
ubuntu@kube-master1:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                   READY   STATUS    RESTARTS   AGE
kube-system   coredns-78fcdf6894-6k56m               0/1     Pending   0          1m
kube-system   coredns-78fcdf6894-cvk8h               0/1     Pending   0          1m
kube-system   etcd-kube-master1                      1/1     Running   0          54s
kube-system   kube-apiserver-kube-master1            1/1     Running   0          1m
kube-system   kube-controller-manager-kube-master1   1/1     Running   0          1m
kube-system   kube-proxy-c79zn                       1/1     Running   0          1m
kube-system   kube-scheduler-kube-master1            1/1     Running   0          51s
ubuntu@kube-master1:~$
```

Note: corends won't start until the network pod is up and running.

5. Install flannel

```
kubectl apply -f https://raw.githubusercontent.com/coreos/flannel/master/
↪Documentation/kube-flannel.yml
```

Note: You must install a *pod* network add-on so that your *pods* can communicate with each other. **It is necessary to do this before you try to deploy any applications to your cluster**, and before "coredns" will start up.

6. If everything installs and starts as expected you should have "coredns" and all services status "Running". To check the status of core services, you can run the following command:

```
kubectl get pods --all-namespaces
```

The output should show all services as running.

```
ubuntu@kube-master1:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                   READY   STATUS    RESTARTS   AGE
kube-system   coredns-78fcdf6894-6k56m               1/1     Running   0          3m
kube-system   coredns-78fcdf6894-cvk8h               1/1     Running   0          3m
kube-system   etcd-kube-master1                      1/1     Running   0          2m
kube-system   kube-apiserver-kube-master1            1/1     Running   0          2m
kube-system   kube-controller-manager-kube-master1   1/1     Running   0          2m
kube-system   kube-flannel-ds-amd64-54d4q            1/1     Running   0          15s
kube-system   kube-proxy-c79zn                       1/1     Running   0          3m
kube-system   kube-scheduler-kube-master1            1/1     Running   0          2m
ubuntu@kube-master1:~$
```

Important: Before moving to the next lab, "Setup the Nodes" wait for all system pods to show status "Running".

7. Additional kubernetes status checks.

```
kubectl get cs
```

```
ubuntu@kube-master1:~$ kubectl get cs
NAME                 STATUS    MESSAGE              ERROR
scheduler            Healthy   ok
controller-manager   Healthy   ok
etcd-0               Healthy   {"health": "true"}
ubuntu@kube-master1:~$
```

```
kubectl cluster-info
```

```
ubuntu@kube-master1:~$ kubectl cluster-info
Kubernetes master is running at https://10.1.10.21:6443
KubeDNS is running at https://10.1.10.21:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
ubuntu@kube-master1:~$
```

Hint: If you made a mistake and need to re-initialize the cluster run the following commands:

```
# If you followed the instructions you should be currently connected as user␣
↪**ubuntu**
# When prompted for password enter "default" without the quotes
su -

# This resets the master to default settings
kubeadm reset --force

# This removes the admin references to the broken cluster
rm -rf /home/ubuntu/.kube /root/.kube
```

3.2.3 Lab 2.3 - Setup the Nodes

Once the master is setup and running, we need to join our *nodes* to the cluster.

Important: The following commands need to be run on the worker **nodes only** unless otherwise specified.

1. To join the master we need to run the command highlighted during the master initialization. You'll need to use the command saved to notepad in an earlier step.

   Warning:

   - This following is just an example!! **DO not cut/paste the one below.** You should have saved this command after successfully initializing the master in the previous lab. Scroll up in your CLI history to find the hash your kube-master1 generated to add nodes.
   - This command needs to be run on **node1** and **node2** only!

   Hint: If you missed the step to save the "kubeadm join. . . " command from the previous lab, run the following and use the output to join your nodes to the cluster.

   ```
   kubeadm token create --print-join-command
   ```

   ```
   kubeadm join 10.1.10.21:6443 --token 12rmdx.z0cbklfaoixhhdfj --discovery-token-ca-
   ↪cert-hash␣
   ↪sha256:c624989e418d92b8040a1609e493c009df5721f4392e90ac6b066c304cebe673
   ```

   The output should be similar to this:

```
root@kube-node1:~# kubeadm join 10.1.10.21:6443 --token o3wtpq.8wnefwyt44xgz63o --discovery-token-ca-cert-hash
sha256:d61b81374b20262c400eba37905c2fe91a559bb15f87fab80f9b1e74c885aa58
[preflight] running pre-flight checks
I0926 18:24:56.528047   31568 kernel_validator.go:81] Validating kernel version
I0926 18:24:56.528371   31568 kernel_validator.go:96] Validating kernel config
        [WARNING SystemVerification]: docker version is greater than the most recently validated version. Docke
r version: 18.06.1-ce. Max validated version: 17.03
[discovery] Trying to connect to API Server "10.1.10.21:6443"
[discovery] Created cluster-info discovery client, requesting info from "https://10.1.10.21:6443"
[discovery] Requesting info from "https://10.1.10.21:6443" again to validate TLS against the pinned public key
[discovery] Cluster info signature and contents are valid and TLS certificate validates against pinned roots, w
ill use API Server "10.1.10.21:6443"
[discovery] Successfully established connection with API Server "10.1.10.21:6443"
[kubelet] Downloading configuration for the kubelet from the "kubelet-config-1.11" ConfigMap in the kube-system
 namespace
[kubelet] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[preflight] Activating the kubelet service
[tlsbootstrap] Waiting for the kubelet to perform the TLS Bootstrap...
[patchnode] Uploading the CRI Socket information "/var/run/dockershim.sock" to the Node API object "kube-node1"
 as an annotation

This node has joined the cluster:
* Certificate signing request was sent to master and a response
  was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the master to see this node join the cluster.
root@kube-node1:~#
```

2. To verify the *nodes* have joined the cluster, run the following command on the **kube-master1**:

```
kubectl get nodes
```

You should see your cluster (ie *master* + *nodes*)

```
ubuntu@kube-master1:~$ kubectl get nodes
NAME           STATUS    ROLES     AGE       VERSION
kube-master1   Ready     master    8m        v1.11.3
kube-node1     Ready     <none>    1m        v1.11.3
kube-node2     Ready     <none>    1m        v1.11.3
ubuntu@kube-master1:~$
```

3. Verify all the services are started as expected (run on the **kube-master1**) Don't worry about last 5 characters matching on most services, as they are randomly generated:

```
kubectl get pods --all-namespaces
```

```
ubuntu@kube-master1:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                   READY     STATUS    RESTARTS   AGE
kube-system   coredns-78fcdf6894-6k56m               1/1       Running   0          9m
kube-system   coredns-78fcdf6894-cvk8h               1/1       Running   0          9m
kube-system   etcd-kube-master1                      1/1       Running   0          8m
kube-system   kube-apiserver-kube-master1            1/1       Running   0          8m
kube-system   kube-controller-manager-kube-master1   1/1       Running   0          8m
kube-system   kube-flannel-ds-amd64-54d4q            1/1       Running   0          6m
kube-system   kube-flannel-ds-amd64-64vp7            1/1       Running   0          1m
kube-system   kube-flannel-ds-amd64-8hfnz            1/1       Running   0          2m
kube-system   kube-proxy-57pxx                       1/1       Running   0          1m
kube-system   kube-proxy-c79zn                       1/1       Running   0          9m
kube-system   kube-proxy-ksc6x                       1/1       Running   0          2m
kube-system   kube-scheduler-kube-master1            1/1       Running   0          8m
ubuntu@kube-master1:~$
```

Attention: CONGRATUATIONS! You just did the hardest part of todays lab - building a Kubernetes cluster. While we didn't cover each step in great detail, due to time of other labs we need to complete today, this is one path to the overall steps to build your own cluster with a few linux boxes in your own lab. All this content is publicly online/available at clouddocs.f5.com.

3.2.4 Lab 2.4 - Setup the Kubernetes UI

Important: The following commands need to be run on the **master** only.

Note: You have two options to install the UI:

1. Run the included script from the cloned git repo.
2. Manually run each command.

Both options are included below.

1. "git" the demo files

   Note: These files should be here by default, if **NOT** run the following commands.

```
git clone https://github.com/f5devcentral/f5-agility-labs-containers.git ~/
↪agilitydocs

cd ~/agilitydocs/kubernetes
```

2. Run the following commands to configure the UI

   Note: A script is included in the cloned git repo from the previous step. In the interest of time you can simply use the script.

```
cd /home/ubuntu/agilitydocs/kubernetes

./create-kube-dashboard
```

or run through the following steps:

```
kubectl create serviceaccount kubernetes-dashboard -n kube-system

kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-
↪admin --serviceaccount=kube-system:kubernetes-dashboard
```

Warning: These commands create a service account with full admin rights. In a typical deployment this would be overkill.

Create a file called kube-dashboard.yaml with the following content:

```
# ------------------- Dashboard Deployment ------------------- #

kind: Deployment
apiVersion: apps/v1beta2
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
```

```yaml
  namespace: kube-system
spec:
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      k8s-app: kubernetes-dashboard
  template:
    metadata:
      labels:
        k8s-app: kubernetes-dashboard
    spec:
      containers:
      - name: kubernetes-dashboard
        image: k8s.gcr.io/kubernetes-dashboard-amd64:v1.10.0
        ports:
        - containerPort: 9090
          protocol: TCP
        args:
          # Uncomment the following line to manually specify Kubernetes API server Host
          # If not specified, Dashboard will attempt to auto discover the API server and connect
          # to it. Uncomment only if the default does not work.
          # - --apiserver-host=http://my-address:port
        volumeMounts:
          # Create on-disk volume to store exec logs
        - mountPath: /tmp
          name: tmp-volume
        livenessProbe:
          httpGet:
            path: /
            port: 9090
          initialDelaySeconds: 30
          timeoutSeconds: 30
      volumes:
      - name: tmp-volume
        emptyDir: {}
      serviceAccountName: kubernetes-dashboard
      # Comment the following tolerations if Dashboard must not be deployed on master
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule

---
# ------------------- Dashboard Service ------------------- #

kind: Service
apiVersion: v1
metadata:
  labels:
    k8s-app: kubernetes-dashboard
  name: kubernetes-dashboard
  namespace: kube-system
spec:
  ports:
  - port: 80
```

```
    targetPort: 9090
  type: NodePort
  selector:
    k8s-app: kubernetes-dashboard
```

Apply Kubernetes manifest file:

```
kubectl apply -f kube-dashboard.yaml
```

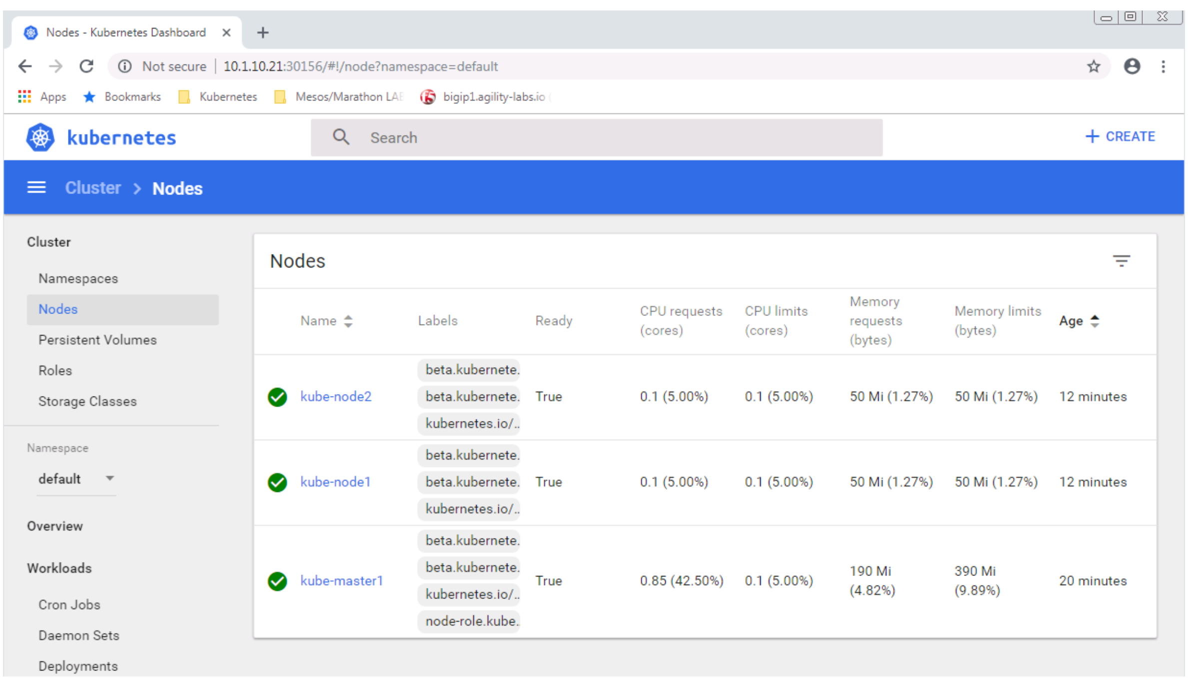
3. To access the dashboard, you need to see which port it is listening on. You can find this information with the following command:

```
kubectl describe svc kubernetes-dashboard -n kube-system
```

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl describe svc kubernetes-dashboard -n kube-system
Name:                     kubernetes-dashboard
Namespace:                kube-system
Labels:                   k8s-app=kubernetes-dashboard
Annotations:              <none>
Selector:                 k8s-app=kubernetes-dashboard
Type:                     NodePort
IP:                       10.108.186.58
Port:                     <unset>  80/TCP
TargetPort:               9090/TCP
NodePort:                 <unset>  30156/TCP
Endpoints:                10.244.2.4:9090
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

Note: In our service we are assigned port "30156" (NodePort), you'll be assigned a different port.

We can now access the dashboard by connecting to the following uri http://10.1.10.21:30156



3.3 Module 3: F5 Container Connector with Kubernetes

3.3.1 Overview

The Container Connector makes L4-L7 services available to users deploying microservices-based applications in a containerized infrastructure. The CC - Kubernetes allows you to expose a Kubernetes Service outside the cluster as a virtual server on a BIG-IP® device entirely through the Kubernetes API.

See also:

The official F5 documentation is here: F5 Container Connector - Kubernetes

3.3.2 Architecture

The Container Connector for Kubernetes comprises the *f5-k8s-controller* and user-defined "F5 resources". The *f5-k8s-controller* is a Docker container that can run in a *Kubernetes Pod*. The "F5 resources" are *Kubernetes ConfigMap* resources that pass encoded data to the f5-k8s-controller. These resources tell the f5-k8s-controller:

- What objects to configure on your BIG-IP.
- What *Kubernetes Service* the BIG-IP objects belong to (the frontend and backend properties in the *ConfigMap*, respectively).

The f5-k8s-controller watches for the creation and modification of F5 resources in Kubernetes. When it discovers changes, it modifies the BIG-IP accordingly. For example, for an F5 virtualServer resource, the CC - Kubernetes does the following:

- Creates objects to represent the virtual server on the BIG-IP in the specified partition.
- Creates pool members for each node in the Kubernetes cluster, using the NodePort assigned to the service port by Kubernetes.
- Monitors the F5 resources and linked Kubernetes resources for changes and reconfigures the BIG-IP accordingly.
- The BIG-IP then handles traffic for the Service on the specified virtual address and load-balances to all nodes in the cluster.
- Within the cluster, the allocated NodePort is load-balanced to all pods for the Service.

3.3.3 Prerequisites

Before being able to use the F5 Container Connector, you need to confirm the following:

- You must have a fully active/licensed BIG-IP
- A BIG-IP partition needs to be setup for the Container Connector.
- You need a user with administrative access to this partition
- Your kubernetes environment must be up and running already

Lab 3.1 - F5 Container Connector Setup

The BIG-IP Controller for Kubernetes installs as a Deployment object

See also:

The official CC documentation is here: Install the BIG-IP Controller: Kubernetes

BIG-IP Setup

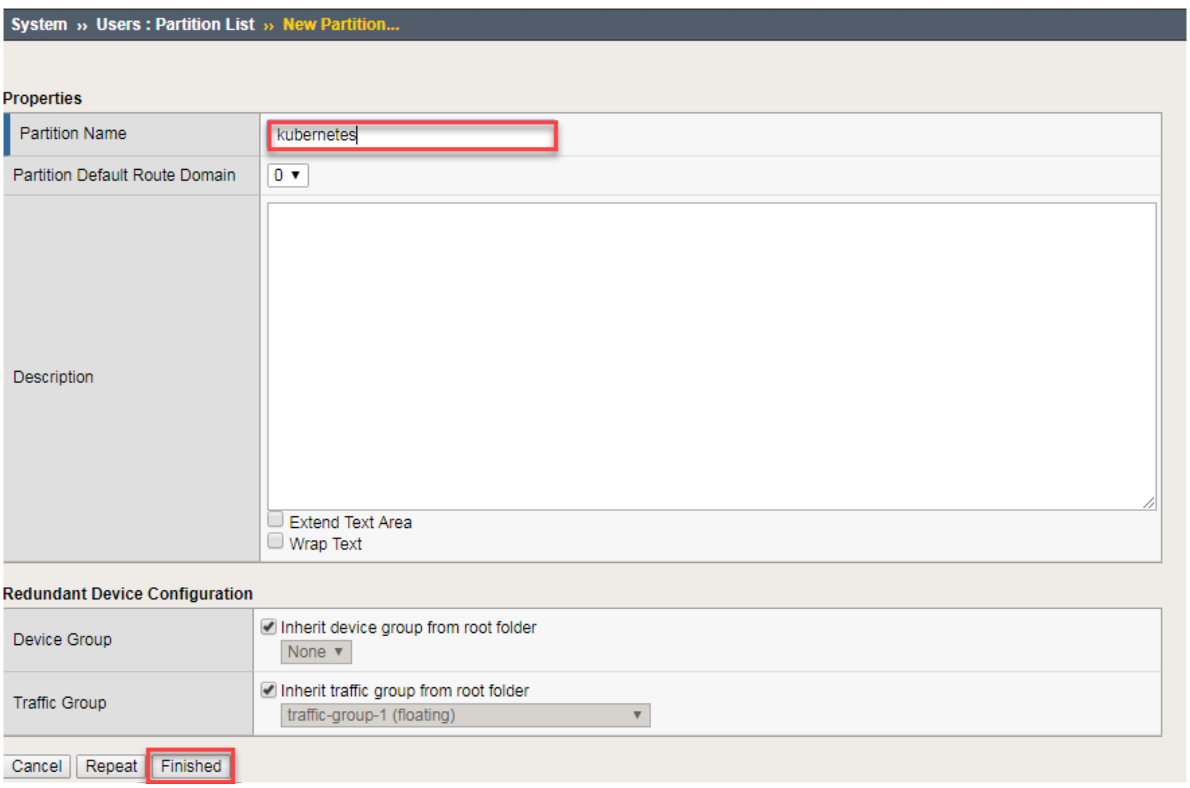
To use F5 Container connector, you'll need a BIG-IP up and running first.

Through the Jumpbox, you should have a BIG-IP available at the following URL: https://10.1.1.245

> **Warning:** Connect to your BIG-IP and check it is active and licensed. Its login and password are: **admin/admin**
>
> If your BIG-IP has no license or its license expired, renew the license. You just need a LTM VE license for this lab. No specific add-ons are required (ask a lab instructor for eval licenses if your license has expired)

1. You need to setup a partition that will be used by F5 Container Connector.

```
# From the CLI:
tmsh create auth partition kubernetes

# From the UI:
GoTo System --> Users --> Partition List
- Create a new partition called "kubernetes" (use default settings)
- Click Finished
```



With the new partition created, we can go back to Kubernetes to setup the F5 Container connector.

Container Connector Deployment

See also:

For a more thorough explanation of all the settings and options see F5 Container Connector - Kubernetes

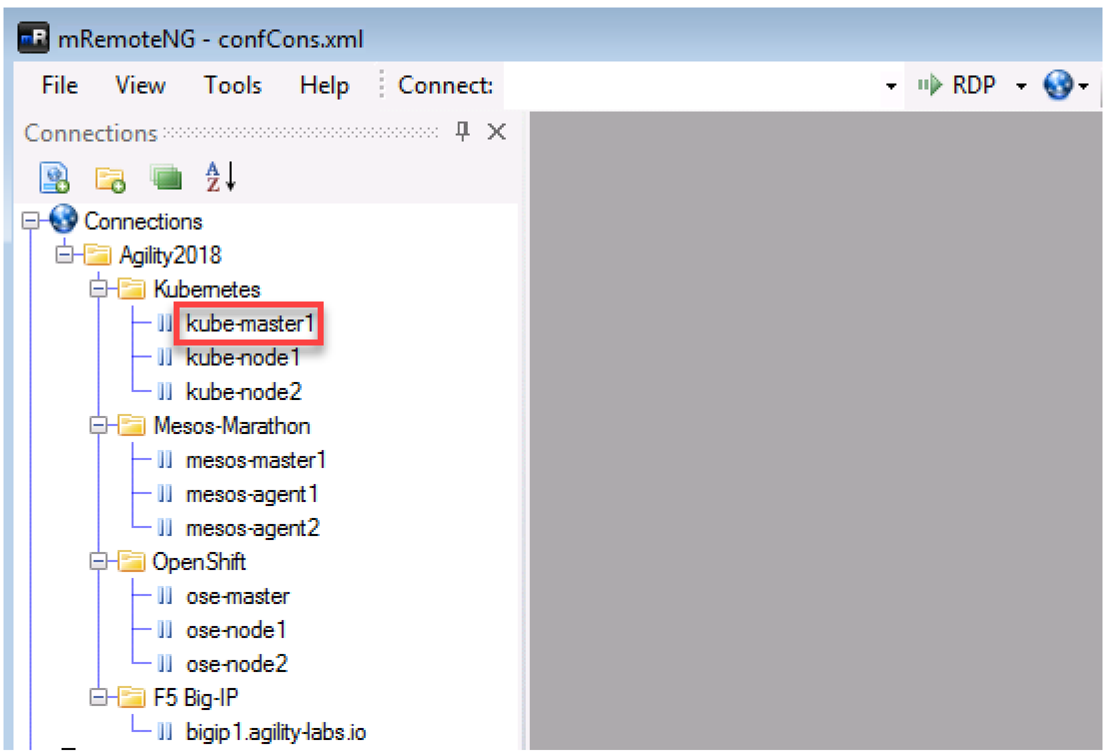
Now that BIG-IP is licensed and prepped with the "kubernetes" partition, we need to define a Kubernetes deployment and create a Kubernetes secret to hide our bigip credentials.

1. From the jumpbox open **mRemoteNG** and start a session with Kube-master.

   Tip:

   - These sessions should be running from the previous lab.
   - As a reminder we're utilizing a wrapper called **MRemoteNG** for Putty and other services. MRNG hold credentials and allows for multiple protocols(i.e. SSH, RDP, etc.), makes jumping in and out of SSH connections easier.

   On your desktop select **MRemoteNG**, once launched you'll see a few tabs similar to the example below. Open up the Kubernetes / Kubernetes-Cluster folder and double click kube-master1.

2. "git" the demo files

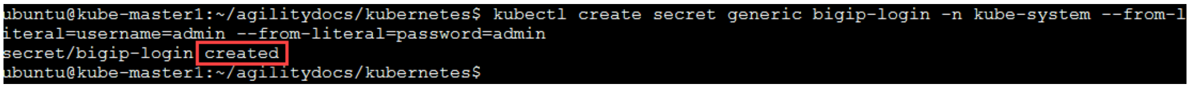
   Note: These files should be here by default, if **NOT** run the following commands.

```
git clone https://github.com/f5devcentral/f5-agility-labs-containers.git ~/
↪agilitydocs

cd ~/agilitydocs/kubernetes
```

3. Create bigip login secret

```
kubectl create secret generic bigip-login -n kube-system --from-
↪literal=username=admin --from-literal=password=admin
```

You should see something similar to this:

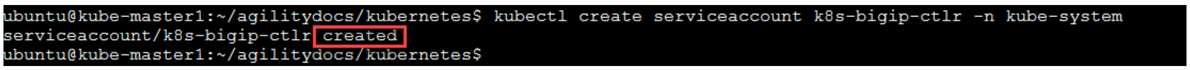


```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl create secret generic bigip-login -n kube-system --from-l
iteral=username=admin --from-literal=password=admin
ecret/bigip-login created
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

4. Create kubernetes service account for bigip controller

```
kubectl create serviceaccount k8s-bigip-ctlr -n kube-system
```

You should see something similar to this:

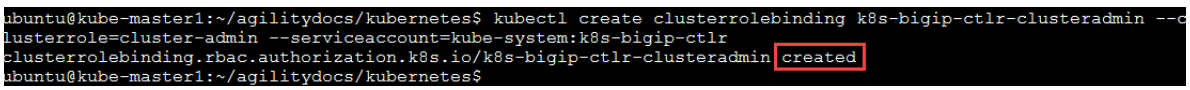


```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl create serviceaccount k8s-bigip-ctlr -n kube-system
serviceaccount/k8s-bigip-ctlr created
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

5. Create cluster role for bigip service account (admin rights, but can be modified for your environment)

```
kubectl create clusterrolebinding k8s-bigip-ctlr-clusteradmin --
↪clusterrole=cluster-admin --serviceaccount=kube-system:k8s-bigip-ctlr
```

You should see something similar to this:



```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl create clusterrolebinding k8s-bigip-ctlr-clusteradmin --c
lusterrole=cluster-admin --serviceaccount=kube-system:k8s-bigip-ctlr
clusterrolebinding.rbac.authorization.k8s.io/k8s-bigip-ctlr-clusteradmin created
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

6. At this point we have two deployment mode options, Nodeport or Cluster. For more information see BIG-IP Controller Modes

Important: This lab will focus on **Nodeport**. In Class 4 Openshift we'll use **ClusterIP**.

7. **Nodeport mode** `f5-nodeport-deployment.yaml`

Note:

- For your convenience the file can be found in /home/ubuntu/agilitydocs/kubernetes (downloaded earlier in the clone git repo step).
- Or you can cut and paste the file below and create your own file.
- If you have issues with your yaml and syntax (**indentation MATTERS**), you can try to use an online parser to help you : Yaml parser

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: k8s-bigip-ctlr-deployment
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      name: k8s-bigip-ctlr
```

```
      labels:
        app: k8s-bigip-ctlr
    spec:
      serviceAccountName: k8s-bigip-ctlr
      containers:
        - name: k8s-bigip-ctlr
          image: "f5networks/k8s-bigip-ctlr:latest"
          imagePullPolicy: IfNotPresent
          env:
            - name: BIGIP_USERNAME
              valueFrom:
                secretKeyRef:
                  name: bigip-login
                  key: username
            - name: BIGIP_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: bigip-login
                  key: password
          command: ["/app/bin/k8s-bigip-ctlr"]
          args: [
            "--bigip-username=$(BIGIP_USERNAME)",
            "--bigip-password=$(BIGIP_PASSWORD)",
            "--bigip-url=10.1.10.60",
            "--bigip-partition=kubernetes",
            "--namespace=default",
            "--pool-member-type=nodeport"
            ]
```

8. Once you have your yaml file setup, you can try to launch your deployment. It will start our f5-k8s-controller container on one of our nodes (may take around 30sec to be in a running state):

```
kubectl create -f f5-nodeport-deployment.yaml
```

9. Verify the deployment "deployed"

```
kubectl get deployment k8s-bigip-ctlr-deployment --namespace kube-system
```

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl get deployment k8s-bigip-ctlr-deployment --namespace kube-system
NAME                        DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
k8s-bigip-ctlr-deployment   1         1         1            1           21s
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

10. To locate on which node the container connector is running, you can use the following command:

```
kubectl get pods -o wide -n kube-system
```

We can see that our container is running on kube-node2 below.

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl get pods -o wide -n kube-system
NAME                                        READY     STATUS    RESTARTS   AGE       IP            NODE
TED NODE
coredns-78fcdf6894-6k56m                    1/1       Running   0          27m       10.244.0.4    kube-master1
coredns-78fcdf6894-cvk8h                    1/1       Running   0          27m       10.244.0.5    kube-master1
etcd-kube-master1                           1/1       Running   0          27m       10.1.10.21    kube-master1
k8s-bigip-ctlr-deployment-5b74dd769-x55vx   1/1       Running   0          1m        10.244.1.4    kube-node1
kube-apiserver-kube-master1                 1/1       Running   0          27m       10.1.10.21    kube-master1
kube-controller-manager-kube-master1        1/1       Running   0          27m       10.1.10.21    kube-master1
kube-flannel-ds-amd64-54d4q                 1/1       Running   0          24m       10.1.10.21    kube-master1
kube-flannel-ds-amd64-64vp7                 1/1       Running   0          20m       10.1.10.23    kube-node2
kube-flannel-ds-amd64-8hfnz                 1/1       Running   0          20m       10.1.10.22    kube-node1
kube-proxy-57pxx                            1/1       Running   0          20m       10.1.10.23    kube-node2
kube-proxy-c79zn                            1/1       Running   0          27m       10.1.10.21    kube-master1
kube-proxy-ksc6x                            1/1       Running   0          20m       10.1.10.22    kube-node1
kube-scheduler-kube-master1                 1/1       Running   0          26m       10.1.10.21    kube-master1
kubernetes-dashboard-6d4bc79449-b6756       1/1       Running   0          11m       10.244.2.4    kube-node2
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

Troubleshooting

If you need to troubleshoot your container, you have two different ways to check the logs of your container, kubectl command or docker command.

1. Using kubectl command: you need to use the full name of your pod as showed in the previous image

   ```
   # For example:
   kubectl logs k8s-bigip-ctlr-deployment-5b74dd769-x55vx -n kube-system
   ```

   ```
   ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl logs k8s-bigip-ctlr-deployment-5b74dd769-x55vx -n kube-system
   2018/09/26 23:36:11 [INFO] Starting: Version: v1.6.1, BuildInfo: n1190-426970563
   2018/09/26 23:36:11 [INFO] ConfigWriter started: 0xc42026be30
   2018/09/26 23:36:11 [INFO] Started config driver sub-process at pid: 13
   2018/09/26 23:36:11 [INFO] NodePoller (0xc4202775f0) registering new listener: 0x407e20
   2018/09/26 23:36:12 [INFO] NodePoller started: (0xc4202775f0)
   2018/09/26 23:36:12 [INFO] Registered BigIP Metrics
   2018/09/26 23:36:12 [INFO] Wrote 0 Virtual Server and 0 IApp configs
   2018/09/26 23:36:13 [INFO] [2018-09-26 23:36:13,189 __main__ INFO] entering inotify loop to watch /tmp/k8s-bigip-ctl
   nfig.json
   ubuntu@kube-master1:~/agilitydocs/kubernetes$
   ```

2. Using docker logs command: From the previous check we know the container is running on kube-node1. Via mRemoteNG open a session to kube-node1 and run the following commands:

   ```
   sudo docker ps
   ```

   Here we can see our container ID is "01a7517b50c5"

   ```
   ubuntu@kube-node1:~$ sudo docker ps
   CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
   01a7517b50c5        9da1258ca9a1        "/app/bin/k8s-bigip-…"   6 minutes ago       Up 6 minutes
   ```

   Now we can check our container logs:

   ```
   sudo docker logs 01a7517b50c5
   ```

   ```
   ubuntu@kube-node1:~$ sudo docker logs 01a7517b50c5
   2018/09/26 23:36:11 [INFO] Starting: Version: v1.6.1, BuildInfo: n1190-426970563
   2018/09/26 23:36:11 [INFO] ConfigWriter started: 0xc42026be30
   2018/09/26 23:36:11 [INFO] Started config driver sub-process at pid: 13
   2018/09/26 23:36:11 [INFO] NodePoller (0xc4202775f0) registering new listener: 0x407e20
   2018/09/26 23:36:12 [INFO] NodePoller started: (0xc4202775f0)
   2018/09/26 23:36:12 [INFO] Registered BigIP Metrics
   2018/09/26 23:36:12 [INFO] Wrote 0 Virtual Server and 0 IApp configs
   2018/09/26 23:36:13 [INFO] [2018-09-26 23:36:13,189 __main__ INFO] entering inotify loop to watch /tmp/k8s-bigip-ctl
   nfig.json
   ubuntu@kube-node1:~$
   ```

   Note: The log messages here are identical to the log messages displayed in the previous kubectl logs command.

3. You can connect to your container with kubectl as well:

```
kubectl exec -it k8s-bigip-ctlr-deployment-79fcf97bcc-48qs7 -n kube-system  -- /
↪bin/sh

cd /app

ls -la

exit
```

Lab 3.2 - F5 Container Connector Usage

Now that our container connector is up and running, let's deploy an application and leverage our F5 CC.

For this lab we'll use a simple pre-configured docker image called "f5-hello-world". It can be found on docker hub at f5devcentral/f5-hello-world

To deploy our application, we will need to do the following:

1. Define a Deployment: this will launch our application running in a container.
2. Define a ConfigMap: this can be used to store fine-grained information like individual properties or coarse-grained information like entire config files or JSON blobs. It will contain the BIG-IP configuration we need to push.
3. Define a Service: this is an abstraction which defines a logical set of *pods* and a policy by which to access them. Expose the *service* on a port on each node of the cluster (the same port on each *node*). You'll be able to contact the service on any <NodeIP>:NodePort address. If you set the type field to "NodePort", the Kubernetes master will allocate a port from a flag-configured range **(default: 30000-32767)**, and each Node will proxy that port (the same port number on every Node) into your *Service*.

App Deployment

On **kube-master1** we will create all the required files:

1. Create a file called `f5-hello-world-deployment.yaml`

Tip: Use the file in /home/ubuntu/agilitydocs/kubernetes

```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: f5-hello-world
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: f5-hello-world
    spec:
      containers:
      - name: f5-hello-world
        image: "f5devcentral/f5-hello-world:latest"
```

```
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
          protocol: TCP
```

2. Create a file called `f5-hello-world-configmap.yaml`

Tip: Use the file in /home/ubuntu/agilitydocs/kubernetes

Attention: The schema version below (for example 1.7) comes from the releases of big-ip-controller. For more information, head over to the following link for a quick review: https://clouddocs.f5.com/containers/v2/releases_and_versioning.html#schema-table

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: f5-hello-world
  namespace: default
  labels:
    f5type: virtual-server
data:
  schema: "f5schemadb://bigip-virtual-server_v0.1.7.json"
  data: |
    {
      "virtualServer": {
        "frontend": {
          "balance": "round-robin",
          "mode": "http",
          "partition": "kubernetes",
          "virtualAddress": {
            "bindAddr": "10.1.10.81",
            "port": 80
          }
        },
        "backend": {
          "serviceName": "f5-hello-world",
          "servicePort": 8080,
          "healthMonitors": [{
            "interval": 5,
            "protocol": "http",
            "send": "HEAD / HTTP/1.0\r\n\r\n",
            "timeout": 16
          }]
        }
      }
    }
```

3. Create a file called `f5-hello-world-service.yaml`

Tip: Use the file in /home/ubuntu/agilitydocs/kubernetes

```
apiVersion: v1
kind: Service
metadata:
  name: f5-hello-world
  labels:
    run: f5-hello-world
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  type: NodePort
  selector:
    run: f5-hello-world
```

4. We can now launch our application:

```
kubectl create -f f5-hello-world-deployment.yaml
kubectl create -f f5-hello-world-configmap.yaml
kubectl create -f f5-hello-world-service.yaml
```

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl create -f f5-hello-world-deployment.yaml
deployment.extensions/f5-hello-world created
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl create -f f5-hello-world-configmap.yaml
configmap/f5-hello-world created
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl create -f f5-hello-world-service.yaml
service/f5-hello-world created
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

5. To check the status of our deployment, you can run the following commands:

```
kubectl get pods -o wide

# This can take a few seconds to a minute to create these hello-world containers to running state.
```

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl get pods -o wide
NAME                             READY     STATUS    RESTARTS   AGE       IP            NODE
f5-hello-world-6c8cc75ddf-2fp2d  1/1       Running   0          2m        10.244.1.5    kube-node1
f5-hello-world-6c8cc75ddf-kwnx5  1/1       Running   0          2m        10.244.2.5    kube-node2
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

```
kubectl describe svc f5-hello-world
```

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl describe svc f5-hello-world
Name:                     f5-hello-world
Namespace:                default
Labels:                   run=f5-hello-world
Annotations:              <none>
Selector:                 run=f5-hello-world
Type:                     NodePort
IP:                       10.96.224.25
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  30778/TCP
Endpoints:                10.244.1.5:8080,10.244.2.5:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

6. To test the app you need to pay attention to:

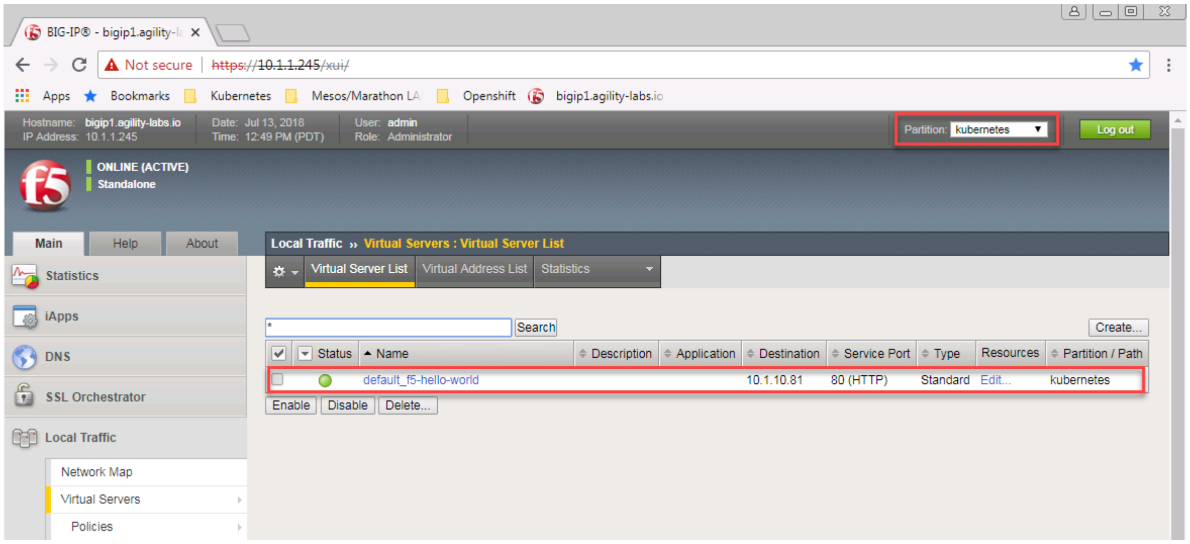
The NodePort value, that's the port used by Kubernetes to give you access to the app from the outside. Here it's "30507", highlighted above.

The Endpoints, that's our 2 instances (defined as replicas in our deployment file) and the port assigned to the service: port 8080.

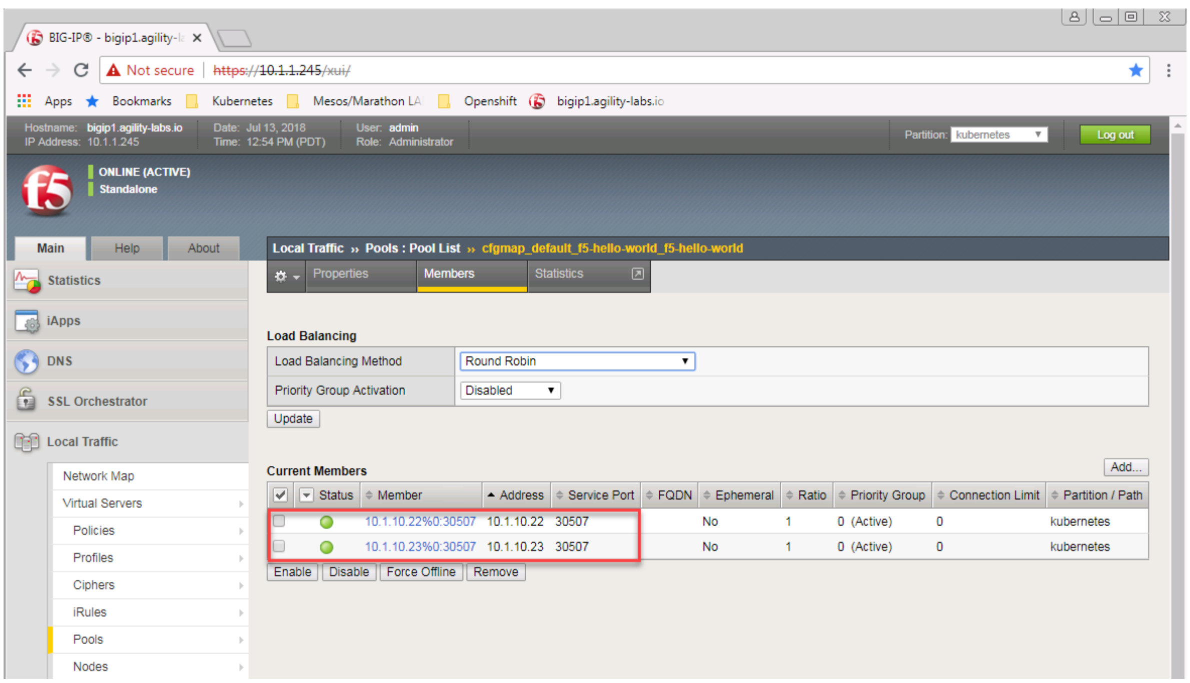
Now that we have deployed our application sucessfully, we can check our BIG-IP configuration. From the browser open https://10.1.1.245

> **Warning:** Don't forget to select the "kubernetes" partition or you'll see nothing.

Here you can see a new Virtual Server, "default_f5-hello-world" was created, listening on 10.1.10.81.

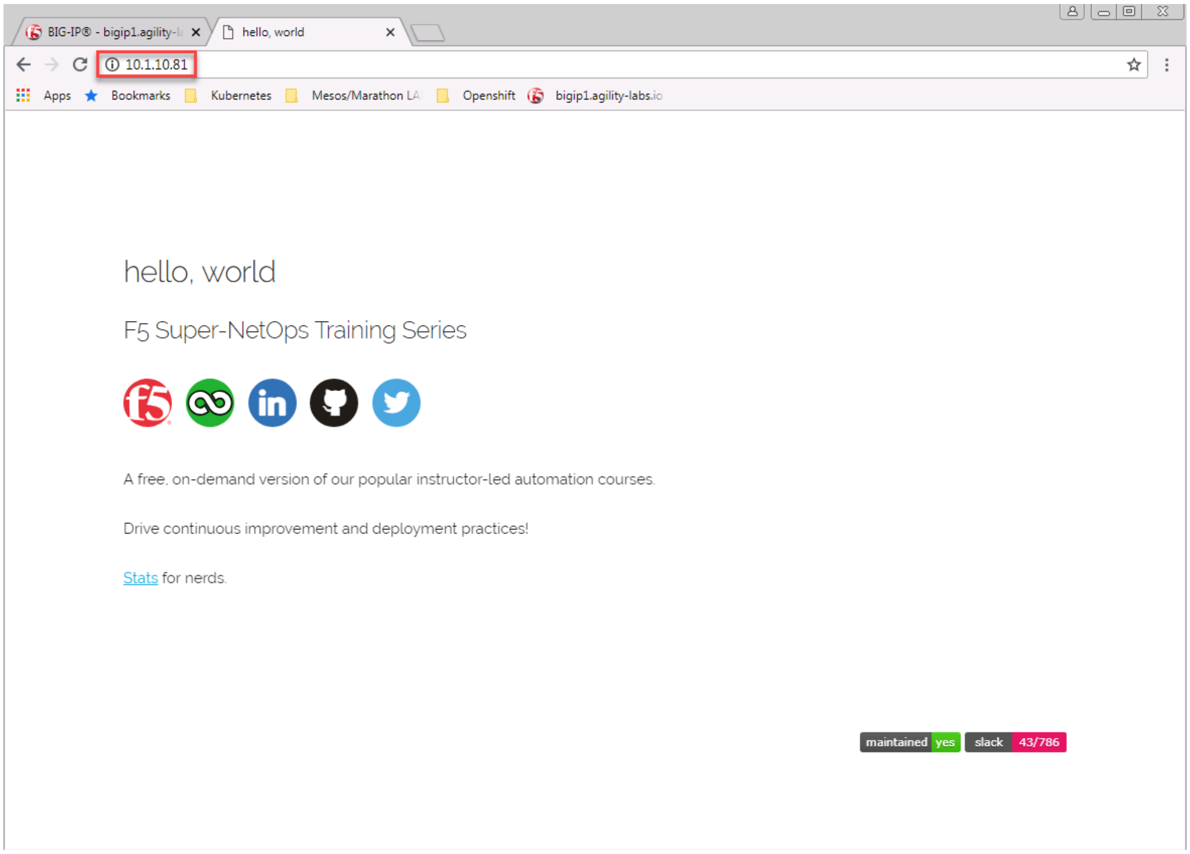


Check the Pools to see a new pool and the associated pool members: Local Traffic –> Pools –> "cfgmap_default_f5-hello-world_f5-hello-world" –> Members



Note: You can see that the pool members listed are all the kubernetes nodes. (**NodePort mode**)

7. Now you can try to access your application via your BIG-IP VIP: 10.1.10.81



8. Hit Refresh many times and go back to your **BIG-IP** UI, go to Local Traffic –> Pools –> Pool list –> cfgmap_default_f5-hello-world_f5-hello-world –> Statistics to see that traffic is distributed as expected.

/kubernetes/cfgmap_default_f5-hello-world_f5 Search Reset Search

	Status		Pool	Pool Member	Partition / Path	Bits In	Bits Out	Packets In	Packets Out	Connections Current	Connections Maximum	Connections Total	Requests Total	Request Queue Depth	Request Queue Maximum Age
			cfgmap_default_f5-hello-world_f5-hello-world		kubernetes	82.6K	1.4M	91	75	0	4	4	17	0	0
				10.1.10.22%0:30507	kubernetes	10.9K	24.1K	8	6	0	1	1	2	0	0
				10.1.10.23%0:30507	kubernetes	71.7K	1.4M	83	69	0	3	3	15	0	0

9. How is traffic forwarded in Kubernetes from the <node IP>:30507 to the <container IP>:8080? This is done via iptables that is managed via the kube-proxy instances. On either of the nodes, SSH in and run the following command:

```
sudo iptables-save | grep f5-hello-world
```

This will list the different iptables rules that were created regarding our service.

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ sudo iptables-save | grep f5-hello-world
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/f5-hello-world:" -m tcp --dport 30778 -j KUBE-MARK-MASQ
-A KUBE-NODEPORTS -p tcp -m comment --comment "default/f5-hello-world:" -m tcp --dport 30778 -j KUBE-SVC-UHN6QQRK2R6JGUDX
-A KUBE-SEP-74I6YQHNDEKUPAEL -s 10.244.2.5/32 -m comment --comment "default/f5-hello-world:" -j KUBE-MARK-MASQ
-A KUBE-SEP-74I6YQHNDEKUPAEL -p tcp -m comment --comment "default/f5-hello-world:" -m tcp -j DNAT --to-destination 10.244.2.
5:8080
-A KUBE-SEP-EQDA44RHUIVZM5KA -s 10.244.1.5/32 -m comment --comment "default/f5-hello-world:" -j KUBE-MARK-MASQ
-A KUBE-SEP-EQDA44RHUIVZM5KA -p tcp -m comment --comment "default/f5-hello-world:" -m tcp -j DNAT --to-destination 10.244.1.
5:8080
-A KUBE-SERVICES ! -s 10.244.0.0/16 -d 10.96.224.25/32 -p tcp -m comment --comment "default/f5-hello-world: cluster IP" -m t
cp --dport 8080 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.96.224.25/32 -p tcp -m comment --comment "default/f5-hello-world: cluster IP" -m tcp --dport 8080 -j
KUBE-SVC-UHN6QQRK2R6JGUDX
-A KUBE-SVC-UHN6QQRK2R6JGUDX -m comment --comment "default/f5-hello-world:" -m statistic --mode random --probability 0.50000
000000 -j KUBE-SEP-EQDA44RHUIVZM5KA
-A KUBE-SVC-UHN6QQRK2R6JGUDX -m comment --comment "default/f5-hello-world:" -j KUBE-SEP-74I6YQHNDEKUPAEL
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

10. Scale the f5-hello-world app

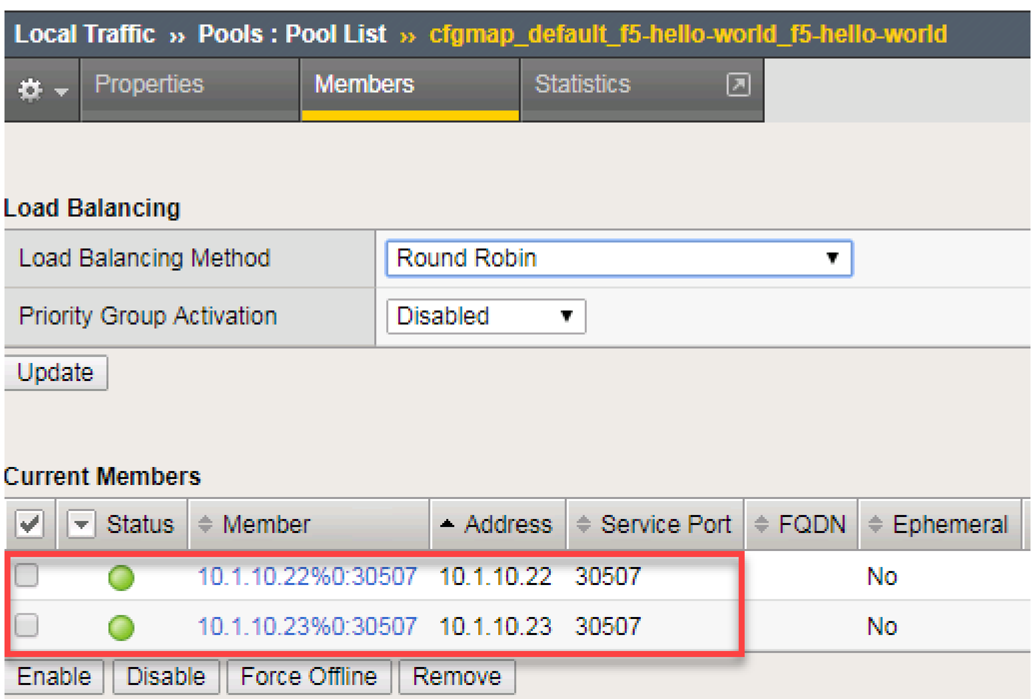
```
kubectl scale --replicas=10 deployment/f5-hello-world -n default
```

11. Check that the pods were created

```
kubectl get pods
```

```
ubuntu@kube-master1:~/agilitydocs/kubernetes$ kubectl get pods
NAME                              READY     STATUS    RESTARTS   AGE
f5-hello-world-6c8cc75ddf-2fp2d   1/1       Running   0          6m
f5-hello-world-6c8cc75ddf-5d8nv   1/1       Running   0          18s
f5-hello-world-6c8cc75ddf-72nsh   1/1       Running   0          18s
f5-hello-world-6c8cc75ddf-7g79r   1/1       Running   0          18s
f5-hello-world-6c8cc75ddf-k8fhs   1/1       Running   0          18s
f5-hello-world-6c8cc75ddf-kwnx5   1/1       Running   0          6m
f5-hello-world-6c8cc75ddf-l2mzt   1/1       Running   0          18s
f5-hello-world-6c8cc75ddf-mf95w   1/1       Running   0          18s
f5-hello-world-6c8cc75ddf-s22b4   1/1       Running   0          18s
f5-hello-world-6c8cc75ddf-wphsk   1/1       Running   0          18s
ubuntu@kube-master1:~/agilitydocs/kubernetes$
```

12. Check the pool was updated on big-ip



Attention: Why are there only 2 pool members?

Expected time to complete: **1 hours**

3.4 Lab Setup

We will leverage the following setup to configure the Kubernetes environment.

Hostname	IP-ADDR	Credentials
jumpbox	10.1.1.250	user/Student!Agility!
bigip1	10.1.1.245 10.1.10.60	admin/admin root/default
kube-master1	10.1.10.21	ubuntu/ubuntu root/default
kube-node1	10.1.10.22	ubuntu/ubuntu root/default
kube-node2	10.1.10.23	ubuntu/ubuntu root/default

4 Class 3: Introduction to Mesos / Marathon

This introductory class covers the following topics:

4.1 Module 1: Introduction to Mesos / Marathon

The purpose of this module is to give you a basic understanding of Mesos / Marathon concepts and components

4.1.1 Mesos / Marathon Overview

The F5 Marathon Container Integration consists of the F5 Marathon BIG-IP Controller.

The F5 Marathon BIG-IP Controller configures a BIG-IP to expose applications in a Mesos cluster as BIG-IP virtual servers, serving North-South traffic.

See also:

The official F5 documentation is available here: F5 Marathon Container Integration

You can either setup the whole F5 solutions yourself or use some scripts to automatically deploy everything.

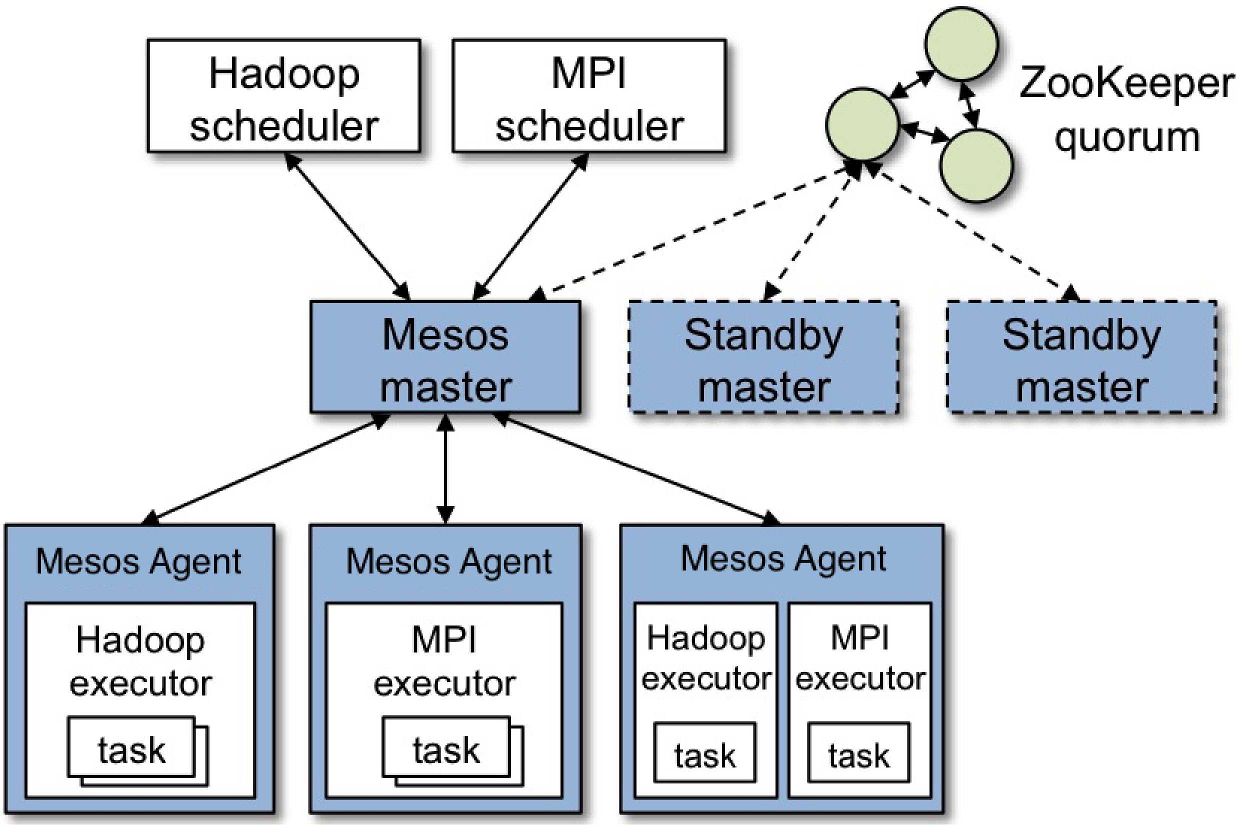
We also provide some ansible playbooks if you need to setup a Mesos/Marathon env.

Before working on the installation itself, you need to understand the different components involved in this setup:

- Master / Agent functions
- The different components involved in the Master / Agent architecture
- How High availability is achieved
- Marathon overview

Mesos Architecture

This is an extract from Mesos Architecture

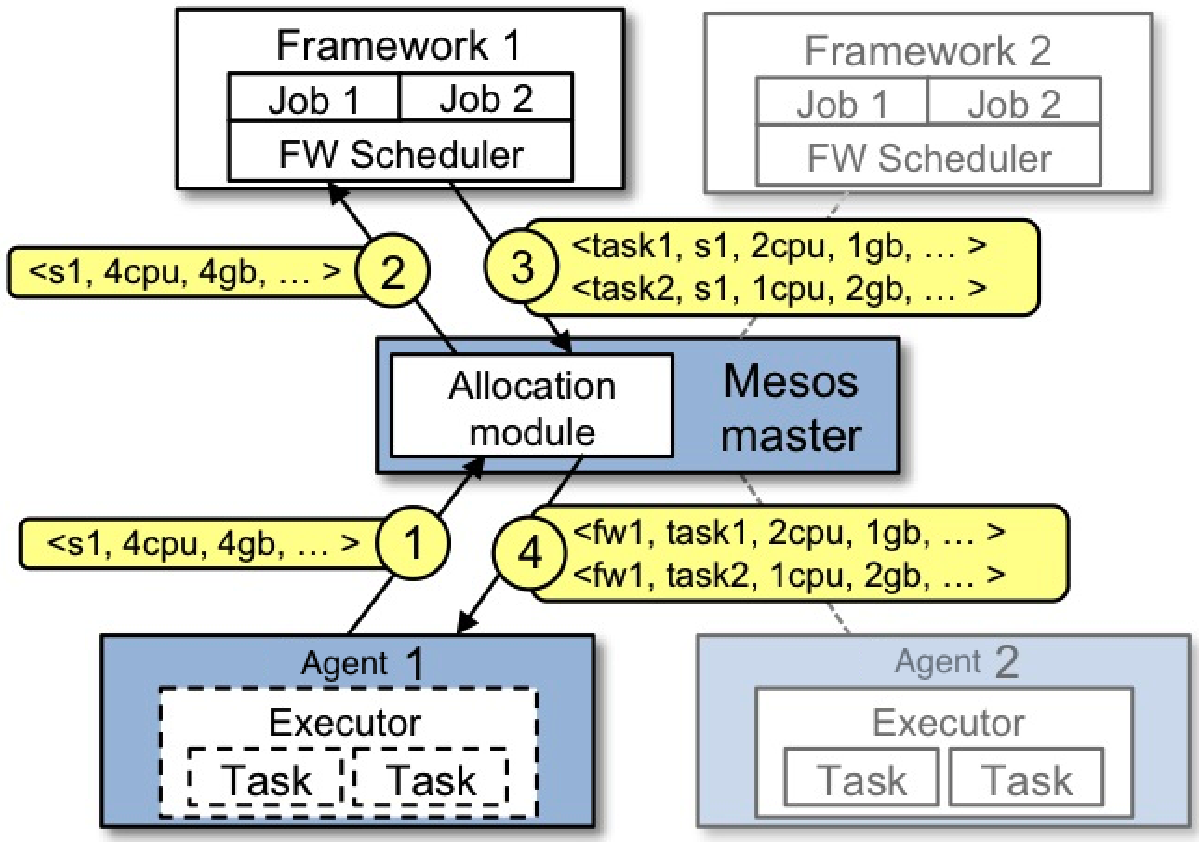


Some of the involved components:

- Master: aggregates resource offers from all agent nodes and provides them to registered frameworks.
- Agent: runs a discrete Mesos task on behalf of a framework. It is an agent instance registered with the Mesos master. The synonym of agent node is worker or slave node. You can have private or public agent nodes. Agent daemon can run on the same component than the master daemon. This is useful when you need a small environment for testing
- Framework: "Applications" running on Mesos. It is composed of a scheduler, which registers with the master to receive resource offers, and one or more executors, which launches tasks on slaves. Examples of Mesos frameworks include Marathon, Chronos and Hadoop
- Offer: a list of a agent's available CPU and memory resources. All agents send offers to the master, and the master provides offers to registered frameworks
- Executors: launched on agent nodes to run tasks for a service.
- Task: a unit of work that is scheduled by a framework, and is executed on an agent node. A task can be anything from a bash command or script, to an SQL query, to a Hadoop job, a docker image
- Apache ZooKeeper: software that is used to coordinate the master nodes and achieve High availability
- Service discovery: When your app is up and running, you need a way to send traffic to it, from other applications on the same cluster, and from external clients.

Example of resource offer

This is an extract from Apache Mesos website Mesos Architecture



Let's walk through the events in the figure.

1. Agent 1 reports to the master that it has 4 CPUs and 4 GB of memory free. The master then invokes the allocation policy module, which tells it that framework 1 should be offered all available resources.
2. The master sends a resource offer describing what is available on agent 1 to framework 1.
3. The framework's scheduler replies to the master with information about two tasks to run on the agent, using <2 CPUs, 1 GB RAM> for the first task, and <1 CPUs, 2 GB RAM> for the second task.
4. Finally, the master sends the tasks to the agent, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, the allocation module may now offer them to framework 2.

In addition, this resource offer process repeats when tasks finish and new resources become free.

Service Discovery

One way to enable service discovery is to leverage Mesos DNS. Mesos DNS provides service discovery through domain name system (DNS).

Mesos-DNS periodically queries the Mesos master(s), retrieves the state of all running tasks from all running frameworks, and generates DNS records for these tasks (A and SRV records). As tasks start, finish, fail, or restart on the Mesos cluster, Mesos-DNS updates the DNS records to reflect the latest state.

Running tasks can be discovered by looking up A and, optionally, SRV records within the Mesos domain.

- An A record associates a hostname to an IP address

- An SRV record associates a service name to a hostname and an IP port

High Availability

Marathon supports high availability be leveraging Zookeeper. High availability allows applications to keep running if an instance becomes unavailable. This is accomplished by running several Marathon instances that point to the same ZooKeeper quorum. ZooKeeper is used to perform leader election in the event that the currently leading Marathon instance fails.

If you want to learn more about Zookeeper, refer to their website Zookeeper

With Zookeeper, it is recommended to have an odd number of servers.

Marathon

Marathon is a production-proven Apache Mesos framework for container orchestration. the github project can be found here: Github Marathon , documentation is here

Marathon is a framework for Mesos that is designed to launch long-running applications, and, in Mesosphere, serves as a replacement for a traditional *init* system. It has many features that simplify running applications in a clustered environment, such as high-availability, application health checks, . . . It adds its scaling and self-healing capabilities to the Mesosphere feature set.

Marathon can be used to start other Mesos frameworks, and it can also launch any process that can be started in the regular shell. As it is designed fo long-running applications, it will ensure that applications it has launched will continue running, even if the slave node(s) they are running on fails.

Main features

1. High Availability. Marathon runs as an active/passive cluster with leader election for 100% uptime.
2. Multiple container runtimes. Marathon has first-class support for both Mesos containers (using cgroups) and Docker.
3. Stateful apps. Marathon can bind persistent storage volumes to your application. You can run databases like MySQL and Postgres, and have storage accounted for by Mesos.
4. UI.
5. Constraints. e.g. Only one instance of an application per rack, node, etc.
6. Service Discovery & Load Balancing. Several methods available.
7. Health Checks. Evaluate your application's health using HTTP or TCP checks.
8. Event Subscription. Supply an HTTP endpoint to receive notifications - for example to integrate with an external load balancer.
9. Metrics. Query them at /metrics in JSON format or push them to systems like graphite, statsd and Datadog.
10. Complete REST API for easy integration and script-ability.

4.2 Module 2: Build a Mesos / Marathon Cluster

Attention: THIS MODULE CAN BE SKIPPED. THE BLUEPRINT IS PRE-CONFIGURED WITH A WORKING CLUSTER. THIS MODULE IS FOR DOCUMENTION PURPOSES ONLY.

In this module, we will build a 3 node cluster (1x masters and 2x nodes) utilizing Ubuntu server images.

As a reminder, in this module, our cluster setup is:

Hostname	IP-ADDR	Role
mesos-master1	10.2.10.21	Master
mesos-agent1	10.2.10.22	Agent
mesos-agent2	10.2.10.23	Agent

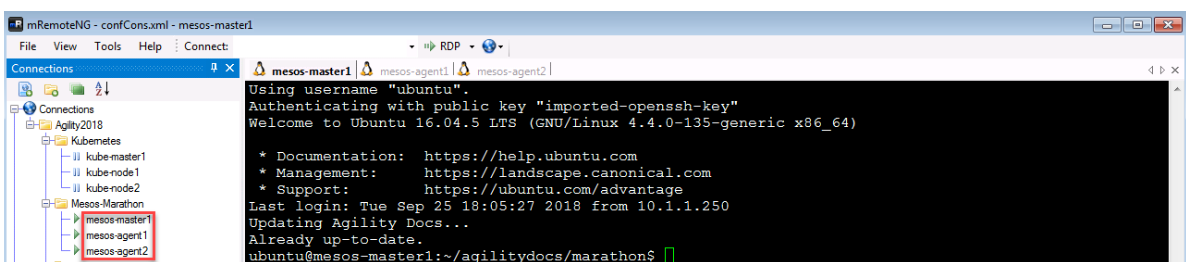
4.2.1 Lab 2.1 - Prep Ubuntu

Note: This installation will utilize Ubuntu v16.04 (Xenial)

Important: The following commands need to be run on all three nodes unless otherwise specified.

1. From the jumpbox open **mRemoteNG** and start a session to each of the following servers. The sessions are pre-configured to connect with the default user "ubuntu".

   - mesos-master1
   - mesos-agent2
   - mesos-agent3

2. Elevate to "root"

   ```
   su -

   #When prompted for password enter "default" without the quotes
   ```

3. For your convenience we've already added the host IP & names to /etc/hosts. Verify the file

   ```
   cat /etc/hosts
   ```

   The file should look like this:

```
root@mesos-master1:~# cat /etc/hosts
127.0.0.1       localhost
127.0.1.1       ubuntu.localdomain      ubuntu

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters

10.2.10.21 mesos-master1
10.2.10.22 mesos-agent1
10.2.10.23 mesos-agent2
root@mesos-master1:~#
```

If entries are not there add them to the bottom of the file be editing "/etc/hosts" with 'vim'

```
vim /etc/hosts

#cut and paste the following lines to /etc/hosts

10.2.10.21    mesos-master1
10.2.10.22    mesos-agent1
10.2.10.23    mesos-agent2
```

4. Ensure the OS is up to date, run the following command

```
apt update && apt upgrade -y

#This can take a few seconds to several minute depending on demand to download the latest updates for the OS.
```

5. Add the docker repo

```
curl \-fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add \-

add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable"
```

6. Install the docker packages

```
apt update && apt install docker-ce -y
```

7. Verify docker is up and running

```
docker run --rm hello-world
```

If everything is working properly you should see the following message

```
root@mesos-master1:~# docker run --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
d1725b59e92d: Pull complete
Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

root@mesos-master1:~#
```

8. Install java for the mesos and marathon processes.

```
apt install -y openjdk-8-jdk

export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
```

4.2.2 Lab 2.2 - Setup the Master

Important: The following commands need to be run on the **master** only unless otherwise specified.

Install Mesos, Marathon and Zookeeper

1. Add the mesos/marathon repo

   Run the following commands:

```
apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF

cat <<EOF >> /etc/apt/sources.list.d/mesosphere.list
deb http://repos.mesosphere.com/ubuntu $(lsb_release -cs) main
EOF
```

2. Install the mesos, marathon and zookeeper packages

```
apt update && apt install mesos marathon zookeeperd -y
```

Setup Zookeeper

Note: `2181` is zookeeper's default port.

1. Setup a unique ID per zookeeper instance. Update `/etc/zookeeper/conf/myid` to `1`, `2` or `3` depending on the number of master nodes. In our case `1`

```
echo 1 > /etc/zookeeper/conf/myid
```

2. Modify the zookeeper config file on each master

```
sed -i /^#server.1/s/#server.1=zookeeper1/server.1=10.2.10.21/ /etc/zookeeper/
↪conf/zoo.cfg
```

Setup Mesos

1. Create mesos *ip* file /etc/mesos-master/ip

```
echo "10.2.10.21" > /etc/mesos-master/ip
```

2. Create mesos *hostname* file /etc/mesos-master/hostname (specify the IP address of your node)

```
echo "10.2.10.21" > /etc/mesos-master/hostname
```

3. Change the quorum value to reflect our cluster size. It should be set over 50% of the number of master instances. In this case it should be `1` because we have only one master

```
echo 1 > /etc/mesos-master/quorum
```

4. Point zookeeper to the master instance. This is done in the file /etc/mesos/zk

```
echo "zk://10.2.10.21:2181/mesos" > /etc/mesos/zk
```

Setup Marathon

1. First we need to specify the zookeeper masters that marathon will connect to (for information and things like scheduling). We can copy the previous file we setup for mesos:

```
echo "MARATHON_MASTER=`cat /etc/mesos/zk`" > /etc/default/marathon
```

2. We also need to have marathon store its own state in zookeper (since it runs on all three masters):

```
echo "MARATHON_ZK=zk://10.2.10.21:2181/marathon" >> /etc/default/marathon
```

Start your services

1. When you install mesos, the master and slave services are enabled (called mesos-master and mesos-slave). Here, we want our master to focus on this tasks so we need to disable the slave service. Do this on *all the master* nodes:

```
systemctl stop mesos-slave
echo manual > /etc/init/mesos-slave.override
```

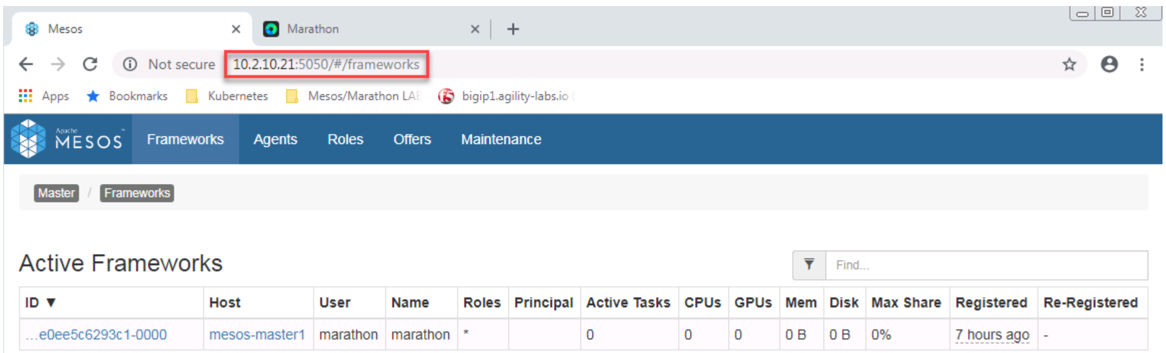
2. We need to restart zookeeper and start mesos-master and marathon process on *all master* nodes:

```
systemctl restart zookeeper

systemctl start mesos-master
systemctl enable mesos-master
```

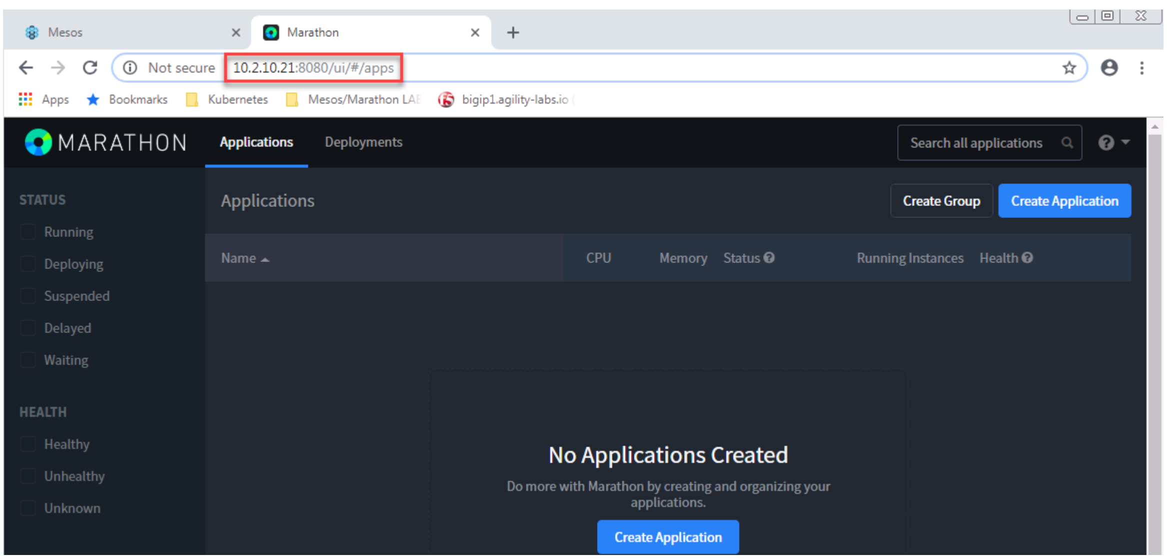
```
systemctl start marathon
```

3. We can validate that it works by connecting to mesos and marathon via a browser. Mesos runs on port 5050 (http) and marathon runs on port 8080 (http).

   Mesos:



   Marathon:

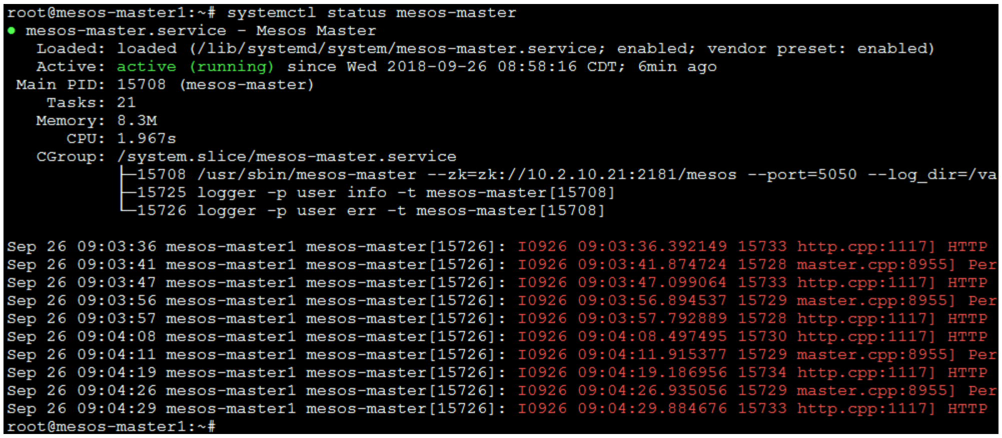


4. If you want to check whether the service started as expected, you can use the following commands:

```
systemctl status mesos-master

systemctl status marathon
```

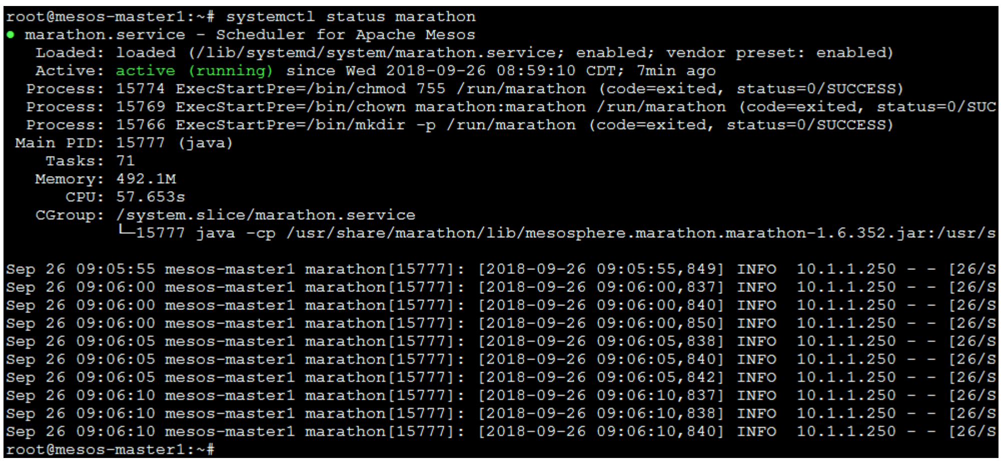
   You should see something like the following:

   Mesos:



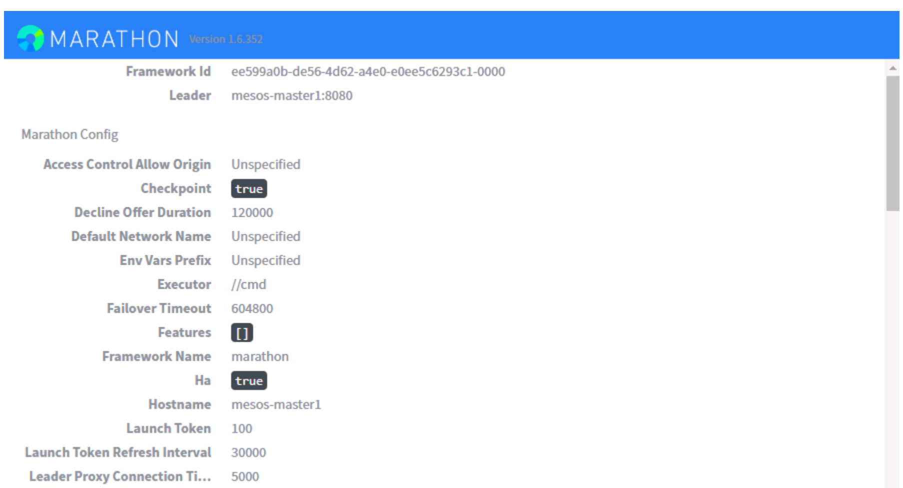
```
root@mesos-master1:~# systemctl status mesos-master
● mesos-master.service - Mesos Master
   Loaded: loaded (/lib/systemd/system/mesos-master.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2018-09-26 08:58:16 CDT; 6min ago
 Main PID: 15708 (mesos-master)
    Tasks: 21
   Memory: 8.3M
      CPU: 1.967s
   CGroup: /system.slice/mesos-master.service
           ├─15708 /usr/sbin/mesos-master --zk=zk://10.2.10.21:2181/mesos --port=5050 --log_dir=/va
           ├─15725 logger -p user info -t mesos-master[15708]
           └─15726 logger -p user err -t mesos-master[15708]

Sep 26 09:03:36 mesos-master1 mesos-master[15726]: I0926 09:03:36.392149 15733 http.cpp:1117] HTTP
Sep 26 09:03:41 mesos-master1 mesos-master[15726]: I0926 09:03:41.874724 15728 master.cpp:8955] Per
Sep 26 09:03:47 mesos-master1 mesos-master[15726]: I0926 09:03:47.099064 15733 http.cpp:1117] HTTP
Sep 26 09:03:56 mesos-master1 mesos-master[15726]: I0926 09:03:56.894537 15729 master.cpp:8955] Per
Sep 26 09:03:57 mesos-master1 mesos-master[15726]: I0926 09:03:57.792889 15728 http.cpp:1117] HTTP
Sep 26 09:04:08 mesos-master1 mesos-master[15726]: I0926 09:04:08.497495 15730 http.cpp:1117] HTTP
Sep 26 09:04:11 mesos-master1 mesos-master[15726]: I0926 09:04:11.915377 15729 master.cpp:8955] Per
Sep 26 09:04:19 mesos-master1 mesos-master[15726]: I0926 09:04:19.186956 15734 http.cpp:1117] HTTP
Sep 26 09:04:26 mesos-master1 mesos-master[15726]: I0926 09:04:26.935056 15729 master.cpp:8955] Per
Sep 26 09:04:29 mesos-master1 mesos-master[15726]: I0926 09:04:29.884676 15733 http.cpp:1117] HTTP
root@mesos-master1:~#
```

Marathon:



```
root@mesos-master1:~# systemctl status marathon
● marathon.service - Scheduler for Apache Mesos
   Loaded: loaded (/lib/systemd/system/marathon.service; enabled; vendor preset: enabled)
   Active: active (running) since Wed 2018-09-26 08:59:10 CDT; 7min ago
  Process: 15774 ExecStartPre=/bin/chmod 755 /run/marathon (code=exited, status=0/SUCCESS)
  Process: 15769 ExecStartPre=/bin/chown marathon:marathon /run/marathon (code=exited, status=0/SUC
  Process: 15766 ExecStartPre=/bin/mkdir -p /run/marathon (code=exited, status=0/SUCCESS)
 Main PID: 15777 (java)
    Tasks: 71
   Memory: 492.1M
      CPU: 57.653s
   CGroup: /system.slice/marathon.service
           └─15777 java -cp /usr/share/marathon/lib/mesosphere.marathon.marathon-1.6.352.jar:/usr/s

Sep 26 09:05:55 mesos-master1 marathon[15777]: [2018-09-26 09:05:55,849] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:00 mesos-master1 marathon[15777]: [2018-09-26 09:06:00,837] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:00 mesos-master1 marathon[15777]: [2018-09-26 09:06:00,840] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:00 mesos-master1 marathon[15777]: [2018-09-26 09:06:00,850] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:05 mesos-master1 marathon[15777]: [2018-09-26 09:06:05,838] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:05 mesos-master1 marathon[15777]: [2018-09-26 09:06:05,840] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:05 mesos-master1 marathon[15777]: [2018-09-26 09:06:05,842] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:10 mesos-master1 marathon[15777]: [2018-09-26 09:06:10,837] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:10 mesos-master1 marathon[15777]: [2018-09-26 09:06:10,838] INFO  10.1.1.250 - - [26/S
Sep 26 09:06:10 mesos-master1 marathon[15777]: [2018-09-26 09:06:10,840] INFO  10.1.1.250 - - [26/S
root@mesos-master1:~#
```

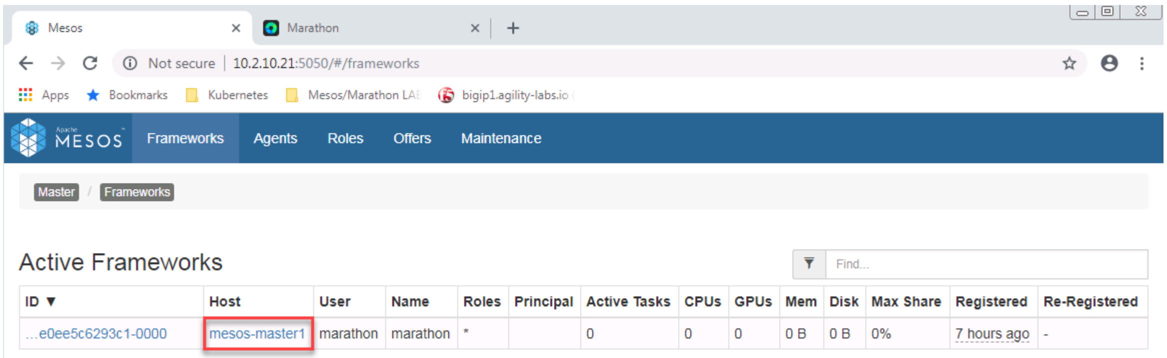
5. For more information about the marathon service, check the *about* section in marathon by clicking the `?` drop down in the upper right hand side of the marathon page.



6. If multiple `masters` were configured for high availability you can do the following to test the HA of marathon:

Attention: For our lab we have only one master so this step is for documentation purposes.

- Figure out which mesos is running the framework marathon (based on our screenshot above, it is available on master1)
- Restart this master and you should see the framework was restarted automatically on another host. “mesos-master1” would change to “mesos-master2, 3, etc.”



4.2.3 Lab 2.3 - Setup the Agents

Once the master is setup and running, we need to setup and join our **agents** to the cluster.

Important: The following commands need to be run on both **agent** nodes unless otherwise specified.

Install Mesos

1. Add the mesos/marathon repo

   Run the following commands:

```
apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF

cat <<EOF >> /etc/apt/sources.list.d/mesosphere.list
deb http://repos.mesosphere.com/ubuntu $(lsb_release -cs) main
EOF
```

2. Install the mesos packages

```
apt update && apt-get install mesos -y
```

Setup Mesos

1. Create mesos *ip* file /etc/mesos-slave/ip
2. Create mesos *hostname* file /etc/mesos-slave/hostname (specify the IP address of your node)
3. Point zookeeper to the master instance. This is done in the file /etc/mesos/zk

```
# On agent1
echo "10.2.10.22" > /etc/mesos-slave/ip
echo "10.2.10.22" > /etc/mesos-slave/hostname
echo "zk://10.2.10.21:2181/mesos" > /etc/mesos/zk

# On agent2
echo "10.2.10.23" > /etc/mesos-slave/ip
echo "10.2.10.23" > /etc/mesos-slave/hostname
echo "zk://10.2.10.21:2181/mesos" > /etc/mesos/zk
```

4. Make the following changes to allow "docker" containers with mesos.

```
#Add the ability to use docker containers
echo 'docker,mesos' > /etc/mesos-slave/containerizers

#Increase the timeout to 5 min so that we have enough time to download any needed docker image
echo '5mins' > /etc/mesos-slave/executor_registration_timeout

#Allow users other then "marathon" to create and run jobs on the agents
echo 'false' > /etc/mesos-slave/switch_user
```

Start Services

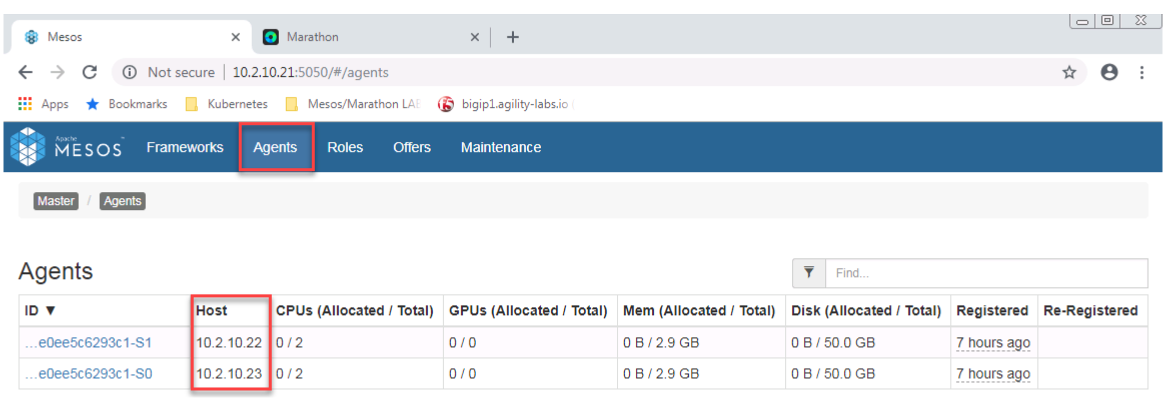
1. First we need to make sure that zookeeper and mesos-master don't run on the agents.

```
systemctl stop zookeeper
echo manual > /etc/init/zookeeper.override

systemctl stop mesos-master
echo manual > /etc/init/mesos.master.override
```

2. Start & enable the agent process called mesos-slave

```
systemctl start mesos-slave
systemctl enable mesos-slave
```

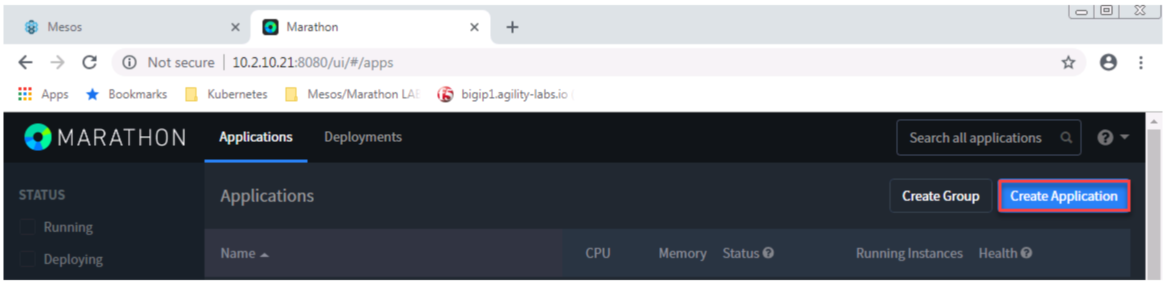
3. Check on master with mesos interface (port 5050) if your agents registered successfully. You should see both agent1 and agent2 on the agent page.



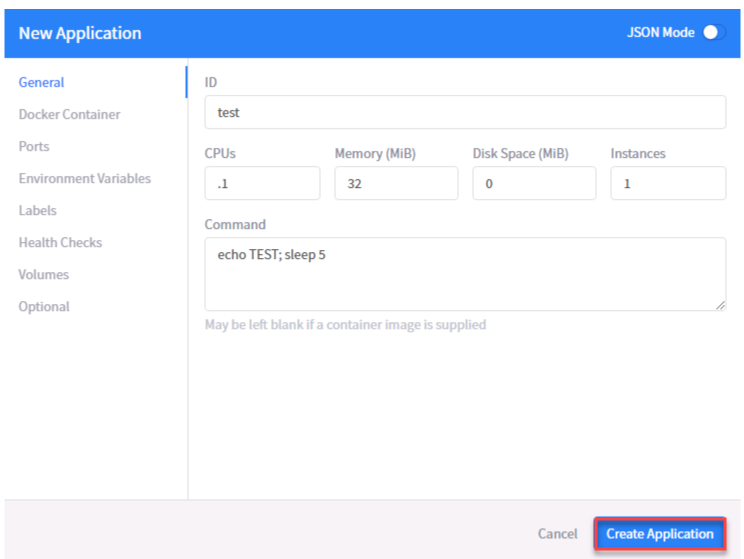
Test Your Setup

Connect to Marathon through one of the master (8080) and launch an application.

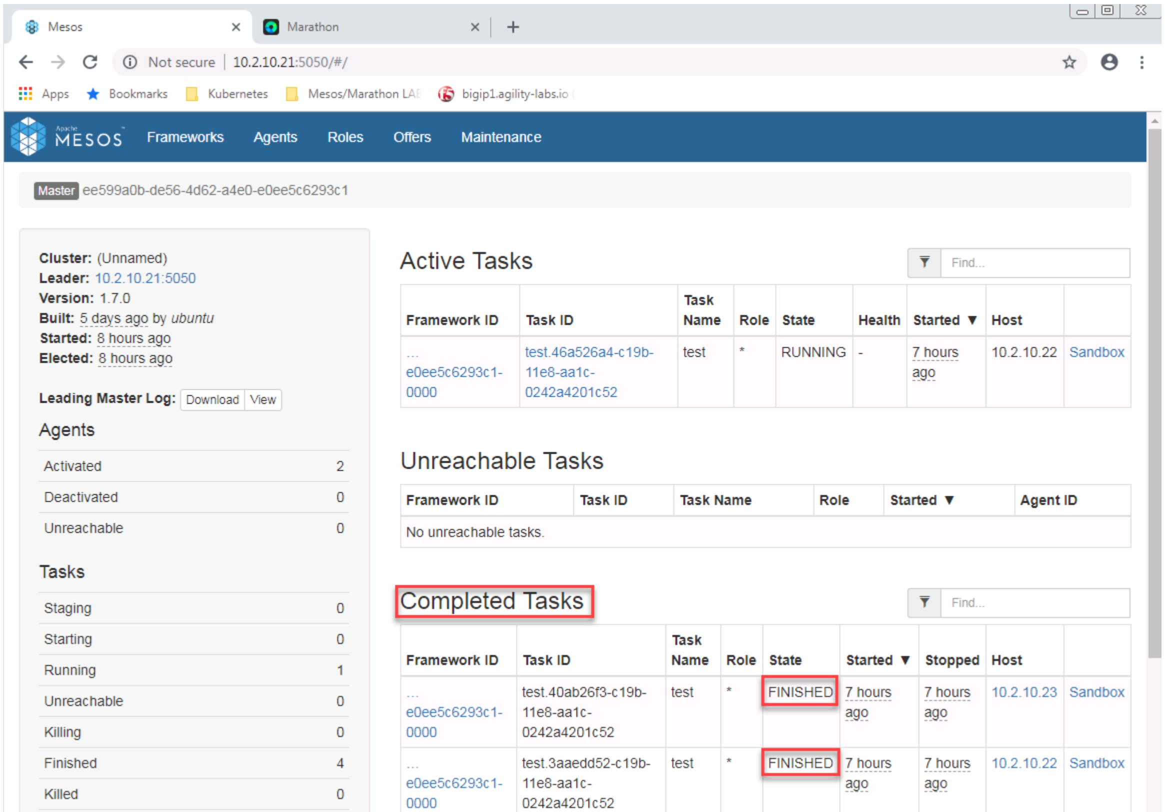
1. Click on *create application*



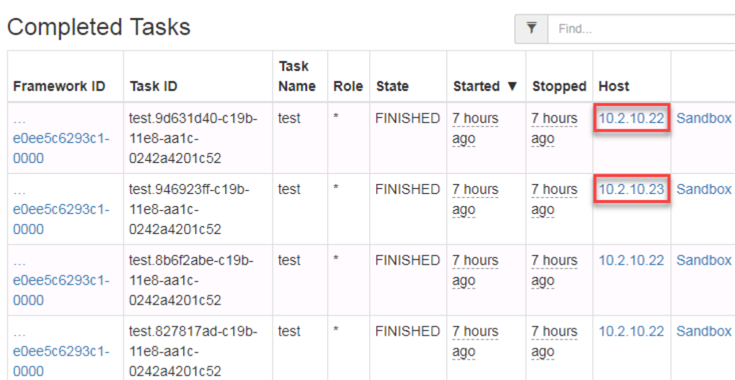
2. Make the following settings and click "Create Application"
   - ID: test
   - CPU: 0.1
   - Memory: 32M
   - Command: echo TEST; sleep 5



3. Once it starts, connect to the mesos framework. Here you should see more and more completed tasks. Name of the task should be "test" (our ID).

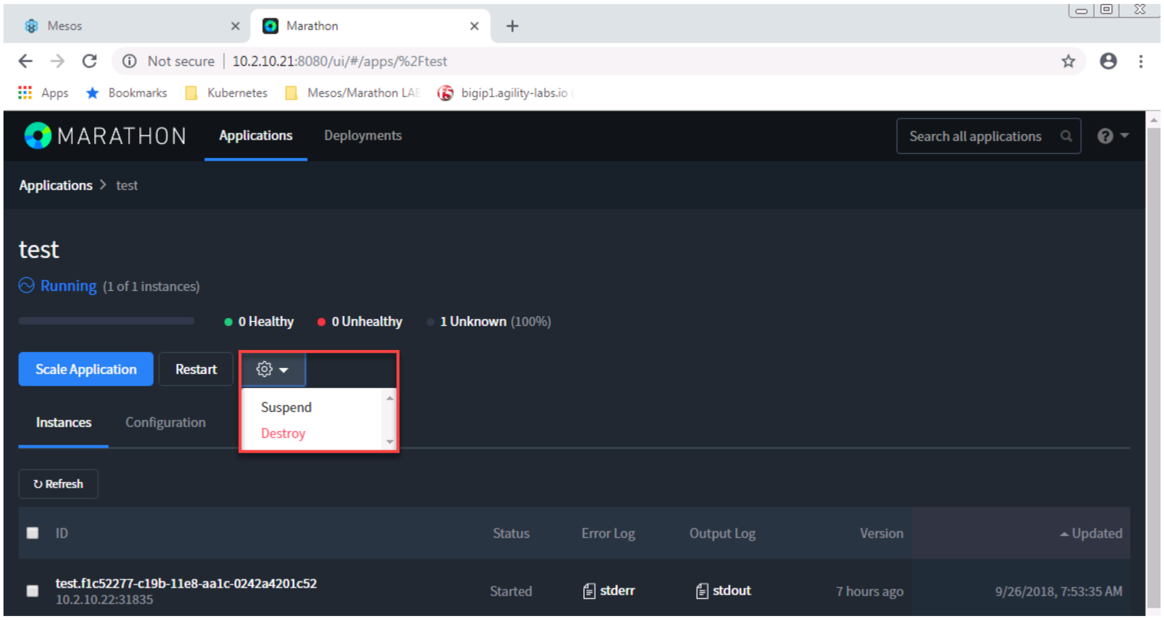


4. If you let it run for a while, you'll see more and more "Completed Tasks". You can see that the Host being selected to run those tasks is not always the same.



Framework ID	Task ID	Task Name	Role	State	Started ▼	Stopped	Host	
... e0ee5c6293c1-0000	test.9d631d40-c19b-11e8-aa1c-0242a4201c52	test	*	FINISHED	7 hours ago	7 hours ago	10.2.10.22	Sandbox
... e0ee5c6293c1-0000	test.946923ff-c19b-11e8-aa1c-0242a4201c52	test	*	FINISHED	7 hours ago	7 hours ago	10.2.10.23	Sandbox
... e0ee5c6293c1-0000	test.8b6f2abe-c19b-11e8-aa1c-0242a4201c52	test	*	FINISHED	7 hours ago	7 hours ago	10.2.10.22	Sandbox
... e0ee5c6293c1-0000	test.827817ad-c19b-11e8-aa1c-0242a4201c52	test	*	FINISHED	7 hours ago	7 hours ago	10.2.10.22	Sandbox

5. Go Back to Marathon, click on our application *test* and click on the setting button and select *destroy* to remove it.



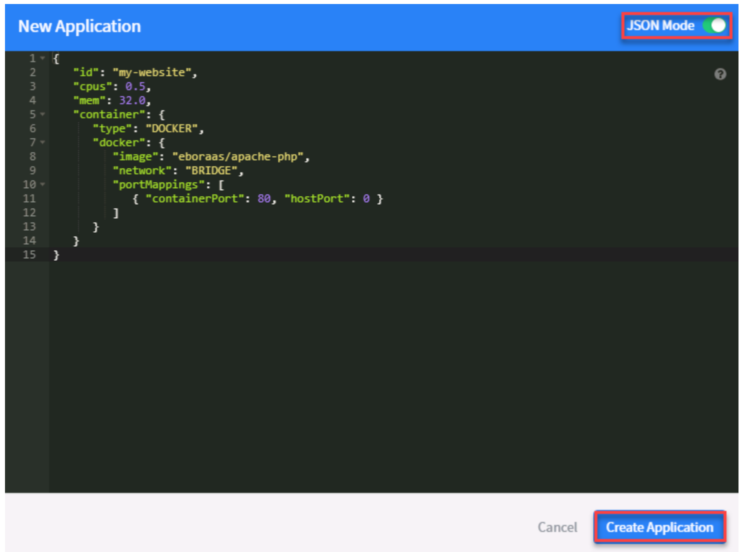
Launch A Container

To test our containers from marathon. We will start a simple apache container.

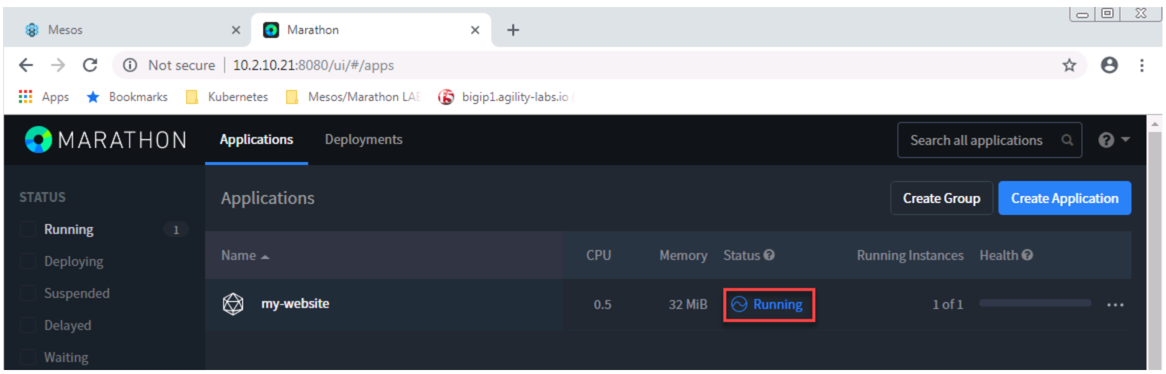
1. Click on create an application, switch to JSON mode and replace the default 8 lines of json with the following and Click "Create Application"

   Note: This may takes some time since we will have to retrieve the image first

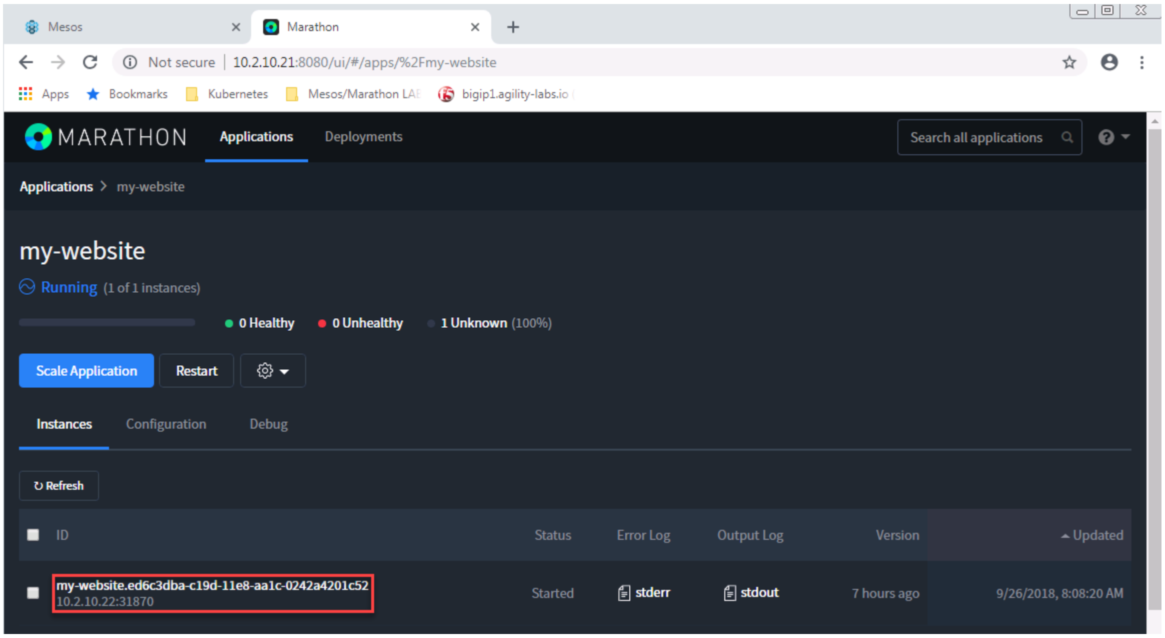
```
{
   "id": "my-website",
   "cpus": 0.5,
   "mem": 32.0,
   "container": {
      "type": "DOCKER",
      "docker": {
         "image": "eboraas/apache-php",
         "network": "BRIDGE",
         "portMappings": [
            { "containerPort": 80, "hostPort": 0 }
         ]
      }
   }
}
```



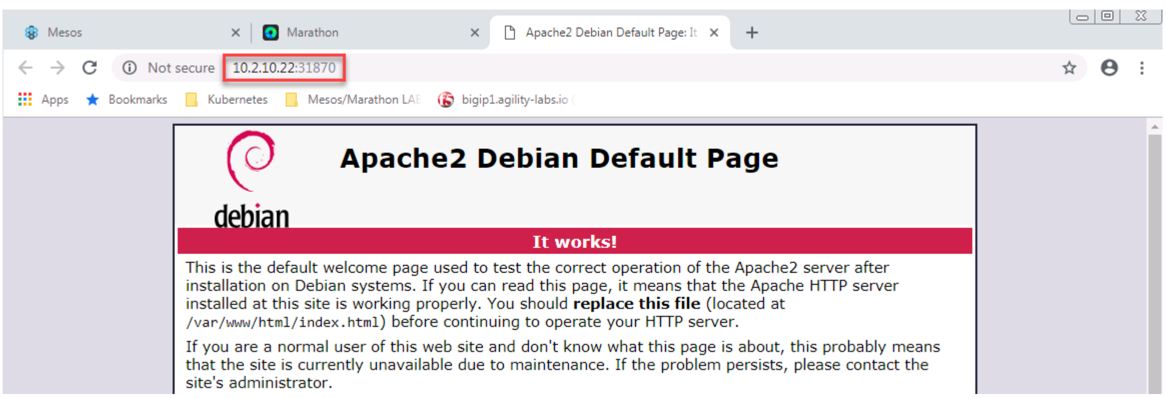
2. It may take some time to switch from `Deploying` to `Running`.



3. Once it's in a `Running` state, find the port used by the container and try to access it at agent IP:port. Click on your application "my-website". Here you'll see the port associated to your instance. In this case it's `31870` and on `agent1 - 10.2.10.22`



4. Use your browser to connect to the application:



4.2.4 Lab 2.4 - Setup Mesos-DNS

If you want to be able to do service discovery with Mesos/Marathon, you need to install and setup mesos-dns.

To leverage marathon for scalability and HA, we will launch mesos-dns as an application from Marathon.

We will setup mesos-dns on **mesos-agent1** (we will force mesos-dns to start on mesos-agent1 in Marathon - 10.2.10.22).

We need to do the following tasks:

- Download the latest mesos-dns binaries
- Configure mesos-dns
- Launch the mesos-dns binary from Marathon

See also:

To retrieve the binary, go to Mesos DNS releases and select the latest version. For this class we'll use the following binary: Mesos DNS release v0.6.0

Download & Configure Mesos-DNS

1. SSH to **mesos-agent1** and do the following:

```
curl -O -L https://github.com/mesosphere/mesos-dns/releases/download/v0.6.0/mesos-
↪dns-v0.6.0-linux-amd64

mkdir /etc/mesos-dns
```

2. Create a file in /etc/mesos-dns/ called config.json and add the json block

```
vim /etc/mesos-dns/config.json
```

```
{
   "zk": "zk://10.2.10.21:2181/mesos",
   "masters": ["10.2.10.21:5050"],
   "refreshSeconds": 60,
   "ttl": 60,
   "domain": "mesos",
   "port": 53,
   "resolvers": ["8.8.8.8"],
   "timeout": 5,
   "httpon": true,
   "dnson": true,
   "httpport": 8123,
   "externalon": true,
   "SOAMname": "ns1.mesos",
   "SOARname": "root.ns1.mesos",
   "SOARefresh": 60,
   "SOARetry": 600,
   "SOAExpire": 86400,
   "SOAMinttl": 60,
   "IPSources": ["mesos", "host"]
}
```

3. Now setup the binary in a proper location:

```
mkdir /usr/local/mesos-dns

mv ./mesos-dns-v0.6.0-linux-amd64 /usr/local/mesos-dns/

chmod +x /usr/local/mesos-dns/mesos-dns-v0.6.0-linux-amd64
```

4. To test your setup do the following:

```
/usr/local/mesos-dns/mesos-dns-v0.6.0-linux-amd64 -config /etc/mesos-dns/config.
↪json -v 10
```

5. This will start your mesos-dns app and you can test it.

```
root@mesos-agent1:~# /usr/local/mesos-dns/mesos-dns-v0.6.0-linux-amd64 -config /etc/mesos-dns/confi
VERY VERBOSE: 2018/09/26 10:35:38 config.go:128: config loaded from "/etc/mesos-dns/config.json"
VERY VERBOSE: 2018/09/26 10:35:38 config.go:205: Mesos-DNS configuration:
VERY VERBOSE: 2018/09/26 10:35:38 config.go:206:    - Masters: 10.2.10.21:5050
VERY VERBOSE: 2018/09/26 10:35:38 config.go:207:    - Zookeeper:  zk://10.2.10.21:2181/mesos
VERY VERBOSE: 2018/09/26 10:35:38 config.go:208:    - ZookeeperDetectionTimeout:  30
VERY VERBOSE: 2018/09/26 10:35:38 config.go:209:    - RefreshSeconds:  60
VERY VERBOSE: 2018/09/26 10:35:38 config.go:210:    - Domain: mesos
VERY VERBOSE: 2018/09/26 10:35:38 config.go:211:    - Listener: 0.0.0.0
VERY VERBOSE: 2018/09/26 10:35:38 config.go:212:    - HTTPListener: 0.0.0.0
VERY VERBOSE: 2018/09/26 10:35:38 config.go:213:    - Port:  53
VERY VERBOSE: 2018/09/26 10:35:38 config.go:214:    - DnsOn:  true
VERY VERBOSE: 2018/09/26 10:35:38 config.go:215:    - TTL:  60
VERY VERBOSE: 2018/09/26 10:35:38 config.go:216:    - Timeout:  5
VERY VERBOSE: 2018/09/26 10:35:38 config.go:217:    - StateTimeoutSeconds:  300
VERY VERBOSE: 2018/09/26 10:35:38 config.go:218:    - Resolvers: 8.8.8.8
VERY VERBOSE: 2018/09/26 10:35:38 config.go:219:    - ExternalOn:  true
VERY VERBOSE: 2018/09/26 10:35:38 config.go:220:    - SOAMname: ns1.mesos.
VERY VERBOSE: 2018/09/26 10:35:38 config.go:221:    - SOARname: root.ns1.mesos.
VERY VERBOSE: 2018/09/26 10:35:38 config.go:222:    - SOASerial:  1537976138
VERY VERBOSE: 2018/09/26 10:35:38 config.go:223:    - SOARefresh:  60
VERY VERBOSE: 2018/09/26 10:35:38 config.go:224:    - SOARetry:  600
VERY VERBOSE: 2018/09/26 10:35:38 config.go:225:    - SOAExpire:  86400
VERY VERBOSE: 2018/09/26 10:35:38 config.go:226:    - SOAExpire:  60
VERY VERBOSE: 2018/09/26 10:35:38 config.go:227:    - RecurseOn:  true
VERY VERBOSE: 2018/09/26 10:35:38 config.go:228:    - HttpPort:  8123
VERY VERBOSE: 2018/09/26 10:35:38 config.go:229:    - HttpOn:  true
VERY VERBOSE: 2018/09/26 10:35:38 config.go:230:    - ConfigFile:  /etc/mesos-dns/config.json
VERY VERBOSE: 2018/09/26 10:35:38 config.go:231:    - EnforceRFC952:  false
VERY VERBOSE: 2018/09/26 10:35:38 config.go:232:    - IPSources:  [mesos host]
VERY VERBOSE: 2018/09/26 10:35:38 config.go:233:    - EnumerationOn true
VERY VERBOSE: 2018/09/26 10:35:38 config.go:234:    - MesosHTTPSOn false
VERY VERBOSE: 2018/09/26 10:35:38 config.go:235:    - CACertFile
VERY VERBOSE: 2018/09/26 10:35:38 config.go:236:    - MesosAuthentication:
```

6. You can now test your dns setup. Open a new command prompt from the windows jumpbox and start *nslookup*

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\user>nslookup
Default Server:  b.resolvers.Level3.net
Address:  4.2.2.2

> server 10.2.10.22
Default Server:  [10.2.10.22]
Address:  10.2.10.22

> www.google.com
Server:  [10.2.10.22]
Address:  10.2.10.22

Non-authoritative answer:
Name:    www.google.com
Addresses:  2607:f8b0:4007:80e::2004
         172.217.14.100

> master.mesos
Server:  [10.2.10.22]
Address:  10.2.10.22

Name:    master.mesos
Address:  10.2.10.21

>
```

7. Stop your test mesos-dns app by typing "CTRL-c"

Warning: The next steps will fail if you don't stop your test mesos-dns app

Launch Mesos-DNS In Marathon

1. Launch the mesos-dns image in marathon. Connect to marathon, click on *Create Application* and enable *JSON Mode*. Copy the following JSON block over the default and click *Create Application*.

```
{
   "cmd": "/usr/local/mesos-dns/mesos-dns-v0.6.0-linux-amd64 -config=/etc/mesos-
↪dns/config.json -v=10",
   "cpus": 0.2,
   "mem": 256,
   "id": "mesos-dns",
   "instances": 1,
   "constraints": [["hostname", "CLUSTER", "10.2.10.22"]]
}
```

2. Update /etc/resolv.conf on **all agents** by adding our mesos-dns nameserver to our /etc/resolv.conf file. SSH to mesos-agent1 & 2.

```
sed -i /nameserver/s/.*/"nameserver 10.2.10.22"/ /etc/resolv.conf
```

Note: If you have deployed your instances in a cloud like AWS, it is likely that you'll lose your DNS setup after a reboot. If you want to make your changes persist, you need to update /etc/dhcp/dhclient.conf to supersede the dhcp setup. More information here: Static DNS server in a EC2 instance

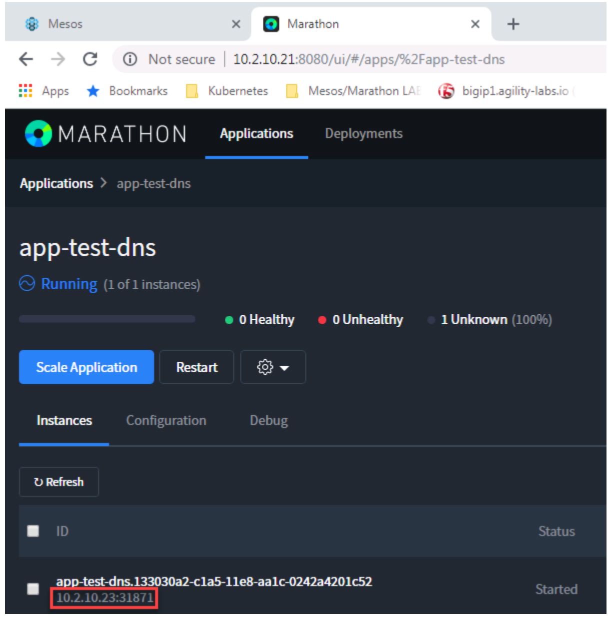
Test Mesos-DNS

To test our Mesos DNS setup, we will start a new application and check if it automatically gets a DNS name.

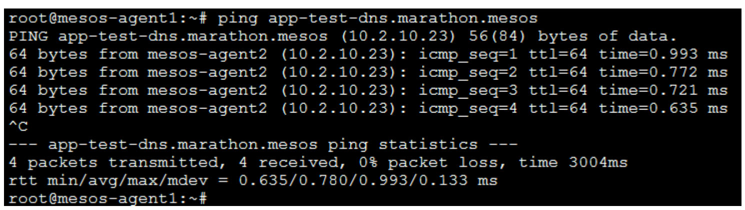
1. Start a new app in marathon:

```
{
   "id": "app-test-dns",
   "cpus": 0.5,
   "mem": 32.0,
   "container": {
      "type": "DOCKER",
      "docker": {
         "image": "eboraas/apache-php",
         "network": "BRIDGE",
         "portMappings": [
            { "containerPort": 80, "hostPort": 0 }
         ]
      }
   }
}
```

2. Once it's running, go to one of your slaves and run ping app-test-dns.marathon.mesos. It should work and return the agent IP.



3. If you don't try to ping from mesos-agent1 or mesos-agent2, make sure your client can reach mesos-dns server first (10.2.10.22)



4.3 Module 3: F5 Container Connector with Mesos / Marathon

4.3.1 Overview

F5 Container connector in Mesos / Marathon is called: F5 Marathon BIG-IP Controller.

The F5 Marathon BIG-IP Controller is a container-based Marathon Application – marathon-bigip-ctlr. You can launch the F5 Marathon BIG-IP Controller in Marathon via the Marathon REST API or the Marathon Web Interface.

The marathon-bigip-ctlr watches the Marathon API for special "F5 Application Labels" that tell it:

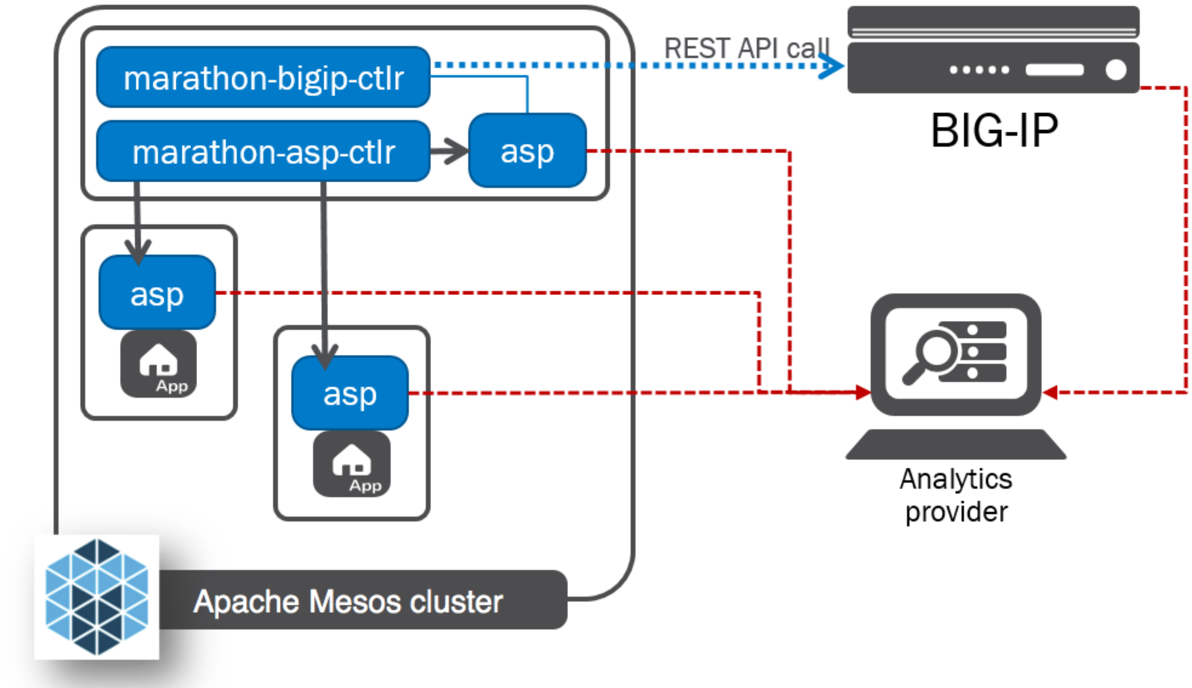
- What Application we want it to manage
- How we want to configure the BIG-IP for that specific Application.

You can manage BIG-IP objects directly, or deploy iApps, with the F5 Marathon BIG-IP Controller.

See also:

The official F5 documentation is here: F5 Container Connector - Marathon

4.3.2 Architecture



In Marathon, you can associate labels with Application tasks for tracking/reporting purposes. F5 has developed a set of custom "F5 Application Labels" as a way notify the F5 Marathon BIG-IP Controller that they have work to do.

When the F5 Marathon BIG-IP Controller discovers Applications with new or updated F5 Application Labels, it dynamically creates iApps or virtual servers, pools, pool members, and HTTP health monitors for each of the Application's tasks.

See also:

If you want to have more details about the F5 Application Labels, you may go to the F5 official documentation here: F5 BIG-IP Controller for Marathon

4.3.3 Prerequisites

Before being able to use the Container Connecter, you need to handle some prerequisites

- You must have a fully active/licensed BIG-IP
- A BIG-IP partition needs to be setup for the Container connector. You need to have access to a user with the right privileges
- You need a user with administrative access to this partition
- Your Mesos / Marathon environment must be up and running already

Lab 3.1 - F5 Container Connector Setup

The BIG-IP Controller for Marathon installs as an Application

See also:

The official CC documentation is here: Install the BIG-IP Controller: Marathon

BIG-IP Setup

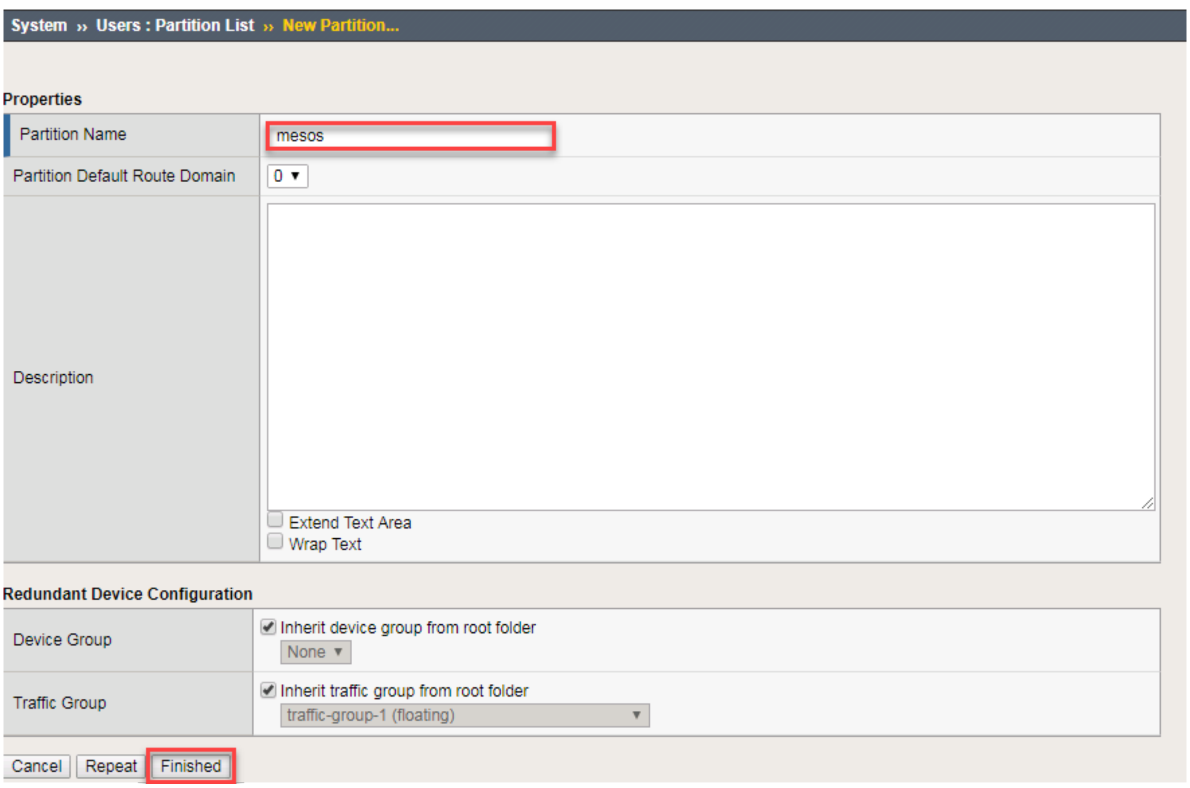
To use F5 Container connector, you'll need a BIG-IP up and running first.

Through the Jumpbox, you should have a BIG-IP available at the following URL: https://10.1.1.245

> **Warning:** Connect to your BIG-IP and check it is active and licensed. Its login and password are: **admin/admin**
>
> If your BIG-IP has no license or its license expired, renew the license. You just need a LTM VE license for this lab. No specific add-ons are required (ask a lab instructor for eval licenses if your license has expired)

1. You need to setup a partition that will be used by F5 Container Connector.

```
# From the CLI:
tmsh create auth partition mesos

# From the UI:
GoTo System --> Users --> Partition List
- Create a new partition called "mesos" (use default settings)
- Click Finished
```



With the new partition created, we can go back to Marathon to setup the F5 Container connector.

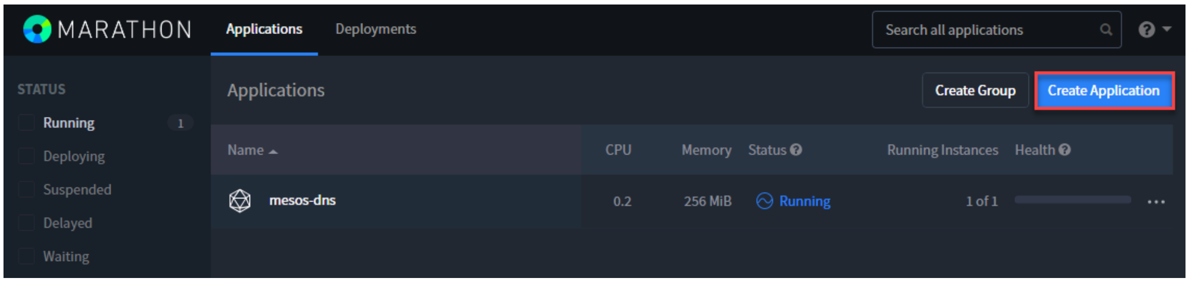
Container Connector Deployment

See also:

For a more thorough explanation of all the settings and options see F5 Container Connector - Marathon

Now that BIG-IP is licensed and prepped with the “mesos” partition, we need to deploy our Marathon BIG-IP Controller, we can either use Marathon UI or use the Marathon REST API. For this class we will be using the Marathon UI.

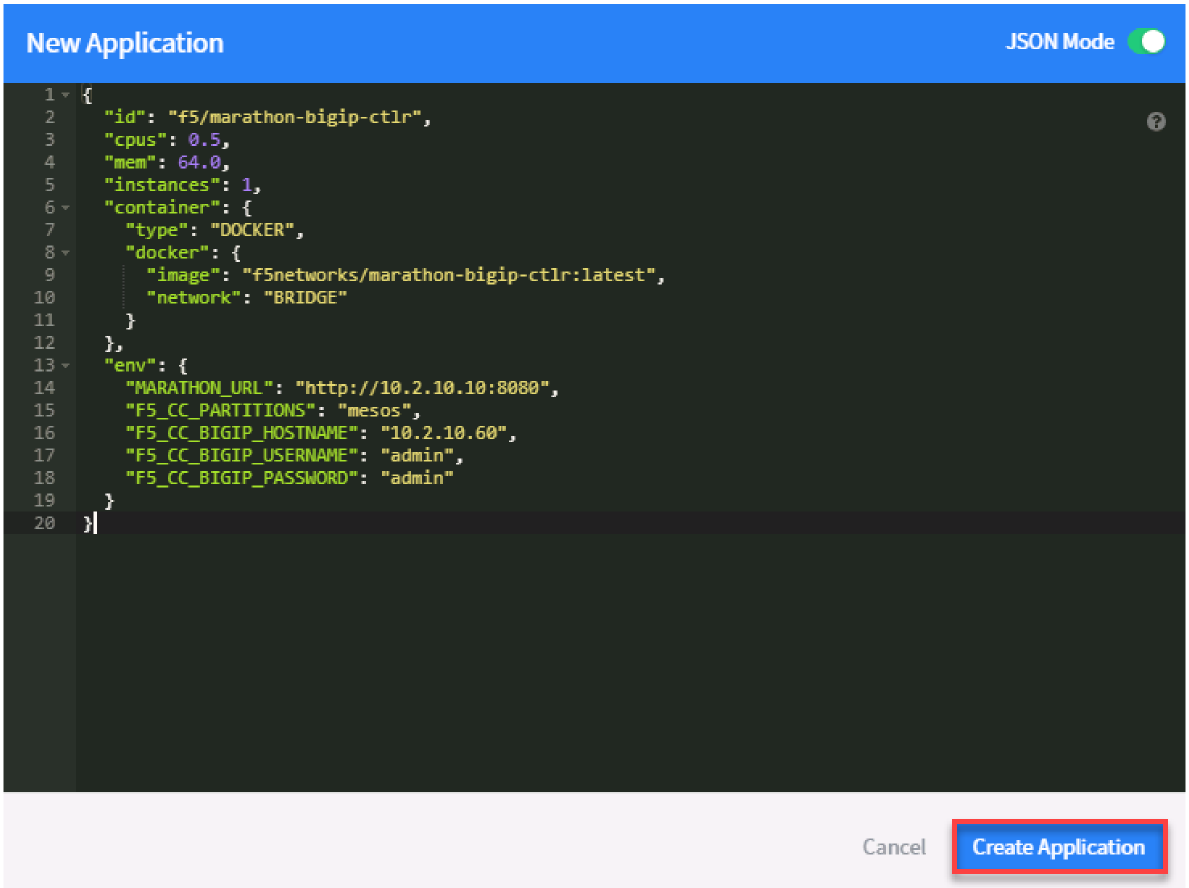
1. From the jumpbox connect to the Marathon UI at http://10.2.10.21:8080 and click “Create Application”.



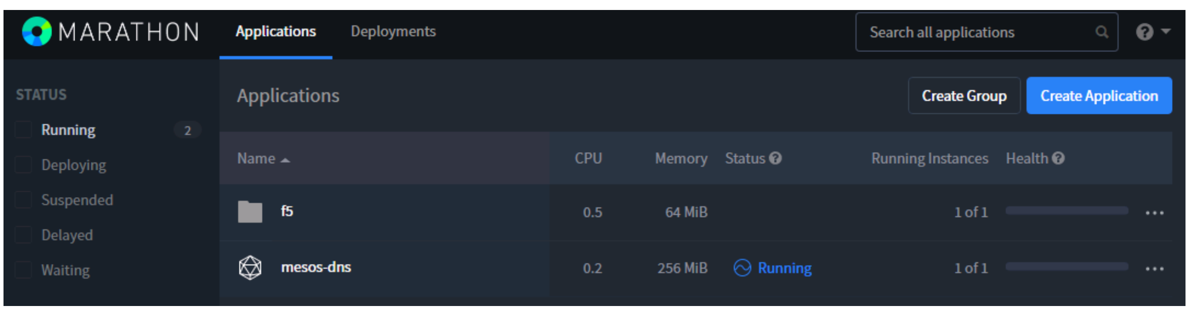
2. Click on “JSON mode” in the top-right corner

3. **REPLACE** the 8 lines of default JSON code shown with the following JSON code and click Create Application

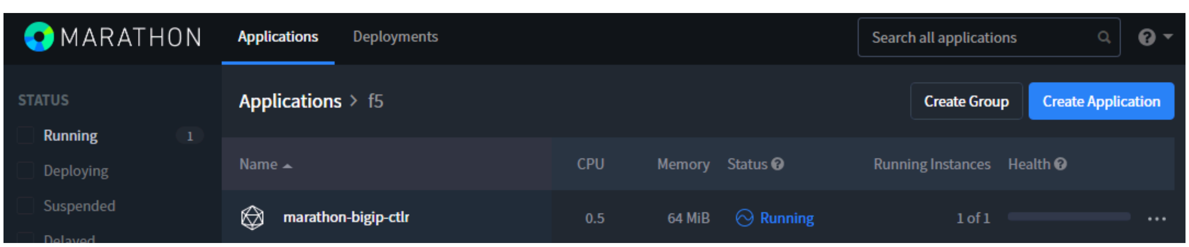
```
{
  "id": "f5/marathon-bigip-ctlr",
  "cpus": 0.5,
  "mem": 64.0,
  "instances": 1,
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "f5networks/marathon-bigip-ctlr:latest",
      "network": "BRIDGE"
    }
  },
  "env": {
    "MARATHON_URL": "http://10.2.10.21:8080",
    "F5_CC_PARTITIONS": "mesos",
    "F5_CC_BIGIP_HOSTNAME": "10.2.10.60",
    "F5_CC_BIGIP_USERNAME": "admin",
    "F5_CC_BIGIP_PASSWORD": "admin"
  }
}
```



4. After a few seconds you should have a new folder labeled "f5" as shown in this picture.



5. Click on the "f5" folder and you should have "Running", the BIG-IP North/South Controller labeled marathon-bigip-ctrl.



Note: If you're running the lab outside of Agility, you need may need to update the field *image* with the appropriate path to your image:

- Load it on **all your agents/nodes** with the docker pull command. **sudo docker pull f5networks/marathon-bigip-ctlr:latest** for the latest version.
- Load it on a system and push it into your registry if needed.
- If your Mesos environment use authentication, here is a link explaining how to handle authentication with the Marathon BIG-IP Controller: Set up authentication to your secure DC/OS cluster

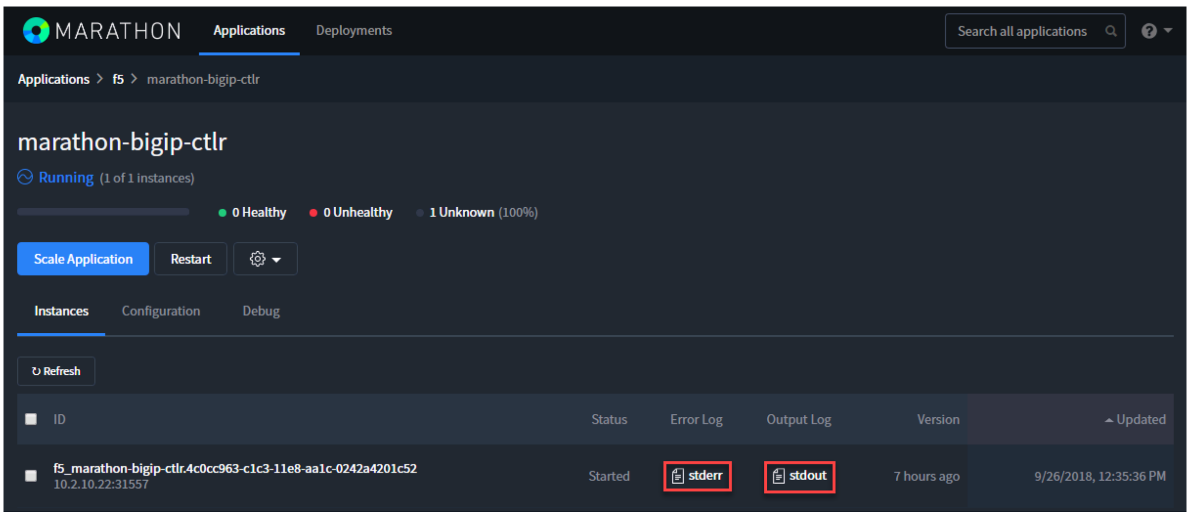
---

Troubleshooting

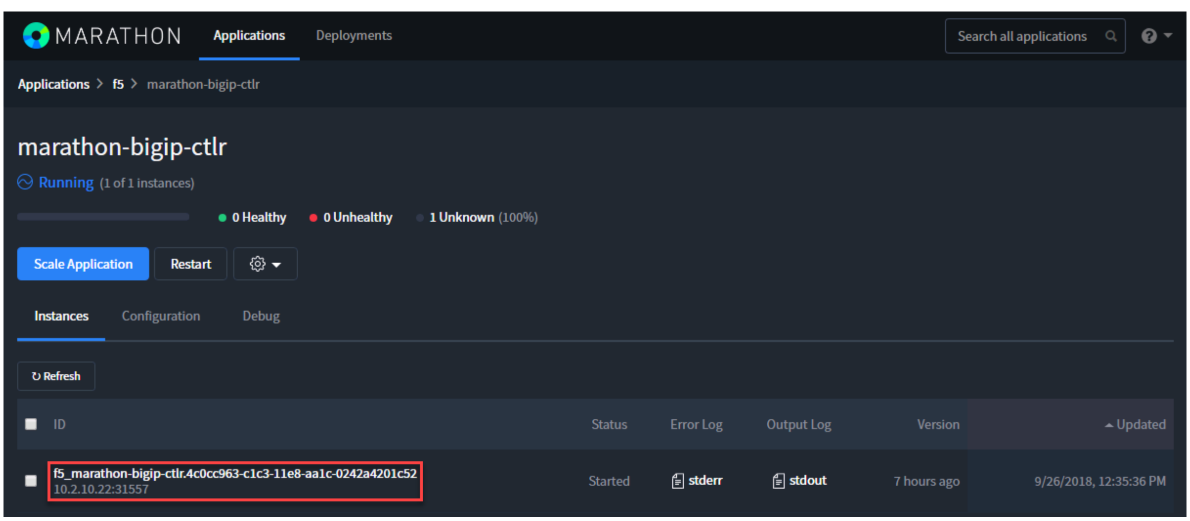
If you need to troubleshoot your container, you have two different ways to check the logs of your container, Marathon UI or Docker command.

1. Using the Marathon UI Click on Applications –> the f5 folder –> marathon-bigip-ctlr. From here you can download and view the logs from the text editor of choice.

   You should see something like this:

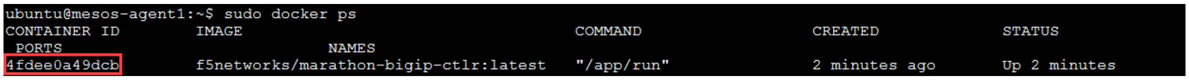
2. Using docker log command: You need to identify where the Controller is running. From the previous step we can see it's running on 10.2.10.22 (which is **mesos-agent1**).

   Connect via SSH to **mesos-agent1** and run the following commands:

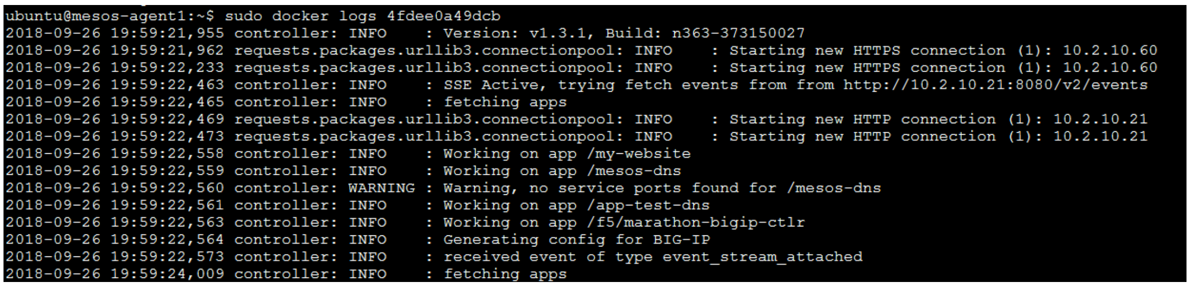
```
sudo docker ps
```

This command will give us the Controllers Container ID, here it is: 4fdee0a49dcb. We need this ID for the next command.


```
ubuntu@mesos-agent1:~$ sudo docker ps
CONTAINER ID        IMAGE                                    COMMAND             CREATED             STATUS
 PORTS                          NAMES
4fdee0a49dcb        f5networks/marathon-bigip-ctlr:latest    "/app/run"          2 minutes ago       Up 2 minutes
```

To check the logs of our Controller:

```
sudo docker logs 4fdee0a49dcb
```


```
ubuntu@mesos-agent1:~$ sudo docker logs 4fdee0a49dcb
2018-09-26 19:59:21,955 controller: INFO    : Version: v1.3.1, Build: n363-373150027
2018-09-26 19:59:21,962 requests.packages.urllib3.connectionpool: INFO    : Starting new HTTPS connection (1): 10.2.10.60
2018-09-26 19:59:22,233 requests.packages.urllib3.connectionpool: INFO    : Starting new HTTPS connection (1): 10.2.10.60
2018-09-26 19:59:22,463 controller: INFO    : SSE Active, trying fetch events from from http://10.2.10.21:8080/v2/events
2018-09-26 19:59:22,465 controller: INFO    : fetching apps
2018-09-26 19:59:22,469 requests.packages.urllib3.connectionpool: INFO    : Starting new HTTP connection (1): 10.2.10.21
2018-09-26 19:59:22,473 requests.packages.urllib3.connectionpool: INFO    : Starting new HTTP connection (1): 10.2.10.21
2018-09-26 19:59:22,558 controller: INFO    : Working on app /my-website
2018-09-26 19:59:22,559 controller: INFO    : Working on app /mesos-dns
2018-09-26 19:59:22,560 controller: WARNING : Warning, no service ports found for /mesos-dns
2018-09-26 19:59:22,561 controller: INFO    : Working on app /app-test-dns
2018-09-26 19:59:22,563 controller: INFO    : Working on app /f5/marathon-bigip-ctlr
2018-09-26 19:59:22,564 controller: INFO    : Generating config for BIG-IP
2018-09-26 19:59:22,573 controller: INFO    : received event of type event_stream_attached
2018-09-26 19:59:24,009 controller: INFO    : fetching apps
```

3. You can connect to your container with docker as well:

```
sudo docker exec -it 4fdee0a49dcb /bin/sh

cd /app

ls -la

exit
```

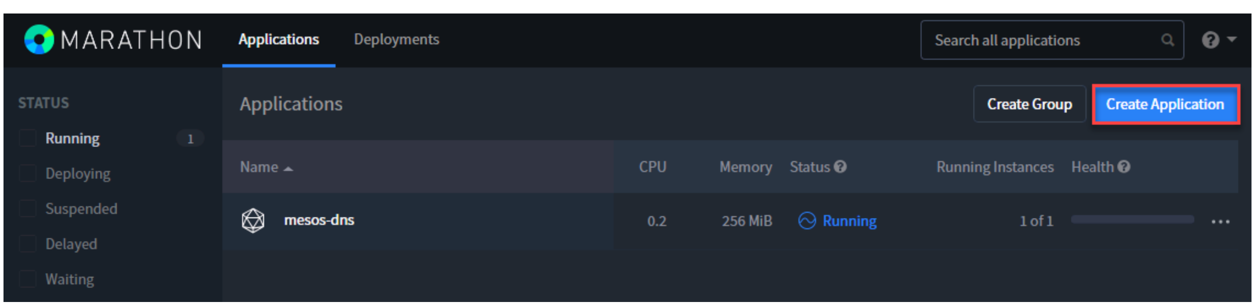
Lab 3.2 - F5 Container Connector Usage

Now that our container connector is up and running, let's deploy an application and leverage our F5 CC.

For this lab we'll use a simple pre-configured docker image called "f5-hello-world". It can be found on docker hub at f5devcentral/f5-hello-world

App Deployment

From the jumpbox connect to the Marathon UI at http://10.2.10.21:8080 and click "Create Application".



1. Click on "JSON mode" in the top-right corner

2. **REPLACE** the 8 lines of default JSON code shown with the following JSON code and click Create Application

```
1  {
2    "id": "f5-hello-world",
3    "cpus": 0.1,
4    "mem": 128.0,
5    "instances": 2,
6    "container": {
7      "type": "DOCKER",
8      "docker": {
9        "image": "f5devcentral/f5-hello-world:latest",
10       "network": "BRIDGE",
11       "forcePullImage": false,
12       "portMappings": [
13         { "containerPort": 8080, "hostPort": 0, "protocol": "tcp" }
14       ]
15     }
16   },
17   "labels": {
18     "F5_PARTITION": "mesos",
19     "F5_0_BIND_ADDR": "10.2.10.81",
20     "F5_0_MODE": "http",
21     "F5_0_BALANCE": "round-robin",
22     "F5_0_PORT": "80"
23   },
24   "healthChecks": [
25     {
26       "protocol": "HTTP",
27       "portIndex": 0,
28       "path": "/",
29       "gracePeriodSeconds": 5,
30       "intervalSeconds": 16,
31       "maxConsecutiveFailures": 3
32     }
33   ]
34 }
```

3. F5-Hello-World is “Deploying”

Note: The JSON app definition specified several things:

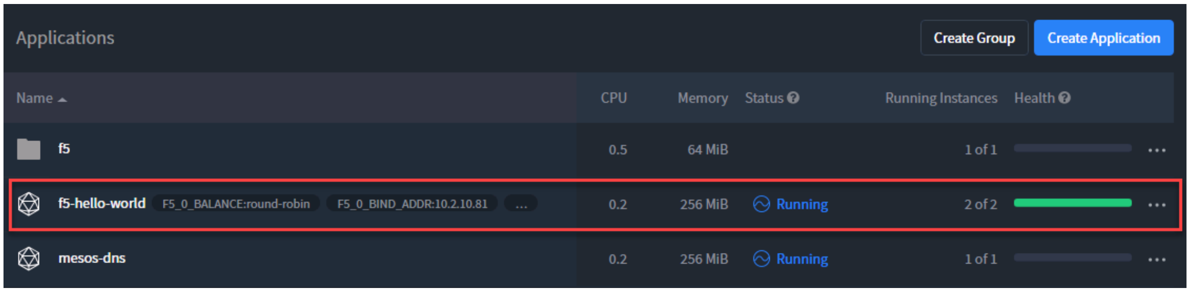
(a) What container image to use (line 9)

(b) The BIG-IP configuration (Partition, VS IP, VS Port).

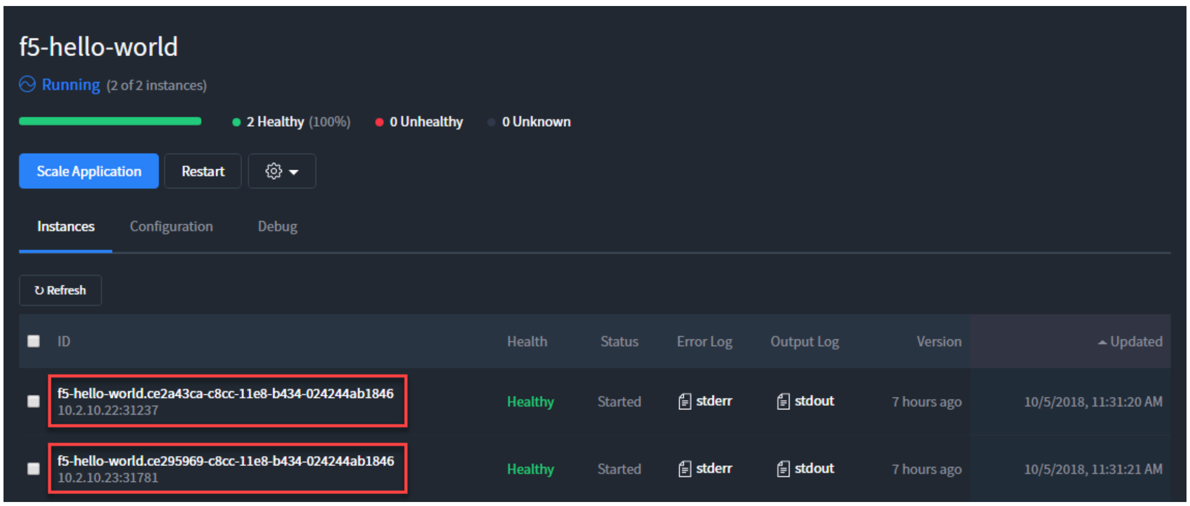
(c) The Marathon health check for this app. The BIG-IP will replicate those health checks.

(d) The number of instances (line 5)

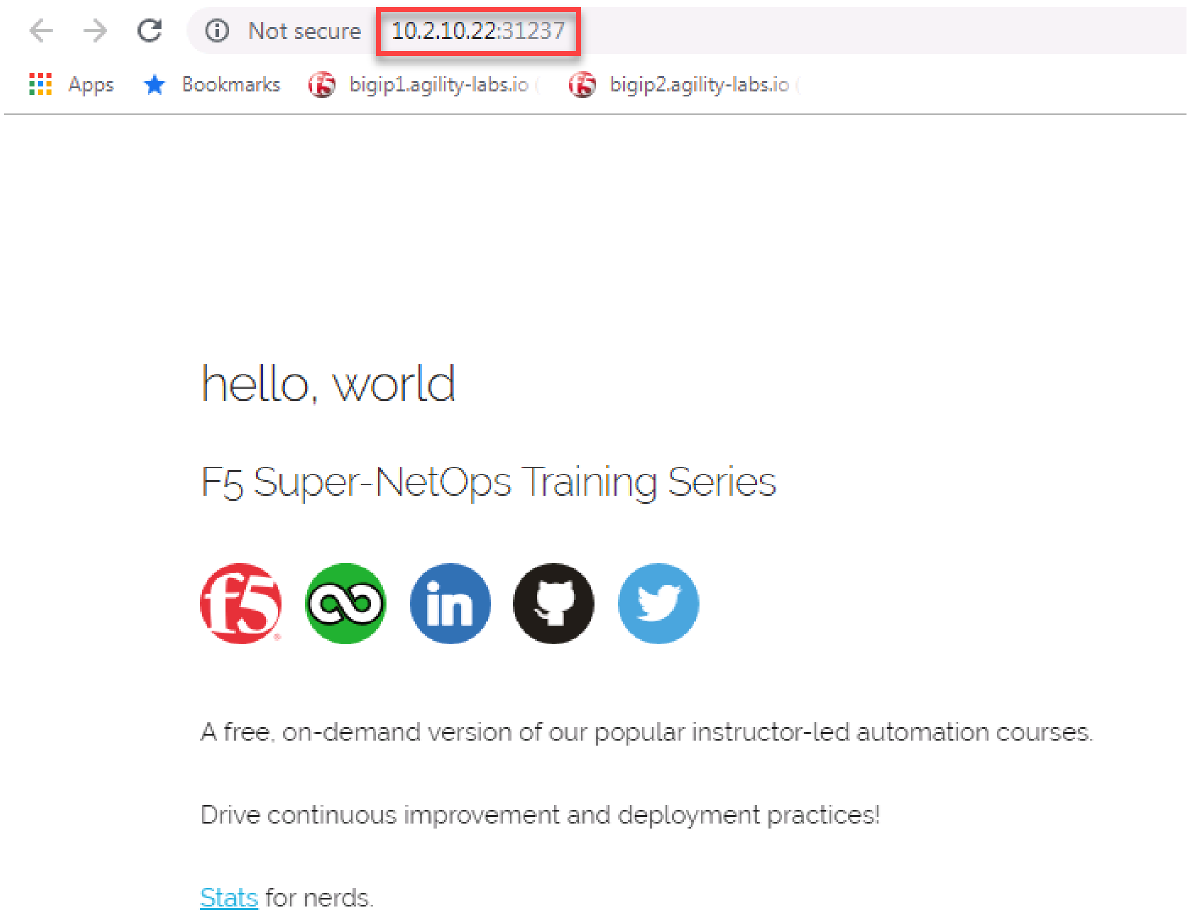
Wait for your application to be successfully deployed and be in a running state.



4. Click on “f5-hello-world”. Here you will see two instance deployed, with their node IP and Port.



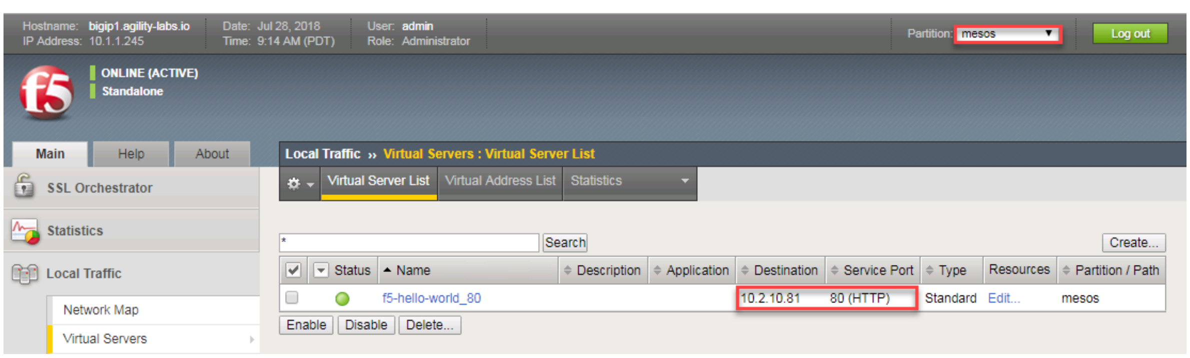
5. Click on one of the <IP:Port> assigned to be redirect there:



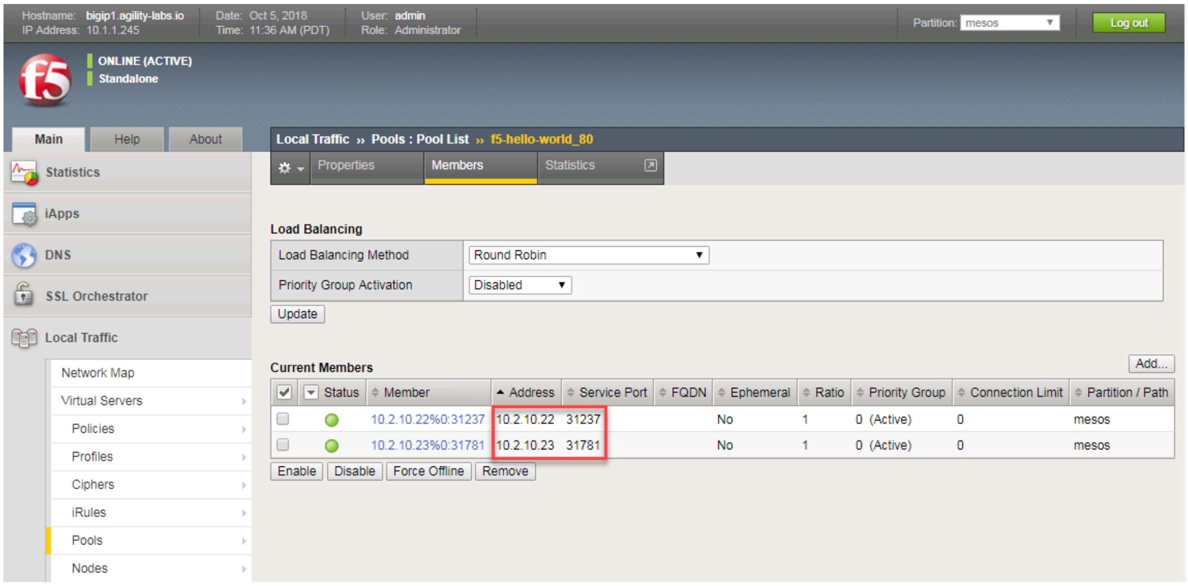
6. We can check whether the Marathon BIG-IP Controller has updated our BIG-IP configuration accordingly. Connect to your BIG-IP on https://10.1.1.245 and go to Local Traffic –> Virtual Server.

> **Warning:** Don't forget to select the "mesos" partition or you'll see nothing.

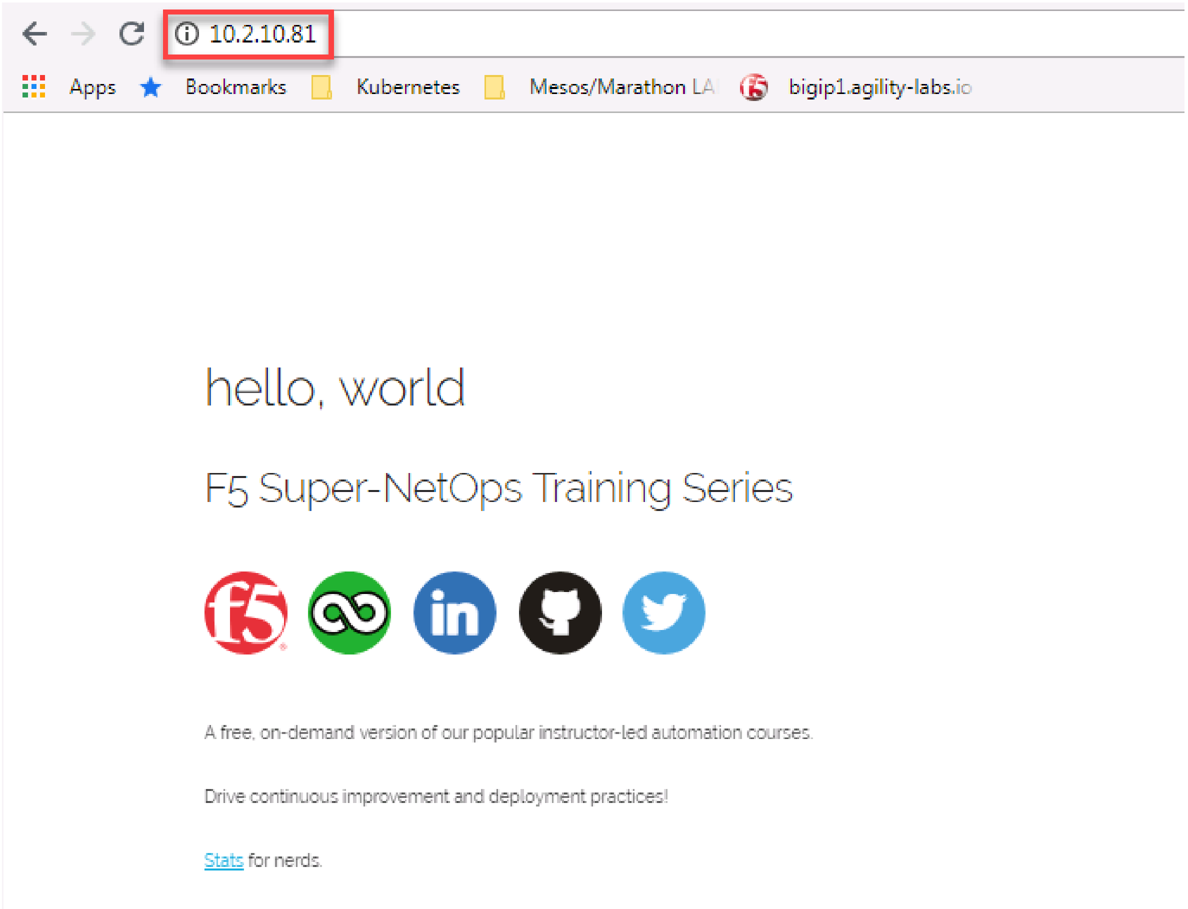
You should have something like this:



7. Go to Local Traffic –> Pool –> "f5-hello-world_80" –> Members. Here we can see that two pool members are defined and the IP:Port match ou deployed app in Marathon.

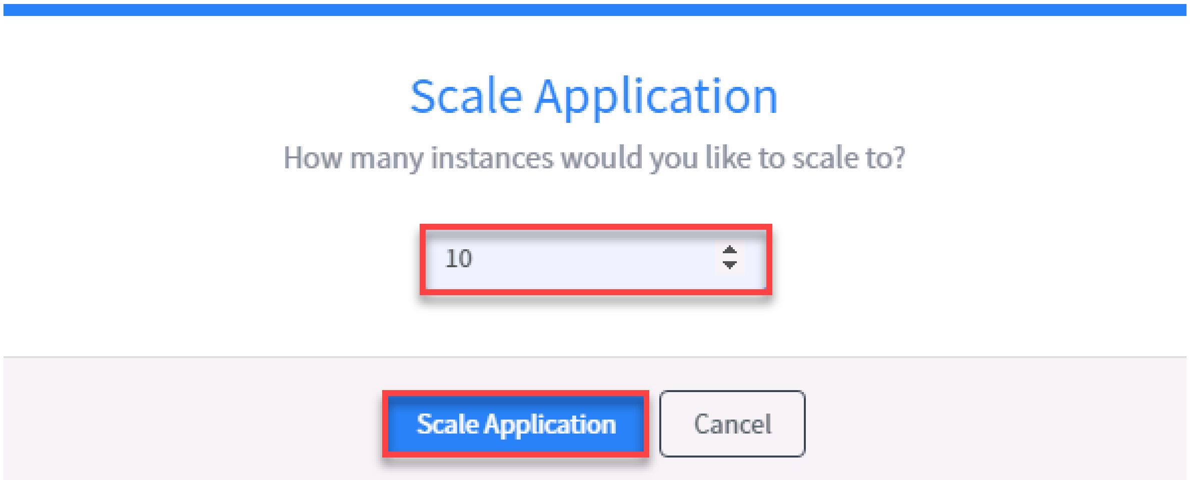


8. You should be able to access the application. In your browser try to connect to http://10.2.10.81

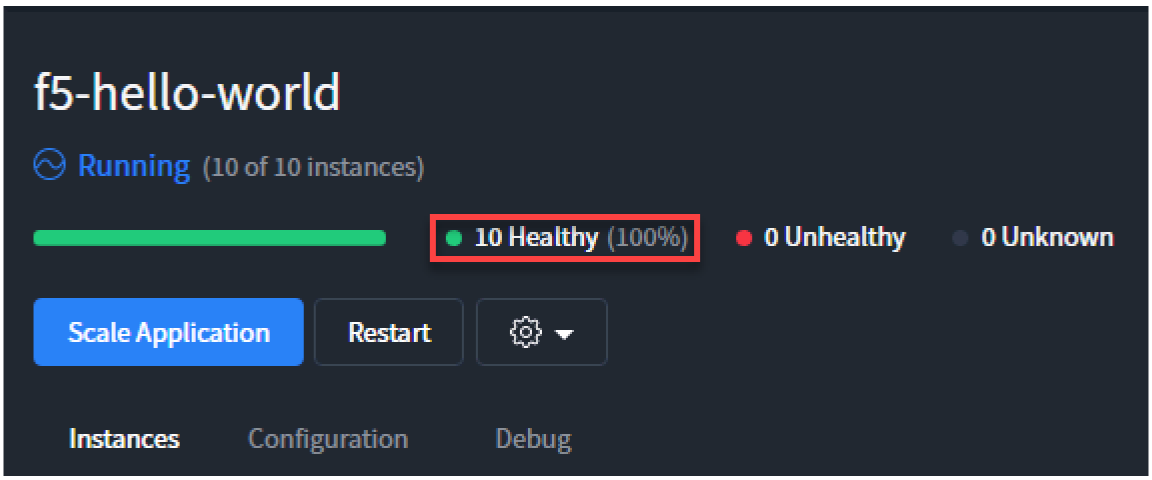


9. Scale the f5-hello-world app. Go back to the Marathon UI (http://10.2.10.21:8080). Go to Applications –> “f5-hello-world” and click “Scale Application”.

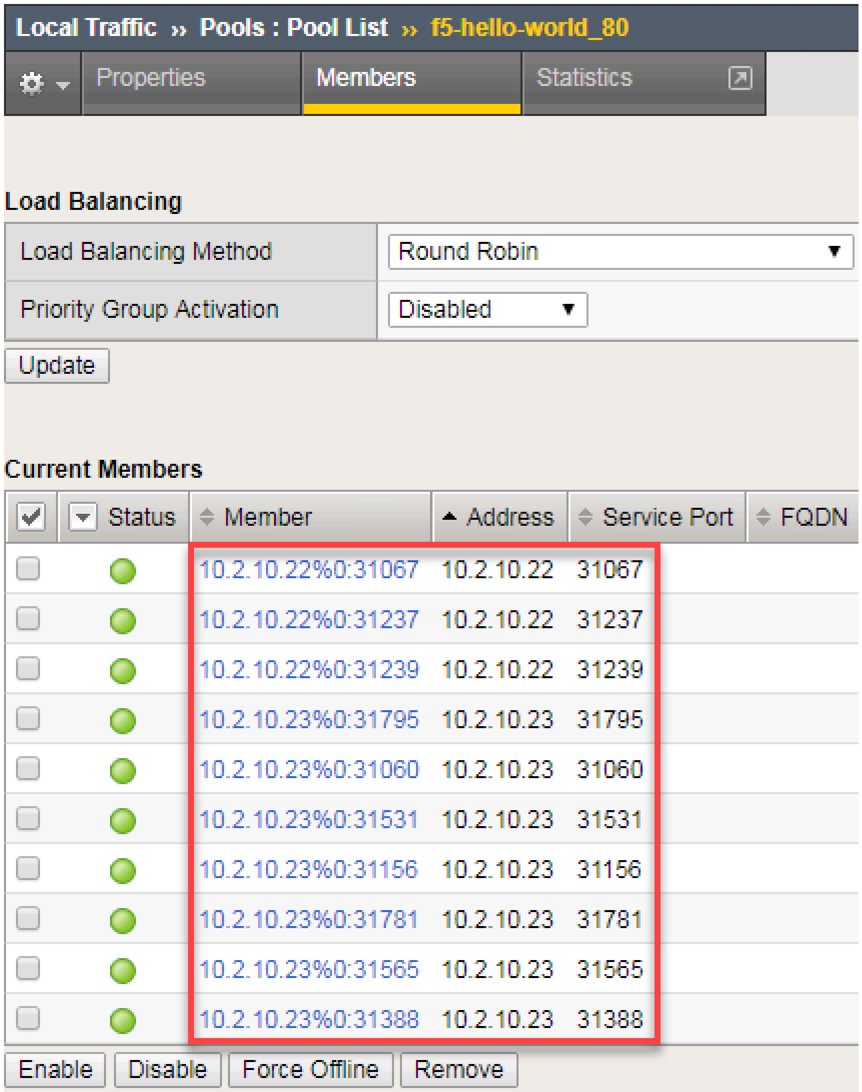
   Let’s increase the number from *2* to *10* instances and click on “Scale Application”.



Once it is done you should see 10 “healthy instances” running in Marathon UI.



You can also check your pool members list on your BIG-IP.



As we can see, the Marathon BIG-IP Controller is adapting the pool members setup based on the number of instances delivering this application automatically.

10. Scale back the application to *2* to save resources for the next labs.

Expected time to complete: **1 hours**

> **Attention: MODULE 2: BUILD A MESOS / MARATHON CLUSTER CAN BE SKIPPED. THE BLUEPRINT IS PRE-CONFIGURED WITH A WORKING CLUSTER. THIS MODULE IS FOR DOCUMENTION PURPOSES ONLY.**

4.4 Lab Setup

We will leverage the following setup to configure the Mesos / Marathon environment.

Hostname	IP-ADDR	Credentials
jumpbox	10.1.1.250	user/Student!Agility!
bigip1	10.1.1.245 10.2.10.60	admin/admin root/default
mesos-master1	10.2.10.21	ubuntu/ubuntu root/default
mesos-agent1	10.2.10.22	ubuntu/ubuntu root/default
mesos-agent2	10.2.10.23	ubuntu/ubuntu root/default

5 Class 4: Introduction to RedHat OpenShift

This introductory class covers the following topics:

5.1 Module 1: Build an Openshift Cluster

Attention: THIS MODULE CAN BE SKIPPED. THE BLUEPRINT IS PRE-CONFIGURED WITH A WORKING CLUSTER. THIS MODULE IS FOR DOCUMENTION PURPOSES ONLY.

In this module, we will build a 3 node cluster (1x master and 2x nodes) utilizing CentOS server images.

As a reminder, in this module, our cluster setup is:

Hostname	IP-ADDR	Role
ose-master1	10.3.10.21	Master
ose-node1	10.3.10.22	Node
ose-node2	10.3.10.23	Node

5.1.1 Lab 1.1 - Prep CentOS

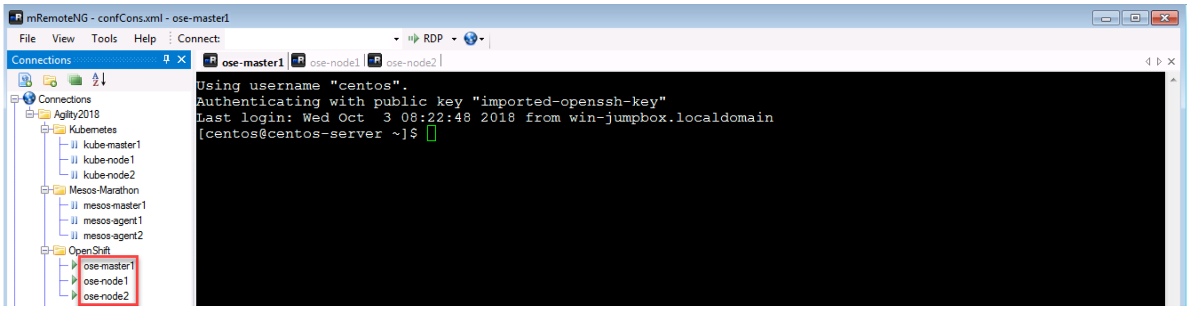
Note:

- This installation will utilize centOS v7.5.
- SSH keys were configured to allow the jumphost to login without a passwd as well as between the master & nodes to facilitate the Ansible playbooks.

Important: The following commands need to be run on all three nodes unless otherwise specified.

1. From the jumpbox open **mRemoteNG** and start a session to each of the following servers. The sessions are pre-configured to connect with the default user "centos".
    - ose-master1

- ose-node1
- ose-node2



2. For your convenience we've already added the host IP & names to /etc/hosts. Verify the file

```
cat /etc/hosts
```

The file should look like this:

```
[centos@ose-master1 openshift]$ cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6

10.3.10.21 ose-master1
10.3.10.22 ose-node1
10.3.10.23 ose-node2
[centos@ose-master1 openshift]$
```

If entries are not there add them to the bottom of the file be editing "/etc/hosts" with 'vim'

```
sudo vim /etc/hosts

#cut and paste the following lines to /etc/hosts

10.3.10.21    ose-master1
10.3.10.22    ose-node1
10.3.10.23    ose-node2
```

3. Ensure the OS is up to date

```
sudo yum update -y

#This can take a few seconds to several minutes depending on demand to download
↪the latest updates for the OS.
```

4. Install the docker packages

```
sudo yum install -y docker
sudo systemctl start docker && sudo systemctl enable docker
```

5. Verify docker is up and running

```
sudo docker run --rm hello-world
```

If everything is working properly you should see the following message

```
[centos@centos-server ~]$ sudo docker run --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

[centos@centos-server ~]$
```

5.1.2 Lab 1.2 - Install Openshift

Important: The following commands need to be run on the **master** only, unless otherwise specified.

1. Install Ansible

```
sudo yum install -y epel-release
sudo yum install -y ansible
```

2. Disable "epel-release"

```
vim /etc/yum.repos.d/epel.repo

# Change the value enabled=1 to 0 (zero).
```

Note: This is done to prevent future OS updates from including packages from outside the standard packages.

3. Prep openshift AUTH

```
sudo mkdir -p /etc/origin/master/
sudo touch /etc/origin/master/htpasswd
```

4. Clone the openshift-ansible repo

```
git clone -b release-3.10 https://github.com/openshift/openshift-ansible.git
↪$HOME/openshift-ansible
```

5. Install Openshift with Ansible

```
ansible-playbook -i $HOME/agilitydocs/openshift/ansible/inventory.ini $HOME/
↪openshift-ansible/playbooks/prerequisites.yml
ansible-playbook -i $HOME/agilitydocs/openshift/ansible/inventory.ini $HOME/
↪openshift-ansible/playbooks/deploy_cluster.yml
```

Note: If needed, to uninstall Openshift run the following command:

```
ansible-playbook -i $HOME/agilitydocs/openshift/ansible/inventory.ini $HOME/
↪openshift-ansible/playbooks/adhoc/uninstall.yml
```

Here's the "inventory" (refrenced by our ansible playbook) used for the deployment.

```
[OSEv3:children]
masters
nodes
etcd

[masters]
10.3.10.21 openshift_ip=10.3.10.21

[etcd]
10.3.10.21 openshift_ip=10.3.10.21

[nodes]
10.3.10.21 openshift_ip=10.3.10.21 openshift_schedulable=true openshift_node_
↪group_name="node-config-master-infra"
10.3.10.22 openshift_ip=10.3.10.22 openshift_schedulable=true openshift_node_
↪group_name="node-config-compute"
10.3.10.23 openshift_ip=10.3.10.23 openshift_schedulable=true openshift_node_
↪group_name="node-config-compute"

[OSEv3:vars]
ansible_ssh_user=centos
ansible_become=true
enable_excluders=false
enable_docker_excluder=false
ansible_service_broker_install=false

containerized=true
openshift_disable_check=disk_availability,memory_availability,docker_storage,
↪docker_image_availability

deployment_type=origin
openshift_deployment_type=origin

openshift_master_identity_providers=[{'name': 'htpasswd_auth', 'login': 'true',
↪'challenge': 'true', 'kind': 'HTPasswdPasswordIdentityProvider'}]

openshift_public_hostname=ose-master1
openshift_master_api_port=8443
openshift_master_console_port=8443

openshift_metrics_install_metrics=false
openshift_logging_install_logging=false
```

6. Enable oc bash completion

```
oc completion bash >>/etc/bash_completion.d/oc_completion
```

7. Add user "centos" to openshift users

```
sudo htpasswd -b /etc/origin/master/htpasswd centos centos
```

8. Add user "centos" to "cluster-admin"

```
oc adm policy add-cluster-role-to-user cluster-admin centos
```

5.2 Module 2: F5 Container Connector with RedHat OpenShift

Red Hat's OpenShift Origin is a containerized application platform with a native Kubernetes integration. The BIG-IP Controller for Kubernetes enables use of a BIG-IP device as an edge load balancer, proxying traffic from outside networks to pods inside an OpenShift cluster. OpenShift Origin uses a pod network defined by the OpenShift SDN.

The F5 Integration for Kubernetes overview describes how the BIG-IP Controller works with Kubernetes. Because OpenShift has a native Kubernetes integration, the BIG-IP Controller works essentially the same in both environments. It does have a few OpenShift-specific prerequisites.

Today we are going to go through a prebuilt OpenShift environment with some locally deployed yaml files.

5.2.1 Lab 2.1 - F5 Container Connector Setup

The BIG-IP Controller for OpenShift installs as a Deployment object

See also:

The official CC documentation is here: Install the BIG-IP Controller: Openshift

BIG-IP Setup

To use F5 Container connector, you'll need a BIG-IP up and running first.

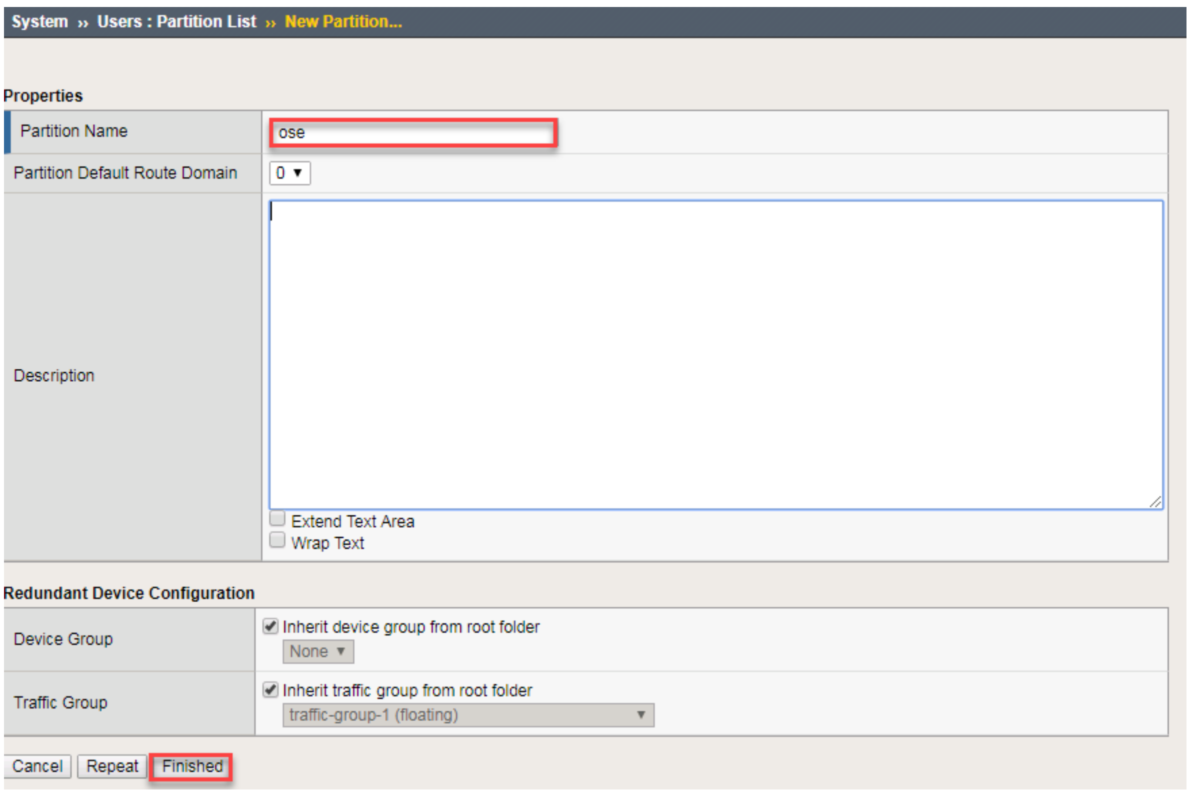
Through the Jumpbox, you should have a BIG-IP available at the following URL: https://10.1.1.245

Warning:

- Connect to your BIG-IP and check it is active and licensed. Its login and password are: **admin/admin**
- If your BIG-IP has no license or its license expired, renew the license. You just need a LTM VE license for this lab. No specific add-ons are required (ask a lab instructor for eval licenses if your license has expired)
- Be sure to be in the `Common` partition before creating the following objects.

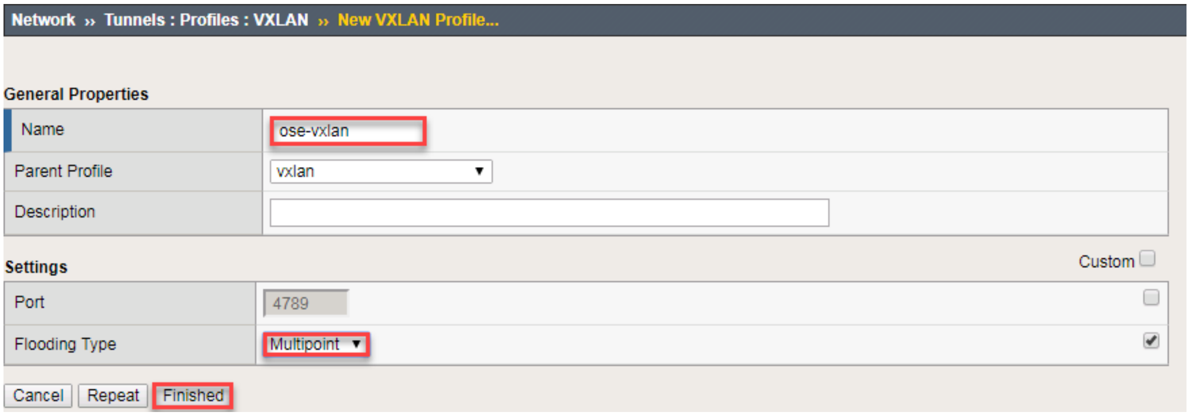
1. You need to setup a partition that will be used by F5 Container Connector.

```
# From the CLI:
tmsh create auth partition ose

# From the UI:
GoTo System --> Users --> Partition List
- Create a new partition called "ose" (use default settings)
- Click Finished
```



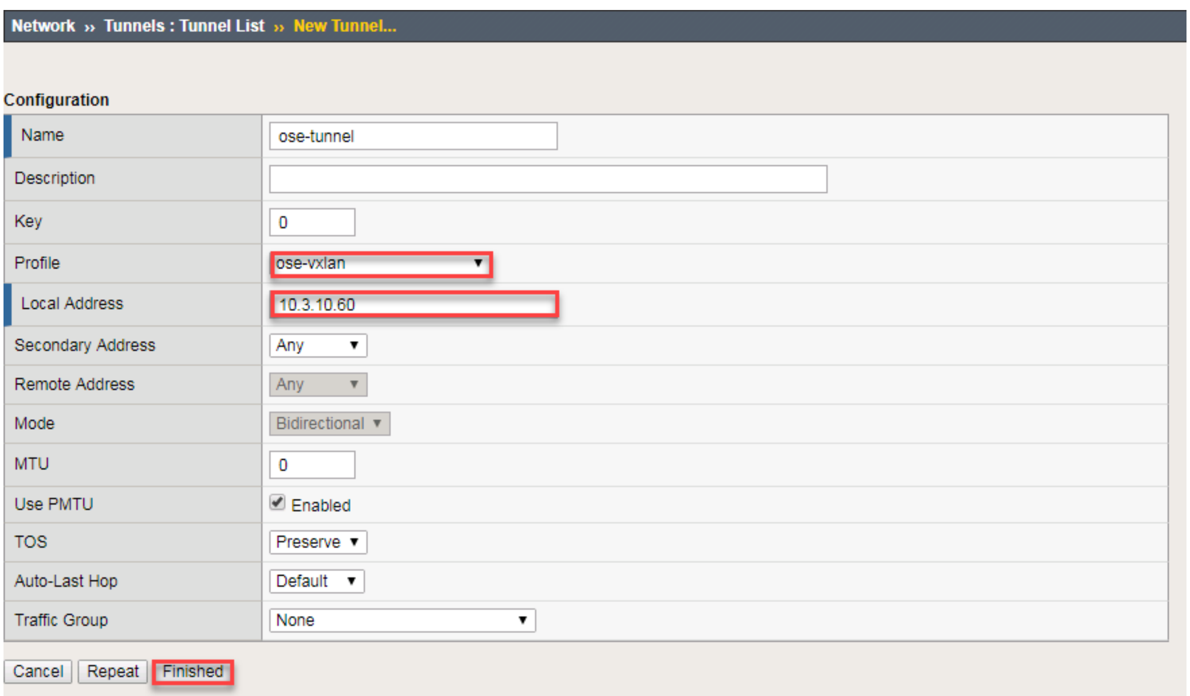
2. Create a vxlan tunnel profile

```
# From the CLI:
tmsh create net tunnel vxlan ose-vxlan {app-service none flooding-type multipoint}

# From the UI:
GoTo Network --> Tunnels --> Profiles --> VXLAN
- Create a new profile called "ose-vxlan"
- Set the Flooding Type = Multipoint
- Click Finished
```



3. Create a vxlan tunnel

```
# From the CLI:
tmsh create net tunnel tunnel ose-tunnel {key 0 local-address 10.3.10.60 profile ose-vxlan}

# From the UI:
GoTo Network --> Tunnels --> Tunnel List
- Create a new tunnel called "ose-tunnel"
- Set the Local Address to 10.3.10.60
- Set the Profile to the one previously created called "ose-vxlan"
- Click Finished
```



Container Connector Deployment

See also:

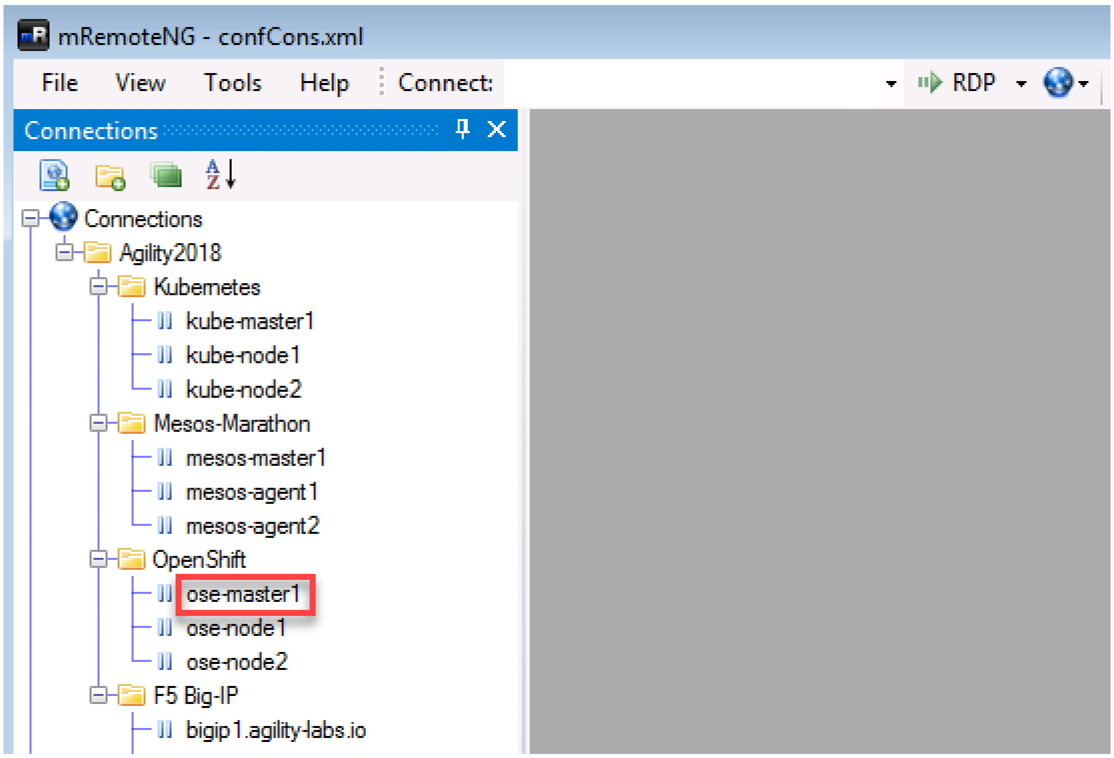
For a more thorough explanation of all the settings and options see F5 Container Connector - Openshift

Now that BIG-IP is licensed and prepped with the “ose” partition, we need to define a Kubernetes deployment and create a Kubernetes secret to hide our bigip credentials.

1. From the jumpbox open **mRemoteNG** and start a session with ose-master.

   Note: As a reminder we’re utilizing a wrapper called **MRemoteNG** for Putty and other services. MRNG hold credentials and allows for multiple protocols(i.e. SSH, RDP, etc.), makes jumping in and out of SSH connections easier.

   On your desktop select **MRemoteNG**, once launched you’ll see a few tabs similar to the example below. Open up the OpenShift Enterprise / OSE-Cluster folder and double click ose-master.

2. “git” the demo files

   Note: These files should be here by default, if **NOT** run the following commands.

```
git clone https://github.com/f5devcentral/f5-agility-labs-containers.git ~/
↪agilitydocs

cd ~/agilitydocs/openshift
```

3. Log in with an Openshift Client.

   Note: Here we’re using a user “centos”, added when we built the cluster. When prompted for password, enter “centos”.

```
oc login -u centos -n default
```

```
[centos@ose-master1 openshift]$ oc login -u centos -n default
Authentication required for https://ose-master1:8443 (openshift)
Username: centos
Password:
Login successful.

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

  * default
    kube-public
    kube-service-catalog
    kube-system
    management-infra
    openshift
    openshift-infra
    openshift-logging
    openshift-node
    openshift-sdn
    openshift-template-service-broker
    openshift-web-console

Using project "default".
[centos@ose-master1 openshift]$
```

Important: Upon logging in you'll notice access to several projects. In our lab well be working from the default "default".

---

4. Create bigip login secret

```
oc create secret generic bigip-login -n kube-system --from-literal=username=admin --from-literal=password=admin
```

You should see something similar to this:

```
[centos@ose-master1 openshift]$ oc create secret generic bigip-login -n kube-system --from-literal=username=admin
 --from-literal=password=admin
secret "bigip-login" created
[centos@ose-master1 openshift]$
```

5. Create kubernetes service account for bigip controller

```
oc create serviceaccount k8s-bigip-ctlr -n kube-system
```

You should see something similar to this:

```
[centos@ose-master1 openshift]$ oc create serviceaccount k8s-bigip-ctlr -n kube-system
serviceaccount "k8s-bigip-ctlr" created
[centos@ose-master1 openshift]$
```

6. Create cluster role for bigip service account (admin rights, but can be modified for your environment)

```
oc create clusterrolebinding k8s-bigip-ctlr-clusteradmin --clusterrole=cluster-admin --serviceaccount=kube-system:k8s-bigip-ctlr
```

You should see something similar to this:

```
[centos@ose-master1 openshift]$ oc create clusterrolebinding k8s-bigip-ctlr-clusteradmin --clusterrole=cluster-ad
min --serviceaccount=kube-system:k8s-bigip-ctlr
clusterrolebinding.rbac.authorization.k8s.io "k8s-bigip-ctlr-clusteradmin" created
[centos@ose-master1 openshift]$
```

7. Next let's explore the f5-hostsubnet.yaml file

```
cd /root/agilitydocs/openshift

cat f5-bigip-hostsubnet.yaml
```

You'll see a config file similar to this:

```
apiVersion: v1
kind: HostSubnet
metadata:
  name: openshift-f5-node
  annotations:
    pod.network.openshift.io/fixed-vnid-host: "0"
host: openshift-f5-node
hostIP: 10.3.10.60
subnet: "10.131.0.0/23"
```

> **Attention:** This YAML file creates an OpenShift Node and the Host is the BIG-IP with an assigned /23 subnet of IP 10.131.0.0 (3 imagas down).

8. Next let's look at the current cluster, you should see 3 members (1 master, 2 nodes)

```
oc get hostsubnet
```

```
[centos@ose-master1 openshift]$ oc get hostsubnet
NAME           HOST           HOST IP       SUBNET          EGRESS IPS
ose-master1    ose-master1    10.3.10.21    10.128.0.0/23   []
ose-node1      ose-node1      10.3.10.22    10.130.0.0/23   []
ose-node2      ose-node2      10.3.10.23    10.129.0.0/23   []
[centos@ose-master1 openshift]$
```

9. Now create the connector to the BIG-IP device, then look before and after at the attached devices

```
oc create -f f5-bigip-hostsubnet.yaml
```

You should see a successful creation of a new OpenShift Node.

```
[centos@ose-master1 openshift]$ oc create -f f5-bigip-hostsubnet.yaml
hostsubnet.network.openshift.io "openshift-f5-node" created
[centos@ose-master1 openshift]$
```

10. At this point nothing has been done to the BIG-IP, this only was done in the OpenShift environment.

```
oc get hostsubnet
```

You should now see OpenShift configured to communicate with the BIG-IP

```
[centos@ose-master1 openshift]$ oc get hostsubnet
NAME                 HOST                 HOST IP       SUBNET          EGRESS IPS
openshift-f5-node    openshift-f5-node    10.3.10.60    10.131.0.0/23   []
ose-master1          ose-master1          10.3.10.21    10.128.0.0/23   []
ose-node1            ose-node1            10.3.10.22    10.130.0.0/23   []
ose-node2            ose-node2            10.3.10.23    10.129.0.0/23   []
[centos@ose-master1 openshift]$
```

Important: The Subnet assignment, in this case is 10.131.0.0/23, was assigned by the **subnet: "10.131.0.0/23"** line in "HostSubnet" yaml file.

Note: In this lab we're manually assigning a subnet. We have the option to let openshift auto assign ths by removing **subnet: "10.131.0.0/23"** line at the end of the "hostsubnet" yaml file and setting the

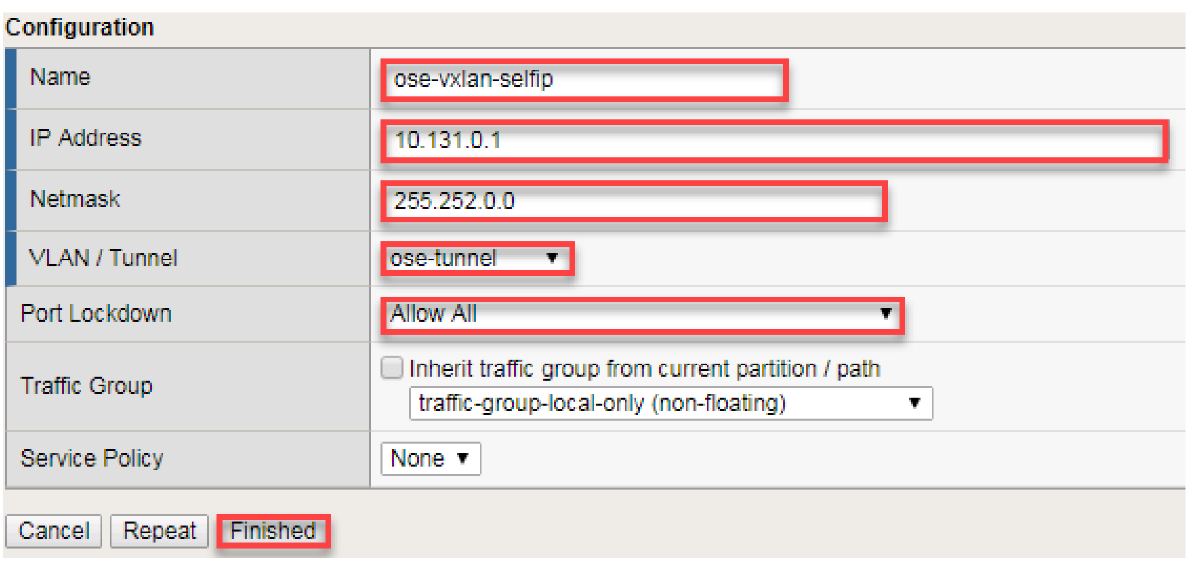
assign-subnet: "true". It would look like this:

```
apiVersion: v1
kind: HostSubnet
metadata:
   name: openshift-f5-node
   annotations:
      pod.network.openshift.io/fixed-vnid-host: "0"
      pod.network.openshift.io/assign-subnet: "true"
host: openshift-f5-node
hostIP: 10.3.10.60
```

11. Create the vxlan tunnel self-ip

Tip: For your SELF-IP subnet, remember it is a /14 and not a /23 - Why? The Self-IP has to be able to understand those other /23 subnets are local in the namespace in the example above for Master, Node1, Node2, etc. Many students accidently use /23, but then the self-ip will be only to communicate to one subnet on the openshift-f5-node. When trying to ping across to services on other /23 subnets from the BIG-IP for instance, communication will fail as your self-ip doesn't have the proper subnet mask to know those other subnets are local.

```
# From the CLI:
tmsh create net self ose-vxlan-selfip address 10.131.0.1/14 vlan ose-tunnel

# From the UI:
GoTo Network --> Self IP List
- Create a new Self-IP called "ose-vxlan-selfip"
- Set the IP Address to "10.131.0.1". (An IP from the subnet assigned in the
↪previous step.)
- Set the Netmask to "255.252.0.0"
- Set the VLAN / Tunnel to "ose-tunnel" (Created earlier)
- Set Port Lockdown to "Allow All"
- Click Finished
```



12. Now we'll create an Openshift F5 Container Connector to do the API calls to/from the F5 device. First we need the "deployment" file.

```
cd /root/agilitydocs/openshift

cat f5-cluster-deployment.yaml
```

You'll see a config file similar to this:

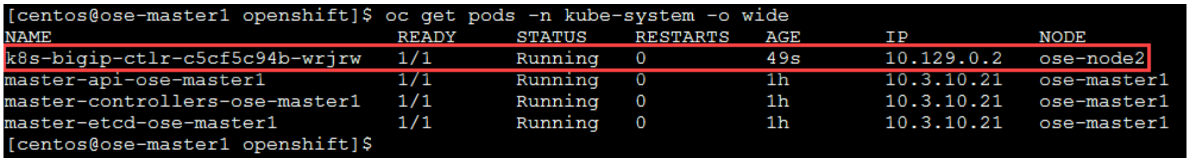
```yaml
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: k8s-bigip-ctlr
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      name: k8s-bigip-ctlr
      labels:
        app: k8s-bigip-ctlr
    spec:
      serviceAccountName: k8s-bigip-ctlr
      containers:
        - name: k8s-bigip-ctlr
          image: "f5networks/k8s-bigip-ctlr:latest"
          imagePullPolicy: IfNotPresent
          env:
            - name: BIGIP_USERNAME
              valueFrom:
                secretKeyRef:
                  name: bigip-login
                  key: username
            - name: BIGIP_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: bigip-login
                  key: password
          command: ["/app/bin/k8s-bigip-ctlr"]
          args: [
            "--bigip-username=$(BIGIP_USERNAME)",
            "--bigip-password=$(BIGIP_PASSWORD)",
            "--bigip-url=10.3.10.60",
            "--bigip-partition=ose",
            "--namespace=default",
            "--pool-member-type=cluster",
            "--openshift-sdn-name=/Common/ose-tunnel"
          ]
```

13. Create the container connector deployment with the following command

```
oc create -f f5-cluster-deployment.yaml
```

14. Check for successful creation:

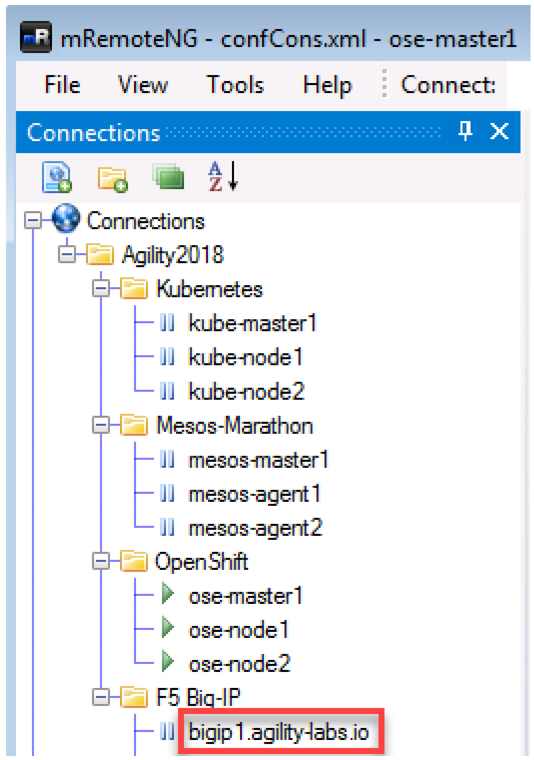
```
oc get pods -n kube-system -o wide
```



```
[centos@ose-master1 openshift]$ oc get pods -n kube-system -o wide
NAME                             READY     STATUS    RESTARTS   AGE       IP            NODE
k8s-bigip-ctlr-c5cf5c94b-wrjrw   1/1       Running   0          49s       10.129.0.2    ose-node2
master-api-ose-master1           1/1       Running   0          1h        10.3.10.21    ose-master1
master-controllers-ose-master1   1/1       Running   0          1h        10.3.10.21    ose-master1
master-etcd-ose-master1          1/1       Running   0          1h        10.3.10.21    ose-master1
[centos@ose-master1 openshift]$
```

15. If the tunnel is up and running big-ip should be able to ping the cluster nodes. SSH to big-ip and run one or all of the following ping tests.

Hint: To SSH to big-ip use mRemoteNG and the bigip1 shortcut




```
# ping ose-master
ping 10.128.0.1

# ping ose-node1
ping 10.129.0.1

# ping ose-node2
ping 10.130.0.1
```

5.2.2 Lab 2.2 - F5 Container Connector Usage

Now that our container connector is up and running, let's deploy an application and leverage our F5 CC.

For this lab we'll use a simple pre-configured docker image called "f5-hello-world". It can be found on docker hub at f5devcentral/f5-hello-world

To deploy our application, we will need to do the following:

1. Define a Deployment: this will launch our application running in a container.

2. Define a ConfigMap: this can be used to store fine-grained information like individual properties or coarse-grained information like entire config files or JSON blobs. It will contain the BIG-IP configuration we need to push.
3. Define a Service: this is an abstraction which defines a logical set of *pods* and a policy by which to access them. Expose the *service* on a port on each node of the cluster (the same port on each *node*). You'll be able to contact the service on any <NodeIP>:NodePort address. If you set the type field to "NodePort", the Kubernetes master will allocate a port from a flag-configured range **(default: 30000-32767)**, and each Node will proxy that port (the same port number on every Node) into your *Service*.

App Deployment

On the **ose-master** we will create all the required files:

1. Create a file called `f5-hello-world-deployment.yaml`

Tip: Use the file in /root/f5-agility-labs-containers/openshift

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: f5-hello-world
spec:
  replicas: 2
  template:
    metadata:
      labels:
        run: f5-hello-world
    spec:
      containers:
      - name: f5-hello-world
        image: "f5devcentral/f5-hello-world:develop"
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
          protocol: TCP
```

2. Create a file called `f5-hello-world-configmap.yaml`

Tip: Use the file in /root/f5-agility-labs-containers/openshift

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: f5-hello-world
  namespace: default
  labels:
    f5type: virtual-server
data:
  schema: "f5schemadb://bigip-virtual-server_v0.1.7.json"
  data: |
    {
      "virtualServer": {
```

```
        "frontend": {
          "balance": "round-robin",
          "mode": "http",
          "partition": "ose",
          "virtualAddress": {
            "bindAddr": "10.3.10.81",
            "port": 80
          }
        },
        "backend": {
          "serviceName": "f5-hello-world",
          "servicePort": 8080,
          "healthMonitors": [{
            "interval": 5,
            "protocol": "http",
            "send": "HEAD / HTTP/1.0\r\n\r\n",
            "timeout": 16
          }]
        }
      }
    }
```

3. Create a file called `f5-hello-world-service.yaml`

Tip: Use the file in /root/f5-agility-labs-containers/openshift

```
apiVersion: v1
kind: Service
metadata:
  name: f5-hello-world
  labels:
    run: f5-hello-world
spec:
  ports:
  - port: 8080
    protocol: TCP
    targetPort: 8080
  type: ClusterIP
  selector:
    run: f5-hello-world
```

4. We can now launch our application:

```
oc create -f f5-hello-world-deployment.yaml
oc create -f f5-hello-world-configmap.yaml
oc create -f f5-hello-world-service.yaml
```

```
[centos@ose-master1 openshift]$ oc create -f f5-hello-world-deployment.yaml
deployment.extensions "f5-hello-world" created
[centos@ose-master1 openshift]$ oc create -f f5-hello-world-configmap.yaml
configmap "f5-hello-world" created
[centos@ose-master1 openshift]$ oc create -f f5-hello-world-service.yaml
service "f5-hello-world" created
[centos@ose-master1 openshift]$
```

5. To check the status of our deployment, you can run the following commands:

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

```
[centos@ose-master1 openshift]$ oc get pods -o wide
NAME                              READY     STATUS    RESTARTS   AGE       IP            NODE
docker-registry-1-deploy          0/1       Pending   0          52m       <none>        <none>
f5-hello-world-5d76b7999b-6pcnz   1/1       Running   0          1m        10.129.0.3    ose-node2
f5-hello-world-5d76b7999b-f6mwc   1/1       Running   0          1m        10.130.0.2    ose-node1
registry-console-1-dmdws          1/1       Running   0          50m       10.128.0.3    ose-master1
router-1-deploy                   0/1       Pending   0          54m       <none>        <none>
[centos@ose-master1 openshift]$
```

```
oc describe svc f5-hello-world
```

```
[centos@ose-master1 openshift]$ oc describe svc f5-hello-world
Name:              f5-hello-world
Namespace:         default
Labels:            run=f5-hello-world
Annotations:       <none>
Selector:          run=f5-hello-world
Type:              ClusterIP
IP:                172.30.176.105
Port:              <unset>  8080/TCP
TargetPort:        8080/TCP
Endpoints:         10.129.0.3:8080,10.130.0.2:8080
Session Affinity:  None
Events:            <none>
[centos@ose-master1 openshift]$
```

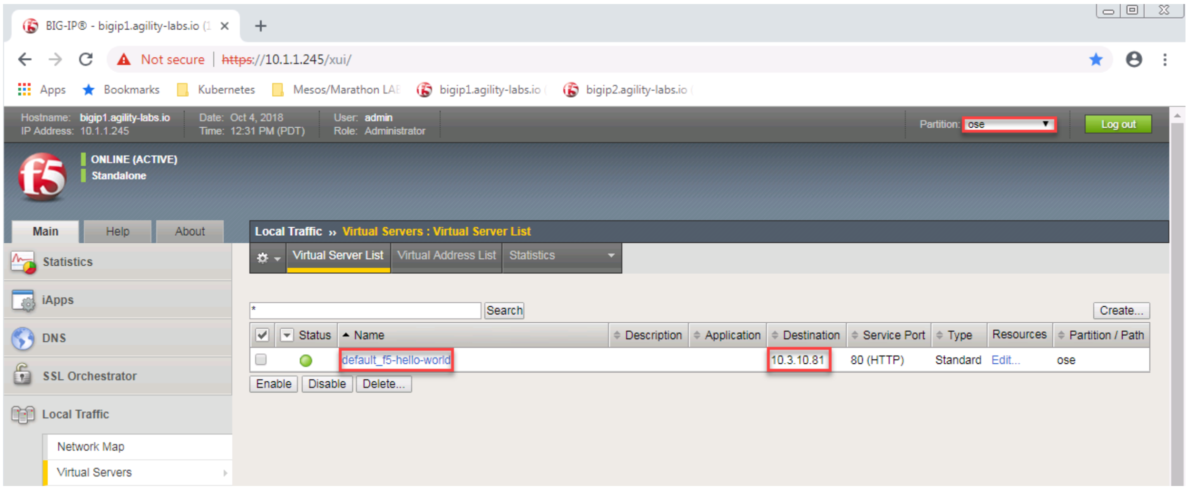
6. To test the app you need to pay attention to:

   The Endpoints, that's our 2 instances (defined as replicas in our deployment file) and the port assigned to the service: port 8080.

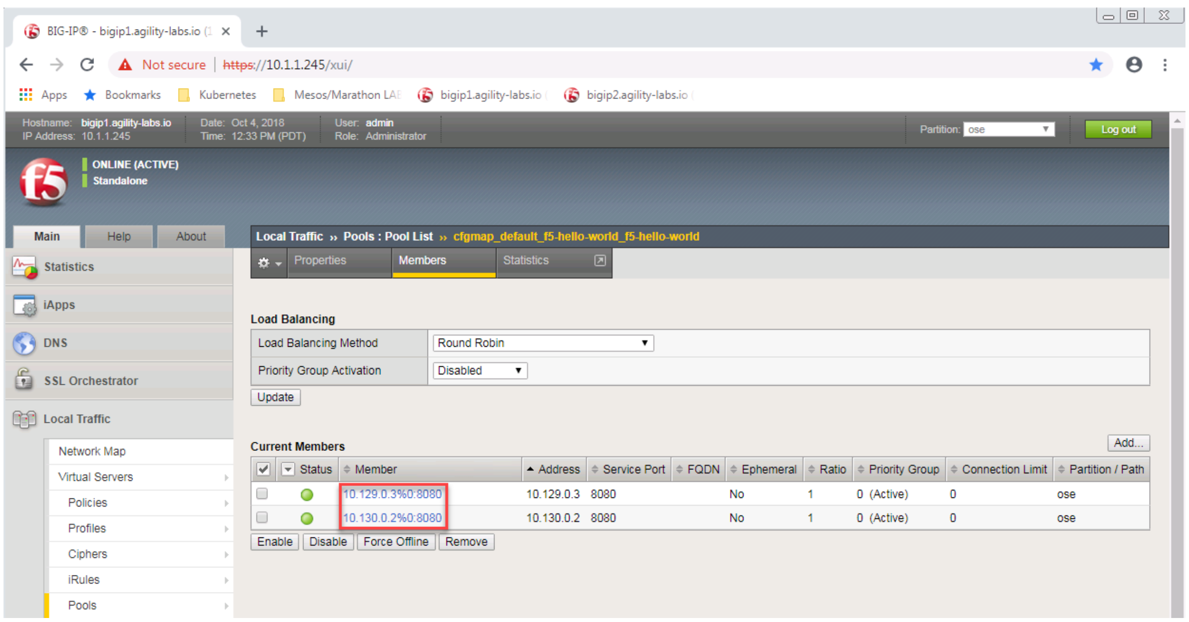
   Now that we have deployed our application sucessfully, we can check our BIG-IP configuration. From the browser open https://10.1.1.245

   > **Warning:** Don't forget to select the "ose" partition or you'll see nothing.

   Here you can see a new Virtual Server, "default_f5-hello-world" was created, listening on 10.3.10.81 in partition "ose".



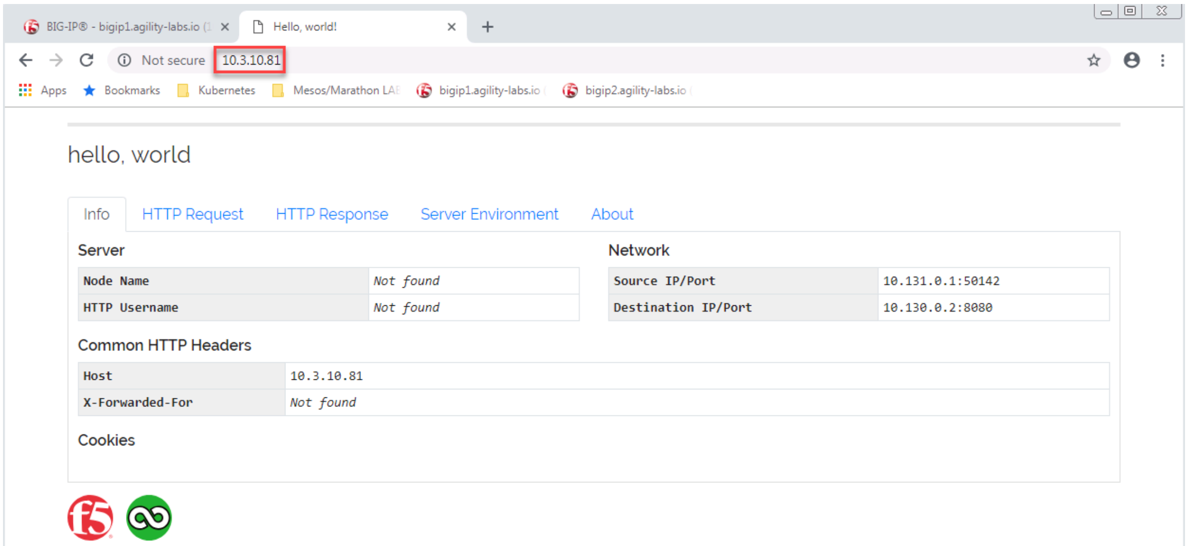
Check the Pools to see a new pool and the associated pool members: Local Traffic –> Pools –> “cfgmap_default_f5-hello-world_f5-hello-world” –> Members



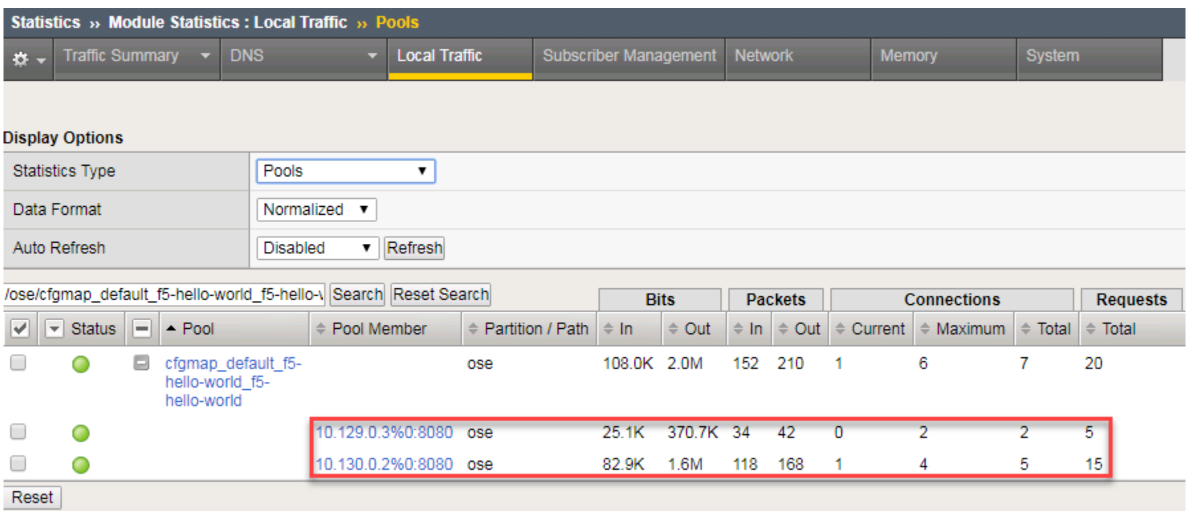
Note: You can see that the pool members IP addresses are assigned from the overlay network (**ClusterIP mode**)

---

7. Now access your application via the BIG-IP VIP: 10.3.10.81



8. Hit Refresh many times and go back to your **BIG-IP** UI, go to Local Traffic –> Pools –> Pool list –> cfgmap_default_f5-hello-world_f5-hello-world –> Statistics to see that traffic is distributed as expected.



9. Scale the f5-hello-world app

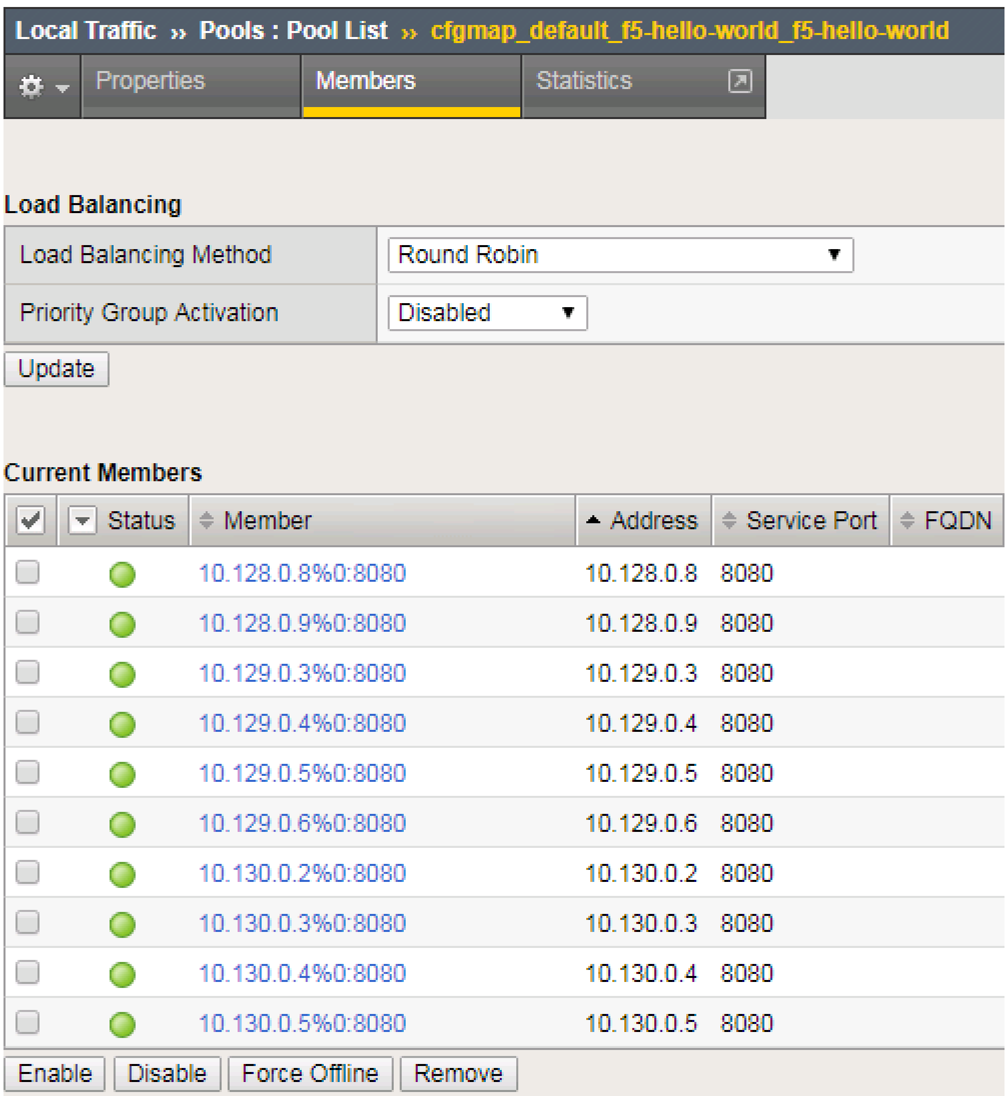
```
oc scale --replicas=10 deployment/f5-hello-world
```

10. Check the pods were created

```
oc get pods
```

```
[centos@ose-master1 openshift]$ oc get pods
NAME                               READY     STATUS    RESTARTS   AGE
docker-registry-1-deploy           0/1       Pending   0          1h
f5-hello-world-5d76b7999b-4gnrz    1/1       Running   0          43s
f5-hello-world-5d76b7999b-4gr4z    1/1       Running   0          43s
f5-hello-world-5d76b7999b-6pcnz    1/1       Running   0          13m
f5-hello-world-5d76b7999b-d6crr    1/1       Running   0          43s
f5-hello-world-5d76b7999b-drmrw    1/1       Running   0          43s
f5-hello-world-5d76b7999b-f6mwc    1/1       Running   0          13m
f5-hello-world-5d76b7999b-j7dcr    1/1       Running   0          43s
f5-hello-world-5d76b7999b-n8zvg    1/1       Running   0          43s
f5-hello-world-5d76b7999b-slsvl    1/1       Running   0          43s
f5-hello-world-5d76b7999b-x244m    1/1       Running   0          43s
registry-console-1-dmdws           1/1       Running   0          1h
router-1-deploy                    0/1       Pending   0          1h
[centos@ose-master1 openshift]$
```

11. Check the pool was updated on big-ip



Attention: Which network(s) are the IPs allocated from?

Expected time to complete: **30 minutes**

Attention: MODULE 1: BUILD AN OPENSHIFT CLUSTER CAN BE SKIPPED. THE BLUEPRINT IS PRE-CONFIGURED WITH A WORKING CLUSTER. THIS MODULE IS FOR DOCUMENTION PURPOSES ONLY.

5.3 Lab Setup

We will leverage the following setup to configure the OpenShift environment.

Hostname	IP-ADDR	Credentials
jumpbox	10.1.1.250	user/Student!Agility!
bigip1	10.1.1.245 10.3.10.60	admin/admin root/default
ose-master1	10.3.10.21	centos/centos root/default
ose-node1	10.3.10.22	centos/centos root/default
ose-node2	10.3.10.23	centos/centos root/default

6 Class 5: Advanced Labs for Red Hat OpenShift Container Platform (OCP)

The purpose of this lab is to give you more visibility on

6.1 Module 1: Welcome to OpenShift!

This lab guide is the F5 Advanced Labs for Red Hat OpenShift Container Platform (OCP). This lab guide and blueprint was created using OCP version 3.7. This lab provides a quick tour of the console to help you get familiar with the user interface along with some key terminology we will use in subsequent lab content.

6.1.1 Key Terms

We will be using the following terms throughout the workshop labs so here are some basic definitions you should be familiar with. And you'll learn more terms along the way, but these are the basics to get you started.

- Container - Your software wrapped in a complete filesystem containing everything it needs to run
- Image - We are talking about Docker images; read-only and used to create containers
- Pod - One or more docker containers that run together
- Service - Provides a common DNS name to access a pod (or replicated set of pods)
- Project - A project is a group of services that are related logically (for this workshop we have setup your account to have access to just a single project)
- Deployment - an update to your application triggered by a image change or config change
- Build - The process of turning your source code into a runnable image
- BuildConfig - configuration data that determines how to manage your build
- Route - a labeled and DNS mapped network path to a service from outside OpenShift
- Master - The foreman of the OpenShift architecture, the master schedules operations, watches for problems, and orchestrates everything
- Node - Where the compute happens, your software is run on nodes

Step 1: Access the Win7 Jump box

Use the following username and password:

- username: **user**
- password: **Student!Agility!**

Note: Use the Send Text to Client option to paste the password.

- We are using RHEL in this blueprint
- We updated on all the nodes (ose-node1, ose-node2) the /etc/hosts file so that each node is reachable via its name

```
[root@ose-node01 ~]# cat /etc/hosts
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
10.10.199.100 ose-mstr01 ose-mstr01.f5.local
10.10.199.101 ose-node01 ose-node01.f5.local
10.10.199.102 ose-node02 ose-node02.f5.local
```

- On ose-mstr01, we created some ssh keys for user that we copied on all the nodes. This way you can use ose-mstr01 as needed to connect to all nodes without authentication if wanting to jump around using ssh i.e. SSH root@10.10.199.101 from ose-mstr01

Step 2: Access the master using the mRemoteNG client from the Win7 Jump box (there is a shortcut in the taskbar). In the nRemoteNG client, Expand **Connections > Agility2018 > OpenShiftenterprise > OSE-cluster**. Here, you'll have shortcuts to the different Openshift nodes (Master and nodes) and to your BIG-IPs.

- Master Mgmt IP: 10.10.199.100 **root/default**
- BIGIP01 – 10.10.200.98 **root/default admin/admin**
- BIGIP02 – 10.10.200.99 **root/default admin/admin**

6.1.2 Accessing OpenShift

OpenShift provides a web console that allow you to perform various tasks via a web browser. Additionally, you can utilize a command line tool to perform tasks. Let's get started by logging into both of these and checking the status of the platform.

Step 3: Login to OpenShift master

Open a terminal on the master (click on **ose-master** in the mRemoteNG client) and login using the same URI/user/password with following command:

```
oc login https://ose-mstr01.f5.local:8443 --insecure-skip-tls-verify=true
```

Use the following username and password username: **demouser** password: **demouser**

```
[root@ose-mstr01 ~]# oc login https://ose-mstr01.f5.local:8443 --insecure-skip-tls-
↪verify=true
Authentication required for https://ose-mstr01.f5.local:8443 (openshift)
Username: demouser
Password:
Login successful.

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

```
   default
   f5demo
   guestbook
   kube-public
   kube-service-catalog
   * kube-system
   logging
   management-infra
   openshift
   openshift-infra
   openshift-node
   openshift-template-service-broker
   yelb

Using project "kube-system".
[root@ose-mstr01 ~]#
```

Step 4: Check the OpenShift status

The **oc status** command shows a high level overview of the project currently in use, with its components and their relationships, as shown in the following example:

```
[root@ose-mstr01 ~]# oc status
In project kube-system on server https://ose-mstr01.f5.local:8443

You have no services, deployment configs, or build configs.
Run 'oc new-app' to create an application.
[root@ose-mstr01 ~]#
```

Step 5: Check the OpenShift nodes

You can manage nodes in your instance using the CLI. The CLI interacts with node objects that are representations of actual node hosts. The master uses the information from node objects to validate nodes with health checks.

To list all nodes that are known to the master:

```
[root@ose-mstr01 ~]# oc get nodes
NAME                  STATUS                     AGE       VERSION
ose-mstr01.f5.local   Ready,SchedulingDisabled   24d       v1.7.6+a08f5eeb62
ose-node01            Ready                      24d       v1.7.6+a08f5eeb62
ose-node02            Ready                      24d       v1.7.6+a08f5eeb62
[root@ose-mstr01 ~]#
```

If the **node** status shows **NotReady** or **SchedulingDisabled** contact the lab proctor. The node is not passing the health checks performed from the master and Pods cannot be scheduled for placement on the node.

Note: “SchedulingDisabled” for the **Master** is normal.

To get more detailed information about a specific node, including the reason for the current condition use the oc describe node command. This does provide alot of very useful information and can assist with throubleshooting issues.

```
[root@ose-mstr01 ~]# oc describe node ose-mstr01.f5.local
Name:                   ose-mstr01.f5.local
```

```
Role:
Labels:                 beta.kubernetes.io/arch=amd64
                        beta.kubernetes.io/os=linux
                        kubernetes.io/hostname=ose-mstr01.f5.local
                        openshift-infra=apiserver
Annotations:            volumes.kubernetes.io/controller-managed-attach-detach=true
Taints:                 <none>
CreationTimestamp:      Fri, 22 Jun 2018 15:53:34 -0700
Conditions:
Type                   Status  LastHeartbeatTime                                      ␣
 ↪LastTransitionTime                     Reason                              Message
----                   ------  -----------------                                ----------------
 ↪--                       ------                                 -------
OutOfDisk              False   Tue, 17 Jul 2018 12:08:16 -0700          Fri, 22 Jun␣
 ↪2018 15:53:34 -0700              KubeletHasSufficientDisk        kubelet has␣
 ↪sufficient disk space available
MemoryPressure         False   Tue, 17 Jul 2018 12:08:16 -0700          Fri, 22 Jun␣
 ↪2018 15:53:34 -0700          KubeletHasSufficientMemory      kubelet has sufficient␣
 ↪memory available
DiskPressure           False   Tue, 17 Jul 2018 12:08:16 -0700          Fri, 22 Jun␣
 ↪2018 15:53:34 -0700          KubeletHasNoDiskPressure        kubelet has no disk␣
 ↪pressure
Ready                  True    Tue, 17 Jul 2018 12:08:16 -0700          Tue, 17 Jul␣
 ↪2018 11:07:28 -0700            KubeletReady                        kubelet is posting␣
 ↪ready status
Addresses:
InternalIP:   10.10.199.100
Hostname:     ose-mstr01.f5.local
Capacity:
cpu:           4
memory:        16266916Ki
pods:          40
Allocatable:
cpu:           4
memory:        16164516Ki
pods:          40
System Info:
Machine ID:                    8bd4148d1a6249a7bca6e753d64862b3
System UUID:                   564DADCC-A795-99FC-F2EA-24AFEAD600C3
Boot ID:                       16b282b5-5ee0-4e1a-be6a-b8e1e2ae2449
Kernel Version:                3.10.0-862.3.3.el7.x86_64
OS Image:                      OpenShift Enterprise
Operating System:              linux
Architecture:                  amd64
Container Runtime Version:     docker://1.13.1
Kubelet Version:               v1.7.6+a08f5eeb62
Kube-Proxy Version:            v1.7.6+a08f5eeb62
ExternalID:                     ose-mstr01.f5.local
Non-terminated Pods:            (2 in total)
Namespace                      Name                                    CPU Requests   ␣
 ↪CPU Limits      Memory Requests       Memory Limits
---------                      ----                                    ------------   ␣
 ↪----------      ---------------       -------------
kube-service-catalog           apiserver-56t4l                         0 (0%)         ␣
 ↪0 (0%)          0 (0%)                0 (0%)
kube-service-catalog           controller-manager-m2mbt                0 (0%)         ␣
 ↪0 (0%)          0 (0%)                0 (0%)
Allocated resources:
```

```
(Total limits may be over 100 percent, i.e., overcommitted.)
CPU Requests  CPU Limits      Memory Requests Memory Limits
------------  ----------      --------------- -------------
0 (0%)        0 (0%)          0 (0%)          0 (0%)
Events:
FirstSeen     LastSeen        Count   From                            SubObjectPath  ␣
↪Type             Reason              Message
---------     --------        -----   ----                            -------------  ␣
↪--------         ------              -------
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal               NodeAllocatableEnforced Updated Node Allocatable limit across␣
↪pods
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal           Starting                Starting kubelet.
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal           NodeHasSufficientDisk   Node ose-mstr01.f5.local status is now:␣
↪NodeHasSufficientDisk
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal           NodeHasSufficientMemory Node ose-mstr01.f5.local status is now:␣
↪NodeHasSufficientMemory
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal           NodeHasNoDiskPressure   Node ose-mstr01.f5.local status is now:␣
↪NodeHasNoDiskPressure
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Warning          Rebooted                Node ose-mstr01.f5.local has been rebooted,␣
↪boot id: 16b282b5-5ee0-4e1a-be6a-b8e1e2ae2449
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal           NodeNotReady            Node ose-mstr01.f5.local status is now:␣
↪NodeNotReady
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal           NodeNotSchedulable      Node ose-mstr01.f5.local status is now:␣
↪NodeNotSchedulable
1h            1h              1       kubelet, ose-mstr01.f5.local                   ␣
↪Normal           NodeReady               Node ose-mstr01.f5.local status is now:␣
↪NodeReady
[root@ose-mstr01 ~]#
```

Step 6: Check to see what projects you have access to:

```
[root@ose-mstr01 ~]# oc get projects
NAME                                DISPLAY NAME   STATUS
default                                            Active
f5demo                                             Active
guestbook                                          Active
kube-public                                        Active
kube-service-catalog                               Active
kube-system                                        Active
logging                                            Active
management-infra                                   Active
openshift                                          Active
openshift-infra                                    Active
openshift-node                                     Active
openshift-template-service-broker                  Active
yelb                                               Active
```

You will be using these projects in the lab

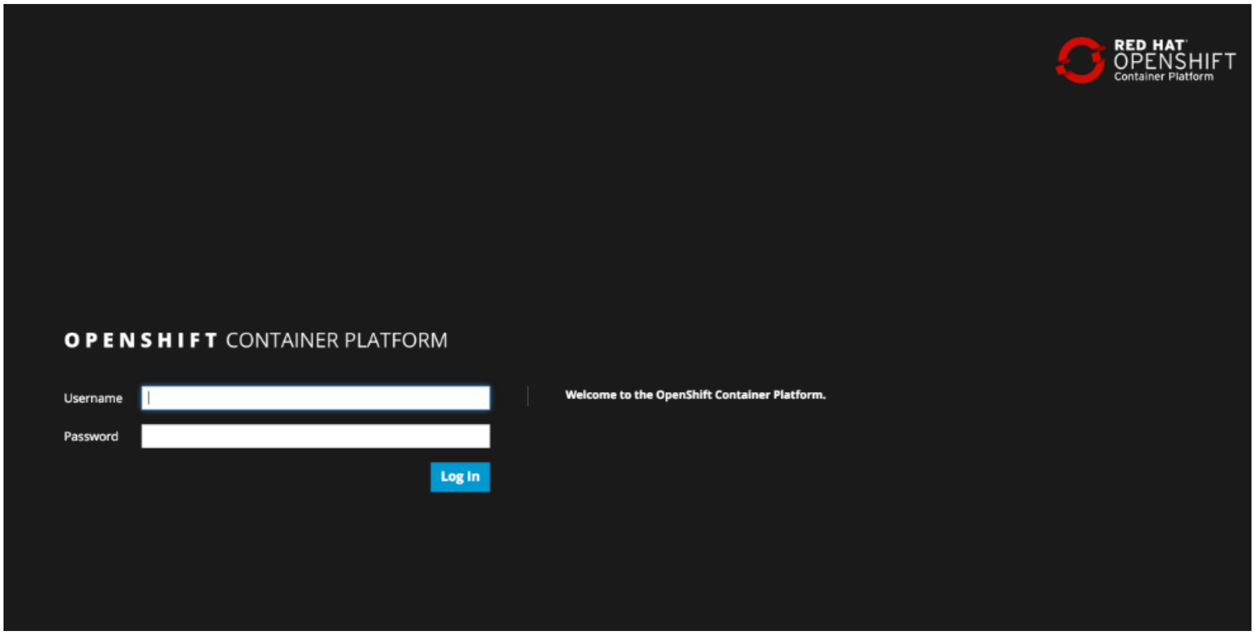
Step 7: Check to see what host subnets are created on OpenShift:

```
[root@ose-mstr01 ~]# oc get hostsubnets
NAME                 HOST                 HOST IP         SUBNET          EGRESS IPS
ose-mstr01.f5.local  ose-mstr01.f5.local  10.10.199.100   10.130.0.0/23   []
ose-node01           ose-node01           10.10.199.101   10.128.0.0/23   []
ose-node02           ose-node02           10.10.199.102   10.129.0.0/23   []
[root@ose-mstr01 ~]#
```

Step 8: Access OpenShift web console

From the jumpbox navigate to the URI provided by your instructor and login with the user/password provided (there is a favorite on **chrome** called **Login - OpenShift Container Platform**).

Use the following username and password username: **demouser** password: **demouser**



6.1.3 Troubleshooting OpenShift!

If you have a problem in your OpenShift Container Platform 3 environment, how do you investigate

- How can I troubleshoot it?
- What logs can I inspect?
- How can I modify the log level / detail that openshift generates?
- I need to provide supporting data to technical support for analysis. What information is needed?

A starting point for data collection from an OpenShift master or node is a sosreport that includes docker and OpenShift related information. The process to collect a sosreport is the same as with any other Red Hat Enterprise Linux (RHEL) based system:

Note: The following is provided for informational purposes. You do not need to run these commands for the lab.

```
# yum update sos
# sosreport
```

Openshift has five log message severities. Messages with FATAL, ERROR, WARNING and some INFO severities appear in the logs regardless of the log configuration.

```
0 - Errors and warnings only
2 - Normal information
4 - Debugging-level information
6 - API-level debugging information (request / response)
8 - Body-level API debugging information
```

This parameter can be set in the OPTIONS for the relevant services environment file within /etc/sysconfig/

For example to set OpenShift master's log level to debug, add or edit this line in /etc/sysconfig/atomic-openshift-master

```
OPTIONS='--loglevel=4'

and then restart the service with

systemctl restart atomic-openshift-master
```

Key files / directories

```
/etc/origin/{node,master}/
/etc/origin/{node,master}/{node.master}-config.yaml
```

6.2 Module 2: Working with BIG-IP HA Pairs or Device Groups

Each Container Connector is uniquely suited to its specific container orchestration environment and purpose, utilizing the architecture and language appropriate for the environment. Application Developers interact with the platform's API; the CCs watch the API for certain events, then act accordingly.

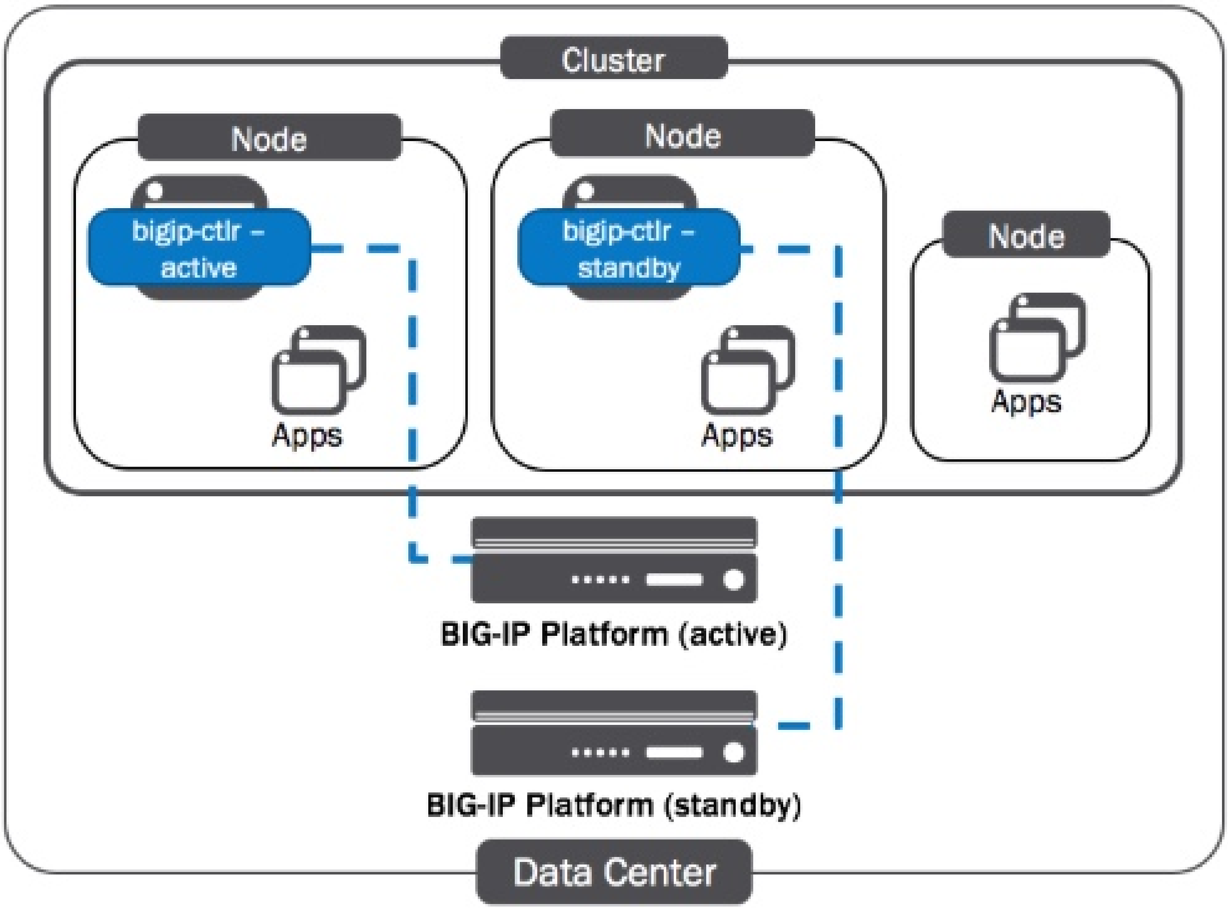
The Container Connector is stateless (Stateless means there is no record of previous interactions and each interaction request has to be handled based entirely on information that comes with it). The inputs are:

- the container orchestration environment's config
- the BIG-IP device config
- the CC config (provided via the appropriate means from the container orchestration environment).

Wherever a Container Connector runs, it always watches the API and attempts to bring the BIG-IP up-to-date with the latest applicable configurations.

6.2.1 Managing BIG-IP HA Clusters in OpenShift

You can use the F5 Container Connectors (also called F5 BIG-IP Controller) to manage a BIG-IP HA active-standby pair or device group. The deployment details vary depending on the platform. For most, the basic principle is the same: You should run one BIG-IP Controller instance for each BIG-IP device. You will deploy two BIG-IP Controller instances - one for each BIG-IP device. To help ensure Controller HA, you will deploy each Controller instance on a separate Node in the cluster.



6.2.2 BIG-IP Config Sync

Important: Each Container Connector monitors the BIG-IP partition it manages for configuration changes. If its configuration changes, the Connector reapplies its own configuration to the BIG-IP. F5 does not recommend making configuration changes to objects in any partition managed by a F5 Container Connector via any other means (for example, the configuration utility, TMOS, or by syncing configuration from another device or service group). Doing so may result in disruption of service or unexpected behavior.

The Container Connector for OpenShift uses FDB entries and ARP records to identify the Cluster resources associated with BIG-IP Nodes. Because BIG-IP config sync doesn't include FDB entries or ARP records, F5 does not recommend using automatic configuration sync when managing a BIG-IP HA pair or cluster with the F5 Container Connector. You must disable config sync when using tunnels.

Complete the steps below to set up the solution shown in the diagram. Be sure to use the correct IP addresses and subnet masks for your OpenShift Cluster

Step	Task
1.	*Initial BIG-IP HA Setup*
2.	*Upload the HostSubnet Files to the OpenShift API Server* • openshift create hostsubnets ha • openshift upload hostsubnets ha • openshift verify hostsubnets ha
3.	*Set up VXLAN on the BIG-IP Devices* • creating OCP partition create • ocp-profile create • openshift create vxlan profile ha • penshift create vxlan tunnel ha • openshift vxlan selfIP ha • openshift vxlan floatingip ha
4.	*Deploy the BIG-IP Controller (F5 Container Connector)* • openshift rbac ha • openshift create deployment ha • openshift upload deployment ha

6.2.3 Initial BIG-IP HA Setup

Step 1:

The purpose of this lab is not to cover BIG-IP High Availability (HA) in depth but focus on OpenShift configuration with BIG-IP. Some prior BIG-IP HA knowledge is required. We have created the BIG-IPs base configuration for bigip01 and bigip02 to save time. Below is the initial configuration used on each BIG-IP:

Note: The following is provided for informational purposes. You do not need to run these commands for the lab.

---

bigip01.f5.local

```
tmsh modify sys global-settings hostname bigip01.f5.local
tmsh modify sys global-settings mgmt-dhcp disabled
tmsh create sys management-ip 10.10.200.98/24
tmsh create sys management-route 10.10.200.1
tmsh create net vlan external interfaces add {1.1}
tmsh create net vlan internal interfaces add {1.2}
tmsh create net vlan ha interfaces add {1.3}
tmsh create net self 10.10.199.98/24 vlan internal
tmsh create net self 10.10.201.98/24 vlan external
tmsh create net self 10.10.202.98/24 vlan ha allow-service default
tmsh create net route default gw 10.10.201.1
tmsh mv cm device bigip1 bigip01.f5.local
tmsh modify cm device bigip01.f5.local configsync-ip 10.10.202.98
tmsh modify cm device bigip01.f5.local unicast-address {{ip 10.10.202.98} {ip␣
↪management-ip}}
```

```
tmsh modify cm trust-domain ca-devices add {10.10.200.99} username admin password␣
↪admin
tmsh create cm device-group ocp-devicegroup devices add {bigip01.f5.local bigip02.f5.
↪local} type sync-failover auto-sync disabled
tmsh run cm config-sync to-group ocp-devicegroup
tmsh save sys config
```

bigip02.f5.local

```
tmsh modify sys global-settings hostname bigip02.f5.local
tmsh modify sys global-settings mgmt-dhcp disabled
tmsh create sys management-ip 10.10.200.99/24
tmsh create sys management-route 10.10.200.1
tmsh create net vlan external interfaces add {1.1}
tmsh create net vlan internal interfaces add {1.2}
tmsh create net vlan ha interfaces add {1.3}
tmsh create net self 10.10.199.99/24 vlan internal
tmsh create net self 10.10.201.99/24 vlan external
tmsh create net self 10.10.202.99/24 vlan ha allow-service default
tmsh create net route default gw 10.10.201.1
tmsh modify sys global-settings gui-setup disabled
tmsh mv cm device bigip1 bigip02.f5.local
tmsh modify cm device bigip02.f5.local configsync-ip 10.10.202.99
tmsh modify cm device bigip02.f5.local unicast-address {{ip 10.10.202.99} {ip␣
↪management-ip}}
tmsh save sys config
```

Tip: Before adding the BIG-IP devices to OpenShift make sure your High Availability (HA) device trust group, license, selfIP, vlans are configured correctly.

Note: You have shortcuts to connect to your BIG-IPs in Chrome. Login: **admin**, Password: **admin**

Validate that SDN services license is active

Attention: In your lab environment the BIG-IP VE LAB license includes the SDN license. The following is provided as a reference of what you may see in a production license. The SDN license is also included in the -V16 version of the BIG-IP VE license.

General Properties	
License Type	Evaluation
Licensed Date	Jun 26, 2018
License Expiration Date	Aug 11, 2018
Active Modules	• Local Traffic Manager, VE-1G (FXYMBOM-KCSXPHY) ◦ LTM to Best Bundle Upgrade, 1Gbps ◦ Rate Shaping ◦ SDN Services, VE ◦ APM, Limited ◦ ASM, VE ◦ DNS-GTM, Base, 1Gbps ◦ SSL, VE ◦ Max Compression, VE ◦ AFM, VE ◦ APM, Base, VE GBB (500 CCU, 2500 Access Sessions) ◦ DNSSEC ◦ Anti-Virus Checks ◦ Base Endpoint Security Checks ◦ Firewall Checks ◦ Network Access ◦ Secure Virtual Keyboard ◦ APM, Web Application ◦ Machine Certificate Checks ◦ Protected Workspace ◦ Remote Desktop ◦ App Tunnel ◦ CGN, BIG-IP VE, AFM ONLY ◦ DNS Rate Limit, 1000 QPS ◦ GTM Rate, 1000 ◦ Routing Bundle, VE ◦ PSM, VE

Validate the vlan configuration

☑	▲ Name	◆ Application	◆ Tag	Untagged Interfaces	Tagged Interfaces	◆ Partition / Path
☐	external		4094	1.1		Common
☐	ha		4092	1.3		Common
☐	internal		4093	1.2		Common

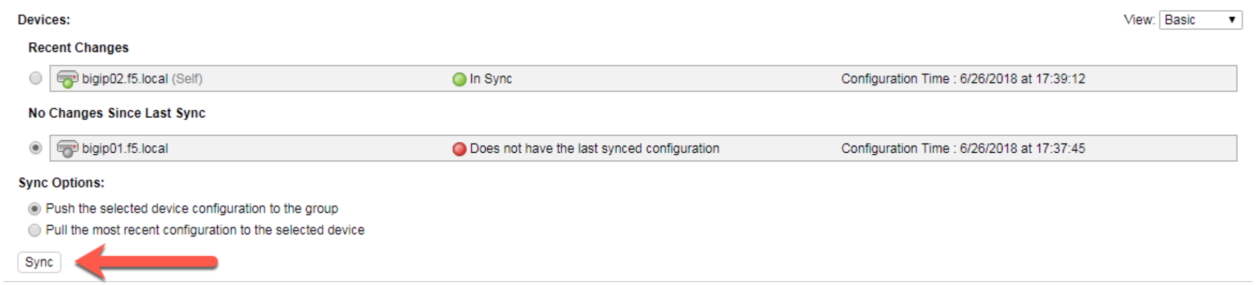
Validate bigip01 self IP configuration

☑	◆ Name	◆ Application	◆ IP Address	◆ Netmask	◆ VLAN / Tunnel	◆ Traffic Group	◆ Partition / Path
☐	10.10.199.98/24		10.10.199.98	255.255.255.0	internal	traffic-group-local-only	Common
☐	10.10.201.98/24		10.10.201.98	255.255.255.0	external	traffic-group-local-only	Common
☐	10.10.202.98/24		10.10.202.98	255.255.255.0	ha	traffic-group-local-only	Common

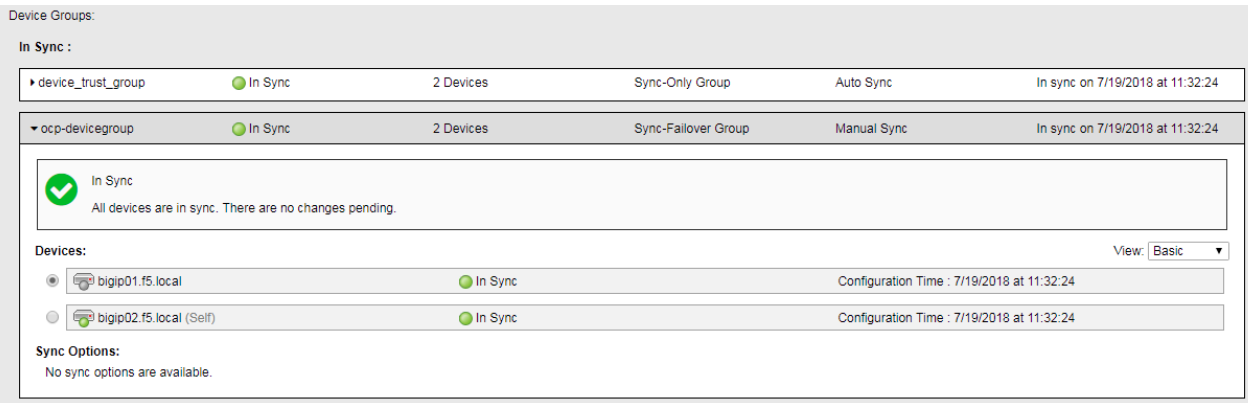
Validate bigip02 self IP configuration

☑	◆ Name	◆ Application	◆ IP Address	◆ Netmask	◆ VLAN / Tunnel	◆ Traffic Group	◆ Partition / Path
☐	10.10.199.99/24		10.10.199.99	255.255.255.0	internal	traffic-group-local-only	Common
☐	10.10.201.99/24		10.10.201.99	255.255.255.0	external	traffic-group-local-only	Common
☐	10.10.202.99/24		10.10.202.99	255.255.255.0	ha	traffic-group-local-only	Common

Validate the device group HA settings and make sure bigip01 and bigip02 are in sync. If out of sync, sync the bigip



All synced. Note the sync-failover configuration is set to manual sync



The diagram below displays the BIG-IP deployment with the OpenShift cluster in High Availability (HA) active-standby pair or device group. Note this solution applies to BIG-IP devices v13.x and later only. To accomplish High Availability (HA) active-standby pair or device group with OpenShift the BIG-IP needs to create a floating vxlan tunnel address with is currently only available in BIG-IP 13.x and later.

6.2.4 Upload the HostSubnet Files to the OpenShift API Server

Step 2: Create a new OpenShift HostSubnet

HostSubnets must use valid YAML. You can upload the files individually using separate oc create commands.

Create one HostSubnet for each BIG-IP device. These will handle health monitor traffic.

Also create one HostSubnet to pass client traffic. You will create the floating IP address for the active device in this subnet as shown in the diagram above.

> **Attention:** We have created the YAML files to save time. The files are located at **/root/agility2018/ocp** on the **master** (ose-master)
>
> cd /root/agility2018/ocp

6.2.5 Define HostSubnets

hs-bigip01.yaml

```
{
    "apiVersion": "v1",
    "host": "openshift-f5-bigip01",
```

```
    "hostIP": "10.10.199.98",
    "kind": "HostSubnet",
    "metadata": {
        "name": "openshift-f5-bigip01"
    },
    "subnet": "10.131.0.0/23"
}
```

hs-bigip02.yaml

```
{
    "apiVersion": "v1",
    "host": "openshift-f5-bigip02",
    "hostIP": "10.10.199.99",
    "kind": "HostSubnet",
    "metadata": {
        "name": "openshift-f5-bigip02"
    },
    "subnet": "10.131.2.0/23"
}
```

hs-bigip-float.yaml

```
{
    "apiVersion": "v1",
    "host": "openshift-f5-bigip-float",
    "hostIP": "10.10.199.200",
    "kind": "HostSubnet",
    "metadata": {
        "name": "openshift-f5-bigip-float"
    },
    "subnet": "10.131.4.0/23"
}
```

Create the HostSubnet files to the OpenShift API server. Run the following commands from the **master**

```
oc create -f hs-bigip01.yaml
oc create -f hs-bigip02.yaml
oc create -f hs-bigip-float.yaml
```

Verify creation of the HostSubnets:

```
[root@ose-mstr01 ocp]# oc get hostsubnet
NAME                        HOST                        HOST IP         SUBNET         ␣
↪EGRESS IPS
openshift-f5-bigip-float    openshift-f5-bigip-float    10.10.199.200   10.131.4.0/23  ␣
↪[]
openshift-f5-bigip01        openshift-f5-bigip01        10.10.199.98    10.131.0.0/23  ␣
↪[]
openshift-f5-bigip02        openshift-f5-bigip02        10.10.199.99    10.131.2.0/23  ␣
↪[]
ose-mstr01.f5.local         ose-mstr01.f5.local         10.10.199.100   10.130.0.0/23  ␣
↪[]
ose-node01                  ose-node01                  10.10.199.101   10.128.0.0/23  ␣
↪[]
ose-node02                  ose-node02                  10.10.199.102   10.129.0.0/23  ␣
↪[]
[root@ose-mstr01 ocp]#
```

6.2.6 Set up VXLAN on the BIG-IP Devices

Important: The BIG-IP OpenShift Controller cannot manage objects in the /Common partition.

Its recommended to create all HA using the /Common partition

Tip: You can copy and paste the following commands to be run directly from the OpenShift **master** (ose-mstr01). To paste content into mRemoteNG; use your right mouse button.

Step 3.1: Create a new partition on your BIG-IP system

- ssh root@10.10.200.98 tmsh create auth partition ocp
- ssh root@10.10.200.99 tmsh create auth partition ocp

Step 3.2: Creating ocp-profile

- ssh root@10.10.200.98 tmsh create net tunnels vxlan ocp-profile flooding-type multipoint
- ssh root@10.10.200.99 tmsh create net tunnels vxlan ocp-profile flooding-type multipoint

Step 3.3: Creating floating IP for underlay network

- ssh root@10.10.200.98 tmsh create net self 10.10.199.200/24 vlan internal traffic-group traffic-group-1
- ssh root@10.10.200.98 tmsh run cm config-sync to-group ocp-devicegroup

Step 3.4: Creating vxlan tunnel ocp-tunnel

- ssh root@10.10.200.98 tmsh create net tunnels tunnel ocp-tunnel key 0 profile ocp-profile local-address 10.10.199.200 secondary-address 10.10.199.98 traffic-group traffic-group-1
- ssh root@10.10.200.99 tmsh create net tunnels tunnel ocp-tunnel key 0 profile ocp-profile local-address 10.10.199.200 secondary-address 10.10.199.99 traffic-group traffic-group-1

Step 3.5: Creating overlay self-ip

- ssh root@10.10.200.98 tmsh create net self 10.131.0.98/14 vlan ocp-tunnel
- ssh root@10.10.200.99 tmsh create net self 10.131.2.99/14 vlan ocp-tunnel

Step 3.6: Creating floating IP for overlay network

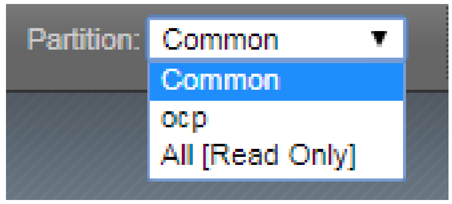
- ssh root@10.10.200.98 tmsh create net self 10.131.4.200/14 vlan ocp-tunnel traffic-group traffic-group-1
- ssh root@10.10.200.98 tmsh run cm config-sync to-group ocp-devicegroup

Step 3.7: Saving configuration

- ssh root@10.10.200.98 tmsh save sys config
- ssh root@10.10.200.99 tmsh save sys config

Before adding the BIG-IP controller to OpenShift validate the partition and tunnel configuration

Validate that the OCP bigip partition was created



Validate bigip01 self IP configuration

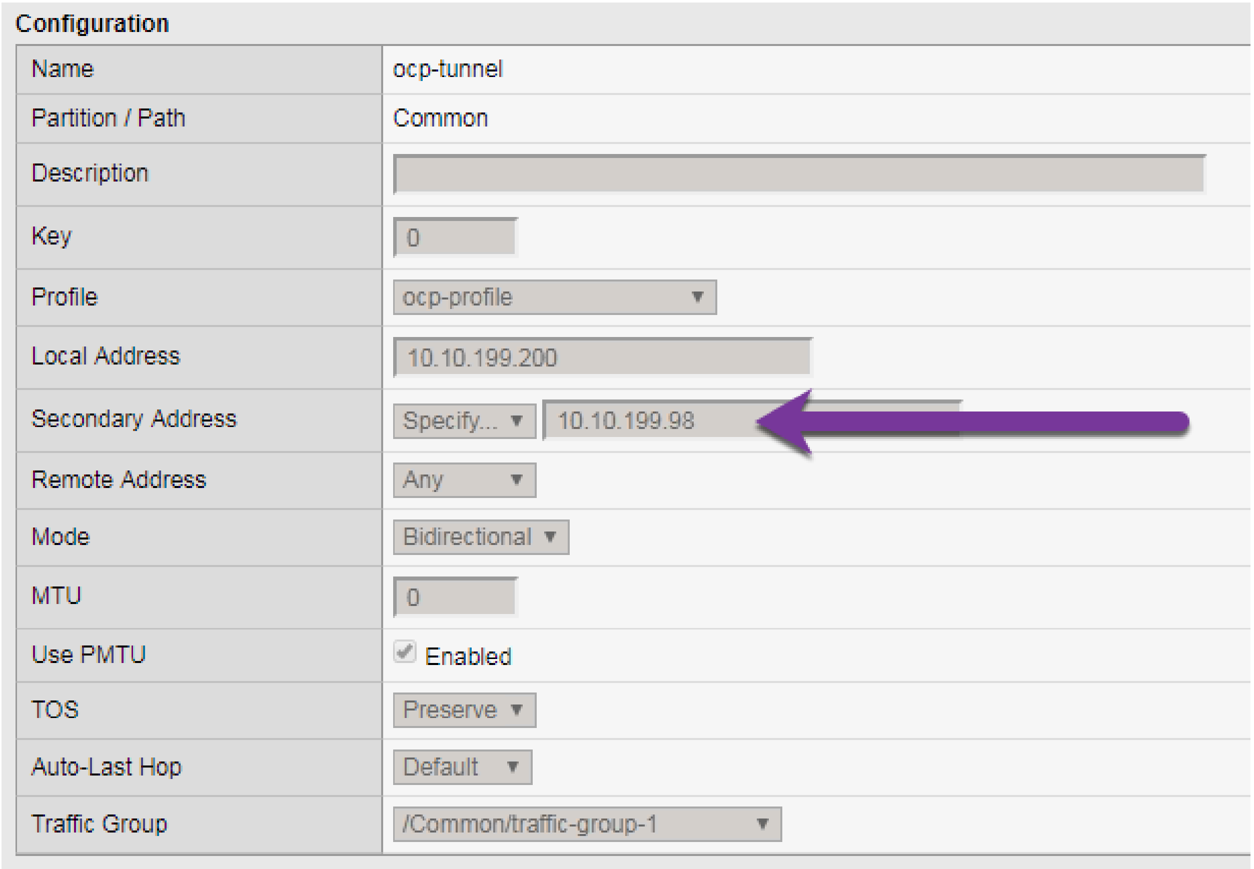
Note: On the active device, there is floating IP address in the subnet assigned by the OpenShift SDN.

Name	Application	IP Address	Netmask	VLAN / Tunnel	Traffic Group	Partition / Path
10.10.199.200/24		10.10.199.200	255.255.255.0	internal	traffic-group-1	Common
10.10.199.98/24		10.10.199.98	255.255.255.0	internal	traffic-group-local-only	Common
10.10.201.98/24		10.10.201.98	255.255.255.0	external	traffic-group-local-only	Common
10.10.202.98/24		10.10.202.98	255.255.255.0	ha	traffic-group-local-only	Common
10.131.0.98/14		10.131.0.98	255.252.0.0	ocp-tunnel	traffic-group-local-only	Common
10.131.4.200/14		10.131.4.200	255.252.0.0	ocp-tunnel	traffic-group-1	Common

Validate bigip02 self IP configuration

Name	Application	IP Address	Netmask	VLAN / Tunnel	Traffic Group	Partition / Path
10.10.199.200/24		10.10.199.200	255.255.255.0	internal	traffic-group-1	Common
10.10.199.99/24		10.10.199.99	255.255.255.0	internal	traffic-group-local-only	Common
10.10.201.99/24		10.10.201.99	255.255.255.0	external	traffic-group-local-only	Common
10.10.202.99/24		10.10.202.99	255.255.255.0	ha	traffic-group-local-only	Common
10.131.2.99/14		10.131.2.99	255.252.0.0	ocp-tunnel	traffic-group-local-only	Common
10.131.4.200/14		10.131.4.200	255.252.0.0	ocp-tunnel	traffic-group-1	Common

Check the ocp-tunnel configuration (under Network -> Tunnels). Note the local-address 10.10.199.200 and secondary-address are 10.10.199.98 for bigip01 and 10.10.199.99 for bigip02. The secondary-address will be used to send monitor traffic and the local address will be used by the active device to send client traffic.



6.2.7 Deploy the BIG-IP Controller (F5 Container Connector)

Take the steps below to deploy a contoller for each BIG-IP device in the cluster.

6.2.8 Set up RBAC

The F5 BIG-IP Controller requires permission to monitor the status of the OpenSfhift cluster. The following will create a "role" that will allow it to access specific resources.

You can create RBAC resources in the project in which you will run your BIG-IP Controller. Each Controller that manages a device in a cluster or active-standby pair can use the same Service Account, Cluster Role, and Cluster Role Binding.

Step 4.1: Create a Service Account for the BIG-IP Controller

```
[root@ose-mstr01 ocp]# oc create serviceaccount bigip-ctlr -n kube-system
serviceaccount "bigip-ctlr" created
```

Step 4.2: Create a Cluster Role and Cluster Role Binding with the required permissions.

The following file has already being created **f5-kctlr-openshift-clusterrole.yaml** which is located in **/root/agility2018/ocp** on the **master**

```
# For use in OpenShift clusters
apiVersion: v1
kind: ClusterRole
```

```
metadata:
  annotations:
    authorization.openshift.io/system-only: "true"
  name: system:bigip-ctlr
rules:
- apiGroups: ["", "extensions"]
  resources: ["nodes", "services", "endpoints", "namespaces", "ingresses", "routes" ]
  verbs: ["get", "list", "watch"]
- apiGroups: ["", "extensions"]
  resources: ["configmaps", "events", "ingresses/status"]
  verbs: ["get", "list", "watch", "update", "create", "patch" ]
- apiGroups: ["", "extensions"]
  resources: ["secrets"]
  resourceNames: ["<secret-containing-bigip-login>"]
  verbs: ["get", "list", "watch"]

---

apiVersion: v1
kind: ClusterRoleBinding
metadata:
    name: bigip-ctlr-role
userNames:
- system:serviceaccount:kube-system:bigip-ctlr
subjects:
- kind: ServiceAccount
  name: bigip-ctlr
roleRef:
  name: system:bigip-ctlr
```

```
[root@ose-mstr01 ocp]# oc create -f f5-kctlr-openshift-clusterrole.yaml
clusterrole "system:bigip-ctlr" created
clusterrolebinding "bigip-ctlr-role" created
```

6.2.9 Create Deployments

Step 4.3: Deploy the BIG-IP Controller

Create an OpenShift Deployment for each Controller (one per BIG-IP device). You need to deploy a controller for both f5-bigip-node01 and f5-bigip-node02

- Provide a unique metadata.name for each Controller.
- Provide a unique –bigip-url in each Deployment (each Controller manages a separate BIG-IP device).
- Use the same –bigip-partition in all Deployments.

bigip01-cc.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: bigip01-ctlr
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
```

```
      name: k8s-bigip-ctlr1
      labels:
        app: k8s-bigip-ctlr1
    spec:
      serviceAccountName: bigip-ctlr
      containers:
        - name: k8s-bigip-ctlr
          image: "f5networks/k8s-bigip-ctlr:latest"
          command: ["/app/bin/k8s-bigip-ctlr"]
          args: [
            "--credentials-directory=/tmp/creds",
            "--bigip-url=10.10.200.98",
            "--bigip-partition=ocp",
            "--pool-member-type=cluster",
            "--manage-routes=true",
            "--node-poll-interval=5",
            "--verify-interval=5",
            "--namespace=demoproj",
            "--namespace=yelb",
            "--namespace=guestbook",
            "--namespace=f5demo",
            "--route-vserver-addr=10.10.201.120",
            "--route-http-vserver=ocp-vserver",
            "--route-https-vserver=ocp-https-vserver",
            "--openshift-sdn-name=/Common/ocp-tunnel"
          ]
          volumeMounts:
          - name: bigip-creds
            mountPath: "/tmp/creds"
            readOnly: true
      volumes:
      - name: bigip-creds
        secret:
          secretName: bigip-login
      imagePullSecrets:
        - name: f5-docker-images
```

bigip02-cc.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: bigip02-ctlr
  namespace: kube-system
spec:
  replicas: 1
  template:
    metadata:
      name: k8s-bigip-ctlr1
      labels:
        app: k8s-bigip-ctlr1
    spec:
      serviceAccountName: bigip-ctlr
      containers:
        - name: k8s-bigip-ctlr
          image: "f5networks/k8s-bigip-ctlr:latest"
          command: ["/app/bin/k8s-bigip-ctlr"]
          args: [
```

```
            "--credentials-directory=/tmp/creds",
            "--bigip-url=10.10.200.99",
            "--bigip-partition=ocp",
            "--pool-member-type=cluster",
            "--manage-routes=true",
            "--node-poll-interval=5",
            "--verify-interval=5",
            "--namespace=demoproj",
            "--namespace=yelb",
            "--namespace=guestbook",
            "--namespace=f5demo",
            "--route-vserver-addr=10.10.201.120",
            "--route-http-vserver=ocp-vserver",
            "--route-https-vserver=ocp-https-vserver",
            "--openshift-sdn-name=/Common/ocp-tunnel"
          ]
          volumeMounts:
          - name: bigip-creds
            mountPath: "/tmp/creds"
            readOnly: true
      volumes:
      - name: bigip-creds
        secret:
          secretName: bigip-login
      imagePullSecrets:
        - name: f5-docker-images
```

```
[root@ose-mstr01 ocp]# oc create -f  bigip01-cc.yaml
deployment "bigip01-ctlr" created
[root@ose-mstr01 ocp]# oc create -f  bigip02-cc.yaml
deployment "bigip02-ctlr" created
```

Step 4.4: Verify Pod creation

Verify the deployment and pods that are created

```
[root@ose-mstr01 ocp]# oc get deployment -n kube-system
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
bigip01-ctlr   1         1         1            1           42s
bigip02-ctlr   1         1         1            1           36s
```

Note: Check in your lab that you have your two controllers as **AVAILABLE**. If Not, you won't be able to do the lab. It may take up to 10 minutes for them to be available

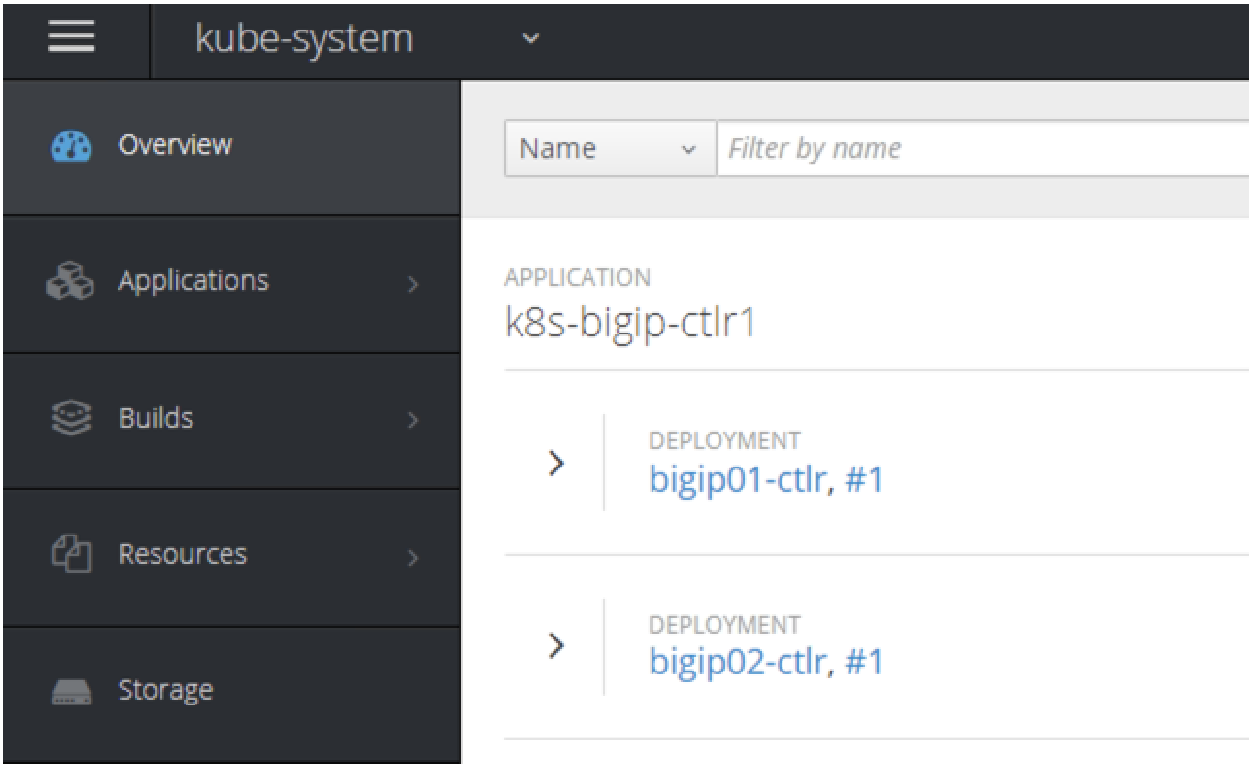
---

```
[root@ose-mstr01 ocp]# oc get deployment bigip01-ctlr -n kube-system
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
bigip01-ctlr   1         1         1            1           1m
```

```
[root@ose-mstr01 ocp]# oc get pods -n kube-system
NAME                            READY     STATUS    RESTARTS   AGE
bigip01-ctlr-242733768-dbwdm    1/1       Running   0          1m
bigip02-ctlr-66171581-q87kb     1/1       Running   0          1m
[root@ose-mstr01 ocp]#
```

You can also use the web console in OpenShift to view the bigip controller (login: **demouser**, password:

demouser). Go the kube-system project



6.2.10 Upload the Deployments

Step 4.5: Upload the Deployments to the OpenShift API server. Use the pool-only configmap to configuration project namespace: f5demo on the bigip

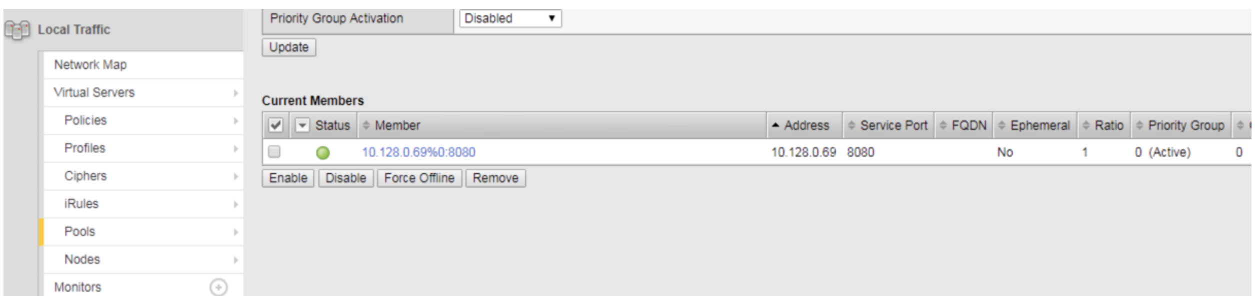
pool-only.yaml

```
kind: ConfigMap
apiVersion: v1
metadata:
  # name of the resource to create on the BIG-IP
  name: k8s.poolonly
  # the namespace to create the object in
  # As of v1.1, the k8s-bigip-ctlr watches all namespaces by default
  # If the k8s-bigip-ctlr is watching a specific namespace(s),
  # this setting must match the namespace of the Service you want to proxy
  # -AND- the namespace(s) the k8s-bigip-ctlr watches
  namespace: f5demo
  labels:
    # the type of resource you want to create on the BIG-IP
    f5type: virtual-server
data:
  schema: "f5schemadb://bigip-virtual-server_v0.1.7.json"
  data: |
    {
      "virtualServer": {
        "backend": {
          "servicePort": 8080,
          "serviceName": "f5demo",
```

```
          "healthMonitors": [{
            "interval": 3,
            "protocol": "http",
            "send": "GET /\r\n",
            "timeout": 10
          }]
        },
        "frontend": {
          "virtualAddress": {
            "port": 80
          },
          "partition": "ocp",
          "balance": "round-robin",
          "mode": "http"
        }
      }
    }
```

```
[root@ose-mstr01 ocp]# oc create -f pool-only.yaml
configmap "k8s.poolonly" created
[root@ose-mstr01 ocp]#
```

Step 4.6: Check **bigip01** and **bigip02** to make sure the pool got created (make sure you are looking at the **"ocp" partition**). Validate that green

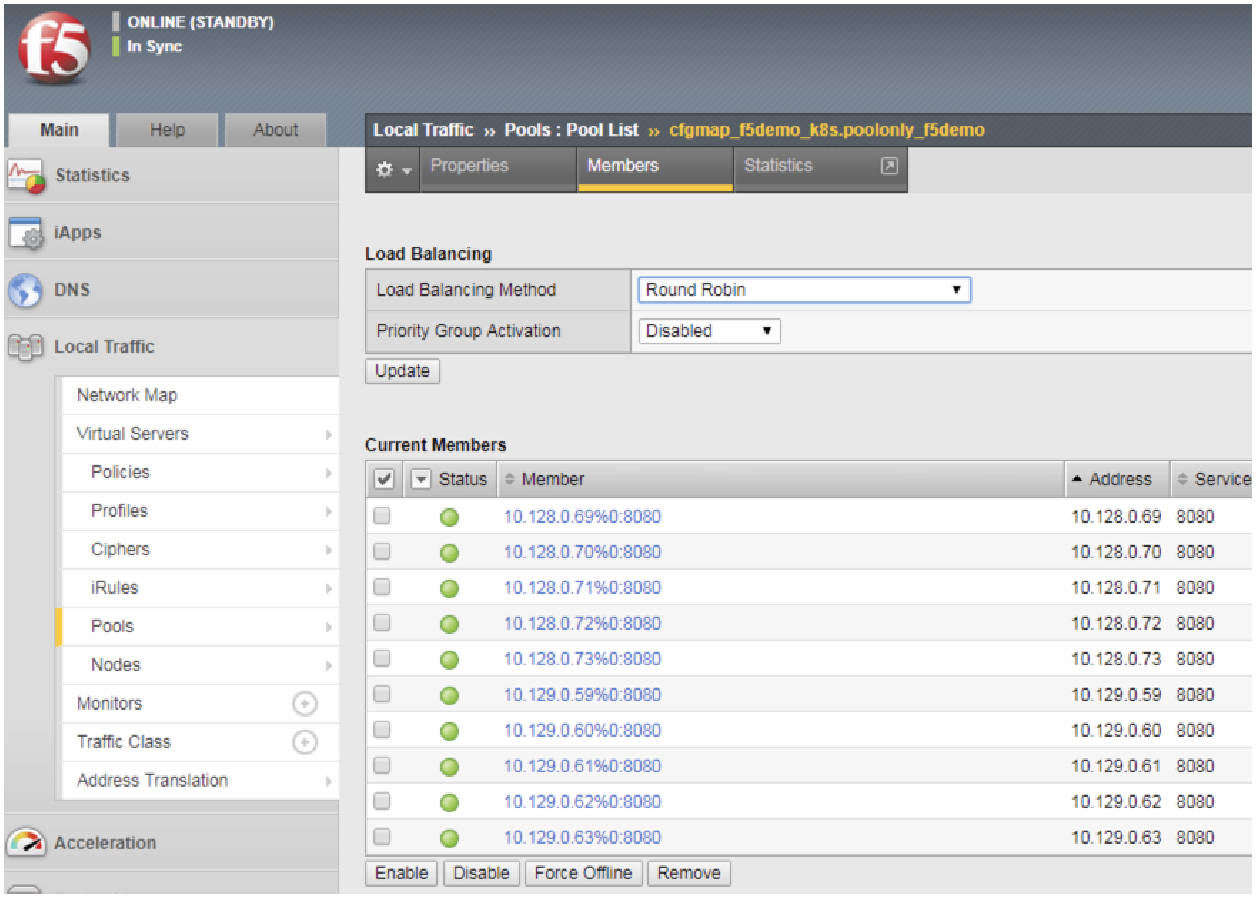


Step 4.7: Increase the replicas of the f5demo project pods. Replicas specified the required number of instances to run

```
[root@ose-mstr01 ocp]# oc scale --replicas=10 deployment/f5demo -n f5demo
deployment "f5demo" scaled
[root@ose-mstr01 ocp]#
```

Note: It may take time to have your replicas up and running. Don't hesitate to track this by using the following command. to check the number of **AVAILABLE** instances:

oc get deployment f5demo -n f5demo



Validate that bigip01 and bigip02 are updated with the additional pool members and their health monitor works. If the monitor is failing check the tunnel and selfIP.

6.2.11 Validation and Troubleshooting

Now that you have HA configured and uploaded the deployment, it is time to generate traffic through our BIG-IPs.

Step 5.1: Create a virtual IP address for the deployment

Add a virtual IP to the the configmap. You can edit the pool-only.yaml configmap. There are multiple ways to edit the configmap which will be covered in module 3. In this task remove the deployment, edit the yaml file and re-apply the deployment

```
[root@ose-mstr01 ocp]# oc delete -f pool-only.yaml
configmap "k8s.poolonly" deleted
[root@ose-mstr01 ocp]#
```

Edit the pool-only.yaml and add the bindAddr

vi pool-only.yaml

```
"frontend": {
   "virtualAddress": {
      "port": 80,
      "bindAddr": "10.10.201.220"
```

Tip: Do not use TAB in the file, only spaces. Don't forget the "," at the end of the ""port": 80," line.

Create the modified pool-only deployment

```
[root@ose-mstr01 ocp]# oc create -f pool-only.yaml
configmap "k8s.poolonly" created
[root@ose-mstr01 ocp]#
```

Connect to the virtual server at http://10.10.201.220. Does the connection work? If not, try the following troubleshooting options:

1. Capture the http request to see if the connection is established with the BIG-IP
2. Follow the following network troubleshooting section

6.2.12 Network Troubleshooting

Attention: How do I verify connectivity between the BIG-IP VTEP and the OSE Node?

1. Ping the Node's VTEP IP address. Use the `-s` flag to set the MTU of the packets to allow for VxLAN encapsulation.

```
[root@bigip01:Standby:Changes Pending] config # ping -s 1600 -c 4 10.10.199.101
PING 10.10.199.101 (10.10.199.101) 1600(1628) bytes of data.
1608 bytes from 10.10.199.101: icmp_seq=1 ttl=64 time=2.94 ms
1608 bytes from 10.10.199.101: icmp_seq=2 ttl=64 time=2.21 ms
1608 bytes from 10.10.199.101: icmp_seq=3 ttl=64 time=2.48 ms
1608 bytes from 10.10.199.101: icmp_seq=4 ttl=64 time=2.47 ms

--- 10.10.199.101 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 2.210/2.527/2.946/0.267 ms
```

2. Ping the Pod's IP address (use the output from looking at the pool members in the previous steps). Use the `-s` flag to set the MTU of the packets to allow for VxLAN encapsulation.

```
[root@bigip01:Standby:Changes Pending] config # ping -s 1600 -c 4 10.128.0.54
PING 10.128.0.54 (10.128.0.54) 1600(1628) bytes of data.

--- 10.128.0.54 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time 12999ms
```

3. Now change the MTU to 1400

```
[root@bigip01:Standby:Changes Pending] config # ping -s 1400 -c 4 10.128.0.54
PING 10.128.0.54 (10.128.0.54) 1400(1428) bytes of data.
1408 bytes from 10.128.0.54: icmp_seq=1 ttl=64 time=1.74 ms
1408 bytes from 10.128.0.54: icmp_seq=2 ttl=64 time=2.43 ms
1408 bytes from 10.128.0.54: icmp_seq=3 ttl=64 time=2.77 ms
1408 bytes from 10.128.0.54: icmp_seq=4 ttl=64 time=2.25 ms

--- 10.128.0.54 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 1.748/2.303/2.774/0.372 ms
```

Note: When pinging the VTEP IP directly the BIG-IP was L2 adjacent to the device and could send a large MTU.

In the second example, the packet is dropped across the VxLAN tunnel.

In the third example, the packet is able to traverse the VxLAN tunnel.

4. In a TMOS shell, output the REST requests from the BIG-IP logs.

   - Do a `tcpdump` of the underlay network.

     Example showing two-way communication between the BIG-IP VTEP IP and the OSE node VTEP IPs.

     Example showing traffic on the overlay network; at minimum, you should see BIG-IP health monitors for the Pod IP addresses.

```
[root@bigip01:Standby:Changes Pending] config # tcpdump -i ocp-tunnel -c 10 -nnn
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on ocp-tunnel, link-type EN10MB (Ethernet), capture size 65535 bytes
09:05:55.962408 IP 10.131.0.98.53404 > 10.128.0.54.8080: Flags [S], seq␣
↪1597206142, win 29200, options [mss 1460,sackOK,TS val 441031289 ecr 0,nop,
↪wscale 7], length 0 out slot1/tmm0 lis=
09:05:55.963532 IP 10.128.0.54.8080 > 10.131.0.98.53404: Flags [S.], seq␣
↪1644640677, ack 1597206143, win 27960, options [mss 1410,sackOK,TS val 3681001␣
↪ecr 441031289,nop,wscale 7], length 0 in slot1/tmm1 lis=
09:05:55.964361 IP 10.131.0.98.53404 > 10.128.0.54.8080: Flags [.], ack 1, win␣
↪229, options [nop,nop,TS val 441031291 ecr 3681001], length 0 out slot1/tmm0␣
↪lis=
09:05:55.964367 IP 10.131.0.98.53404 > 10.128.0.54.8080: Flags [P.], seq 1:10,␣
↪ack 1, win 229, options [nop,nop,TS val 441031291 ecr 3681001], length 9: HTTP:␣
↪GET / out slot1/tmm0 lis=
09:05:55.965630 IP 10.128.0.54.8080 > 10.131.0.98.53404: Flags [.], ack 10, win␣
↪219, options [nop,nop,TS val 3681003 ecr 441031291], length 0 in slot1/tmm1 lis=
09:05:55.975754 IP 10.128.0.54.8080 > 10.131.0.98.53404: Flags [P.], seq 1:1337,␣
↪ack 10, win 219, options [nop,nop,TS val 3681013 ecr 441031291], length 1336:␣
↪HTTP: HTTP/1.1 200 OK in slot1/tmm1 lis=
09:05:55.975997 IP 10.128.0.54.8080 > 10.131.0.98.53404: Flags [F.], seq 1337,␣
↪ack 10, win 219, options [nop,nop,TS val 3681013 ecr 441031291], length 0 in␣
↪slot1/tmm1 lis=
09:05:55.976108 IP 10.131.0.98.53404 > 10.128.0.54.8080: Flags [.], ack 1337, win␣
↪251, options [nop,nop,TS val 441031302 ecr 3681013], length 0 out slot1/tmm0␣
↪lis=
09:05:55.976114 IP 10.131.0.98.53404 > 10.128.0.54.8080: Flags [F.], seq 10, ack␣
↪1337, win 251, options [nop,nop,TS val 441031303 ecr 3681013], length 0 out␣
↪slot1/tmm0 lis=
09:05:55.976488 IP 10.131.0.98.53404 > 10.128.0.54.8080: Flags [.], ack 1338, win␣
↪251, options [nop,nop,TS val 441031303 ecr 3681013], length 0 out slot1/tmm0␣
↪lis=
10 packets captured
10 packets received by filter
0 packets dropped by kernel
```

5. In a TMOS shell, view the MAC address entries for the OSE tunnel. This will show the mac address and IP addresses of all of the OpenShift endpoints.

```
root@(bigip02)(cfg-sync In Sync)(Active)(/Common)(tmos)# show /net fdb tunnel ocp-
↪tunnel
```

```
----------------------------------------------------------------
Net::FDB
Tunnel      Mac Address        Member                    Dynamic
----------------------------------------------------------------
ocp-tunnel  0a:0a:0a:0a:c7:64  endpoint:10.10.199.100%0  no
ocp-tunnel  0a:0a:0a:0a:c7:65  endpoint:10.10.199.101%0  no
ocp-tunnel  0a:0a:0a:0a:c7:66  endpoint:10.10.199.102%0  no
ocp-tunnel  0a:58:0a:80:00:60  endpoint:10.10.199.101    yes
```

6. In a TMOS shell, view the ARP entries.

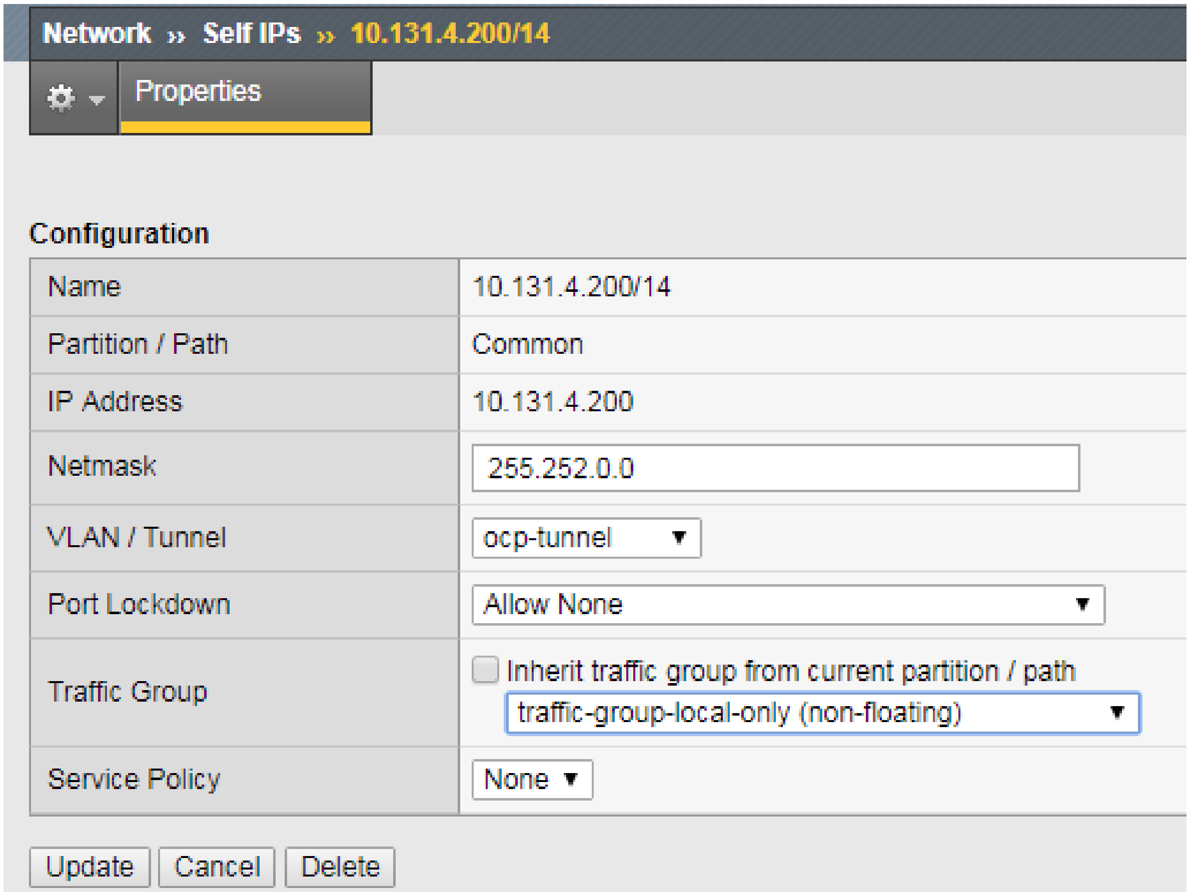
   Note: run the command "tmsh" if you do not see "(tmos)" in your shell.

   This will show all of the ARP entries; you should see the VTEP entries on the `ocpvlan` and the Pod IP addresses on `ose-tunnel`.

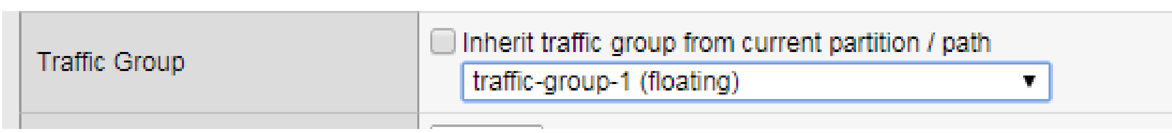
```
root@(bigip02)(cfg-sync In Sync)(Active)(/Common)(tmos)# show /net arp

---------------------------------------------------------------------------------
↪----------
Net::Arp
Name           Address        HWaddress          Vlan                Expire-in-
↪sec  Status
---------------------------------------------------------------------------------
↪----------
10.10.199.100  10.10.199.100  2c:c2:60:49:b2:9d  /Common/internal    41          ␣
↪  resolved
10.10.199.101  10.10.199.101  2c:c2:60:58:62:64  /Common/internal    70          ␣
↪  resolved
10.10.199.102  10.10.199.102  2c:c2:60:51:65:a0  /Common/internal    41          ␣
↪  resolved
10.10.202.98   10.10.202.98   2c:c2:60:1f:74:62  /Common/ha          64          ␣
↪  resolved
10.128.0.96    10.128.0.96    0a:58:0a:80:00:60  /Common/ocp-tunnel  7           ␣
↪  resolved

root@(bigip02)(cfg-sync In Sync)(Active)(/Common)(tmos)#
```

7. Validate floating traffic for ocp-tunnel self-ip

   Check if the configuration is correct from step 3.6. Make sure the floating IP is set to traffic-group-1 floating. A floating traffic group is request for the response traffic from the pool-member. If the traffic is local change to floating



change to floating



Connect to the viutal IP address

8. Test failover and make sure you can connect to the virtual.

Congratulations for completing the HA clustering setup. Before moving to the next module cleanup the deployed resource:

```
[root@ose-mstr01 ocp]# oc delete -f pool-only.yaml
configmap "k8s.poolonly" created
[root@ose-mstr01 ocp]#
```

6.3 Module 3: Container Connector in Action

This section of the lab will cover creating OpenShift resources that the F5 Container Connector will process and use to update the BIG-IP configuration and leverages the work you did in the previous sections.

6.3.1 Operational Overview

The Container Connector watches for events being generated by the Openshift API server and takes action when it sees an OpenShift ConfigMap or Route resource that has an F5-specific label defined. The Container Connector parses the ConfigMap or Route resource and updates the BIG-IP configuration to match the desired state as defined by those resources.

In addition to watching and responding to events in real time, the Container Connector periodically queries the OpenShift API for the current status and updates the BIG-IP as needed. This interval (verify-interval) is

30 seconds by default but is a startup value that can be modified.

An instance of the Container Connector can watch for changes in all namespaces (projects), a single namespace or a discrete list of namespaces. Additionally, an instance of the Container Connector is configured to make configuration changes in a single non-Common BIG-IP partition.

OpenShift runs on top of Kubernetes and the same Container Connector works for both, but many of the Container Connector features apply to both while some apply only to OpenShift, like Routes, while others, like Ingress, apply only to Kubernetes.

You can find detailed information about configuring, deploying and using the F5 Container Connector as well as configuration options for ConfigMaps and Routes https://clouddocs.f5.com/containers/v2/#

Additionally, you can get more detailed information about an OpenShift command by using **oc <command> -help**. So, for example, if you wanted to find out more about the **oc create** command, you would do the following:

```
[root@ose-mstr01 garrison]# oc create -help
```

In the following exercises, you will create the following OpenShift resource types:

- ConfigMaps
- Routes

Additionally, you will also create variations of each resource type.

Note: You will use the same Windows jumpbox as you used in the previous sections to complete the exercises in this section.

Unless otherwise noted, all the resource definition yaml files have been pre-created and can be found on the **ose-master** server under **/root/agility2018/apps/module3**

6.3.2 Exercise 1: ConfigMap - Basic

An OpenShift ConfigMap is one of the resource types that the F5 Container Connector watches for. The Container Connector will read the ConfigMap and create a virtual server, node(s), a pool, pool member(s) and a pool health monitor.

In this exercise, you will create a ConfigMap that defines the objects that the Container Connector should configure on the BIG-IP.

To complete this exercise, you will perform the following steps:

- Step 1: Deploy demo application
- Step 2: Create a service to expose the demo application
- Step 3: Create a ConfigMap that declares desired BIG-IP configuration
- Step 4: Review the BIG-IP configuration
- Step 5: Test the application
- Step 6: Scale the application
- Step 7: Test the scaled application
- Step 8: Cleanup deployed resources

Step 1: Deploy demo application

From the **ose-master**, review the following Deployment configuration: **f5-demo-app-deployment.yaml**

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: f5-demo-app
  namespace: f5demo
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: f5-demo-app
    spec:
      containers:
      - name: f5-demo-app
        image: chen23/f5-demo-app:openshift
        ports:
         - containerPort: 8080
           protocol: TCP
```

Now that you have reviewed the Deployment,you need to actually create the Deployment by deploying it to OpenShift by using the **oc create** command.

From **ose-master** server, run the following command:

> **Attention:** Be sure to change the working directory on **ose-mstr01**:
>
> cd /root/agility2018/apps/module3

```
[root@ose-mstr01 module3]# oc create -f f5-demo-app-deployment.yaml
deployment "f5-demo-app" created
```

Step 2: Create Service to expose application

In order for an application to be accessible outside of the OpenShift cluster, a Service must be created. The Service uses a label selector to reference the application to be exposed. Additionally, the service also specifies the container port (8080) that the application is listening on.

From **ose-master**, review the following Service: f5-demo-app-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: f5-demo-app
  labels:
    name: f5-demo-app
  namespace: f5demo
spec:
  type: ClusterIP
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: f5-demo-app
```

Now that you have reviewed the Service,you need to actually create the Service by deploying it to OpenShift by using the **oc create** command.

From ose-master server, run the following command:

```
[root@ose-mstr01 module3]# oc create -f f5-demo-app-service.yaml
service "f5-demo-app" created
```

Step 3: Create ConfigMap

A ConfigMap is used to define the BIG-IP objects that need to be created to enable access to the application via the BIG-IP.

The label, **f5type: virtual-server**, in the ConfigMap definition is what triggers the F5 Container Connector to process this ConfigMap.

In addition to the label, there are several F5-specific sections defined:

- **virtualServer:** Beginning of F5-specific configuration
- **backend:** Represents the server-side of the virtual server definition
- **healthMonitors:** Health monitor definition for the pool
- **frontend:** Represents the client-side of the virtual server
- **virtualAddress:** IP address and port of virtual server

A **ConfigMap** points to a **Service** which points to one or more **Pods** where the application is running.

From ose-master, review the ConfigMap resource f5-demo-app-configmap.yaml

```
kind: ConfigMap
apiVersion: v1
metadata:
  # name of the resource to create on the BIG-IP
  name: f5-demo-app
  # The namespace to create the object in.
  # The k8s-bigip-ctlr watches all namespaces by default (as of v1.1).
  # If the k8s-bigip-ctlr is watching a specific namespace(s),
  # this setting must match the namespace of the Service you want to proxy
  # -AND- the namespace(s) the k8s-bigip-ctlr watches.
  namespace: f5demo
  labels:
    # tells the k8s-bigip-ctlr to watch this ConfigMap
    f5type: virtual-server
data:
  # NOTE: schema v0.1.4 is required as of k8s-bigip-ctlr v1.3.0
  schema: "f5schemadb://bigip-virtual-server_v0.1.7.json"
  data: |
    {
      "virtualServer": {
        "backend": {
          "servicePort": 8080,
          "serviceName": "f5-demo-app",
          "healthMonitors": [{
            "interval": 30,
            "protocol": "http",
            "send": "GET /\r\n",
            "timeout": 120
          }]
        },
```

```
        "frontend": {
          "virtualAddress": {
            "port": 80,
            "bindAddr": "10.10.201.130"
          },
          "partition": "ocp",
          "balance": "least-connections-node",
          "mode": "http"
        }
      }
    }
```

> **Attention:** *Knowledge Check: How does the BIG-IP know which pods make up the application?*

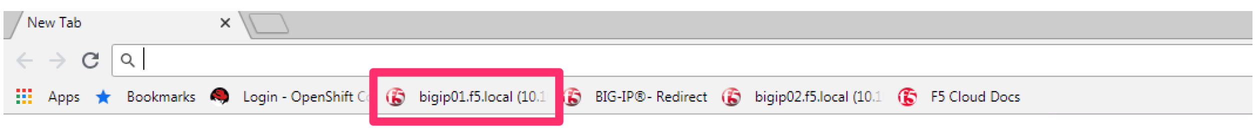
Now that you have reviewed the ConfigMap, you need to actually create the ConfigMap by deploying it to OpenShift by using the **oc create** command:

```
[root@ose-mstr01 module3]# oc create -f f5-demo-app-configmap.yaml
configmap "f5-demo-app" created
```

Step 4: Review BIG-IP configuration

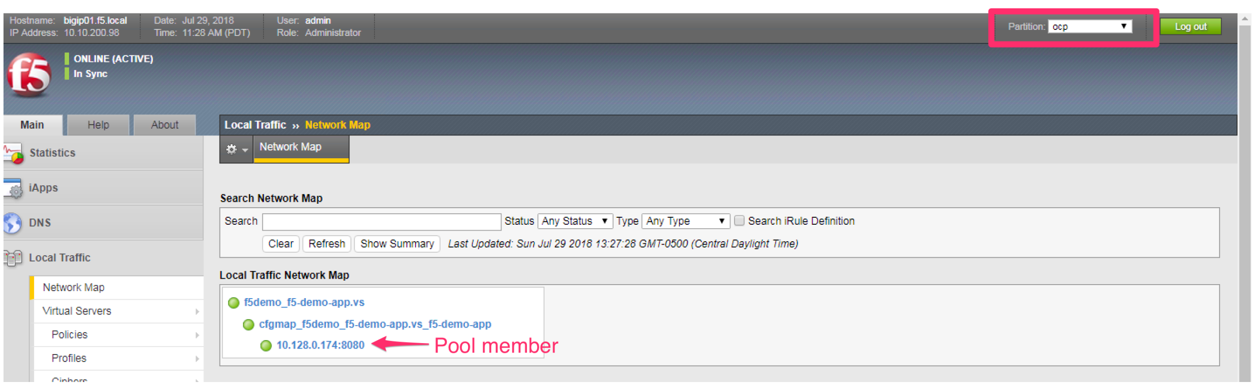
In this step, you will examine the BIG-IP configuration that was created by the Container Connector when it processed the ConfigMap created in the previous step.

Launch the Chrome browser and click on the bookmark named **bigip01.f5.local** to access the BIG-IP GUI:



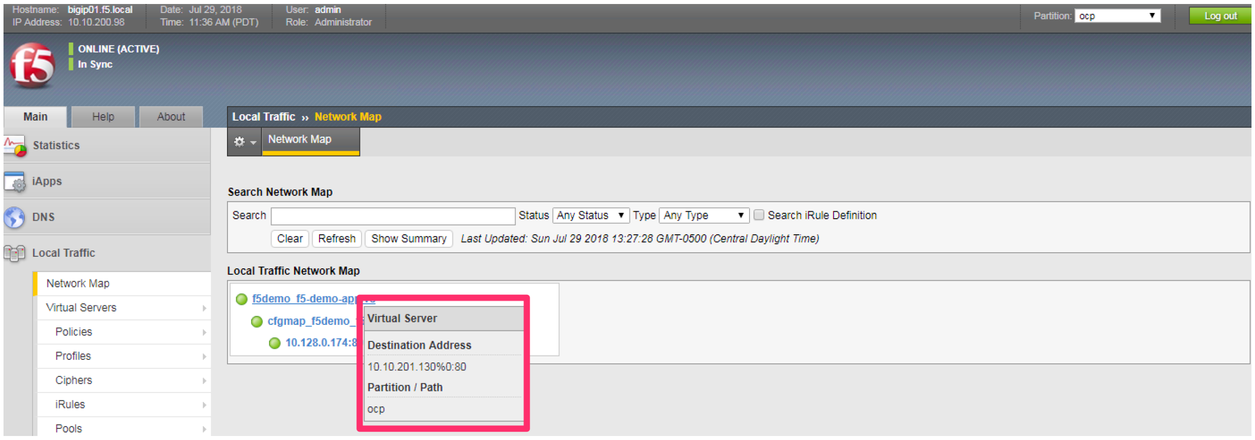
From the BIG-IP login page, enter username=admin and password=admin and click the **Log in** button:

Navigate to **Local Traffic –> Network Map** and change the partition to **ocp** using the dropdown in the upper right. The network map view shows a virtual server, pool and pool member. All of these objects were created by the Container Connector using the declarations defined in the ConfigMap.



> **Attention:** *Knowledge Check: In the network map view, what OpenShift object type does the pool member IP address represent? How was the IP address assigned?*

To view the IP address of the virtual server, hover your cursor over the name of the virtual server:

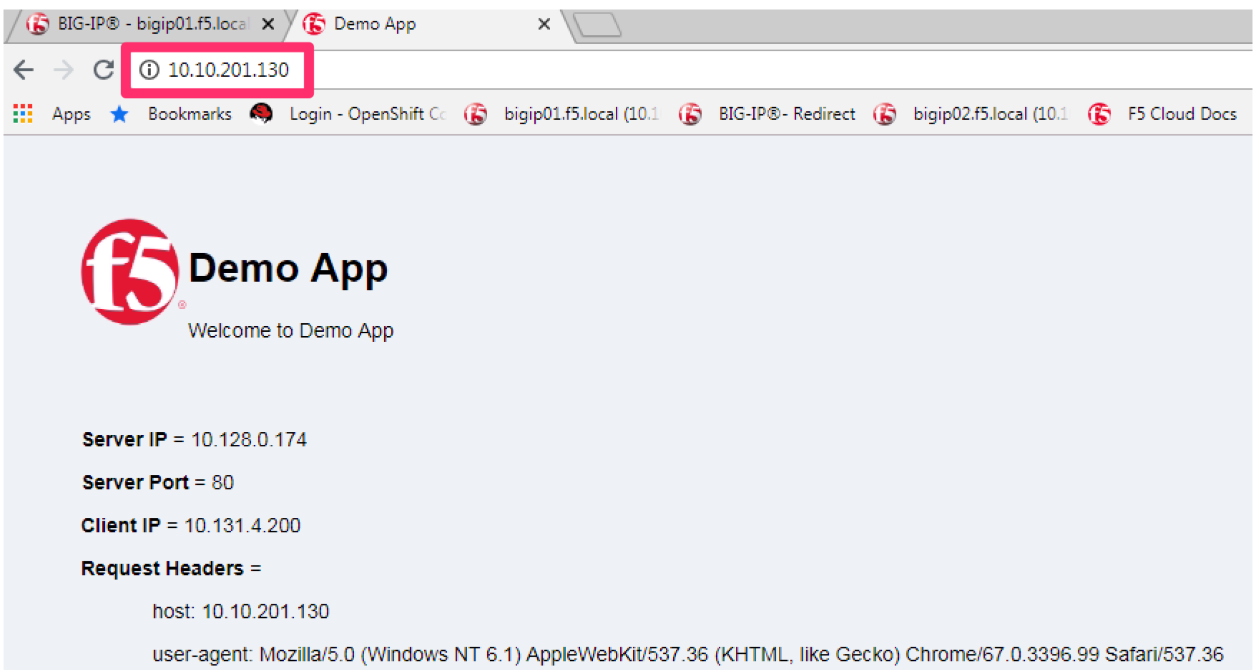


Attention: *Knowledge Check: What OpenShift resource type was used to define the virtual server IP address?*

Step 5: Test the application

In this step, you will use the Chrome browser to access the application you previously deployed to OpenShift.

Open a new browser tab and enter the IP address assigned to the virtual server in to the address bar:



Note: On the application page, the **Server IP** is the pool member (pod) IP address; the **Server Port** is the port of the virtual server; and the **Client IP** is the IP address of the Windows jumpbox you are using.

Step 6: Scale the application

The application deployed in step #1 is a single replica (instance). In this step,you are going to increase the number of replicas and then check the BIG-IP configuration to see what's changed.

When the deployment replica count is scaled up or scaled down, an OpenShift event is generated and the Container Connector sees the event and adds or removes pool members as appropriate.

To scale the number of replicas, you will use the OpenShift **oc scale** command. You will be scaling the demo app deployment and so You first need to get the name of the deployment.

From ose-master, issue the following command:

```
[root@ose-mstr01 module3]# oc get deployment -n f5demo
NAME           DESIRED    CURRENT    UP-TO-DATE    AVAILABLE    AGE
f5-demo-app    1          1          1             1            1m
```

You can see from the output that the deployment is named **f5-demo-app**. You will use that name for the next command.

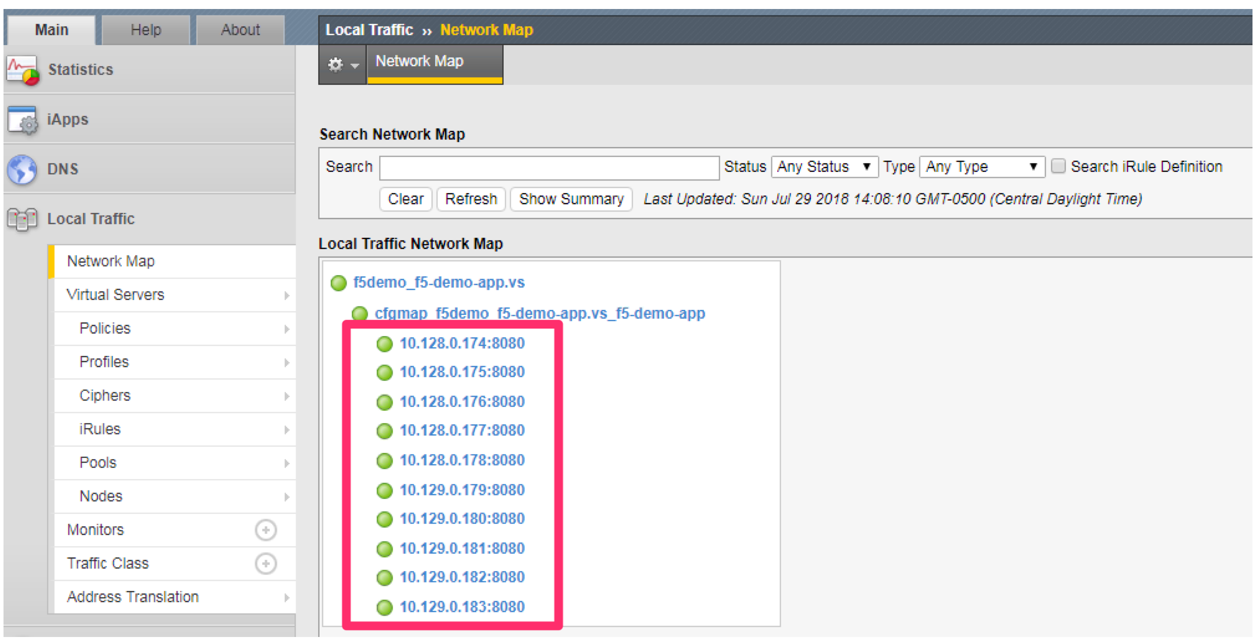
From the ose-master host, entering the following command to set the replica count for the deployment to 10 instances:

```
[root@ose-mstr01 module3]# oc scale --replicas=10 deployment/f5-demo-app -n f5demo
deployment "f5-demo-app" scaled
```

Step 7: Review the BIG-IP configuration

In this step, you will examine the BIG-IP configuration for changes that occured after the application was scaled up.

Navigate to **Local Traffic –> Network Map** and change the partition to **ocp** using the dropdown in the upper right.

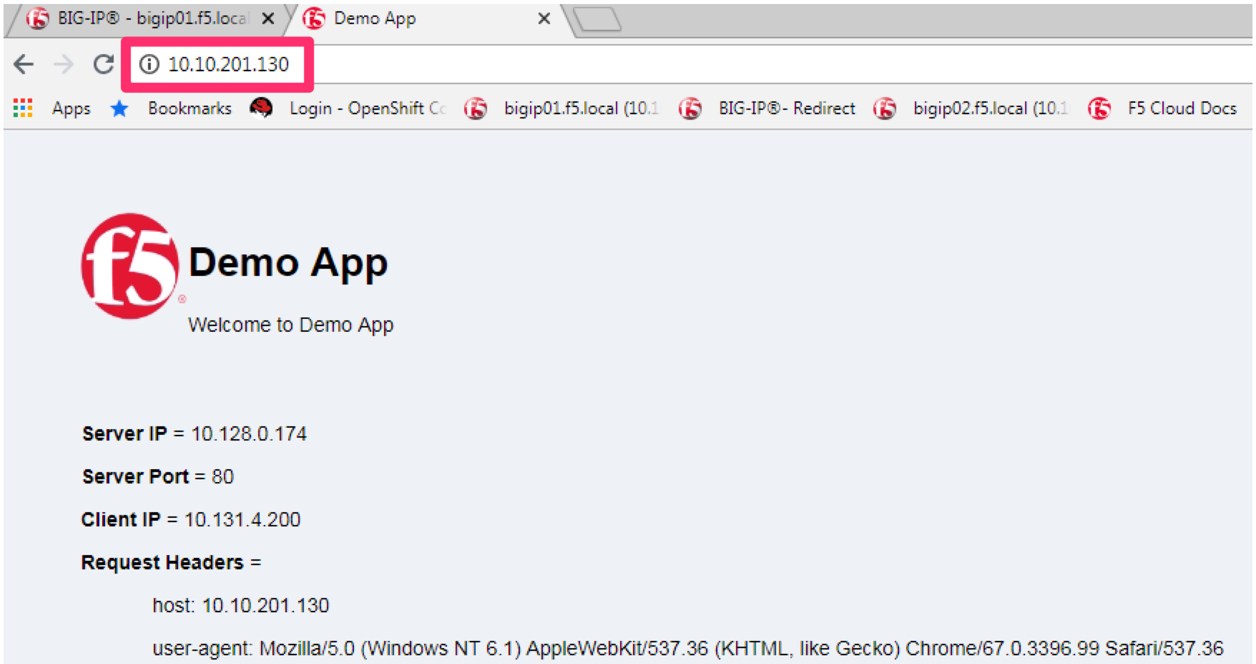


> **Attention:** *Knowledge Check: How many pool members are shown in the network map view? What do you think would happen if you scaled the deployment back to one replica?*

Step 8: Test the scaled application

In this step, you will use the Chrome browser to access the application that you scaled to 10 replicas in the previous step.

Open a new Chrome browser tab and enter the IP address assigned to the virtua server in to the address bar:



If you reload the page every few seconds, you should see the **Server IP** address change. Because there is more than one instance of the application running, the BIG-IP load balances the application traffic amongst multiple pods.

Step 9: Cleanup deployed resources

In this step, you will remove the OpenShift Deployment, Service and ConfigMap resources you created in the previous steps using the OpenShift **oc delete** command.

From ose-master server, issue the following commands:

```
[root@ose-mstr01 tmp]# oc delete -f f5-demo-app-configmap.yaml
configmap "f5-demo-app" deleted

[root@ose-mstr01 tmp]# oc delete -f f5-demo-app-deployment.yaml
deployment "f5-demo-app" deleted

[root@ose-mstr01 module3]# oc delete -f f5-demo-app-service.yaml
service "f5-demo-app" deleted
```

6.3.3 Exercise 2: Route - Basic

An OpenShift Route is one of the resource types that the F5 Container Connector watches for. A Route defines a hostname or URI mapping to an application. For example, the hostname “customer.example.com” could map to the application “customer”, hostname “catalog.example.com”, might map to the application “catalog”, etc.

Similarly, a Route can refer to a URI path so, for example, the URI path “/customer” might map to the application called “customer” and URI path “/catalog”, might map to the application called “catalog”. If a Route only specifies URI paths, the Route applies to all HTTP request hostnames.

Additionally, a Route can refer to both a hostname and a URI path such as mycompany.com/customer or mycompany.com/catalog

The F5 Container Connector reads the Route resource and creates a virtual server, node(s), a pool per route path and pool members. Additionally, the Container Connector creates a layer 7 BIG-IP traffic policy and associates it with the virtual server. This layer 7 traffic policy evaluates the hostname or URI path from the request and forwards the traffic to the pool associated with that path.

A **Route** points to a **Service** which points to one or more **Pods** where the application is running.

Note: All Route resources share two virtual servers:

- **ose-vserver** for HTTP traffic, and
- **https-ose-vserver** for HTTPS traffic

The Container Connector assigns the names shown above by default. To set custom names, define **route-http-vserver** and **route-https-vserver** in the BIG-IP Container Connector Deployment. Please see the documentation at: http://clouddocs.f5.com for more details.

To complete this exercise, you will perform the following steps:

- Step 1: Deploy demo application and associated Service
- Step 2: Create a Route that defines routing rules based on hostname
- Step 3: Review the BIG-IP configuration

Step 1: Deploy demo application and its associated Service

In the previous exercise, you created the Deployment and Service separately. This step demonstrates creating both the Deployment and the Service from single configuration file. A separator of 3 dashes (`---`) is used to separate one resource definition from the next resource definition.

From ose-master, review the following deployment: f5-demo-app-route-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: f5-demo-app-route
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: f5-demo-app-route
    spec:
      containers:
      - name: f5-demo-app-route
        image: chen23/f5-demo-app:openshift
        ports:
         - containerPort: 8080
           protocol: TCP
---
apiVersion: v1
kind: Service
metadata:
  name: f5-demo-app-route
  labels:
    name: f5-demo-app-route
```

```
  namespace: f5demo
spec:
  type: ClusterIP
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: f5-demo-app-route
```

Now that you have reviewed the Deployment, you need to actually create it by deploying it to OpenShift by using the **oc create** command:

```
[root@ose-mstr01 tmp]# oc create -f f5-demo-app-route-deployment.yaml -n f5demo
deployment "f5-demo-app-route" created
service "f5-demo-app-route" created
```

Step 2: Create OpenShift Route

In this step, you will create an OpenShift Route.

From ose-master server, review the following Route: f5-demo-app-route-route.yaml

```
apiVersion: v1
kind: Route
metadata:
  labels:
    name: f5-demo-app-route
  name: f5-demo-app-route
  namespace: f5demo
  annotations:
    # Specify a supported BIG-IP load balancing mode
    virtual-server.f5.com/balance: least-connections-node
    virtual-server.f5.com/health: |
      [
        {
          "path": "mysite.f5demo.com/",
          "send": "HTTP GET /",
          "interval": 5,
          "timeout": 10
        }
      ]
spec:
  host: mysite.f5demo.com
  path: "/"
  port:
    targetPort: 8080
  to:
    kind: Service
    name: f5-demo-app-route
```

> **Attention:** *Knowledge Check: How does the Container Connector know what application the Route refers to?*

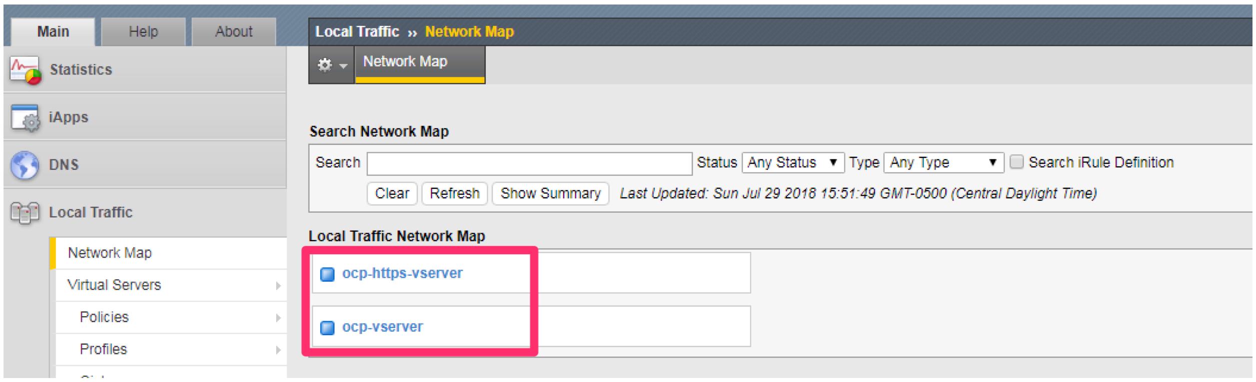
Now that you have reviewed the Route, you need to actually create it by deploying it to OpenShift by using the **oc create** command:

```
[root@ose-mstr01 tmp]# oc create -f f5-demo-app-route-route.yaml -n f5demo
route "f5-demo-app-route" created
```

Step 3: Review the BIG-IP configuration

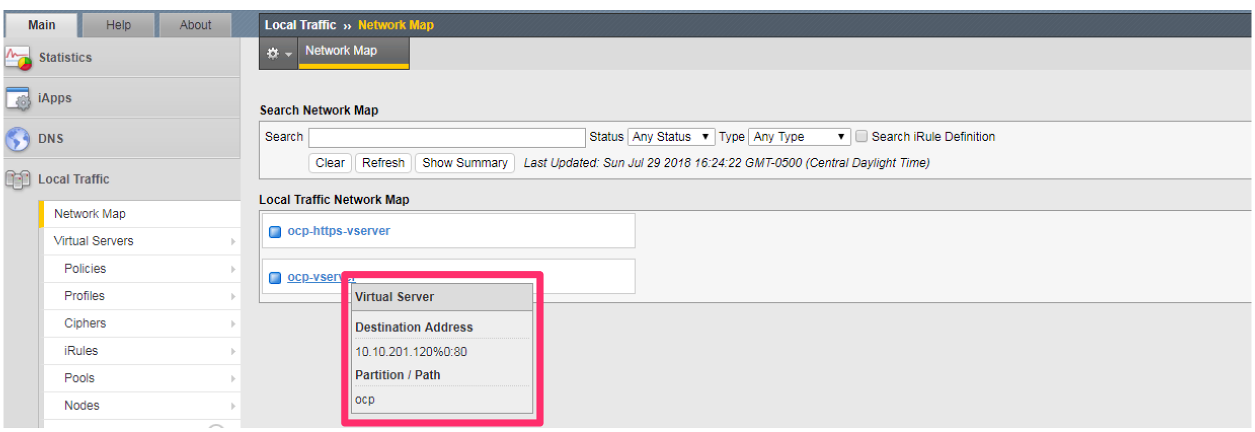
In this step, you will examine the BIG-IP configuration for changes that occured after the the OpenShift Route was deployed.

Using the Chrome browser, navigate to **Local Traffic –> Network Map** and change the partition to **ocp** using the dropdown in the upper right.



The network map view shows two virtual servers that were created by the Container Connector when it procssed the Route resource created in the previous step. One virtual server is for HTTP client traffic and the other virtual server is for HTTPS client traffic.

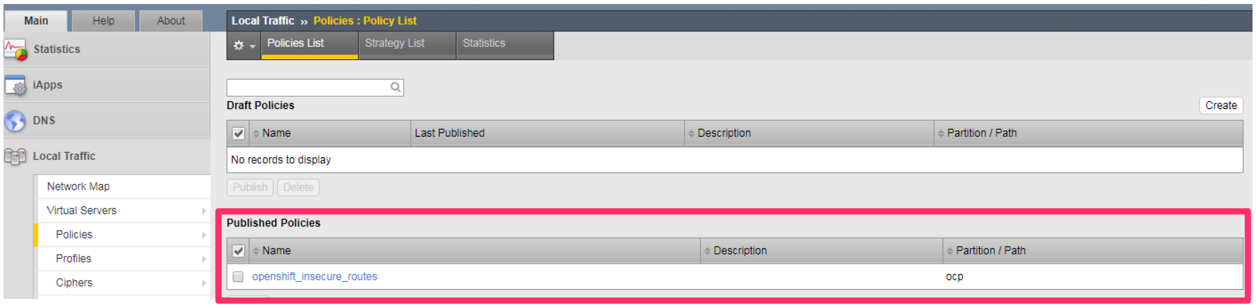
To view the IP address of the virtual server, hover your cursor over the virtual server named **ocp-vserver**



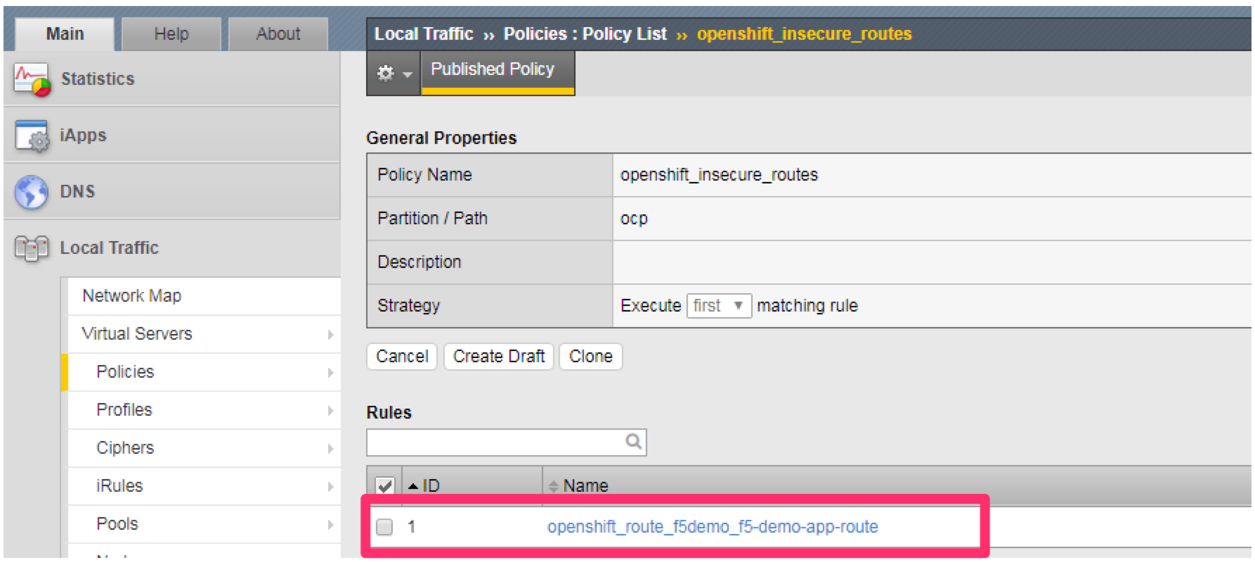
Attention: *Knowledge Check: Which OpenShift resource type defines the names of the two virtual servers?*

Next, you will view the traffic policy that was created by the Container Connector when it processed the OpenShift Route.

Navigate to **Local Traffic –> Policies –> Policy List** and change the partition to **ocp** using the dropdown in the upper right.

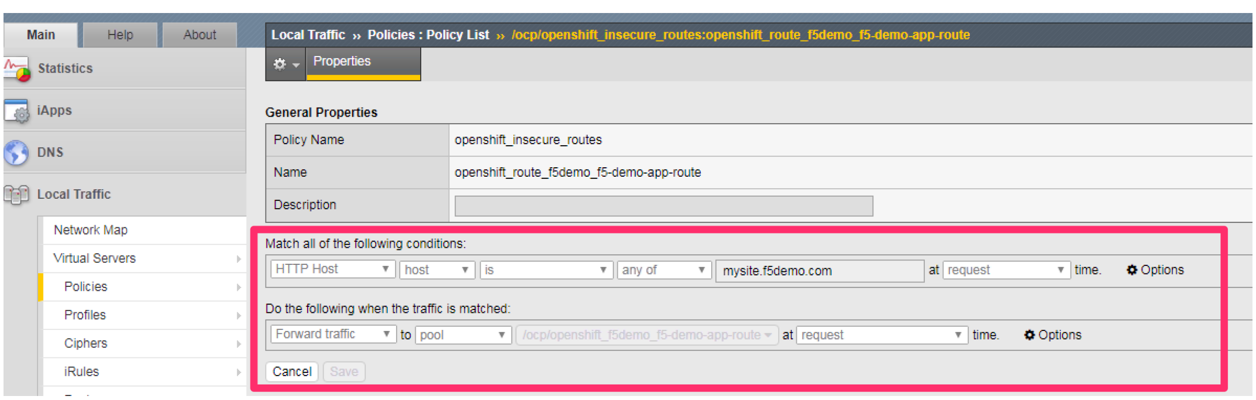


Click on the traffic policy listed uner **Published Policies** to view the policy page for the selected policy:



Next, click on the rule name listed under the **Rules** section of the policy page to view the rule page for the selected rule:

> **Warning:** Due to the version of TMOS used in this lab you will not see the correct "hostname" due to a GUI issue.



On the rule page, review the configuration of the rule and note the match condition and rule action settings.

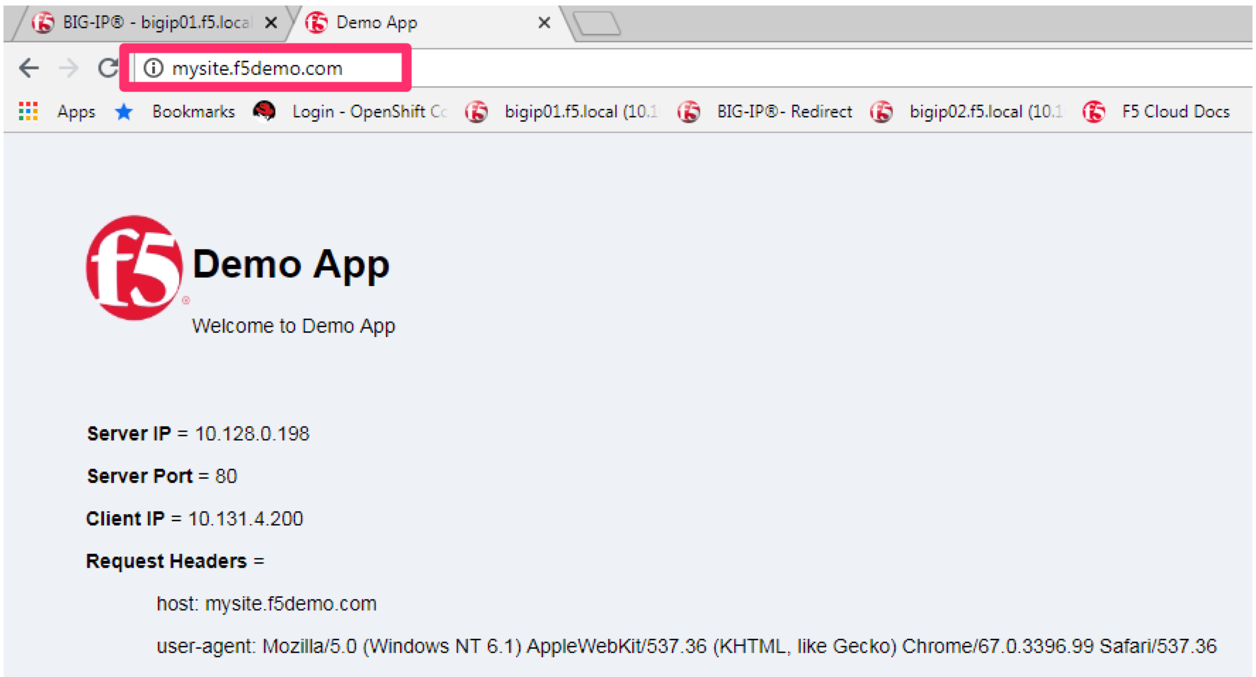
Attention: *Knowledge Check: Which OpenShift resource type defines the hostname to match against?*

Step 5: Test the application

In this step, you will use the Chrome browser to access the application you previously deployed.

Because the Route resource you created specifies a hostname for the path, you will need to use a hostname instead of an IP address to access the demo application.

Open a new Chrome browser tab and enter the hostname **mysite.f5demo.com** in to the address bar:



On the application page, the **Server IP** is the pool member (pod) IP address; the **Server Port** is the port of the virtual server; and the **Client IP** is the IP address of the Windows jumpbox you are using.

Step 6: Cleanup deployed resources

In this step, you will remove the Deployment, Service and Route resources you created in the previous steps using the OpenShift **oc delete** command.

From ose-master server, issue the following commands:

```
[root@ose-mstr01 tmp]# oc delete -f f5-demo-app-route-route.yaml -n f5demo
route "f5-demo-app-route" deleted

[root@ose-mstr01 tmp]# oc delete -f f5-demo-app-route-deployment.yaml -n f5demo
deployment "f5-demo-app-route" deleted
service "f5-demo-app-route" deleted
```

6.3.4 Exercise 3: Route - Blue/Green Testing

The F5 Container Connector supports Blue/Green application testing e.g testing two different versions of the same application, by using the **weight** parameter of OpenShift Routes. The **weight** parameter allows you to establish relative ratios between application **Blue** and application **Green**. So, for example, if the first route specifies a weight of 20 and the second a weight of 10, the application associated with the first route ill get twice the number of requests as the application associated with the second route.

Just as in the previous exercise, the F5 Container Connector reads the Route resource and creates a virtual server, node(s), a pool per route path and pool members.

However, in order to support Blue/Green testing using OpenShift Routes, the Container Connector creates an iRule and a datagroup on the BIG-IP. Troubleshooting handles the connection routing based on the assigned weights.

Note: At smaller request volumes, the ratio of requests to the **Blue** application and the requests to the **Green** application may not match the relative weights assigned in the OpenShift Route. However, as the number of requests increases, the ratio of requests between the **Blue** application and the **Green** application should closely match the weights assigned in the OpenShift Route.

To complete this exercise, you will perform the following steps:

- Step 1: Deploy version 1 and version 2 of demo application and their related Services
- Step 2: Create an OpenShift Route for Blue/Green testing
- Step 3: Review BIG-IP configuration
- Step 4: Test the application
- Step 5: Generate some request traffic
- Step 6: Review the BIG-IP configuration
- Step 7: Cleanup deployed resources

Step 1: Deploy version 1 and version 2 of demo application and their associated Services

From ose-master, review the following deployment: f5-demo-app-bg-deployment.yaml

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: node-blue
  namespace: f5demo
spec:
  replicas: 1
  template:
    metadata:
      labels:
        run: node-blue
    spec:
      containers:
      - image: "chen23/f5-demo-app"
        env:
        - name: F5DEMO_APP
          value: "website"
        - name: F5DEMO_NODENAME
          value: "Node Blue (No SSL)"
        - name: F5DEMO_NODENAME_SSL
          value: "Node Blue (SSL)"
        - name: F5DEMO_COLOR
          value: "0000FF"
        - name: F5DEMO_COLOR_SSL
          value: "0000FF"
        imagePullPolicy: IfNotPresent
```

```yaml
        name: node-blue
        ports:
        - containerPort: 80
        - containerPort: 443
          protocol: TCP

---

apiVersion: v1
kind: Service
metadata:
  name: node-blue
  labels:
    run: node-blue
  namespace: f5demo
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
    name: http
  - port: 443
    protocol: TCP
    targetPort: 443
    name: https
  type: ClusterIP
  selector:
    run: node-blue

---

apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: node-green
  namespace: f5demo
spec:
  replicas: 1
  template:
    metadata:
      labels:
        run: node-green
    spec:
      containers:
      - image: "chen23/f5-demo-app"
        env:
        - name: F5DEMO_APP
          value: "website"
        - name: F5DEMO_NODENAME
          value: "Node Green (No SSL)"
        - name: F5DEMO_COLOR
          value: "99FF99"
        - name: F5DEMO_NODENAME_SSL
          value: "Node Green (SSL)"
        - name: F5DEMO_COLOR_SSL
          value: "00FF00"
        imagePullPolicy: IfNotPresent
        name: node-green
```

```
        ports:
        - containerPort: 80
        - containerPort: 443
          protocol: TCP

---

apiVersion: v1
kind: Service
metadata:
  name: node-green
  labels:
    run: node-green
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
    name: http
  type: ClusterIP
  selector:
    run: node-green
```

Now that you have reviewed the Deployment, you need to actually create it by deploying it to OpenShift by using the **oc create** command:

```
[root@ose-mstr01 tmp]# oc create -f f5-demo-app-bg-deployment.yaml -n f5demo
deployment "node-blue" created
service "node-blue" created
deployment "node-green" created
service "node-green" created
```

Step 2: Create OpenShift Route for Blue/Green Testing

The basic Route example from the previous exercise only included one path. In order to support Blue/Green application testing, a Route must be created that has two paths. In OpenShift, the second (and subsequent) path is defined in the **alternateBackends** section of a Route resource.

From ose-master, review the following Route: f5-demo-app-bg-route.yaml

```
apiVersion: v1
kind: Route
metadata:
  labels:
    name: f5-demo-app-bg-route
  name: f5-demo-app-bg-route
  namespace: f5demo
  annotations:
    # Specify a supported BIG-IP load balancing mode
    virtual-server.f5.com/balance: least-connections-node
    virtual-server.f5.com/health: |
      [
        {
          "path": "mysite-bg.f5demo.com/",
          "send": "HTTP GET /",
          "interval": 5,
          "timeout": 10
        }
```

```
      ]
spec:
  host: mysite-bg.f5demo.com
  port:
    targetPort: 80
  to:
    kind: Service
    name: node-blue
    weight: 20
  alternateBackends:
  - kind: Service
    name: node-green
    weight: 10
```

Note: How the Route resource refers to two different services: The first service is for the **Blue** application with a weight of 20 and the second service is for the **Green** application with a weight of 10.

Attention: *Knowledge Check: How many requests will the **Blue** application receive relative to the **Green** application?*

Now that you have reviewed the Route, you need to actually create it by deploying it to OpenShift by using the **oc create** command:

```
[root@ose-mstr01 module3]# oc create -f f5-demo-app-bg-route.yaml
route "f5-demo-app-bg-route" created
```

Verify that the Route was successfully creating by using the OpenShift **oc get route** command. Note that, under the **SERVICES** column, the two applications are listed along with their request distribution percentages.

```
[root@ose-mstr01 tmp]# oc get route -n f5demo
NAME                   HOST/PORT              PATH      SERVICES                        ␣
↪   PORT       TERMINATION    WILDCARD
f5-demo-app-bg-route   mysite-bg.f5demo.com   /         node-blue(66%),node-green(33
↪%)    80                     None
```

Attention: *Knowledge Check: What would the Route percentages be if the weights were 10 and 40?*

Step 3: Review BIG-IP configuration

In this step, you will examine the BIG-IP configuration for changes made by the Container Connector after the the OpenShift Route was deployoed.

Using the Chrome web browser, navigate to **Local Traffic –> Pools –> Pool List** and change the partition to **ocp** using the dropdown in the upper right.

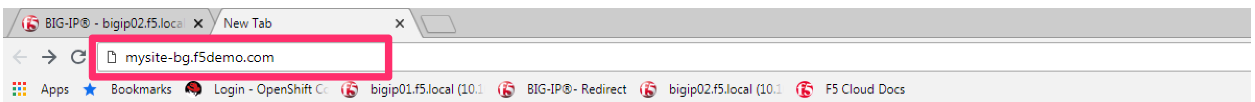
Note: There are two pools defined: one pool for the **Blue** application and a second pool for the **Green** application. Additionally, the Container Connector also creates an iRule and a datagroup that the BIG-IP uses to distribute traffic based on the weights assigned in the OpenShift Route.

Step 4: Test the application

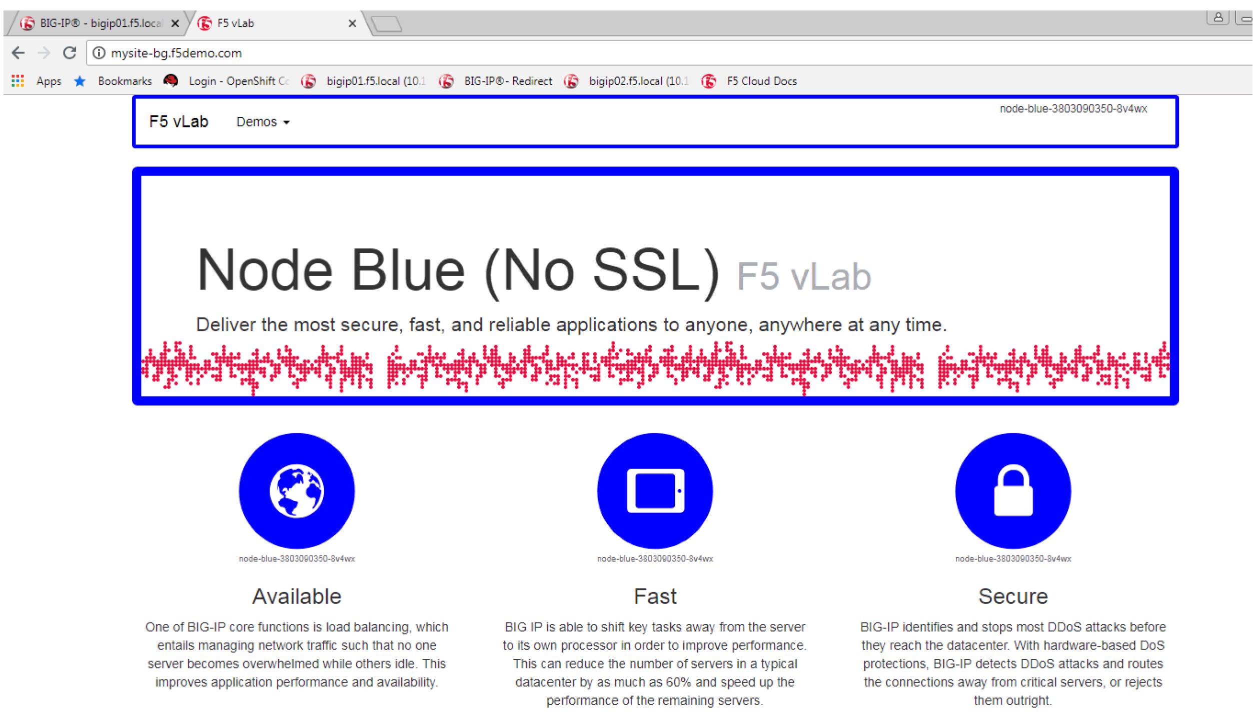
In this step, you will use the Chrome browser to access blue and green applications you previously deployed.

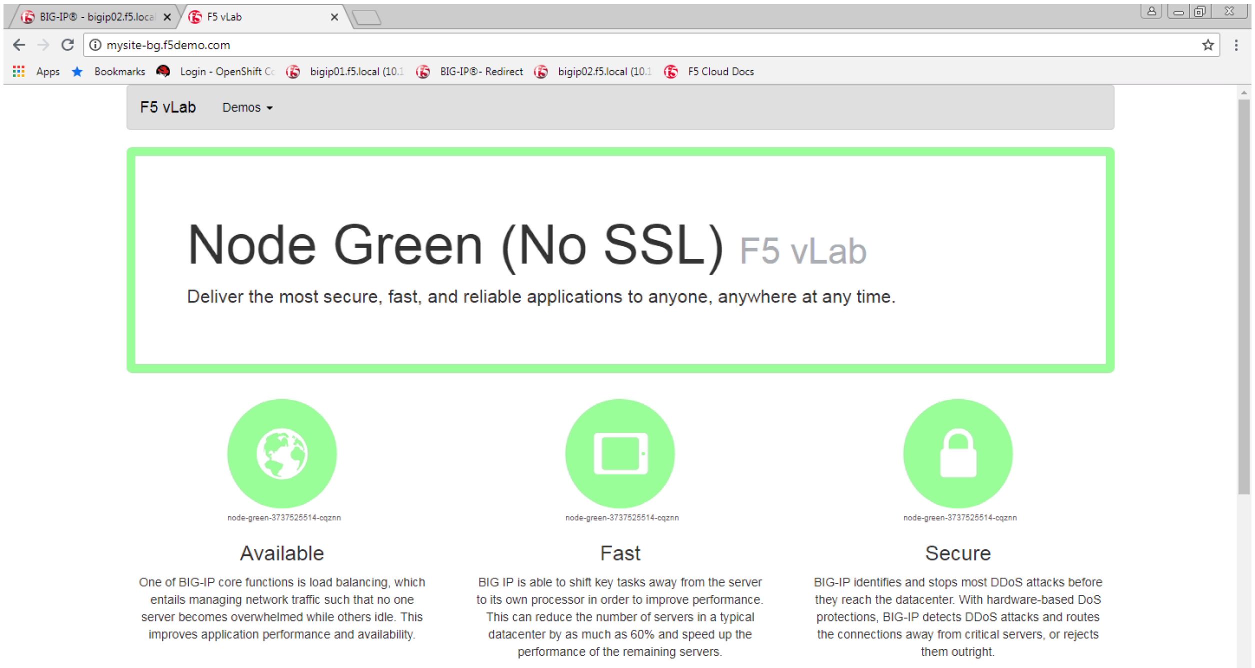
Because the Route resource you created specifies a hostname for the path, you will need to use a hostname instead of an IP address to access the demo application.

Open a new browser tab and enter the hostname **mysite-bg.f5demo.com** in to the address bar:



Refresh the browser periodically and you should see the web page change from the **Blue** application to the **Green** application and back to the **Blue** application as noted by the colors on the page.





Step 5: Generate some request traffic

As the number of requests increases, the relative number of requests between the **Blue** application and the **Green** application begins to approach the weights that have been defined in the OpenShift Route.

In this step, you will use the Linux **curl** utility to send a large volume of requests to the application.

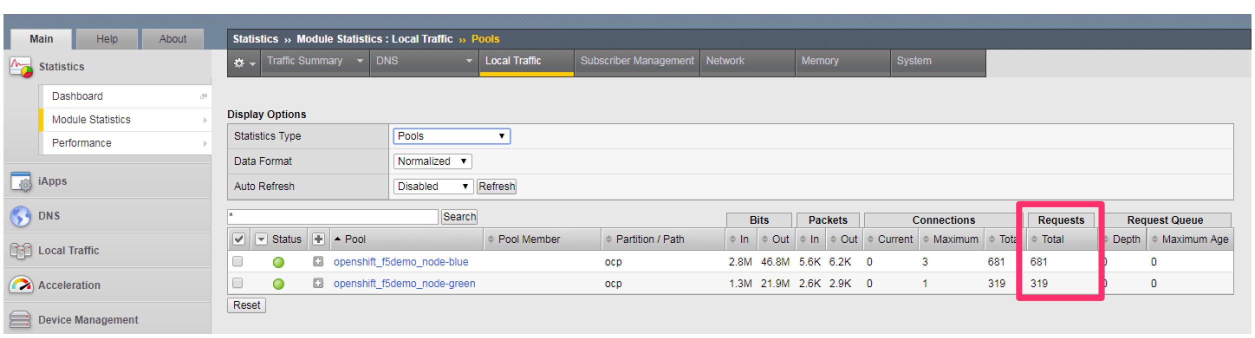
From the ose-master server, run the following command to make 1000 requests to the application:

```
[root@ose-mstr01 ~]# for i in {1..1000}; do curl -s -o /dev/null http://mysite-bg.
↪f5demo.com; done
```

Step 6: Review the BIG-IP configuration

In the previous step, you used the **curl** utility to generate a large volume of requests. In this step, you will review the BIG-IP pool statistics to see how the requests were distributed between the **Blue** application and the **Green** application.

Using the Chrome web browser, navigate to **Local Traffic -> Pools -> Statistics** and change the partition to **ocp** using the dropdown in the upper right.



Step 7: Cleanup deployed resources

In this step, you will remove the Deployment, Service and Route resources you created in the previous steps using the OpenShift **oc delete** command.

From ose-master server, run the following commands:

```
[root@ose-mstr01 tmp]# oc delete -f f5-demo-app-bg-route.yaml -n f5demo
route "f5-demo-app-bg-route" deleted

[root@ose-mstr01 tmp]# oc delete -f f5-demo-app-bg-deployment.yaml -n f5demo
deployment "node-blue" deleted
service "node-blue" deleted
deployment "node-green" deleted
service "node-green" deleted
```

Expected time to complete: **3 hours**

6.4 Lab Setup

In the environment, there is a three-node OpenShift cluster with one master and two nodes. There is a pair of BIG-IPs setup in an HA configuration:

Hostname	IP-ADDR	Credentials
jumpbox	10.10.200.199	user/Student!Agility!
bigip01	10.10.200.98	admin/admin root/default
bigip02	10.10.200.99	admin/admin root/default
ose-mstr01	10.10.199.100	root/default
ose-node01	10.10.199.101	root/default
ose-node02	10.10.199.102	root/default