User Manual



Contents

1. Introduction 3
   1. Getting Started 5
2. Setting Up a Cluster 8
   1. Setting up a cluster and verifying members of the cluster. 8
      1. Preparing Cloud-Config file 8
      2. Generating etdc discovery token 9
3. Verifying Service High Availability and Node High availability in CoreOS cluster 13
   1. Preparing and Running services 13
   2. Verifying High Availability 19
4. Conclusion 22

# Introduction

In this test drive we are going to see how CoreOS is going to maintain high-availability of the service and node in a cluster. In this test drive we are going to spin up the vms and use cloud config file to form cluster. We are going see how etcd and fleet tools help us to manage the cluster.

We are going to run two sample services using Docker container. One is simple service which will run globally and the other service is Apache service which will start on any of the node in the cluster.

Once the node (where Apache service is running) is down the other node automatically picks up the service and runs it to maintain high availability.

As part of this test drives four public IPs , three NICs, one Vnet, one availability set, one load balancer (internal),NSG and three virtual machines with CoreOS stable (Container Linux 1235.6.0) operating system are deployed.

Complete details of the use case will be presented in the coming sections of the user manual. Let us understand few basics about the platform.

CoreOS

CoreOS is a minimal and powerful operating system with the key idea that you never ever need to worry about system updates. At its core, CoreOS runs a minimal system kernel and ships with out-of-the-box support for container systems like [Docker](https://www.docker.com/), rkt and [nspawn](http://www.freedesktop.org/software/systemd/man/systemd-nspawn.html).

With CoreOS, we’re able to spin up a cluster easily and the entire system is built for high-availability clusters. Security patches and system updates are applied automatically. The required system reboots are handled automatically with the help of **etcd**. By default, only one machine at a time will reboot within the cluster.

CoreOS takes the high-availability approach to a new level and utilizes components like **etcd**, **fleet** and **docker** to pursue its goal.

etcd

Running applications within a cluster requires a consistent set of global data on each cluster node. Additionally, the aspect of service discovery has an important role within a cluster and therefore, a tool called etcd is developed.

etcd is a distributed key-value store designed for high-availability and cluster consensus. Its idea is to provide consistent reads and writes within the cluster even in case of host failure. Data is distributed and stored across all machines within the cluster. etcd uses the Raft consensus algorithm and is robust against leader failures. If the leader fails, it automatically performs leader election and makes one follower the leader.

CoreOS uses etcd for automatic host detection for a cluster. Each CoreOS cluster has its own cluster identifier and etcd handles the connection of new hosts into the cluster. Further, etcd is targeted towards service discovery to let applications announce itself to etcd.

fleet

We mentioned earlier that, we need to change our thinking from single virtual machine to clusters when using CoreOS. This also includes the change in thinking about how to start services on our cluster. For single CoreOS VM we can use systemd to start services and for CoreOS clusters, we have a tool called fleet.

fleet is a cluster-level service scheduler and enables cluster orchestration. With fleet you can manage your cluster of machines as if they share a single init system. Imagine fleet as follows: instead of starting services on a specific machine, you use fleet to submit services into your cluster and the cluster manager (fleet itself) decides on the machine to start it.

CoreOS ships with fleet and the command line tool fleetctl for management purposes. With the help of fleetctl, you’re able to get the state and information of your services submitted and running within your cluster. Besides the fact that you start applications (called services) on your cluster, you can define additional service conditions. A condition can be the placement of services on a specific host next to another service.

fleet uses systemd to start processes on machines. The processes require a defined systemd unit file with custom options. Once a service configuration is passed into fleet, you can manage them at cluster level from any other cluster machine.

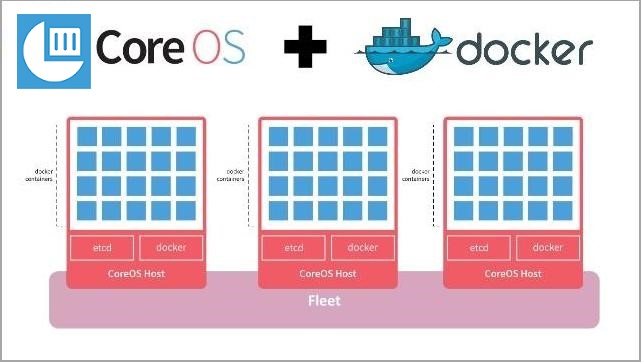
Docker

CoreOS itself follows the philosophy of providing a minimal kernel and use containers as its “package manager” for applications. Every service and application runs isolated from one another. CoreOS currently leverages Docker as its platform to run applications on clusters.

Docker itself is a containerization platform that allows you to create stand-alone containers of any application. Containers run (long lived) processes in isolation which makes it more convenient to distribute applications throughout the cluster.

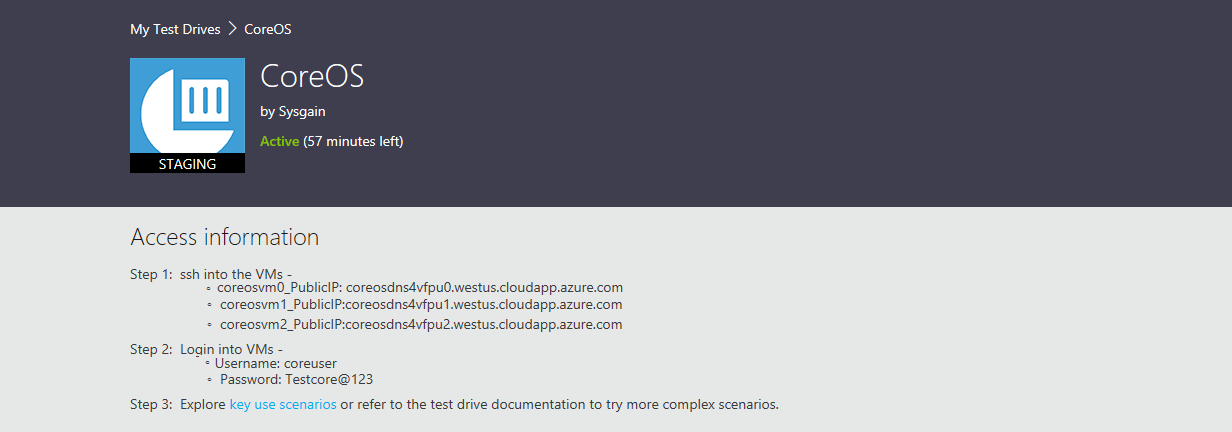
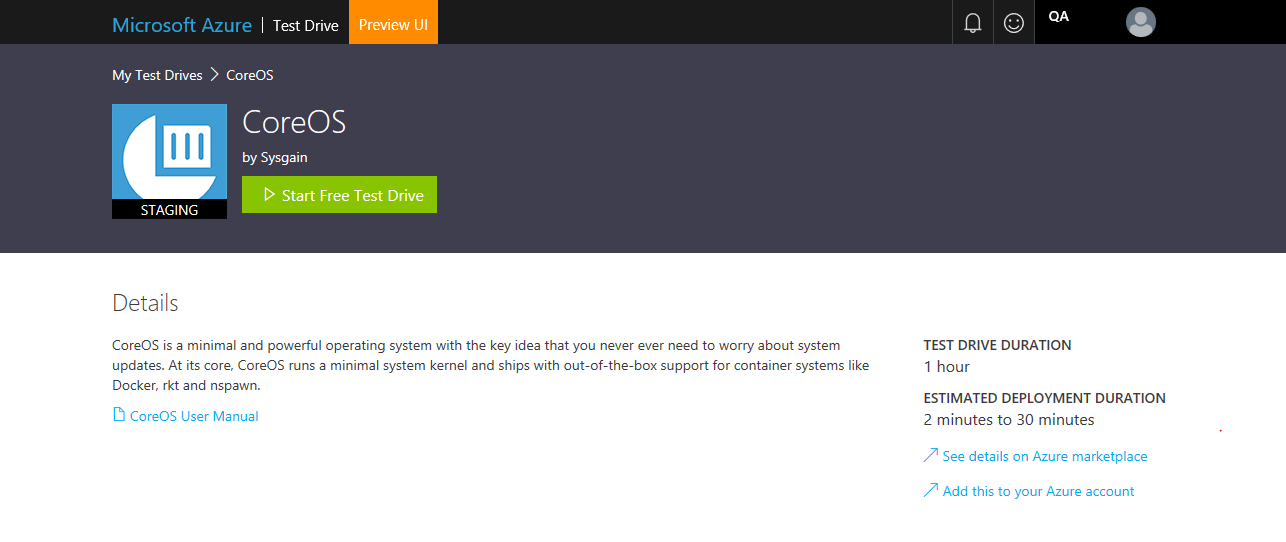
Docker allows you to package and distribute your application including all required dependencies. CoreOS uses Docker containers to run applications on top of the minimal host system. Multiple applications can run side-by-side on the same host. Running applications within containers makes cluster orchestration a lot easier. They can be moved within the cluster without the need to install application dependencies on every host.

Integration of the above components can be depicted as below.

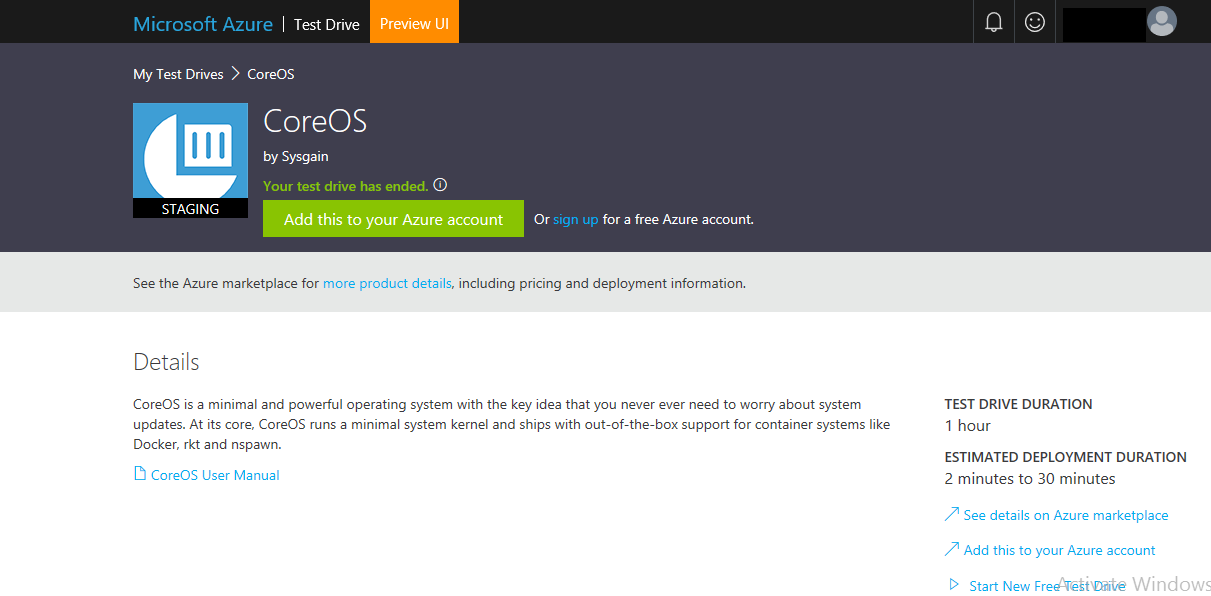


# Getting Started

In test drive portal page click on CoreOS and in the CoreOS test drive page click on **Start Free Test Drive**, test drive will be started and the access information for the test drive will be displayed as shown below.



Once we done we can end the test drive or once time is up test drive will end automatically.



After test drive provisioning is complete, login credentials are provided in the test drive launch page as well as by e-mail. With the username, password, and client SSH url provided, SSH into the Virtual machine from your terminal or SSH client.

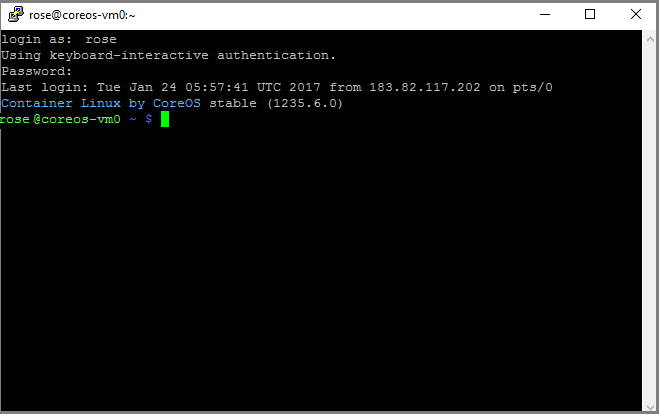
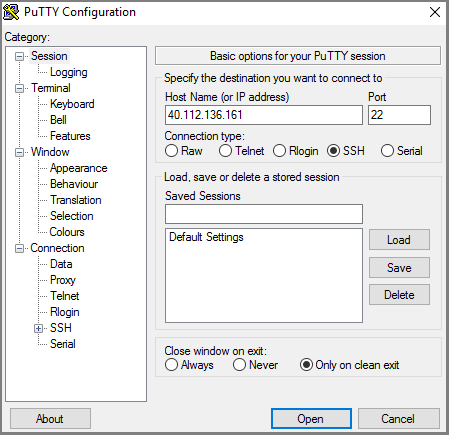
Download the putty from here:

<http://www.putty.org/>

Login into the virtual machines using the putty as shown below. You should use the login information provided under the Access Information.

We use either IP address or the DNS name to login into the virtual machine using putty. In this test drive we will be having three virtual machines which will form a cluster. Assume that the virtual machine names as “coreos-vm0”, “coreos-vm1” and “coreos-vm2”,

Let’s login into first virtual machine i.e. “coreos-vm0” using putty for windows and terminal for mac OS by using DNS or public IP



# Setting up a Cluster

# Setting up a cluster and verifying members of the cluster

2.1.1 Preparing Cloud-Config file

# Once we are inside the virtual machine, we create cloud-config file named user\_data.yml. Using vi editor as shown below.

sudo vi user\_data.yml

Sample user\_data Yaml script is provided below, copy and paste the same in the VI editor and replace the <token> in the discovery URL with newly generated token.

filename: user\_data.yml

#cloud-config

coreos:

etcd2:

discovery: https://discovery.etcd.io/<token>

# multi-region and multi-cloud deployments need to use $public\_ipv4

advertise-client-urls: http://$public\_ipv4:2379

initial-advertise-peer-urls: http://$private\_ipv4:2380

# listen on both the official ports and the legacy ports

# legacy ports can be omitted if your application doesn't depend on them

listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001

listen-peer-urls: http://$private\_ipv4:2380,http://$private\_ipv4:7001

fleet:

public-ip: $public\_ipv4

flannel:

interface: $public\_ipv4

units:

# To use etcd2, comment out the above service and uncomment these

# Note: this requires a release that contains etcd2

- name: etcd2.service

command: start

- name: fleet.service

drop-ins:

- name: norestart.conf

content: |

[Service]

Restart=no

command: start

# 2.1.2. Generating etdc discovery token

To spin up a cluster easily, etcd uses a discovery token. You can obtain a discovery token with a cluster size of 3 when just using the following URL. The token value is the alphanumeric string at the end of the returned URL.

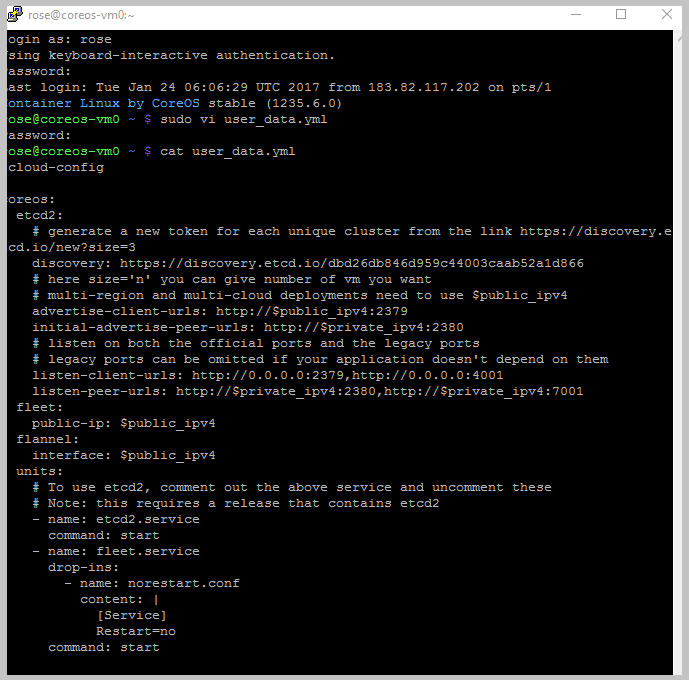
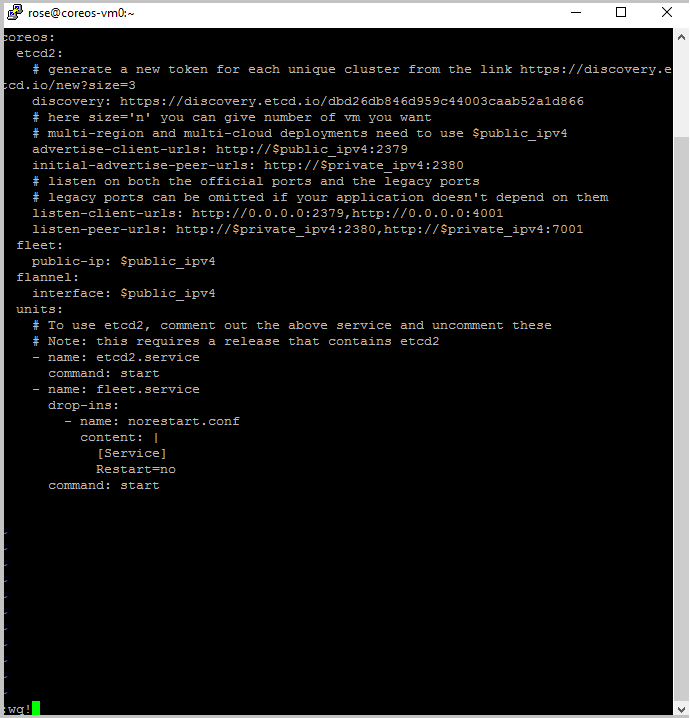
<https://discovery.etcd.io/new?size=3>

Pass the size of your cluster as a query parameter to the endpoint. Use? Size=n and replace n with your desired cluster size. In this guide, we’ll use a cluster size of 3.

Once we generate the discovery token replace it in the script in place of <token> and save the file using *:wq* or *:wq!* command as shown in the screenshot below.

***Note:*** VI editor operates in two modes “insert mode” and “command mode”. Once we open the VI editor to enter the data we need to press “I” key in the key board, now VI editor will be in “insert” mode. Once we enter the data in file we need to press “Esc” key to go to command mode and then enter *:wq* command to save the file and exit from the VI editor.

To verify the content we cat use “cat” command (Syntax: cat filename).

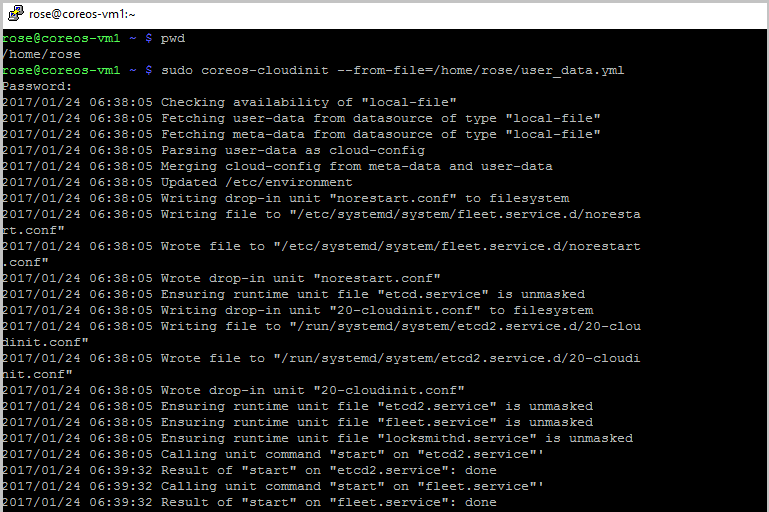


Once we verify everything is fine, we need to execute the following command to add the current host machine to the cluster.

sudo coreos-cloudinit --from-file*=<full path of the cloud-config file>*

ex: sudo coreos-cloudinit --from-file=/home/rose/user\_data.yaml

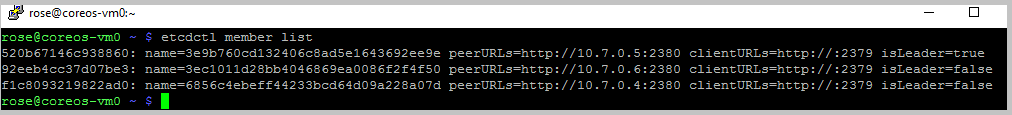
The above command starts all the required services like etdc and fleet.



We should copy the same file in other two virtual machines (coreos-vm1 and coreos-vm2) “user\_data.yml” file in the other virtual machines. And execute the above command to form the cluster.

Once we execute the above command in all the three machines, we can verify the cluster members using the below command in any one of the virtual machine.

etcdctl member list



# Verifying Service High Availability and Node High availability in CoreOS cluster

# Preparing and Running services

Let us create a sample global service called “helloglobal.service” in the “coreos-vm0” machine. Copy paste the below code in VI editor and save it with the name “helloglobal.service”

***filename: helloglobal.service***

[Unit]

Description=My Service

After=docker.service

[Service]

TimeoutStartSec=0

ExecStartPre=-/usr/bin/docker kill hello

ExecStartPre=-/usr/bin/docker rm hello

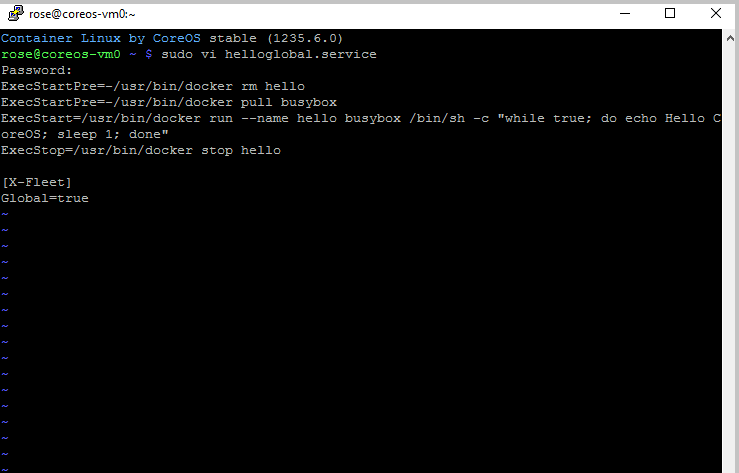
ExecStartPre=-/usr/bin/docker pull busybox

ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"

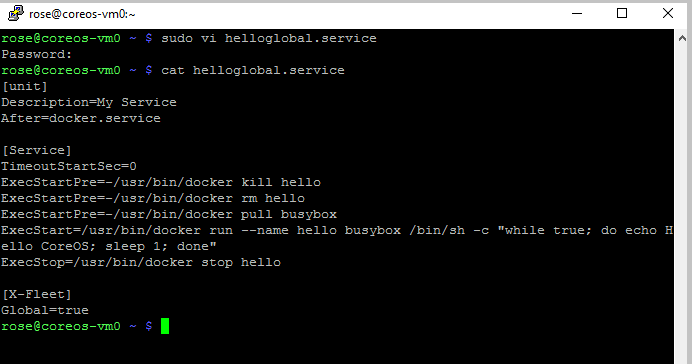
ExecStop=/usr/bin/docker stop hello

[X-Fleet]

Global=true

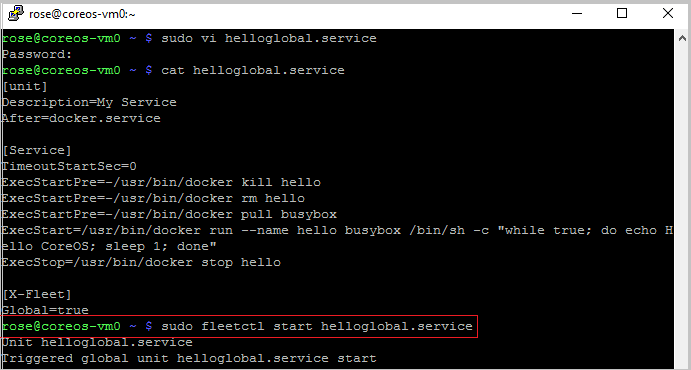


The same can be verified using the cat command as shown below.



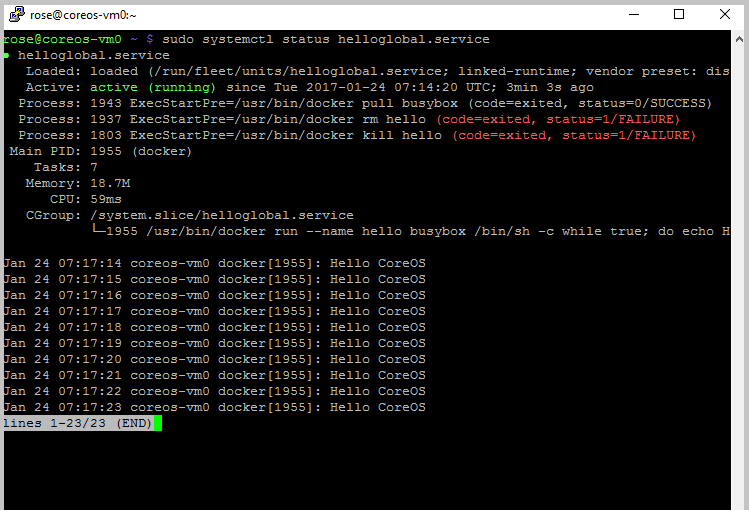
Start the service using the fleet tool as shown below by using the following command

sudo fleetctl start helloglobal.service



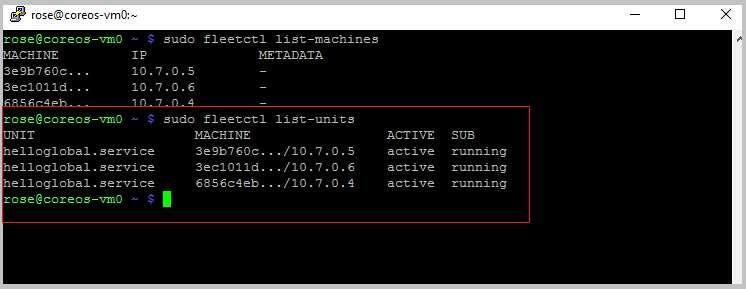
We can check the status of the service using the following command in any one of the virtual machines, as it is a global service it will run on all the machines.

sudo systemctl status helloglobal.service



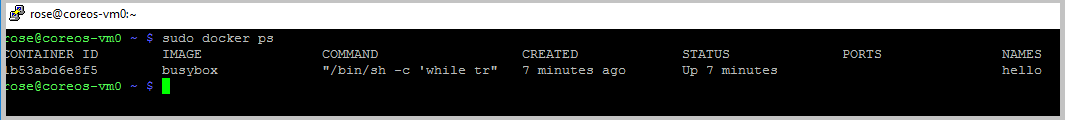
We can verify the service status in all the three machines as shown below

sudo fleetctl list-units



We can verify the Docker container using the following command and the out of the same is shown below.

sudo docker ps



Now we will create another service, this is Apache web service. The name of the service is [apachet@.service](mailto:apachet@.service). Copy and paste the following code in the VI editor and create a file with the name “[apachet@.service](mailto:apachet@.service)”. Follow the below screens for creating and verifying the file.

[Unit]

Description=Apache web server service on port %i

# Requirements

Requires=etcd2.service

Requires=docker.service

Wants=apachet-discovery@%i.service

# Dependency ordering

After=etcd2.service

After=docker.service

Before=apachet-discovery@%i.service

[Service]

# Let processes take awhile to start up (for first run Docker containers)

TimeoutStartSec=0

# Change killmode from "control-group" to "none" to let Docker remove

# work correctly.

KillMode=none

# Get CoreOS environmental variables

EnvironmentFile=/etc/environment

# Pre-start and Start

## Directives with "=-" are allowed to fail without consequence

ExecStartPre=-/usr/bin/docker kill apachet.%i

ExecStartPre=-/usr/bin/docker rm apachet.%i

ExecStartPre=-/usr/bin/docker pull coreos/apache

ExecStart=/usr/bin/docker run --name apachet.%i -p ${COREOS\_PUBLIC\_IPV4}:%i:80 coreos/apache /usr/sbin/apache2ctl -D FOREGROUND

Restart=always

RestartSec=30s

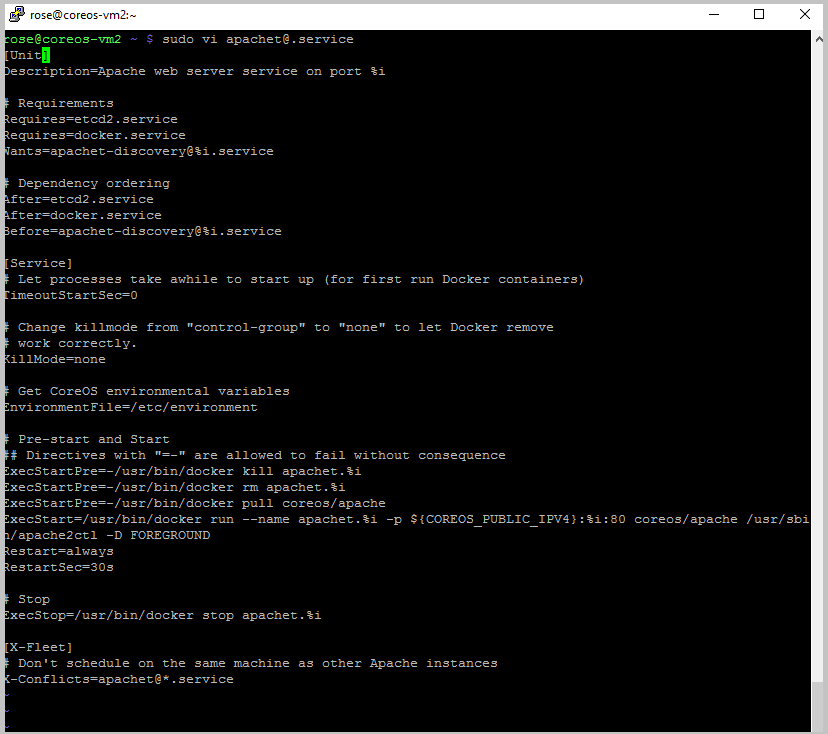
# Stop

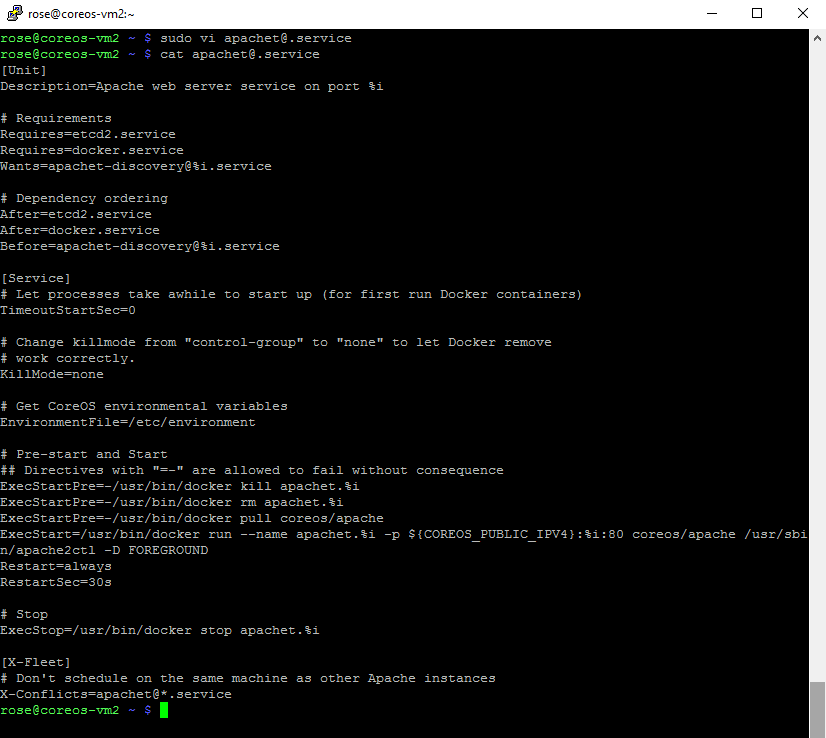
ExecStop=/usr/bin/docker stop apachet.%i

[X-Fleet]

# Don't schedule on the same machine as other Apache instances

X-Conflicts=apachet@\*.service

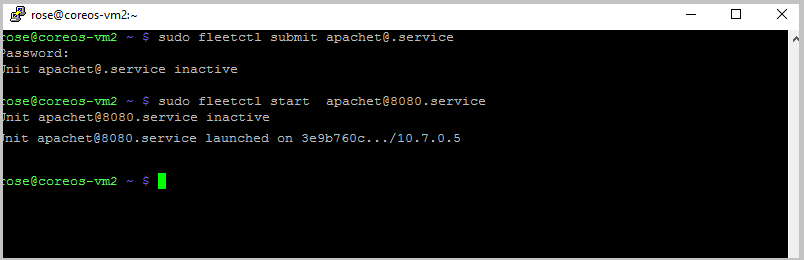




Once we done with the editing and verifying the file. We can submit the service and start the service as shown below. While starting the service we need to mention the port number. In our case we are starting the Apache web service at port number 8080.

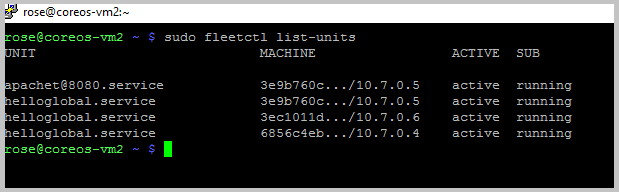
sudo fleetctl submit [apachet@.service](mailto:apachet@.service)

sudo fleetctl start [apachet@8080.service](mailto:apachet@8080.service)



We can verify the service in all the machines by using the following command

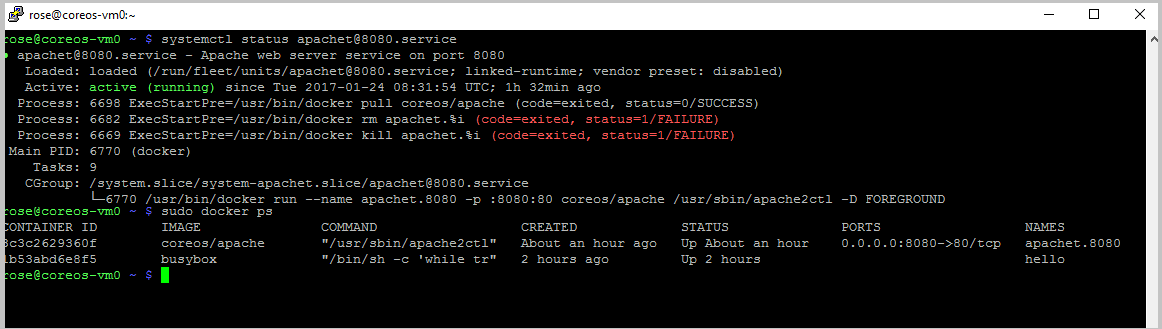
sudo fleetctl list-units



We can go to the virtual machine where the Apache web service was started and we can verify the status of the service. We can verify the Docker containers for both the services using the docker ps command as shown below.

systemctl status [apachet@8080.service](mailto:apachet@8080.service)

sudo docker ps



# Verifying High Availability

We can verify the service high availability and node high availability. First let’s look high availability of the service by stopping the docker container.

sudo docker stop apachet.8080

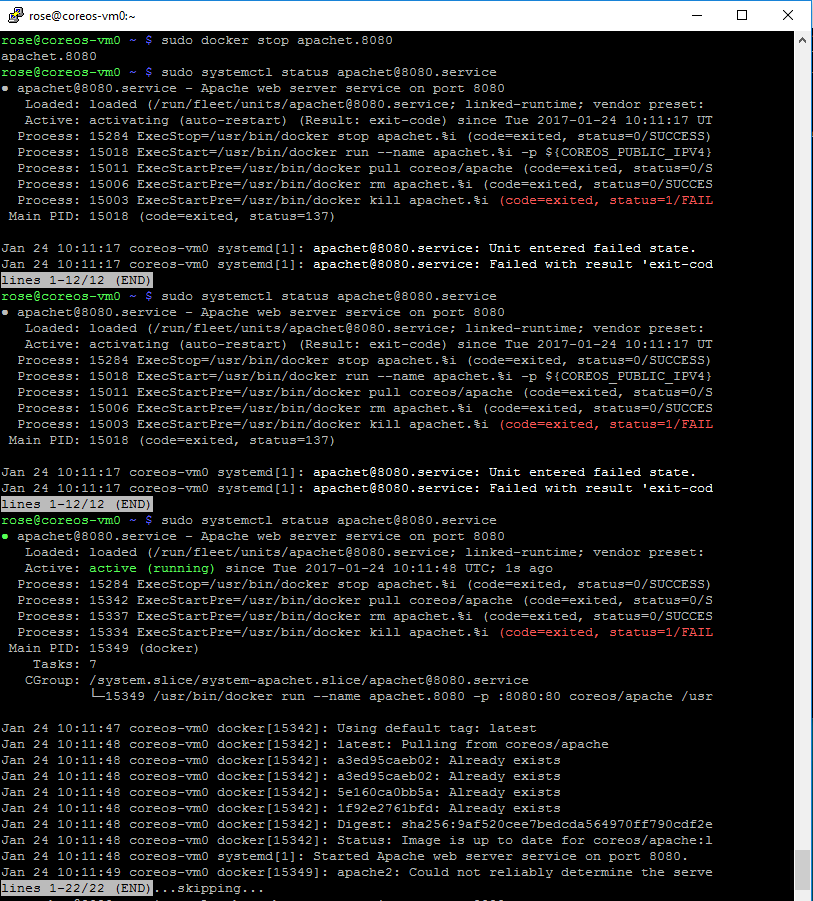
When we stop the docker container the service will not is available but as we mentioned *Restart=always and RestartSec=30s*. The service automatically starts after 30s.

After executing the above command we can check the status of the apache service by using the below command.

sudo systemctl status apachet@8080.service

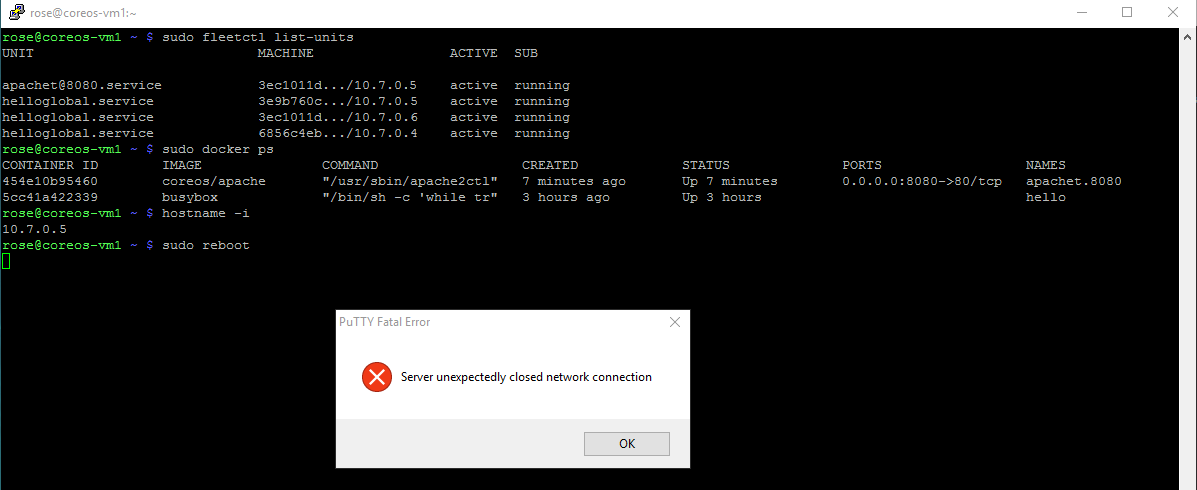
Keep checking the status it will initially shows the status as activating, after 30 seconds it shows the status as active(running).

The following screenshot will depict the same. This way we can achieve the high availability of the service.



Now let us look the node high availability, to demonstrate this we need to go the virtual machine where the apache service is running and reboot the system a shown below.

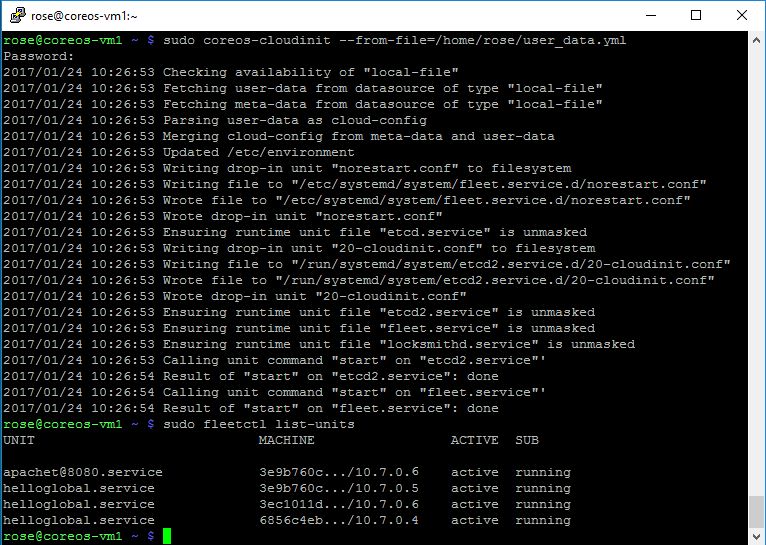
Note: If we use putty application right click on the top and click on restart session then login into the machine using the same credential provided in the beginning.



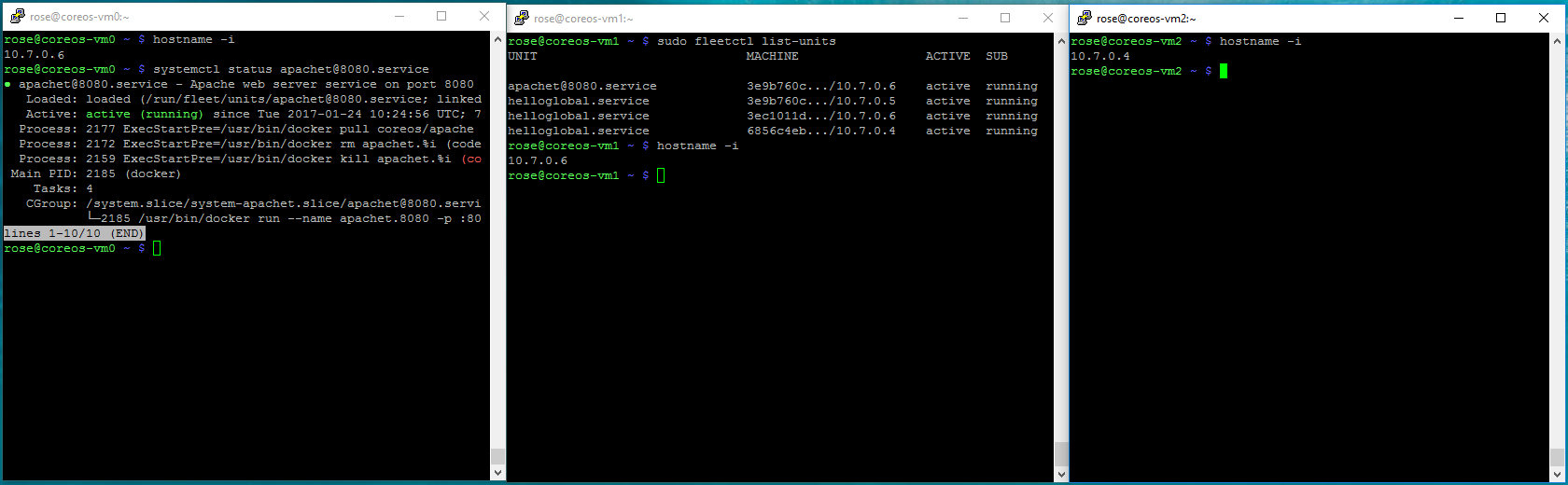
Once the system is up we can run the following command to start all the services in the machine.

sudo coreos-cloudinit --from-file*=<full path of the cloud-config file>*

ex: sudo coreos-cloudinit --from-file=/home/rose/user\_data.yaml



Now if we observe initially the service was started in 10.7.0.5 and as we killed that node the service is automatically moved to another node that is 10.7.0.6 in our example. We can go to that node and check the status of the service as shown below.



# Conclusion

In this test drive we have seen how the CoreOS is going to provide high availability and the use of tools like fleet and etdc.

We also seen how to run the Docker containers in CoreOS.