

Welcome to Docker! This guide will take you through the various mini-tutorials to help you understand Docker.

## Setup

The first step is to ensure that you install Docker on your machine. Since this tutorial was first written, Docker is now available for installation on a number of Linux distributions and also available for Windows and Mac operating systems.

Please visit <https://docs.docker.com/engine/installation/#supported-platforms> and select the Docker CE for installation on your platform. For the rest of the series, I shall be using Docker for Mac.

Once you have the setup done, I suggest that you test out your installation via the command given below:

Let's try the hello-world example image. Give the following command at the prompt:

**\$ docker run hello-world**

This should download the very small hello-world image and print a "Hello from Docker" message.

Understand what we are running!

A lot of stuff happened behind the scenes to get the Hello World running. While we may not go into the specifics of all that for now, these are roughly the steps:

You used the docker client application via the docker command.

You gave the command run hello-world to the docker client application.

In other words, you told the docker client to create an instance of a Docker Image titled hello-world.

Where is this Docker Image? On your local system ? Somewhere on the Internet ? What is going on ?

Well, the default behaviour is a sensible one i.e. the docker client looked for an Image titled hello-world in your local repository i.e. on your local machine. It did not find it (obviously)—so it went to the Internet and hit a URL at Docker Registry ( a public repository of Docker Images hosted by the company behind Docker). It found it there (I cheated ... since I knew the name "hello-world" is one of the existing images out there. But you get the point).

Once found, it started to download the Image (all its layers) and once it was present locally, it launched an instance (Container) based on that image.

The default command was then executed to print out some message on the console, which is what you saw.

Try out the following Docker commands

Since the image “hello-world” is now present locally, it should be available in our local repository.

Try out :

```
$ docker images
```

and verify that you see the image listed there.

Try launching another container based on the same image. Give the following command:

```
$ docker run hello-world
```

It should print out the same message that you saw earlier.

The command `$ docker ps` gives you a list of running containers. Try it out and see if you can find any running.

Try out:

```
$ docker ps --all
```

and see if they are visible now.

Use ‘—help’ option. For e.g. `docker ps—help`

Even if you do not understand everything at this point in time, that is fine. These commands were meant to get you to start thinking of the typical operations you will need to understand/execute while dealing with Images / Containers.

## Part 2 Docker Basic Commands

Now that you have installed Docker, it is time to try out the basic commands that you can do via the docker client program.

### Recap

Recollect that the Docker toolset consists of:

1. Docker Daemon
2. Docker Client
3. Docker Hub

Now, when we work with the docker client, what is happening is that the commands are being sent to the Docker Daemon, which then interprets the command and executes it for you.

### docker client help

The docker client can understand several commands. And in this hands-on step, we are going to take a look at various commands that you will primary use while running docker.

To get help at any point in time try out the following command:

**\$ docker help**

This will give you a full listing of commands that the docker client can understand. Take some time to go through this. Most of the commands are self-explanatory and are typical ones that you will use while dealing with containers.

At any point in time, if you need more help on any COMMAND, you can get that via the following:

**\$ docker COMMAND --help**

## Initial List of commands

The next few sections will take you through various commands and you should try out every single command. Before you try any of that, ensure that Docker has been started. You can visit the terminal and give the following command.

### \$ docker version

This will show you the current docker version.

Fun fact : Docker is written in [Go language](#). The language is similar to C and has been the favorite of developers writing infrastructure software. Pick it up now!

### \$ docker info

This will show you several pieces of information about the OS.

## Run a few Unix Commands/Utilities

Let's understand what we are doing here now. We are on a Windows machine and we wish to run a few Unix commands/utilities to get a bit familiar with them.

So this is what the steps look like with Docker now:

1. There is a useful Docker Image called busybox (just like we had hello-world) that someone has already created for Docker.
2. We will use the docker run command to run a container i.e. create an instance of that image.
3. By running, what we want to do is to go inside that container and run a few commands there.

Let us check some steps i.e. docker commands that we will run—not all are necessary but we are doing this to get you a bit familiar with the commands. We will be looking at some of these commands in more detail in subsequent sessions.

### \$ docker search busybox

This command will search the online Docker registry for a Image named busybox. On my machine, the output shown is as follows (only the top few rows are shown):

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker search busybox
```

NAME	DESCRIPTION	STARS	OFFICIAL
busybox	Busybox base image.	1164	[OK]
progrum/busybox		66	
hypriot/rpi-busybox-httpd	Raspberry Pi compatible Docker Image with ...	39	
radial/busyboxplus	Full-chain, Internet enabled, busybox made...	17	
hypriot/armhf-busybox	Busybox base image for ARM.	8	
armhf/busybox	Busybox base image.	4	
arm32v7/busybox	Busybox base image.	3	
s390x/busybox	Busybox base image.	2	
aarch64/busybox	Busybox base image.	2	
prom/busybox	Prometheus Busybox Docker base images	2	
onsi/grace-busybox		2	
armel/busybox	Busybox base image.	2	
p7ppc64/busybox	Busybox base image for ppc64.	2	
i386/busybox	Busybox base image.	1	
ppc64le/busybox	Busybox base image.	1	
flynn/busybox	Busybox from Ubuntu 13.10 with libc	1	
arm32v6/busybox	Busybox base image.	1	
spotify/busybox	Spotify fork of <a href="https://hub.docker.com/_/b...">https://hub.docker.com/_/b...</a>	1	
joeshaw/busybox-nonroot	Busybox container with non-root user nobody	1	
concourse/busyboxplus		0	
cfgarden/garden-busybox		0	
amd64/busybox	Busybox base image.	0	
arm64v8/busybox	Busybox base image.	0	
yauritux/busybox-curl	Busybox with CURL	0	
trollin/busybox		0	

Let us understand the output here, by paying attention to the columns:

1. The first column is NAME and it gives you the name of the Docker image.
2. The second column is DESCRIPTION and it is obvious what that means.
3. The next column is STARS and if you noticed the list of images that matched the search term Docker have been listed in the descending order of the number of people who have starred the project. This is a very useful indicator of the popularity/correctness of the Image. Often if confused among which Docker image to go with, I usually pick the one with the most STARS.

The first column, to reiterate, was the NAME of the Docker image. This is a unique name and you must use this name for some of the commands given below.

So, let's say that we are fine with the busybox image name and now want to create an instance (Container) of this image. To do that, all we need to do is use the docker runcommand as given below:

**\$ docker run -t -i busybox**

The run command does something interesting and this will help you understand the Docker architecture, which we have seen earlier. The run command does the following:

1. It checks if you already have a busybox image in your local repository.

2. If it does not find that (which will be the case first time), it will pull the image from the Docker hub. Pulling the image is similar to downloading it and it could take a while to do that depending on your internet connection.
3. Once it is pulled successfully, it is present in your local repository and hence it is then able to create a container based on this image.
4. We have provided `-i -t` as the parameters to the run command and this means that it is interactive and attaches the tty input.

Note: If the image was present locally, it would have directly run the container for you.

On successful launch of the container, you will be led into the bash shell for busybox. To keep it simple for Windows users, we are now logged into the busybox container and are at the command prompt, as shown below:

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker run -t -i busybox
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
0ffadd58f2a6: Pull complete
Digest: sha256:bbc3a03235220b170ba48a157dd097dd1379299370e1ed99ce976df0355d24f0
Status: Downloaded newer image for busybox:latest
/ # █
```

You can run a few commands as shown below:

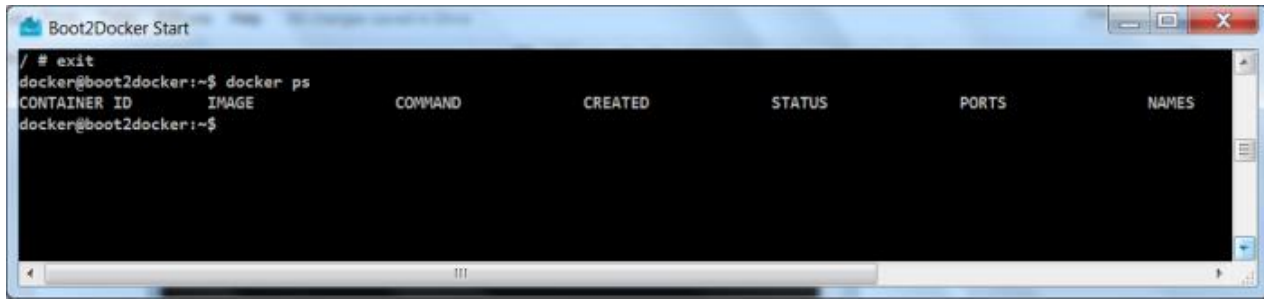
```
/ # ls
bin  dev  etc  home  proc  root  sys  tmp  usr  var
/ # whoami
root
/ # date
Thu Dec 14 02:59:44 UTC 2017
/ # uptime
02:59:46 up 34 min,  0 users,  load average: 0.00, 0.00, 0.00
/ #
```

Note from the prompt that you are now inside the container. You can exit the container by simply giving the exit command.

```
/ # exit
Romin-Iranis-MacBook-Pro:~ romin-irani$ █
```

When you give the exit command, the container has stopped running. To verify that, you can give another command as given below:

## **\$ docker ps**



```
Boot2Docker Start
/ # exit
docker@boot2docker:~$ docker ps
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
docker@boot2docker:~$
```

This gives you a list of all the running containers. You will notice from the output that there are no container running.

Try out the following:

If you want to find out the containers that were running earlier but are not in the terminated state, you can use the `-a` flag for the `docker ps` command. Give it a try.

## **Get List of Docker Images**

At this point in time, if you want to know what images are already present on your docker setup locally, try the following command:

### **\$ docker images**

You will find that it has the busybox image listed.

Note the columns that the output gives (2 important ones are given below):

1. REPOSITORY
2. TAG

The REPOSITORY column is obvious since it is the name of the Image itself. The TAG is important, you will find that the TAG value is mentioned as latest. But there was no indication given by us about that.

The fact is that when we gave the following command earlier:

### **\$ docker run -t -i busybox**

We only specified the name and by default if just the IMAGE name is specified, then it gets the latest image by default. The tag value 'latest' is sort of implicitly used by the Docker client in the absence of an explicit tag value provided by you.

In other words, you could have specified it as:

### **\$ docker run -t -i busybox:latest**

Similarly, there is a clear possibility that there will be multiple versions of any image present in the Docker Hub. We will see all that in a while, but for now, keep in mind that lets say there were the following versions available of busybox:

1. Image Name : busybox , Version TAG : 1.0
2. Image Name : busybox, Version TAG : 2.0
3. Image Name : busybox, Version TAG : 3.0

We could mention the version TAG as needed:

**\$ docker run -t -i busybox:1.0**

**\$ docker run -t -i busybox:2.0**

and so on.

## Docker Containers

When you executed the docker ps command, you noticed that no container was running. This was because you exited out of the container. Which means that the container only exists as long as its parent process is running.

Now, let us do a docker ps -all command. This should show you the container that was launched a few minutes ago by you. For e.g. on my system, I see the following:

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker ps -all
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
MES ab0e37214358	busybox	"sh"	2 minutes ago	Exited (0) About a minute ago	

Notice the following columns:

- CONTAINER\_ID : A Unique ID for the container that was launched.
- IMAGE : This was the IMAGE that you launched i.e. busybox
- COMMAND : Important stuff here. This was the default command that was executed when the container was launched. If you recollect, when the container based on busybox image was launched, it led you to the Unix Prompt i.e. the Shell was launched. And that is exactly what the program in /bin/sh does. This should give you a hint that in case you want to package your own Server in a Docker image, your default command here would typically be the command to launch the Server and put it in a listening mode. Getting it ?



## Relaunch a Container

To start a stopped container, you can use the `docker start` command. All you need to do is give the Container ID to the `docker start` command.

So, look at the `docker ps -all` output and note down the `CONTAINER_ID`. On my machine, the `docker ps -all` command gives me the following output:

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker ps -all
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
MES					
ab0e37214358	busybox	"sh"	2 minutes ago	Exited (0) About a minute ago	

I note down the `CONTAINER_ID` i.e. `cfb007d616b9` and then give the following command:

### \$ `docker start ab0e37214358`

Note the `-i` for going into interactive mode. You will find that if everything went fine, the Container was restarted and you were back again at the Prompt, as given below:

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker start -i ab0e37214358
```

```
/ #
```

```
/ #
```

Type `exit` and come out of the container. We are back to where we were with no containers running.

## Attach to a running Container

Now, its time for something interesting to help us understand some more commands. We will continue with our example around `busybox` Image.

First up, we will relaunch our container without the `-i` (interactive) mode.

Give the following command:

### \$ `docker start ab0e37214358`

**ab0e37214358**

Whoops ! What happened ?

The only output that we got was the `CONTAINER ID` back.

What has just happened is that the Container has got launched and all that docker client has done is give you the Container ID back.

Give the following command to check out the current running containers (the command should be familiar to you now):

**\$ docker ps**

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS
ab0e37214358   busybox    "sh"                    6 minutes ago   Up 8 seconds
```

This gives us the output that the CONTAINER is running (Check the STATUS column. You will find that it says it is Up!)

We can attach to a running Container via the docker attach command. Let us attach to it:

**\$ docker attach ab0e37214358**

This will get you back to the Prompt i.e. you are now inside the busybox container. Type exit to exit the container and then try the docker ps command. There will be no running containers.

**Note: If you want to stop a running container, you can give the docker stop <ContainerId> command. Try it.**

**Tip: You need not always give the full CONTAINER\_ID value. Type a few letters from the start of the CONTAINER\_ID and it should work. Neat, isn't it ?**

## Part 3 – Images and Containers

In this section, we shall get acquainted with various other Docker commands and parameters that will give you a good grasp of how to deal with Images, running containers, container networking to some extent and more.

To learn that, it is best to dive deep into running one of the popular servers of all time , the Apache Web Server and working with its Dockerized Image from the Docker Hub.

### Running Apache Web Server

Now, let's try another experiment to help you understand how you will eventually envision your applications running inside Docker.

The title of this section indicates that we want to run the Apache Web Server. To do that, you should now start thinking that there must be a Docker image that someone must have created for Apache Web Server.

So, the first step is to do the following:

**\$ docker search httpd**

This gives the following output:

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker search httpd
```

NAME	DESCRIPTION	STARS	OFFICIAL
httpd	The Apache HTTP Server Project	1383	[OK]
hypriot/rpi-busybox-httpd	Raspberry Pi compatible Docker Image with ...	39	
centos/httpd		15	
armhf/httpd	The Apache HTTP Server Project	8	
centos/httpd-24-centos7	Platform for running Apache httpd 2.4 or b...	6	
macadmins/netboot-httpd	use in combination with bruienne/bsdpy	4	
lolhens/httpd	Apache httpd 2 Server	2	
salim1983hoop/httpd24	Dockerfile running apache config	2	
fboaventura/dckr-httpd	Small footprint http server to use with ot...	1	

We will go with the OFFICIAL image for httpd. So let's learn a new command i.e. docker pull

This command only pulls (downloads) the Docker image to your local repository and does not run it.

Give the following command at the boot2Docker prompt:

**\$ docker pull httpd**

Be Patient. This will download the httpd image. If you run a docker images command, you should see http listed in it.

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
httpd	latest	7239615c0645	46 hours ago	177MB
busybox	latest	6ad733544a63	5 weeks ago	1.13MB

The [official documentation](#) for the httpd Docker image is something that you should look at. In fact make it a habit to go through the documentation page for each of the images present since it will give you clear instructions on how to run it and any other configuration details.

To make the default Apache Web Server run in the container, all we need to do is the following (Note that we are using the -d parameter i.e. running the Container in Detached mode). We have also given a name to our container via the —name parameter.

**\$ docker run -d --name MyWebServer httpd  
54c39730ab784a998c3bb4522cf6421a4a0c7d46db54afa75202e76741032216**

This will launch the default Apache HTTP Server based on the httpd image and since we have launched the container in detached mode, we got back a Container ID. Your ID need to be similar to the one that I got “54c39730ab784a998c3bb4522cf6421a4a0c7d46db54afa75202e76741032216”

Now, let us check if the Container is running. To do that we will use our familiar docker ps command as shown below:

**\$ docker ps**

```
Romin-Iranis-MacBook-Pro:~ romin-irani$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAM
54c39730ab78	httpd	"httpd-foreground"	13 seconds ago	Up 13 seconds	80/tcp	Myw

Note that it is in running status. Also notice that the NAMES column now has the name that we specified for our Container i.e. MyWebServer.

## Detour: Stop the Container

If you wish to stop the Container at any point in time, you can now use the handy name that you provided instead of the CONTAINER\_ID value in the docker stop command.

Try it now:

1. To stop the Container, give the following command:

```
$ docker stop MyWebServer
```

2. Try the :

```
$ docker ps command.
```

3. I also suggest to remove the Container via the following command:

```
docker rm MyWebServer
```

Great! Now, let us start it back again.

```
$ docker run -d --name MyWebServer httpd  
<SOME_LARGE_CONTAINER_ID>
```

## What is that PORTS column when you do a docker ps?

You will notice that there is an entry in the PORTS column. The value for that is “tcp:80”. This means that port 80 is being exposed by the Container. This looks reasonable since Apache Web Server default port for HTTP is 80. If you are adventurous enough, you can take a look at the Dockerfile (don’t worry about what a Dockerfile is) for this project is [here](#).

Go right to the end and you will find 2 entries in the file, which I am reproducing here:

```
EXPOSE 80  
CMD httpd-foreground
```

You can see that port 80 is exposed. All looks good for the moment.

## Accessing our web site

Now, the next check you would do is use a utility like curl or even the browser on your local machine to check out the site.

But what would the IP address be?

If you are running the Docker for Mac or Docker for Windows, it should be the localhost. If you are running in a VirtualBox environment, find that out.

With that information in hand, let us launch the local browser and open the localhost URL.

Oops! No page appears. We were expecting at least the Apache Home page to appear over here but that does not seem to be the case.

This is because the default port that Apache Web Server runs on is port 80 but that port has not been exposed on the host side and as a result of that the web site is not accessible.

## Show me the Proof?

Well I mentioned to you that port 80 is exposed by the Container but not by the Host. In other words, port 80 is the private port exposed by the Container but there does not seem to be a port mapped for the public facing host because of which we are seeing the problem.

It seems there is a docker command for that too. And its intuitively called port.

Try out the following command :

```
$ docker port MyWebServer
$
```

And the output is as expected i.e. there is no mapping done for the public facing port of the Host. Luckily, help is on the way !

## Random Port Mapping

First up, stop and remove the Container so that we can use the same Container Name i.e. MyWebServer.

```
$ docker stop MyWebServer
MyWebServer
```

```
$ docker rm MyWebServer
MyWebServer
```

Now, let us start the httpd Container with an extra parameter i.e. -P. What this parameter does is that it “Publish all exposed ports to random ports”. So in our case, the port 80 should get mapped to a random port, which is going to be the public port.

Execute the following command:

```
docker@boot2docker:~$ docker run -d --name MyWebServer -P httpd
60debd0d57bf292b0c3f006e4e52360feaa575e45ae3caea97637bb26b490b10
```

Next, let us use the port command again to see what has happened:

```
$ docker port MyWebServer
```

80/tcp -> 0.0.0.0:32769

We can see that port 80 has got mapped to port 32769. So if we access our web site at <http://<HostIP>/<HostPort>>

it should work now.

So on my machine it is <http://192.168.59.103:32769> and it works!



## Specific Port Mapping

So what if we wanted to map it to a port number other than 32769. You can do that via the `-p` (note the lowercase) parameter.

This parameter format is as follows:

`-p HostPort:ContainerPort`  
For e.g. `-p 80:80` or `-p 8080:80`

The first parameter is the Host port and we are now making that 80. The second port is what Apache httpd exposes i.e. 80.

Let us check out everything again:

```
$ docker stop MyWebServer  
MyWebServer
```

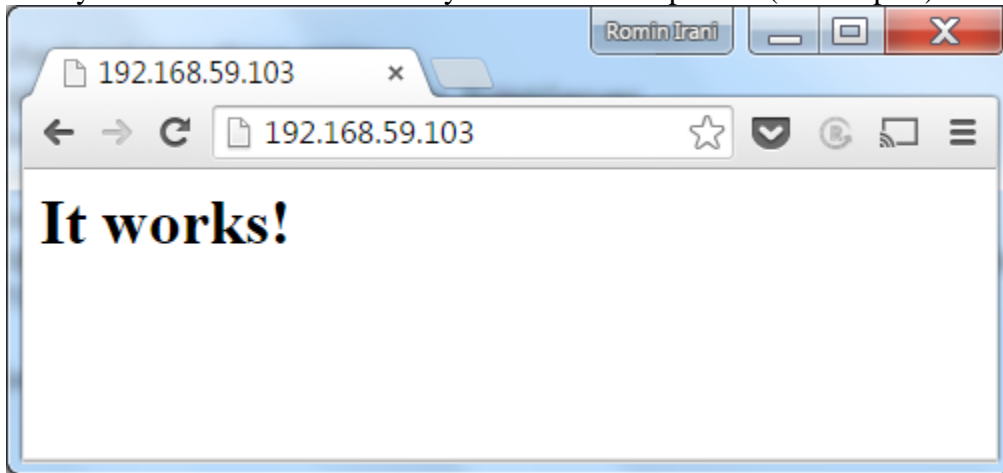
```
$ docker rm MyWebServer  
MyWebServer
```

```
$ docker run -d --name MyWebServer -p 80:80 httpd  
02ce77550940cb1f37361b74af8913e46d6a507d06c2579b8a8b49e389b1e75f
```

Now, let us give the port command again:

```
$ docker port MyWebServer
80/tcp -> 0.0.0.0:80
```

Now you should be able to access your web site via port 80 (default port):



This brings us to an end of this section. You will still have a lot of questions in terms of Apache Web Server. Some of those would include where should you put your Web site files (HTML, CSS, etc) instead of this default one.

ss

Those concerns are valid but the point in this session was to get you comfortable with various Docker commands, managing Containers, their ports and so on. When we come to the section for writing Docker files, things will get much clearer.

## Part 4 – Docker Hub

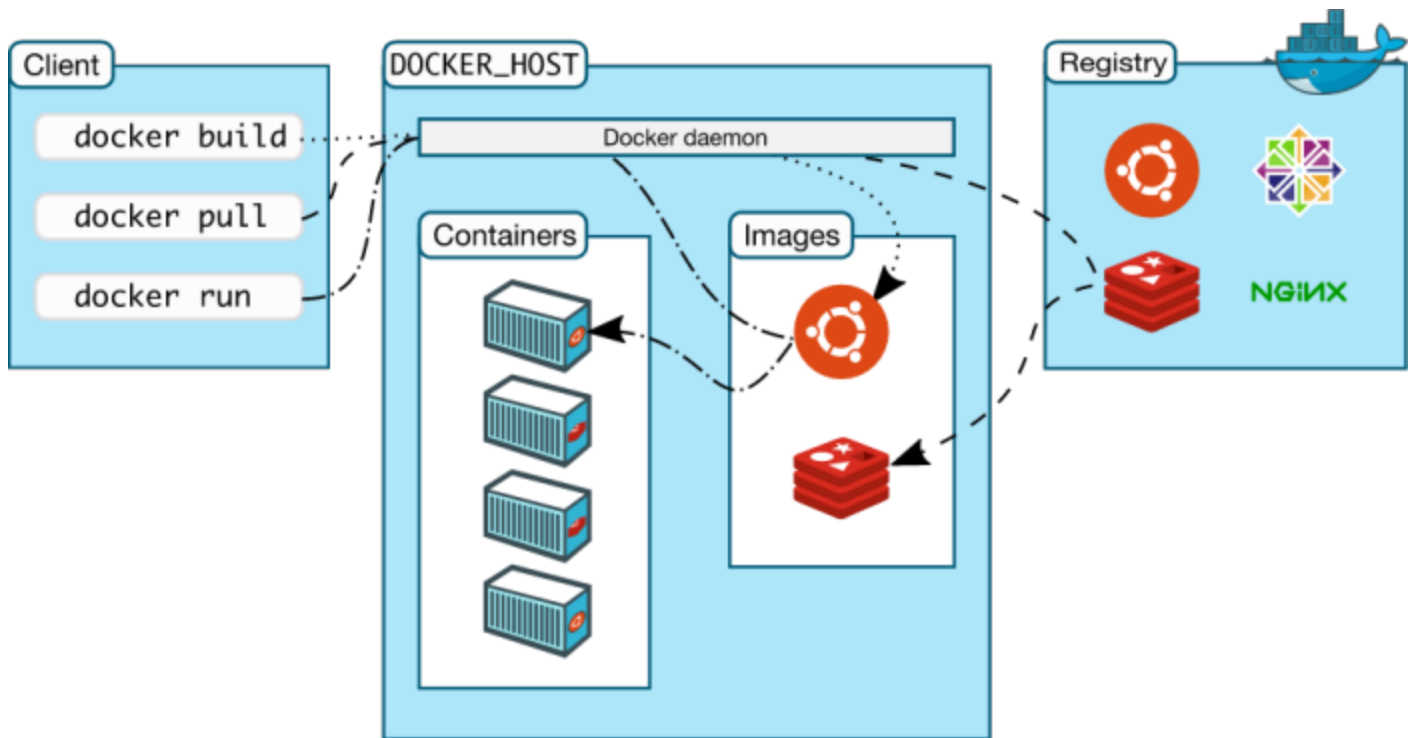
In this session, we shall look at working with the [Docker registry](#). Some of this has already been covered, but it is nice to go through these commands again plus this session has a little bit more on the Docker Registry (Hub).

### Recap

Recollect that the Docker toolset consists of:

1. Docker Daemon
2. Docker Client
3. Docker Hub

The Docker Architecture diagram is reproduced below again:



Let us focus on the Registry part in the diagram. The Registry is also called the Hub. So for the purpose of this document, the 2 terms are interchangeable.

Docker hosts public repositories called the Docker Registry (Hub) where you can find a list of public Docker images for your use. This is handy, since a lot of developers have worked hard to get the images ready for us and all we need to do is pull those images and start launching containers based on them.

The implications of this for the developer is huge. It cuts down your time drastically in downloading/setting up applications that you want to have running locally as fast as possible. For e.g. consider that you have to setup MySQL. One option available to you is to go down the traditional path and download the installer binary from the official MySQL site. Then follow the instructions to set things up and more. You know that this process is time consuming and error prone.

By now, you would have realised that a standard format for containers makes things not just easy and quick with a toolchain like Docker but you are also sure that it's going to run not just on your machine but if you need to give that image to someone else, you are confident of it running well there too.

So to summarize, do the following steps:

1. Visit the Docker Hub : <https://hub.docker.com>
2. Do sign up for the hub, this can help you push your own Docker images into the Hub too.
3. Check out the above Hub site. See the huge list of Docker images available for almost any software that you have used so far.

## Searching for Docker Images

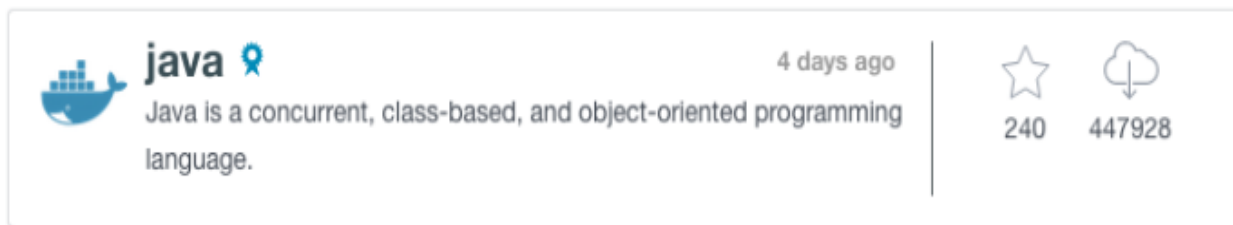


Assuming you are logged in with your Account (Search actually works without an account too!)—try searching for any images via the Web Search box as shown below:



For e.g. type in Java, press <return> and see the list of repositories that are returned.

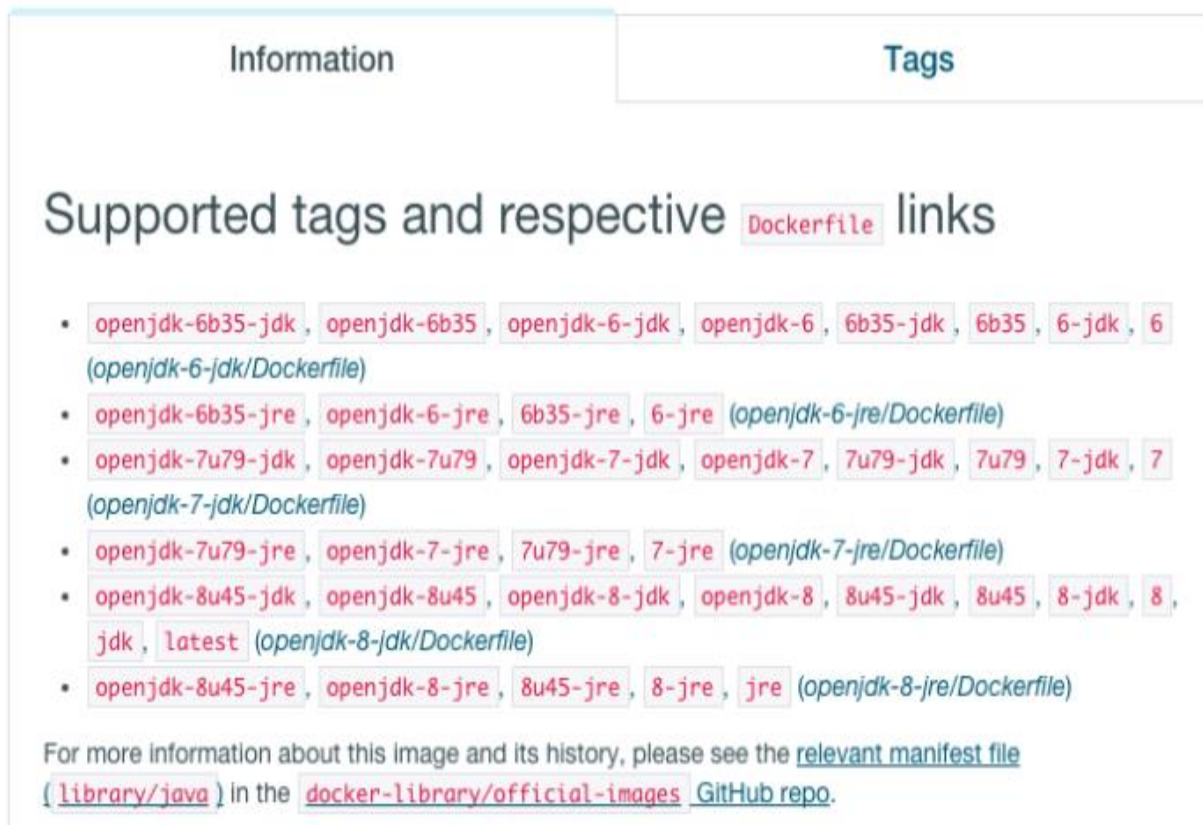
The screenshot below is the first repository that was listed:



Notice the stars and the downloads. It gives you a good idea of its popularity.

Click on that repository and you will be lead to a page that will show various images (by their tags) and instructions for running a container based on that image and so on.

Shown below is a screenshot of the images and their tags:



Try It Out : In the image above, search for the latest tag. It will be there. Click that and it will lead you to a Dockerfile (more on that in another hands-on). Even if you do not understand all aspects of the Dockerfile, do take a look slowly and you will begin to see a pattern image. Hint : OS -> Software -> Any other dependencies , etc are slowly built one by one into your image.

### **Pulling an Image**

Now that you are familiar with IMAGES and their TAGS, it should be clear to you that those are the only two pieces of information that you need to pull down an image locally to your step.

You can do that via the docker pull command as shown below:

```
$ docker pull java
```

or

```
$ docker pull java:latest
```

or

```
$ docker pull java:8
```

and so on.

### **Viewing a list of images**

You can check out the list of images that you have locally via the docker images command:

```
$ docker images
```

Understand the output. It will have REPOSITORY and the TAG in the output result.

### **Launching a Container**

Now that you have a local image, you can launch a container via :

```
$ docker run <imagename>:<tag>
```

### **Searching for Images**

Earlier we saw how we could search for Docker images via the Docker Hub Web interface. We can do that via the docker command line too.

```
$ docker search <TERM>
```

For e.g.

```
$ docker search httpd
```

```
$ docker search java
```

```
$ docker search mysql
```

## Part 5 – Building Your Own Docker Images

In this section, we are going to take our first steps to building our own image. We are going to keep it simple at first by adding our software on top of images that are already present. Later on in another chapter, we will look at writing our own files.

Once we have created our image, we will also push this image to the Docker Hub. Keep your Docker Hub username handy. **In case you have not yet registered for the Docker Hub, I suggest that you do so now at the following [link](#).**

## What happened to my data?

It is important that we first understand the key difference between containers and images. To summarize it, it is important to remember that “Images are Immutable and Containers are Ephemeral”. Let us see that in action now.

Let us begin with starting **boot2docker** utility and then working with the base **ubuntu** image that we have already downloaded earlier. First make sure that you do have ubuntu latest image to get going here.

Fire up the following command:

```
$ docker images
```

This should give you a list of Docker images that you have and that **ubuntu:latest** should be present on it. If not, I suggest that you do a

```
$ docker pull ubuntu:latest
```

Let us launch a container from the **ubuntu:latest** image by giving the following command:

```
$ docker run -it --name mycontainer1 --rm ubuntu:latest
```

This should lead you to a prompt. For e.g. on my machine, I have the following prompt:

```
root@ea503e60bae3:/#
```

Now, let's install the popular Git software on this container instance by running the following commands:

```
sudo apt-get update
```

This should output a stream of messages. Let the process continue its work till you see a message as given below:

```
Reading package lists... Done
```

You will be back at the prompt. Now, let us install Git via the command given below:

```
sudo apt-get install git
```

It will prompt you with a message:

```
Do you want to continue? [Y/n]
```

Please go ahead with a **Y**.

It will take a few seconds to download the Git software and install it. You should finally be at the prompt once everything is done.

To verify that Git is installed, simply type git at the prompt as shown below:

```
$ git --version
```

This should print out the Git version at the console as shown below:

```
git version 1.9.1
```

Now, let us exit the container by typing `exit`. Keep in mind that we had provided the `--rm` flag while starting the container, which means that the container is removed on termination.

Now, let us launch another instance of the same ubuntu container as shown below.

```
$ docker run -it --name mycontainer1 --rm ubuntu:latest
```

Type **git** at the prompt. It displays the message that git is not found:

```
root@42f5b5340f27:/# git  
bash: git: command not found  
root@42f5b5340f27:/#
```

What happened? Didn't we just install Git on ubuntu and expecting it to be present. The point is that each Container instance launched this way is independent and is depending on the master image i.e. `ubuntu:latest`, which does not have Git installed.

## Committing your Image

To ensure that next time we have a version of ubuntu with Git installed, we need to commit our image. What this means is that we started with a container based on the ubuntu image. Then we added some software to it i.e. git. This means that we modified the state of the container. Hence we need to save that state of the container as an image, so that can relaunch additional new containers from that image. This way they will have the git software installed too.

Let us exit from the container in the previous section and following these steps:

- Launch the container based on ubuntu, this time without the `--rm` flag

```
$ docker run -it --name mycontainer1 ubuntu:latest
```

- Update the OS and install Git via the commands given below:

```
$ sudo apt-get update  
$ sudo apt-get install git
```

- Verify that Git is installed via the command given below:

```
$ git --version
```

- Exit the container by typing **exit**.
- Run the `docker ps -all` command to see the `mycontainer1` that we launched:

**\$ docker ps -all**

- This should show you the **mycontainer1** that we had launched. It is currently in exited state.
- Commit the container image via the `docker commit` command as shown below:

`docker commit [ContainerID] [Repository[:Tag]]`

- In our case, we use the `mycontainer1` as the containerid since we have given it a name. If you had not started that container with a name, you could have used the Container Id that is visible in the **docker ps -all** command.
- The Repository name is important. Eventually (and which is what we will do) we will want to push this to the Docker Hub. So the format of the Repository name that is recommended is the following:

**<dockerhubusername>/<repositoryname>**

- In our case, the repository-name can be something like `ubuntu-git` and you will need to substitute your `<dockerhubusername>` tag above with your Docker Hub user name.
- If you do not give the tag, it will be marked as **'latest'**, which is fine for us. For e.g. my Docker Hub user name is `rominirani` and hence I will commit it in the following manner:

**\$ docker commit mycontainer1 rominirani/ubuntu-git**

- This will give you back the image ID.
- Check the list of docker images

**\$ docker images**

- You should see at the top of the list an image that is like the following entry that I have:

**\$ docker images**

```
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
rominirani/ubuntu-git latest 23cf273faded About a minute ago 247.1 MB
```

This shows that our Repository **rominirani/ubuntu-git** has got created.

# Launching a Container from your Image

We can now launch a container from our newly created Docker image i.e. **yourusername/ubuntu-git** via the docker run command.

```
$ docker run -it --name c1 <yourusername>/ubuntu-git
```

The above command will launch the container based on our image. It will then take you to the prompt, where you can verify that it comes with Git by giving the git—version command.

## Pushing the Image to the Docker Hub

To push the image to the Docker Hub via the following steps:

1. Get an account at [Docker Hub](#)
2. From **boot2docker** prompt, execute the command **docker login** and follow the instructions for your username and password.
3. On successful login, you should get a **Login Succeeded** message.
4. To a docker push as shown below:

```
$ docker push <yourusername>/ubuntu-git
```

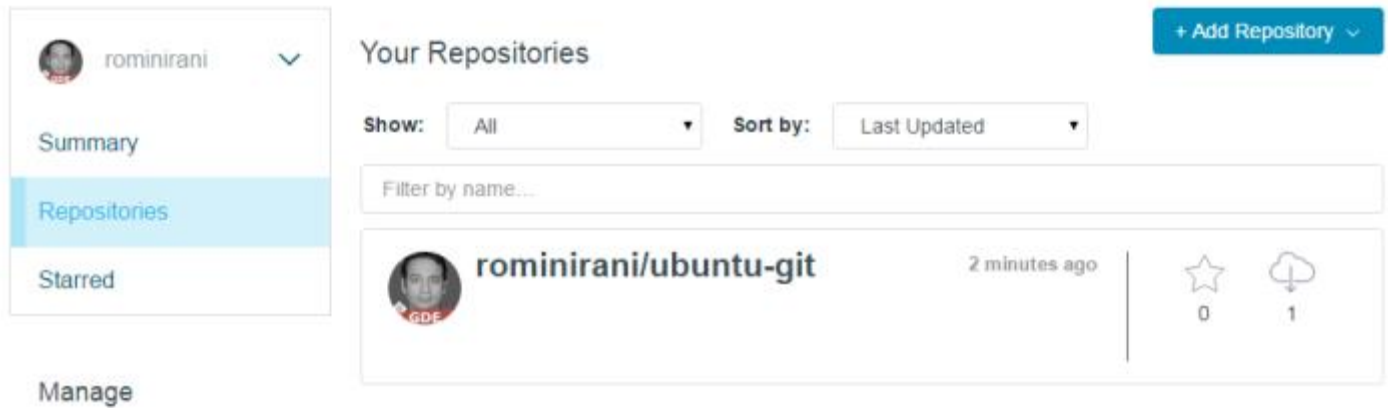
This will take a while to upload the details. Note that Docker is smart in the sense that it already will detect if the base image i.e. ubuntu already exist and hence it will cleverly chose only those layers that need to be uploaded. A sample run of the process is shown below:

```
$ docker push rominirani/ubuntu-git
```

```
The push refers to a repository [rominirani/ubuntu-git] (len: 1)
0644edf7f9c6: Image already exists
23cf273fafed: Pushing 5.505 MB/37.93 MB
23cf273fafed: Image successfully pushed
d0955f21bf24: Image successfully pushed
9fec74352904: Image successfully pushed
a1a958a24818: Image successfully pushed
f3c84ac3a053: Image successfully pushed
511136ea3c5a: Image successfully pushed
Digest: sha256:397eac492cd67b1e4b1371fc74a676c58d059756fe20022f6346d6f99b908a1b
```

```
docker@boot2docker:~$
```

Check for the repository in Docker Hub (web):



You can also search for the repository via the docker search command. Try the following command:

```
$ docker search ubuntu-git
```

You should be able to locate your repository in the search results.

**Exercise:** Try out some other repositories, put your files on top of it and then commit your own image.



## Part 6 - Docker Private Registry

In this part we shall take a look at how you can host a local Docker registry. In an earlier part, we had looked at the Docker Hub, which is a public registry that is hosted by Docker. While the Docker Hub plays an important role in giving public visibility to your Docker images and for you to utilize quality Docker images put up by others, there is a clear need to setup your own private registry too for your team/organization.

### Pull down the Registry Image

You might have guessed by now that the registry must be available as a Docker image from the Docker Hub and it should be as simple as pulling the image down and running that. You are correct!

A [Docker Hub search](#) on the keyword 'registry' brings up the following image as the top result:



Assuming that you have **boot2docker** running, execute the standard command to pull down the registry image.

```
$ docker pull registry
```

This will pull down the '**latest**' registry image and once it is pulled successfully, you should be able to see that in via the docker images command.

### Run the local Registry

The registry Docker image is configured to start on port 5000 in the container, so we will expose the host port also as 5000.

You can launch the registry via the following command:

```
$ docker run -d -p 5000:5000 --name localregistry registry
```

On successful launch, it will print the Container ID on the console.

To recap, we are starting a container named '**localregistry**' based on the '**registry**' image. The container is started in detached mode and the host:container port mapping has been done for both the port numbers to be 5000.

Check if the our container named '**localregistry**' has started via the **docker ps** command as shown below:

```
$ docker ps
```

The STATUS column should indicate that it is up.

## Pull down a few images and push to local Registry

Now, let us pull down a few images first and then push them into the local Registry. Let us do that in 2 steps:

### Step 1: Pull down busybox and alpine Linux Images

Execute the pull commands for the following two images as shown below:

```
$ docker pull busybox  
$ docker pull alpine
```

Once the images have been pulled down, verify that they are present in your Images list via the docker images command.

#### Quick Exercise:

If at this point, we try to pull the alpine image from our local registry, it should ideally respond with the image not found, shouldn't it ? Yes, it will. Remember that we have already started the registry container on port 5000. The result of this operation is shown below:

```
$ docker pull localhost:5000/alpine  
Using default tag: latest  
Error response from daemon: manifest for localhost:5000/alpine:latest not found
```

Notice that the format for specifying the Image in a specific registry is as:

[REGISTRY\_HOSTNAME:REGISTRY\_PORT]/IMAGENAME

For public Docker Hub, we were not specifying the **[REGISTRY\_HOSTNAME:REGISTRY\_PORT]** option. But for our local registry we need to specify that so that the docker client will look there.

## Step 2 : Push busybox and alpine Linux Images into the local Registry

Now, we will push the two images that we downloaded (**busybox** and **alpine**) into the local registry. Remember that these are two images that you downloaded directly from the Docker Hub. We have not modified it in any way. But you could create your own modified image (as we saw earlier in this series) and and push that into the local Registry too.

The step to push your image into the local Registry is done as follows:

- The first step is to take your image or container. Let us work with the alpine image that we have pulled earlier. The fully qualified name for this image is **alpine:latest** if you do a `docker images` command. Execute the following command to tag the **alpine:latest** image with the tag of the local registry to which we are going to push it.

```
$ docker tag alpine:latest localhost:5000/alpine:latest
```

If you now run a `docker images` command, you will see both the **alpine** and the **localhost:5000/alpine** images listed.

The next step is to push this tagged image or container into the local registry.

This is done via the standard `docker push` command that you saw earlier. All we have to do is use the new tagged **localhost:5000/alpine** image. The command is given below:

```
$ docker push localhost:5000/alpine:latest
```

The push refers to a repository [localhost:5000/alpine]  
**04a094fe844e: Pushed**

**latest: digest: sha256:5cb04fce748f576d7b72a37850641de8bd725365519673c643ef2d14819b42c6**  
**size: 528**

## Further Reading

In this part, we looked at running a private registry. This was a registry that did not enforce any authentication on the part of the client. While this is useful from an understanding point of view, keep in mind that you should still follow the best practices for running a secure registry. It is not just about running a secure registry but also about making decisions around storage options for your private registry repositories. Towards this, I suggest to read up on this [article](#).

## Part 7 - Data Volumes

In this part, we shall take a look at Docker Volumes. By Docker Volumes, we are essentially going to look at how to manage data within your Docker containers.

If you have been following the series so far, we have launched Containers from various images and even created a basic image for our own use. To reiterate the point, Containers are Ephemeral and once a container is removed, it is gone. What about scenarios where you want the applications running inside the container to write to some files/data and then ensure that the data is still present. For e.g. let's say that you are running an application that is generating data and it creates files or writes to a database and so on. Now, even if the container is removed and in the future you launch another container, you would like that data to still be there.

In other words, the fundamental thing that we are trying to get over here is to separate out the container lifecycle from the data. Ideally we want to keep these separate so that the data generated is not destroyed or tied to the container lifecycle and can thus be reused. This is done via Volumes, which we shall see via several examples.

As per the official documentation, there are 2 ways in which you can manage data in Docker:

- Data volumes
- Data volume containers

Let us work through examples for both the above cases.

### Few points to keep in mind about Data Volumes

- A data volume is a specially designed directory in the container.

- It is initialized when the container is created. By default, it is not deleted when the container is stopped. It is not even garbage collected when there is no container referencing the volume.
- The data volumes are independently updated. Data volumes can be shared across containers too. They could be mounted in read-only mode too.

## Mounting a Data volume

Let us begin first with the most basic operation i.e. mounting a data volume in one of our containers. We will be working with the **busybox** image to keep things simple.

We are going to use the `-v [/VolumeName]` as an option to mount a volume for our container. Let us launch a container as given below:

```
$ docker run -it -v /data --name container1 busybox
```

This will launch a container (named `container1`) in interactive mode and you will be at the prompt in the container.

Give the `ls` command as shown below:

```
$ docker run -it -v /data --name container1 busybox
/ # ls
bin  data dev  etc  home proc root sys  tmp  usr  var
/ #
```

Notice that a volume named **data** is visible now.

Let us do a `cd` inside the data volume and create a file named `file1.txt` as shown below:

```
/ # cd data
/data # touch file1.txt
/data # ls
file1.txt
/data #
```

So what we have done so far is to mount a volume `/data` in the container. We navigated to that directory (`/data`) and then created a file in it.

Now, let us exit the container by typing `exit` and going back to the terminal.

```
/data # exit
$
```

Now, if we do a `docker ps -a`, we should see our container (container1) currently in the exited state as shown below:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
22ab6644ea6b	busybox	"/bin/sh"	3 minutes ago	Exited (0) 23 sec ago		container1

Now, let us inspect the container and see what Docker did when we started this container.

Give the following command:

```
$ docker inspect container1
```

This will give out a JSON output and you should look for Mounts attribute in the output. A sample output from my machine is shown below:

```
"Mounts": [
{
  "Type": "volume",
  "Name": "568258a7940182c5ac52f0637c60c1d1f81e9ec77f3e4694647b4879c2ff003c",
  "Source": "/var/lib/docker/volumes/568258a7940182c5ac52f0637c60c1d1f81e9ec77f3e4694647b4879c2ff003c/_data",
  "Destination": "/data",
  "Driver": "local",
  "Mode": "",
  "RW": true,
  "Propagation": ""
}],
```

This tells you that when you mounted a volume (**/data**), it has created a folder **/var/lib/....** for you, which is where it puts all the files, etc that you would have created in that volume. Note that we had created a **file1.txt** over there (we will come to that in a while).

Also notice that the **RW** mode is set to true i.e. Read and Write.

Now that the container is stopped i.e. exited, let us restart the container (**container1**) and see if our volume is still available and that **file1.txt** exists.

```
$ docker restart container1
container1
```

```
$ docker attach container1
```

```
/ # ls
```

```
bin data dev etc home proc root sys tmp usr var
```

```
/ # cd data
```

```
/data # ls
```

```
file1.txt
```

```
/data #
```

The sequence of steps above are as follows:

1. We restarted the container (**container1**)
2. We attached to the running container (**container1**)
3. We did a **ls** and found that our volume **/data** is still there.
4. We did a **cd** into the **/data** volume and did a **ls** there. And our file is still present.

Hopefully this explains to you how you can persist your data in volumes.

Now, let's do an interesting thing. Exit the container and remove the container.

```
/data # exit
```

```
$ docker rm container1
```

```
container1
```

```
$ docker volume ls
```

DRIVER	VOLUME NAME
local	568258a7940182c5ac52f0637c60c1d1f81e9ec77f3e4694647b4879c2ff003c

This shows to you that though you have removed the **container1**, the data volume is still present on the host. This is a dangling or ghost volume and could remain there on your machine consuming space. Do remember to clean up if you want. Alternatively, there is also a **-v** option while removing the container as shown in the help:

```
$ docker rm --help
```

```
Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

```
Remove one or more containers
```

```
-f, --force=false Force the removal of a running container (uses SIGKILL)
```

```
--help=false Print usage
```

```
-l, --link=false Remove the specified link
```

```
-v, --volumes=false Remove the volumes associated with the container
```

This will always remove your volumes when the container is also removed.

Exercise: What happens if you launch another container with the same /data volume. Is the file still there or does each container get its own file system? Try it out.

## Mounting a Host Directory as a Data volume

Now that we have seen how to mount a volume in the container, the next step is to look at the same process of mounting a volume but this time we will mount an existing host folder in the Docker container. This is an interesting concept and is very useful if you are looking to do some development where you regularly modify a file in a folder outside and expect the container to take note or even sharing the volumes across different containers.

In our example, we are using Docker for Mac. Click on that and then go to preferences and then File Sharing. You will see a screenshot similar to this:





To mount a host volume while launching a Docker container, we have to use the following format for volume **-v** :

**-v HostFolder:ContainerVolumeName**

So, let us start a busybox container as shown below:

```
$ docker run -it --name container1 -v /Users:/datavol busybox  
/#
```

What we have done here is that we have mapped the host folder /Users to a volume /datavol that will be mounted inside our container (container1).

Now, if we do a **ls** , we can see that the /datavol has been mounted. Do a cd into that folder and a ls, and you should be able to see the folder contents of C:\Users on your machine.

```
/ # cd datavol  
/datavol # ls  
Shared          romin-irani  
/datavol #
```

Hope this makes it clear on how to use host folders.

Exercise:

1. Try adding a file directly from your laptop/machine in **/Users/<username>** folder and then go back to your running container and do a ls there. You should be able to see that new file there.
2. From the container shell, go to the **/datavol** folder and then add a file there. Then go back to your machine/laptop and do a ls. You should see the new files there.

**Additional Note:** Optionally if you require, you can mount a single host file too as a data volume.

Start thinking now of how you would use host folders that have been mounted as data volumes. Assume you are doing development and have the Apache Web Server or any other Web Server running in the container. You could have started the container and mounted a host director that the Web Server can use. Now on your host machine, you

could make changes using your tools and those would then get reflected directly into your Docker container.

## Data volume containers

We now come to the next part i.e. creating a Data volume container. This is very useful if you want to share data between containers or you want to use the data from non-persistent containers. The process is really two step:

1. You first create a Data volume container
2. Create another container and mount the volume from the container created in Step 1.

Let us see that in action:

We will first create a container (**container1**) and mount a volume inside of that:

```
$ docker run -it -v /data --name container1 busybox
```

Now, let us go into the /data volume and create two dummy files in it as shown below:

```
/ # cd data
/data # ls
/data # touch file1.txt
/data # touch file2.txt
/data #
```

Now launch another terminal.

Now, if we do a **docker ps**, we should see our running container:

```
$ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
006f7ba16783	busybox	"/bin/sh"	27 seconds ago	Up	26 seconds

Now, if we execute a command on the running **container1** i.e. see the contents of our **/data** volume, you can see that the two files are present.

```
$ docker exec container1 ls /data
file1.txt
file2.txt
```

Great ! Now, let us launch another container (**container2**) but it will mount the data volume from container1 as given below:

```
$ docker run -it --volumes-from container1 --name container2 busybox
```

Notice above that we have launched it in interactive mode and have used a new parameters—**volumes-from** <containername> that we have specified. This tells **container2** to mount the volumes that **container1** mounted.

Now, if we do a **ls**, we can see that the data folder is present and if we do a **ls** inside of that, we can see our two files: **file1.txt** and **file2.txt**

```
/ # ls
bin dev home lib64 media opt root sbin tmp var
data etc lib linuxrc mnt proc run sys usr
/ # cd data
/data # ls
file1.txt file2.txt
/data #
```

You can launch multiple containers too , all using the same data volume from **container1**. For e.g.

```
$ docker run -it --volumes-from container1 --name container3 busybox
$ docker run -it --volumes-from container1 --name container4 busybox
```

## Additional Reading

Do check out the [official documentation on data volumes](#). It goes through similar examples and contains an important section on how you can backup, restore and migrate data volumes.

# Docker Tutorial Series: Part 8: Linking Containers

In this part, we shall take a look at how to link Docker Containers. By linking containers, you provide a secure channel via which Docker containers can communicate to each other.

Think of a sample web application. You might have a Web Server and a Database Server. When we talk about linking Docker Containers, what we are talking about here is the following:

1. We can launch one Docker container that will be running the Database Server.
2. We will launch the second Docker container (Web Server) with a link flag to the container launched in Step 1. This way, it will be able to talk to the Database Server via the link name.

This is a generic and portable way of linking the containers together rather than via the networking port that we saw earlier in the series. Keep in mind that this chapter covers Linking Containers via the—link flag. A new tool [Docker Compose](#) is the recommended way moving forward but for this tutorial, I will cover the—link flag only and leave it to the reader to look at Docker Compose.

Let us begin first by launching the popular NoSQL Data Structure Server Redis. Like other software, Redis too has its official Docker image available in the Docker Hub.

First, let us pull down the Redis image via the following command:

```
$ docker pull redis
```

Next, let us launch a Redis container (named **redis1**) in detached mode as follows:

```
$ docker run -d --name redis1 redis  
37f174130f758083d243541e8adab7e2d8be2012e555cbdbd5fc67ca9d26526d
```

We can check that **redis1** container has started via the following command:

**\$ docker ps**

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
37f174130f75	redis	"/entrypoint.sh redi	30 seconds ago	Up	29 seconds	6379/tcp redis1

Notice that it has started on port 6379.

Now, let us run a another container, a **busybox** container as shown below:

**\$ docker run -it --link redis1:redis --name redisclient1 busybox**

Notice the—**link** flag. The value provided to the—**link** flag is **sourcecontainername:containeraliasname**. We have chosen the value **redis1** in the **sourcecontainername** since that was the name that was given to our first container that we launched earlier. The **containeraliasname** has been selected as **redis** and it could be any name of your choice.

The above launch of container (**redisclient1**) will lead you to the shell prompt.

Now, what has this launch done for you. Let us observe first what entry has got added in the **/etc/hosts** file of the **redisclient1** container:

```
/ # cat /etc/hosts
172.17.0.23 26c37c8982e9
127.0.0.1 localhost
::1 localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.17.0.21 redis 37f174130f75 redis1
/ #
```

Notice an entry at the end, where the container **redis1** has got associated with the **redis** name.

Now, if you do a ping by the host name i.e. alias name (**redis**)—it will work:

```
/ # ping redis
PING redis (172.17.0.21): 56 data bytes
64 bytes from 172.17.0.21: seq=0 ttl=64 time=0.218 ms
64 bytes from 172.17.0.21: seq=1 ttl=64 time=0.135 ms
64 bytes from 172.17.0.21: seq=2 ttl=64 time=0.140 ms
64 bytes from 172.17.0.21: seq=3 ttl=64 time=0.080 ms
```

If you print out the environment variables you will see the following (I have removed the other environment variables):

```
/ # set
```

```
....  
REDIS_ENV_REDIS_DOWNLOAD_SHA1='0e2d7707327986ae652df717059354b358b83358'  
REDIS_ENV_REDIS_DOWNLOAD_URL='http://download.redis.io/releases/redis-3.0.3.tar.gz'  
REDIS_ENV_REDIS_VERSION='3.0.3'  
REDIS_NAME='/redisclient1/redis'  
REDIS_PORT='tcp://172.17.0.21:6379'  
REDIS_PORT_6379_TCP='tcp://172.17.0.21:6379'  
REDIS_PORT_6379_TCP_ADDR='172.17.0.21'  
REDIS_PORT_6379_TCP_PORT='6379'  
REDIS_PORT_6379_TCP_PROTO='tcp'  
....
```

You can see that various environment variables were auto-created for you to help reach out to the **redis1** server from the **redisclient1**.

Sounds good? Now, let us get a bit more adventurous.

Exit the **redis1** container and come back to the **terminal**.

Let us launch a container based on the **redis** image but this time instead of the default command that will launch the redis server, we will simply go into the shell so that all the redis client tools are ready for us. Note that the **redis1** server (container) that we launched is still running.

```
$ docker run -it --link redis1:redis --name client1 redis sh
```

This will lead you to the prompt. Do a ping of redis i.e. our alias name and it should work fine.

```
# ping redis
```

```
PING redis (172.17.0.21): 48 data bytes  
56 bytes from 172.17.0.21: icmp_seq=0 ttl=64 time=0.269 ms  
56 bytes from 172.17.0.21: icmp_seq=1 ttl=64 time=0.248 ms  
56 bytes from 172.17.0.21: icmp_seq=2 ttl=64 time=0.205 ms
```

Next thing is to launch the redis client (**redis-cli**) and connect to our redis server (running in another container and to which we have linked) as given below:

```
# redis-cli -h redis
redis:6379>
```

You can see that we have been able to successfully connect to the redis server via the alias name that we specified in the—**link** flag while launching the container. Ofcourse if we were running the Redis server on another port (other than the standard 6379) we could have provided the **-p** parameter to the **redis-cli** command and used the value of the environment variable over here (**REDIS\_PORT\_6379\_TCP\_PORT**). Hope you are getting the magic!

Now, let us execute some standard Redis commands:

```
redis:6379> PING
PONG
redis:6379> set myvar DOCKER
OK
redis:6379> get myvar
"DOCKER"
redis:6379>
```

Everything looks fine. Let us exit this container and launch another client (**client2**) that wants to connect to the same Redis Server that is still running in the first container that we launched and to which we added a string key / value pair.

```
$ docker run -it --link redis1:redis --name client2 redis sh
```

Once again, we execute a few commands to validate things:

```
# redis-cli -h redis
redis:6379> get myvar
"DOCKER"
redis:6379>
```

You are all set now to link containers together.

## Additional Reading

Check out [Docker Compose](#), which provides a mechanism to do the linking but by specifying the containers and their links in a single file. Then all one needs to do is use the docker-compose tool to run this file and it will figure out the relationships (which container needs to launch first, etc) and launch the containers for you. Moving forward, do expect Docker Compose to be the standard way to do linking.

## Part 9 — Writing a Dockerfile

In this part, we shall take a look at how to build our own Docker images via a Dockerfile. We saw building our own image earlier via running a Container, installing our software and doing a commit to create the image in [Part 5](#). However, writing a Dockerfile is a more consistent and repeatable way to build your own images.

A Dockerfile is a text file that has a series of instructions on how to build your image. It supports a simple set of commands that you need to use in your Dockerfile. There are several commands supported like FROM, CMD, ENTRYPOINT, VOLUME, ENV and more. We shall look at some of them.

Let us first start with the the overall flow, which goes something like this:

1. You create a Dockerfile with the required instructions.
2. Then you will use the docker build command to create a Docker image based on the Dockerfile that you created in step 1.

With this information, let us get going.

First up, launch boot2docker and create a folder named images as shown below. This will be our root folder where we will create our Dockerfile.

```
docker@boot2docker:~$ mkdir images
```

Then we navigate into that directory via cd images.

Now, open up the vi editor and create our first Dockerfile as shown below:

```
FROM busybox:latest
MAINTAINER Romin Irani (email@domain.com)
```

Since, a Docker image is nothing but a series of layers built on top of each other, we start with a base image. The [FROM](#) command sets the base image for the rest of the instructions. The MAINTAINER command tells who is the author of the generated images. This is a good practice. You could have taken any other base image in the FROM instruction too, for e.g. ubuntu:latest or ubuntu:14.04, etc.

Now, save the file and come back to the prompt.

Execute the following in the **/images** folder as shown below:

```
$ docker build -t myimage:latest .  
Sending build context to Docker daemon 2.048 kB
```



```
Sending build context to Docker daemon
Step 0 : FROM busybox:latest
---> 8c2e06607696
Step 1 : MAINTAINER Romin Irani (email@domain.com)
---> Running in 5d70f02a83e1
---> 3bc3545a1f64
Removing intermediate container 5d70f02a83e1
Successfully built 3bc3545a1f64
$
```

What we have done here is the step 2 that we highlighted in the overall process above i.e. we have executed the docker build command. This command is used to build a Docker image. The parameters that we have passed are:

- -t is the Docker image tag. You can give a name to your image and a tag.
- The second parameter (a '.') specifies the location of the Dockerfile that we created. Since we created the Dockerfile in the same folder in which we are running the docker build, we specified the current directory.

Notice the various steps that the build process goes through to build out your image.

If you run a docker images command now, you will see the myimage image listed in the output as shown below:

```
$ docker images
REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE
myimage latest 3bc3545a1f64 3 minutes ago 2.433 MB
```

You can now launch a container, any time via the standard docker run command:

```
$ docker run -it myimage
/#
```

And we are back into the **myimage** shell.

Now, let us modify the Dockerfile with a new instruction [CMD](#) as shown below:

```
FROM busybox:latest
MAINTAINER Romin Irani (email@domain.com)
CMD ["date"]
```

Now, build the image and run a container based on it again. You will find that it printed out the date for you as shown below:

```
$ docker run -it myimage
```

```
Thu Dec 14 11:14:42 UTC 2017
```

The CMD instruction takes various forms and when it is used individually in the file without the ENTRYPOINT command (which we will see in a while), it takes the following format:

```
CMD ["executable","param1","param2"]
```

So in our case, we provided the date command as the executable and when we ran a container based on the myimage now, it printed out the data.

In fact, while launching the container, you can override the default CMD by providing it at the command line as shown below. In this example, we are saying to launch the shell , thereby overriding the default CMD instruction for the Docker Image. Notice that it will lead us into the shell.

```
$ docker run -it myimage /bin/sh
```

```
/ #
```

Exercise: Try out modifying the CMD instruction. Give some other commands like CMD [“ls”,“-al”] , build the image and run a container based on that. The best practice is to use another command [ENTRYPOINT](#) together with CMD. The ENTRYPOINT is used to specify the default app that you want to run (This is the way to configure a container that will run as an executable.) The CMD will then provide only the list of parameters to that ENTRYPOINT application. You could still override the CMD parameters at the command line while launching the container. Let us understand that with the following example.

Change your Dockerfile to the following:

```
FROM busybox
MAINTAINER Romin Irani (email@domain.com)
ENTRYPOINT [“/bin/cat”]
CMD [“/etc/passwd”]
```

Now when you build the image and run a container as follows:

```
$ docker run -it myimage
```

```
root:x:0:0:root:/root:/bin/sh
```

```
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
```

```
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:100:sync:/bin:/bin/sync
mail:x:8:8:mail:/var/spool/mail:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
operator:x:37:37:Operator:/var:/bin/sh
haldaemon:x:68:68:hald:/:/bin/sh
dbus:x:81:81:dbus:/var/run/dbus:/bin/sh
ftp:x:83:83:ftp:/home/ftp:/bin/sh
nobody:x:99:99:nobody:/home:/bin/sh
sshd:x:103:99:Operator:/var:/bin/sh
default:x:1000:1000:Default non-root user:/home/default:/bin/sh
```

So, what happened what it took the default ENTRYPOINT i.e. cat command and used the parameters that the CMD provided to run the command.

Try to override the CMD by running the container with a non-existent file name:

```
$ docker run -it myimage somefile.txt
```

```
cat: can't open 'somefile.txt': No such file or directory
```

You get the point?

Now, let us look at another Dockerfile shown below:

```
FROM ubuntu
MAINTAINER Romin Irani (email@domain.com)
RUN apt-get update
RUN apt-get install -y nginx
ENTRYPOINT ["/usr/sbin/nginx","-g","daemon off;"]
EXPOSE 80
```

Here, what we are building is an image that will run the nginx proxy server for us. Look at the set of instructions and it should be pretty clear. After the standard FROM and MAINTAINER instructions, we are executing a couple of RUN instructions. A [RUN](#) instruction is used to execute any commands. In this case we are running a package update and then installing nginx. The ENTRYPOINT is then running the nginx executable and we are using the EXPOSE command here to inform what port the container will be listening on. Remember in our earlier chapters, we saw that if we use the -P command, then the EXPOSE port will be used by default. However, you can always change the host port via the -p parameter as needed.

If you build the image and run the container as follows:

**`docker run -d -p 80:80 --name webserver myimage`**

You will find that it will have nginx started on port 80. And if you visit the page via the host IP, you will see the following point:

# Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to [nginx.org](http://nginx.org).  
Commercial support is available at [nginx.com](http://nginx.com).

*Thank you for using nginx.*

Now, how about adding your own pages into the nginx server instead of the default page.

So, let's say that you were in the images folder still and have the Dockerfile present over there. Create an index.html file over there with just some simple text like `<h1>Hello Docker</h1>`

The updated Dockerfile is shown below:

```
FROM ubuntu
MAINTAINER Romin Irani
RUN apt-get update
RUN apt-get install -y nginx
COPY index.html /usr/share/nginx/html/
ENTRYPOINT ["usr/sbin/nginx","-g","daemon off;"]
EXPOSE 80
```

Notice a new command COPY. This will copy the files/folders from the source to the destination in the container. On building and running this container, you will see your default page. Learn about the [ADD](#) instruction too.

Using these commands now, it should be clear to you how you can now begin to package Docker images with your software. Just think in terms of all the steps that you would normally do to install and run your software.

There is a lot more to writing Dockerfiles and I suggest that via this part, you have understood the basics of writing them. The best way is to jump into writing a file, playing with the commands and using some of the best practices (an article which I have referenced in the Additional Reading section).

## Additional Reading

There are plenty of resources on writing Dockerfiles. Here are 3 that you should go through:

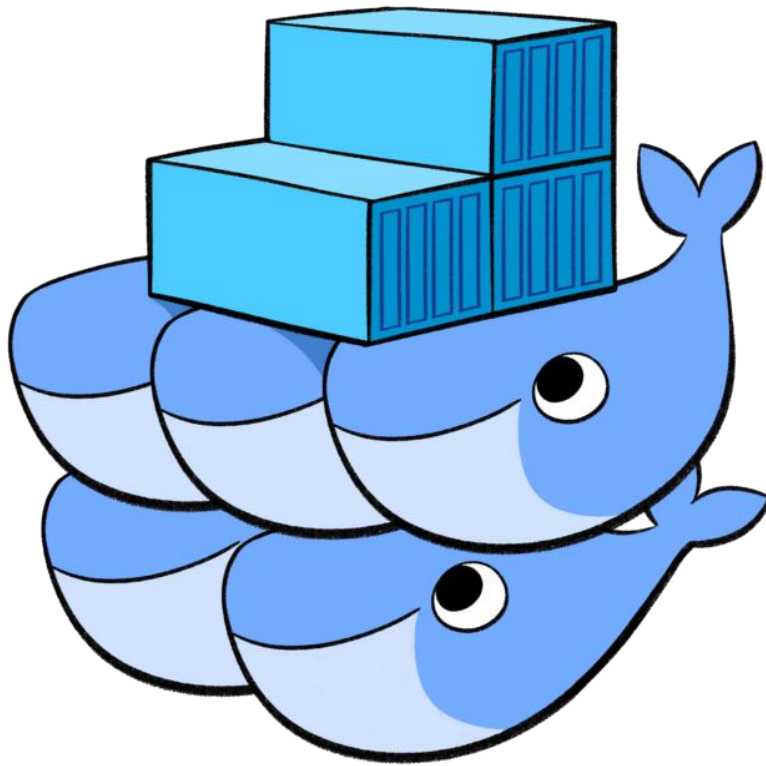
- [Official documentation](#) as a reference to understand any command. You should go through this.
- [Best Practices](#) article on writing Docker files that I recommend.
- [Test](#) your knowledge on Dockerfile.

## Final Comments

With this episode, we come to the end of my Docker Tutorial Series. I hope that it has given you enough to understand how Docker works and the possibilities that it can open up. All the best in your Docker journey.

## Part 10 - Docker Swarm Tutorial

Container Orchestration systems is where the next action is likely to be in the movement towards Building → Shipping → Running containers at scale. The list of software that currently provides a solution for this are [Kubernetes](#), [Docker Swarm](#), Apache Mesos and other.



This tutorial is going to be about exploring the new [Docker Swarm](#) mode, where the Container Orchestration support got baked into the Docker toolset itself.

While I do understand Kubernetes and have tried it out, this blog post represents my own learnings and exploring out Docker Swarm mode.

*Update: Once you are done with this tutorial, you might to check up a follow up tutorial on how to run [Docker Swarm on Google Compute Engine](#).*

*Download a mini book (about 50 pages) that contains both the Docker Swarm Tutorial and Docker Swarm on Google Compute Engine. Click [here](#) to download the PDF.*

# Why do we want a Container Orchestration System?

To keep this simple, imagine that you had to run hundreds of containers. You can easily see that if they are running in a distributed mode, there are multiple features that you will need from a management angle to make sure that the cluster is up and running, is healthy and more.

Some of these necessary features include:

- Health Checks on the Containers
- Launching a fixed set of Containers for a particular Docker image
- Scaling the number of Containers up and down depending on the load
- Performing rolling update of software across containers
- and more...

Let us look at how we can do some of that using Docker Swarm. The Docker Documentation and tutorial for trying out Swarm mode has been excellent.

## Pre-requisites

- You are familiar with basic Docker commands
- You have [Docker Toolbox](#) installed on your system
- You have the Docker version 1.12 atleast

## Create Docker Machines

The first step is to create a set of Docker machines that will act as nodes in our Docker Swarm. I am going to create 6 Docker Machines, where one of them will act as the Manager (Leader) and the other will be worker nodes. You can create less number of machines as needed.

I use the standard command to create a Docker Machine named **manager1** as shown below:

```
docker-machine create --driver hyperv manager1
```

Keep in mind that I am doing this on Windows 10, which uses the native Hyper-V manager so that's why I am using that driver. If you are using the Docker Toolbox with Virtual Box, it would be something like this:

**docker-machine create --driver virtualbox manager1**

Similarly, create the other worder nodes. In my case, as mentioned, I have created 5 other worker nodes.

After creating, it is advised that you fire the **docker-machine ls** command to check on the status of all the Docker machines (I have omitted the DRIVER .

NAME	DRIVER	URL	STATE
manager1	hyperv	tcp://192.168.1.8:2376	Running
worker1	hyperv	tcp://192.168.1.9:2376	Running
worker2	hyperv	tcp://192.168.1.10:2376	Running
worker3	hyperv	tcp://192.168.1.11:2376	Running
worker4	hyperv	tcp://192.168.1.12:2376	Running
worker5	hyperv	tcp://192.168.1.13:2376	Running

Note down the IP Address of the manager1, since you will be needing that. I will call that **MANAGER IP** in the text later.

One way to get the IP address of the manager1 machine is as follows:

```
$ docker-machine ip manager1
192.168.1.8
```

You should be comfortable with doing a SSH into any of the Docker Machines. You will need that since we will primarily be executing the docker commands from within the SSH session to that machine.

Keep in mind that using docker-machine utility, you can SSH into any of the machines as follows:

**docker-machine ssh <machine-name>**

As an example, here is my SSH into **manager1** docker machine.

```
$ docker-machine ssh manager1
```

[illegible]



[illegible]

On my machine, it looks like this:

```
docker@manager1:~$ docker swarm init -- advertise-addr 192.168.1.8
Swarm initialized: current node (5oof62fetd4gry7o09jd9e0kf) is now a manager.
To add a worker to this swarm, run the following command:
docker swarm join \
  -- token SWMTKN-1-5mgvf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-ad7b1k8k3bl3aa3k3q13zivqd \
  192.168.1.8:2377
To add a manager to this swarm, run 'docker swarm join-token manager' and follow the instructions.
docker@manager1:~$
```

Great!

You will also notice that the output mentions the docker swarm join command to use in case you want another node to join as a worker. Keep in mind that you can have a node join as a worker or as a manager. At any point in time, there is only one **LEADER** and the other manager nodes will be as backup in case the current **LEADER** opts out.

At this point you can see your Swarm status by firing the following command as shown below:

```
docker@manager1:~$ docker node ls
ID             HOSTNAME STATUS AVAILABILITY MANAGER STATUS
5oof62fetd.*   manager1 Ready Active Leader
```

This shows that there is a single node so far i.e. manager1 and it has the value of Leader for the **MANAGER** column.

Stay in the SSH session itself for **manager1**.

## Joining as Worker Node

To find out what docker swarm command to use to join as a node, you will need to use the join-token <role> command.

To find out the join command for a **worker**, fire the following command:

```
docker@manager1:~$ docker swarm join-token worker
To add a worker to this swarm, run the following command:
docker swarm join \
  -- token SWMTKN-1-5mgvf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-ad7b1k8k3bl3aa3k3q13zivqd \
  192.168.1.8:2377
docker@manager1:~$
```

## Joining as Manager Node

To find out the the join command for a manager, fire the following command:

```
docker@manager1:~$ docker swarm join-token manager
To add a manager to this swarm, run the following command:
docker swarm join \
— token SWMTKN-1-5mgvf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-8xo0cmd6bryjrsh6w7op4enos \
192.168.1.8:2377
docker@manager1:~$
```

Notice in both the above cases, that you are provided a token and it is joining the Manager node (you will be able to identify that the IP address is the same the **MANAGER\_IP** address).

Keep the SSH to manager1 open. And fire up other command terminals for working with other worker docker machines.

## Adding Worker Nodes to our Swarm

Now that we know how to check the command to join as a worker, we can use that to do a SSH into each of the worker Docker machines and then fire the respective join command in them.

In my case, I have 5 worker machines (worker1/2/3/4/5). For the first worker1 Docker machine, I do the following:

- SSH into the worker1 machine i.e. `docker-machine ssh worker1`
- Then fire the respective command that I got for joining as a worker. In my case the output is shown below:

```
docker@worker1:~$ docker swarm join \
— token SWMTKN-1-5mgvf6ehuc5pfbmar00njd3oxv8nmjhteejaald3yzbef7osl1-ad7b1k8k3bl3aa3k3q13zivqd \
192.168.1.8:2377
This node joined a swarm as a worker.
docker@worker1:~$
```

I do the same thing by launching SSH sessions for worker2/3/4/5 and then pasting the same command since I want all of them to be worker nodes.

After making all my worker nodes join the Swarm, I go back to my manager1 SSH session and fire the following command to check on the status of my Swarm i.e. see the nodes participating in it:

```

docker@manager1:~$ docker node ls
ID                HOSTNAME STATUS AVAILABILITY MANAGER STATUS
1ndqsslh7fpqc7fi35leig54   worker4 Ready Active
1qh4aat24nts5izo3cgsboy77   worker5 Ready Active
25nwmw5eg7a5ms4ch93aw0k03   worker3 Ready Active
5oof62fetd4gry7o09jd9e0kf * manager1 Ready Active Leader
5pm9f2pZR8ndijqkkblkqbsf    worker2 Ready Active
9yq4lcmfg0382p39euk8lj9p4    worker1 Ready Active
docker@manager1:~$

```

As expected, you can see that I have 6 nodes, one as the manager (manager1) and the other 5 as workers.

We can also do execute the standard docker info command here and zoom into the Swarm section to check out the details for our Swarm.

```

Swarm: active
NodeID: 5oof62fetd4gry7o09jd9e0kf
Is Manager: true
ClusterID: 6z3sqr1aqank2uimyzijzapz3
Managers: 1
Nodes: 6
Orchestration:
  Task History Retention Limit: 5
Raft:
  Snapshot Interval: 10000
  Heartbeat Tick: 1
  Election Tick: 3
Dispatcher:
  Heartbeat Period: 5 seconds
CA Configuration:
  Expiry Duration: 3 months
Node Address: 192.168.1.8

```

Notice a few of the properties:

- The Swarm is marked as active. It has 6 Nodes in total and 1 manager among them.
- Since I am running the **docker info** command on the manager1 itself, it shows the **Is Manager** as **true**.
- The Raft section is the Raft consensus algorithm that is used. Check out the details [here](#).

## Create a Service

Now that we have our swarm up and running, it is time to schedule our containers on it. This is the whole beauty of the orchestration layer. We are going to focus on the app and not worry about where the application is going to run.

All we are going to do is tell the manager to run the containers for us and it will take care of scheduling out the containers, sending the commands to the nodes and distributing it.

To start a service, you would need to have the following:

- What is the Docker image that you want to run. In our case, we will run the standard **nginx** image that is officially available from the Docker hub.
- We will expose our service on port 80.
- We can specify the number of containers (or instances) to launch. This is specified via the **replicas** parameter.
- We will decide on the name for our service. And keep that handy.

What I am going to do then is to launch 5 replicas of the nginx container. To do that, I am again in the SSH session for my manager1 node. And I give the following **docker service create** command:

```
docker service create --replicas 5 -p 80:80 --name web nginx
ctolqlt4h2o859t69j9pptye
```

What has happened is that the Orchestration layer has now got to work.

You can find out the status of the service, by giving the following command:

```
docker@manager1:~$ docker service ls
ID          NAME REPLICAS IMAGE COMMAND
ctolqlt4h2o8 web  0/5     nginx
```

This shows that the replicas are not yet ready. You will need to give that command a few times.

In the meanwhile, you can also see the status of the service and how it is getting orchestrated to the different nodes by using the following command:

```
docker@manager1:~$ docker service ps web
ID NAME IMAGE NODE DESIRED STATE CURRENT STATE ERROR
```

```

7i* web.1 nginx worker3 Running    Preparing 2 minutes ago
17* web.2 nginx manager1 Running   Running 22 seconds ago
ey* web.3 nginx worker2 Running    Running 2 minutes ago
bd* web.4 nginx worker5 Running    Running 45 seconds ago
dw* web.5 nginx worker4 Running    Running 2 minutes ago

```

This shows that the nodes are getting setup. It could take a while.

But notice a few things. In the list of nodes above, you can see that the 5 containers are being scheduled by the orchestration layer on **manager1, worker2, worker3, worker4 and worker5**. There is no container scheduled for **worker1** node and that is fine.

A few executions of **docker service ls** shows the following responses:

```

docker@manager1:~$ docker service ls
ID            NAME REPLICAS IMAGE COMMAND
ctolq1t4h2o8 web  3/5    nginx
docker@manager1:~$

```

and then finally:

```

docker@manager1:~$ docker service ls
ID            NAME REPLICAS IMAGE COMMAND
ctolq1t4h2o8 web  5/5    nginx
docker@manager1:~$

```

If we look at the service processes at this point, we can see the following:

```

docker@manager1:~$ docker service ps web
ID NAME IMAGE NODE    DESIRED STATE CURRENT STATE    ERROR
7i* web.1 nginx worker3 Running    Running 4 minutes ago
17* web.2 nginx manager1 Running    Running 7 minutes ago
ey* web.3 nginx worker2 Running    Running 9 minutes ago
bd* web.4 nginx worker5 Running    Running 8 minutes ago
dw* web.5 nginx worker4 Running    Running 9 minutes ago

```

If you do a **docker ps** on the manager1 node right now, you will find that the nginx daemon has been launched.

```

docker@manager1:~$ docker ps
CONTAINER ID   IMAGE      COMMAND                  CREATED        STATUS        PORTS
933309b04630   nginx:latest "nginx -g 'daemon off'" 2 minutes ago  Up 2 minutes  80/tcp, 443/tcp

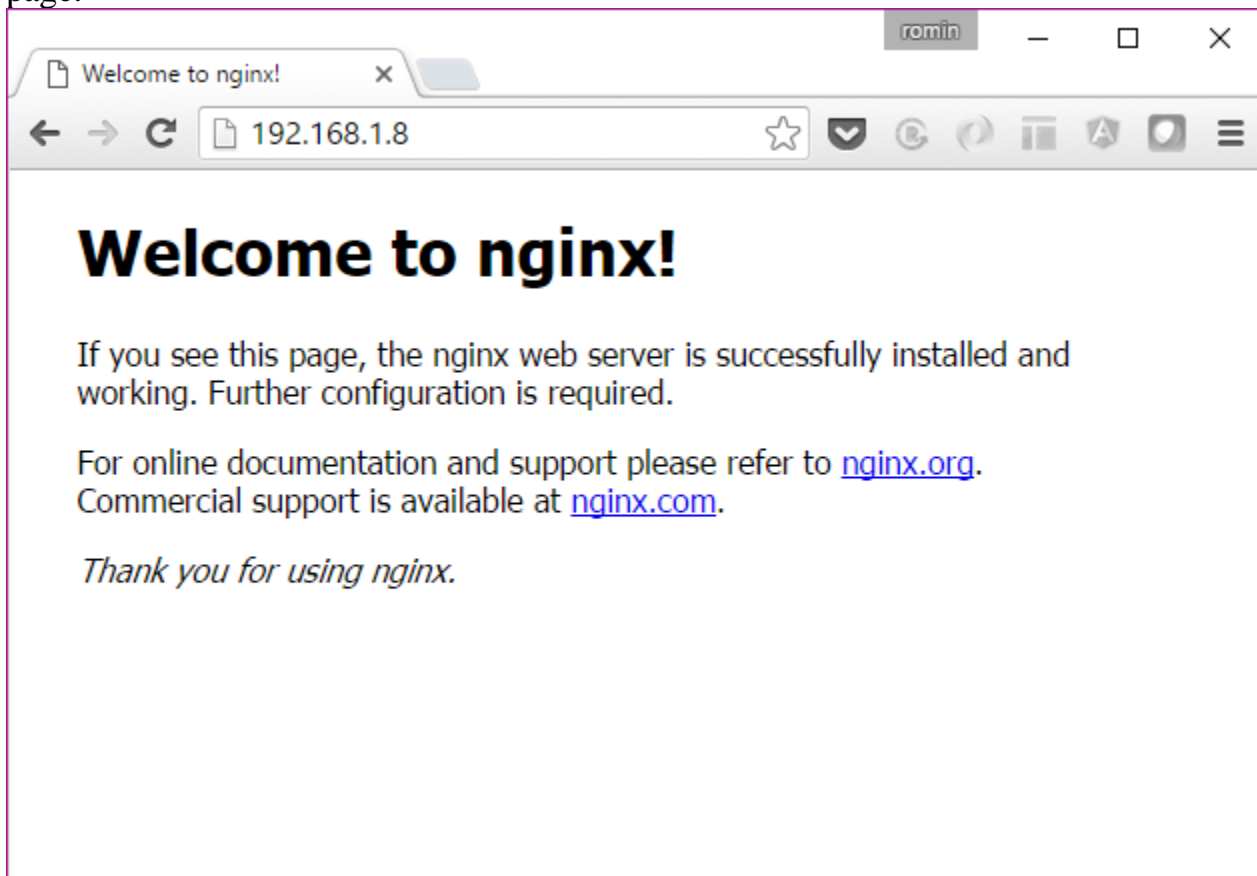
```

```
web.2.17d502y6qjhd1wqjle13nmjvc  
docker@manager1:~$
```

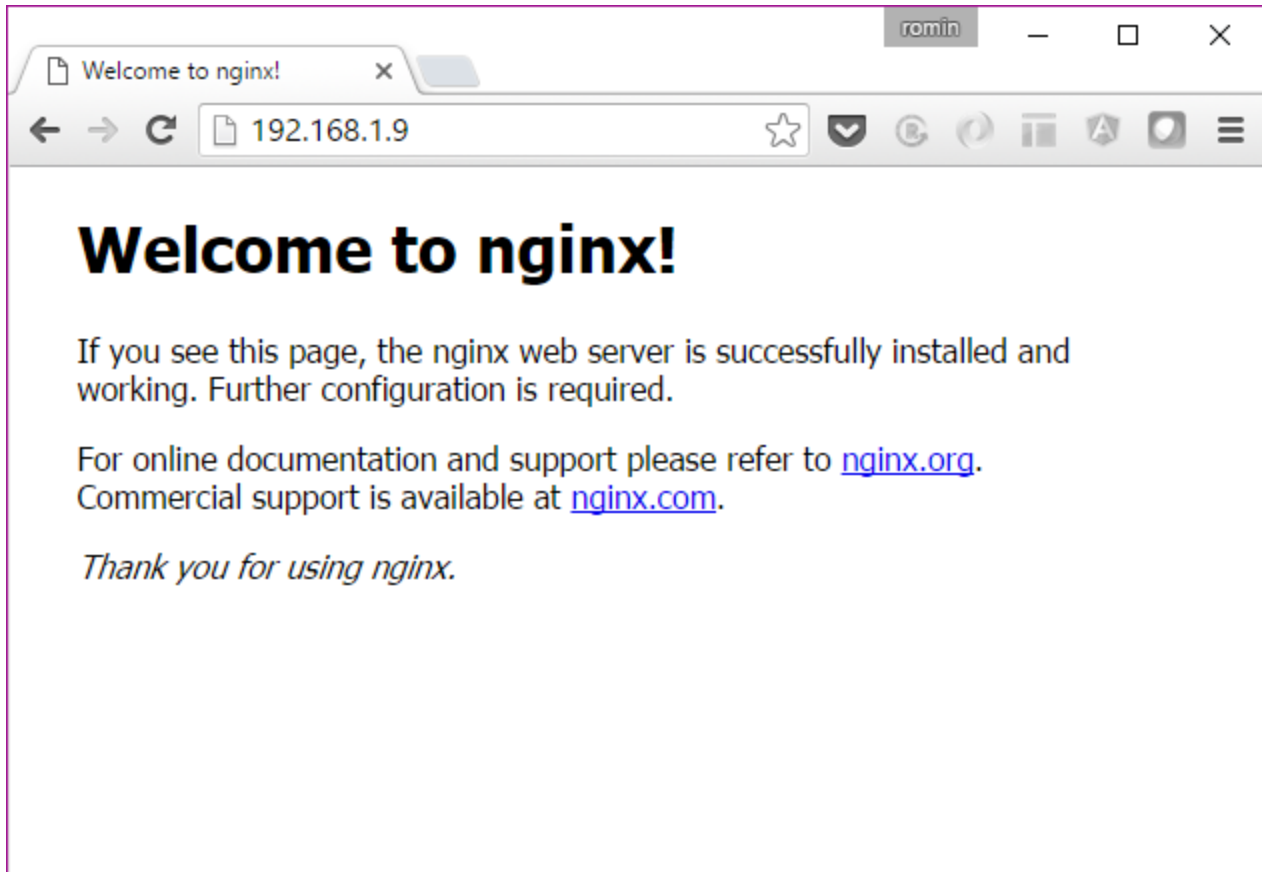
## Accessing the Service

You can access the service by hitting any of the manager or worker nodes. It does not matter if the particular node does not have a container scheduled on it. That is the whole idea of the swarm.

Try out a curl to any of the Docker Machine IPs (manager1 or worker1/2/3/4/5) or hit the URL (`http://<machine-ip>`) in the browser. You should be able to get the standard NGINX Home page.



or if we hit the worker IP:



Nice, isn't it?

Ideally you would put the Docker Swarm service behind a Load Balancer.

## Scaling up and Scaling down

This is done via the **docker service scale** command. We currently have 5 containers running. Let us bump it up to 8 as shown below by executing the command on the **manager1** node.

```
$ docker service scale web=8
web scaled to 8
```

Now, we can check the status of the service and the process tasks via the same commands as shown below:

```
docker@manager1:~$ docker service ls
ID                NAME REPLICAS IMAGE COMMAND
ctolq1t4h2o8     web   5/8      nginx
```



In the ps web command below, you will find that it has decided to schedule the new containers on worker1 (2 of them) and manager1(one of them)

```
docker@manager1:~$ docker service ps web
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
7i*	web.1	nginx	worker3	Running	Running 14 minutes ago	
17*	web.2	nginx	manager1	Running	Running 17 minutes ago	
ey*	web.3	nginx	worker2	Running	Running 19 minutes ago	
bd*	web.4	nginx	worker5	Running	Running 17 minutes ago	
dw*	web.5	nginx	worker4	Running	Running 19 minutes ago	
8t*	web.6	nginx	worker1	Running	Starting about a minute ago	
b8*	web.7	nginx	manager1	Running	Ready less than a second ago	
0k*	web.8	nginx	worker1	Running	Starting about a minute ago	

We wait for a while and then everything looks good as shown below:

```
docker@manager1:~$ docker service ls
```

ID	NAME	REPLICAS	IMAGE	COMMAND
ctolqlt4h2o8	web	8/8	nginx	

```
docker@manager1:~$ docker service ps web
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	ERROR
7i*	web.1	nginx	worker3	Running	Running 16 minutes ago	
17*	web.2	nginx	manager1	Running	Running 19 minutes ago	
ey*	web.3	nginx	worker2	Running	Running 21 minutes ago	
bd*	web.4	nginx	worker5	Running	Running 20 minutes ago	
dw*	web.5	nginx	worker4	Running	Running 21 minutes ago	
8t*	web.6	nginx	worker1	Running	Running 4 minutes ago	
b8*	web.7	nginx	manager1	Running	Running 2 minutes ago	
0k*	web.8	nginx	worker1	Running	Running 3 minutes ago	

```
docker@manager1:~$
```

## Inspecting nodes

You can inspect the nodes anytime via the docker node inspect command.

For example if you are already on the node (for example manager1) that you want to check, you can use the name self for the node.

```
$ docker node inspect self
```

Or if you want to check up on the other nodes, give the node name. For e.g.

```
$ docker node inspect worker1
```

# Draining a node

If the node is **ACTIVE**, it is ready to accept tasks from the Master i.e. Manager. For e.g. we can see the list of nodes and their status by firing the following command on the **manager1** node.

```
docker@manager1:~$ docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
1ndqsslh7fpquc7fi35leig54	worker4	Ready	Active	
1qh4aat24nts5izo3cgsboy77	worker5	Ready	Active	
25nwmw5eg7a5ms4ch93aw0k03	worker3	Ready	Active	
5oof62fetd4gry7o09jd9e0kf *	manager1	Ready	Active	Leader
5pm9f2pzr8ndijqkblkgqbsf	worker2	Ready	Active	
9yq4lcmfg0382p39euk8lj9p4	worker1	Ready	Active	

```
docker@manager1:~$
```

You can see that their **AVAILABILITY** is set to **READY**.

As per the documentation, When the node is active, it can receive new tasks:

- during a service update to scale up
- during a rolling update
- when you set another node to Drain availability
- when a task fails on another active node

But sometimes, we have to bring the Node down for some maintenance reason. This meant by setting the Availability to Drain mode. Let us try that with one of our nodes.

But first, let us check the status of our processes for the web services and on which nodes they are running:

```
docker@manager1:~$ docker service ps web
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE
ERROR					
7i*	web.1	nginx	worker3	Running	Running 54 minutes ago
17*	web.2	nginx	manager1	Running	Running 57 minutes ago
ey*	web.3	nginx	worker2	Running	Running 59 minutes ago
bd*	web.4	nginx	worker5	Running	Running 57 minutes ago
dw*	web.5	nginx	worker4	Running	Running 59 minutes ago
8t*	web.6	nginx	worker1	Running	Running 41 minutes ago
b8*	web.7	nginx	manager1	Running	Running 39 minutes ago
0k*	web.8	nginx	worker1	Running	Running 41 minutes ago

You find that we have 8 replicas of our service:

- 2 on manager1
- 2 on worker1
- 1 each on worker2, worker3, worker4 and worker5

Now, let us use another command to check what is going on in node **worker1**.

```
docker@manager1:~$ docker node ps worker1
ID      NAME      IMAGE      NODE      DESIRED STATE  CURRENT STATE      8t*  web.6
nginx   worker1   Running    Running    Running 44 minutes ago
0k*     web.8     nginx     worker1   Running    Running 44 minutes ago
docker@manager1:~$
```

We can also use the `docker node inspect` command to check the availability of the node and as expected, you will find a section in the output as follows:

```
$ docker node inspect worker1
....
"Spec": {
  "Role": "worker",
  "Availability": "active"
},
...
```

or

```
docker@manager1:~$ docker node inspect --pretty worker1
ID: 9yq4lcmfg0382p39euk8lj9p4
Hostname: worker1
Joined at: 2016-09-16 08:32:24.5448505 +0000 utc
Status:
  State: Ready
  Availability: Active
Platform:
  Operating System: linux
  Architecture: x86_64
Resources:
  CPUs: 1
  Memory: 987.2 MiB
Plugins:
  Network: bridge, host, null, overlay
  Volume: local
Engine Version: 1.12.1
Engine Labels:
  - provider = hypervdocker@manager1:~$
```

We can see that it is “**Active**” for its Availability attribute.

Now, let us set the **Availability** to **DRAIN**. When we give that command, the Manager will stop tasks running on that node and launches the replicas on other nodes with ACTIVE availability.

So what we are expecting is that the Manager will bring the 2 containers running on worker1 and schedule them on the other nodes (manager1 or worker2 or worker3 or worker4 or worker5).

This is done by updating the node by setting its availability to “drain”.

```
docker@manager1:~$ docker node update --availability drain worker1
worker1
```

Now, if we do a process status for the service, we see an interesting output (*I have trimmed the output for proper formatting*):

```
docker@manager1:~$ docker service ps web
ID      NAME      IMAGE      NODE      DESIRED STATE  CURRENT STATE
7i*     web.1     nginx     worker3   Running        Running about an hour ago
17*     web.2     nginx     manager1  Running        Running about an hour ago
ey*     web.3     nginx     worker2   Running        Running about an hour ago
bd*     web.4     nginx     worker5   Running        Running about an hour ago
dw*     web.5     nginx     worker4   Running        Running about an hour ago
2u*     web.6     nginx     worker4   Running        Preparing about a min ago
8t*     \_ web.6  nginx     worker1   Shutdown       Shutdown about a min ago
b8*     web.7     nginx     manager1  Running        Running 49 minutes ago
7a*     web.8     nginx     worker3   Running        Preparing about a min ago
0k*     \_ web.8  nginx     worker1   Shutdown       Shutdown about a min ago
docker@manager1:~$
```

You can see that the containers on worker1 (which we have asked to be drained) are being rescheduled on other workers. In our scenario above, they got scheduled to worker2 and worker3 respectively. This is required because we have asked for 8 replicas to be running in an earlier scaling exercise.

You can see that the two containers are still in “Preparing” state and after a while if you run the command, they are all running as shown below:

```
docker@manager1:~$ docker service ps web
ID      NAME      IMAGE      NODE      DESIRED STATE  CURRENT STATE
7i*     web.1     nginx     worker3   Running        Running about an hour ago
17*     web.2     nginx     manager1  Running        Running about an hour ago
ey*     web.3     nginx     worker2   Running        Running about an hour ago
```

bd*	web.4	nginx	worker5	Running	Running about an hour ago
dw*	web.5	nginx	worker4	Running	Running about an hour ago
2u*	web.6	nginx	worker4	Running	Running 8 minutes ago
8t*	\_ web.6	nginx	worker1	Shutdown	Shutdown 8 minutes ago
b8*	web.7	nginx	manager1	Running	Running 56 minutes ago
7a*	web.8	nginx	worker3	Running	Running 8 minutes ago
0k*	\_ web.8	nginx	worker1	Shutdown	Shutdown 8 minutes ago

This makes for cool demo, isn't it?

## Remove the Service

You can simply use the service rm command as shown below:

```
docker@manager1:~$ docker service rm web
web
docker@manager1:~$ docker service ls
ID NAME REPLICAS IMAGE COMMAND
docker@manager1:~$ docker service inspect web
[]
Error: no such service: web
```

## Applying Rolling Updates

This is straight forward. In case you have an updated Docker image to roll out to the nodes, all you need to do is fire an service update command.

For e.g.

```
$ docker service update --image <imagename>:<version> web
```

## Conclusion

I am definitely impressed with the simplicity of Docker Swarm. Just like the basic commands that come with the standard Docker toolset, it has been a good move to introduce the Swarm commands within the same toolset.

There is a lot more to Docker Swarm and I suggest that you dig further into the documentation.

I am not in a position to know whether Kubernetes or Swarm will emerge a winner but there is no doubt that we will have to understand both to see what their capabilities are and then take a decision.

*Update: Once you are done with this tutorial, you might to check up a follow up tutorial on how to run [Docker Swarm on Google Compute Engine](#).*

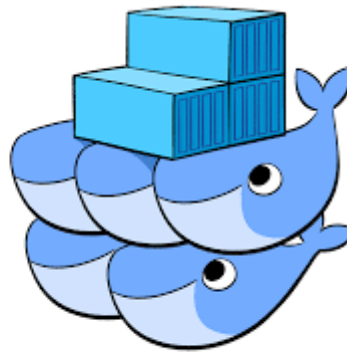
*Download a mini book (about 50 pages) that contains both the Docker Swarm Tutorial and Docker Swarm on Google Compute Engine. Click [here](#) to download the PDF.*

## Part 11— Docker Swarm on Google Compute Engine

This is a follow up of my [Docker Swarm Tutorial](#). I strongly suggest that you go through that if you are not familiar with the basics of Docker Swarm.

This tutorial builds on the previous tutorial by going through the following:

- Setup a Docker Swarm on [Google Compute Engine](#)
- Experiment multiple scenarios like setting up multiple Swarm Manager, bringing a Swarm Manager down, setting up an Overlay network and more.



*Download a mini book (about 50 pages) that contains both the Docker Swarm Tutorial and Docker Swarm on Google Compute Engine. Click [here](#) to download the PDF.*

The first step is to create the Docker Swarm cluster on Google Compute Engine.

### Prerequisites

This tutorial assumes that you have setup the following on your machine:

- Latest version of [Docker Toolbox](#). Ensure that it is 1.12 or higher.
- Latest version of [Google Cloud SDK Tools](#). Please download it from the link.

### Google Cloud Platform Project

I suggest that you create a new [Google Cloud Platform Project](#) for this tutorial. But if you are familiar with the platform and wish to use an existing project, that is fine too. Note down the Project Id for the Google Cloud Platform project.

On your local machine, where you have already setup Docker Toolbox + Google Cloud Platform tools and assuming that you have the Google Cloud Platform project id handy, **initialize** the project using the gcloud utility that you have setup.

```
$ gcloud init
```

and go ahead with the rest of the steps, ensure that you select the zone/region of your choice and most importantly the project id.

To ensure that you are all set, just fire the following command and note if the properties are setup correctly. You should see similar values.

```
$ gcloud config list
```

```
Your active configuration is: [<your-config-name>]
[compute]
region = <your-selected-region> e.g. us-central1
zone = <your-selected-zone> e.g. us-central1-a
[core]
account = <your-email-id>
disable_usage_reporting = True
project = <YOUR_GCP_PROJECT_ID>
```

If the project is not set, I suggest that you do so with the following command:

```
$ gcloud config set project <YOUR_GCP_PROJECT_ID>
```

## Creating the Docker Machines

The first step to creating the swarm is to provision the Docker machines. By that, we mean that we will be provisioning Compute Engine instances. We are going to have the following setup:

- 5 Compute Engine instances, all setup with docker and provisioned using the **docker-machine** utility that is part of the Docker Toolbox.



- We are going to have 3 Managers in the Swarm and 2 Workers in the Swarm. We need to give the names to our Compute Engine instances, so we will name them **mgr-1, mgr-2, mgr-3** and **w-1 & w-2**.

To provision a Docker machine (Host) on compute engine, we use the following command (for mgr-1). Note that we are using the Google Cloud **driver**, specifying the **machine type** (n1-standard-1), giving a **tag** to all our machines and specifying the **google-project-id**. This is standard **docker-machine** create stuff.

```
$ docker-machine create mgr-1 \
    -d google \
    --google-machine-type n1-standard-1
    --google-tags myswarm
    --google-project <YOUR_GCP_PROJECT_ID>
```

Running pre-create checks...

(mgr-1) Check that the project exists

(mgr-1) Check if the instance already exists

Creating machine...

(mgr-1) Generating SSH Key

(mgr-1) Creating host...

(mgr-1) Opening firewall ports

(mgr-1) Creating instance

(mgr-1) Waiting for Instance

(mgr-1) Uploading SSH Key

Waiting for machine to be running, this may take a few minutes...

Detecting operating system of created instance...

Waiting for SSH to be available...

Detecting the provisioner...

Provisioning with ubuntu(systemd)...

Installing Docker...

Copying certs to the local machine directory...

Copying certs to the remote machine...

Setting Docker configuration on the remote daemon...

Checking connection to Docker...

Docker is up and running!

To see how to connect your Docker Client to the Docker Engine running on this virtual machine, run: `docker-machine env mgr-1`

We do the same for mgr-2, mgr-3, w-1 and w-2. On successful creation, we can use the `docker-machine ls` command to check on our Docker machines. The output should be similar to the one that I got below (Note that I have removed the SWARM and the ERRORS column from the output):

```
$ docker-machine ls
```

NAME	ACTIVE	DRIVER	STATE	URL	DOCKER
mgr-1	—	google	Running	tcp://130.211.199.228:2376	v1.12.1
mgr-2	—	google	Running	tcp://104.154.244.185:2376	v1.12.1
mgr-3	—	google	Running	tcp://104.154.56.35:2376	v1.12.1

```
w-1    -    google Running tcp://107.178.213.86:2376  v1.12.1
w-2    -    google Running tcp://8.34.214.144:2376   v1.12.1
```

At this point, you could also use the `gcloud compute instances list` command to see the list of VMs that have been provisioned. In the listing you should see Compute Engine VMs as listed below:

#### **\$ gcloud compute instances list**

NAME	ZONE	MACHINE_TYPE	PREEMPTIBLE	INTERNAL_IP	EXTERNAL_IP	STATUS
mgr-1	us-central1-a	n1-standard-1	10.240.0.2	130.211.199.228	RUNNING	
mgr-2	us-central1-a	n1-standard-1	10.240.0.3	104.154.244.185	RUNNING	
mgr-3	us-central1-a	n1-standard-1	10.240.0.4	104.154.56.35	RUNNING	
w-1	us-central1-a	n1-standard-1	10.240.0.5	107.178.213.86	RUNNING	
w-2	us-central1-a	n1-standard-1	10.240.0.6	8.34.214.144	RUNNING	

Each of the VMs has been assigned an internal and external IP. The status of all the machines is also in RUNNING state.

## Creating the Swarm











In the [Docker Swarm tutorial](#), we had seen how to create the Swarm. To reiterate, we are going to create a Swarm with:

- 3 Manager nodes (This will make 1 as the LEADER and the other 2 as Available)
- 2 Worker nodes

## SSH into Google Compute Engine VMs

One of the nice features of Google Compute Engine VMs is that you have a SSH button right next to your list of Compute Engine VMs, which you can click and get into a SSH session with that VM.

If you go to the Google Cloud Console and then select Compute Engine, you will see a list that looks something like this:

<input type="checkbox"/> Name ^	Zone	Machine type	Recommendation	In use by	Internal IP	External IP	
<input type="checkbox"/>  mgr-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.2	130.211.199.228 	S
<input type="checkbox"/>  mgr-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.3	104.154.244.185 	S
<input type="checkbox"/>  mgr-3	us-central1-a	1 vCPU, 3.75 GB			10.240.0.4	104.154.56.35 	S
<input type="checkbox"/>  w-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.5	107.178.213.86 	S
<input type="checkbox"/>  w-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.6	8.34.214.144 	S

Notice the SSH button to the extreme right for each machine that we created. Click on that to launch the SSH session for any of the machines. I will use the title SSH to mgr-1 session and so on to indicate which machine I am on.

*Note: You can also use the **docker-machine env** command on your local machine to set the environment variables that will allow the docker client to connect to a specific machine. If you are comfortable with it, use it by all means.*

*Note: You will notice the **sudo** prefix before the docker commands. If you want to avoid that, you should consider adding the user with root privileges to the docker user group. E.g. `sudo usermod -aG docker <user_name>`*

## Initialize the Swarm

First up, note down the Internal IP address of the **mgr-1** instance. You will find that in the Compute Engine VM listing that we saw about in my case, it is 10.240.0.2.

- SSH to mgr-1 docker machine
- Give the following command:

```
romin_irani@mgr-1:~$ sudo docker swarm init --advertise-addr 10.240.0.2
Swarm initialized: current node (616qh3d1b6hps9ic095wsor27) is now a manager.
To add a worker to this swarm, run the following command:
docker swarm join \
--token SWMTKN-1-4lon4th27153xvrpruohbld5lciux02rxs9go9fdt2672cdkhu-
69j9grxvmri7wni2k134m4dmw \
10.240.0.2:2377
To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.
romin_irani@mgr-1:~$
```

At this point, we have only one node in our Swarm as shown below:

```
romin_irani@mgr-1:~$ sudo docker node ls
ID            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
616.. * mgr-1    Ready   Active                Leader
```

To join the other nodes as workers or managers, we just have to know what is the token and ip to use as part of the docker swarm join command. This is made easy to simply executing the join-token <role> on the current master i.e. mgr-1 and noting down the commands:

```
romin_irani@mgr-1:~$ sudo docker swarm join-token manager
```

To add a manager to this swarm, run the following command:

```
docker swarm join \
  --token SWMTKN-1-4lon4th27l53xvrpruohbld5lciux02rxs9go9fdt2672cdkhu-
43c7vexnensp8mkv
wl09preu7 \
  10.240.0.2:2377
```

```
romin_irani@mgr-1:~$ sudo docker swarm join-token worker
```

To add a worker to this swarm, run the following command:

```
docker swarm join \
  --token SWMTKN-1-4lon4th27l53xvrpruohbld5lciux02rxs9go9fdt2672cdkhu-
69j9grxvmri7wni2
k134m4dmw \
  10.240.0.2:2377
```

You can now open up SSH sessions on each of the nodes. Remember that on mgr-2 and mgr-3 , we want to execute the command to join as a manager. And on w-1 and w-2 nodes, we want to execute the command to join as a worker.

Complete the following and your Docker Swarm is now ready.

If we run the command to list down the Swarm nodes on mgr-1, we should get the following output:

```
romin_irani@mgr-1:~$ sudo docker node ls
ID            HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS
011..        mgr-2    Ready   Active                Reachable
4e3..        mgr-3    Ready   Active                Reachable
616.. * mgr-1    Ready   Active                Leader
7xo..        w-1     Ready   Active
8ie..        w-2     Ready   Active
```

You can see from the output above that we have 5 nodes running. Our mgr-1 (Manager) from where we launched the Swarm cluster is now a **Leader**. The other managers are in a **Reachable** state.

Great. Everything looks good for now.

## Creating the Overlay Network

Let us create an overlay network now. An overlay network supports multi-host networking. We are going to be using the overlay network for our swarm services. When you specify an overlay network for your services, Swarm automatically assigns addresses to the containers.

Stay in the SSH session for **mgr-1**. We can look at our current list of networks as follows:

```
romin_irani@mgr-1:~$ sudo docker network list
```

NETWORK ID	NAME	DRIVER	SCOPE
d4b360ee71b4	bridge	bridge	local
9a55643d34e8	docker_gwbridge	bridge	local
b6348ecb0afa	host	host	local
dzzo90eqcmt2	ingress	overlay	swarm
46f4630544d0	none	null	local

You will notice that at the local scope, you have the default bridge and the host network. You will also notice that Docker Swarm created a default overlay network called ingress. As per the [documentation](#), “the swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm.”

Let us go ahead now and create our own overlay network named **nw1**. So on the manager node (**mgr-1**) do the following:

```
$ sudo docker network create --driver overlay nw17ffh8lexsm9fhiukslkyiml02
```

We can now inspect the list of network services as follows:

```
romin_irani@mgr-1:~$ sudo docker network list
```

NETWORK ID	NAME	DRIVER	SCOPE
d4b360ee71b4	bridge	bridge	local
9a55643d34e8	docker_gwbridge	bridge	local
b6348ecb0afa	host	host	local
dzzo90eqcmt2	ingress	overlay	swarm
46f4630544d0	none	null	local
<b>7ffh8lexsm9f</b>	<b>nw1</b>	<b>overlay</b>	<b>swarm</b>

You can see that the overlay network (named **nw1**) is created.

# Creating the Service

Similar to the tutorial on Docker Swarm, we are going to create our standard NGINX service with 6 replicas. This time however, we are going to use the **overlay network (nw1)** that we created so that we can later on see how it all comes together when working with multiple services.

On the **mgr-1** node, execute the following command:

```
romin_irani@mgr-1:~$ sudo docker service create --replicas 6
--network nw1 -p 80:80/tcp --name nginx nginx
0omlto8a98zahgbsqs0ajz159
```

We can see the services as follows:

```
romin_irani@mgr-1:~$ sudo docker service ls
ID                NAME      REPLICAS  IMAGE  COMMAND
0omlto8a98za     nginx    6/6       nginx
```











We can see that 6 containers have been launched for NGINX image. To understand the distribution of these **6 containers** on the **5 nodes** that we have, we can use the following command:

```
romin_irani@mgr-1:~$ sudo docker service ps nginx
ID      NAME      IMAGE  NODE   DESIRED STATE  CURRENT STATE
9z*     nginx.1   nginx  mgr-2  Running        Running 32 seconds ago
6e*     nginx.2   nginx  w-1    Running        Running 32 seconds ago
1o*     nginx.3   nginx  w-1    Running        Running 32 seconds ago
6n*     nginx.4   nginx  mgr-1  Running        Running 32 seconds ago
8l*     nginx.5   nginx  w-2    Running        Running 32 seconds ago
8p*     nginx.6   nginx  mgr-3  Running        Running 32 seconds ago
```

We can see that the Swarm Manager distributed the containers across all the 5 nodes : running 2 containers on **worker node w-1** and distributing the other containers equally across all the remaining nodes.

So at this point, we have the standard NGINX container running on our 5 nodes. These 5 nodes are nothing but our VMs i.e. Google Compute Engine instances. And if you recollect, each of these Compute Engine instances were provided both an internal IP Address and an external IP Address.

You could list out the output of the **gcloud compute instances list** command again. You could either use that from your laptop or just go to the Compute Engine instances list in the Google Cloud console.

<input type="checkbox"/> Name ^	Zone	Machine type	Recommendation	In use by	Internal IP	External IP	
<input type="checkbox"/>  mgr-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.2	130.211.199.228	
<input type="checkbox"/>  mgr-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.3	104.154.244.185	
<input type="checkbox"/>  mgr-3	us-central1-a	1 vCPU, 3.75 GB			10.240.0.4	104.154.56.35	
<input type="checkbox"/>  w-1	us-central1-a	1 vCPU, 3.75 GB			10.240.0.5	107.178.213.86	
<input type="checkbox"/>  w-2	us-central1-a	1 vCPU, 3.75 GB			10.240.0.6	8.34.214.144	

## Internal Connectivity

You can SSH into any of the Compute Engine instances. Notice that the Internal IPs from the list of instances above. Simply use 'curl <InternalIPAddress>' for any of the instances and you should get back the HTML content of the default NGINX home page. So in short, the instances can communicate to each other internally via the Internal IP Addresses.

## External Connectivity

To do this, we will need to create a Firewall rule to allow traffic from outside targeted towards port 80 and we should target the Compute Engine instances that we had tagged earlier with the **myswarm** tag, which we used as value for the

```
--google-tags myswarm
```

while creating the Docker machine.


From the gcloud utility on your local machine, fire the following command:


```
$ gcloud compute firewall-rules create my-swarm-rule --allow tcp:80  
--description "nginx service" --target-tags myswarm
```


On successful creation, you will notice from your web Google Cloud console, that in the Networking → Firewalls list, you have an entry as shown below:


<input type="checkbox"/> Name ^	Source tag / IP range / Subnetworks	Allowed protocols / ports	Target tags	Ne
<input type="checkbox"/> my-swarm-rule	0.0.0.0/0	tcp:80	myswarm	de


The details of which are shown below:



**Networking**



**Firewall rule details**


 Networks


 External IP addresses


 **Firewall rules**

 Routes

 Load balancing

 Cloud DNS

 VPN

 Cloud Routers

**my-swarm-rule**

**Description**  
 nginx service

**Network**  
 default

**Source filter**  
 Allow from any source (0.0.0.0/0)

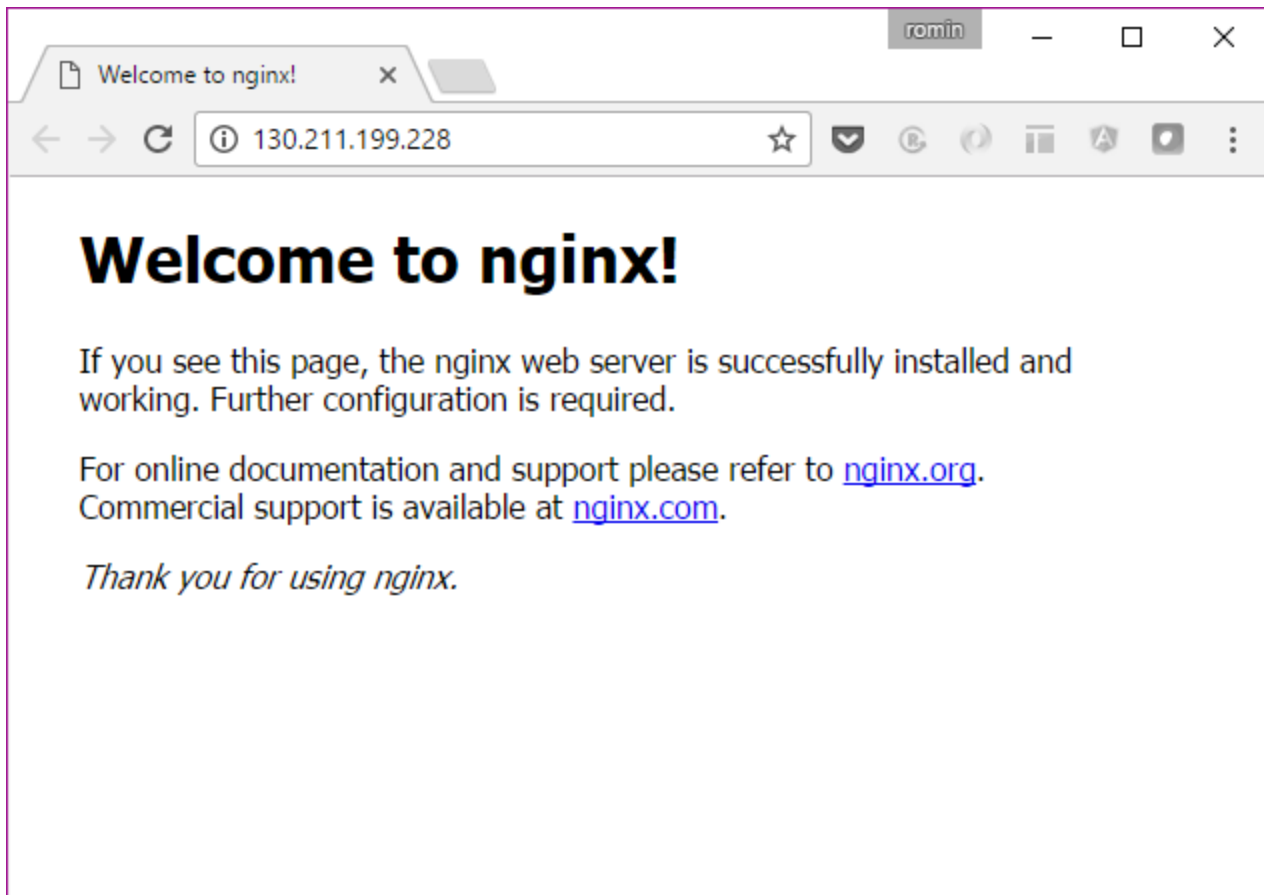
**Allowed protocols and ports**  
 tcp:80

**Target tags**  
 myswarm

Equivalent [REST](#)

Now, if you hit any of the external IP Addresses from the Compute Engine instances list, you will get the NGINX home page as shown below, when I access one of the External IPs:





So, we are now able to access our service from any of the External IP addresses. But this is not what we want to do. We want to put these machines behind a Load Balancer. And simply by hitting the Load Balancer public IP, the traffic should get routed to any of the instances i.e. the nodes in our Docker Swarm Cluster.

For this, we need to create the Load Balancer, that Google Compute Engine provides.

## Creating the Load Balancer

To create the Load Balancer, do the following:

- Go to Google Cloud console.
- Go to Networking → Load Balancing. Click on **Create Load Balancer** button.
- You will see 3 options, click the **Start configuration** button in the **TCP Load Balancing** option as shown below:

### HTTP(S) Load Balancing

Layer 7 load balancing for HTTP and HTTPS applications [Learn more](#)

**Configure**

HTTP LB  
HTTPS LB

**Options**

Internet-facing only  
Multi-region  
Single-region

Start configuration

### TCP Load Balancing

Layer 4 load balancing or proxy for applications that rely on TCP/SSL protocol [Learn more](#)

**Configure**

TCP LB  
SSL Proxy

**Options**

Internet-facing only  
Single or multi-region

Start configuration

### UDP Load Balancing

Layer 4 load balancing for applications rely on UDP protocol [Learn more](#)

**Configure**

UDP LB

**Options**

Internet-facing only  
Single-region

Start configuration

- Click on Continue in the next section and then you will reach the configuration for Backend and Frontend as shown below. Give your Load Balancer a name as shown below:

← New TCP load balancer

Name ?

nginx-lb

- **Backend configuration**  
You have not configured your backend yet
- **Frontend configuration**  
You have not configured your frontend yet
- ⓘ **Review and finalize**  
Optional

Create

Cancel

- Click on Backend configuration next. Select the **region** for your load balancer—I went with the ones in which I had my instances. Then select existing instances, and pick all the nodes (compute engine instances) that we created. Finally select a standard http port 80 healthcheck. This will enable the load balancer to check the health of each nodes. And since our nginx service is running on port 80, a simple healthcheck on port 80 is good enough for now.

### Backend configuration

Name ?

nginx-lb

Region ?

us-central1

Backends ?

Select existing instance groups

Select existing instances

mgr-1 (us-central1-a)

×

mgr-2 (us-central1-a)

×

mgr-3 (us-central1-a)

×

w-1 (us-central1-a)

×

w-2 (us-central1-a)

×

Add an instance

▼

Backup pool ? (Optional)

None

Failover ratio ?

10

%

Health check ?

http-healthcheck

port: 80, timeout: 5s, check interval: 5s, unhealthy threshold: 2 attempts

- Now go to **Front end configuration** and enter the port as 80. We are going with an ephemeral ip for now, but you could also get yourself a Static IP Address.

## Frontend configuration

Specify an IP address, port and protocol. This IP address is the frontend IP for your clients requests.

Protocol	IP	Port	
TCP	Ephemeral	80	X
<a href="#">+ Add frontend IP and port</a>			

- Your review and finalize should look this:

← New TCP load balancer

Name ⓘ

nginx-lb

✓ Backend configuration

Your backend is configured

✓ Frontend configuration

Your frontend is configured

ⓘ Review and finalize

Optional

→

Create

Cancel

Review and finalize

Backend

Name: **nginx-lb** Region: **us-central1** Session affinity: **None** Health check: **http-healthcheck**

Instances ^

mgr-1

mgr-2

mgr-3

w-1

w-2

Frontend

Protocol ^

IP:Port

TCP

Ephemeral:80

Click on **Create** button to provision your Load Balancer. Give it some time.

Once it is created, you can inspect it by clicking on the Load Balancer name. The details are shown below:

## ✓ nginx-lb

### Frontend

Protocol ^	IP:Port
TCP	104.197.188.61:80

### Backend

Name: **nginx-lb** Region: **us-central1** Session affinity: **None** Health check: **http-healthcheck**

Instances ^	104.197.188.61
mgr-1	✓
mgr-2	✓
mgr-3	✓
w-1	✓
w-2	✓

You can see that the Frontend part of it has been assigned an IP Address, which has been highlighted above. We can now hit this IP Address on port 80 and it will divert the traffic to be served by any of the healthy instances. All these instances are nothing but our nodes and since our NGINX service is running on these nodes on port 80, any of them will be able to serve it.

Go ahead, launch the browser and visit the Load Balancer IP in the browser. You should be fine:



Additionally, if you go to the list of Compute Engine instances, you will see that the nodes are now in use by our Load Balancer as shown below:

<input type="checkbox"/> Name ^	Zone	Machine type	Recommendation	In use by	Internal IP	External IP	
<input type="checkbox"/> mgr-1	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.2	130.211.199.228	
<input type="checkbox"/> mgr-2	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.3	104.154.244.185	
<input type="checkbox"/> mgr-3	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.4	104.154.56.35	
<input type="checkbox"/> w-1	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.5	107.178.213.86	
<input type="checkbox"/> w-2	us-central1-a	1 vCPU, 3.75 GB		nginx-lb	10.240.0.6	8.34.214.144	

We are looking good for now. Let us do a little deep dive into our overlay network and see what is going on.

## Understand the overlay network

Let us go back to the list of networks that we have. We can do that from any node. I suggest that you SSH first to mgr-1 and do the following:

```
$ sudo docker network list
```

NETWORK ID	NAME	DRIVER	SCOPE
------------	------	--------	-------

d4b360ee71b4	bridge	bridge	local
9a55643d34e8	docker_gwbridge	bridge	local
b6348ecb0afa	host	host	local
dzzo90eqcmt2	ingress	overlay	swarm
46f4630544d0	none	null	local
7ffh8lexsm9f	nw1	overlay	swarm

Notice again that created an overlay network **nw1** and its Network ID is **7ffh8lexsm9f**.

Let me also show you the output from the node listing in our cluster to understand where our 6 containers are currently running:

```
romin_irani@mgr-1:~$ sudo docker service ps nginx
ID        NAME      IMAGE      NODE      DESIRED STATE   CURRENT STATE      32 seconds ago
9z*       nginx.1   nginx      mgr-2     Running          Running            32 seconds ago
6e*       nginx.2   nginx      w-1       Running          Running            32 seconds ago
1o*       nginx.3   nginx      w-1       Running          Running            32 seconds ago
6n*       nginx.4   nginx      mgr-1     Running          Running            32 seconds ago
8l*       nginx.5   nginx      w-2       Running          Running            32 seconds ago
8p*       nginx.6   nginx      mgr-3     Running          Running            32 seconds ago
```

You can notice that there is only one container running on node **mgr-1** and that it's the **nginx.4 container**.

On the manager (**mgr-1**) node, you can now inspect the network by giving the following command:

```
romin_irani@mgr-1:~$ sudo docker network inspect nw1
[
  {
    "Name": "nw1",
    "Id": "7ffh8lexsm9fhiukslkyiml02",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "10.0.0.0/24",
          "Gateway": "10.0.0.1"
        }
      ]
    },
    "Internal": false,
    "Containers": {
      "3d32b7128776a8398e741ad542427f0edee0f165c3d8de66fd9b3777f6194a24":
```

```
{
    "Name": "nginx.4.6nlj4cg74wxx76r6cumwrnfto",
    "EndpointID":
"2a99474d5e67433e7b0a371fb56225ec8d330a0af4fbe9d573b7c10e890292b3",
    "MacAddress": "02:42:0a:00:00:03",
    "IPv4Address": "10.0.0.3/24",
    "IPv6Address": ""
  },
  "Options": {
    "com.docker.network.driver.overlay.vxlanid_list": "257"
  },
  "Labels": {}
}
]
```

First notice the Network ID that is the same value as what we have seen in the network listing and that it is a swarm overlay network.

Now, notice the interesting part in the section for **Containers**. You will find that it is running one container, which is what we expect and you will see the same name i.e. nginx.4. What this means is that 1 container is bound to that overlay network.

Now, go to w-1 or any other worker node i.e. SSH into it. In our output, we have 2 containers (**nginx.2** and **nginx.3**) running on that node, as highlighted below:

```
romin_irani@mgr-1:~$ sudo docker service ps nginx
ID        NAME        IMAGE        NODE    DESIRED STATE    CURRENT STATE    32 seconds ago
9z*       nginx.1     nginx        mgr-2   Running           Running 32 seconds ago
6e*       nginx.2    nginx       w-1    Running         Running 32 seconds ago
1o*       nginx.3    nginx       w-1    Running         Running 32 seconds ago
6n*       nginx.4     nginx        mgr-1   Running           Running 32 seconds ago
8l*       nginx.5     nginx        w-2     Running           Running 32 seconds ago
8p*       nginx.6     nginx        mgr-3   Running           Running 32 seconds ago
```

On **w-1** node, if we inspect our overlay network, we will get the following output:

```
romin_irani@w-1:~$ sudo docker network inspect nw1
[
  {
    "Name": "nw1",
    "Id": "7ffh8lexsm9fhiukslkyiml02",
    "Scope": "swarm",
    "Driver": "overlay",
    "EnableIPv6": false,
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
```



```

        {
            "Subnet": "10.0.0.0/24",
            "Gateway": "10.0.0.1"
        }
    ],
    "Internal": false,
    "Containers": {
        "89c37d465f8e3c064a796f24fd4fa82326e7b2fd0b364e64e1f8d2eddf23c84b":
    {
        "Name": "nginx.3.1oicz8rkfcggh978b76t5ijg",
        "EndpointID":
"2433172095f9090b9502308be3069ffae2d1664a062ca85fe922f21b29dfe93a",
        "MacAddress": "02:42:0a:00:00:08",
        "IPv4Address": "10.0.0.8/24",
        "IPv6Address": ""
    },
        "b3af0472d70e5c7210025ccbccb4432cb3c80e0ce2f907131ef4db6efb7c9d3e":
    {
        "Name": "nginx.2.6ezc0zmi6jbb3nmtjt0vpwpvt",
        "EndpointID":
"9b256f06c7f4d1b68f5592c7934a0f9770e7187dd49d5da6b033ea879b539cb0",
        "MacAddress": "02:42:0a:00:00:07",
        "IPv4Address": "10.0.0.7/24",
        "IPv6Address": ""
    }
    },
    "Options": {
        "com.docker.network.driver.overlay.vxlanid_list": "257"
    },
    "Labels": {}
}
]

```

In the container list, you will find that 2 containers (nginx.2 and nginx.3) correctly bound to the network **nw1**.

***Note: You should SSH on other nodes too and inspect the network from there too!***

Now that we have inspected the network, let us understand that in an overlay network, we have a Virtual IP Address and a DNS name for each service by default. So in essence, when someone hits our service via the service name, it will resolve to a Virtual IP Address.

To understand that, we can inspect the service from the manager node. In the SSH session for mgr-1, inspect the nginx service as shown below:

```

romin_irani@mgr-1:~$ sudo docker service inspect nginx
[
    {

```

```
"ID": "1e68lm0x00zsqzdpje7nwql3j",
"Version": {
  "Index": 54
},
"CreatedAt": "2016-09-25T09:57:13.328195699Z",
"UpdatedAt": "2016-09-25T09:57:13.336401804Z",
"Spec": {
  "Name": "nginx",
  "TaskTemplate": {
    "ContainerSpec": {
      "Image": "nginx"
    },
    "Resources": {
      "Limits": {},
      "Reservations": {}
    },
    "RestartPolicy": {
      "Condition": "any",
      "MaxAttempts": 0
    },
    "Placement": {}
  },
  "Mode": {
    "Replicated": {
      "Replicas": 6
    }
  },
  "UpdateConfig": {
    "Parallelism": 1,
    "FailureAction": "pause"
  },
  "Networks": [
    {
      "Target": "7ffh8lexsm9fhiukslkyiml02"
    }
  ],
  "EndpointSpec": {
    "Mode": "vip",
    "Ports": [
      {
        "Protocol": "tcp",
        "TargetPort": 80,
        "PublishedPort": 80
      }
    ]
  }
},
"Endpoint": {
  "Spec": {
    "UpdatedAt": "2016-09-25T09:57:13.336401804Z",
    "Mode": "vip",
    "Ports": [
      {
        "Protocol": "tcp",
        "TargetPort": 80,
        "PublishedPort": 80
      }
    ]
  }
}
```

```

        }
    ],
    "Ports": [
        {
            "Protocol": "tcp",
            "TargetPort": 80,
            "PublishedPort": 80
        }
    ],
    "VirtualIPs": [
        {
            "NetworkID": "dzzo90eqcmt2bvylygb59x0i3",
            "Addr": "10.255.0.8/16"
        },
        {
            "NetworkID": "7ffh8lexsm9fhiukslkyiml02",
            "Addr": "10.0.0.2/24"
        }
    ]
},
"UpdateStatus": {
    "StartedAt": "0001-01-01T00:00:00Z",
    "CompletedAt": "0001-01-01T00:00:00Z"
}
}
]

```

Scroll down to the VirtualIPs section and you will notice that for the overlay network nw1, whose Network Id is “7fff...”, the associated address is 10.0.0.2 as shown above. So in short, the service name nginx resolves to that Virtual IP address, which in return will then hit any of the nodes servicing that request via the internal load balancing providing by Swarm.

We will see this at the end of the blog post, when we create another service, go into that container instance and then are able to lookup the nginx service by name. But before that, this section should suffice to tell you how it is constructed behind the scenes and in case you need to debug, you know how to go about it , one by one.

Let us first look at a few other features, just to test them out, so that we better understand what is going on in Docker Swarm.

## Bring the Leader down

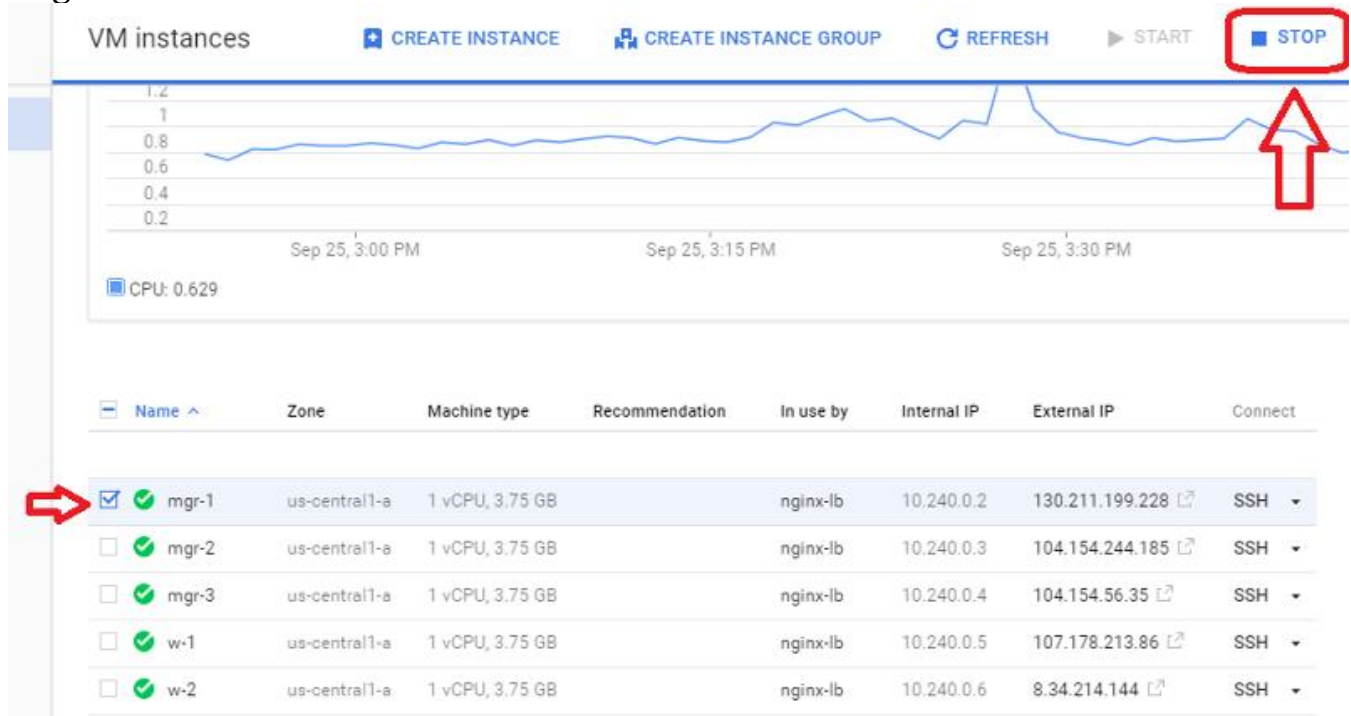
The first test that we will try is to bring out Leader down. Let me list out the current set of nodes and their Status in our Docker Swarm cluster.

```
romin_irani@mgr-1:~$ sudo docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
011..	mgr-2	Ready	Active	Reachable
4e3..	mgr-3	Ready	Active	Reachable
616.. *	mgr-1	Ready	Active	Leader
7xo..	w-1	Ready	Active	
8ie..	w-2	Ready	Active	

So, we have **mgr-1** as the Leader and we have two other managers, who are reachable. What we expect is that if we bring **mgr-1** down, then one of the other managers should take over as Leader. I suggest you also read up on [Raft Consensus protocol](#) to understand how the negotiation could take place and what would be some constraints on the number of managers and workers that you need have consensus from.

So, I am going to go and stop the current running instance on Google Compute Engine. You can do that from the Web console as shown below:



Wait till it has stopped. Now we can SSH into mgr-2 instance and see what is going on:

```
romin_irani@mgr-2:~$ sudo docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
01.. *	<b>mgr-2</b>	<b>Ready</b>	<b>Active</b>	<b>Leader</b>
4e..	mgr-3	Ready	Active	Reachable
61..	mgr-1	Down	Active	Unreachable

7x..	w-1	Ready	Active
8i..	w-2	Ready	Active

It is interesting to see that mgr-2 became the Leader. If you hit the Load Balancer IP, everything is still working fine.

Let us look at what happened to our existing container instances. Remember that we had mentioned that we want **6 replicas** of the **nginx** service. And if you recollect, one container was running on **mgr-1**.

```
romin_irani@mgr-2:~$ sudo docker service ps nginx
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE	STATE
9z..	nginx.1	nginx	mgr-2	Running	Running	29 min ago
6e..	nginx.2	nginx	w-1	Running	Running	29 min ago
1o..	nginx.3	nginx	w-1	Running	Running	29 min ago
1s..	nginx.4	nginx	mgr-2	Running	Running	3 min ago
6n..	\_ nginx.4	nginx	mgr-1	Shutdown	Running	29 min ago
8l..	nginx.5	nginx	w-2	Running	Running	29 min ago
8p..	nginx.6	nginx	mgr-3	Running	Running	29 min ago

You will notice that the container nginx.4 which was running on mgr-1 was taken down and relaunched on mgr-2. Looks good!

## Bringing the original Leader back up again

What happens if we bring mgr-1 back up again. Will it take over as the Leader again, since it was the original leader or will be be a Manager node but cannot become a leader just by coming up again.

It's straightforward to try this. Simply go to the Cloud console and restart the instance. Wait till the instance is powered on and running:

If you are still in the SSH session on mgr-2, you can try:

```
romin_irani@mgr-2:~$ sudo docker node ls
```

ID	HOSTNAME	STATUS	AVAILABILITY	MANAGER STATUS
01.. *	mgr-2	Ready	Active	Leader
4e..	mgr-3	Ready	Active	Reachable
6l..	mgr-1	Ready	Active	Reachable
7x..	w-1	Ready	Active	
8i..	w-2	Ready	Active	

You find that mgr-2 is still the Leader. mgr-1 is now Reachable but did not get instantly promoted to be a Leader.

What about our 6 containers, would some of them get relaunched on mgr-1, just because it came up. Let's see:

```
romin_irani@mgr-2:~$ sudo docker service ps nginx
ID      NAME      IMAGE    NODE    DESIRED STATE    CURRENT STATE    AGE
9z..    nginx.1    nginx    mgr-2    Running           Running          35 min ago
6e..    nginx.2    nginx    w-1     Running           Running          35 min ago
1o..    nginx.3    nginx    w-1     Running           Running          35 min ago
1s..    nginx.4    nginx    mgr-2    Running           Running          9 min ago
6n..    \_nginx.4  nginx    mgr-1    Shutdown          Running          35 min ago
8l..    nginx.5    nginx    w-2     Running           Running          35 min ago
8p..    nginx.6    nginx    mgr-3    Running           Running          35 min ago
```

Well, it did not! Docker Swarm does not assign containers to newly joined nodes unless the service is scaled or some other nodes are drained and so on. On how to scale, you can follow the [Docker Swarm Tutorial](#) that I earlier wrote.

*Note: You can try scaling the service up by a few more replicas and see what happens. Try it as an exercise.*

*Hint: **\$ docker service scale nginx=8***

## Bringing a Node down and backup

This should be straightforward to predict and try out. I will leave it as an exercise for the reader. Just stop **w-1** node and then check on the status of the nodes in the swarm and also how it relauches containers on the other remaining RUNNING nodes.

*Do keep in mind that as we saw earlier, bringing up the node, does not mean that it will immediately get assigned some containers.*

## Creating another Service

It is time now to see how the overlay network is working. To reiterate, the overlay network allows containers across multiple hosts to communicate to each other. What this means is that you should be able to simply access any service by its name in any of the containers on the same overlay network.

By referring to the service by its name, it also allows us to scale the number of containers up and down, make them join the swarm and still keep accessing them via a uniform service name.

I am going to use the example from the Docker Swarm overlay network service documentation and use it over here.

First up, we will create a new Docker Swarm service. And then from the containers running this new service, we will see that we can access the service by name.

SSH in mgr-1 or mgr-2 instance. And create the new service as shown below. Note that we are going to use the **same overlay network nw1**.

```
romin_irani@mgr-1:~$ sudo docker service create --name my-busybox --network nw1
busybox sleep 3000
azeevpytjcwsfoyvuv2pj4vdu
romin_irani@mgr-1:~$ sudo docker service ls
```

ID	NAME	REPLICAS	IMAGE	COMMAND
1e68lm0x00zs	nginx	6/6	nginx	azeevpytjcws my-busybox 1/1
busybox	sleep 3000			

```
romin_irani@mgr-1:~$ sudo docker service ps my-busybox
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE
4elkoyujbyausx7zi5u5i7999	my-busybox.1	busybox	mgr-1	Running	Running

21 seconds ago

You will notice that in the first command, we are starting up a **busybox** service named **my-busybox**, we want only one replica of it, we are using the same overlay network **nw1**. Notice that we gave a delay of **3000s** so that the container is alive for a while before shutting down because on its own the busybox container will just exit otherwise.

Great! The next command that you see above is the standard service listing and you can see that it has 2 services now: **nginx** and **my-busybox** service.

Similarly, the last command is to find out where the **my-busybox** service containers are running. We find that it is running on **mgr-1**.

Now, let us get into the Bash shell for the running container for the my-busybox service.

```
romin_irani@mgr-1:~$ sudo docker service ps my-busybox
```

ID	NAME	IMAGE	NODE	DESIRED STATE	CURRENT STATE
4elkoyujbyausx7zi5u5i7999	my-busybox.1	busybox	mgr-1	Running	Running

21 seconds ago

Notice that we have the following attributes:

- NAME is my-busybox.1
- ID is 4elkoyujbyausx7zi5u5i7999

In summary, you can form a unique name for the Container as NAME.ID

To go into the bash shell for this container, execute the following command:

```
romin_irani@mgr-1:~$ sudo docker exec -it my-busybox.1.4elkoyujbyausx7zi5u5i7999
/bin/sh
/ #
/ #
/ #
```

Now inside this shell, we can do a lookup for our service **nginx** by name.

```
/ #
/ #
/ # nslookup nginx
Server: 127.0.0.11
Address 1: 127.0.0.11
Name: nginx
Address 1: 10.0.0.2
/ #
/ #
```

You can even do a wget inside over here to validate that we are able to hit the service and get the NGINX default home page:

```
/ # wget -O - nginx
Connecting to nginx (10.0.0.2:80)
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
```



```
</body>
</html>
- 100%
| ***** | 612
0:00:00 ETA
/ #
```

This shows that we now have a network where each of the services are available to all containers on the network. This way you can link up multiple containers as needed.

Due to the fact that we have a service abstraction now, you can scale your nodes—add / remove them—and not affect the containers that are accessing it by service name. They will not be worried about where the containers are running i.e. on which nodes.

## Conclusion

I like the simplicity of Docker Swarm and conducting these experiments gave me a good sense of understanding how it is working behind the scenes, what to expect and most importantly, to actually see it work.

Please let me know in the comments if you have any feedback.

*Download a mini book (about 50 pages) that contains both the Docker Swarm Tutorial and Docker Swarm on Google Compute Engine. Click [here](#) to download the PDF.*

## Part 12 — Getting Started with Kubernetes using Minikube

*If you are looking for running Kubernetes on your Mac, go to [this tutorial](#).*

I have recently covered multiple posts ([1](#) & [2](#)) on getting started with Docker Swarm.

I personally like the simplicity of Docker Swarm and have found in my teaching experience with developers, that it was easier for most people to understand what Container Management solutions are all about when they see a few simple commands in Docker Swarm and are able to relate to stuff like scaling up/down, rolling updates, etc.



Personally I think if you are looking for a container management solution in today's world, you have to invest your time in [Kubernetes](#) (k8s). There is no doubt about that because of multiple factors. To the best of my understanding, these points include:

- Kubernetes is [Open Source](#)
- Great momentum in terms of activities & contribution at its [Open Source Project](#)
- Decades of experience running its [predecessor](#) at Google
- Support of multiple OS and infrastructure software vendors
- Rate at which features are being released
- Production readiness (Damn it, [Pokemon Go](#) met its scale due to Kubernetes)
- Number of features available. Check out the list of features at the [home page](#).

## What this post is about?

I do not want to spend time explaining about what Kubernetes is and its building blocks like Pods, Replication Controllers, Services, Deployments and more. There are multiple articles on that and I suggest that you go through it.

I have written a couple of other articles that go through a high level overview of Kubernetes:

- [Introduction to Kubernetes](#)
- [Kubernetes Building Blocks](#)

It is important that you go through some basic material on its concepts, so that we can directly get down into its commands.

The general perception about a management solution like Kubernetes is that it would require quite a bit of setup for you to try it out locally. What this means is that it would take some time to set it up but more than setting it up, you might probably get access to it only during staging phase or something like that. Ideally you want a similar environment in your development too, so that you are as close to what it takes to run your application. The implications of this is that you want it running on your laptop/desktop, where you are likely to do your development.

This was the goal behind the [minikube](#) project and the team has put in fantastic effort to help us setup and run Kubernetes on our development machines. This is as simple and portable as it can get. The tagline of minikube project says it all: **“Run Kubernetes locally”**.



# minikube

*Side Note: The [design of the minikube logo](#) makes for interesting reading.*

This post is going to take you through setting up Minikube on your Windows development machine and then taking it for a Hello World spin to see a local Kubernetes cluster in action. Along the way, I will highlight my environment and what I had to do to get the experimental build of minikube working on my Windows machine. Yes, it is experimental software, but it works!

If you are not on Windows, the instructions to setup minikube on either your Linux machine or Mac machine are also available [here](#). Check it out. You can then safely skip over the setup and go to the section where we do a quick Hello World to test drive Kubernetes locally.

Keep in mind that Minikube gives you a single node cluster that is running in a VM on your development machine.

*Of course, once you are done with what you see in this blog, I strongly recommend that you also look at Managed Container Orchestration solutions like Google Container Engine.*

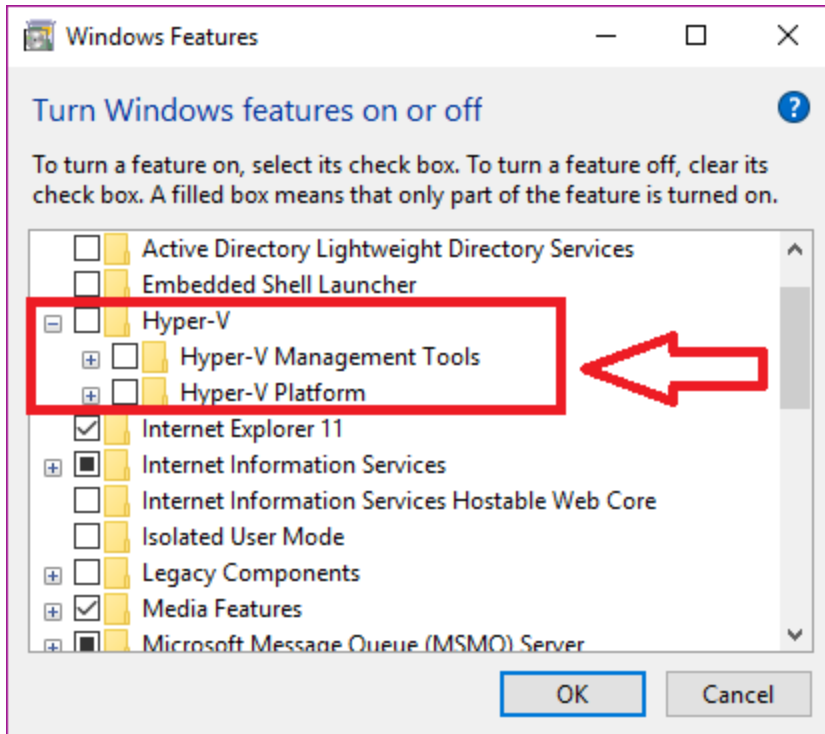
Let's get started now with installation of minikube. But first, we must make sure that our development machine has some of the pre-requisites required to run it. Do not ignore that!

## Using VirtualBox and not Hyper-V

VirtualBox and Hyperv (which is available on Windows 10) do not make a happy pair and you are bound to run into situations where the tools get confused. I preferred to use VirtualBox and avoid all esoteric command-line switches that we need to provide to enable creation of the underlying Docker hosts, etc.

To disable Hyper-V, go to Turn Windows features on or off and you will see a dialog with list of Windows features as shown below. Navigate to the Hyper-V section and disable it completely.

***This will require a restart to the machine to take effect and on my machine, it even ended up doing a Windows Update, configuring it and a good 10 minutes later, it was back up.***



Great! We have everything now to get going.

## Development Machine Environment

I am assuming that you have a setup that is similar to this. I believe, you should be fine on Windows 7 too and it would not have the HyperV stuff, instructions of which I will give in a while.

- Windows 10 Laptop. VT-x/AMD-v virtualization must be enabled in BIOS.
- [Docker Toolbox v1.12.0](#). The toolbox sets up VirtualBox and I have gone with that.
- **kubectl** command line utility. This is the CLI utility for the Kubernetes cluster and you need to install it and have it available in your PATH. To install the latest 1.4 release, do the following: Go to the browser and give the following URL : <http://storage.googleapis.com/kubernetes-release/release/v1.4.0/bin/windows/amd64/kubectl.exe>. This will download the kubectl CLI executable. Please make it available in the environment PATH variable.

*Note: kubectl versions are available at a generic location as per the following*

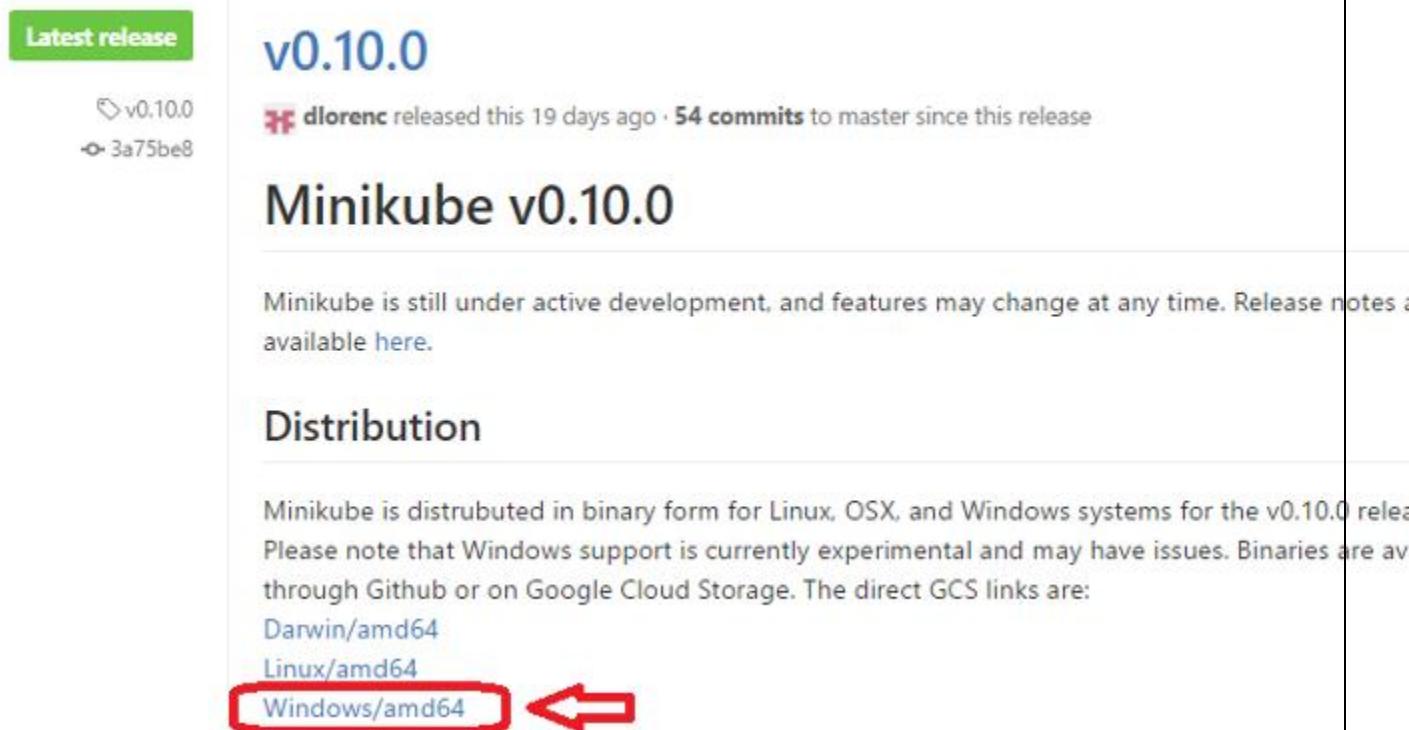
*format: [http://storage.googleapis.com/kubernetes-release/release/\\${K8S\\_VERSION}/bin/\\${GOOS}/\\${GOARCH}/\\${K8S\\_BINARY}](http://storage.googleapis.com/kubernetes-release/release/${K8S_VERSION}/bin/${GOOS}/${GOARCH}/${K8S_BINARY})*

# Minikube installation

The first step is to take the **kubectl.exe** file that you downloaded in the previous step and place that in the **C:\** folder.

The next step is to download the minikube binary from the following location: <https://github.com/kubernetes/minikube/releases>

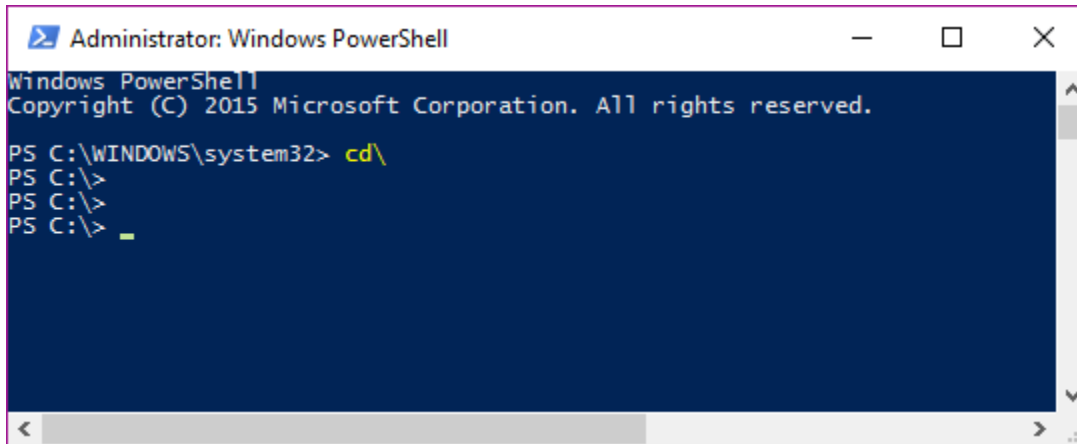
Go to the Windows download link as shown below:



This will start downloading the **v0.10.0** release of the executable. The file name is **minikube-windows-amd64.exe**. Just rename this to **minikube.exe** and place it in **C:\** drive, alongside the **kubectl.exe** file from the previous section.

You are all set now to launch a local Kubernetes one-node cluster!

**All the steps moving forward are being done in Powershell. Launch Powershell in Administrative mode (Ctrl-Shift-Enter) and navigate to C:\ drive where the kubectl.exe and minikube.exe files are present.**

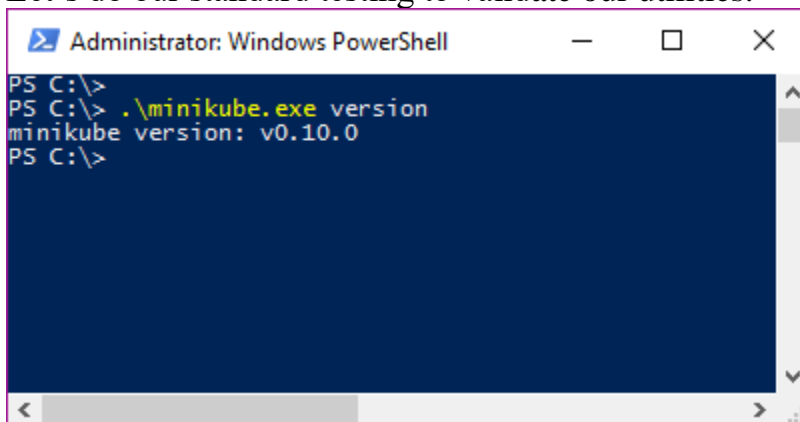


```
Administrator: Windows PowerShell
Windows PowerShell
Copyright (C) 2015 Microsoft Corporation. All rights reserved.

PS C:\WINDOWS\system32> cd\
PS C:\>
PS C:\>
PS C:\>
```

## A few things to note

Let's do our standard testing to validate our utilities.



```
Administrator: Windows PowerShell
PS C:\>
PS C:\> .\minikube.exe version
minikube version: v0.10.0
PS C:\>
```

If you go to your %HOMEPATH%\minikube folder now, you will notice that several folders got created. Take a look!

There are multiple commands that Minikube supports. You can use the standard `--help` option to see the list of commands that it has:

```
PS C:\> .\minikube --help
```

Minikube is a CLI tool that provisions and manages single-node Kubernetes clusters optimized for development workflows

Usage:

minikube [command]

Available Commands:

dashboard Opens/displays the kubernetes dashboard URL for your local cluster

delete Deletes a local kubernetes cluster.

docker-env sets up docker env variables; similar to '\$(docker-machine env)'

get-k8s-versions Gets the list of available kubernetes versions available for minikube.

ip Retrieve the IP address of the running cluster.

logs	Gets the logs of the running localkube instance, used for debugging minikube, not user code.
config	Modify minikube config
service	Gets the kubernetes URL for the specified service in your local cluster
ssh	Log into or run a command on a machine with SSH; similar to 'docker-machine ssh'
start	Starts a local kubernetes cluster.
status	Gets the status of a local kubernetes cluster.
stop	Stops a running local kubernetes cluster.
version	Print the version of minikube.

#### Flags:

- alsologtostderr[=false]: log to standard error as well as files
- log-flush-frequency=5s: Maximum number of seconds between log flushes
- log\_backtrace\_at=:0: when logging hits line file:N, emit a stack trace
- log\_dir="": If non-empty, write log files in this directory
- logtostderr[=false]: log to standard error instead of files
- show-libmachine-logs[=false]: Whether or not to show logs from libmachine.
- stderrthreshold=2: logs at or above this threshold go to stderr
- v=0: log level for V logs
- vmodule=: comma-separated list of pattern=N settings for file-filtered logging

Use "minikube [command] --help" for more information about a command.

I have highlighted a couple of **Global flags** that you can use in all the commands for minikube. These flags are useful to see what is going on inside the hood at times and also for seeing the output on the standard output (console/command).

Minikube supports multiple versions of Kubernetes and the latest version is v1.4.0. To check out the different versions supported try out the following command:

**PS C:\> .\minikube get-k8s-versions**

The following Kubernetes versions are available:

- v1.4.0
- v1.3.7
- v1.3.6
- v1.3.5
- v1.3.4
- v1.3.3
- v1.3.0

## Starting our Cluster

We are now ready to launch our Kubernetes cluster locally. We will use the **start** command for it.

*Note: You might run into multiple issues while starting a cluster the first time. I have several of them and have created a section at the end of this blog post on Troubleshooting. Take a look at it, in case you run into any issues.*



You can check out the help and description of the command/flags/options via the help option as shown below:

```
PS C:\> .\minikube.exe start --help
```

You will notice several **Flags** that you can provide to the **start** command and while there are some useful defaults, we are going to be a bit specific, so that we can better understand things.

We want to use Kubernetes v1.4.0 and while the VirtualBox driver is default on windows, we are going to be explicit about it. At the same time, we are going to use a couple of the Global Flags that we highlighted earlier, so that we can see what is going on under the hood.

All we need to do is give the following command (I have separated the flags on separate line for better readability). The output is also attached.

```
PS C:\> .\minikube.exe start --kubernetes-version="v1.4.0"  
--vm-driver="virtualbox"  
--show-libmachine-logs --alsologtostderr
```

```
W1004 13:01:30.429310 9296 root.go:127] Error reading config file at C:\Users\irani_r\.minikube\config\config.json:  
o
```

```
pen C:\Users\irani_r\.minikube\config\config.json: The system cannot find the file specified.
```

```
I1004 13:01:30.460582 9296 notify.go:103] Checking for updates...
```

#### **Starting local Kubernetes cluster...**

```
Creating CA: C:\Users\irani_r\.minikube\certs\ca.pem
```

```
Creating client certificate: C:\Users\irani_r\.minikube\certs\cert.pemRunning pre-create checks...
```

#### **Creating machine...**

```
(minikube) Downloading C:\Users\irani_r\.minikube\cache\boot2docker.iso from  
file://C:/Users/irani_r/.minikube/cache/iso  
/minikube-0.7.iso...
```

```
(minikube) Creating VirtualBox VM...
```

```
(minikube) Creating SSH key...
```

```
(minikube) Starting the VM...
```

```
(minikube) Check network to re-create if needed...
```

```
(minikube) Waiting for an IP...
```

```
Waiting for machine to be running, this may take a few minutes...
```

```
Detecting operating system of created instance...
```

```
Waiting for SSH to be available...
```

```
Detecting the provisioner...
```

```
Provisioning with boot2docker...
```

```
Copying certs to the local machine directory...
```

```
Copying certs to the remote machine...
```

```
Setting Docker configuration on the remote daemon...
```

```
Checking connection to Docker...
```

#### **Docker is up and running!**

```
I1004 13:03:06.480550 9296 cluster.go:389] Setting up certificates for IP: %s 192.168.99.100
```

```
I1004 13:03:06.567686 9296 cluster.go:202] sudo killall localkube || true
I1004 13:03:06.611680 9296 cluster.go:204] killall: localkube: no process killed
I1004 13:03:06.611680 9296 cluster.go:202]
# Run with nohup so it stays up. Redirect logs to useful places.
sudo sh -c 'PATH=/usr/local/sbin:$PATH nohup /usr/local/bin/localkube --generate-certs=false --logtostderr=true --
node
-ip=192.168.99.100 > /var/lib/localkube/localkube.err 2> /var/lib/localkube/localkube.out < /dev/null & echo $! > /var/r
un/localkube.pid &'
I1004 13:03:06.658605 9296 cluster.go:204]
Kubectl is now configured to use the cluster.
PS C:\>
```

Let us understand what it is doing behind the scenes in brief. I have also highlighted some of the key lines in the output above:

1. It generates the certificates and then proceeds to provision a local Docker host. This will result in a VM created inside of VirtualBox.
2. That host is provisioned with the boot2Docker ISO image.
3. It does its magic of setting it up, assigning it an IP and all the works.
4. Finally, it prints out a message that kubectl is configured to talk to your local Kubernetes cluster.

You can now check on the status of the local cluster via the **status** command:

```
PS C:\> .\minikube.exe status
minikubeVM: Running
localkube: Running
```

You can also use the kubectl CLI to get the cluster information:

```
PS C:\> .\kubectl.exe cluster-info
Kubernetes master is running at https://192.168.99.100:8443
kubernetes-dashboard is running at https://192.168.99.100:8443/api/v1/proxy/namespaces/kube-
ashboard
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

# Kubernetes Client and Server version

Let us do a quick check of the Kubernetes version at the client and server level. Execute the following command:

```
PS C:\> .\kubectl version
Client Version: version.Info{Major:"1", Minor:"4", GitVersion:"v1.4.0",
GitCommit:"a16c0a7f71a6f93c7e0f222d961f4675cd97a
46b", GitTreeState:"clean", BuildDate:"2016-09-26T18:16:57Z", GoVersion:"go1.6.3", Compiler:"gc",
Platform:"windows/amd6
4"}
Server Version: version.Info{Major:"1", Minor:"4", GitVersion:"v1.4.0",
GitCommit:"a16c0a7f71a6f93c7e0f222d961f4675cd97a
46b", GitTreeState:"dirty", BuildDate:"1970-01-01T00:00:00Z", GoVersion:"go1.7.1", Compiler:"gc",
Platform:"linux/amd64"
}
```

You will notice that both client and server are at version **1.4**.

## Cluster IP Address

You can get the IP address of the cluster via the **ip** command:

```
PS C:\> .\minikube.exe ip
192.168.99.100
```

## Kubernetes Dashboard

You can launch the Kubernetes Dashboard at any point via the **dashboard** command as shown below:

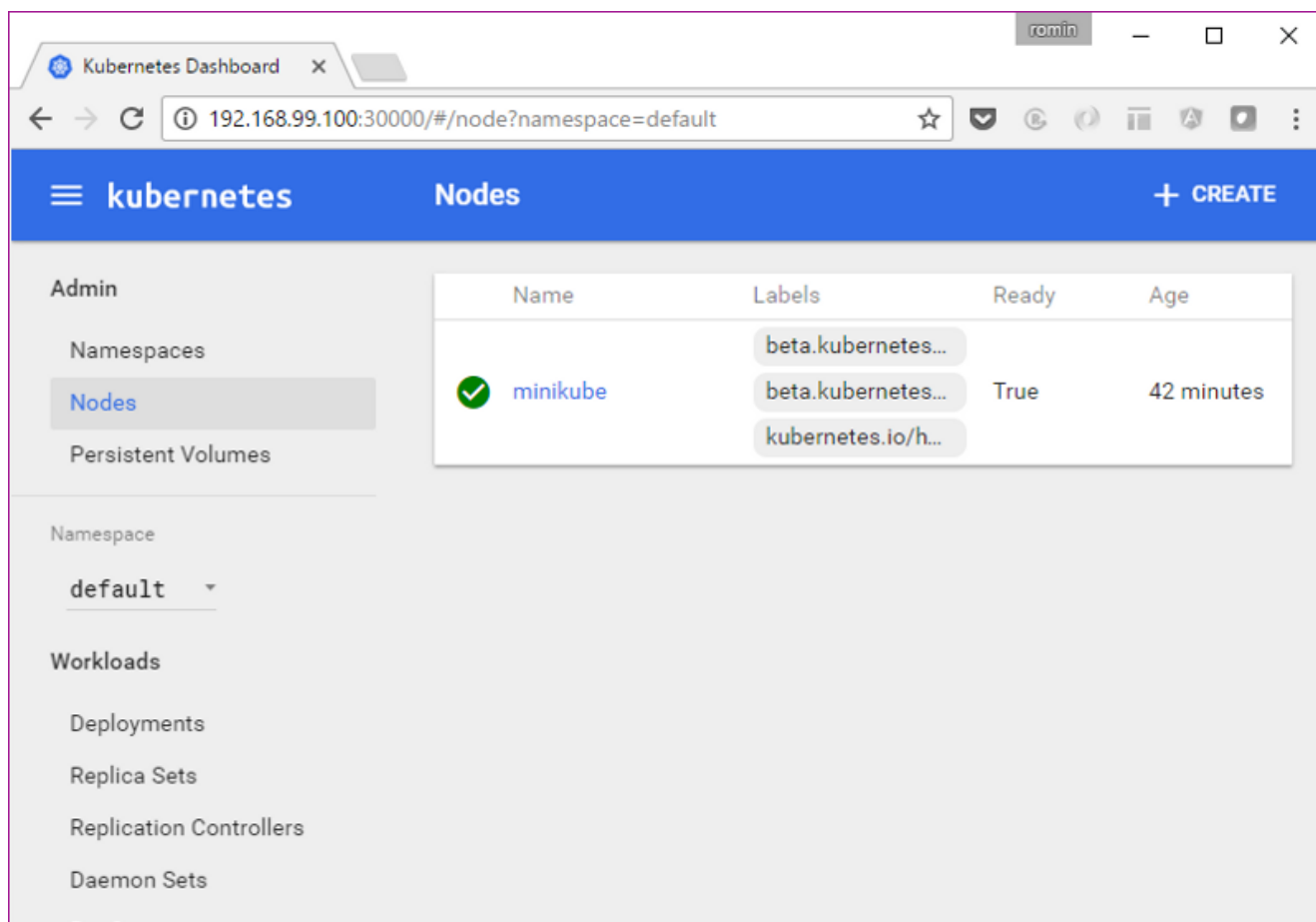
```
PS C:\> .\minikube.exe dashboard
```

This will automatically launch the Dashboard in your local browser. However if you just want to nab the Dashboard URL, you can use the following flag:

```
PS C:\> .\minikube.exe dashboard --url=true
http://192.168.99.100:30000
```

There is a [great post](#) on how the Kubernetes Dashboard underwent a design change in version 1.4. It explains how the information is split up into respective sections i.e. Workloads , Services and Discovery, Storage and Configuration, which are present on the left-side menu and via which you can sequentially introspect more details of your cluster. All of this is provided by a nifty filter for the Namespace value above.

If you look at the Kubernetes dashboard right now, you will see that it indicates that nothing has been deployed. Let us step back and think what we have so far. We have launched a **single-node cluster** .. right? Click on the Node link and you will see that information:



The above node information can also be obtained by using the kubectl CLI to get the list of nodes.

```
PS C:\> .\kubectl.exe get nodes
NAME      STATUS    AGE
minikube  Ready     51m
```

Hopefully, you are now able to relate how some of the CLI calls are reflected in the Dashboard too. Let's move forward. But before that, one important tip!

## Tip: use-context minikube

If you had noticed closely when we started the cluster, there is a statement in the output that says “**Kubectl is now configured to use the cluster.**” What this is supposed to do is to eventually set the current context for the kubectl utility so that it knows which cluster it is talking to. Behind the scenes in your %HOMEPATH%\kube directory, there is a config file that contains information about your Kubernetes cluster and the details for connecting to your various clusters is present over there.

In short, we have to be sure that the kubectl is pointing to the right cluster. In our case, the cluster name is minikube.

In case you see an error like the one below (I got it a few times), then you need to probably set the context again.

```
PS C:\> kubectl get nodes
```

```
error: You must be logged in to the server (the server has asked for the client to provide credentials)
```

The command for that is:

```
PS C:\> kubectl config use-context minikube
```

```
switched to context "minikube".
```

## Running a Workload

Let us proceed now to running a simple [Nginx container](#) to see the whole thing in action:

We are going to use the run command as shown below:

```
PS C:\> .\kubectl.exe run hello-nginx --image=nginx --port=80
```

```
deployment "hello-nginx" created
```

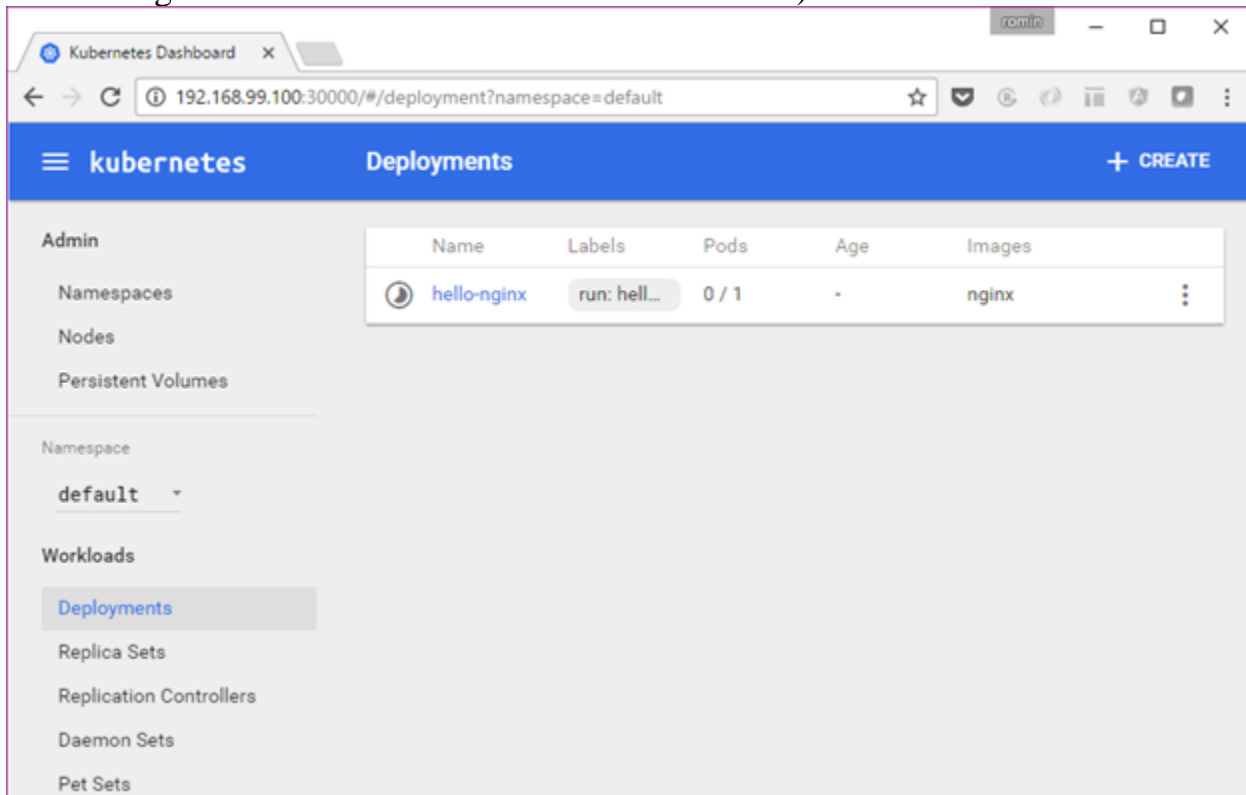
This creates a deployment and we can investigate into the Pod that gets created, which will run the container:

```
PS C:\> .\kubectl.exe get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-nginx-24710...	0/1	ContainerCreating	0	2m

You can see that the STATUS column value is **ContainerCreating**.

Now, let us go back to the Dashboard (I am assuming that you either have it running or can launch it again via the **minikube dashboard** command):



You can notice that if we go to the Deployments option, the Deployment is listed and the status is still in progress. You can also notice that the Pods value is 0/1.

If we wait for a while, the Pod will eventually get created and it will be ready as the command below shows:

```
PS C:\> .\kubectl.exe get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hello-nginx-24710...	1/1	Running	0	3m

If we see the Dashboard again, the Deployment is ready now:

Kubernetes Dashboard

192.168.99.100:30000/#/deployment?namespace=default

### kubernetes Deployments + CREATE

Admin

- Namespaces
- Nodes
- Persistent Volumes

Namespace

default

Workloads

- Deployments
- Replica Sets
- Replication Controllers
- Daemon Sets
- Pet Sets

Name	Labels	Pods	Age	Images
hello-nginx	run: hell...	1 / 1	57 seconds	nginx

If we visit the Replica Sets now, we can see it:

Kubernetes Dashboard

192.168.99.100:30000/#/replicaset?namespace=default

### kubernetes Replica Sets + CREATE

default

Workloads

- Deployments
- Replica Sets
- Replication Controllers
- Daemon Sets
- Pet Sets
- Jobs
- Pods

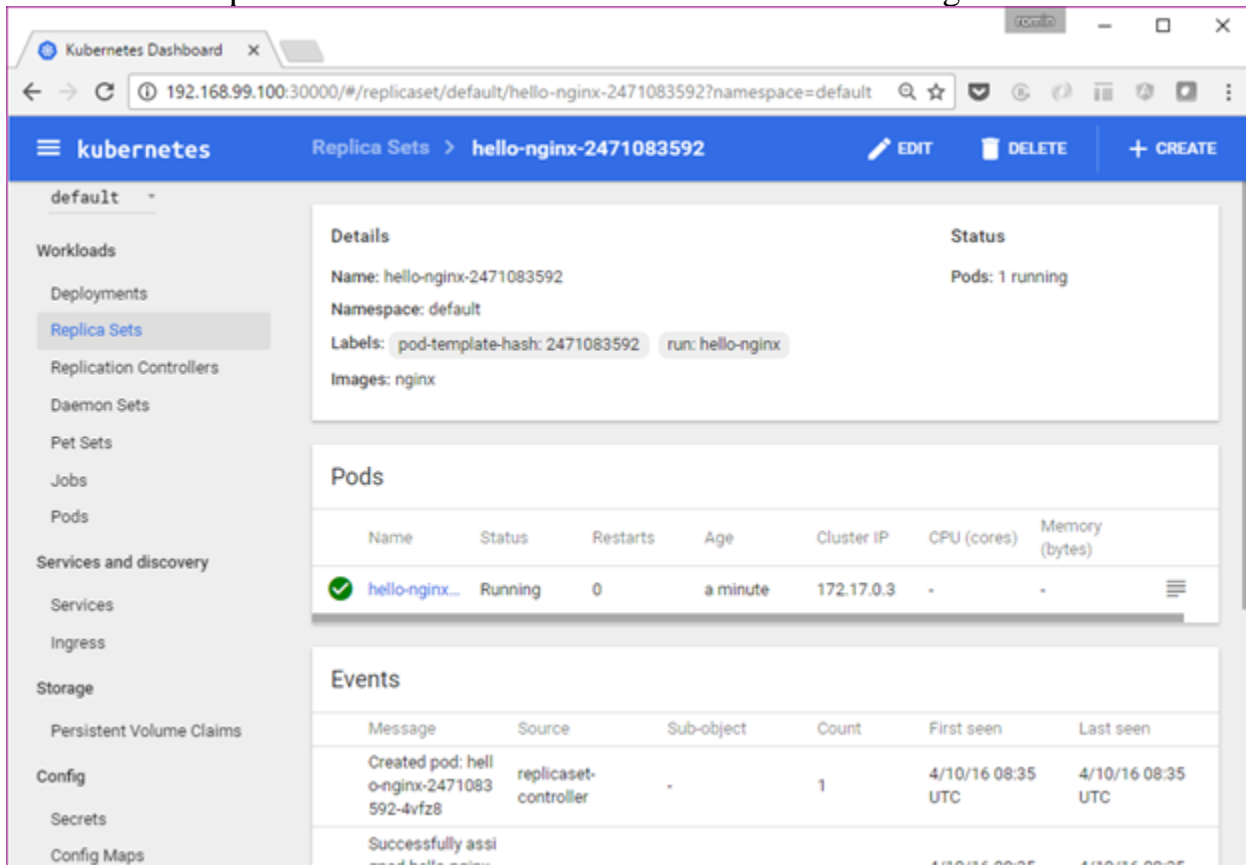
Services and discovery

- Services
- Ingress

Storage

Name	Labels	Pods	Age	Images
hello-nginx...	pod-tem... run: hell...	1 / 1	a minute	nginx

Click on the Replica Set name and it will show the Pod details as given below:



The screenshot shows the Kubernetes Dashboard interface. The left sidebar contains a navigation menu with categories: Workloads, Services and discovery, Storage, and Config. Under Workloads, 'Replica Sets' is selected. The main content area shows the details for the Replica Set 'hello-nginx-2471083592' in the 'default' namespace. The details include the Name, Namespace, Labels, and Images. Below the details, there is a 'Pods' section with a table showing one running pod. At the bottom, there is an 'Events' section with a table showing two events.

**Details**

Name: hello-nginx-2471083592  
Namespace: default  
Labels: pod-template-hash: 2471083592 run: hello-nginx  
Images: nginx

**Status**

Pods: 1 running

**Pods**

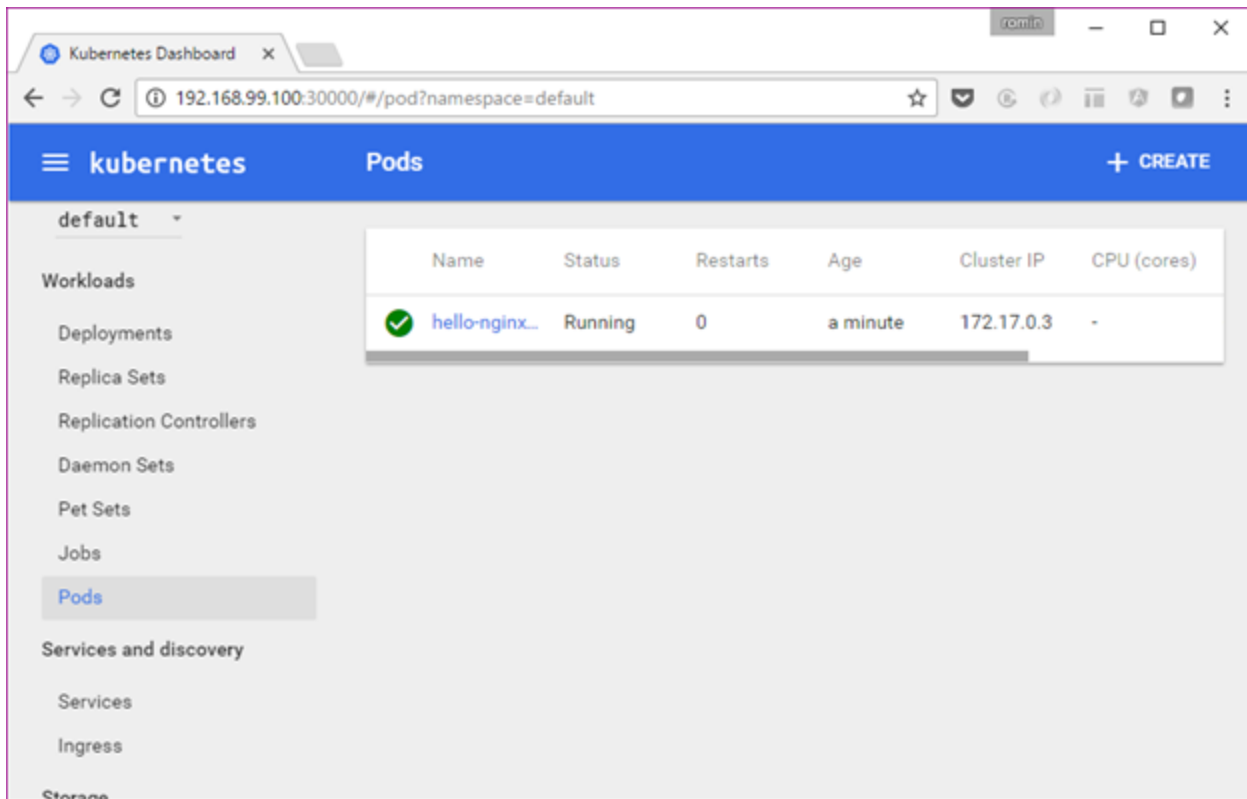
Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
hello-nginx...	Running	0	a minute	172.17.0.3	-	-

**Events**

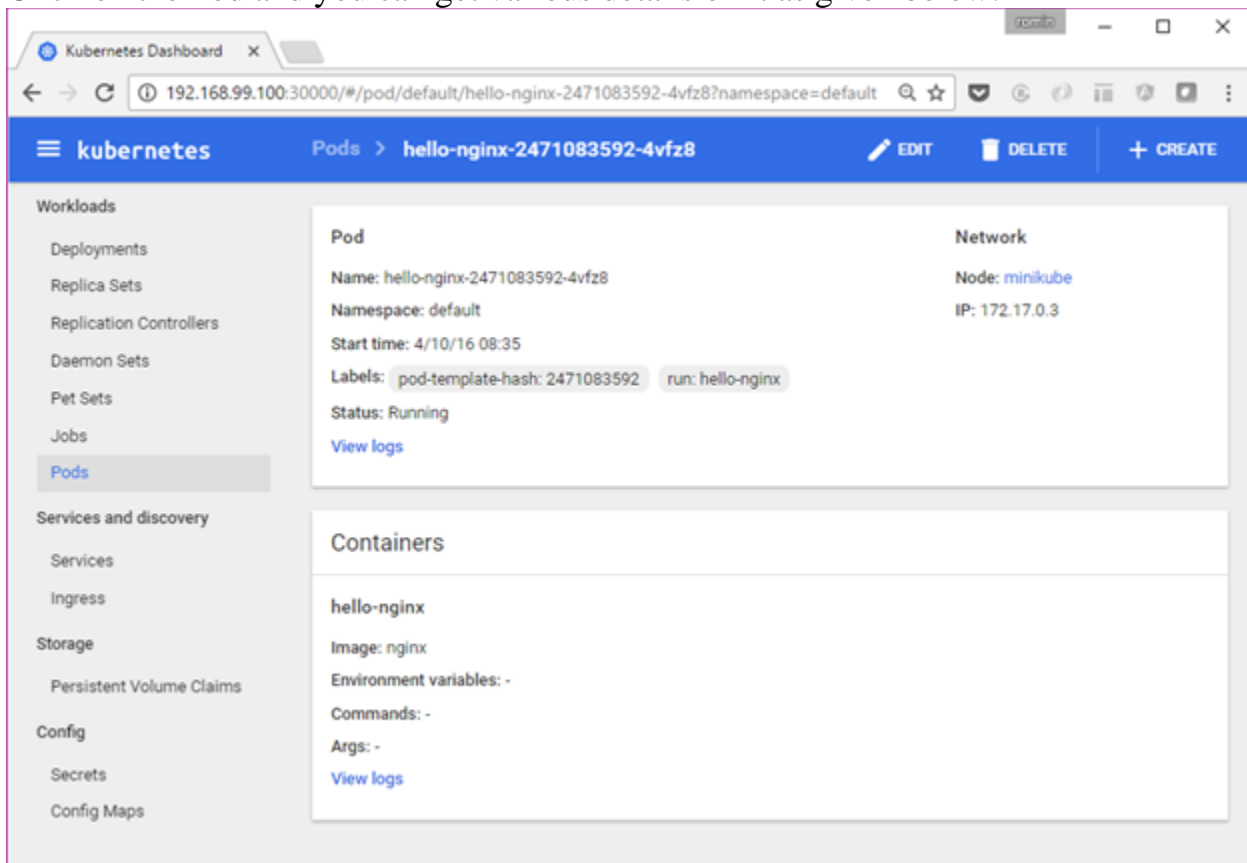
Message	Source	Sub-object	Count	First seen	Last seen
Created pod: hello-nginx-2471083592-4vzf8	replicaset-controller	-	1	4/10/16 08:35 UTC	4/10/16 08:35 UTC
Successfully assigned pod hello-nginx...				4/10/16 08:35	4/10/16 08:35

Alternately, you can also get to the Pods via the **Pods** link in the Workloads as shown below:





Click on the Pod and you can get various details on it as given below:



You can see that it has been given some default labels. You can see its IP address. It is part of the node named minikube. And most importantly, there is a link for **View Logs** too.

The 1.4 dashboard greatly simplifies using Kubernetes and explaining it to everyone. It helps to see what is going on in the Dashboard and then the various commands in kubectl will start making sense more.

We could have got the Node and Pod details via a variety of **kubectl describe node/pod** commands and we can still do that. An example of that is shown below:

```
PS C:\> .\kubectl.exe describe pod hello-nginx-2471083592-4vfz8
```

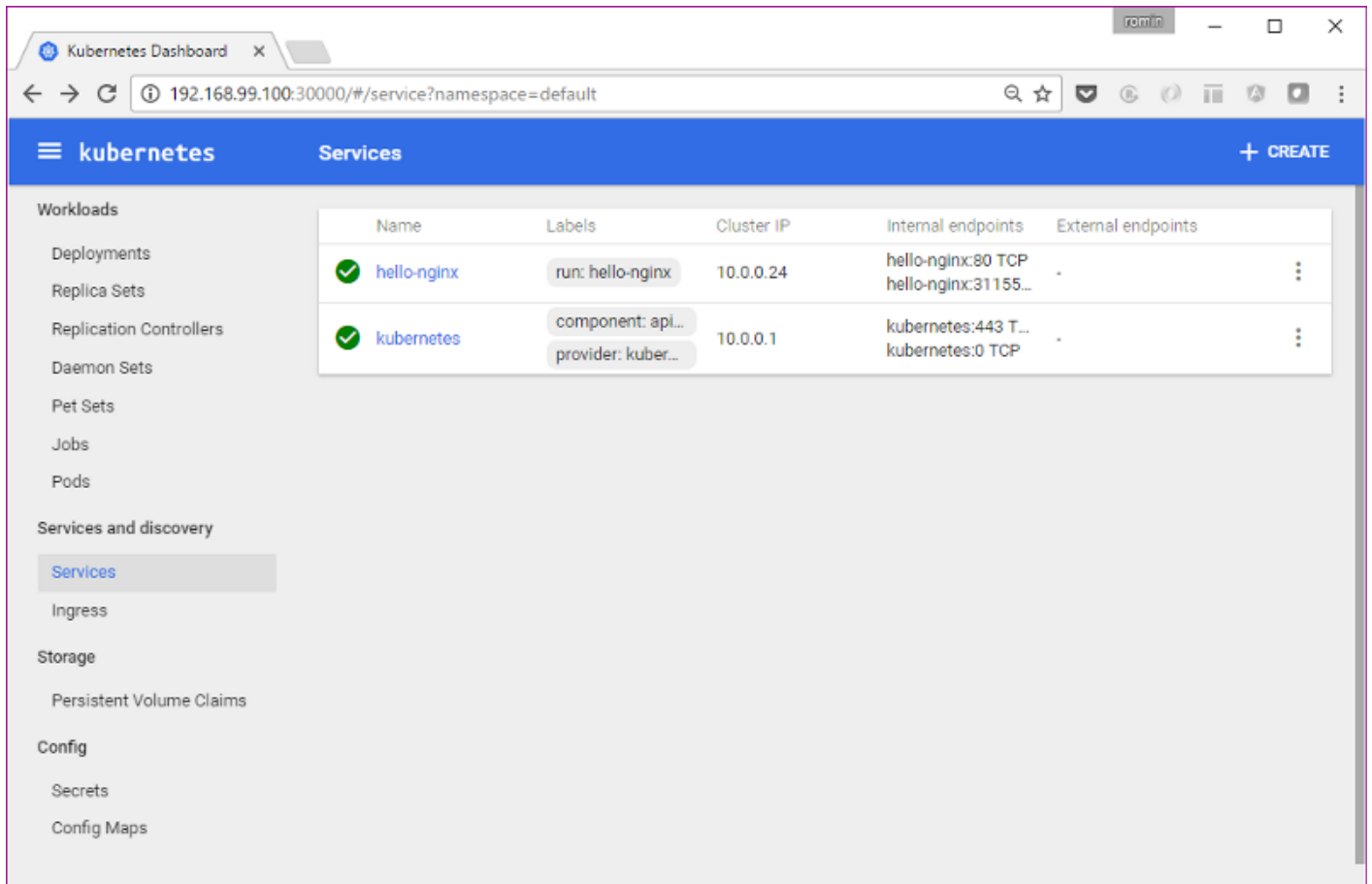
```
Name:      hello-nginx-2471083592-4vfz8
Namespace:  default
Node:       minikube/192.168.99.100
Start Time: Tue, 04 Oct 2016 14:05:15 +0530
Labels:     pod-template-hash=2471083592
            run=hello-nginx
Status:      Running
IP:          172.17.0.3
Controllers: ReplicaSet/hello-nginx-2471083592
Containers:
  hello-nginx:
    Container ID:  docker://98a9e303f0dbf21db80a20aea744725c9bd64f6b2ce2764379151e3ae422fc18
    Image:         nginx
    Image ID:      docker://sha256:ba6bed934df2e644fdd34e9d324c80f3c615544ee9a93e4ce3cfddfcf84bdbc2
    Port:          80/TCP
    State:         Running
      Started:     Tue, 04 Oct 2016 14:06:02 +0530
    Ready:         True
    Restart Count:  0
    Volume Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-rie7t (ro)
    Environment Variables:  <none>
..... /// REST OF THE OUTPUT ///
```

## Expose a Service

It is time now to expose our basic Nginx deployment as a service. We can use the command shown below:

```
PS C:\> .\kubectl.exe expose deployment hello-nginx --type=NodePort
service "hello-nginx" exposed
```

If we visit the Dashboard at this point and go to the Services section, we can see out **hello-nginx** service entry.



Alternately, we can use kubectl too, to check it out:

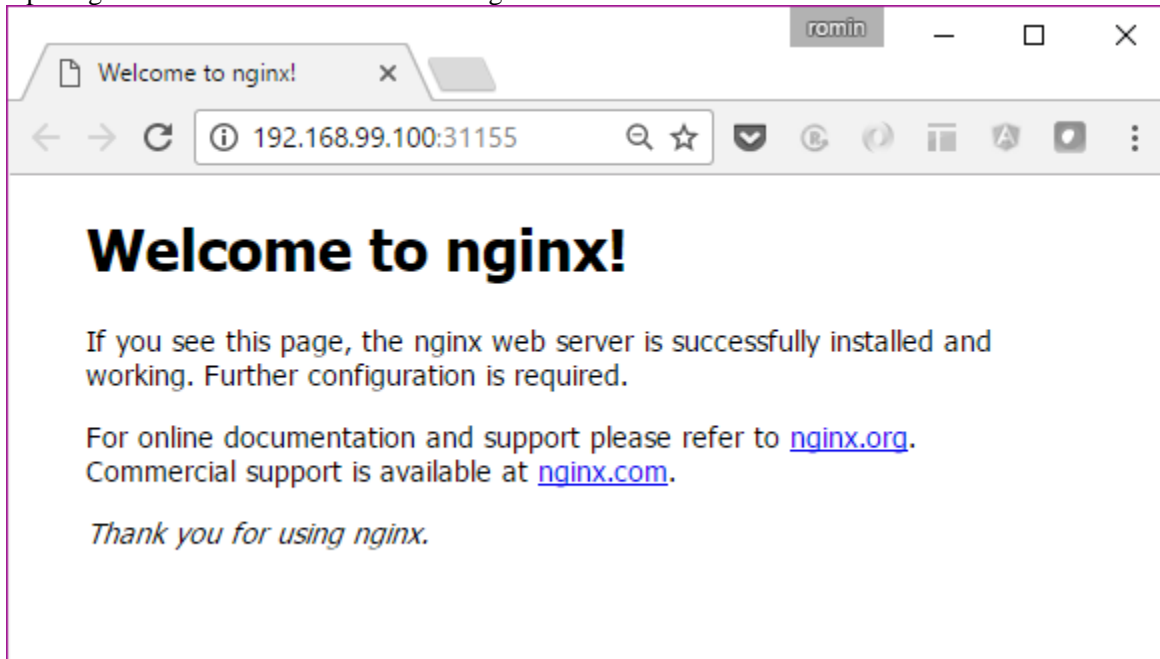
```
PS C:\> .\kubectl.exe get services
NAME          CLUSTER-IP  EXTERNAL-IP  PORT(S)  AGE
hello-nginx   10.0.0.24   <nodes>      80/TCP   3m
kubernetes    10.0.0.1    <none>       443/TCP  1h
PS C:\> .\kubectl.exe describe service hello-nginx
Name:         hello-nginx
Namespace:    default
Labels:       run=hello-nginx
Selector:     run=hello-nginx
Type:         NodePort
IP:           10.0.0.24
Port:         <unset> 80/TCP
NodePort:     <unset> 31155/TCP
Endpoints:    172.17.0.3:80
Session Affinity:  None
No events.
```

We can now use the minikube service to understand the URL for the service as shown below:

```
PS C:\> .\minikube.exe service --url=true hello-nginx  
http://192.168.99.100:31155
```

Alternately, if we do not use the url flag, then it can directly launch the browser and hit the service endpoint:

```
PS C:\> .\minikube.exe service hello-nginx  
Opening kubernetees service default/hello-nginx in default browser...
```



## View Logs

Assuming that you have accessed the service once in the browser as shown above, let us look at an interesting thing now. Go to the Service link in the Dashboard.

The screenshot shows the Kubernetes Dashboard in a web browser. The browser's address bar shows the URL `192.168.99.100:30000/#/service?namespace=default`. The dashboard's header is blue and contains the 'kubernetes' logo, the 'Services' tab, and a '+ CREATE' button. The left sidebar lists various Kubernetes resources under categories like 'Workloads', 'Services and discovery', and 'Storage'. The 'Services' tab is selected and highlighted. The main content area displays a table of services.

Name	Labels	Cluster IP	Internal endpoints	External endpoints
✓ <a href="#">hello-nginx</a>	run: hell...	10.0.0.24	hello-nginx... hello-nginx...	-
✓ <a href="#">kubernetes</a>	compon... provider...	10.0.0.1	kubernet... kubernet...	-

Click on the **hello-nginx** service. This will also show the list of Pods (single) as shown below. Click on the icon for Logs as highlighted below:

Kubernetes Dashboard

Services > hello-nginx

Workloads

- Deployments
- Replica Sets
- Replication Controllers
- Daemon Sets
- Pet Sets
- Jobs
- Pods

Services and discovery

- Services
- Ingress

Storage

- Persistent Volume Claims

Resource Details

Details

Name: hello-nginx

Namespace: default

Label selector: run: hello-nginx

Labels: run: hello-nginx

Type: NodePort

Connection

Cluster IP: 10.0.0.24

Internal endpoints: hello-nginx:80 TCP  
hello-nginx:31155 TCP

Pods

Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
hello-nginx-2471083592-4vfz8	Running	0	28 minutes	172.17.0.3	-	-

192.168.99.100:30000/#/log/default/hello-nginx-2471083592-4vfz8/?namespace=default

This will show the logs for that particular Pod and with HTTP Request calls that was just made.

Kubernetes Dashboard

Logs

Admin

- Namespaces
- Nodes
- Persistent Volumes

Namespace

default

Workloads

- Deployments
- Replica Sets
- Replication Controllers

Logs from hello-nginx in hello-nginx-2471083592-4vfz8

```

2016-10-04T09:00:33.300266903Z 172.17.0.1 - - [04/Oct/2016:09:00:33
+0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Windows NT 10.0;
Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/53.0.2785.143 Safari/537.36" "-"
2016-10-04T09:00:33.548308476Z 2016/10/04 09:00:33 [error] 5#5: *1
open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file
or directory), client: 172.17.0.1, server: localhost, request: "GET
/favicon.ico HTTP/1.1", host: "192.168.99.100:31155", referer:
"http://192.168.99.100:31155/"
2016-10-04T09:00:33.548358270Z 172.17.0.1 - - [04/Oct/2016:09:00:33
+0000] "GET /favicon.ico HTTP/1.1" 404 571
"http://192.168.99.100:31155/" "Mozilla/5.0 (Windows NT 10.0; Win64;

```

Logs from 10/4/16 2:30 PM to 10/4/16 2:30 PM

You could do the same by using the logs <podname> command for the kubectl CLI:

```
PS C:\> .\kubectl logs hello-nginx-2471083592-4vfz8
```

```
172.17.0.1 - - [04/Oct/2016:09:00:33 +0000] "GET / HTTP/1.1" 200 612 "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36" "-"
2016/10/04 09:00:33 [error] 5#5: *1 open() "/usr/share/nginx/html/favicon.ico" failed (2: No such file or directory), client: 172.17.0.1, server: localhost, request: "GET /favicon.ico HTTP/1.1", host: "192.168.99.100:31155", referrer: "http://192.168.99.100:31155/"
172.17.0.1 - - [04/Oct/2016:09:00:33 +0000] "GET /favicon.ico HTTP/1.1" 404 571 "http://192.168.99.100:31155/" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36" "-"
PS C:\>
```

## Scaling the Service

OK, I am not yet done!

When we created the deployment, we did not mention about the number of instances for our service. So we just had one Pod that was provisioned on the single node.

Let us go and see how we can scale this via the scale command. We want to scale it to 3 Pods.

```
PS C:\> .\kubectl scale --replicas=3 deployment/hello-nginx
deployment "hello-nginx" scaled
```

We can see the status of the deployment in a while:

```
PS C:\> .\kubectl.exe get deployment
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
hello-nginx	3	3	3	3	1h

Now, if we visit the Dashboard for our Deployment:

Kubernetes Dashboard

192.168.99.100:30000/#/deployment?namespace=default

kubernetes Deployments + CREATE

Admin

- Namespaces
- Nodes
- Persistent Volumes

Namespace

default

Workloads

- Deployments
- Replica Sets
- Replication Controllers
- Daemon Sets
- Pet Sets

Name	Labels	Pods	Age	Images
✓ hello-nginx	run: hell...	3 / 3	an hour	nginx

We have the 3/3 Pods available. Similarly, we can see our Service or Pods.



Kubernetes Dashboard

192.168.99.100:30000/#/service/default/hello-nginx?namespace=default

kubernetes Services > hello-nginx EDIT DELETE + CREATE

Replica Sets  
Replication Controllers  
Daemon Sets  
Pet Sets  
Jobs  
Pods

Services and discovery  
Services  
Ingress

Storage  
Persistent Volume Claims

Config  
Secrets

**Details**

Name: hello-nginx  
Namespace: default  
Label selector: run: hello-nginx  
Labels: run: hello-nginx  
Type: NodePort

**Connection**

Cluster IP: 10.0.0.24  
Internal endpoints: hello-nginx:80 TCP  
hello-nginx:31155 TCP

**Pods**

Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
hello-nginx...	Running	0	-	172.17.0.5	-	-
hello-nginx...	Running	0	an hour	172.17.0.3	-	-
hello-nginx...	Running	0	-	172.17.0.4	-	-

or the Pod list:

Kubernetes Dashboard

192.168.99.100:30000/#/pod?namespace=default

kubernetes Pods + CREATE

Namespace  
default

Workloads  
Deployments  
Replica Sets  
Replication Controllers  
Daemon Sets  
Pet Sets  
Jobs  
Pods

Services and discovery  
Services  
Ingress

Name	Status	Restarts	Age	Cluster IP	CPU (cores)	Memory (bytes)
hello-nginx...	Running	0	12 seconds	172.17.0.5	-	-
hello-nginx...	Running	0	an hour	172.17.0.3	-	-
hello-nginx...	Running	0	12 seconds	172.17.0.4	-	-

## Stopping and Deleting the Cluster

This is straightforward. You can use the **stop** and **delete** commands for minikube utility.

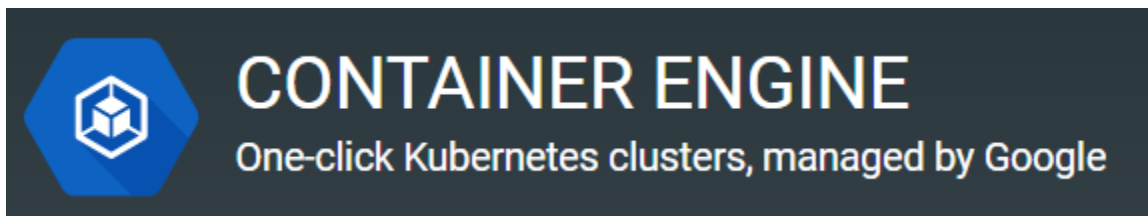
## Limitations

Minikube is a work in progress at this moment and it does not support all the features of Kubernetes. Please [refer to the minikube documentation](#), where it clearly states what is currently supported.

## A Note on Google Container Engine

Now that you have been able to see the basic building blocks of Kubernetes run on your machine, the next step for you might be to try it out on an actual cluster of machines (VMs) and see it all work.

One of the best managed solutions out there for k8s is Google Container Engine. A single command and you are up and running with a fully managed Kubernetes cluster with a few VMs in a matter of minutes. You can then set the configuration for the kubectl command line utility to work with that cluster and perform the operations. Give it a try.



<https://cloud.google.com/container-engine/>

Google hosts some excellent codelabs on Google Container Engine to get you started. I have learnt a lot through them. Check them out:

- [Running a Container in Kubernetes with Container Engine](#)
- [Orchestrating the Cloud with Kubernetes](#)
- [Hello Node Kubernetes Codelab](#)

# Troubleshooting Issues on Windows 10

My experience to get the experimental build of minikube working on Windows was not exactly a smooth one, but that is to be expected from anything that calls itself experimental.

I faced several issues and hope that it will save some time for you. I do not have the time to investigate deeper into why some of the stuff worked for me since my focus is to get it up and running on Windows. So if you have some specific comments around that, that will be great and I can add to this blog post.

In no order of preference, here you go:

## Use PowerShell

I used Powershell and not command line. Ensure that Powershell is launched in Administrative mode. This means Ctrl + Shift + Enter.

## Put minikube.exe file in C:\ drive

I saw some issues that mentioned to do that. I did not experiment too much and went with C:\ drive.

## Clear up .minikube directory

If there were some issues starting up minikube first time and then you try to start it again, you might see errors that say stuff like “Starting Machine” or “Machine exists” and then a bunch of errors before it gives up. I suggest that you clear up the .minikube directory that is present in %HOMEPATH%\minikube directory. In my case, it is C:\Users\irani\_r\minikube. You will see a bunch of folders there. Just delete them all and start all over again.

To see detailed error logging, give the following flags while starting up the cluster:

```
--show-libmachine-logs --alsologtostderr
```

Example of the error trace for me was as follows:

```
PS C:\> .\minikube start --show-libmachine-logs --alsologtostderr
W1003 15:59:52.796394 12080 root.go:127] Error reading config file at C:\Users\irani_r\minikube\config\config.json:
o
pen C:\Users\irani_r\minikube\config\config.json: The system cannot find the file specified.
I1003 15:59:52.800397 12080 notify.go:103] Checking for updates...
Starting local Kubernetes cluster...
I1003 15:59:53.164759 12080 cluster.go:75] Machine exists!
```

```
I1003 15:59:54.133728 12080 cluster.go:82] Machine state: Error
E1003 15:59:54.133728 12080 start.go:85] Error starting host: Error getting state for host: machine does not exist. Re
trying.
I1003 15:59:54.243132 12080 cluster.go:75] Machine exists!
I1003 15:59:54.555738 12080 cluster.go:82] Machine state: Error
E1003 15:59:54.555738 12080 start.go:85] Error starting host: Error getting state for host: machine does not exist. Re
trying.
I1003 15:59:54.555738 12080 cluster.go:75] Machine exists!
I1003 15:59:54.790128 12080 cluster.go:82] Machine state: Error
E1003 15:59:54.790128 12080 start.go:85] Error starting host: Error getting state for host: machine does not exist. Re
trying.
E1003 15:59:54.790128 12080 start.go:91] Error starting host: Error getting state for host: machine does not exist
Error getting state for host: machine does not exist
Error getting state for host: machine does not exist
```

## Disable Hyper-V


As earlier mentioned, VirtualBox and Hyper-V are not the happiest of co-workers. Definitely disable one of them on your machine. As per the documentation of minikube, both virtualbox and hyperv drivers are supported on Windows. I will do a test of Hyper-V someday but I went with disabling Hyper-V and used VirtualBox only. The steps to disable Hyper-V correctly were shown earlier in this blog post.

## Conclusion

Hope this blog post gets you started with Kubernetes on your Windows development machine by using minikube. Please let me know about your experience in the comments, it will help me tune the article—especially the Troubleshooting section :-)

## Part 13 – Docker Management Commands

The recent release of **Docker 1.13.0** saw some interesting updates. One of the updates that interests me as an instructor is the release of the Docker Management Commands.

 About Docker



# Docker

Version 1.13.0 (9795)

Channel: Stable

0c6d765

Copyright © 2016 Docker Inc. All Rights Reserved.

Docker and the Docker logo are trademarks of Docker Inc.

Registered in the U.S. and other countries.

But first let me tell you what I am talking about. Often I noticed that the students in my Docker class used to struggle a bit with the commands when it came to which resource they were dealing with. For e.g. were you working with an image or do you need to give the container id and so on.

As an example, consider the following docker command:

```
$ docker rm
```

When typing these commands, there is often the confusion in the minds of the user if we need to provide an image id or does this apply to container id. Often this resulted in looking up for help which would tell you what was expected, as given below:

```
$ docker rm
```

“docker rm” requires at least 1 argument(s).

See ‘docker rm — help’.

Usage: docker rm [OPTIONS] CONTAINER [CONTAINER...]

Remove one or more containers

This is now addressed in Docker 1.13.0 with the release of Management Commands. If you simply type `docker` at the terminal, you will see the section on Management Commands as given below:

Management Commands:

checkpoint Manage checkpoints  
container Manage containers  
image Manage images  
network Manage networks  
node Manage Swarm nodes  
plugin Manage plugins  
secret Manage Docker secrets  
service Manage services  
stack Manage Docker stacks  
swarm Manage Swarm  
system Manage Docker  
volume Manage volumes

So if you are dealing with images, you know that you can begin your command logically with

```
$ docker image <command>
```

or if you are dealing with containers, it should logically be:

```
$ docker container <command>
```

This is good design in my opinion and it is something that I have seen across other Command Line Interfaces (CLI) too, that make you mentally decide which resource you want to manage and then there would be typically commands like listing, adding, removing, monitoring, etc.

## Additional comments

- To see the management commands, ensure that you are running Docker version 1.13.0
- The Management Commands as well as the older commands work side by side. The older commands have not got removed and most likely won't due to the 1000s of scripts that might be automated everywhere.
- There could be a bit of confusion in the commands. To illustrate that, consider the remove command. Prior to Docker 1.13.0, it was `docker rmi` for removing images and `docker rm` for removing containers. Both of these commands are still available. However, if you consider the new management commands, then there is a single command now called

'rm'. So to remove the images, you would use `docker image rm` and to remove the containers, you would use `docker container rm`. This makes perfect sense but could take a bit to getting used to typing, especially if you have spent hours typing and practicing the older commands.