Hands on Docker

By
Navin Sabharwal
Bibin W

Hands on Docker

Dedicated to the people I love and the God I trust.

— Navin Sabharwal

Dedicated to people who made my life worth living and carved me into an individual I am today and to God who shades every step of my life.

— Bibin W

Contents at a Glance

Contents

ॐ शान्तिः शान्तिः शान्तिः।।

ॐ असतो मा सद्गमय
तमसो मा ज्योतिर्गमय
मृत्योर्मा अमृतं गमय।।

Om Asato ma sadgamaya
Tamaso ma jyotirgamaya
Mrityorma amritam gamaya

बहद रणयक उपनषिद 1.3.28

From untruth, lead me to the truth;
From darkness, lead me to the light;
From death, lead me to immortality.

About the Authors

Navin Sabharwal is an innovator, thought leader, author, and consultant in the areas of virtualization, cloud computing, big data and analytics.

Navin has created niche award-winning products and solutions and has filed numerous patents in diverse fields such as IT services, virtual machine placement, cloud capacity analysis, assessment engines, ranking algorithms, capacity planning engines, and knowledge management.

Navin holds a Masters in Information Technology and is a Certified Project Management Professional.

Navin has authored the following books: Cloud Computing First Steps (Publisher: CreateSpace, ISBN#: 978-1478130086), Apache Cloudstack Cloud Computing (Publisher: Packt Publishing, ISBN#: 978-1782160106), Cloud Capacity Management (Publisher Apress, ISBN #: 978-1430249238)

Bibin W has been working with virtualization and cloud technologies, he is a subject matter expert in VMware, Linux Container, Docker, Amazon Web Services, Chef and Azure.

Bibin holds a Masters in Computer Science from SRM University, Chennai.

The authors can be reached at architectbigdata@gmail.com.

Acknowledgments

Special thanks go out to the people who have helped in creation of this book Dheeraj Raghav for his creative inputs in the design of this book, Piyush Pandey for his reviews and insights into the content.

The authors will like to acknowledge the creators of virtualization technologies and the open source community for providing such powerful tools and technologies and enable products and solutions which solve real business problems easily and quickly.

Preface

Docker is making waves in the technology circles and is rapidly gaining mindshare from developers, startups, technology companies and architects.

We all know how virtualization has changed the datacenter and cloud forever, virtualization has allowed enterprises and cloud providers to make the datacenter more agile, manageable, cloud friendly and application friendly. However virtualization has overheads of the guest operating system and costly licensing for virtualization software, thus limiting the utilization of the host.

The Containerization technology is seeing resurgence with Docker, containerization has been around since many years, and however it is now that Docker has revived the interest of the technology community in containers.

Fundamental support for containerization was actually included in the Linux 2.6.24 kernel to provide operating system-level virtualization and allow a single host to operate multiple isolated Linux instances, called Linux Containers (LXC). LXC is based on Linux control groups (cgroups) where every control group can offer applications complete resource isolation (including processor, memory and I/O access). Linux Containers also offer complete isolation for the container's namespace, so supporting functions like file systems, user IDs, network IDs and other elements usually associated with operating systems are unique for each container.

Docker uses the container technology but creates a layer above the LXC layer for packaging, deployment and migration of workloads to different hosts.

Docker container technology has taken the cloud and application development world by storm since it was open-sourced a little over a year ago, offering a way to package and deploy applications across a variety of Linux instances.

Enterprises stand to gain by further reducing the datacenter footprint and using the host’s resources to their maximum using the Docker and LXC technology. Coupled with the ease of migration and fast scale out of containers it is turning

out to be a technology which is well suited for the cloud use case.

Docker is also going to have an impact on the devops lifecycle, by providing capabilities to support immutable infrastructure model, technologies like Docker may fundamentally change the way the operations world works, rather than updating the current running instances of operating systems, organizations may move to a model where the server container itself is replaced with a newer version and the old ones are taken out.

This book will help our readers to appreciate the Docker technology, the benefits and features provided by Docker and get a deep dive technical perspective on architecting solutions using Docker.

The book will enable a reader to appreciate, install, configure, administer and deploy applications on the Docker platform.

We sincerely hope our readers will enjoy reading the book as much as we have enjoyed writing it.

About this book

This book

- Introduces Docker to readers, the core concepts and technology behind Docker.
- Provides hands on examples for installing and configuring Docker
- Provides insight into packaging applications using Docker and deploying them.
- Provides step by step guidelines to have your Docker setup ready
- Detailed coverage of Mesosphere for Docker deployment and management
- Detailed coverage of Kubernetes clusters and Fleet.
- Hands on coverage of deployment tools including Fig, Shipyard and Panamax
- Step by Step guidelines to help you package your application on Docker
- Introduction to Google Container Engine for Docker

What you need for this book

Docker supports the most popular Linux and UNIX platforms.

Download the latest stable production release of Docker from the following URL:

https://docs.Docker.com/installation/

In this book we have focused on using Docker on a 64-bit Ubuntu 14.04 platform and at places have cited references on how to work with Docker running on other Linux and windows platforms.

At the time of writing, the latest stable Docker production release is 1.3.1

We will be using 64-bit Ubuntu 14.04 for examples of the installation process.

Conventions Used In the Book

Italic indicates important points, commands.

This is used to denote the Code Commands

This is the Output of the command.....

This is used for Example commands

 This icon indicates statistics figures

 This icon indicates examples

 This icon indicates points to be noted.

 This icon indicates further reading links or references.

Who this book is for

This book would be of interest to Virtualization Professionals, Cloud Architects, technology enthusiasts, Application Developers.

The book covers aspects on Docker and provides advanced guidance on planning and deploying the Docker technology for creating Infrastructure as a Service Clouds to using the technology to package and deploy your applications.

1
Linux Containers

In this chapter we will cover the basics of Linux containers.

Virtualization refers to the creation of virtual machines which have an independent Operating Systems but the execution of software running on the virtual machine is separated from the underlying hardware resources. Also it is possible that multiple virtual machines can share the same underlying hardware.

The actual machine that runs the virtualization software is called host machine and the virtual machine running on top of the virtualization software is called the guest machine. The software that provides virtualization capabilities and abstracts the hardware is called a “Virtual Machine Manager” or a “Hypervisor”. Popular hypervisor platforms are VMware, HyperV, Xen and KVM.

Docker works on a technology called Linux containers. Linux containers have a different approach than virtualization; you may call it an OS level virtualization, which means all the containers run on top of one Linux operating system.

You can run the host OS directly on the hardware or it can be running on a virtual machine. Each container run’s as a fully isolated operating system.

Linux containers are light weight virtualization system running on top of an operating system. It provides an isolated environment almost similar to a standard Linux distribution.

Docker works with LXC Container-based virtualization. It is also called operating system virtualization One of the first container technologies on x86 was actually on FreeBSD, in the form of FreeBSD Jails.
In container virtualization rather than having an entire Operating System guest OS, containers isolate the guest but do not virtualize the hardware. For running containers one needs a patched kernel and user tools, the kernel provides process isolation and performs resource management. Thus all containers are running under the same kernel but they still have their own file system, processes,

memory etc.

Unlike virtual machines all containers running on a host use the same kernel. Moreover starting and stopping a container is much faster than a virtual machine. It delivers an environment as close as possible to a standard Linux distribution. Containers from the inside are like a VM and from outside like a bunch of Linux processes.

With container-based virtualization, installing a guest OS is done using a container template.

In container approach one is usually limited to a single operating system, thus you cannot run Linux and windows together.

There are various advantages of using containers as compared to virtualization in terms of performance and scalability. A container based solution works well if you intend to run many hundreds of guests with a particular operating system, because they carry lesser overhead. The number of virtual machines available with container approach can be much higher as compared to virtualization as resources are available to the application rather than being consumed by multiple Guest OS instances running on a host.

One area where containers are weaker than VMs is isolation. VMs can take advantage of ring -1 hardware isolation such as that provided by Intel's VT-d and VT-x technologies. Such isolation prevents VMs from 'breaking out' and interfering with each other. Containers don't yet have any form of hardware isolation, which makes them susceptible to exploits.

Docker works well within a VM, which allows it to be used on existing virtual infrastructure, private clouds and public clouds. Thus Virtualization and Containerization will co-exist and in future there may be a hybrid approach which provides a unified way to leverage and manage Virtualization and Containerization.

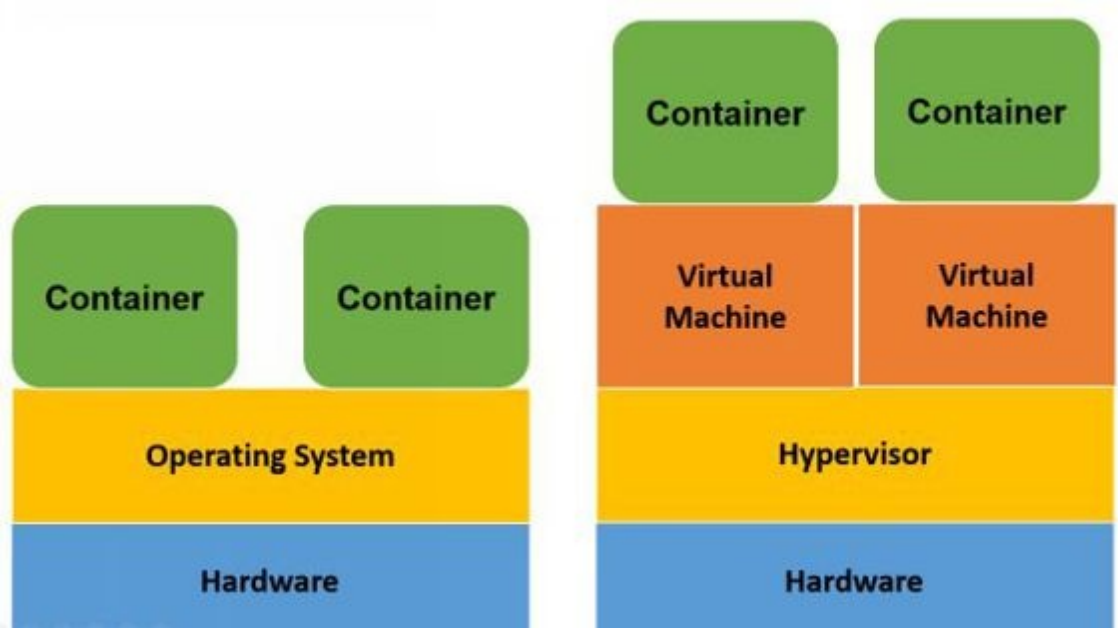


Fig 1-1: *Linux Containers*

Containers work on the concept of process level virtualization. Process level virtualization has been used by technologies like Solaris zones and BSD jails for years. But the drawback of these system is that they need custom kernels and cannot run on mainstream kernels. As opposed to Solaris zones and BSD rails, LXC containers have been gaining popularity in recent years because they can run on any Linux platform. This led to the adoption of containerization by various cloud based hosting services.

If you look into Linux based containers there are two main concepts involved,

1. Namespaces and
2. Cgroups (Control groups.)

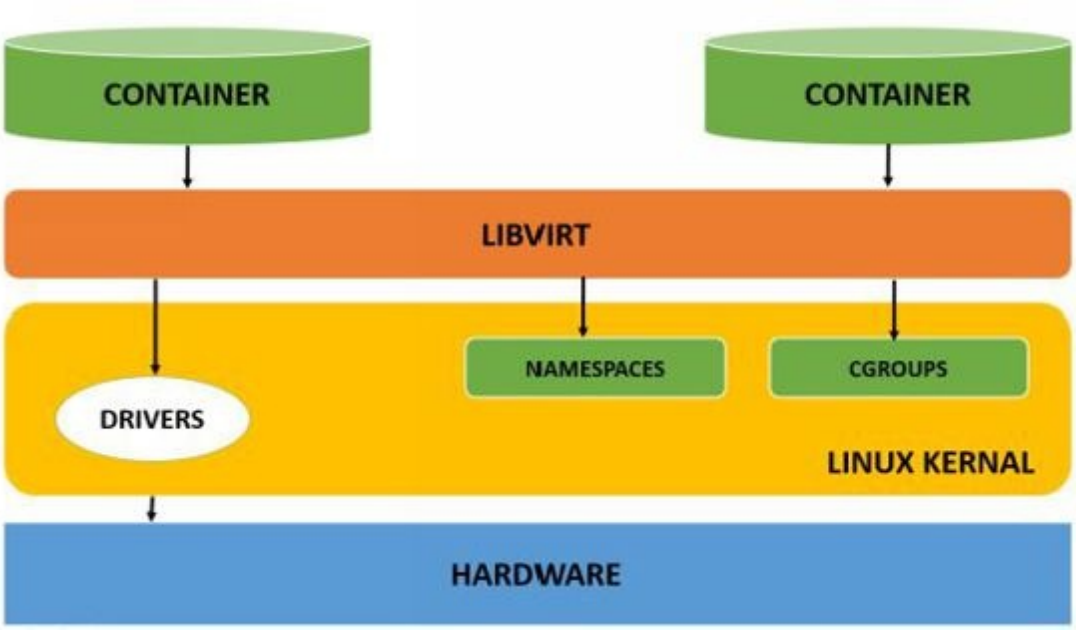


Fig 1-2: *Namespaces and Cgroups*

Namespaces:

In Linux there are six kinds of namespaces which can offer process level isolation for Linux resources. Namespaces ensure that each container sees only its own environment and doesn't affect or get access to processes running inside other containers. In addition, namespaces provide restricted access to file systems like chroot, by having a directory structure for a container.

The container can see only that directory structure and doesn't have any access to any level above it. Namespaces also allow containers to have its own network devices, so that each container can have its own IP address and hostname. This lets each container run independently of each other. Let's have a look at each namespace in detail.

Pid Namespace

This namespace is considered as most important isolation factor in containers. Every pid namespace forms its own hierarchy and it will be tracked by the kernel. Another important feature is that the parent pid's can control the children pid's but the children pid's cannot signal or control the parent pid.

Let's say we have ten child pid's with various system calls and these pid's are meaningful only inside the parent namespace. It does not have control outside its

parent namespace. So each isolated pid namespace will be running a container and when a container is migrated to another host the child pid’s will remain the same.

Net namespace

This namespace is used for controlling the networks. Each net namespace can have its own network interface. Let’s say we have two containers running with two different pid namespace and we want two different instances of Nginx server running on those containers. This can be achieved by net namespaces because each net namespace would contain its own network interface connected to an Ethernet bridge for connection between containers and the host.

Ipc namespace

This namespace isolates the inter-process communication.

Mnt namespace

This namespace isolates the file system mount points for a set of processes. It works more like an advanced and secure chroot option. A file system mounted to a specific mnt namespace and can only be accessed by the process associated with it.

Uts namespace

This namespace provides isolation for hostname and NIS domain name. This can be useful for scripts to initialize and configure actions based on these names. When hostname is changed in a container, it changes the hostname only for the process associated with that namespace.

User namespace

This namespace isolates the user and group ID namespaces. User namespace allows per-namespace mappings of user and group IDs. This means that a process's user and group IDs inside a user namespace will be different from its IDs outside of the namespace.

Moreover, a process can have a nonzero user ID outside a namespace while at the same time having a user ID of zero inside the namespace; in other words, outside its user namespace all the processes will have unprivileged access for

operations.

Cgroups

Cgroups (control groups) is a feature of Linux kernel for accounting, limiting and isolation of resources. It provides means to restrict resources that a process can use. For example, you can restrict an apache web server or a MySQL database to use only a certain amount of disk IO's.

So, Linux container is basically a process or a set of processes than can run in an isolated environment on the host system.

Before getting into Docker let's understand another important aspect of containers *"copy on write file system"*.

Copy on write file system:

In normal file system like ext4, all the new data will be overwritten on top of existing data and creates a new copy. Unlike other Linux file systems copy on write file system never overwrites the live data, instead it does all the updating using the existing unused blocks in the disk using copy on write functionality (COW). The new data will be live only when all the data has been updated to the disk.

For example, consider how data is stored in file system. File systems are divided in to number of blocks, let's say 16 blocks. So each innode will have 16 pointers to blocks. If a file stored is less than 16 blocks, the innode will point to the block directly. If the data exceeds 16 blocks, the 16 block will become a pointer to more blocks creating an indirect pointer.

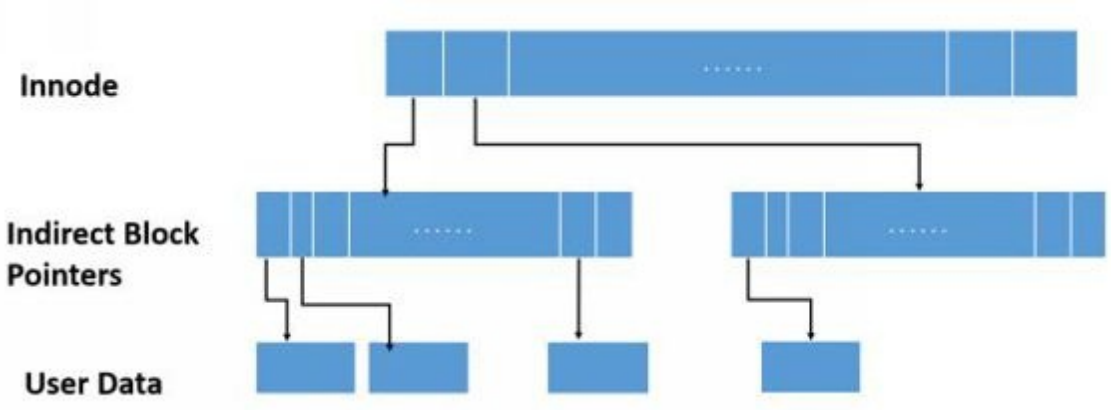


Fig 1-3: *copy on write*

When you modify an existing data, it will be written on unused blocks in the file system leaving the original data unmodified. All the indirect block pointers have to be modified in order to point to the new blocks of data. But the file system will copy all the existing pointers to modify the copy. File system will then update the innode again by modifying the copy to refer to the new blocks of indirect pointers. Once the modification is complete, the pointers to original data remain unmodified and there will be new set of pointers, blocks and innode for the updated data.

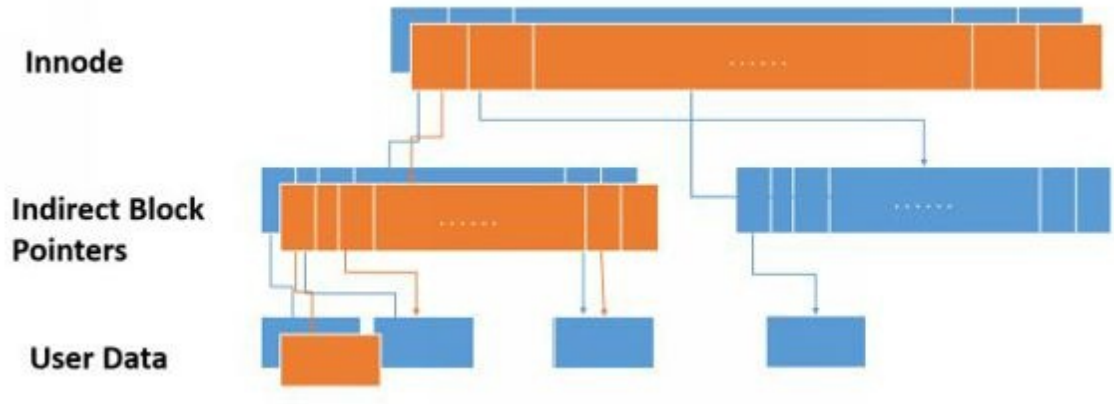


Fig 1-4: *copy on write mechanism*

One of the file systems used by Docker is BTRFS. Resources are handles using Copy on Write (COW) when same data is utilized by multiple tasks. When an application requests data from a file, the data is sent to memory or cache. Individual applications then have their own memory space. In the case when multiple applications request the same data, only one memory space is allowed by COW and that single memory space is pointed to by all applications. An

application, which is changing data, is given its own memory space with the new updated information. The other applications continue using the older pointers with original data.

BTRFS also uses the file system level snapshotting to implement layers. A snapshot is a read-only, point-in-time copy of the file system state. A storage snapshot is created using pre-designated space allocated to it. Whenever a snapshot is created, the metadata associated with the original data is stored as a copy. Meta data is the data which gives full information about the data stored in the disk. Also snapshot does not create a physical copy and creation of a snapshot is nearly immediate. The future writes to the original data will be logged and the snapshot cautiously keeps tracks of the changing blocks. The duty of the copy-on-write is to transfer the original data block to the snapshot storage, prior to the write onto the block. This in turn makes the data remain consistent in the time based snapshot.

Any “read-requests” to snapshots of unchanged data are reflected to the original volume. Requests are directed to the “copied” block only in the scenario when the requests are related to the changed data. Snapshots maintain meta-data, containing reports pertaining to the data blocks, which have been updated since the last snapshot was performed. Attention must be given to the fact that the data blocks are copied only at once, into the snapshot, on first write instance basis

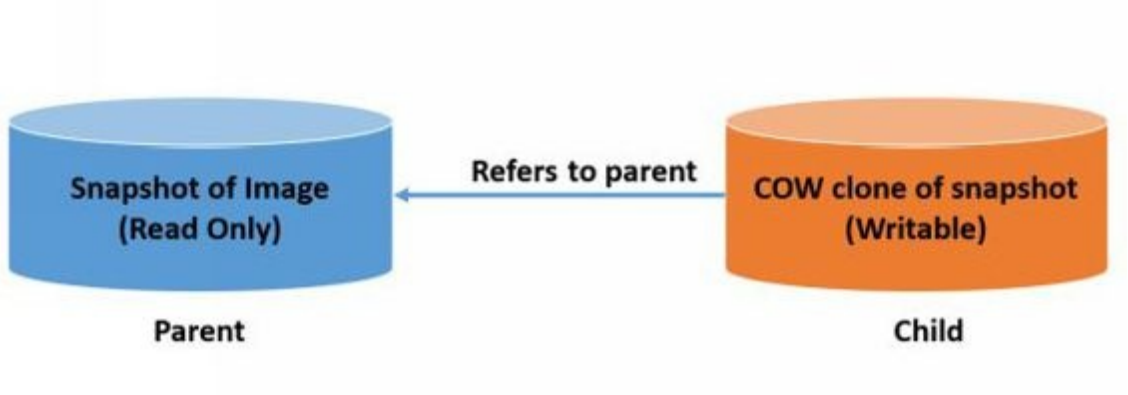


Fig 1-5: *COW image snapshot*

One of the main advantages of copy-on-write technique is its space efficiency. This is due to the fact that space required to create a snapshot is minimal, as it holds only the data which is being updated, also, the data is considered to be valid only when the original copy is available. The performance of original data volume is somewhat impacted by copy-on-write technique of snapshot, because the write requests to data blocks can only be performed when original data is being “copied” to the snapshot. Read requests are diverted to the original volume when the data remains unchanged.

2

Docker

Introduction

The best way to describe Docker is to use the phrase from the Docker web site—Docker is “an open source project to pack, ship and run any application as a lightweight container.” Thus the idea of Docker is to have an abstraction layer that allows the application developers to package any application and then let the containerization technology take care of the deployment aspects to any infrastructure.

Docker is analogous to shipping containers where you can load the goods in standardized containers and ship to different locations without much hassle. The advent of standardized containers made shipping fast and agile. Docker does the same with applications.

Docker platform can be used by developers and system administrators for developing and shipping applications to different environments. The decoupled parts of the application can be integrated and shipped to production environments really fast.

For example, a developer can install and configure an application in Docker container, pass it on to an ops person and he can deploy it on to any server running Docker. The application will run exactly like it ran on the developer’s laptop.

This amazing feature of Docker results in huge savings in the time and effort spent on deploying applications, ensuring that the dependencies are available and troubleshooting the deployment because of issues related to dependencies and conflicts.

Docker technology is well suited for applications deployed on cloud as it makes their migration simpler and faster.

Docker leverages LXC (Linux Containers), which encompasses Linux features like cgroups and namespaces for strong process isolation and resource control. However it is to be noted that Docker is not limited to LXC but can use any

other container technology in future and with the new release they now support libcontainer.

Docker leverages a copy-on-write file system and this allows Docker to instantiate containers quickly because it leverages the pointers to the existing files. Copy-on-write file system also provides layering of containers, thus you can create a base container and then have another container which is based on the base container.

Docker uses a “plain text” configuration language to define and control the configuration of the application container. This configuration file is called a DockerFile.

Docker makes use of Linux kernel facilities such as cGroups, namespaces and SElinux to provide isolation between containers. At first Docker was a front end for the LXC container management subsystem, but release 0.9 introduced libcontainer, which is a native Go language library that provides the interface between user space and the kernel.

Containers sit on top of a union file system, such as AUFS, which allows for the sharing of components such as operating system images and installed libraries across multiple containers.

A container is started from an image, which may be locally created, cached locally, or downloaded from a registry. Docker Inc operates the Docker Hub public registry, which hosts official repositories for a variety of operating systems, middleware and databases.

Most linux applications can run inside a Docker container, containers are started from images and running containers can be converted into images. There are two ways to create application packages for containers Manual and Dockerfile.

Manual builds

A manual build starts by launching a container with a base operating system image. Normal process for installation of an application on the operating system is performed and once the application is installed the container can be exported to a tar file or can be pushed to a registry like Docker Hub.

Dockerfile

This method is more scripted and automated for construction of a Docker Container. The Dockerfile specifies the base image to start and then the other installation on top are defined as a series of commands that are run or files that

are added to the container.

The Dockerfile also can specify other aspects of configuration of a container such as ports, default commands to be run on startup etc. Similar to the manual approach Dockerfile can be exported and the Docker Hub can use an automated build system to build images from a Dockerfile.

Why to use Docker

Let’s look at a few features which make Docker useful and attractive to application developers and infrastructure administrators alike:

Portable Deployments:

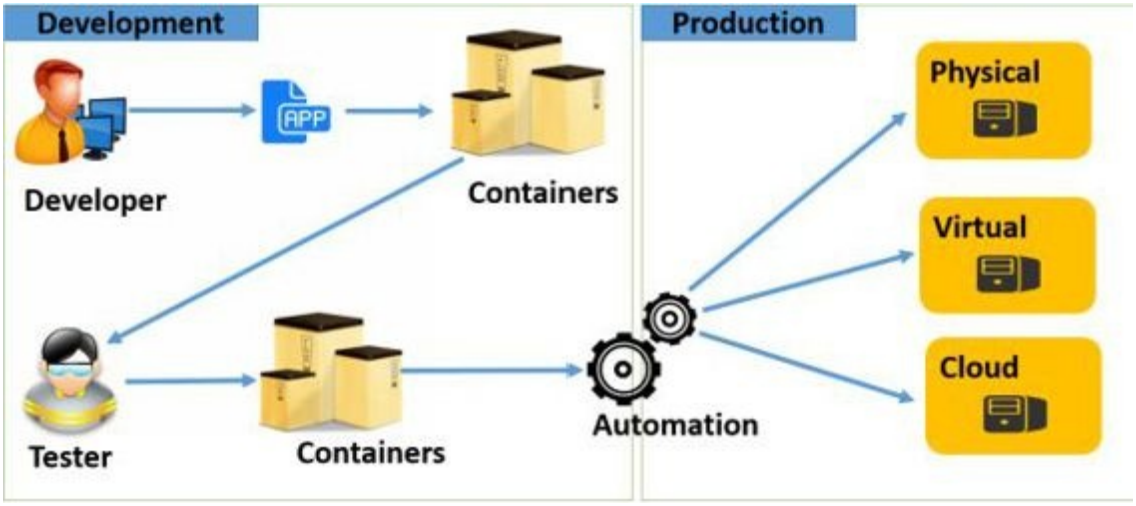
As containers are portable, the applications can be bundled in to a single unit and can be deployed to various environments without making any changes to the container.

Fast application delivery:

The workflow of Docker containers make it easy for developers, system administrators, QA and release teams to collaborate and deploy the applications to production environments really fast.

Because of the standard container format, developers only have to worry about the applications running inside the container and system administrators only have to worry about deploying the container on to the servers. This well segregated Docker management leads to faster application delivery.

Fig 2-1: *Docker application delivery and deployment*



Moreover, building new containers is fast because containers are very light weight and it takes seconds to build a new container. This in turn reduces the time for testing, development and deployment. Also, a container can be built in iterations, thus providing a good visibility on how the final application has been built.

Docker is great for development lifecycle. Docker containers can be built and packaged in developers laptop and can be integrated with continuous integration and deployment tools.

For example, when an application is packaged in a container by the developer, it can be shared among other team members. After that it can be pushed to the test environment for various tests. From the test environment you can then push all the tested containers to the production environment.

Scale and deploy with ease:

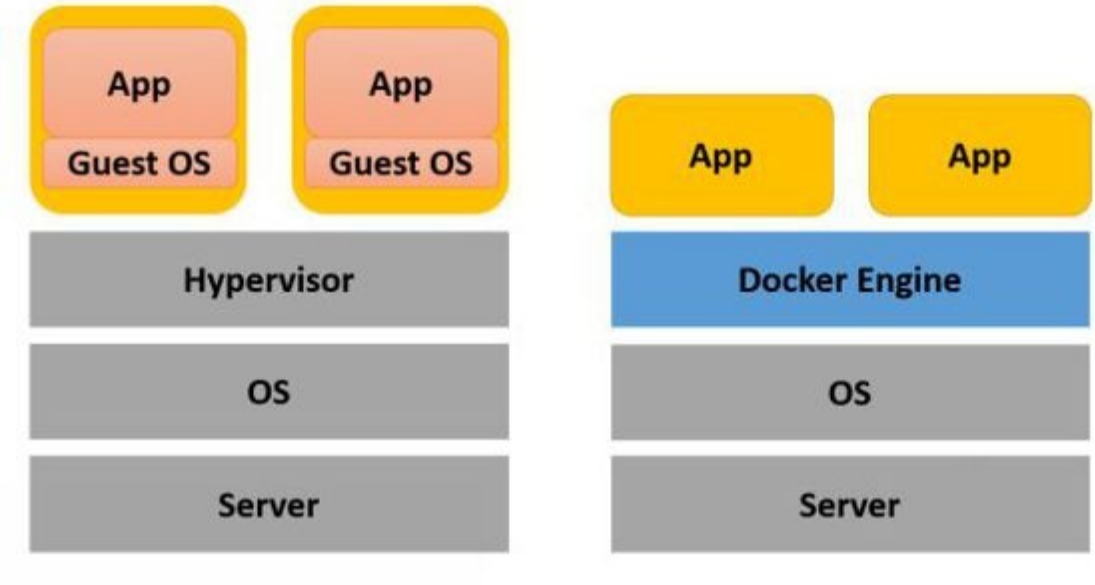
Docker containers can virtually run on any Linux system. Containers can be deployed on cloud environments, desktops, on premise datacenters, physical servers and so on. You can move containers from your desktop environment to cloud and back to physical servers easily and quickly.

Another interesting factor about container is scalability. Scaling up and down containers is blazingly fast. You can scale up containers from one to hundred’s and scale it down when not needed. Thus Docker containers are ideally suited

for scale out applications architected and built for the public cloud platforms.

Higher workloads with greater density:

Fig 2-2: *Virtual machine Vs. Docker containers*



More container applications can be deployed on a host when compared to virtual machines. Since there is no need for Docker to use a hypervisor, the server resources can be well utilized and cost of extra server resources can be reduced. Because Docker containers do not use a full operating system, the resource requirements are lesser as compared to virtual machines.

📌 *Few use cases*

1. Applications can be deployed easily on server with build pipeline.

2. Can be used in production environments with Mesos or Kunbernetes for application HA and better resource utilization.

3. Easy to clone the production environment in developer's workstation.

4. To perform load/scale testing by launching containers.

Docker Architecture:

Docker has client server architecture. It has a Docker client and a Docker daemon. The Docker client instructs the Docker daemon for all the container specific tasks. The communication between the Docker client and Docker daemon is carried out through sockets or through REST API's. Docker daemon

creates runs and distributes the containers based on the instructions from the Docker client. Docker client and Docker daemon can be on the same host or different hosts.

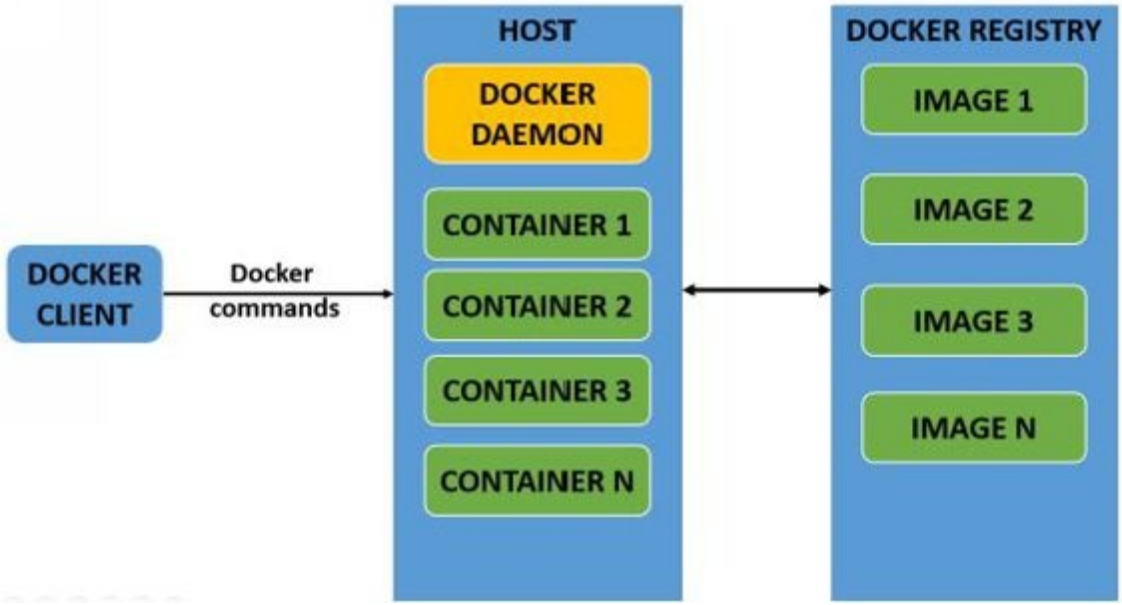


Fig 2-3: *Docker Architecture*

Docker Daemon:

Docker daemon is responsible for all the container operations. It runs on the host machine as shown in Fig 2-3. User cannot interact with the daemon directly instead all the instructions have to be sent through the Docker client.

Docker client:

Docker client can either be installed on the same host as Docker daemon or in a different host. It is the main interface to the Docker daemon. It accepts commands from the user and sends it to the Docker daemon for execution and provides the output to the user.

Docker internal components:

To understand Docker, you need to know about its three internal components. They are,

1. Docker image
2. Docker registry
3. Docker container.

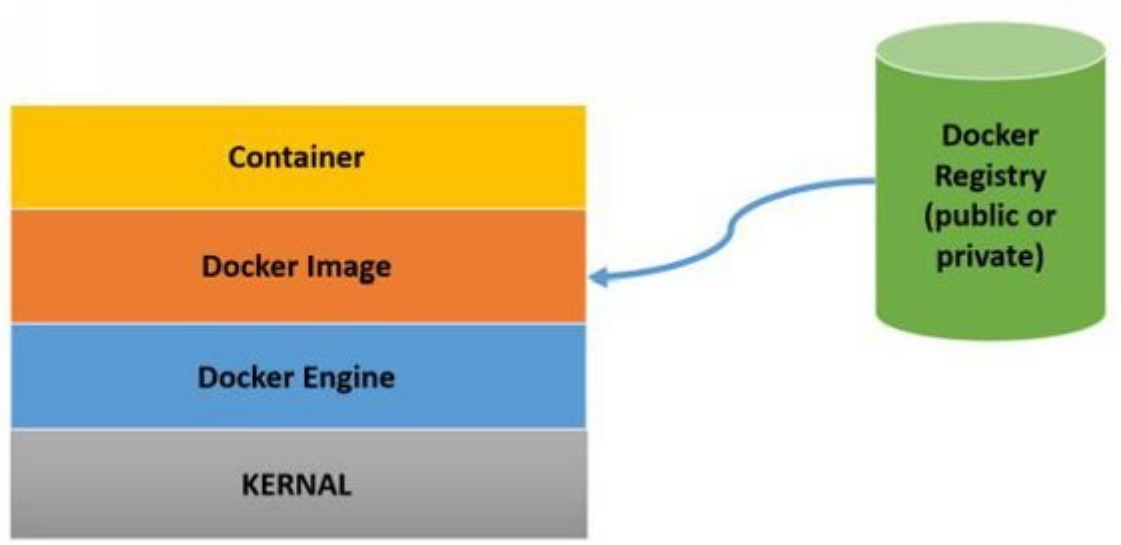


Fig 2-4: *Docker components*

Docker image:

A Docker image is like a golden template. An image consists of OS (Ubuntu, centos etc.,) and applications installed on it. These images are called base images. A Docker base image is the building block of a Docker container from where a container can be created. An image can be built from scratch using Docker inbuilt tools. You can also use Docker images created by other users from Docker public registry (Docker hub) as a base image for your containers.

Docker registry:

Docker registry is a repository for Docker images. It can be public or private. The public Docker registry maintained by Docker is called Docker hub. Users can upload and download images from the Docker registry. The public Docker registry has a vast collection of official and user created images. To create a container, you can either use the public images created by another user or you can use your own images by uploading it to the public or private registry.

Docker container:

A container is more of a directory and an execution environment for applications. It is created on top of a Docker image and it is completely isolated. Each container has its own user space, networking and security settings associated with it. A container holds all the necessary files and configurations for running an application. A container can be created, run, started, moved and

deleted.

Working of Docker:

So far we have learnt about Docker architecture and its components. Now let’s look in to how all the components come together to make Docker work.

Working of Docker image:

We have learnt the basic concept of a Docker image. In this section we will learn how exactly a Docker image works.

Each Docker image is an association of various layers. This layered approach provides a great way of abstraction for creating Docker images. These layers are combined in to a single unit using Uniform file system (AUFS). AUFS stores every layer as a normal directory, files with AUFS metadata. This ensures that all the files and directories are unique to the particular layer. AUFS creates a mount point by combining all the layers associated with the image. Any changes to the image will be written on the topmost layer.

This is the reason why the Docker containers are very light weight.

For example, when you make an update to an existing application, Docker either creates a layer on the existing image or updates the existing layer. The newly created layers will refer its previous layer. Docker does not rebuild the whole image again for the application like virtual machines. When you push the new updated image to the registry it will not redistribute the whole image, instead it updates just the layer on top of the existing base image. This makes Docker fast in creating new applications from the existing images.

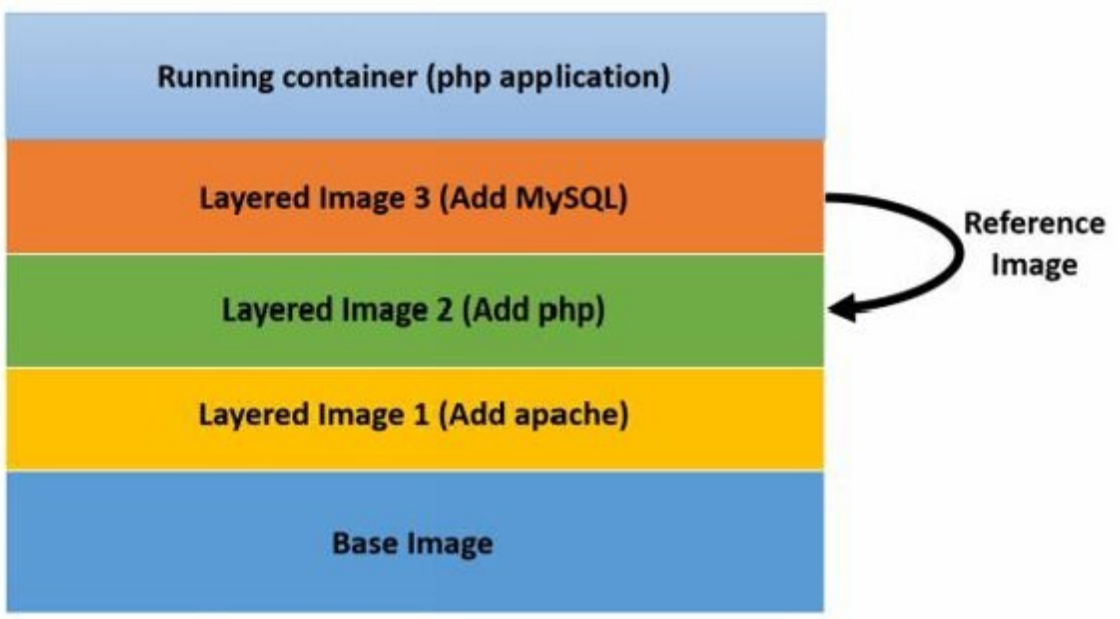


Fig 2-5: *Docker Images*

Points to remember

Every Docker image has a base image (eg: ubuntu ,centos, fedora, debian etc.,)

You can use you own images as a base image for your applications. Lets say you have a ubuntu image with mySQL installed on it. You can use this as a base image for all you database containers.

A configuration file called Dockerfile holds the instructions in descriptive form for building a new image. Dockerfile will have instructions for running commands, creating directories, adding files from host etc., We will learn about Docker file in detail in the subsequent chapters.

When you build a new image from scratch, the base image (specified by user) will be downloaded from the Docker hub and the applications specified in the instructions (Dockerfile) will be created as layers on top of the base image. Then you can use this image as a base image for your applications.

Working of Docker registry

Docker registry is used for storing Docker images. Images can be pushed and pulled from Docker registry for launching containers. There are two types of Docker registries.

1. **Public registry (Docker hub):** It contains official Docker images and user created images.
2. **Private registry:** A registry created by a user for storing private Docker images in your datacenter. You can also create a private registry in Docker hub.

All the images in the registry can be searched using Docker client. The images can be pulled down to the Docker host for creating containers and new images.

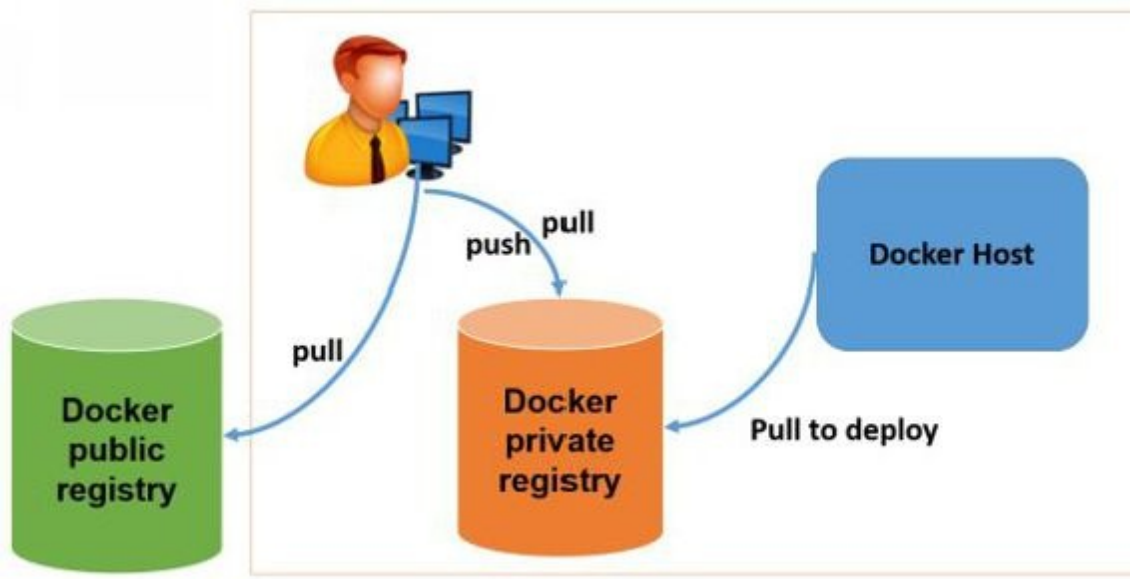


Fig 2-6: *Docker Registry*

Points to remember

1. *Public images can be searched and pulled by any one registered to Docker hub. Images can be searched from Docker client and Docker hub web ui.*
2. *Private images are only accessible by the registry owner and it does not appear on public registry search results.*
3. *You can create private registries in Docker hub.You can create one private registry for free and you need paid subscription for Docker hub to create more than one private registry.*

How container works:

As we learnt earlier, a container is an execution environment for applications. It contains all the operating system files, files added by users and metadata. Basically a container is launched from an image, so the applications which are installed and configured on an image will run on the container created from it. As the images are in ready only format, when a container is launched, a read/write layer will be added on top of the image for the container for the applications to run and make changes to it.

The Docker client receives the command from the Docker binary or REST API to run a container. The basic Docker command to run a container is shown below.

sudo docker run -i -t ubuntu /bin/bash

When you run the above command from Docker binary, here is what happens,

1. *Docker client will be launched with "Docker run" command.*
2. *It tells the daemon, from which image the container should be created. In our example, it is a Ubuntu image.*
3. *"-i" tells the Docker daemon to run the container in interactive mode.*
4. *"-t" represents tty mode for interactive session.*
5. *"/bin/bash" tells the Docker daemon to start the bash shell when the container is launched.*

On successful execution of the Docker run command, Docker will do the following actions at the backend.

1. Docker checks if the specified Docker image in the command is present locally on the host. If it is present locally, Docker will use that image for creating a container. If not, it will download the image from the public or private registry (based on the Docker host configuration) on the host.
2. The local or pulled image will be used for creating a new container.

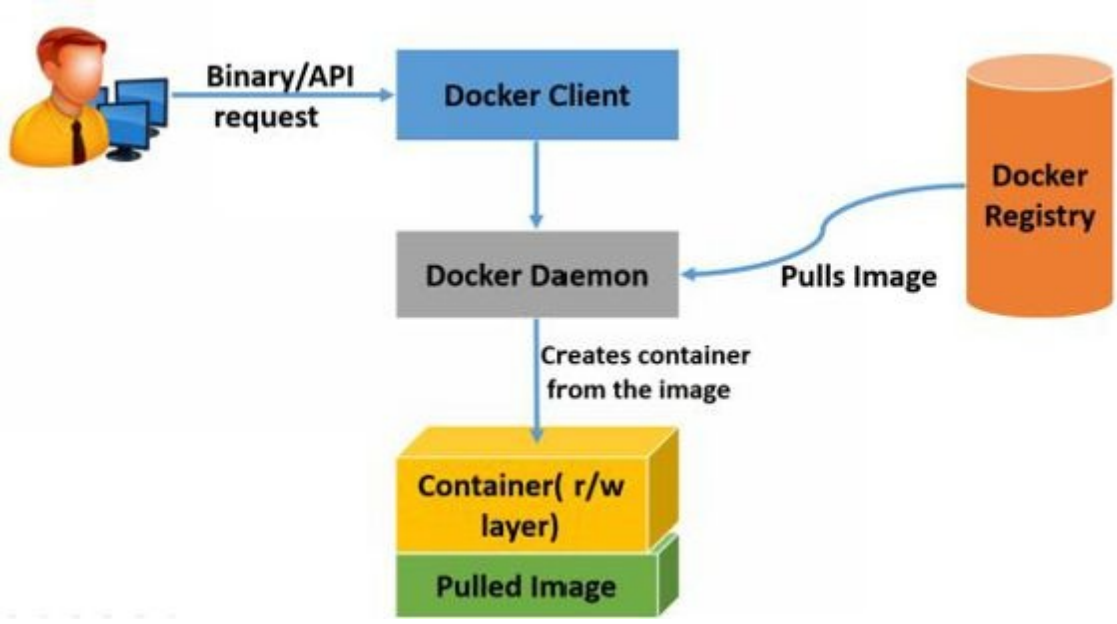


Fig 2-7: *working of a container.*

3. Once the image is set, Docker will create a read/write file system over the image.
4. Then Docker creates the necessary network interfaces for the container to interact with the host machine.
5. Docker checks for available IP address from the pool and allocates one to the container.
6. Then it executes the command specified in the command e.g. /bin/bash shell
7. Finally it logs all the input/output and errors for the user to find out the status of the container.

There are various other actions associated with the “Docker run” command. We will look in to it in the subsequent chapters.

Underlying Technology:

In this section we will see the underlying technology like namespace, cgroups and file systems of Docker.

Containers work on the concept of namespaces. Docker uses the same technology for isolating container execution environments. When you create a Docker container, Docker creates namespaces for it. These namespaces are the main isolation factor for the container.

Every element of a container has its own namespace and all the access remains inside that namespace. Thus the container elements do not have any access outside its namespace.

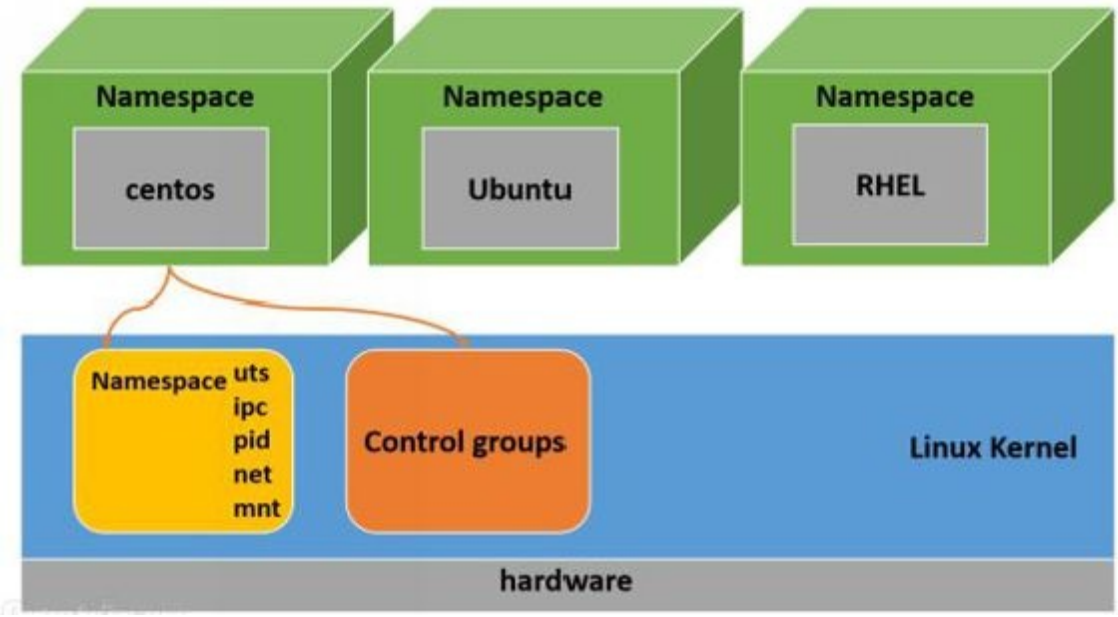


Fig 2-8: *Docker namespaces and cgroups.*

Namespaces:

The namespaces used by Docker are as follows,

1. PID namespace: this namespace isolates all the processes inside the container.
2. Net namespace: All the network isolation factors are taken care by the net namespace.
3. Ipc namespace: the inter-process communication in a container is managed by ipc namespace.
4. Mnt namespace: All the mount points for a container are managed by mnt namespace.
5. Uts namespace: All the kernel and versions are isolated and managed by the uts namespace.

Control groups (cgroups):

One of the interesting things about Docker is that, it can control the amount of resources used by a container. This functionality is achieved by Docker using Linux control groups (cgroups). Along with isolation, cgroups can allow and limit the available resources used by a container.

For example, you can limit the amount of memory used by a webserver container and provide more resources for the database container.

File systems:

In this section we will see the various file systems supported by Docker.

What makes Docker elegant is the well-organized use of layered images. Docker makes use of various features in kernel file system. Selecting a file system depends on your host deployment.

Let’s have a look at each backend file system that can be used by Docker.

Storage:

Docker has an efficient way of abstracting the backend storage. The storage backend stores a set of related image layers identified by a unique name. Following are the concepts involved in storage backend.

1. Every layer in an image is considered as a file system tree, which can be mounted and modified.
2. A layer can be created from scratch or it can be created on top of a parent layer.
3. The layer creation is driven by the concept of copy-on-write to make the layer creation very fast.
4. Every image in Docker is stored as a layer. When modifying an existing image, a layer will be added on top of that.
5. Container is the top most writable layer of an image. Each container has an init layer based on image and a child layer of init where all the container contents reside.
6. When creating a new image by committing a container, all the layers in the containers are bundled to form a new image.

When you install Docker, it will use any one of the file system backends mentioned below.

1. aufs
2. btrfs
3. devicemapper
4. vfs
5. overlayfs

Different OS distributions have different file system support. Docker is designed to choose a file system backed supported by the platform based on priority. Let’s have a look at the main file systems mentioned above.

Btrfs

Btrfs backend is a perfect fit for Docker because it supports the copy on write optimization.

As we learnt in earlier chapters, btrfs has rich features like file system layer snapshotting.

Btrfs is very fast when compared to other file systems used by Docker.

The disadvantage of btrfs is that it does not allow page cache sharing and it is not supported by SELinux.

AUFS

Aufs union file system is used originally by Docker and does not have support for many kernel and distributions. It can be used on Ubuntu based systems. We have learnt how aufs works in “working of images” section.

Devicemapper

The devicemapper backend uses the device-mapper thin provisioning module for implementing layers. Device-mapper is a block-level copy-on-write system.

Immutable infrastructure with Docker

Configuration management tools like chef and puppet allow you to configure a server to a desired state. These servers run for a long period of time by repeatedly applying configuration changes to the system to make in consistent and up to date. While immutable infrastructure is a model where a there is no need for application update, patches or configuration changes. It basically follows the concept of build once, run one or many instances and never change it again.

So, if you want to make configuration changes to an existing environment, a new image or container will be deployed with the necessary configuration.

Immutable infrastructure comprises of immutable components and for every deployment the components will be replaced with the updated component rather than being updated.

Docker container falls in the immutable infrastructure category. Docker is capable of building an immutable infrastructure because of its layered image concept and the fast booting of containers.

But there are few components in the later versions of Docker that are mutable.

Now in Docker containers you can edit /etc/hosts, /etc/hostname and /etc/resolv.conf files. This is helpful in place where other services want to override the default settings of these files.

The downside of this is that, these changes cannot be written to images. So you need a good orchestration tool to make these changes in containers while deployment.

Features of Immutable Infrastructure:

State isolation

The state in an immutable infrastructure is siloed. The borders among layers storing state and the layers that are temporary are clearly drawn and no leakage can possibly happen between those layers.

Atomic deployments

Updating an existing server might break the system and tools like chef and puppet can be used to bring the desired stated of the server back to the desired state.

But this deployment model is not atomic and the state transitions can go wrong resulting in an unknown state.

Deploying immutable servers or containers result in atomic deployments. Layered images in Docker help in achieving atomic deployments.

Easy roll back from preserved history

Every deployment in an immutable infrastructure is based on images. If anything goes wrong in the new deployment, the state can be rolled back easily from the preserved history. For example, in Docker, every update for an image is preserved as a layer. You can easily roll back to the previous layer if there is any issue in the new deployment.

Best practice

Achieving pure immutability is not practical. You can separate the mutable and immutable layers in your infrastructure for better application deployment.

Following are the two things to consider while planning for Immutable Infrastructure:

1. Separate Persistence layers from Immutable Infrastructure application deployment layers
2. Manage Persistence layers with convergence tools like chef, puppet, saltstack etc.

3

Installation

In chapter 1 and 2 we learnt the basics of Docker, its architecture, working and the core components. In this chapter we will learn the following.

1. Installing Docker on Linux platform
2. Installing Docker on Windows using boot2Docker
3. Test the installation by downloading a Docker image from Docker public registry.
4. Docker hub

Supported platforms:

Docker can be installed on all Linux and UNIX based operating systems. For windows and MAC, special wrappers such as boo2Docker and vagrant can be used for Docker deployments. Following are the main supported platforms for Docker.

1. Mac OS
2. Debian
3. RHEL
4. SUSE
5. Microsoft Azure
6. Gentoo
7. Amazon EC2

8. Google Cloud Platform
9. Arch Linux
10. Rackspace
11. IBM Softlayer
12. Digital ocean

Installing Docker on windows:

Windows cannot support Docker native installation. But in order to make Docker work on windows, there is a light weight Linux distribution called boot2Docker. Boot2Docker uses virtual box from oracle as the backend to work with Docker containers. Boot2Docker comes bundled as a ready to install exe package.

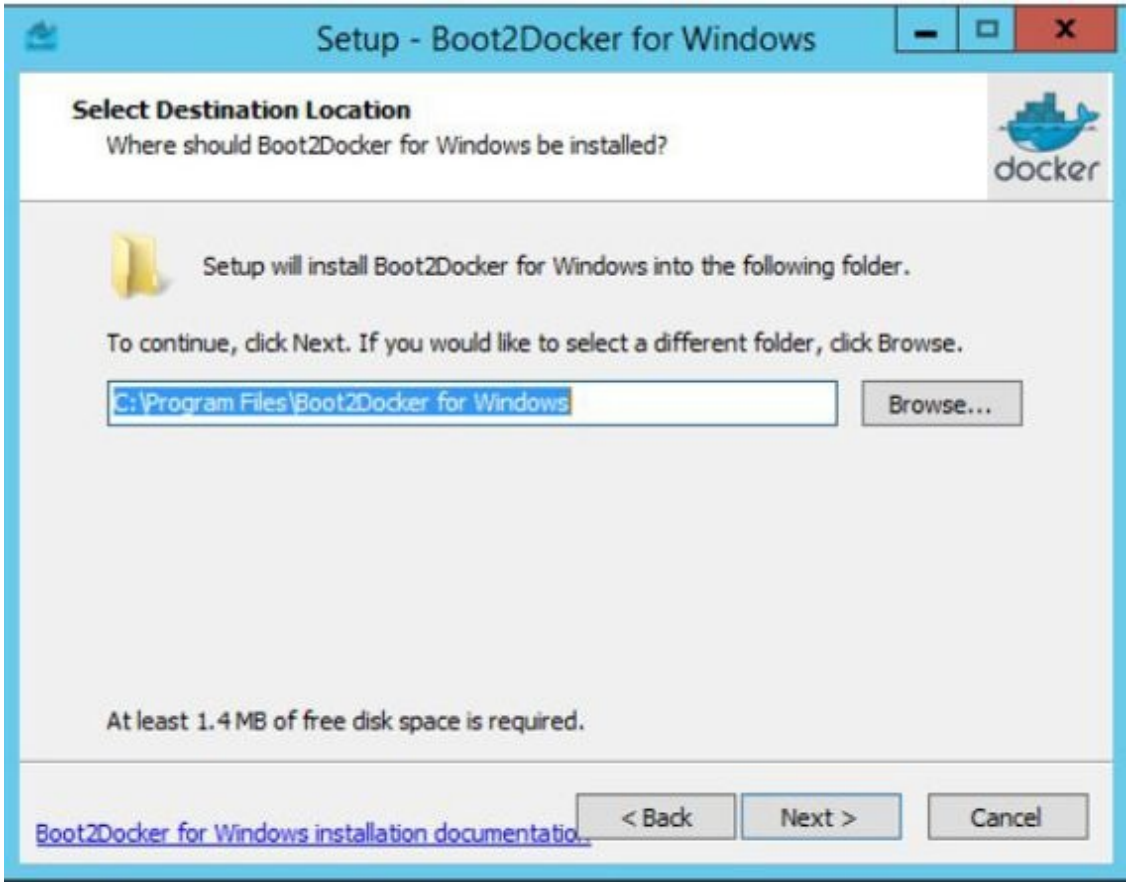
To install boot2Docker on a windows machine:

1. Download the latest boot2Docker application from here https://github.com/boot2Docker/windows-installer/releases

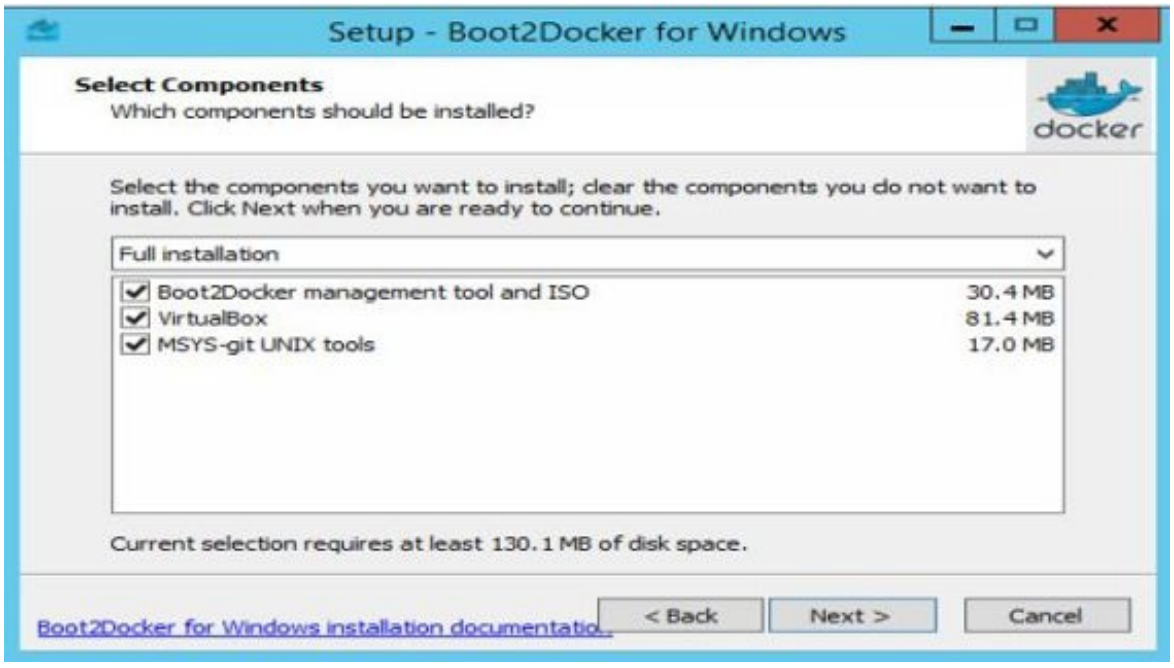
In case the location has changed, please search for the latest version and location.

2. Double click the downloaded application, start the installer and click next.

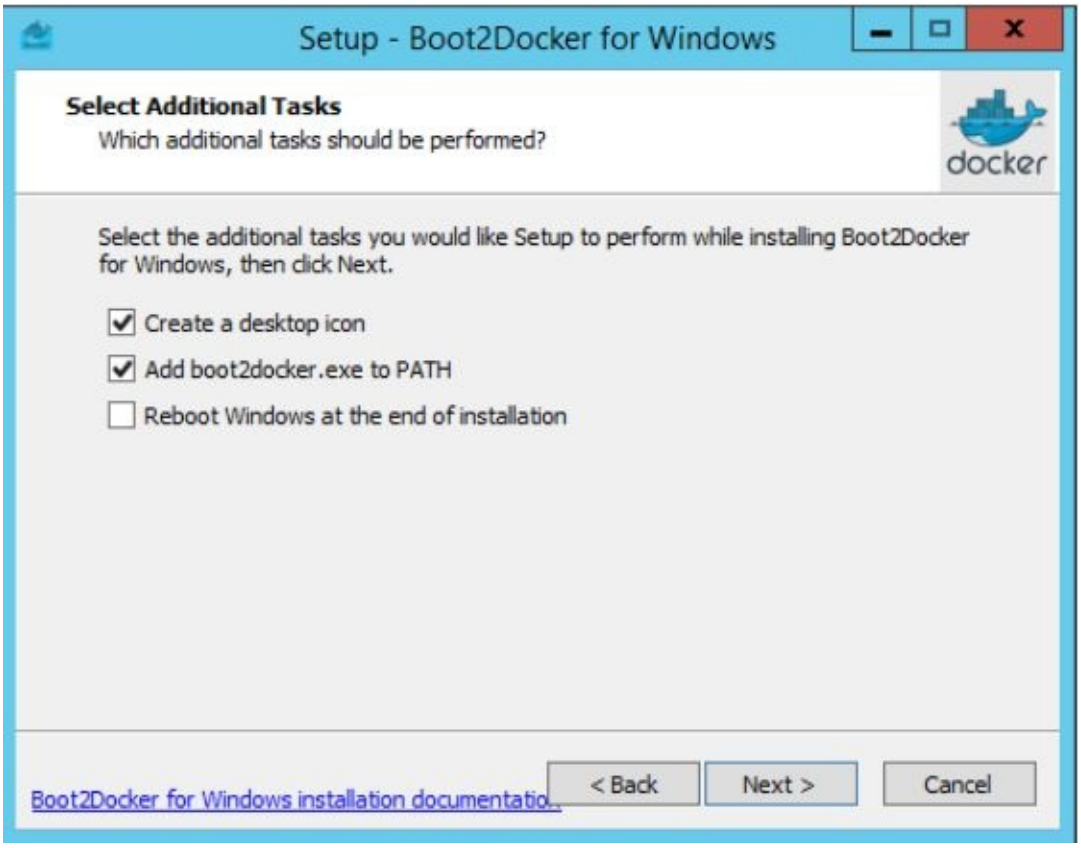
3. Select the installation folder and click next



4. Select the components you want to install. If virtual box is already installed on your workstation, deselect it and install the other two components, else install all components.

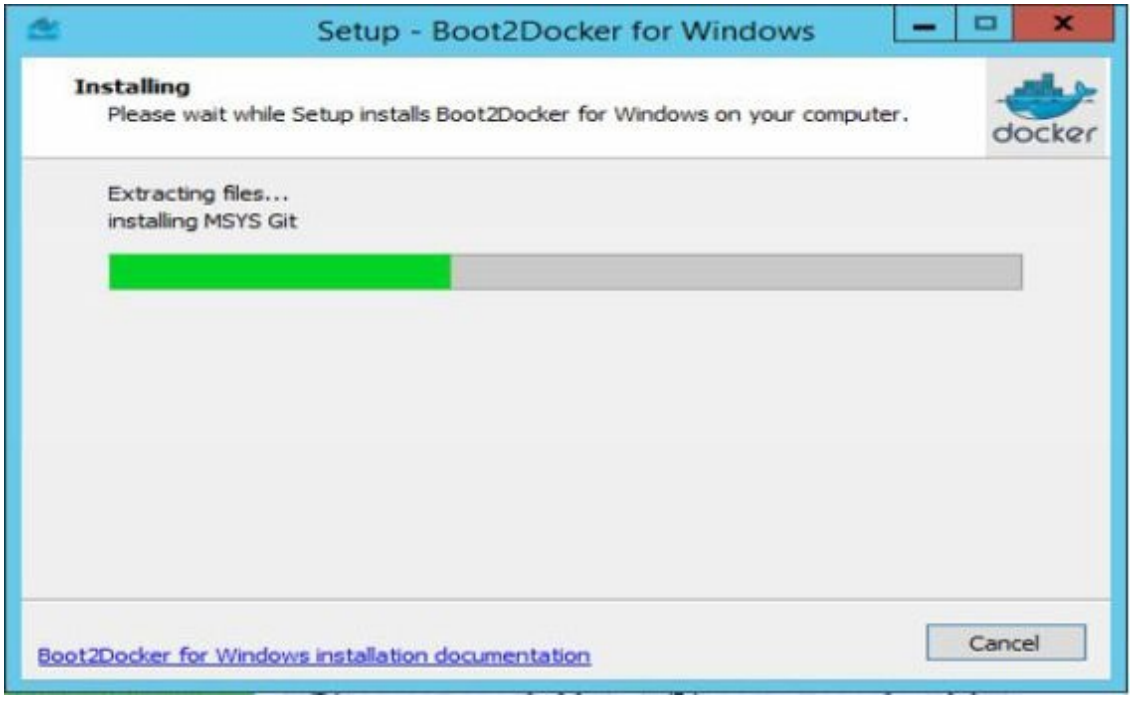


5. Select additional tasks listed below for creating desktop shortcut and PATH variable. Click next to continue.



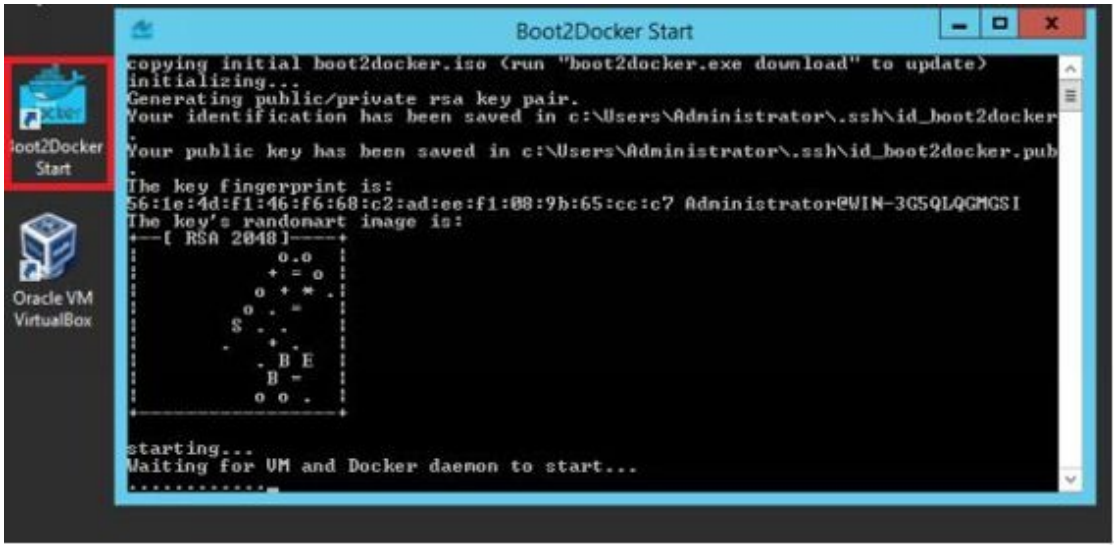
6. Click the install option.

7. It will take a while. Wait for the installation to be complete.



8. Once the installation is complete, click finish.

9. Click and run the boot2Docker script from the shortcut created in the desktop. It will open the Docker terminal which connects to the boo2Docker VM in virtual box.



10. Once the terminal connects to the boot2Docker VM run the following Docker command to test the installation by creating a busybox container.

docker run -i -t busybox /bin/sh

The above command will download the busybox image and start a container

with sh shell.

```
docker@boot2docker:~$ docker run -i -t busybox /bin/sh
Unable to find image 'busybox' locally
latest: Pulling from busybox
511136ea3c5a: Pull complete
df7546f9f060: Pull complete
e433a6c5b276: Downloading 925.3 kB/2.646 MB 29s
e72ac664f4f0: Download complete
e433a6c5b276: Pull complete
e72ac664f4f0: Pull complete
Status: Downloaded newer image for busybox:latest
/ # ls
bin      etc      lib      linuxrc  mnt      proc     run      sys      usr
dev      home     lib64    media    opt      root     sbin     tmp      var
/ #
```

11. Now if you run docker ps command you can view the created containers.

docker ps –a

```
docker@boot2docker:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS                          PORTS               NAMES
936af4565f49        busybox:latest      "/bin/sh"           52 seconds ago
Up 52 seconds                                           grave_bell
fc590d3547c1        busybox:latest      "/bin/sh"           4 minutes ago
Exited (0) About a minute ago                           stupefied_wilson
f04580c008dd        busybox:latest      "/bin/bash"         6 minutes ago
                                                        trusting_pare
3c0fa986f4ee        busybox:latest      "/bin/bash"         7 minutes ago
                                                        loving_ptolemy
docker@boot2docker:~$
```

Installing Docker on Ubuntu:

We will be installing Docker on Ubuntu 14.04 (LTS) (64 bit) server. Follow the steps give below to install and configure Docker on Ubuntu server.

Note: throughout this book Ubuntu server is used for most of the Docker demonstration. So we recommend you to use Ubuntu workstation to try out all the examples.

1. To install the latest Ubuntu package (may not be the latest Docker release) execute the following commands.

sudo apt-get install -y docker.io

```
root@dockerdemo:~# apt-get install -y docker.io
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
aufs-tools cgroup-lite git git-man liberror-perl
Suggested packages:
btrfs-tools debootstrap lxc rinse git-daemon-run git-daemon-sysvinit git-doc
git-el git-email git-gui gitk gitweb git-arch git-bzr git-cvs git-mediawiki
git-svn
The following NEW packages will be installed:
aufs-tools cgroup-lite docker.io git git-man liberror-perl
Setting up docker.io (1.0.1~dfsg1-0ubuntu1~ubuntu0.14.04.1) ...
Adding group `docker' (GID 111) ...
Done.
docker.io start/running, process 2516
Setting up liberror-perl (0.17-1.1) ...
Setting up git-man (1:1.9.1-1) ...
Setting up git (1:1.9.1-1) ...
Setting up cgroup-lite (1.9) ...
cgroup-lite start/running
Processing triggers for libc-bin (2.19-0ubuntu6) ...
Processing triggers for ureadahead (0.100.0-16) ...
```

sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker

```
root@dockerdemo:~# sudo ln -sf /usr/bin/docker.io /usr/local/bin/docker
```

sudo sed -i '$acomplete -F _docker docker' \ /etc/bash_completion.d/docker.io

```
root@dockerdemo:~# sudo sed -i '$acomplete -F _docker docker' /etc/bash_completion.d/docker.io
```

2. To verify that everything has worked as expected, run the following command, which should download the Ubuntu image, and then start bash in a container.

sudo docker run -i -t ubuntu /bin/bash

```
root@dockerdemo:~# sudo docker run -i -t ubuntu /bin/bash
Unable to find image 'ubuntu' locally
Pulling repository ubuntu
826544226fdc: Download complete
511136ea3c5a: Download complete
b3553b91f79f: Download complete
ca63a3899a99: Download complete
ff01d67c9471: Download complete
7428bd008763: Download complete
c7c7108e0ad8: Download complete
root@1a2ff1406d35:/#
```

As you can see in the above screenshot, Docker downloaded Ubuntu image from the Docker public registry (Docker hub) and started bash in to the container

1a2ff1406d35

3. Type exit to exit from the container.

```
exit
```

```
root@c83cd43eb85f:/# exit
exit
root@dockerdemo:~#
```

4. You can check the version of Docker components using the following command.

```
sudo docker version
```

```
root@dockerdemo:~# docker version
Client version: 1.0.1
Client API version: 1.12
Go version (client): go1.2.1
Git commit (client): 990021a
Server version: 1.0.1
Server API version: 1.12
Go version (server): go1.2.1
Git commit (server): 990021a
```

The repository installation will not have the latest release of docker. To have the latest version of docker, you need to install it from the source. If you want to try out the latest version, execute the following curl command. It will download and run the script to install the latest version of docker from its source.

```
curl -sSL https://get.docker.com/ubuntu/ | sudo sh
```

RedHat Linux 7

For installing Docker on RedHat 7 follow the steps given below.

1. Enable the extra channel using the following command.

```
sudo subscription-manager repos --enable=rhel-7-server-extras-rpms
```

2. Install Docker using the following command.

```
sudo yum install docker
```

RedHat 6

You can install Docker on RedHat 6 from the EPEL repository. Enable the EPEL repository on your server and follow the instructions below to install Docker.

1. Install the package using yum

```
sudo yum install docker
```

2. If you have a version of Docker already installed on your system, execute the following command to update Docker to its latest version.

```
sudo yum install docker
```

centOS 7

Docker can be installed on centos7 without adding an extra repository. Execute the following command to install Docker.

```
sudo yum install docker
```

CentOS 6

Centos 6 needs EPEL repository to be enabled for installed Docker. Execute the following command after enabling EPEL repository on your system.

```
sudo yum install docker
```

Note: Even though all the tutorials in this book are based on Ubuntu Docker host, it will also work on other Linux distros with Docker installed.

Now we have a working Docker application in our server. Before getting in to more Docker operations, let's have a look at Docker hub.

Docker Hub

In this section we will learn the basics of Docker hub so that we can start working on creating containers.

Docker hub is a repository for uploading and downloading Docker images. Using Docker hub you can collaborate with other developers by sharing the Docker images.

At Docker hub you can search and use images created by other users. The images created by you can be pushed to the Docker hub if you are registered user. Creating a Docker hub account is free.

You can create a Docker hub account in two ways. Using the website and through a command line.

Follow the steps given below to create an account and authenticate your Docker host against the Docker hub.

1. To create an account using the website, use the following link and signup with your credentials. An email will be sent to your email account for activating the account. The location of below link may change so you may have to search for this.

https://hub.Docker.com/account/signup/

2. To create an account from the command line, execute the following command from the server where we have installed Docker.

sudo docker login

```
root@dockerdemo: # sudo docker login
Username (bibinwilson): hcldevops
Password:
Email (bibin.w@hcl.com): bibin.wilson@yahoo.com
Account created. Please use the confirmation link we sent to your
e-mail to activate it.
```

3. Activate your account using the confirmation email sent to your email account.

So far we have set up the Docker host and created a Docker hub account.

Next we will learn how to create Docker containers.

Launching Docker containers:

In this section we will learn how to launch new containers using Docker.

We tested the Docker installation by launching a container without any application. The main objective of Docker is to run applications inside a container. The command used to run a container is “Docker run”. It takes various parameters like image, commands etc.

Let’s try to create a container which echoes out “hello world”. Execute the following command to create a hello world container.

sudo docker run ubuntu:14.04 /bin/echo 'Hello world'

```
root@dockerdemo: # sudo docker run ubuntu:14.04 /bin/echo 'Hello
world' Hello world
```

Let's look at each step the above Docker command has executed:

1. **Docker run**: Docker binary along with run command tells the Docker daemon to run a container.
2. **Ubuntu: 14.04:** This is the image name from which the container should be created. Docker will try to find this image locally. If it is not present locally, it will pull the image from the Docker hub.
3. **/bin/echo 'hello world':** This is the command we specified to run inside the container. This command got executed once the container is created. The result is shown in the screenshot above.

After /bin/echo command, the container had no commands to be executed on it. So it stopped. If you want to install or configure anything on a container, it should be created in an interactive mode. In the next section we will learn how to create an interactive container.

Creating an interactive container:

To create an interactive container an "-i" flag has to be used with the Docker run command.

Execute the following command to create a container in interactive mode.

sudo docker run -t -i ubuntu:14.04 /bin/bash

```
root@dockerdemo:~# sudo docker run -t -i ubuntu:14.04 /bin/bash
root@0fe8efdbe8df:/#
```

The above command will create a container in interactive mode as you can see in the image above. You can run all Ubuntu specific command in the container now.

Let's understand each option in the above command.

1. "-t" flag assigns a terminal session for the container
2. "-i" assigns an interactive session for the container by getting the STDIN of the container.
3. Ubuntu: 14.04 is the image name.

4. /bin/bash is the command to be executed once the container starts. It opens a bash shell for the interactive session.

Now, try to execute the following basic Linux commands on our interactive container to list the current working directory and the contents inside the directory.

pwd

ls

```
root@0fe8efdbe8df:/# pwd
/
root@0fe8efdbe8df:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@0fe8efdbe8df:/#
```

As you can see in the image above, it shows the present working directory as root and lists all the directories inside root. You can install applications and create directories in the container as you can do with any normal Ubuntu machine.

To exit out of the container, you can use the exit command. This will stop the execution of /bin/bash command inside the container and stops the container.

exit

```
root@0fe8efdbe8df:/# exit
exit
root@dockerdemo:~#
```

The interactive container stops when you exit the container. This is not helpful for running applications, as you want that the machine should keep running.

To have a container in running mode, we have to demonize it. In the next section we will look upon how to demonize a container.

Creating a daemonized container:

A daemonized container can be created using a “-d” flag. This is how we will create most of our Docker applications. The following command will create a container in daemonized mode.

sudo docker run -d ubuntu:14.04 /bin/sh –c “while true; do echo hello world”

```
root@dockerdemo: # sudo docker run -d ubuntu:14.04 /bin/sh -c "while
true; do echo hello world;
a109c69e1d88b73448e473b2eae1b3959db5066de7fd6662254fa1536e796705
root@dockerdemo: #
```

This time we did not use “-t” and “-i” flags because we used the “-d” flag which will run the container in the background. Also, we added the following command with a shell script to the Docker command.

bin/sh -c "while true; do echo hello world; sleep 1; done"

In the daemonized container, “hello world” will be echoed out by the infinite loop in the shell script. You can see everything happening inside the container using the container id. The long id returned right after executing the Docker command is the container id.

a109c69e1d88b73448e473b2eae1b3959db5066de7fd6662254f

The long string above denotes the unique container id. This id can be used to see what’s happening in the container. To make sure the container is running, execute the following Docker command.

docker ps

```
root@dockerdemo:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
a109c69e1d88        ubuntu:14.04        /bin/sh -c 'while tr   4 minutes ago       Up 4 minutes
```

The “Docker ps” command will request the Docker daemon to return all the running containers. As you can see in the output, it shows the short container id, command running inside the container and other information associated with the container.

As we did not assign any name to the container, the Docker daemon automatically assigns a random name to the container. In our case it is hungry_goodall. The name might be different when you run the same command.

Note: You can explicitely specify the name of the contianer. We will look in to it in the subsequent chapters.

We added an echo command in infinite loop to run inside the container. You can see the output of the command running inside the container by checking the container logs using its unique id or name.

Execute the following command to see what is happening inside the container.

sudo docker logs hungry_goodall

```
root@dockerdemo:~# sudo docker logs hungry_goodall
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
hello world
```

As you can see the from the log output, the container is executing the “hello world” command in an infinite loop. The “logs” command will ask the Docker daemon to look inside the container and get the standard output of the container.

Now we have a container with specified command running on it. To stop the running container, execute the following command.

sudo docker stop hungry_goodall

```
root@dockerdemo:~# sudo docker stop hungry_goodall
hungry_goodall
root@dockerdemo:~#
```

The Docker stop command along with the container name will gently stop the running container and it returns the name of the container it stopped. Now if you run the Docker ps command you will see no running container.

sudo docker ps

```
root@dockerdemo:~# docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS   PORTS   NAMES
root@dockerdemo:~#
```

We have successfully stopped our hungry_goodall container.

Connecting remote docker host using docker client

By default docker installation will set up the docker server within the host using the UNIX socket unix:///var/run/docker.sock. You can also set up a docker server to accept connections from a remote client. Docker remote client uses REST

API’s to execute commands and to retrieve information about containers. In this section we will learn how to set up the docker client to run commands on a remote docker server.

Follow the instructions given below to configure a Docker client for docker server remote execution.

Note: The client machine should have docker installed to run in client mode

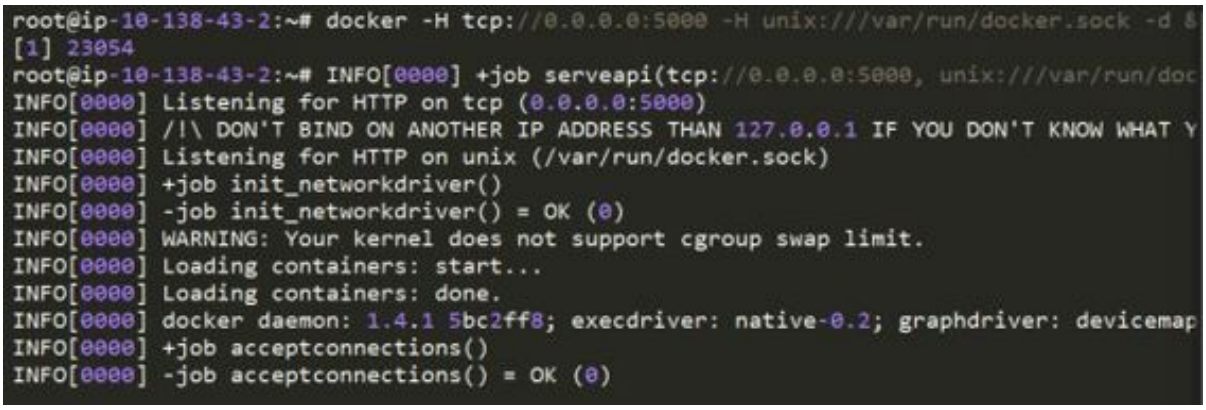
On Docker Server:

1. Stop the docker service using the following command.

```
sudo service docker.io stop
```

2. Start the docker service on port 5000 and on Unix socker docker.sock by executing the following command.

```
docker -H tcp://0.0.0.0:5000 -H unix:///var/run/docker.sock -d &
```



The above command will accept connections from remote docker clients on port 5000 as well as the client inside the host using docker.sock Unix socket.

Note: The port mentioned in the above command can be any tcp port. “0.0.0.0” means, the docker server accepts connections from all incoming docker client connections. It’s not a good practice to open all connections. You can mention a specific ip address instead of “0.0.0.0.”, so that docker server only accepts connections from that particular ip address.

From Docker client:

3. Now from the host which runs the acts as the client, execute the following command to get the list of running containers from the remote docker server. Replace the ip (10.0.0.4) with the ip of the host running docker server.

sudo docker -H 10.0.0.4:5000 ps

```
root@dockerclient:~# docker -H 10.0.0.4:5000 ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
166adf25b6ef        ubuntu:latest       "/bin/bash"         2 minutes ago
Up 2 minutes                            remote-container
root@devopscube:~# nano /etc/hostname
```

You can run all docker commands on the remote server in the same ways you executed “ps” command.

4. Now let’s try creating an interactive container named test-container from the remote client. Execute the following docker run command to create a container on the remote docker server.

docker -H <host-ip>:5000 run -it --name test-container ubuntu

```
root@devopscube:~# docker -H 54.151.211.143:5000 run -it --name test-container ubuntu
root@16b58de435a0:/# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
root@16b58de435a0:/#
```

The equivalent REST requests and actions are shown below.

```
INFO[3041] POST /v1.16/containers/create?name=test-container
INFO[3041] +job create(test-container)
INFO[3041] +job log(create, 16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc
INFO[3041] -job log(create, 16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc
INFO[3041] -job create(test-container) = OK (0)
INFO[3041] POST /v1.16/containers/16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b
INFO[3041] +job container_inspect(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b
INFO[3041] -job container_inspect(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b
INFO[3041] +job attach(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc7f71f
INFO[3042] POST /v1.16/containers/16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b
INFO[3042] +job start(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc7f71f)
INFO[3042] +job allocate_interface(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166
INFO[3042] -job allocate_interface(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166
INFO[3042] +job log(start, 16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc7
INFO[3042] -job log(start, 16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc7
INFO[3042] -job start(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc7f71f)
INFO[3042] POST /v1.16/containers/16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b
INFO[3042] +job resize(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc7f71f
INFO[3042] -job resize(16b58de435a02cf0d887b4b15417d867f9d3842ebb3e16fbb166b9f97bc7f71f
```

Docker server access over https

We have learned how to communicate the docker daemon remotely over http. You can also communicate to docker daemon securely over https by TLS using tlsverify flag and tlscscert flag pointing to a trusted CA certificate.

Note: If you enable TLS, in daemon mode docker will allow connections only authenticated by CA.

To set up a secure access, follow the steps given below.

1. Initialize the CA serial file

echo 01 > ca.srl

2. Generate a ca public and private key files.

openssl genrsa -des3 -out ca-key.pem 2048

```
root@dockerdemo:~/https# openssl genrsa -des3 -out ca-key.pem 2048
Generating RSA private key, 2048 bit long modulus
..................+++
.....................................................................+++
e is 65537 (0x10001)
Enter pass phrase for ca-key.pem:
Verifying - Enter pass phrase for ca-key.pem:
root@dockerdemo:~/https#
```

openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem

```
root@devopscube:~/https# openssl req -new -x509 -days 365 -key ca-key.pem -out ca.pem
Enter pass phrase for ca-key.pem:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:IN
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
root@devopscube:~/https#
```

3. Now, create a server key and certificate signing request (CSR).

openssl genrsa -des3 -out server-key.pem 2048

```
root@dockerdemo:~/https# openssl genrsa -des3 -out server-key.pem 2048
Generating RSA private key, 2048 bit long modulus
..............................+++
..............................+++
e is 65537 (0x10001)
Enter pass phrase for server-key.pem:
Verifying - Enter pass phrase for server-key.pem:
root@dockerdemo:~/https#
```

openssl req -subj '/CN=<hostname here>' -new -key server-key.pem -out server.csr

```
root@dockerdemo:~/https# openssl req -subj '/CN=dockerdemo' -new -key server-key.pem -c
Enter pass phrase for server-key.pem:
root@dockerdemo:~/https#
```

4. Sign the key with CA key.

openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey ca-key.pem -out server-cert.pem

```
root@dockerdemo:~/https# openssl x509 -req -days 365 -in server.csr -CA ca.pem -CAkey
Signature ok
subject=/CN=dockerdemo
Getting CA Private Key
Enter pass phrase for ca-key.pem:
root@dockerdemo:~/https#
```

5. For client to authenticate the server, create relevant client key and signing requests.

openssl genrsa -des3 -out key.pem 2048

```
root@dockerdemo:~/https# openssl genrsa -des3 -out key.pem 2048
Generating RSA private key, 2048 bit long modulus
.............+++
.........................+++
e is 65537 (0x10001)
Enter pass phrase for key.pem:
Verifying - Enter pass phrase for key.pem:
root@dockerdemo:~/https#
```

openssl req -subj '/CN=<hostname here>' -new -key key.pem -out client.csr

```
root@dockerdemo:~/https# openssl req -subj '/CN=dockerdemo' -new -key key.pem -out clie
Enter pass phrase for key.pem:
root@dockerdemo:~/https#
```

6. Now, create an extension config file to make the key suitable for client authentication.

echo extendedKeyUsage = clientAuth > extfile.cnf

```
root@dockerdemo:~/https# echo extendedKeyUsage = clientAuth > extfile.cnf
root@dockerdemo:~/https#
```

7. Sign the key

openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey ca-key.pem -out cert.pem -extfile extfile.cnf

```
root@dockerdemo:~/https# openssl x509 -req -days 365 -in client.csr -CA ca.pem -CAkey
Signature ok
subject=/CN=dockerdemo
Getting CA Private Key
Enter pass phrase for ca-key.pem:
root@dockerdemo:~/https#
```

8. From the server and client keys, remove the passphrase using the following commands.

openssl rsa -in server-key.pem -out server-key.pem

```
root@dockerdemo:~/https# openssl rsa -in server-key.pem -out server-key.pem
Enter pass phrase for server-key.pem:
writing RSA key
root@dockerdemo:~/https#
```

openssl rsa -in key.pem -out key.pem

```
root@devopscube:~/https# openssl rsa -in key.pem -out key.pem
Enter pass phrase for key.pem:
writing RSA key
root@devopscube:~/https#
```

9. Now we have all the server, client certificate and the keys for TLS authentication. Stop the docker service and start it using the following command with all the necessary keys.

sudo service docker.io stop

docker -d --tlsverify --tlscacert=ca.pem --tlscert=server-cert.pem --tlskey=server-key.pem -H=0.0.0.0:2376

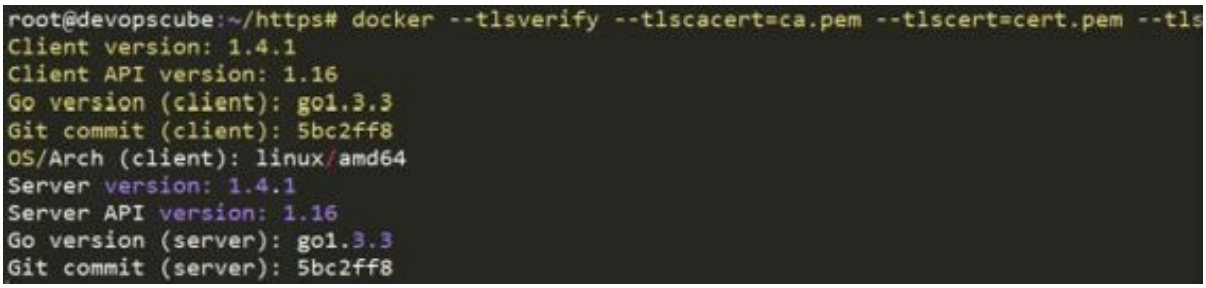
Note: If you running docker on TLS, it should run only on port 2376

```
root@dockerdemo:~/https# docker -d --tlsverify --tlscacert=ca.pem --tlscert=server-cert
INFO[0000] +job serveapi(tcp://0.0.0.0:2376)
INFO[0000] Listening for HTTP on tcp (0.0.0.0:2376)
INFO[0000] +job init_networkdriver()
INFO[0000] -job init_networkdriver() = OK (0)
INFO[0000] WARNING: Your kernel does not support cgroup swap limit.
INFO[0000] Loading containers: start.
INFO[0000] Loading containers: done.
INFO[0000] docker daemon: 1.4.1 5bc2ff8; execdriver: native-0.2; graphdriver: devicemap
INFO[0000] +job acceptconnections()
INFO[0000] -job acceptconnections() = OK (0)
```

10. Now, from client, execute the following “docker version” command with docker server’s ip or DNS name.

Note: make sure you copy the relevant client key files to authenticate against the docker server.

docker --tlsverify --tlscacert=ca.pem --tlscert=cert.pem --tlskey=key.pem -H=172.31.1.21:2376 version



Same way, you can run all the docker commands by connecting the docker server over https.

So far we have learnt,

1. *How to create a container in interactive and daemonized mode.*
2. *To list the running containers*
3. *To check the actions running inside a container using logs*
4. *To stop the running container gracefully.*
5. *To access docker server from a remote client*
6. *To access docker server securely over https.*

In the next chapter we will look at more advanced container operations.

4
Working with containers

In this chapter we will look at some advanced container operations.

In last chapter we have learnt about the following Docker commands,

1. *Docker ps* – for listing all the running containers
2. *Docker logs* – return the STDOUT of a running container
3. *Docker stop* – stops the running container.

The Docker command is associated with several arguments and flags. The standard use of a Docker command is shown below.

Usage: [sudo] docker [flags] [command] [arguments]

For example,

sudo docker run -i –t centos /bin/sh

Let’s start by finding the version of Docker using the following Docker command.

sudo docker version

The above command will give you all the information about Docker including API version, go version etc.

```
root@dockerdemo:~# docker version
Client version: 1.0.1
Client API version: 1.12
Go version (client): go1.2.1
Git commit (client): 990021a
Server version: 1.0.1
Server API version: 1.12
Go version (server): go1.2.1
Git commit (server): 990021a
```

Docker commands:

There are many commands associated with Docker client. To list all the commands, run the following command on the terminal.

sudo docker

It will list all the commands which can be used with Docker for manipulating containers.

```
ubuntu@node2:~$ sudo docker
Commands:
    attach    Attach to a running container
    build     Build an image from a Dockerfile
    commit    Create a new image from a container's changes
    cp        Copy files/folders from the containers filesystem to the host path
    diff      Inspect changes on a container's filesystem
    events    Get real time events from the server
    export    Stream the contents of a container as a tar archive
    history   Show the history of an image
    images    List images
    import    Create a new filesystem image from the contents of a tarball
    info      Display system-wide information
    inspect   Return low-level information on a container
    kill      Kill a running container
    load      Load an image from a tar archive
    login     Register or Login to the docker registry server
    logs      Fetch the logs of a container
    port      Lookup the public-facing port which is NAT-ed to PRIVATE_PORT
    pause     Pause all processes within a container
    ps        List containers
    pull      Pull an image or a repository from the docker registry server
    push      Push an image or a repository to the docker registry server
    restart   Restart a running container
    rm        Remove one or more containers
```

Command usage:

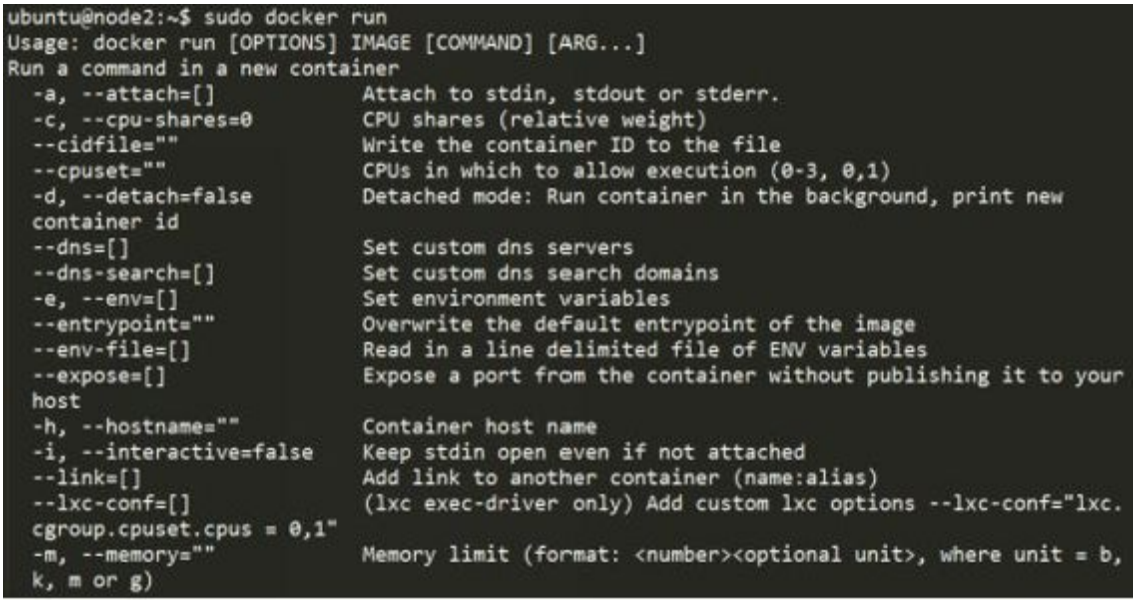
Each Docker command has its own set of flags and arguments. To view the usage of a particular Docker command, run Docker with specific command.

Syntax: Docker [command name]

For example,

sudo docker run

It will return the usage of run command and its associated flags as shown in the image below.



So far in this chapter we have learnt how to use Docker commands and getting help for each command. In next section, we will learn how to run a basic python application inside a container.

Python web application container:

The containers we created before just ran a shell script in it. In this section we will learn how to run a python flask application inside a container.

We will use a preconfigured image (training/webapp) from Docker hub with python flask application configured in it. Let’s start with the Docker run command.

sudo docker run -d -P training/webapp python app.py

```
root@dockerdemo:~# sudo docker run -d -P training/webapp python app.py
Unable to find image 'training/webapp' locally
WARNING: The Auth config file is empty
Pulling repository training/webapp
31fa814ba25a: Download complete
511136ea3c5a: Download complete
f10ebce2c0e1: Download complete
82cdea7ab5b5: Download complete
5dbd9cb5a02f: Download complete
74fe38d11401: Download complete
64523f641a05: Download complete
0e2afc9aad6e: Download complete
e8fc7643ceb1: Download complete
733b0e3dbcee: Download complete
a1feb043c441: Download complete
e12923494f6a: Download complete
a15f98c46748: Download complete
7e4b587f12134f49eb4d21e899d3557168fcddebc906c924f06451fa28623b6d
```

So what happened when we ran the above command? Let's break it down

1. Docker run : command for creating a container
2. "–d": we have already seen the usage of this flag. It demonizes the container.
3. –P: This flag is used for mapping the host port to the container port for accessing the container application. As we did not specify any port numbers, Docker will map the host and container with random ports.
4. training/webapp: This is the image name which contains the python flask application. Docker downloads this image from Docker hub.
5. Python app.py: this is the command which will be executed inside the container once it is created. It starts the python application inside the container.

Now let's see if our python container is running using the following command.

sudo docker ps -l

We have seen the use of Docker ps command before. The "-l" flag will list the last created and running container.

```
root@dockerdemo:~# sudo docker ps -l
CONTAINER ID        IMAGE                     COMMAND             CREATED
STATUS              PORTS                      NAMES
7e4b587f1213        training/webapp:latest    python app.py       9 minutes ago
Up 9 minutes        0.0.0.0:49153->5000/tcp   tender_kowalevski
```

The output from the above image shows that the host port 49153 is mapped on to

port 5000 in the container.

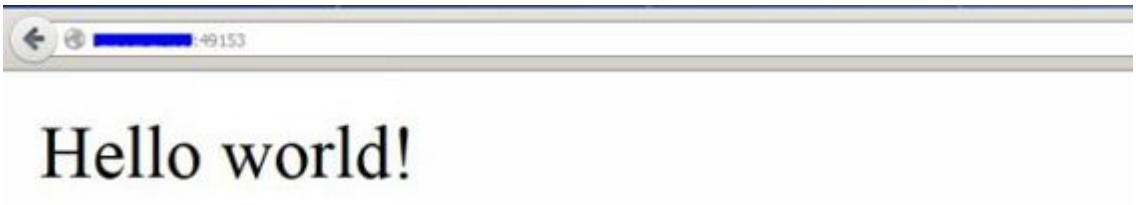
PORTS
0.0.0.0:49155->5000/tcp

The port mapping was done randomly. When the port is not explicitly mentioned in the command, Docker assigns the host port within 49000 and 49900.

Let's try accessing the python application on port 49153 from the browser.

Note: The host port may be different for your application. So get the IP Address from the "Docker ps" output and try it on the browser.

Example: http://<Docker-host-ip-here>:49154



You will see a sample hello world application in the browser.

The port mapping can also be configured manually. Let's create the same python application container with manually configured ports. Run the following command to launch a container to map host port 5000 to container port 5000.

sudo docker run -d -p 5000:5000 training/webapp python app.py

```
root@dockerdemo:~# sudo docker run -d -p 5000:5000 training/webapp python app.py
bdda01f8b9cf1753e25102ce41acbdee2a8961e714f4b659713d09d984513cf5
root@dockerdemo:~#
```

Now if you access the python application on port 5000 from your browser, you will see the same application. This is the advantage of not having one to one port mapping for Docker.

Thus you can have multiple instances of your application running on different ports.

The "Docker ps" output is little cluttered, so if you want to know which host port your container is mapped to , you can use the "Docker port" command along with the container name or id to view the port information.

For example,

sudo docker port tender_kowalevski 5000

```
root@dockerdemo:~# sudo docker port  kickass_fermat 5000
0.0.0.0:49153
root@dockerdemo:~#
```

Now that we have a working python application, we can perform the following operations to get more information about the container.

1. Docker logs – to view the logs of running container.
2. Docker top – to view all the processes running inside the container
3. Docker inspect – to view complete information like networking, name, id etc.

Python container logs:

Run the following command to view the logs of your running python container.

sudo docker logs -f tender_kowalevski

It will list all the logs of actions happening inside the container.

```
root@dockerdemo:~# sudo docker logs -f tender_kowalevski
 * Running on http://0.0.0.0:5000/
59.161.70.47 - - [17/Sep/2014 06:56:43] "GET / HTTP/1.1" 200 -
59.161.70.47 - - [17/Sep/2014 06:56:43] "GET /favicon.ico HTTP/1.1" 404 -
59.161.70.47 - - [17/Sep/2014 06:56:44] "GET /favicon.ico HTTP/1.1" 404 -
59.161.70.47 - - [17/Sep/2014 07:03:18] "GET / HTTP/1.1" 200 -
```

Python container processes:

Run the following command to view all the processes running inside the python container.

sudo docker top tender_kowalevski

It will list all the processes running inside our python container.

```
root@dockerdemo:~# sudo docker top tender_kowalevski
UID                 PID                 PPID                C                   STIME
root                5799                2516                0                   06:37
```

We have only python.py command process running inside our container.

Inspecting python container:

By inspecting a container, you can view the entire network and other configuration information about the container. Execute the following command to inspect the python container.

sudo docker inspect tender_kowalevski

This command will return all the information about the container in JSON format. The sample output is shown below.

For more granular output, you can request for specific information about the container. For example,

sudo docker inspect -f '{{ .NetworkSettings.IPAddress }}' tender_kowalevski

The above command will output the ip address of your container.

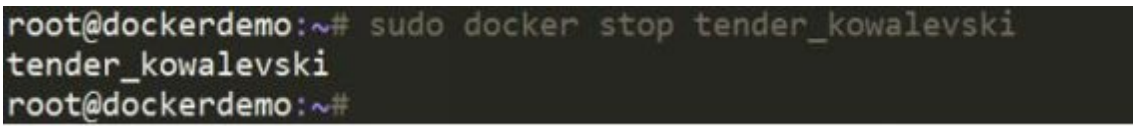
Stopping python container:

You can stop the python container using “docker stop” command.

Syntax: docker stop [container name]

Execute the following command with your container name to stop the container.

sudo docker stop tender_kowalevski



To check if the container has stopped, run the “docker ps” command.

Restarting the python container:

You can restart the container using “docker restart” command.

Syntax: docker restart [container name]

Execute the following command to restart the container.

sudo docker start tender_kowalevski

```
root@dockerdemo:~# sudo docker start tender_kowalevski
tender_kowalevski
root@dockerdemo:~#
```

Now if you execute "docker ps -l" command you can see the started container or you can view the application in the browser.

Removing python container:

You can remove the python container using “docker rm” command. You need to stop the container before executing the remove command. Let’s see what happens if you try to remove the container without stopping it.

sudo docker rm tender_kowalevski

```
root@dockerdemo:~# sudo docker rm tender_kowalevski
Error response from daemon: Impossible to remove a running container, please stop it first or
2014/09/17 07:23:28 Error: failed to remove one or more containers
root@dockerdemo:~#
```

As you can see, it shows an error message for stopping the container. This is useful because it avoids accidental deletion of containers.

Now we will stop the container and try to remove it using the following commands.

sudo docker stop tender_kowalevski

sudo docker rm tender_kowalevski

```
root@dockerdemo:~# sudo docker stop tender_kowalevski
tender_kowalevski
root@dockerdemo:~# sudo docker rm tender_kowalevski
tender_kowalevski
root@dockerdemo:~#
```

The python container is removed now. Once the container is removed, you cannot restart it. You can only recreate a container from the image.

If you want to remove a container without stopping the container, you can use the force flag “-f” with the “Docker rm” command. It is not advisable to force

remove the containers.

docker rm -f tender_kowalevski

```
root@dockerdemo:~# docker rm -f tender_kowalevski
tender_kowalevski
root@dockerdemo:~#
```

If you want to stop and remove all the containers in your Docker host, you can use the following in Linux systems.

docker stop $(Docker ps -a -q)

docker rm $(Docker ps -a -q)

```
root@dockerdemo:~# docker stop $(docker ps -a -q)
bdda01f8b9cf
0c2d9c1f8c55
8dacd254b757
root@dockerdemo:~# docker rm -f  $(docker ps -a -q)
bdda01f8b9cf
0c2d9c1f8c55
8dacd254b757
root@dockerdemo:~#
```

In this chapter we have learnt the following,

1. How to create a container in interactive mode and daemonized mode.
2. How to create a python flask application container.
3. How to get information's like logs, networking settings etc. About the python container.
4. How to stop, start and remove a container.

5
Docker Images

In the last chapter we showed how to use containers and deployed a python application on a container. In this chapter we will learn to work with images.

As you know, Docker images are the basic building blocks of a container. In previous chapter we used preconfigured images from Docker hub for creating containers (Ubuntu and training/webapp).

So far we learnt the following about images,

1. Images are created and stored in layered fashion.
2. All the images downloaded from the Docker hub reside in the Docker host.
3. If an image specified in the Docker run command is not present in the Docker host, by default, the Docker daemon will download the image from the Docker public registry (Docker hub).
4. Container is a writable layer on top on an image.

There are more things you can do with the images. In this section we will look in to the following.

1. How to manage and work with images locally on the Docker host.
2. How to create a basic Docker image.
3. How to upload a modified image to the Docker registry (Docker hub).

Listing Docker images:

You can list all the available images on your Docker host. Run the following command to list all the images on your Docker host.

sudo docker images

Docker host will not have any images by default. All the images shown in the image above were downloaded from Docker hub when we created the sample containers.

Let's have a look at the important information's about the container.

1. **REPOSITORY:** This denotes, from which Docker hub repository the image has been downloaded. A repository will have different versions of an image.
2. **TAG:** The different versions of an image are identified by a tag. In the example shown above, we have an Ubuntu repository with two different tags. Latest and 14.04.
3. **IMAGE ID:** This denotes the unique id of the image.
4. **CREATED:** This provides information on the date, when the image has been created by the user in the Docker hub repository.
5. **VIRTUAL SIZE:** This refers to the virtual size of the image on the Docker host. As we learnt before, Docker image works on copy on write mechanism, so the same images are never duplicated, it will only be referenced by the updated layers. So, a repository might have variants of images e.g. Ubuntu 12.04, 13.10. 14.04 etc.

Specifying the image variant as a tag is good practice while working with

containers because you will know which version of the image you are using for launching containers.

If you want to run a container with Ubuntu 14.04 version, the Docker run command will look like the following.

```
sudo docker run -t -i ubuntu:14.04 /bin/bash
```

Instead, if you mention only the image name without the tag, Docker will download the latest version of the image available in the Docker hub.

While working with images you need to keep the following in mind,

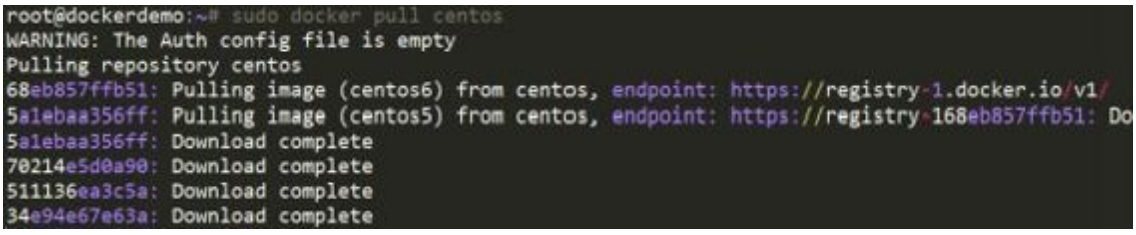
1. *Always use image tags to know exactly which version of the image you are using.*
2. *Always download and use official and trusted images from Docker hub because Docker hub is a public registry where anyone can upload images which may not be as per your organizations policies and needs.*

Downloading an image:

When creating a container, if the image is not available in the Docker host, Docker downloads it from the Docker hub. It is a little time consuming process. To avoid this you can download the images from Docker hub to the host and use it for creating a container.

So if you want to launch containers based on centos container, you can download the centos image first. Execute the following command to pull a centos image from Docker hub.

```
sudo docker pull centos
```



The above image shows Docker downloading a centos image consisting of various layers.

Once the image is downloaded you can create a container in seconds. Now let’s launch an interactive container from the downloaded centos image using the following Docker run command.

sudo docker run -t -i centos /bin/bash

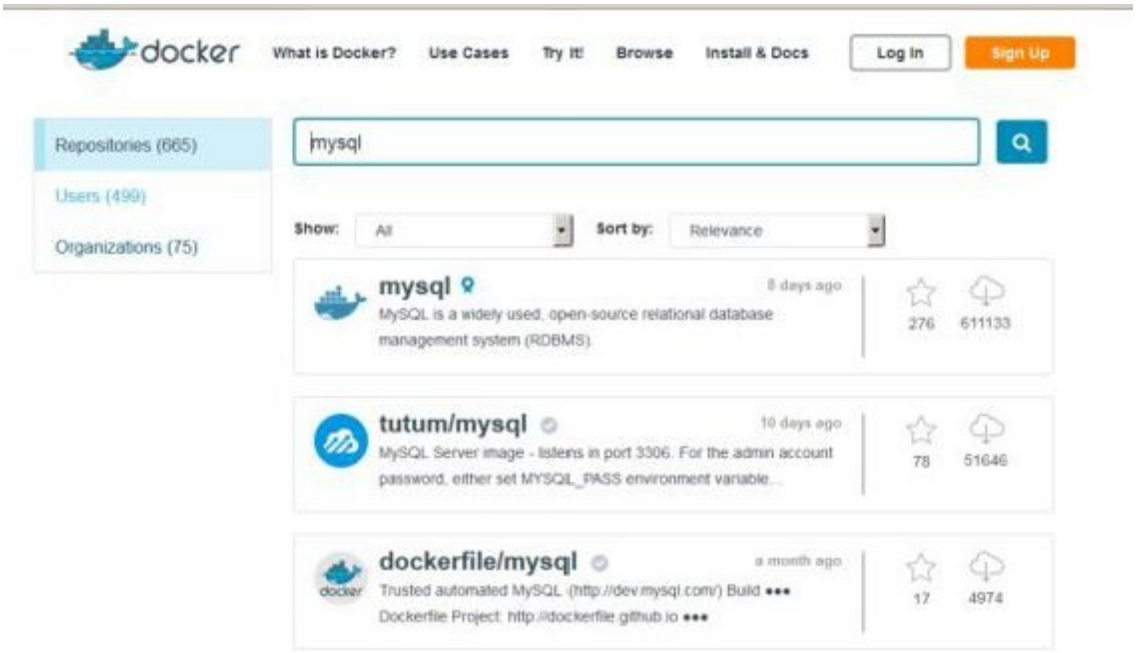
```
root@dockerdemo:~#  sudo docker run -t -i centos /bin/bash
bash-4.2# ls
bin  etc   lib    lost+found  mnt  proc  run   selinux  sys  usr
dev  home  lib64  media       opt  root  sbin  srv      tmp  var
bash-4.2#
```

Searching images:

Docker hub has images created by many users for various purposes. You can search for images with a keyword, for example, rails. Docker hub will return all the images named or tagged with rails. You and do the search from Docker client and Docker hub web UI as well.

Searching images from Docker hub UI:

Visit Docker public registry UI using this link https://registry.hub.Docker.com/ and search for an image. For example if you search for MySQL, It will list all the images named and tagged with MySQL. You can get the name of image from UI and use it with Docker client for creating a container from that image.



Searching images from Docker command line:

Images from Docker hub can be searched from Docker command line. Suppose,

if you want to search an image with Sinatra configured, you can use the “Docker search” command.

Search Syntax: Docker search [image name]

Execute the following command to search all the sinatra images.

sudo docker search sinatra

The search command has returned the images tagged with Sinatra. All the images have a description mentioned by the user the user created it. The stars represent the popularity of the image. More Stars means that the image is trusted by more users.

All the official images on the Docker hub are maintained by stackbery project.

For examples we created in this book we used containers from Ubuntu and training/webapp images. The Ubuntu image is the base image maintained by official Docker Inc. These images are validated and tested. Normally, the base images will have single names e.g. Ubuntu, centos, fedora etc.

Training/webapp image is created by a user in Docker hub. The images created by users will have the usernames in the image. In training/webapp, training is the username.

For our next example we will use the training/sinatra image which appeared at the top of the search. This image has Sinatra configured in it. Let’s try pulling down that image to our Docker host. Run the following command to pull the Sinatra image to our Docker host.

sudo docker pull training/sinatra

```
ubuntu@dockerdemo:~$ sudo docker pull training/sinatra
Pulling repository training/sinatra
3c59e02ddd1a: Pulling image (v2) from training/sinatra, endpoint: https://registry-1.docke
3c59e02ddd1a: Download complete
f0f4ab557f95: Download complete
511136ea3c5a: Download complete
3e76c0a80540: Download complete
be88c4c27e80: Download complete
bfab314f3b76: Download complete
e809f156dc98: Download complete
ce80548340bb: Download complete
79e6bf39f993: Download complete
5e66087f3ffe: Download complete
4d26dd3ebc1c: Download complete
d4010efcfd86: Download complete
99ec81b80c55: Download complete
1fd0d1b3b785: Download complete
08ebafdba908: Download complete
fbc9a83d93d9: Download complete
ubuntu@dockerdemo:~$
```

Now we have the sinatra image in our host. Let's create an interactive Sinatra container from the image using the Docker run command.

sudo docker run -t -i training/sinatra /bin/bash

```
ubuntu@dockerdemo:~$ sudo docker run -t -i training/sinatra /bin/bash
root@c054ad6ec080:/#
```

We learnt how to search for an image, pull it down to the Docker host and launch a container from that image. You can download any other image from the Docker hub and try creating containers from it.

Our own images:

Till now we have used images created by other users. Even though we found these images useful, it might not have all the features we want for our Sinatra application. In this section we will learn how to modify and create our own images.

There are two ways by which we can have our own images,

1. By committing changes to a container created from a preconfigured image
2. Using a Dockerfile to create an image from scratch using instructions specified in the Dockerfile.

Let's have a look at the two approaches.

From preconfigured image:

In this section we will learn how to modify and update a container for creating a new image. For updating an image, we need to have a container running in

interactive mode.

Execute the following command to create an interactive container from training/Sinatra image.

sudo docker run -t -i training/sinatra /bin/bash

```
ubuntu@dockerdemo:~$ sudo docker run -t -i training/sinatra /bin/bash
root@c054ad6ec080:/#
```

As you can see, it has created a unique id (c054ad6ec080) for the container. For committing the changes, we need the unique id or name of the container. Note down the id created for your container .You can also get the container details by running “sudo Docker ps -l” command.

We need to make some changes to container for creating a new image. Let’s install a JSON image in our newly launched container using the following gem command.

gem install JSON

```
root@c054ad6ec080:/# gem install json
Fetching: json-1.8.1.gem (100%)
Building native extensions.  This could take a while...
Successfully installed json-1.8.1
1 gem installed
Installing ri documentation for json-1.8.1...
Installing RDoc documentation for json-1.8.1...
```

Once installed, exit the container by running exit command.

Now we have container with JSON gem installed, which was not available in the downloaded image.

For committing changes to a container, Docker has a command called “Docker commit”.

Syntax: sudo docker commit -m="<your commit message here" -a="<maintainer name here" \ <container id> <docker hub username>/<image name>:<tag>

Execute the following command to commit the changes to our sinatra container.

Note: Replace the argument inside “-a” tag with your name, replace “hcldevops” with your Docker hub username and “c054ad6ec080” with your container id.

*sudo docker commit -m="Added JSON gem" -a="bibin wilson" *

c054ad6ec080 hcldevops/sinatra:v2

```
ubuntu@dockerdemo:~$ sudo docker commit -m="Added json gem" -a="bibin wilson" \
> c054ad6ec080  hcldevops/sinatra:v2
8a2ad0896423d78374b84fd075ecfca08a6d2aaca37fd7def295bd88a7d9d11f
ubuntu@dockerdemo:~$
```

- "-m" flag represents the commit message like we use it in our version control systems like git.
- "-a" represents the maintainer.
- "c054ad6ec080" represents the id of the container to be committed.
- "hcldevops/Sinatra" is the username/imagename
- "v2" is the tag for the image.

We can view the newly created hcldevops/Sinatra image using the "docker images" command. Execute the following command to view your newly created image.

sudo docker images

```
ubuntu@dockerdemo:~$ sudo docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
hcldevops/sinatra   v2                  8a2ad0896423        About a minute ago  451.3 MB
centos              centos5             5a1ebaa356ff        8 days ago          484 MB
centos              centos7             70214e5d0a90        12 days ago         224 MB
centos              latest              70214e5d0a90        12 days ago         224 MB
centos              centos6             68eb857ffb51        12 days ago         212.7 MB
ubuntu              latest              826544226fdc        12 days ago         194.2 MB
training/sinatra    latest              f0f4ab557f95        3 months ago        446.9 MB
training/webapp     latest              31fa814ba25a        3 months ago        278.6 MB
training/sinatra    v2                  3c59e02ddd1a        3 months ago        446.4 MB
```

Docker shows our newly created image at the top with all the image information.

Now let's try to launch a container from out new image.

Syntax: sudo docker run -t -i username/imagename:tag /bin/bash

Execute the following command for creating a container from hcldevops/sinatra image.

Note: replace "hcldevops" with your Docker hub username.

sudo docker run -t -i ouruser/sinatra:v2 /bin/bash

```
ubuntu@dockerdemo:~$ sudo docker run -t -i hcldevops/sinatra:v2 /bin/bash
root@8efe561d829b:/#
```

We have successfully created a container from our newly created image. You can try creating a new image by modifying a container running in interactive mode by following the steps explained above.

Building an image from scratch:

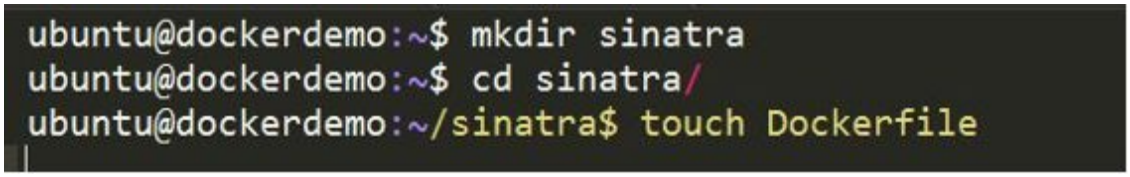
In the last section we learnt how to create an image by modifying a container. If we think of creating a customized image for development team, committing a preconfigured image is cumbersome and not recommended. In this section we will learn how to build Docker images from scratch using Dockerfile for specific development tasks.

Dockerfile:

Dockerfile is a normal text file with instructions for configuring an image. Instructions include tasks such as creating a directory, copying file from host to container etc.

Let's create a sinatra image using a Dockerfile. Create a directory and Dockerfile inside it using the following commands.

```
mkdir sinatra
cd sinatra
touch Dockerfile
```



Every line in a Dockerfile starts with an instruction followed by a statement. Every instruction creates a layer on the image when the image gets built from the Dockerfile.

```
Syntax: INSTRUCTION statement
```

Let's have a look at the Dockerfile for sinatra. Copy the following code snippet on to the Docker file we created.

```
FROM ubuntu:14.04
MAINTAINER bibin wilson <bibin.w@hcl.com>
RUN apt-get update && apt-get install -y ruby ruby-dev
RUN gem install sinatra
```

Let's breakdown the Dockerfile and see what it does.

FROM: It tells the Docker daemon from which base image the new image should be built. In our example, it is Ubuntu: 14.04.

MAINTAINER: This indicates the user maintaining the image.

RUN: This instruction handles all the application installation and other scripts that have to be executed on the image. In our case, we have commands to install ruby development environment and Sinatra application.

Let's try to build the image using the Docker file we created. An image can be built from Docker file using the "Docker build" command. It has the following syntax.

sudo Docker build –t="username/imagename:tag" .

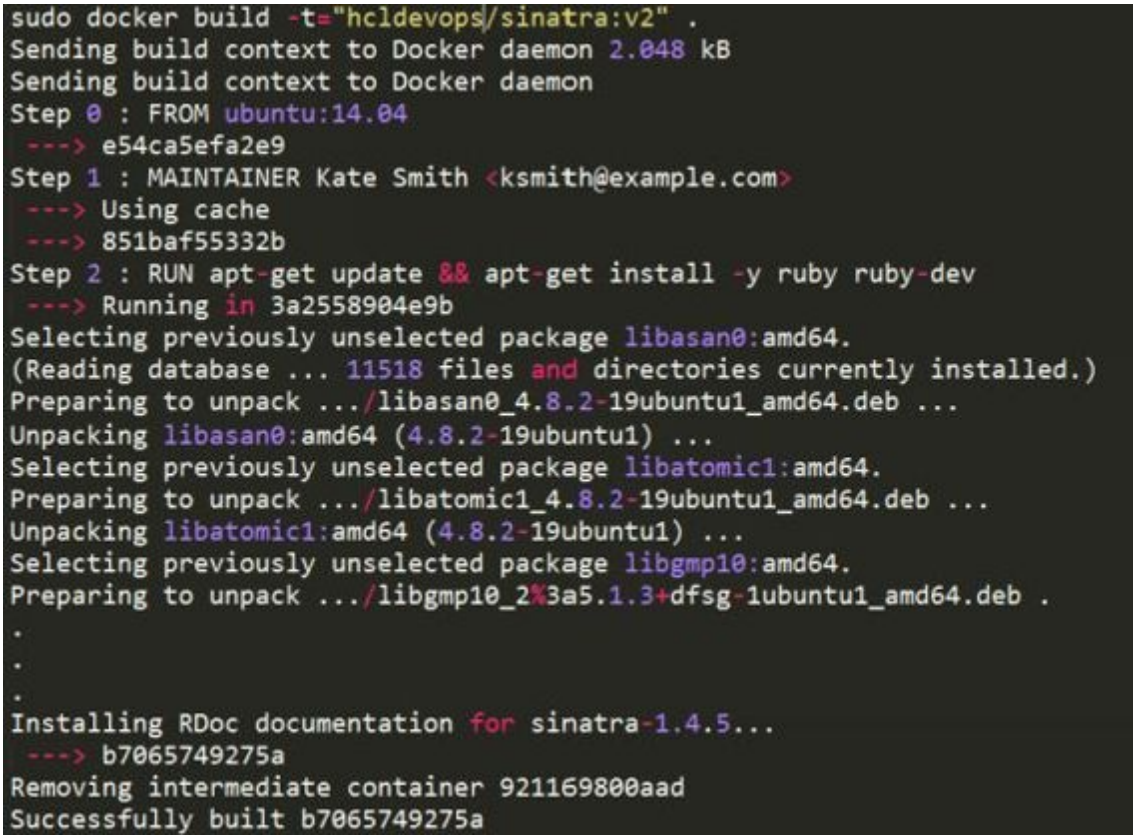
"-t" flag is used for identifying the image that belongs the user.

"." Flag represents that the Docker file is present in the current directory. You can also mention the path to the Docker file if it is not present in the current directory.

Execute the following command to build the sinatra image from the Docker file.

Note: replace "hcldevops" with your username. Also use different tag if you already have one image with the same tag.

sudo docker build -t="hcldevops/sinatra:v2" .



As you can see in the output, Docker client sends a build context to the Docker daemon and in each step it creates an intermediate container while executing commands and creates a layer on top of the image.

At final step it created an image with id b7065749275a and deleted all the intermediate containers. Now we have an image built with id b7065749275a.

Let's launch a container from the new image.

Note: A Docker image can have a maximum of 147 layers.

Execute the following command to create a new Sinatra container.

sudo docker run -t -i hcldevops/sinatra:v2 /bin/bash

```
ubuntu@dockerdemo:~$ sudo docker run -t -i hcldevops/sinatra:v2 /bin/bash
root@e428a356885e:/#
```

Image tagging:

We can tag our new image with a different name. This can be done using the Docker tag command. Let's tag our new image using the following command.

Note: replace the image id and name with your container id and names.

sudo docker tag b7065749275a hcldevops/sinatra:devel

Now that we have tagged our new image, let's view it using the Docker images command.

Execute the following command to view the image named hcldevops/Sinatra

sudo docker images hcldevops/sinatra

```
ubuntu@dockerdemo:~$ sudo docker images hcldevops/sinatra
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
hcldevops/sinatra   v2                  b7065749275a        11 minutes ago      321.3 MB
hcldevops/sinatra   devel               b7065749275a        11 minutes ago      321.3 MB
ubuntu@dockerdemo:~$
```

Uploading the image to Docker Hub:

So we have built a Docker image from scratch and successfully launched a container from it. In order for all developers in your team to get access to the configured image, you need to upload it to the Docker hub (public or private). In this example we will upload it to the public registry. You can push an image to the Docker hub using the Docker push command.

Sysntax: docker push <username>/<imagename>

Execute the following command with your username/imagename to push the image to the Docker hub.

You must be logged in to Docker hub from command line before pushing the image.

sudo docker push bibinwilson/sinatra

We have successfully pushed our image to the Docker hub and it is not publicly accessible. Now you or your team can access the image from any Docker host by specifying the image name. For example:

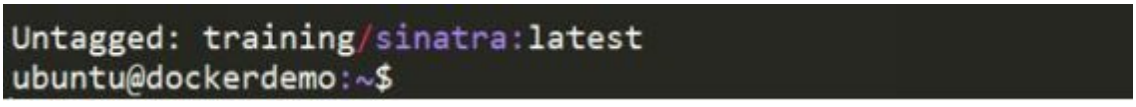
sudo docker pull bibinwilson/sinatra

Removing images from Docker host:

We have learnt how to create an image and push it to the Docker hub. If you think you don't want a particular image in your Docker host, you can remove same like you remove a container. You can use the "Docker rmi" command to remove the image. Let's try to remove our training/Sinatra image from our host using the following command.

Note: before removing an image, make sure all the containers based on that particular image have been stopped or removed.

sudo docker rmi training/sinatra



If you want to remove all the images from the Docker host, you can use a one liner on Linux systems. Execute the following command to remove all the images from the host.

docker rmi $(docker images -q)

In this chapter we have learnt the following,

1. *How to search for Docker images from Docker hub.*
2. *How to pull a Docker images to Docker host.*
3. *How to build an image from existing image and from Docker file.*
4. *How to push a Docker image to the Docker host and*
5. *How to tag Docker images and how to delete an image from the Docker host.*

6
Container linking

In previous chapters we learnt how to connect a service inside the container using host to container mapping. In this chapter we will learn more advanced options for linking containers together.

Container port mapping:

Applications inside a container can be accessed by mapping the host port to the container port. Let's consider the following Docker command.

```
sudo docker run -d -P training/webapp python app.py
```

In the above command, a random host port will be mapped to the containers exposed port 5000 using –P flag. Every container is associated with its own network configuration and IP Addresses.

Let's consider another Docker command.

```
sudo docker run -d -p 5000:5000 training/webapp python app.py
```

In the above command the host port 5000 is mapped to container port 5000. This is a manual assignation using flap –p.

Note: For random port assignation, the flap is "–P" capital letter and for manual port assignation it is "-p" small p.

Let's consider another example,

```
sudo docker run -d -p 127.0.0.1:5000:5000 training/webapp python app.py
```

In the above example, the "-p" flag maps port 5000 on the host to the localhost interface. If you don't apply any interface explicitly, by default, Docker assigns the port specified in "–p" flat to all the interfaces in the host.

Let's say, you want to bind a dynamic port from the local host interface to the container port, then the Docker command will have the following form.

```
sudo docker run -d -p 127.0.0.1::5000 training/webapp
```

```
python app.py
```

Binding UDP ports:

You can also bind UDP ports to your container. Let's see an example for binding the localhost interface with UDP port 5000.

```
sudo docker run -d -p 127.0.0.1:5000:5000/udp training/webapp python app.py
```

The above command will bind the localhost interface UDP port 5000 with container port 5000.

Tip: for configuring multiple ports for a container, you can use the "-p" flag multiple time in the Docker command.

Linking containers together:

For containers to talk to each other, port mapping is not the only way. Docker has an interesting linking feature which allows containers to be linked together and pass information between each other. While linking, certain source container information will be sent to the destination container. This way the destination container can access the required information from the source container.

Naming containers:

Docker needs the names of the container for linking it together. When a container is launched, a random name is assigned to it. Docker has an option to name the containers while launching it. Naming containers while launching is recommended because it has two uses

1. You can remember the name of the container and use it for all the container operations like start, stop, inspect etc.
2. Names are used as a reference for linking container together. For example, linking a webserver and a database container.

A container can be named using the "--name" flag. Let's create a python container with name "web".

Execute the following command for creating a python container named web.

```
sudo docker run -d -P --name web training/webapp python app.py
```

```
ubuntu@dockerdemo:~$ sudo docker run -d -P --name web training/webapp python \
app.py
ae6bcece4532ae493d23032a2a48921aa3f5b79f2d3d8492c0c94cc624cc12da
```

The above command launched a container with name “web”. Run the following command to view the web container.

sudo docker ps –l

```
ubuntu@dockerdemo:~$ sudo docker ps -l
CONTAINER ID        IMAGE                    COMMAND             CREATED
 STATUS              PORTS                     NAMES
ae6bcece4532        training/webapp:latest   python app.py       2 minutes ago
Up 2 minutes        0.0.0.0:49154->5000/tcp   web
ubuntu@dockerdemo:~$
```

Another way to get the name of the container is by using “Docker inspect” command.

Execute the following command to get the name of the container using the container id.

sudo docker inspect -f "{{ .Name }}" ae6bcece4532

```
ubuntu@dockerdemo:~$ sudo docker inspect -f "{{ .Name }}" ae6bcece4532
/web
ubuntu@dockerdemo:~$
```

All the container names should be unique. No two containers can have the same name. If you want to create a container with the same name, you should delete the container running with that name and create a new one.

Let’s take a scenario where you want to have a web container and a database container in the Docker host. By Docker port mapping you can make the web container to talk to the database container. But the efficient way to do this is by using Docker linking feature.

Links are more secure for passing data from the source to the destination. We will look at an example where we will link a web container with a database container.

To link containers together Docker uses the “–link” flag. Execute the following command to create a named database container.

sudo docker run -d --name db training/postgres

```
ubuntu@dockerdemo:~$ sudo docker run -d --name db training/postgres
Unable to find image 'training/postgres' locally
Pulling repository training/postgres
258105bea10d: Pulling image (latest) from training/postgres, endpoint: https://r258105bea10d:
511136ea3c5a: Download complete
35f6dd4dd141: Download complete
7baf0ef6f14a: Download complete
e497c7c1bfbb: Download complete
5cf8fd909c6c: Download complete
8726e050fbc9: Download complete
043c01407567: Download complete
65a89e6a06f8: Download complete
6af9ddfabfd3: Download complete
316f4525806b: Download complete
bfbc096044e3: Download complete
444db2eae2c3: Download complete
e06e512105c3: Download complete
8c8f28d39547441f67aeb369a8a9f8be16ab25b5a63b41d840a6d4d201462421
```

Now we have a running database container named “db”. The container is created from training/postgres image downloaded from Docker hub. Now let’s create a web container with name “web”.

Execute the following command for creating the web container from training/webapp image and link it to the db container using the “—link” flag.

sudo docker run -d -P --name web --link db:db training/webapp python app.py

```
ubuntu@dockerdemo:~$ sudo docker run -d -P --name web --link db:db training/webapp python app.p
e1e4f16edb7269d14c4843dec7e003c1ca0efe5217a6876bfae5f52276292b9b
ubuntu@dockerdemo:~$
```

The link flag has the following syntax,

Syntax: --link name:alias

Name is the name of the container to be linked and “alias” is the alias for the link name.

Now let’s have a look at the launched web and db containers using the docker ps command.

sudo docker ps

```
ubuntu@dockerdemo:~$ sudo docker ps
CONTAINER ID        IMAGE                     COMMAND                CREATED
STATUS              PORTS                     NAMES
e1e4f16edb72        training/webapp:latest    python app.py          About a minute ago
Up About a minute   0.0.0.0:49153->5000/tcp   web
8c8f28d39547        training/postgres:latest  su postgres -c '/usr   16 minutes ago
Up 16 minutes       5432/tcp                  db,web/db
ubuntu@dockerdemo:~$
```

From the output you can see that the containers are named “web” and “db”. For db container you can see “web/db” parameter, which means the web container is linked to the db container. Now the web container can communicate with the db container and access information from it.

We learned that, by linking containers together, the source container will pass on its information to the destination container. So the web application we created can now access the information from the database container. In the backed, a secure tunnel has been created between the web and db containers. If you noticed, when creating the db container we did not use any "-p" flag for port mapping. So the db container does not expose any port for the external world to connect to it. Only the web container which has been linked to the db container can access its data.

How linking works:

When you run the Docker command with the "—link" flag, Docker would pass on the required credentials to the recipient container in the following two ways,

1. Using environment variables
2. /etc/hosts file update.

Now, let's launch a web container with links to the db container to view the environment variables set by Docker.

Execute the following command to launch a web2 container with db links and "env" command to view the list of environment variables set by Docker.

```
sudo docker run --rm --name web2 --link db:db training/webapp env
```

Note: the "-rm" flag will remove the container once it stops running the command inside the container.

The above output lists all the environment variables set by Docker on web2 container.

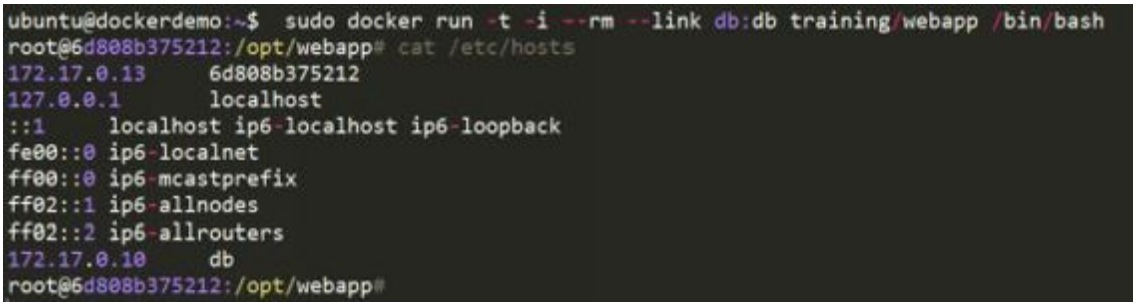
These variables are used to form a secure tunnel between the web and db container. All the variables are prefixed with DB. This is the alias name we mentioned while linking the container with db. If you have given the alias name as "database", the environment variables would have a prefix "database".

Now let’s launch another container to have a look at /etc/hosts file. Execute the following command to start an interactive session for the web container with db links.

sudo docker run -t -i --rm --link db:db training/webapp /bin/bash

Run the following command in the container to view the contents of /etc/hosts file.

cat /etc/hosts



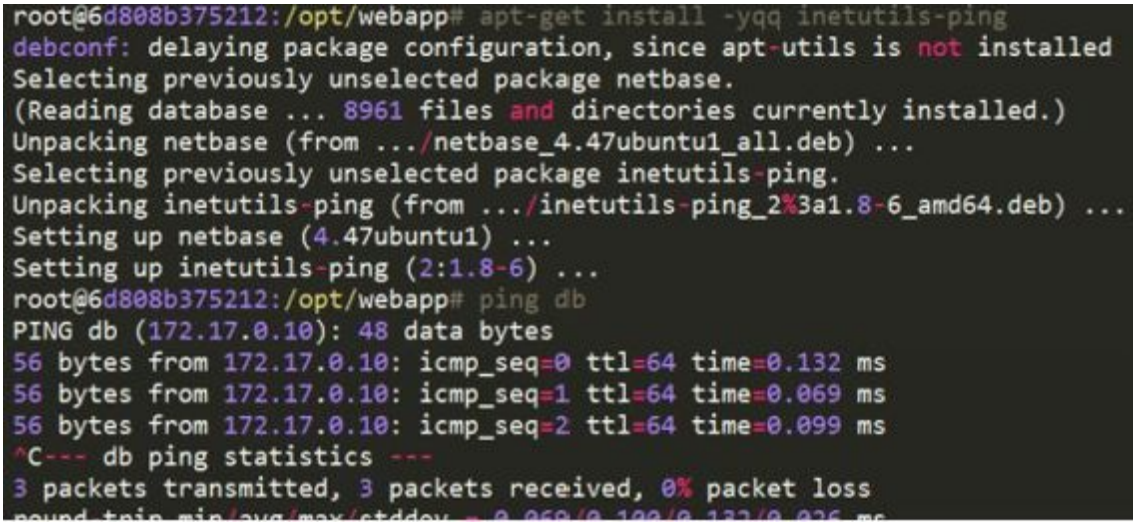
As you can see from the output, the hostname is the id of the container. The ip (172.17.0.10) entry for the db container is mapped to the alias name “db” in the hostname entry.

Now let’s try to ping the db container using its hostname “db”. Container does not come with a ping tool, so we have to install it on the container. Execute the following command to install ping in our web container.

apt-get install -yqq inetutils-ping

Run the following command to ping the db container.

ping db



When we issued the ping command the “db” hostname got resolved to the IP Address (172.17.0.10) of the db container. You can use this hostname in your application to connect to the database.

Tip: you can link multiple containers to the one container. Let’s consider a scenario where you need more than one web container which needs access to a common database. In this case, you can link all the web containers to one db container.

7

Data management in containers

In this chapter, we will learn how to manage data in containers.

The data present inside a container is stateless. Once the container is removed, all the data inside the container will be lost. Docker provides an efficient mechanism called “volumes” to persist the data used by containers. Using container volumes, all the data and logs of the container can be persisted.

For application development, using Docker volume is recommended because rapid code changes in the application will get reflected on the running container. Otherwise you need to launch a new container every time there is a change in application code.

The following figure illustrates the working of Docker volumes in a container.

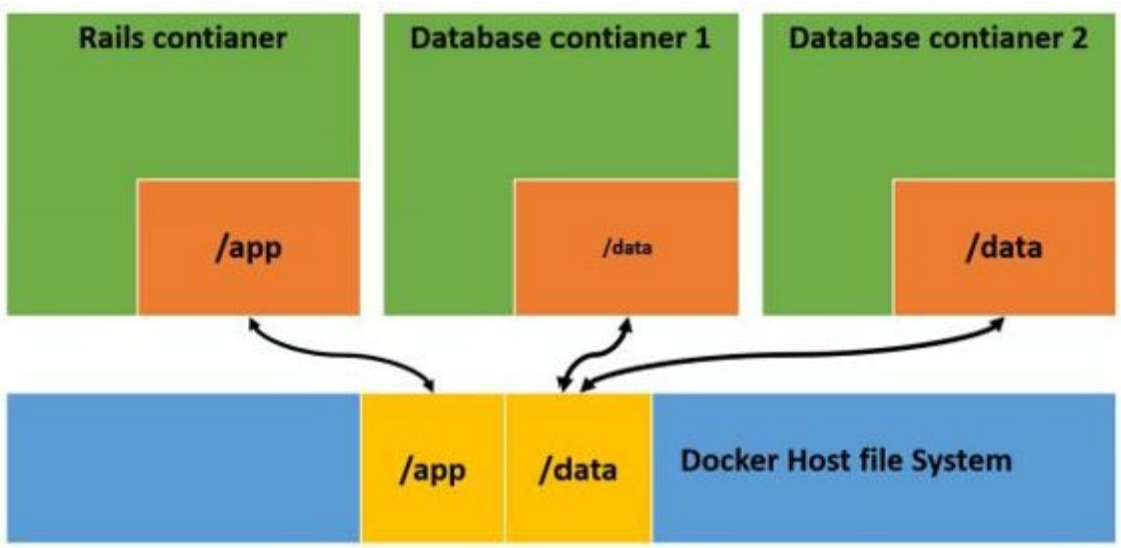


Fig 7-1: *Data volumes in containers.*

There are two ways by which you can manage data in containers

1. Using data volumes and
2. Data volume containers.

Docker data volumes:

Data volumes are special directories created inside a container which circumvents the uniform file system for persisting data inside the volumes. Docker data volume has the following features.

1. Data volumes are sharable and reusable.
2. Direct changes can be made to data volumes.
3. During the image update, the changes made in data volume won't be reflected on the image.

Adding data volumes to containers:

In this section we will learn how to add a data volume to a container. A data volume can be added to container using the –v flag. Execute the following command to add a data volume to python flask application.

```
sudo docker run -d -P --name web -v /webapp
```

training/webapp python app.py

```
ubuntu@dockerdemo:~$ sudo docker run -d -P --name web -v /webapp training/webapp python app.py
67fffbd22b030648c90981f92c66810f1c9b1619f3ca741918c8e5254d23a543
ubuntu@dockerdemo:~$
```

The above command creates a /webapp volume inside the web container.

Tip: You can also add volumes to containers using the Dockerfile. The VOLUME instruction in a Docker file creates volumes with the specified name.

Data volumes from host directories:

The host directories can be mounted as a volume in container as shown in Fig: 7-1. Same –v flag is used for creating host mounted data volumes with change in syntax. It takes the following form

-v /source-directory:/opt/directory

Execute the following command to create a host mounted volume for a web1 container.

sudo docker run -d -P --name web1 -v /src/webapp:/opt/webapp training/webapp python app.py

```
ubuntu@dockerdemo:~$ sudo docker run -d -P --name web1 -v /src/webapp:/opt/webapp training/weba
fcd8df1f2345d42d3cd06244e1dbfb3871048389e8cbee2c70d1528c3311632d
ubuntu@dockerdemo:~$
```

The above command will mount the host directory /src/webapp on to the containers /opt/webapp directory. Mounting host directory is very useful in rapid application development. All the application code can be copied to the host directory and you can view the application changes from the running container. If the specified directory does not exist in the host, Docker will automatically create it.

Creating read only data volume:

When you create a data volume, by default, Docker creates it with read/write mode. You can also create read only data volumes by specifying a “ro” parameter in the command. Execute the following command for creating a read only data volume in web2 container.

sudo docker run -d -P --name web2 -v /src/webapp:/opt/webapp:ro training/webapp python app.py

```
ubuntu@dockerdemo:~$ sudo docker run -d -P --name web2 -v /src/webapp:/opt/webapp:ro training/w
12aa74ea39258df399b52d99d9519776b57cd8ca7d14d96279850db53646b2dd
```

The only difference in creating read only volume is the additional “ro” parameter added to the “/opt/webapp”.

Data volume from host file:

Files in the host can also be mounted on to containers instead of directories. Same “-v” flag is used for mounting host files as volumes in containers. Execute the following command to create a data volume from the host file “bash_history”.

```
sudo docker run --rm -it -v ~/.bash_history:/.bash_history ubuntu /bin/bash
```

The above command launched a container with bash shell. Run the following command in the container to check if the file has been mounted.

```
ls .bash_history
```

The bash_history file has the bash history of the host system. Once you exit the container, the bash_history file in the host will have the bash history of the container.

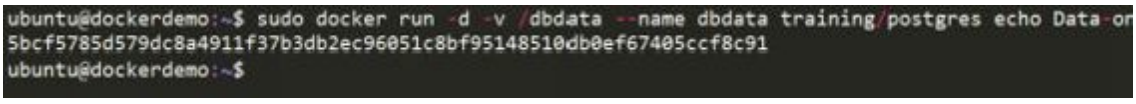
Containers as data volumes:

Non persistent containers can be used as data volumes. This is useful when you want to share persistent data among many containers. These containers are called data volume containers. Let’s try creating a data volume container.

Execute the following command to create a data volume container.

```
sudo docker run -d -v /dbdata --name dbdata training/postgres echo data-only container for postgres
```

The above command creates a data volume container named dbdata with /dbdata as a volume.



Now, let’s try mounting this data volume container to another container. “--volumes-from” flag is used for mounting a volume from a data volume container.

Execute the following command for creating a container with a volume mounted from dbdata container.

sudo docker run -d --volumes-from dbdata --name db1 training/postgres

```
ubuntu@dockerdemo:~$ sudo docker run -d --volumes-from dbdata --name db1 training/postgres
c0a3edcfc4af3bf81a915f514c4c49c7f52f736ff3f41d9a95ba4387ccb2521e
ubuntu@dockerdemo:~$
```

You can mount the dbdata from db3 container to some other container. Let's try creating a container db2 by mounting the volume dbdata from db1 container.

sudo docker run -d --name db3 --volumes-from db1 training/postgres

```
ubuntu@dockerdemo:~$  sudo docker run -d --name db3 --volumes-from db1 training/postgres
eda98745409ab914983f0593e6cdfdb8483af611a7712c07c2942f1edf03876c
ubuntu@dockerdemo:~$
```

This is how you can link the data volume with many containers. The advantage of container data volumes is that, if you delete any container, mounting a volume and linked to another container, the volume will not get deleted.

This enables migration of the data volume to another container. If you want to delete the volume, you need to run the "docker rm –v" command with the volume name.

Execute the following command to remove the data volume dbdata.

sudo docker rm -v dbdata

```
root@dockerdemo:~# sudo docker rm -v dbdata
dbdata
root@dockerdemo:~#
```

Backing up data volumes:

Volumes can be backed up, restored and migrated. For all these actions "--volumes-from" flag is used. Let's try backing up the dbdata volume by launching a container.

Execute the following command to launch a container to back up the dbdata volume.

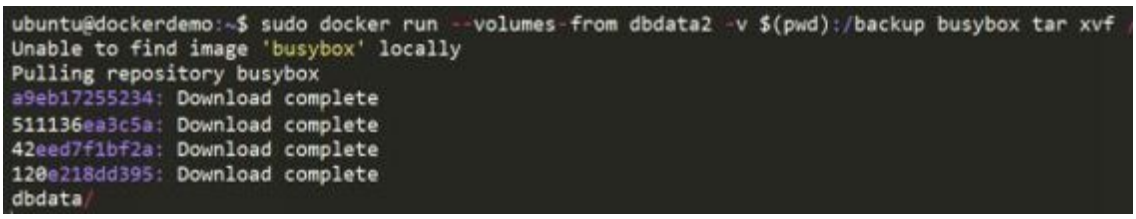
sudo docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf /backup/backup.tar /dbdata

```
ubuntu@dockerdemo:~$ sudo docker run --volumes-from dbdata -v $(pwd):/backup ubuntu tar cvf /ba
tar: Removing leading `/' from member names
/dbdata/
ubuntu@dockerdemo:~$
```

The above command created a container with dbdata volume mounted on it.

Also it created a volume by mounting the host directory. Finally it created a tar archive of dbdata in the host mounted volume. Once the container finished executing all the commands, it stops by leaving a backup data in the host directory.

You can restore the data by creating a new container by mounting the host directory and extracting the files from backup.tar archive.



All the above backup mechanisms can be used for migration, backup automation using various tools.

Working with Docker hub:

We have learnt the basics of Docker hub and its use. In this section we will learn more about Docker hub.

Docker hub is a public image repository for Docker created by Docker Inc. It contains thousands of images created by Docker hub users. Apart from just images, it provides various other features like authentication, build triggers, automatic builds and webhooks.

webhooks

When a repository is pushed successfully, webhooks will automatically trigger REST based actions to other applications. When a webhook is called it will generate a HTTP POST method with a JSON payload.

The generated JSON will look like the following.

```
{
  "push_data":{
    "pushed_at":12385123110,
    "images":[
      "image1",
      "image2",
      "image3"
    ],
    "pusher":"<username>"
  },
  "repository":{
    "status":"Active",
    "description":"descritpion of Docker repository",
    "is_automated":false,
    "full_description":"full description of repo",
    "repo_url":"https://registry.hub.Docker.com/u/user-
name/repo-name/",
    "owner":"<username>",
    "is_official":false,
    "is_private":false,
    "name":"<reponame>",
    "namespace":"<username>",
    "star_count":2,
    "comment_count":3,
    "date_created":1370344400,
    "Dockerfile":"Docker file contents",
```

```
    "repo_name":"<username>/<reponame>"
  }
}
```

Docker hub is an essential element in Docker ecosystem for simplifying Docker workflows. Also users can create private registries to have private images which are not searchable and accessed by other users.

Docker hub commands:

Docker has the following commands to work with Docker hub.

1. Docker login
2. Docker search
3. Docker pull
4. Docker push

Let's have a look at each of the commands.

Docker login:

Docker login command can be used to sign up and sign in to the Docker hub from command line. If you are using "Docker login" command in the command line, it will prompt for the user name and password.

Once you are authenticated against Docker hub a configuration file (.Dockercfg) will be created with Docker hub authentication tokens and placed in your host's home directory. This file is used for further logins.

Note: The username for Docker hub will be used as a namespace for all the images created by you from the authenticated Docker host.

For example, bibinwilson/Jekyll:v1

Docker search:

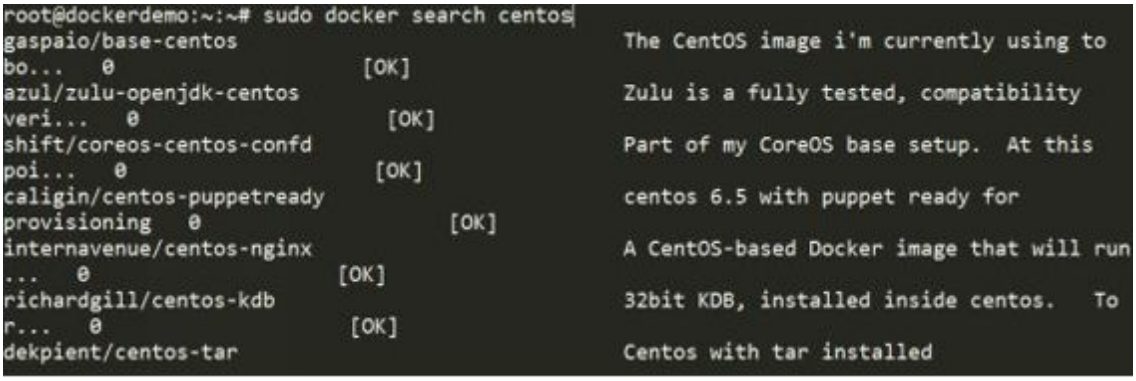
Docker search command is the great way for finding the images with a keyword or an image name. You can also use the Docker hub search interface to find

images. You can grab the image name from there and use it with Docker pull command to pull it down to your Docker host.

Let’s search for a centos image using the “Docker search” command.

docker search centos

The above command will list all the centos images in the Docker hub.



Docker pull:

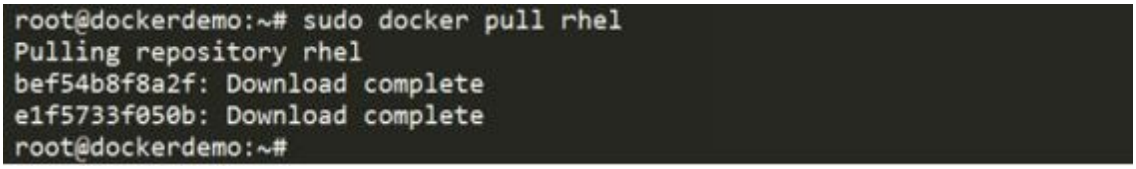
Docker pull command is used for pulling the images from Docker hub. This command has the following syntax.

Syntax: Docker pull <image name>

Let’s pull a RHEL image using the following command.

sudo docker pull rhel

This command will pull the “rhel” image from Docker hub.



Docker push:

Docker push command is used to push a repository to Docker hub. If you have created an image from Docker file or committed an image from a container, then you can push that image to the Docker hub.

Let’s try to push an image to Docker hub by committing a RHEL container.

Execute the following command to create an interactive rhel container.

sudo docker run -i -t --name rhel rhel /bin/bash

```
root@node2:~# docker run -i -t --name rhel rhel /bin/bash
bash-4.2# ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
bash-4.2#
```

Exit the container using the exit command.

exit

Commit the rhel container using the following command. Replace *bibinwilson* with your Docker hub username.

docker commit rhel bibinwilson/rhel:v1

Here v1 is the tag for the image.

```
root@node2:~# docker commit rhel bibinwilson/rhel:v1
6297fced1b765b96565ef406b048099e55bdb083eea9a654b0f0684ceb886cc6
root@node2:~#
```

Now we have a committed image in the name “bibinwilson/rhel:v1”. Execute the following command to push the image to Docker hub.

docker push bibinwilson/rhel:v1

```
root@node2:~# docker push bibinwilson/rhel:v1
The push refers to a repository [bibinwilson/rhel] (len: 1)
Sending image list
Pushing repository bibinwilson/rhel (1 tags)
bef54b8f8a2f: Image already pushed, skipping
6297fced1b76: Image successfully pushed
Pushing tag for rev [6297fced1b76] on {https://cdn-registry-1.docker.io/v1/repos itories/
bibinwilson/rhel/tags/v1}
root@node2:~#
```

We have successfully pushed the rhel image to the Docker hub.

8

Building and testing containers from scratch

In this chapter we will learn how to Dockerize applications from scratch. You can use images from the Docker public registry for your application, but for an in-depth idea for dockerizing applications, we will build our images from scratch.

In this section we create the following

1. A static web application running on apache
2. A MySQL image
3. WordPress application with MySQL database
4. Hosting multiple websites on a Docker host
5. Building and testing containers using Jenkins CI

Building docker images manually by executing commands is a tedious process. There is no need for creating images manually when it can be automated using dockerfile. In this section, we will discuss what a Dockerfile is, what it is capable of doing, and we will build a basic image using dockerfile.

Dockerfile

Dockerfile is a plain text file composed of various instructions (commands) and arguments listed sequentially to automate the process of image building. By executing “docker build” command with the path to the dockerfile, a docker image will be created by executing the set of instructions successively from the dockerfile. Before start building images from a dockerfile, you should understand all the instructions than can be used in a dockerfile.

Every instruction in the dockerfile will have the following syntax.

INSTRUCTION argument

Dockerfile supports the following instructions.

- FROM
- MAINTAINER
- RUN
- ENV
- CMD
- ADD
- EXPOSE
- ENTRYPOINT
- USER
- VOLUME

- WORKDIR

Let’s have look at the functionality of each instruction.

FROM

Every dockerfile should begin with the FROM instruction. It denotes the base image (base Ubuntu, centos RHEL etc.) from which the new image will be created. The base image can be any image, including the image you have created and committed in your docker host. If the image specified in the FROM instruction is not available in the host, docker will pull it from the docker hub.

Syntax

```
# Usage: FROM [image name]
FROM centos
```

MAINTAINER

This instruction sets the author for the image. It can be placed anywhere in the dockerfile as this does not perform any action on the image building process.

Syntax

```
# Usage: MAINTAINER [author name]
MAINTAINER Bibin Wilson
```

RUN

RUN executes a shell command. This instruction takes a Linux command as an argument. It adds a layer on top of the image and the committed changes will be available for the next instruction in the dockerfile.

Syntax

```
# Usage: MAINTAINER [author name]
MAINTAINER Bibin Wilson
```

ENV

ENV sets the environment variables and it takes a key value pair as an argument .The variables set by the ENV instruction can be used by scripts and applications running inside the container. This functionality in docker file provides better flexibility in running programs inside a docker container.

Syntax

```
# Usage: ENV Key Value
ENV  ACCESS_KEY 45dcdfrY
```

CMD

CMD, like RUN it can be used to execute a specific command. However, it will not be executed during the image building process but when a container is created from the build image. For example, if you want to start apache every time you create a container from an image with apache installed, you can specify the command to start apache in the CMD instruction. Also, in a dockerfile, you can specify the CMD instruction only one time. If specified multiple times, all the instruction except the last one will be nullified.

Syntax

Syntax:CMD ["executable","param1","param2"]

CMD ["param1","param2"]

CMD command param1 param2

CMD “echo” “Hello World”

ADD

ADD instruction takes two arguments: a source and a destination. This instruction copies a file from the source to the containers file system. If the source is a url, then the file from the url will be downloaded to the destination. You can also specify wildcard entries in the source path to copy all the files matching the entry.

Syntax

```
# Usage: ADD ADD [source directory or URL] [destination directory]
ADD  /source_folder /destination_folder
ADD *file* /destination_folder
```

COPY

This instruction is also used for copying files and folders from a source to the destination file system of a container. However, COPY instruction does not support url as a source. Multiple sources can be specified and copied to a folder in the destination using COPY. COPY has the same syntax as ADD.

EXPOSE

This instruction associates the specified port for enabling networking between a docker container and the outside world. The default container ports that are accessible from the host cannot be defined using EXPOSE. Host to container mappings can only be done using the “-p” flag with the docker run command.

Syntax

```
# Usage: EXPOSE [port]
EXPOSE 443
EXPOSE [443, 80, 8080]
```

ENTRYPOINT

Using this instruction a specific application can be set as default and start every time a container is created using the image.

```
Syntax: Comes in two flavours
ENTRYPOINT [‘executable’, ‘param1’,’param2’]
ENTRYPOINT command param1 param2
```

ENTRYPOINT can be used with CMD to remove the “application” from CMD leaving only the arguments which will be passed to ENTERYPOINT.

```
CMD “ This is an argument for entrypoint”
ENTRYPOINT echo
```

USER

It sets the UID (username) which has to be used to run the container from the image.

Syntax

```
# Usage: USER [uid]
USER 543
```

VOLUME

This instruction is used to mount a specific file or a directory to a container. The host directory or file mentioned in the instruction will be mounted to the container when created.

Syntax

```
# Usage: VOLUME ["/dir1", "/dir2" ..]
VOLUME [“var/log”]
```

WORKDIR

WORKDIR sets the Working directory for the RUN, CMD and ENTRYPOINT instructions. All the commands will be executed in the directory specified in WORDDIR instruction.

Syntax

```
# Usage: WORKDIR /path
WORKDIR  /root
```

.dockerignorefile

.dockerignore file is like .gitignore file. All files and directories which has to be excluded should present in the .dockerignore file. It is interpreted by new-line separated list of files and directories.

Example dockerfile

A typical dockerfile will look like the following. It is dockerfile for creating a MongoDB image.

```
FROM ubuntu
MAINTAINER Bibin Wilson
RUN apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
RUN echo "deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist 10gen" | tee -a /etc/apt/sources.list.d/10gen.list
RUN apt-get update
RUN apt-get -y install apt-utils
RUN apt-get -y install mongodb-10gen
CMD ["/usr/bin/mongod", "--config", "/etc/mongodb.conf"]
```

Now we will create a normal MongoDB image using the above dockerfile. Follow the steps given below to build an image from dockerfile.

1. Create a directory name MongoDB, cd into that directory, create a file named Dockerfile and copy the above dockerfile contents onto the file.

```
mkdir mongodb && cd mongodb
touch Dockerfile
nano Dockerfile
```

```
root@dockerdemo:~# mkdir mongodb && cd mongodb
root@dockerdemo:~/mongodb# touch Dockerfile
root@dockerdemo:~/mongodb# nano Dockerfile
```

2. “docker build” command is used to build an image. To list all the options associated with docker build command, execute the following command.

docker build --help

```
root@dockerdemo:~/mongodb# docker build --help

Usage: docker build [OPTIONS] PATH | URL | -
Build a new image from the source code at PATH

  --force-rm=false      Always remove intermediate containers, even after unsuccessful b
  --no-cache=false      Do not use cache when building the image
  --pull=false          Always attempt to pull a newer version of the image
  -q, --quiet=false     Suppress the verbose output generated by the containers
  --rm=true             Remove intermediate containers after a successful build
  -t, --tag=""          Repository name (and optionally a tag) to be applied to the resu
```

3. Let’s build our MongoDB image using the following command.

docker build -t mongodb .

Note: “docker build” command is associated with few options. You can view the options using “docker build --help” command. In the above build command we use “-t” to name the image “mongodb” and “.” Represents the location of docker file as current directory. If the dockerfile is present in different location, you need to give the absolute path of dockerfile instead of “.”

```
root@dockerdemo:~/mongodb# docker build -t mongodb .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
ubuntu:latest: The image you are pulling has been verified
511136ea3c5a: Pull complete
c7b7c6419568: Pull complete
70c8faa62a44: Pull complete
d735006ad9c1: Pull complete
04c5d3b7b065: Pull complete
Status: Downloaded newer image for ubuntu:latest
 ---> 04c5d3b7b065
Step 1 : MAINTAINER bibin wilson
 ---> Running in cdb9b00ec9b6
 ---> 5e0ec4d0998e
Removing intermediate container cdb9b00ec9b6
Step 2 : RUN apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
 ---> Running in 37b7eb89aa22
Step 7 : CMD /usr/bin/mongod --config /etc/mongodb.conf
 ---> Running in 010143cbc2f3
 ---> 9aec0bdc5187
Removing intermediate container 010143cbc2f3
Successfully built 9aec0bdc5187
```

Till now we have learned about docker file and its options. Also we have learned

how to build an image from dockerfile. In next section we will look in to some best practices for writing a docker file.

Dockerfile Best Practices

Follow the best practices given below while working with dockerfiles.

- Always make a dockerfile with minimum configuration as possible.
- Use .dockerignore file to exclude all the files and directories which will not be included in the build process. For example .git folder. You can exclude .git folder by including it in the .dockerignore file.
- Avoid all unnecessary package installations to keep the image size minimal.
- Run just one process per container. It is a good practice to decouple your application for better horizontal scaling and container reuse. For example, run the web application and database in different containers and link them together using "- -link" flag.
- Many base images in docker hub are bloated. Use small base images for your dockerfile and make sure you use the official and trusted base images with minimum size.
- Use specific image tags while building image and while using it in FROM instruction.
- Group all the common operations. For example, use "apt-get update" with "apt-get install" using "\" to span multiple lines on your installs.

For example,

```
RUN apt-get update && apt-get install -y \
  git \
  libxml2-dev \
  python \
  build-essential \
  make \
  gcc \
  python-dev \
  locales \
  python-pip
```

- Use build cache while building images. Docker will look for existing image while building an image to reuse it rather than building a duplicate image. If you do not want to use the build cache, you can explicitly specify the “--no-cache=true" flag for not using the cache.

A static website using Apache

In this section, we will create a Docker image and create containers from the image which runs the static website. We need the following to run the apache container with a static website.

1. Dockerfile with all the specifications to run apache.
2. Static website files
3. An apache-config file to configure apache to run the static website.

Follow the steps given below to get the apache container up and running.

Note: You can get the Docker file and associated files in the demo from my github repository. Here is the repository link.
https://github.com/Dockerdemo/apache

1. Create a folder named apache and cd in to the apache directory.

```
mkdir apache && cd apache
```

```
root@node2:~# mkdir apache && cd apache
root@node2:~/apache#
```

2. Create a Docker file

```
touch Dockerfile
```

3. Create a file named apache-config.conf and copy the following contents on to the file.

```
<VirtualHost *:80>
ServerAdmin admin@yourdomain.com
DocumentRoot /var/www/website
<Directory /var/www/website/>
Options Indexes FollowSymLinks MultiViews
AllowOverride All
Order deny,allow
Allow from all
</Directory>
```

```
ErrorLog ${APACHE_LOG_DIR}/error.log
CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

4. Download the static website files to the apache folder from the github link.
5. Copy the following snippet on to the Docker file.

```
FROM ubuntu:latest
MAINTAINER Bibin Wilson <bibin.w@hcl.com>
RUN apt-get update
RUN apt-get -y upgrade
RUN apt-get -y install apache2
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
EXPOSE 80
ADD website /var/www/website
ADD apache-config.conf /etc/apache2/sites-enabled/000-default.conf
CMD /usr/sbin/apache2ctl -D FOREGROUND
```

Here is what the above Dockerfile does.

- Pulls the base image Ubuntu from public repository if it is not available locally in your server
- Updates the image and installs apache2 using RUN
- Set few apache specific environment variables which will be used by the custom apache config file that we will use.
- Exposes port 80
- Adds the website folder from the host on to /var/www location in the container.
- Adds the custom apache config file we created to the container.
- Finally it starts the apache2 service.

6. Our project will contain the following files as shown in the following tree structure.

```
apache
 --Dockerfile
 --website
   --Files
 --apache-config.conf
```

7. Now we have the Docker file, website files and apache config file in place. Next step is to build an image from the Docker file. Run the Docker build command given below to build the apache image from our Dockerfile.

docker build –t apachedemo .

```
root@node2:~/apache# docker build -t apachedemo .
Step 0 : FROM ubuntu:latest
 ---> 1357f421be38
Step 1 : MAINTAINER Bibin Wilson <bibin.w@hcl.com>
 ---> Using cache
 ---> 02919d5612fd
Step 2 : RUN apt-get update
 ---> Using cache
 ---> fa0b97128a52
Step 3 : RUN apt-get -y upgrade
 ---> Using cache
 ---> 7e2c4a0b194b
  .
  .
 Removing intermediate container 2757e1d9a574
Step 13 : CMD /usr/sbin/apache2ctl -D FOREGROUND
 ---> Running in ea8ea88a8732
 ---> 2922b450684f
Removing intermediate container ea8ea88a8732
Successfully built 2922b450684f
```

8. Now we have our apache image ready and we can create containers from it. Run the following Docker command to create a new apache container

docker run -d -p 80:80 --name staticwebsite apachedemo

```
root@node2:~/apache# docker run -d -p 80:80  --name staticwebsite apachedemo
582dcc54c73279c1b184b9a8b439aefcbc015fd8a1df32270ed7a4c67b77c090
root@node2:~/apache#
```

9. We have a running apache container with our static website with port mapped to 80. You can access the website from the browser on port 80 using the host IP of DNS

http://hostip:80

The image shows the static website we had in the website folder.

10. Run the Docker ps command to see more information about the container.

```
root@node2:~/apache# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
ATUS                PORTS               NAMES
582dcc54c732        apachedemo:latest   /bin/sh -c '/usr/sbi   7 minutes ago
 7 minutes          0.0.0.0:80->80/tcp  staticwebsite
root@node2:~/apache#
```

If you do not want to recreate containers every time you update your static website files, you can mount a folder from the host containing the website files on to containers instead of copying the files to container. So that every time you make a change to the file will be reflected on the running container.

Now let's create a container by mounting the website folder to the container.

-v flag is used for mounting a volume to the container. Execute the following command to creating a new container with website folder in the host as a mount point for the container

```
docker run -p 8080:80 -d apachedemo -v \
/root/apache/website:/var/www/website
```

Creating MySQL image and containers

In this section we will create a MySQL Docker image from a base Ubuntu image. We will do the following to create our MySQL container.

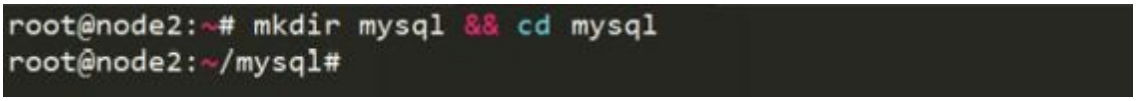
1. Create a Dockerfile with commands to install and configure MySQL server
2. Create a shell script for creating a user, database and staring the server.
3. Build an image named mysql from the created files.
4. Create a container from the mysql image.

Note: You can get the Docker file and associated files in the demo from my github repository. Here is the repository link.
https://github.com/Dockerdemo/mysql

Let's start creating our mysql image.

1. Create a directory name mysql and cd in to the same

```
mkdir mysql && cd mysql
```



2. Create a file named start.sh and copy the following shell script on to the file. This script creates users, databases by getting the values from the environment variables specified in the Dockerfile and restarts the mysql server.

```
#!/bin/bash
/usr/sbin/mysqld &
sleep 5
echo "Creating user"
echo "CREATE USER '$user' IDENTIFIED BY '$password'" |
mysql --default-character-set=utf8
echo "REVOKE ALL PRIVILEGES ON *.* FROM
'$user'@'%'; FLUSH PRIVILEGES" | mysql --default-
```

```
character-set=utf8
echo "GRANT SELECT ON *.* TO '$user'@'%'; FLUSH
PRIVILEGES" | mysql --default-character-set=utf8
echo "finished"
if [ "$access" = "WRITE" ]; then
echo "GRANT ALL PRIVILEGES ON *.* TO '$user'@'%'
WITH GRANT OPTION; FLUSH PRIVILEGES" | mysql --
default-character-set=utf8
fi
mysqladmin shutdown
/usr/sbin/mysqld
```

The above script creates a user with the password specified as the environment variable in the Dockerfile. You can specify the access right for the user in if block. In the above file we have WRITE access which grants all the privileges to the user. At last it restarts the MySQL server.

3. Create a Dockerfile and copy the following snippet on to the file.

```
FROM ubuntu:latest
MAINTAINER Bibin Wilson
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get -y install mysql-client mysql-server curl
RUN sed -i -e"s/^bind-address\s*=\s*127.0.0.1/bind-address
= 0.0.0.0/" /etc/mysql/my.cnf
ENV user Docker
ENV password root
ENV access WRITE
ADD ./start.sh /usr/local/bin/start.sh
RUN chmod +x /usr/local/bin/start.sh
EXPOSE 3306
```

Here is what the Docker file does.

1. Updates the image.
2. Installs MySQL server and client

3. Changes the bind address to 0.0.0.0 on my.cnf file to get remote access
4. Sets few environment variables to be used by the start.sh script.
5. Adds the start.sh file to the image
6. Runs the start.sh script
7. Exposed port 3306 on the container.

4. Now we have the Dockerfile and the start script in place. Run the following Docker build command to build our mysql image.

docker build –t mysql .

```
root@node2:~/mysql# docker build -t mysql .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:latest
 ---> 5506de2b643b
Step 1 : MAINTAINER Bibin Wilson
 ---> Running in 0d94f148e302
 ---> ea20327f05a0
Removing intermediate container 0d94f148e302
Step 2 : RUN apt-get update
 .
 .
Step 9 : ADD ./start.sh /usr/local/bin/start.sh
 ---> c51493c5cde1
Removing intermediate container ba4121f05fb1
Step 10 : RUN chmod +x /usr/local/bin/start.sh
 ---> Running in fe310009b62b
 ---> 6671c448f53b
Removing intermediate container fe310009b62b
Step 11 : EXPOSE 3306
 ---> Running in bd49aa2b5462
 ---> cbbb89fdc7ea
Removing intermediate container bd49aa2b5462
Successfully built cbbb89fdc7ea
```

5. Our mysql image has be successfully built. You can start a mysql container using the following Docker command.

docker run -d -p 3306:3306 --name db mysql

```
root@node2:~/mysql1# docker run -d -p 3306:3306 --name db mysql
b3766c847efc6cdda1d61a253a3f14294b221fd494a1d5a21d88ad363ba23067
```

6. Now if you run the Docker ps command, you can see the running mysql container name db.

docker ps

```
root@node2:~/mysql# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS                    NAMES
b3766c847efc        mysql:latest        /usr/local/bin/start
3 seconds ago       Up 2 seconds        0.0.0.0:3306->3306/tcp   db
root@node2:~/mysql#
```

7. You can now access the database using the containers IP. You can get the full details of the container using *"Docker inspect"* command. You can get the db containers IP using the following command.

docker inspect --format '{{ .NetworkSettings.IPAddress }}' db

Here db is the container's name.

```
root@node2:~/mysql# docker inspect --format '{{ .NetworkSettings.IPAddress }}' db
172.17.0.73
root@node2:~/mysql#
```

8. To access the mysql server running on the db container you should have mysql client installed on the Docker host. Run the following mysql command to access the database. Make sure you use the correct username and password used in the Dockerfile.

```
root@node2:~/mysql# mysql -h 172.17.0.73 -u dockerdemo -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 5.5.40-0ubuntu0.14.04.1 (Ubuntu)

Copyright (c) 2000, 2014, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

You can use this container for backend database for you applications using the container IP and database credentials.

As explained earlier another way of linking containers is using Docker links, we will be using this approach in another example in the following section.

Creating a WordPress container

In this demo, we will see how to create a WordPress image to run a WordPress container.

You can run the backend database in the same container or you can use a

different container for the database. We will use a standalone MySQL container we created to run our WordPress application.

We will do the following to get our WordPress container ready.

1. Create a Docker file with specifications to install all the necessary components needed to run a WordPress CMS
2. Build the wordpress image.
3. Run the wordpress container.

Note: You can get the Docker file and associated files in the demo from my github repository. Here is the repository link.
https://github.com/Dockerdemo/wordpress

A typical WordPress installation should have the following requirements.

1. **A web server** – we will use apache web server to run our WordPress application
2. PhP run time environment
3. **Backend SQL database** – we will use the mysql container we created as the backend database for WordPress.

Let's get started with building the WordPress image.

1. Create a directory named wordpress and cd in to the directory using the following command.

```
mkdir wordpress && cd wordpress
```

2. Create a Dockerfile and copy the following snippet on to the file.

```
FROM ubuntu:latest
MAINTAINER Bibin Wilson <bibin.w@hcl.com>
RUN apt-get update
RUN apt-get -y upgrade
RUN apt-get -y install apache2 libapache2-mod-php5 pwgen
python-setuptools vim-tiny php5-mysql php5-ldap
RUN RUN apt-get -y install php5-curl php5-gd php5-intl
php-pear php5-imagick php5-imap php5-mcrypt php5-
memcache php5-ming php5-ps php5-pspell php5-recode
```

```
php5-sqlite php5-tidy php5-xmlrpc php5-xsl
ENV APACHE_RUN_USER www-data
ENV APACHE_RUN_GROUP www-data
ENV APACHE_LOG_DIR /var/log/apache2
ENV APACHE_LOCK_DIR /var/lock/apache2
ENV APACHE_PID_FILE /var/run/apache2.pid
EXPOSE 80
ADD http://wordpress.org/latest.tar.gz /wordpress.tar.gz
RUN tar xvzf /wordpress.tar.gz
RUN rm -rf /var/www/
RUN mv /wordpress /var/www/
ADD apache-config.conf /etc/apache2/sites-enabled/000-default.conf
CMD /usr/sbin/apache2ctl -D FOREGROUND
```

The above Dockerfile does the following.

- Updates the images
- Installs required apache2 and php elements required for wordpress
- Sets few environment variables for apache, which will be used by the apache conf file associated with the Docker file.
- Exposes port 80 on container
- Downloads the latest wordpress setup files and copies it to the desired folder.
- Adds the apache config file from the host to container.
- Starts apache server.

3. Create an apache-config.conf file and copy the following snippet on to the file.

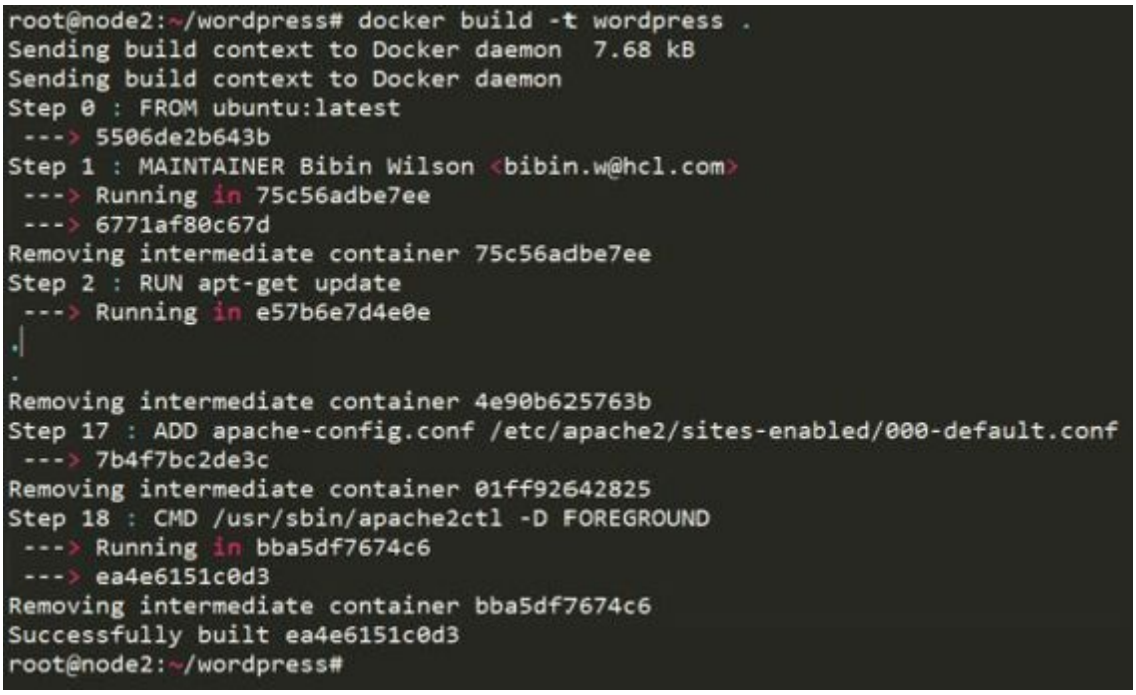
```
<VirtualHost *:80>
ServerAdmin admin@yourdomain.com
DocumentRoot /var/www/wordpress
<Directory /var/www/wordpress/>
Options Indexes FollowSymLinks MultiViews
AllowOverride All
```

```
Order deny,allow
Allow from all
</Directory>
ErrorLog ${APACHE_LOG_DIR}/error.log
CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

The above config file is same as the one we created in the first apache demo

4. Build the wordpress image from the Docker file using the following command.

```
docker build -t wordpress .
```



Now we have a wordpress image. In order to install and setup wordpress, you need a backend SQL database. In our wordpress containers we haven't configured any database.

We will use the mysql container as the backend database for our wordpress application.

There are two ways by which you can link the MySQL container to the WordPress container.

1. Run the mysql container with ports mapped to the host and use the IP address of the container for linking wordpress to the database. In this

case you can link a WordPress container in another host to the host running mysql container.

2. Run the mysql container without mapping it to the host port and link the wordpress container using the Docker –link flag specifying the mysql container name.

Running a two container WordPress application

In this section we will learn how to set up a two container wordpress application using a wordpress container and a mysql container. You can set up wordpress and mysql on the same container but it is advisable to have distinct components for the database and application. Let's look an example configurations for this application.

In this example, we will run a WordPress application using MySQL container for the database.

1. Create a container from our MySQL image by mapping host port 3306 to container port 3306 using the following command.

docker run -d -p 3306:3306 --name db mysql

```
root@node2:~# docker run -d -p 3306:3306 --name db mysql
9edd6a11206cd7747a7ad1144d07bf39301ea0b2bef17f84435d0ddf4bdc4dd3
root@node2:~#
```

2. Create a wordpress container linking the db container we created using the –link flag from the following command.

docker run -d -p 80:80 --name web --link db:db wordpres

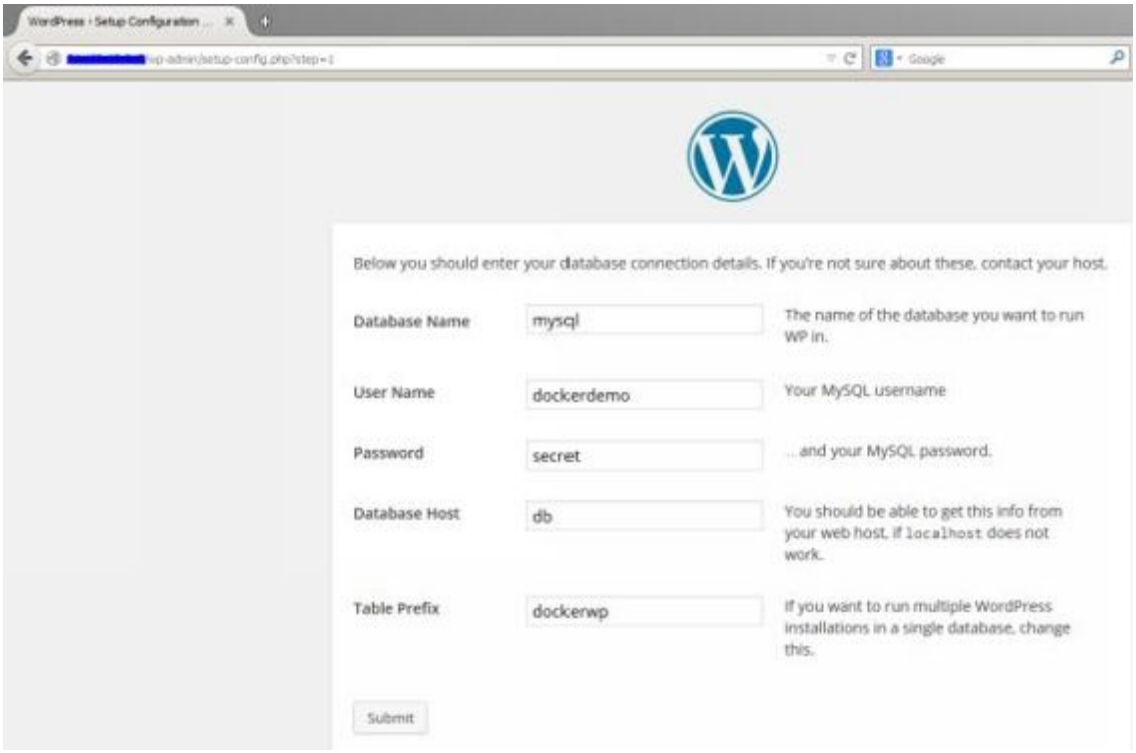
```
root@node2:~# docker run -d -p 80:80 --name web --link db:db wordpress
ac7eb1baee54b04607eff5d9d3de2a74f9ff101b1fced603e51552a1850f3a3e
root@node2:~#
```

3. If you run the Docker ps command, you can see our web container linked with the db container.

docker ps

```
root@node2:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
              STATUS              PORTS                    NAMES
ac7eb1baee54        wordpress:latest    /bin/sh -c '/usr/sbi   3 seconds ago
            Up 2 seconds        0.0.0.0:80->80/tcp       web
9edd6a11206c        mysql:latest        /usr/local/bin/start   3 minutes ago
            Up 3 minutes        0.0.0.0:3306->3306/tcp   db,web/db
root@node2:~#
```

Now you can access the WordPress setup page using the host IP address. Access the setup page and fill in the database name, password and hostname (db container name) and continue for WordPress installation.



4. Once the installation is complete, you will have a running two container WordPress application running on your Docker host.

5. If you want more instances of the configured WordPress, you can commit the container and start new containers from the committed image. Run the following command to commit the web container and create a new configured WordPress image named WordPress-configured.

docker commit web wordpress-configured

```
root@node2:~# docker commit web wordpress-configured
2185d55a1fd50a79bbb5f2391f0d1f501c85b49d32c8c4b8899fbdc2be3bbc3e
root@node2:~#
```

6. Now, if you list the images in your Docker host, you can see the newly created wordpress-configured image. Run the following command to list the wordpress image.

docker images wordpress-configured

```
root@node2:~# docker images wordpress-configured
REPOSITORY              TAG                 IMAGE ID            CREATED
 VIRTUAL SIZE
wordpress-configured    latest              2185d55a1fd5        3 minutes ago
55.3 MB
root@node2:~#
```

7. Now you can create a configured WordPress directly from the WordPress-configured image. In order to test this, stop and remove the web container and create a new WordPress container from the committed image, link it to db container and see if you get the configured WordPress application.

docker rm -f web

docker run -d -p 80:80 --name web –link db:db wordpress-configured

```
root@node2:~# docker rm -f web
web
root@node2:~# docker run -d -p 80:80 --name web --link db:db wordpress-configured
56743e499a2b793dac2a516b65b984ab6700c694cbb69b0f2fcfeb87835c3d1a
root@node2:~# docker start web
web
```

Now you can access the WordPress application from the browser without the initial configuration.

Running multiple websites on a single host using Docker:

In this section we will learn how to run multiple websites on a single host using a reverse proxy HAproxy. The following image illustrates how the architecture will look like.

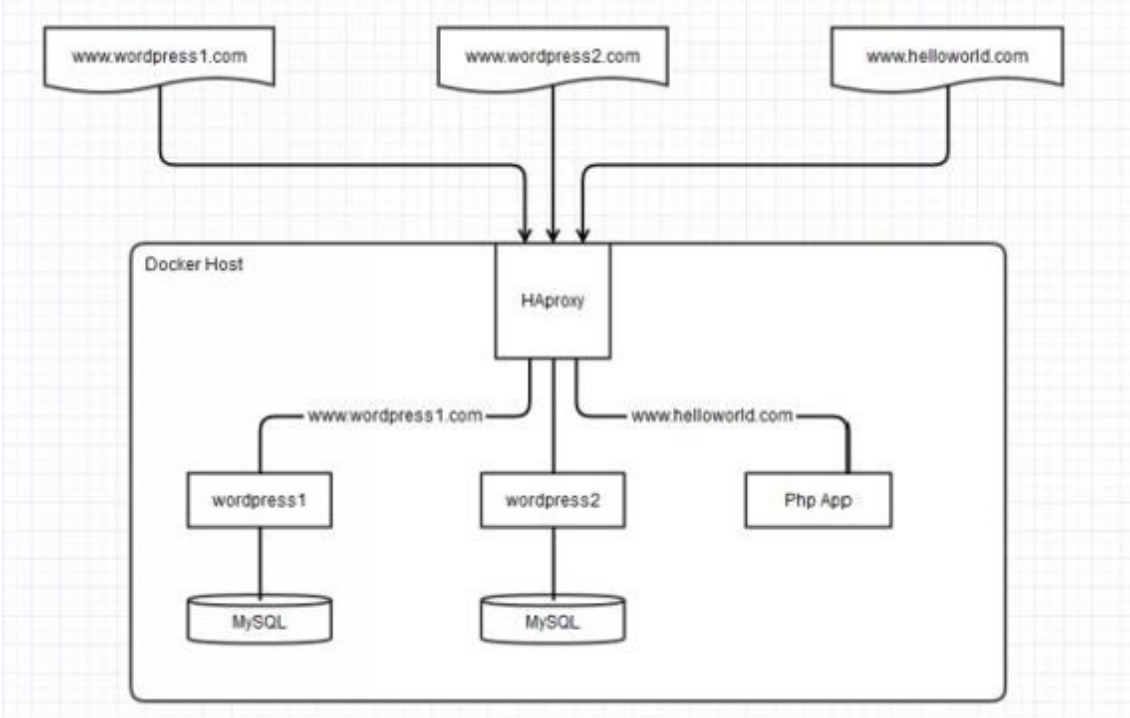


Fig 8-1: *Docker multiple website hosting*

Note: For this demonstration we will be using domain name internal to the host using the hosts file. You can also test this by mapping different domain names to the Docker host.

We already have a working wordpress-configured and mysql image. For this demonstration we will create a HAproxy container from the public HAproxy image named Dockerfile/haproxy and a basic hello world php application using tutum/hello-world image.

Follow the steps give below to setup a multi website Docker host.

1. Create a mysql container from mysql image using the following command.

```
docker run -d --name db mysql
```

```
root@node2:~# docker run -d --name db mysql
efb32f95661ad3eee43d1e14df5207e257f57e73093a86f605e9a2f59870800b
root@node2:~#
```

2. Create a wordpress container named wordpress1 from the wordpress-configured image linked to db container using the following command.

```
docker run -d --name wordpress1 --link db:db wordpress-configured
```

```
root@node2:~# docker run -d --name wordpress1 --link db:db wordpress-final
ba33a6aa158a3a43215c5741a79e9a67c5afcb95161493a2997b4b302af0a327
root@node2:~#
```

3. Create a hello-world php application container using tutum/hello-world public image using the following command.

docker run -d --name hello-world tutum/hello-world

```
root@node2:~# docker run -d --name hello-world tutum/hello-world
11bef59d2c75c24ca4401fe128bc96e366aec956a4fefa64a06b368b5d033e5e
root@node2:~#
```

4. Create two internal DNS entries, test1.domain.com and test2.domain.com in /etc/hosts file with Docker host IP for routing traffic from HAproxy to respective backend applications.
5. Create a haproxy-config directory and create a haproxy.cfg file in that directory. The haproxy.cgf file is given below. In this file we will update the IP's of wordpress1 and hello-world applications.
6. HAproxy container will listen to port 80 of Docker host. Test1.domain.com is mapped to wordpress1 and test2.domain.com is mapped to hello-world container in the file given below.

```
global
log 127.0.0.1   local0
log 127.0.0.1   local1 notice
user haproxy
group haproxy
defaults
log     global
mode    http
option  httplog
option  dontlognull
option forwardfor
option http-server-close
contimeout 5000
clitimeout 50000
srvtimeout 50000
errorfile 400 /etc/haproxy/errors/400.http
```

```
errorfile 403 /etc/haproxy/errors/403.http
errorfile 408 /etc/haproxy/errors/408.http
errorfile 500 /etc/haproxy/errors/500.http
errorfile 502 /etc/haproxy/errors/502.http
errorfile 503 /etc/haproxy/errors/503.http
errorfile 504 /etc/haproxy/errors/504.http
stats enable
stats auth username:password
stats uri /haproxyStats
frontend http-in
bind *:80
# Define hosts based on domain names
acl host_test1 hdr(host) -i test1.domain.com
acl host_test2 hdr(host) -i test2.domain.com
use_backend test1 if host_test1
use_backend test2 if host_test2
backend test1 # test1.domain.com wordpress1 contianer
balance roundrobin
option httpclose
option forwardfor
server s2 172.17.0.33:80 #ip of wordpress1 contianer
backend test2 # test2.domain.com hello-world container
balance roundrobin
option httpclose
option forwardfor
server s1 172.17.0.19:80 #ip pf hello-world container
```

In the above config file, you need to replace the IP addresses under backend section with the IP address of the application containers. The default configuration of HAproxy will be overridden by our haproxy.cfg file.

7. Now we have all the configurations ready. Start the HAproxy container using the following command.

```
docker run -d -p 80:80 --name lb -v ~/haproxy-
```

config:/haproxy-override Dockerfile/haproxy

```
root@node2:~# docker run -d -p 80:80 -v ~/haproxy-config:/haproxy-override
dockerfile/haproxy
Unable to find image 'dockerfile/haproxy' locally
Pulling repository dockerfile/haproxy
74c8d9507be7: Download complete
7b29802615c1: Download complete
07e8966a8e0dcd71fd5f0722d7689487f923a8c483aa59999582cf46431eede7
root@node2:~#
```

8. If you do a Docker ps, you can view the running containers (haproxy, wordpress ,hello-world and MySQL)

docker ps

```
root@node2:~# docker ps
CONTAINER ID        IMAGE                          COMMAND
CREATED              STATUS              PORTS                          NAMES
07e8966a8e0d        dockerfile/haproxy:latest      bash /haproxy-start    About a
minute ago    Up About a minute    443/tcp, 0.0.0.0:80->80/tcp    dreamy_kowalevski
11bef59d2c75        tutum/hello-world:latest       /run.sh                50 minutes
ago        Up 49 minutes        80/tcp                          hello-world
ba33a6aa158a        wordpress-final:latest         /bin/sh -c '/usr/sbi   About an
hour ago     Up About an hour     80/tcp                          wordpress1
efb32f95661a        mysql:latest                   /usr/local/bin/start   About an
hour ago     Up About an hour     3306/tcp                        db,wordpress1/db
```

Now let's test our applications using curl. You can test your application by giving curl request to test1.domain.com and test2.domian.com. If you can map the custom domain names to the Docker host, then you can access the application publicly from the browser. Since we have internal DNS entries, we will only test this internally using curl.

9. Run the following command to test test1.domain.com

curl test1.domain.com

```
root@node2:~# curl test1.domain.com
<html>
<head>
        <title>Hello world!</title>
        <link href='http://fonts.googleapis.com/css?family=Open+Sans:400,700' rel='s

</head>
<body>
        <img id="logo" src="logo.png" />
        <h1>Hello world!</h1>
        <h3>My hostname is 11bef59d2c75</h3>    </body>
</html>
root@node2:~#
```

As you can see, for test1.domain.com, HAproxy directed the request to hello-world application container. Same way, test2.doamin.com will be directed to wordpress1 application.

Building and testing containers using Jenkins

In this section we will learn how to use Jenkins CI for Dockerfile builds. You need the following setups to automate Docker builds using Jenkins.

1. A Jenkins server
2. Github account configured with your laptop for pushing and updating the Dockerfile for builds.

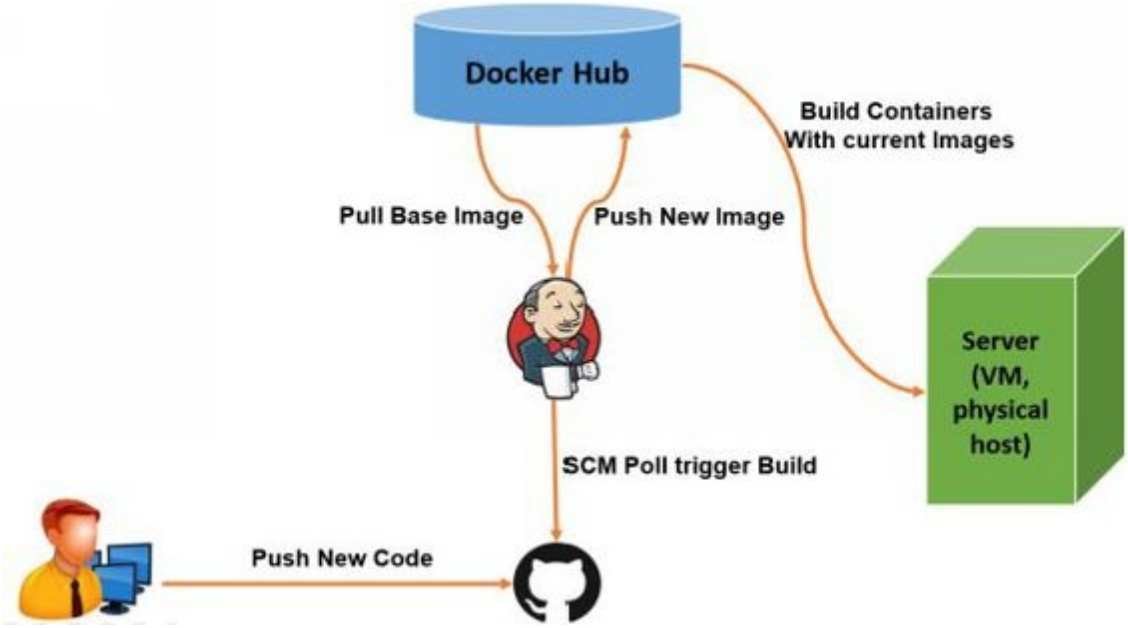


Fig 8-2: *Building and testing containers using Jenkins*

Setting up a Jenkins server

You can set up a Jenkins server manually or you can use chef community cookbook for automatic installation. In this section you will learn how to install Jenkins manually on a RHEL server. Follow the steps given below to set a Jenkins server

1. Login to the server and update the server repositories

```
sudo yum update
```

2. Jenkins needs java to be installed on the server. So if you are using an existing server with java skip to step 3 or else install java using the

following command

```
sudo yum install java-1.6.0-openjdk
```

3. Once java is installed, verify the java version the proceed to the next step
4. Add the Jenkins repository to the server using the following commands.

```
wget -O /etc/yum.repos.d/jenkins.repo http://pkg.jenkins-ci.org/redhat/jenkins.repo
```

Note: If RHEL server does not have wget utility install it using the following command.

```
yum install wget
```

5. Add the repository key

```
rpm --import http://pkg.jenkins-ci.org/redhat/jenkins-ci.org.key
```

6. Install Jenkins

```
sudo yum install Jenkins
```

7. Once installed, add Jenkins to the startup so that it starts every time when you restart the server.

```
sudo chkconfig Jenkins on
```

8. Start the Jenkins service

```
sudo service start Jenkins
```

9. Jenkins UI accepts connection on port 8080 by default. You can access the Jenkins web ui using the public ip followed by the 8080 port number : eg : 54.34.45.56:8080

Github setup

You need to have a version control system configured for configuring Jenkins builds for Docker. In this demonstration we will use github as a version control system. We will use the Dockerfile and files for apache static website we tested earlier.

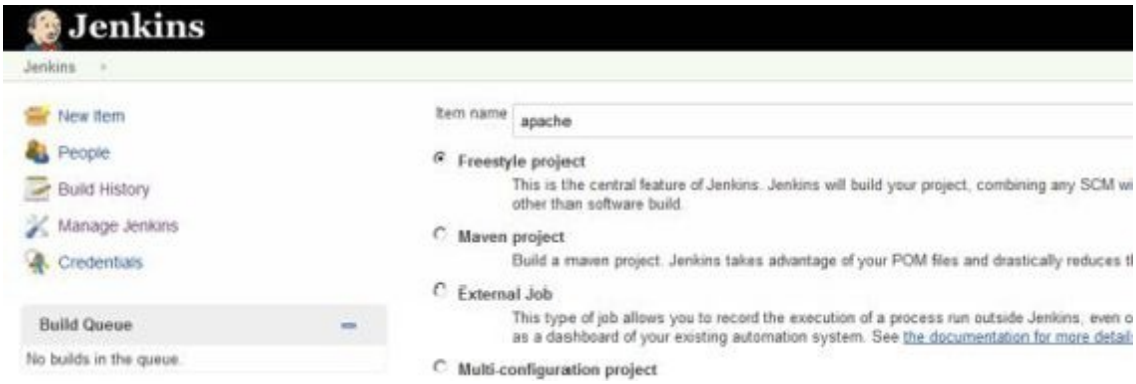
You need to have the following setup as the initial configuration our demonstration.

1. A git hub account with ssh keys configured with your development environment.
2. An apache repository on github with Docker file and files pushed from your development environment.

Configuring Dockerfile build Jenkins

Install git plugin on Jenkins server for configuring automatic Docker builds whenever the updated code and Dockerfile is pushed to github. You can install this plugin from “manage Jenkins” option in the Jenkins dashboard. Follow the steps given below to create a build job for apache container.

1. From the Jenkins dashboard, click “create new jobs” option and select the freestyle project and name it as apache.



2. Under source code management option, select git and copy the repository url for apache from github.

3. Under build triggers section, select “poll SCM” option. Here you can mention the interval for polling your github repository to check for code changes. If you provide all the values as stars, Jenkins will poll every one minute for checking the status of the github repository.

4. Under build section, select the “execute shell” option and copy the following shell script on to the text box.

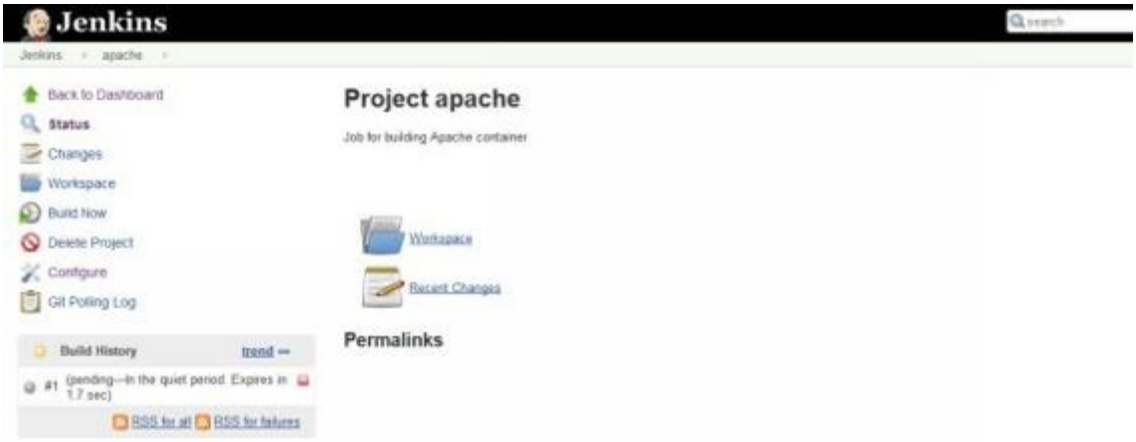
```
echo '>>> Getting the old containers id'
CID=$(sudo Docker ps | grep "apache-website" | awk '{print $1}')
echo $CID

echo '>>> Building new image from Dockerfile'
sudo Docker build -t="apache" . | tee /tmp/Docker_build_result.log
RESULT=$(cat /tmp/Docker_build_result.log | tail -n 1)
if [[ "$RESULT" != *Successfully* ]];
then
```

```
 exit -1
fi

echo '>>> Stopping old container'
if [ "$CID" != "" ];
then
 sudo Docker stop $CID
fi

echo '>>> Restarting Docker'
sudo service Docker.io restart
sleep 5

echo '>>> Starting new container'
sudo Docker run -p 80:80 -d apache

echo '>>> Cleaning up images'
sudo Docker images | grep "^<none>" | head -n 1 | awk
'BEGIN { FS = "[ \t]+" } { print $3 }' | while read -r id ;
do
sudo Docker rmi $id
done
```

Here is what the above shell script does,

1. Gets the old container id if any.
2. Builds an apache image from the Docker file.
3. Stops the old apache-website container if running.
4. Restarts the Docker service
5. Creates a new apache-website container with port 80 mapped on to the host.
6. Deletes all the intermediate images.

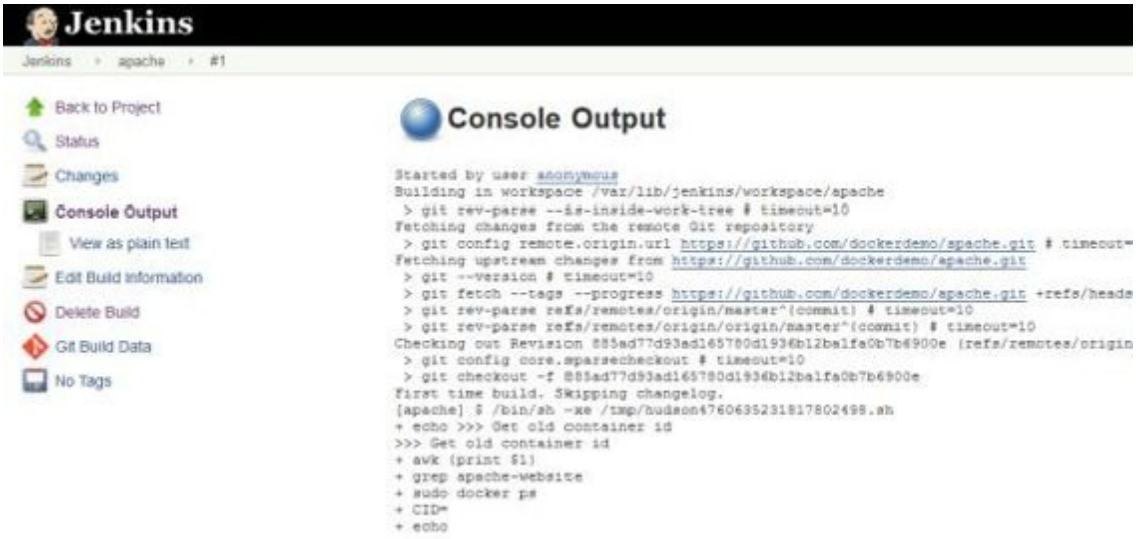
7. Click save and start the build process by clicking the “build now” option at the sidebar. Jenkins will then copy the Dockerfile and other contents from the github url you provided to its workspace. Once the build process starts, you can see the status from the build history option from the sidebar.



8. Once the build is complete, you can use the status option to check the status of your job. Blue button indicates a successful build and red indicated a failure as shown in the image below.

9. You can also check the console output using the “console output” option to see what the shell script has done at the backend. This

option is useful for debugging the build process by knowing what have gone wrong while executing the script.



As shown in the image above, our first build was success and you can view the application on port 80 using the web browser. You can test the build setup by updating the Dockerfile or website files and pushing it to github.

Jenkins will fetch the updated code to its workspace and builds a new image and creates a new container from it. You can see all the changes by viewing the application in the browser.

For continuous deployment, you can create a new job and trigger it based on the status of apache job. There are many ways for deployments.

For example,

1. You can use Chef Jenkins plugin to provision and configure a new instance with Docker host and deploy the successfully build Dockerfile.
2. You can push successfully built new image to docker hub and trigger docker pull from the deployment server.

9

Docker Provisioners

In this chapter we will learn how to provision Docker containers using tools like vagrant and chef.

Docker vagrant provisioner

Vagrant is an open source tool for creating repeatable development environment's using various operating systems. It uses providers to launch virtual machines. By default vagrant uses virtual box as its provider. Like boot2Docker, vagrant can run Docker on non-linux platforms.

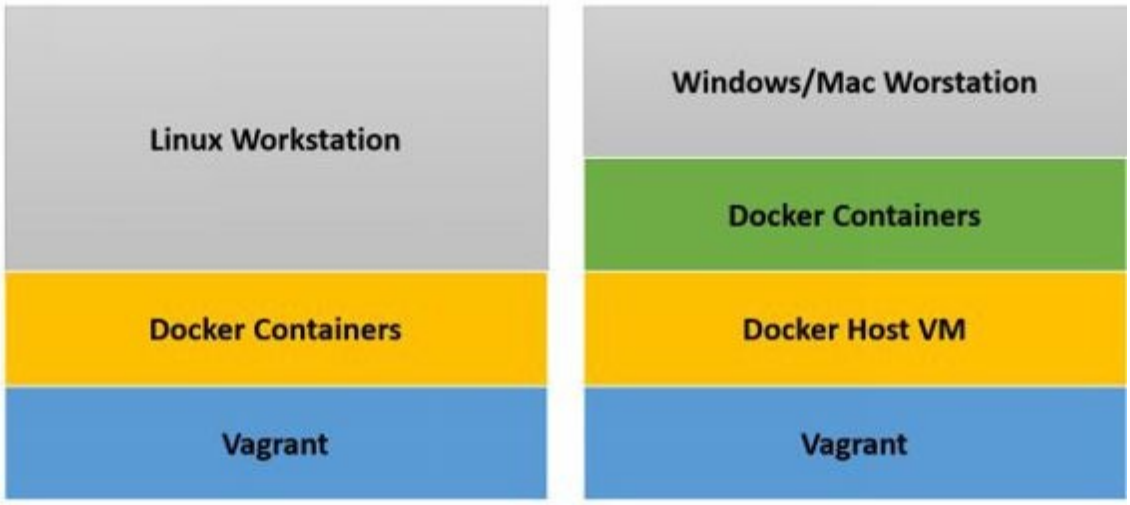


Fig 9-1 :vagrant , Docker architecture

Vagrant has several advantages over tools like boot2Docker. They are as follows.

1. Configure once and run anywhere: vagrant is a Docker wrapper which can run on any machine which supports Docker and in non-supported platforms , it will spin up a VM to deploy containers leaving users not to worry about if their system supports Docker or not.
2. In vagrant, the Docker host is not limited to a single distro like boot2Docker, it rather supports debian, centos, coreOS etc.
3. Vagrant can be used for Docker orchestration.

The Docker provisioner is used for automatically installing Docker, pull Docker images, and configure containers to run on boot.

Vagrant Docker provisioner is a best fit for teams using Docker in development and to build distributed application on it. Also if you are getting started with Docker, vagrant provides an efficient way to automate the container build and deployment process for your development environment.

Along with other vagrant provisioners, you can use Docker provisioner for your application testing by creating a better development workflow.

For example, you can use chef provisioner to install and configure your application and use Docker for the application runtime. You can use chef along with Docker provisioner.

Vagrantfile:

The main configuration for any Vagrant environment is a file called *Vagrantfile* which you need to place in your project's folder. Vagrantfile is a text file which holds all the provisioning configuration required for a project. Each project should have only one vagrant file for all configurations. Vagrantfile is portable and can be used with any system which supports vagrant. The configurations inside a vagrantfile follows ruby syntax but ruby knowledge is not required to create or modify a vagrantfile. It's a good practice to version the vagrantfile using a source control system.

Vagrant options:

Docker vagrant provisioner has various options. These options can be used to build and configure containers. If you do not use any option, vagrant will just install and set up Docker on your workstation. Let's look at the two main options provided by vagrant

1. Images: this option takes input in an array. You can provide a list of images you want it pull it down to your vagrant VM.
2. Version: you can specify the version of Docker you want install. By default it downloads and installs the latest version of Docker.

Apart from the above two options mentioned above, there are other options available for working with Docker.

Following are the options used for building, pulling and running Docker containers.

1. build_image: This option is used for building an image from the Docker file.
2. pull_images: This option is used for pulling images from the Docker hub.
3. Run: This option is used to run the container.

Let's have a look at those options in detail.

Building Images:

Images can be built automatically using the provisioner. Images have to be built before running a container. The syntax for building an image is shown below.

```
Vagrant.configure("2") do |config|
 config.vm.provision "Docker" do |d|
  d.build_image "/vagrant/app"
 end
end
```

build_image has an argument "/vagran/app" which is the path for the Docker build. This folder must exist in the guest machine.

Pulling images

Vagrant can automatically pull images to your Docker host. You can pull several images at a time. There are two ways to do that using arrays and the pull_image function. The syntax for using arrays is given below.

```
Vagrant.configure("2") do |config|
 config.vm.provision "Docker",
  images: ["centos"]
end
```

Syntax for pulling multiple images used pull_image function is given below.

```
Vagrant.configure("2") do |config|
 config.vm.provision "Docker" do |d|
  d.pull_images "fedora"
  d.pull_images "centos"
 end
end
```

Launching containers

After pulling images, vagrant can automatically provision containers from that image. The syntax for launching containers is shown below.

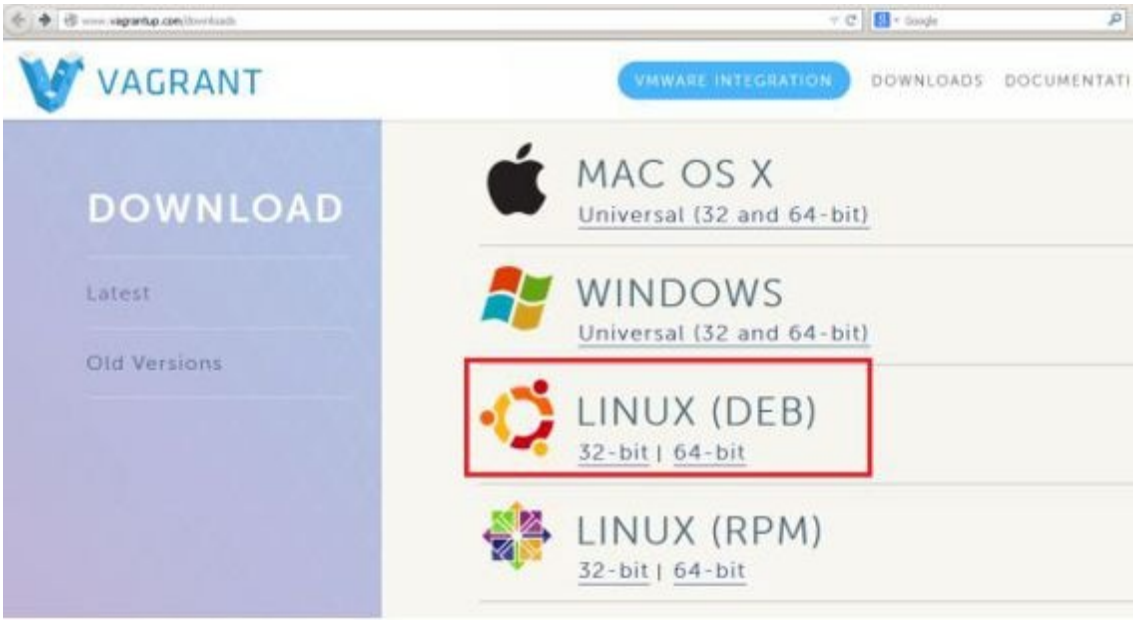
```
Vagrant.configure("2") do |config|
 config.vm.provision "Docker" do |d|
  d.run "redis"
 end
end
```

We have learnt the basic concepts and vagrant file functions for building and launching containers. Now let's look in to the practical way of building containers using vagrant.

Installing vagrant on Ubuntu:

Follow the steps give below to install vagrant on an Ubuntu machine.

1. Head over to http://www.vagrantup.com/downloads



2. Get the download link for Ubuntu 64 bit and download the vagrant installation file

```
root@dockerdemo:~# wget https://dl.bintray.com/mitchellh/vagrant/vagrant_1.6.5_x86_64.deb
--2014-09-18 12:36:51--  https://dl.bintray.com/mitchellh/vagrant/vagrant_1.6.5_x86_64.deb
Resolving dl.bintray.com (dl.bintray.com)... 5.153.24.114
Connecting to dl.bintray.com (dl.bintray.com)|5.153.24.114|:443... connected.
HTTP request sent, awaiting response... 302
```

3. Install the downloaded package.

dpkg -i vagrant_1.6.5_x86_64.deb

```
root@dockerdemo:~# ls
vagrant_1.6.5_x86_64.deb
root@dockerdemo:~# dpkg -i vagrant_1.6.5_x86_64.deb
Selecting previously unselected package vagrant.
(Reading database ... 51927 files and directories currently installed.)
Preparing to unpack vagrant_1.6.5_x86_64.deb ...
Unpacking vagrant (1:1.6.5) ...
Setting up vagrant (1:1.6.5) ...
```

Note: the latest version of Docker comes bundled with Docker provider, so you don't have to install the provider specifically. Also if you are running vagrant Docker in non-linux platforms like MAC, vagrant has the ability to find it automatically and it will create a virtual environment to run Docker containers. This will happen only once and for the subsequent vagrant runs it will make use of already created virtual environment.

Now let's create a vagrant file to work with Docker images and containers.

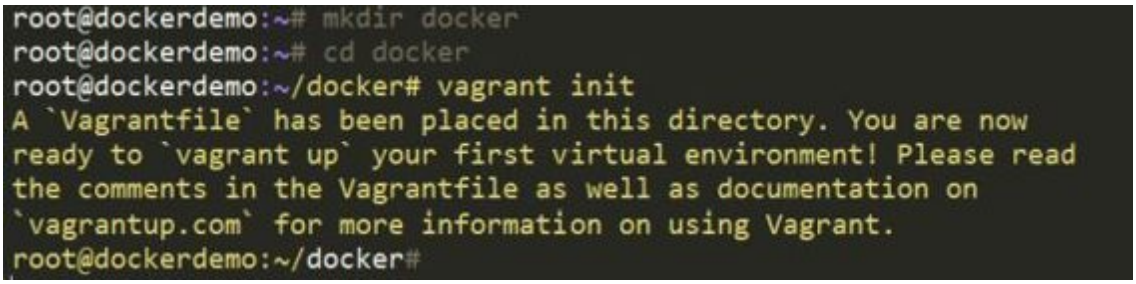
Creating a vagrant file

The configuration for building images and containers are mentioned in the vagrant file.

Follow the steps mentioned below to create a vagrant file.

1. Create a directory, say Docker
2. CD in to the directory and create and file called Vagrantfile or use can use "vagrant init" command for creating a vagrant file

vagrant init



Vagrant file configuration for Docker

In this demo we will provision a web and db Docker containers using the vagrant file. Web container will be built using the Docker file we created in the above section and the db container will be built using the public image from Docker registry.

Open the Vagrantfile, delete all the contents inside that file because we won't be using any other provisioners or virtual machines except Docker.

Copy the configurations mentioned below to the vagrantfile.

```
Vagrant.configure("2") do |config|
 config.vm.define "web" do |app|
  app.vm.provider "Docker" do |d|
   d.image = "olibuijr/ubuntu_apache2"
   d.link "db:db"
  end
 end
config.vm.define "db" do |app|
  app.vm.provider "Docker" do |d|
   d.image = "paintedfox/postgresql"
   d.name = "db"
  end
 end
end
```

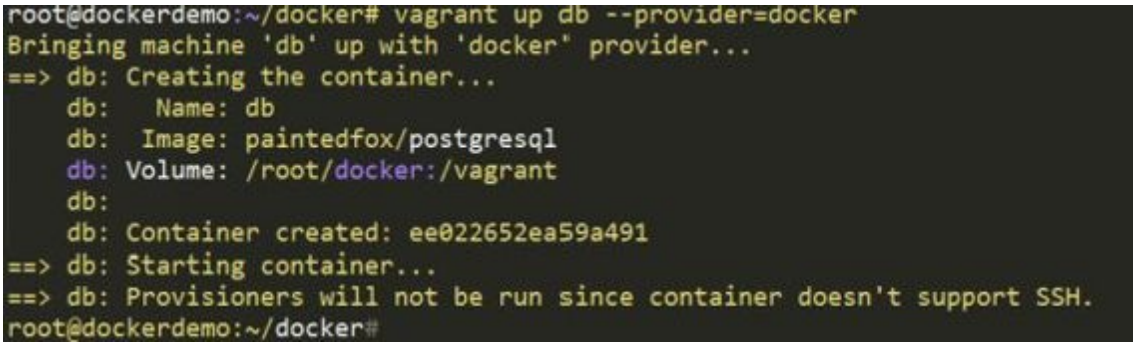
Note that the above two vagrant configurations are for a web and db container which will be linked together.

Building db image using vagrant

Now we have our vagrant file ready to build two images. Let's build the db image first so that the web image can be linked to db.

1. Run the following vagrant command to provision the db image.

```
vagrant up db
```



2. Now run the following command to provision the web image

vagrant up web

```
root@dockerdemo:~/docker# vagrant up web
Bringing machine 'web' up with 'docker' provider...
==> web: Creating the container...
    web:   Name: web
    web:  Image: olibuijr/ubuntu_apache2
    web: Volume: /root/docker:/vagrant
    web:   Link: db:db
    web:
    web: Container created: 1a9ba2f468076e1f
==> web: Starting container...
==> web: Provisioners will not be run since container doesn't support SSH.
```

Now we have two container's running, created from two different images, one web container with apache and another db container with postgres and linked it together for db connection.

3. Run Docker ps to view the running containers we launched using vagrant.

```
root@dockerdemo:~/docker# docker ps
CONTAINER ID        IMAGE                              COMMAND
CREATED             STATUS              PORTS                NAMES
1a9ba2f46807        olibuijr/ubuntu_apache2:latest     /usr/sbin/apache2 -D
18 minutes ago      Up 18 minutes       80/tcp               web
37727d31ed1b        paintedfox/postgresql:latest       /sbin/my_init
54 minutes ago      Up 54 minutes       5432/tcp             db,web/db
```

Vagrant commands

There are three vagrant specific commands for Docker to interact with the containers.

1. Vagrant Docker-logs: Using this command you can view the logs of a running containers.

vagrant docker-logs

```
root@dockerdemo:~/docker# vagrant docker-logs
==> web: apache2: Could not reliably determine the server's fully qualified domain name,
==> db: *** Running /etc/rc.local...
==> db: POSTGRES_USER=super
==> db: POSTGRES_PASS=md4SHdUjo2I0Jaa5
==> db: POSTGRES_DATA_DIR=/data
==> db: Initializing PostgreSQL at /data
==> db: Starting PostgreSQL...
==> db: Creating the superuser: super
==> db: *** Booting runit daemon...
==> db: *** Runit started as PID 13
==> db: 2014-09-19 08:55:42 UTC LOG:  database system was shut down at 2014-09-08 16:55:5
==> db: 2014-09-19 08:55:42 UTC LOG:  database system is ready to accept connections
==> db: 2014-09-19 08:55:42 UTC LOG:  autovacuum launcher started
==> db: NOTICE:  role "super" does not exist, skipping
```

2. Vagrant Docker-run: This commad is used to run commands on a container. Syntax to run this command is shown below.

vagrant docker run db – echo " This is a test"

```
root@dockerdemo:~/docker# vagrant docker-run  db  -- echo " this is a test "
==> db: Creating the container...
    db:   Name: db_1411121202
    db:  Image: paintedfox/postgresql
    db:    Cmd: echo  this is a test
    db: Volume: /root/docker:/vagrant
    db:
    db: Container is starting. Output will stream in below...
    db:
    db:  this is a test
root@dockerdemo:~/docker#
```

Managing Docker using chef

Chef along with Docker can be used for the following,

1. Creating Docker images and deploying containers.
2. To configure Docker containers during boot.
3. Setting up a Docker host

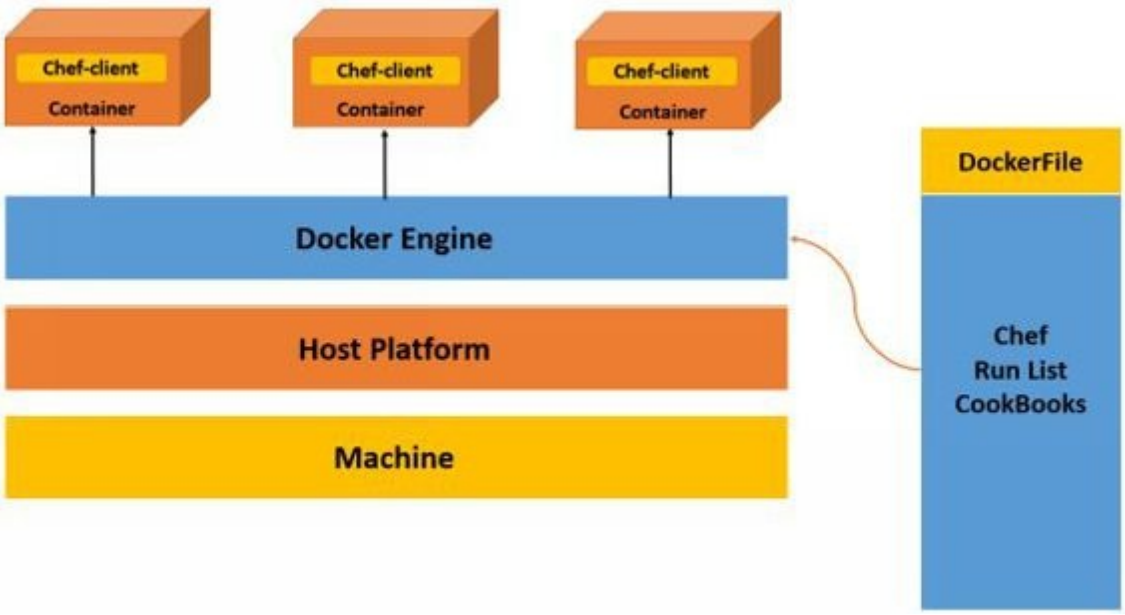


Fig 9-2 :Chef , Docker architecture

There are two main components of chef for managing Docker containers.

1. Chef-container
2. Knife-container

Chef-container:

Chef-container is a version of chef client which can run inside a Docker container. Chef-container uses runit and chef-init as the init system and container entry point. Chef container can configure a container as we configure any other piece of software.

Knife container:

Knife container is a knife plugin for building and managing Docker containers using chef. To manage Docker with chef, you need to have the latest version of chef client and chefdk installed on your host.

Follow the steps given below to manage Docker containers using chef.

1. Install knife-container using the following command.

chef gem install knife-container

```
root@node2:~# chef gem install knife-container
Fetching: knife-container-0.2.4.gem (100%)
Successfully installed knife-container-0.2.4
Parsing documentation for knife-container-0.2.4
Installing ri documentation for knife-container-0.2.4
Done installing documentation for knife-container after 0 seconds
1 gem installed
root@node2:~#
```

2. You have to create a Docker context to initialize all the necessary configurations for a Docker image. In this demo we will create a context for demo/apache2 image and which will use the default Ubuntu:latest image from the Docker index.
3. If you want another image, you need to override the default configuration in the knife.rb file using the knife[:berksfile_source] parameter. Create the Docker context for out demo/apache2 image using the following command.

knife container docker init demo/apache2 -r 'recipe[apache2]' -z –b

```
root@node2:~/chef-repo# knife container docker init demo/apache2 -r 'recipe[apache2]'
Compiling Cookbooks...
Recipe: knife_container::docker_init
  * directory[/var/chef/demo/apache2] action create
    - create new directory /var/chef/demo/apache2
  * template[/var/chef/demo/apache2/Dockerfile] action create
    - create new file /var/chef/demo/apache2/Dockerfile
    - update content in file /var/chef/demo/apache2/Dockerfile from none to 4c70a7
    (diff output suppressed by config)
 run (skipped due to not_if)
Downloading base image: chef/ubuntu-12.04:latest. This process may take awhile...
.
.
Tagging base image chef/ubuntu-12.04 as demo/apache2
Context Created: /var/chef/demo/apache2
```

4. Open the first-boot.JSON file from /var/chef/demo/apache2/chef and add the following to the JSON file.

```
"container_service": {
"apache2": {
"command": "/usr/sbin/apache2 -k start"
}
}
```

The final first-book.JSON file should look like the following.

```
{
"run_list": [
"recipe[apache2]"
],
"container_service": {
"apache2": {
"command": "/usr/sbin/apache2 -k start"
}
}
}
```

5. Now we have the configuration file ready for building the demo/apache2 image. The cookbook apache2 will be downloaded from the chef marketplace with all the dependencies solved using berkshelf.
6. You can also have your own cookbook configured in the chef-repo cookbooks directory. Run the following command to build the image with chef-container configured.

```
knife container docker build demo/apache2
```

```
root@node2:~/chef-repo/.chef# knife container docker build demo/apache2
[2014-10-29T12:42:52+00:00] INFO: Storing updated cookbooks/pacman/README.md in the c
[2014-10-29T12:42:52+00:00] INFO: Storing updated cookbooks/pacman/metadata.json in t
[2014-10-29T12:42:52+00:00] INFO: Storing updated cookbooks/pacman/Gemfile in the cac
[2014-10-29T12:42:52+00:00] INFO: Storing updated cookbooks/pacman/recipes/default.rb
[2014-10-29T12:42:52+00:00] INFO: Processing package[apache2] action install (apache2
[2014-10-29T12:43:14+00:00] INFO: Deleting client key...
 ---> f4c4992d2cfc
Removing intermediate container 6a3fd4276a6f
Step 3 : RUN rm -rf /etc/chef/secure/*
 ---> Running in a5a4d31f8103
Step 5 : CMD ["--onboot"]
 ---> Running in c093033f2272
 ---> a18d6589209e
Removing intermediate container c093033f2272
Successfully built a18d6589209e
```

7. Now we have our image built and you can view the image using the following Docker command.

docker images demo/apache2

```
root@node2:~/chef-repo/.chef# docker images demo/apache2
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTU
demo/apache2        latest              a18d6589209e        2 minutes ago       350.4
root@node2:~/chef-repo/.chef#
```

8. Create an apache2 container from demo/apahce2 image using the following command.

docker run -d --name apache2 demo/apache2

```
root@node2:~/chef-repo/.chef# docker run -d --name apache2 demo/apache2
0b37a0b0bec366304b1a4c477097c56c4efa135da27d5a3276a0018e38cc654f
root@node2:~/chef-repo/.chef#
```

9. If you run a Docker ps command you can view the running apache2 container.

docker ps

```
root@node2:~/chef-repo/.chef# docker ps
CONTAINER ID        IMAGE                 COMMAND             CREATED             
0b37a0b0bec3        demo/apache2:latest   chef-init --onboot  About a minute ago  
root@node2:~/chef-repo/.chef#
```

10. You can check the process running inside the apache container using the following command.

docker top apache2

10
Docker Deployment Tools

In this chapter we will learn about Docker deployment tools like fig, shipyard and panamax.

Fig

Fig is tool for running development environments using Docker specifically for projects which includes multiple containers with connections and can also be used in production environments. Using fig you can have fast isolated Docker environments which can be reproduced anywhere.

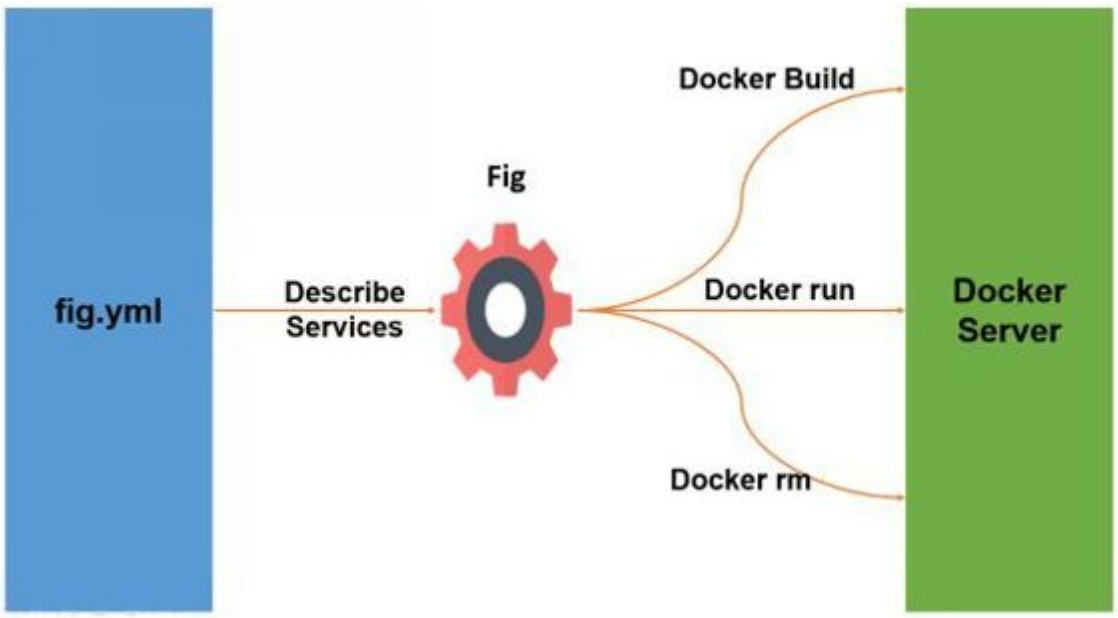


Fig 10-1 :Fig , Docker architecture

For example, if you want to build your images with code, all you need to do is, create a fig.yml with all the container links and run it on a Docker host. Fig will automatically deploy all containers with links specified in the fig.yml file. Also, all the fig managed applications have their own lifecycle. Eg: build, run, stop and scale.

Let's get started with installing fig on a Docker host.

Installing fig:

Fig works with Docker 1.0 or later. In this section, we will install fig on Ubuntu 14.04 64 bit server.

Note: Make sure that you have Docker 1.0 or later installed and configured on the server.

Follow the steps give below to install and configure fig.

1. We will install fig using the binary from github using curl. Execute the following command to download and install fig from github source.

curl -L https://github.com/docker/fig/releases/download/0.5.2/linux > /usr/local/bin/fig

```
root@dockerdemo:~# curl -L https://github.com/docker/fig/releases/download/0.5.2/linu
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   383    0   383    0     0    344      0 --:--:--  0:00:01 --:--:--   344
100 5004k  100 5004k    0     0   710k      0  0:00:07  0:00:07 --:--:-- 1079k
```

The above command installed fig on /usr/local/bin/fig directory.

2. Change the read write permissions for that installation folder using the following command.

chmod +x /usr/local/bin/fig

```
root@dockerdemo:~# chmod +x /usr/local/bin/fig
root@dockerdemo:~#
```

3. To ensure that fig installed as expected, run the following command to check fig's version.

fig --version

```
root@dockerdemo:~# fig --version
fig 0.5.2
root@dockerdemo:~#
```

Fig.yml

All the services that have to be deployed using a container are declared as YAML hashes in the Fig.yml file.

Each service should have an image or build specification associated with it. Parameters inside each service are optional and they are analogous to Docker run commands.

Fig.yml reference

Fig.yml has various options. We will look in to each option associated with fig.yml file.

Image

This option is mandatory for every service specified in the yml file. Image can be private or public. If the image is not present locally, fig will pull the image

automatically from the public repository. Image can be defined in the following formats.

```
image: centos
image: bibinwilson/squid
image: a5fj7d8
```

build

This option is used when you build images from a Docker file. You need to provide the path to the Docker file in this option as shown below.

```
build: /path/to/build/dir
```

Command

This option is to run commands on the image. It has the following syntax.

```
command: < command to be run >
```

links

This option is used to link containers to another service. E.g.: linking a web container to a database container. This option has the following syntax.

```
links:
- db
- db:database
- postgres
```

Ports

This option exposes the port on a container. You can specify which host port has to be associated with the container port or you can leave the host port empty and a random port will be chosen for host to container mapping. This option has the following forms.

```
ports:
- "8080"
- "80:8080"
- "3458:21"
- "127.0.0.1:80:8080"
```

Expose

The ports specified in this option are internal to a container and can only be accessed by linked services. The exposed ports are not associated with the host. It has the following syntax

```
expose:
- "3000"
- "8000"
```

Volumes

This option is used to mount host folders as volumes on a container. This option has the following syntax.

```
volumes:
- /var/www/myapp
- myapp/:/var/www/myapp
```

volumes_from

This option is used to mount volumes from containers in other services. It has the following syntax.

```
volumes_from:
- service_name
```

```
- container_name
```

Environment

This option is used to set environment variables for a container. You can specify this either using an array or dictionary. It has the following syntax.

```
environment:
 FB_USER:usernname
 PASSWORD_SECRET: S3CR3T
environment:
 - FB_USER = usernname
 - PASSWORD_SECRET = S3CR3T
```

Deploying rails application using Fig:

In this section we will look in to rails application deployment using Fig.

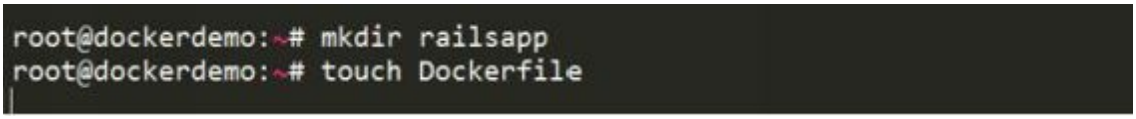
For rails application setup, you need a Docker image configured with rails environment to create our web container. For this you need a Docker file with image configurations. So, let us create a Docker file to build an image for our rails application.

1. Create a directory, say railsapp.

```
mkdir railsapp
```

2. CD in to the directory and create a Docker file.

```
touch Dockerfile
```



3. Open the Docker file and copy the following contents

```
FROM ruby
RUN apt-get update -qq && apt-get install -y build-essential libpq-dev
RUN mkdir /myapp
WORKDIR /myapp
ADD Gemfile /myapp/Gemfile
RUN bundle install
ADD . /myapp
```

The above Dockerfile installs the development environment for rails on a ruby image from Docker hub. We don't have to install ruby because the ruby image comes bundled with ruby environment for rails application. Also we are creating a myapp folder to put out rails code.

4. We need a gem file for the initial configuration and it will be later overwritten by the app. Create a gemfile in the railsapp directory and copy the following contents to it.

```
source 'https://rubygems.org'
gem 'rails', '4.0.2'
```

5. Let's create a fig.yml file for our rails application.

```
touch fig.yml
```

6. Open the file and copy the following fig configurations for the rails application.

```
db:
    image: postgres
    ports:
    "5432"
web:
    build: .
    command: bundle exec rackup -p 3000
    volumes:
        .:/myapp
```

```
    ports:
        "3000:3000"
    links:
        db
```

If you look at the above file, we have two services one web service and one db service.

These services are declared in YAML hashes.

Db service:

```
db:
image: postgres
ports:
"5432"
```

Db service uses the postgres public Docker image and exposes port 5432.

Web service:

```
web:
    build: .
    command: bundle exec rackup -p 3000
    volumes:
    .:/myapp
    ports:
    "3000:3000"
    links:
    db
```

Web service builds the web Docker image from the Docker file we created in step 1. (build: . looks for Docker file in current directory). Also it creates a myapp folder in the containers which will be mounted to the current directory where we will have the rails code.

7. Now we have to pull postgres and ruby images to configure out app using fig. Execute the following command to pull the images specified in the fig file and to create a new rails application on the web container.

```
fig run web rails new . --force --database=postgresql --skip-bundle
```

Note: We will look in to all fig commands later in this section.

```
Creating railsapp_db_1...
Pulling image postgres...
build-essential is already the newest version.
libpq-dev is already the newest version.
Use 'apt-get autoremove' to remove it.
0 upgraded, 0 newly installed, 0 to remove and 76 not upgraded.
 ---> 4c5f5722de86
Removing intermediate container ff315547b20c
Step 2 : RUN mkdir /myapp
 ---> Running in 16cab894a165
 ---> 7b6f58403d76
Removing intermediate container 16cab894a165
Step 3 : WORKDIR /myapp
 ---> Running in 8d92c14c266e
 ---> 282cbdf003fb
Removing intermediate container 8d92c14c266e
Step 4 : ADD Gemfile /myapp/Gemfile
Removing intermediate container 04314eca90fb
Step 5 : RUN bundle install
 ---> Running in 362ebdde973d
```

8. Once the above command executed successfully, you can view the new rails app created in the railsapp folder. This will be mounted to myapp folder of the web container.

```
root@dockerdemo:~/railsapp# ls
app  config     db          fig.yml  lib  public    README.rdoc  tmp
bin  config.ru  Dockerfile  Gemfile  log  Rakefile  test         vendor
root@dockerdemo:~/railsapp#
```

9. Now we have to uncomment the rubytracer gem in the gemfile to get the javascript runtime and rebuild the image using the following fig command.

```
fig build
```

```
root@dockerdemo:~/railsapp# fig build
db uses an image, skipping
Building web...
 ---> 5cd733bb7b03
Step 1 : RUN apt-get update -qq && apt-get install -y build-essential libpq-dev
 ---> Using cache
 ---> 4c5f5722de86
Step 2 : RUN mkdir /myapp
 ---> Using cache
 ---> 7b6f58403d76
Step 3 : WORKDIR /myapp
 ---> Using cache
 ---> 282cbdf003fb
Step 4 : ADD Gemfile /myapp/Gemfile
 ---> f5b7b8dbf19c
Removing intermediate container 17abaddf5e4f
Step 5 : RUN bundle install
 ---> Running in a16d344de2de
Removing intermediate container a16d344de2de
Step 6 : ADD . /myapp
 ---> 65b8b1854ff7
Removing intermediate container c540e6474e58
Successfully built 65b8b1854ff7
```

10. The new rails app has to be connected to the postgres database, so edit the database.yml file to change the host to db container db_1. Replace all the entries with the following configuration.

```
development: &default
adapter: postgresql
encoding: unicode
database: postgres
pool: 5
username: postgres
password:
host: db_1
test:
<<: *default
database: myapp_test
```

11. Now we have everything in place and we can boot up the rails application using the following fig command.

```
fig up
```

```
root@dockerdemo:~/railsapp# fig up
Recreating railsapp_db_1...
Recreating railsapp_web_1...
Attaching to railsapp_db_1, railsapp_web_1
web_1 | [2014-09-30 12:52:24] INFO  WEBrick 1.3.1
web_1 | [2014-09-30 12:52:24] INFO  ruby 2.1.3 (2014-09-19) [x86_64-linux]
web_1 | [2014-09-30 12:52:24] INFO  WEBrick::HTTPServer#start: pid=1 port=3000
web_1 | 14.98.245.32 - - [30/Sep/2014 12:52:31] "GET / HTTP/1.1" 200 - 0.1673
```

12. Open a new terminal and create the db using following command.

fig run web rake db:create

13. Now you have a rails application up and running. You can access the application on port 3000 from your host ip.

We have built a two container rails application using Rails and Postgresql.

Now we will see how to set up a four container complex auto load balancing application using HAproxy, Serf , Apache and MySQL.

What is Serf?

Serf is a cluster membership tool which is decentralized, highly available and fault tolerant. Serf works on gossip protocol. For example, serf can be used for scaling web servers under a load balancer to maintain the list of web servers under a load balancer. Serf attains this by maintaining a cluster membership list and wherever membership changes it runs handler scripts to register or deregister a webserver from the load balancer.

We will be using serf in our auto load balancing fig application to register and deregister the web server containers under HAproxy.

Deploying four container Auto load balancing application using Fig

In this section we will deploy a four container application using fig. Our application will run wordpress with mysql database with HAproxy as a load balancer.

Note: We will be using preconfigured images from the Docker registry which contains prebuilt serf configurations. You can also build your own images with serf configurations.

Create a fig.yml file and copy the following contents for launching our auto load balancing application.

```
serf:
 image: ctlc/serf
 ports:
  - 7373
  - 7946
lb:
 image: ctlc/haproxy-serf
 ports:
  - 80:80
 links:
  - serf
 environment:
  HAPROXY_PASSWORD: qa1N76pWAri9
web:
 image: ctlc/wordpress-serf
 ports:
  - 80
 environment:
  DB_PASSWORD: qa1N76pWAri9
 links:
  - serf
  - db
```

```
 volumes:
  - /root/wordpress:/app
db:
 image: ctlc/mysql
 ports:
  - 3306
 volumes:
  - /mysql:/var/lib/mysql
 environment:
  MYSQL_DATABASE: wordpress
  MYSQL_ROOT_PASSWORD: qa1N76pWAri9
```

The above yml file has service description for serf, HAproxy (lb), wordpress (web), mysql (db).

For wordpress to be available for installation, you need to have the wordpress set up file in your fig host. Then we will mount that wordpress folder to the /app folder of web container. In the yml file, we have mentioned it as follows.

```
volumes:
  - /root/wordpress:/app
```

We have put the wordpress files in root folder. It can be anywhere inside your Docker host containing fig. Follow the steps give below to set up the application

1. Once you have the fig.yml file ready, you can launch the containers using the following command.

```
fig up –d
```

Note: Make sure you are running the above command from the directory where you have the fig.yml file.

```
root@dockerdemo:~/wordpress1# fig up -d
Creating wordpress1_serf_1...
Creating wordpress1_db_1...
Creating wordpress1_web_1...
Creating wordpress1_lb_1...
root@dockerdemo:~/wordpress1#
```

2. After launching the containers, you can view the wordpress configuration in your browser using the IP of your Docker host.

Welcome to WordPress. Before getting started, we need some information on the database. You will need to know the following items before proceeding.

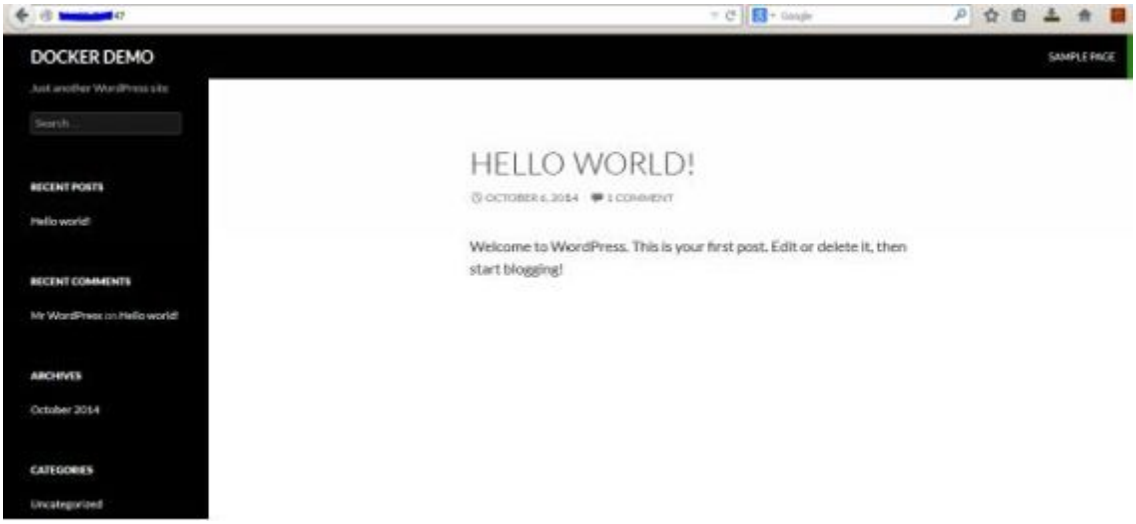
1. Database name
2. Database username
3. Database password
4. Database host
5. Table prefix (if you want to run more than one WordPress in a single database)

We're going to use this information to create a `wp-config.php` file. **If for any reason this automatic file creation doesn't work, don't worry. All this does is fill in the database information to a configuration file. You may also simply open `wp-config-sample.php` in a text editor, fill in your information, and save it as `wp-config.php`.** Need more help? We got it.

In all likelihood, these items were supplied to you by your Web Host. If you do not have this information, then you will need to contact them before you can continue. If you're all ready...

Let's go!

3. Continue with the normal wordpress installation with the credentials we have in the fig.yml file. In the host entry use db_1. Once installed you will have a running wordpress application.



4. Now you can scale up and scale down the web containers and the newly launched containers will be automatically be registered to HAproxy using serf. To scale up the web containers, run the following fig command.

fig scale web=3

```
root@dockerdemo:~/wordpress1# fig scale web=3
Starting wordpress1_web_2...
Starting wordpress1_web_3...
root@dockerdemo:~/wordpress1#
```

5. You can stop all the running containers of auto load balancing application using fig kill command.

fig kill

```
root@dockerdemo:~/wordpress1# fig kill
Killing wordpress1_lb_1...
Killing wordpress1_web_3...
Killing wordpress1_web_2...
Killing wordpress1_web_1...
Killing wordpress1_db_1...
Killing wordpress1_serf_1...
```

6. Once stopped, you can remove all the containers using fig rm command.

fig rm

```
root@dockerdemo:~/wordpress1# fig rm
Going to remove wordpress1_serf_1, wordpress1_db_1, wordpress1_web_3, wordpress1_web
Are you sure? [yN] y
Removing wordpress1_serf_1...
Removing wordpress1_db_1...
Removing wordpress1_web_3...
Removing wordpress1_web_2...
Removing wordpress1_web_1...
Removing wordpress1_lb_1...
```

Shipyard

Shipyard is a simple Docker UI build on Docker cluster manager citadel. Using shipyard, you can deploy Docker containers from a web interface. Also shipyard can be used with Jenkins for managing containers during the build process. This is useful because failed containers may eat up you disk space. Shipyard is a very light weight application without any dependencies. You can host it on any server as a Docker client UI for managing containers locally and remotely.

Setting up shipyard

Shipyard has two components.

1. RethinkDb and
2. API

Both components are available in form of containers from Docker hub.

RethinkDB

1. Run the following command to set up rethinkDB container on your Docker Host.

docker run -it -P -d --name rethinkdb shipyard/rethinkdb

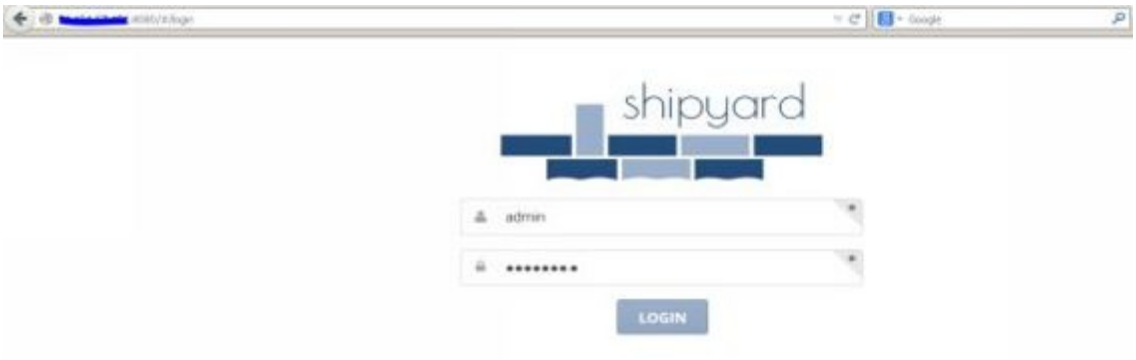
```
root@dockerdemo:~# docker run -it -P -d --name rethinkdb shipyard/rethinkdb
Unable to find image 'shipyard/rethinkdb' locally
Pulling repository shipyard/rethinkdb
e71002615a56: Pulling image (latest) from shipyard/rethinkdb, endpoint: https://e71002615a56:
511136ea3c5a: Download complete
1c9383292a8f: Download complete
9942dd43ff21: Download complete
d92c3c92fa73: Download complete
0ea0d582fd90: Download complete
46a08a0745aa: Download complete
58c9293ca066941b7bc9ee61424f3aa2b7142c3f6b4e4678e9b10d8436ca629c
```

2. API container on the Docker host gets links to rethinkDB. In this demo we are using single host setup, so we have to bind the API container with the Docker socket.
3. Run the following command on your Docker host to create the API container and bind it with the Docker socket.

*docker run -it -p 8080:8080 -d *
*-v /var/run/Docker.sock:/Docker.sock *
*--name shipyard --link shipyard-rethinkdb:rethinkdb *
shipyard/shipyard

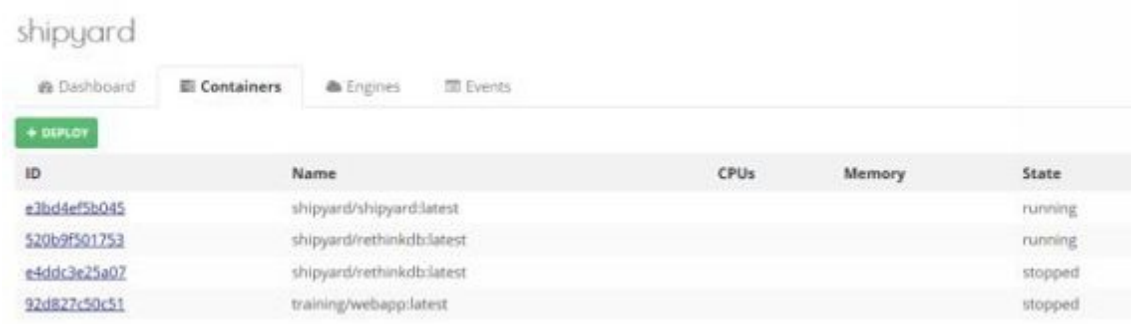
```
root@dockerdemo:~# docker run -it -p 8080:8080 -d \
>     -v /var/run/docker.sock:/docker.sock \
>     --name shipyard --link shipyard-rethinkdb:rethinkdb \
>     shipyard/shipyard
e3bd4ef5b045d2f77126bde17bc34f2484229b48fcda1c63539ce212d2f2f783
root@dockerdemo:~#
```

4. Now you will be able to access the shipyard dashboard on port 8080 of your Docker host.



Note: By default shipyard creates user "admin" and password "shipyard" to login to the application.

5. Once you are logged in, you will be able to view all the containers on your Docker host.



Deploying a container

You can launch a container form shipyard dashboard using the deploy option under containers.

Click the deploy option and fill in the parameters for your new container. Parameters involved in container deployment are explained below.

You have the following options in shipyard while deploying the container.

Name

Name of the image from which the container should be deployed

CPUs

CPU resource required for the container.

Memory

Memory required for the container.

Type

There are three types of containers in shipyard,

1. service,
2. unique
3. Host.

Service containers use the labels used by engines in the container host. Unique containers will be launched only if there are no instances of containers available on that host. Host will deploy a container on the specified host.

Hostname

Hostname sets the hostname for the container.

Domain

The Domain option sets the domain name for the container.

Env

This parameter is used to set environment variables for the container.

Arg

This option is used to pass arguments for the container.

Label

Named labels are used for container scheduling

Port

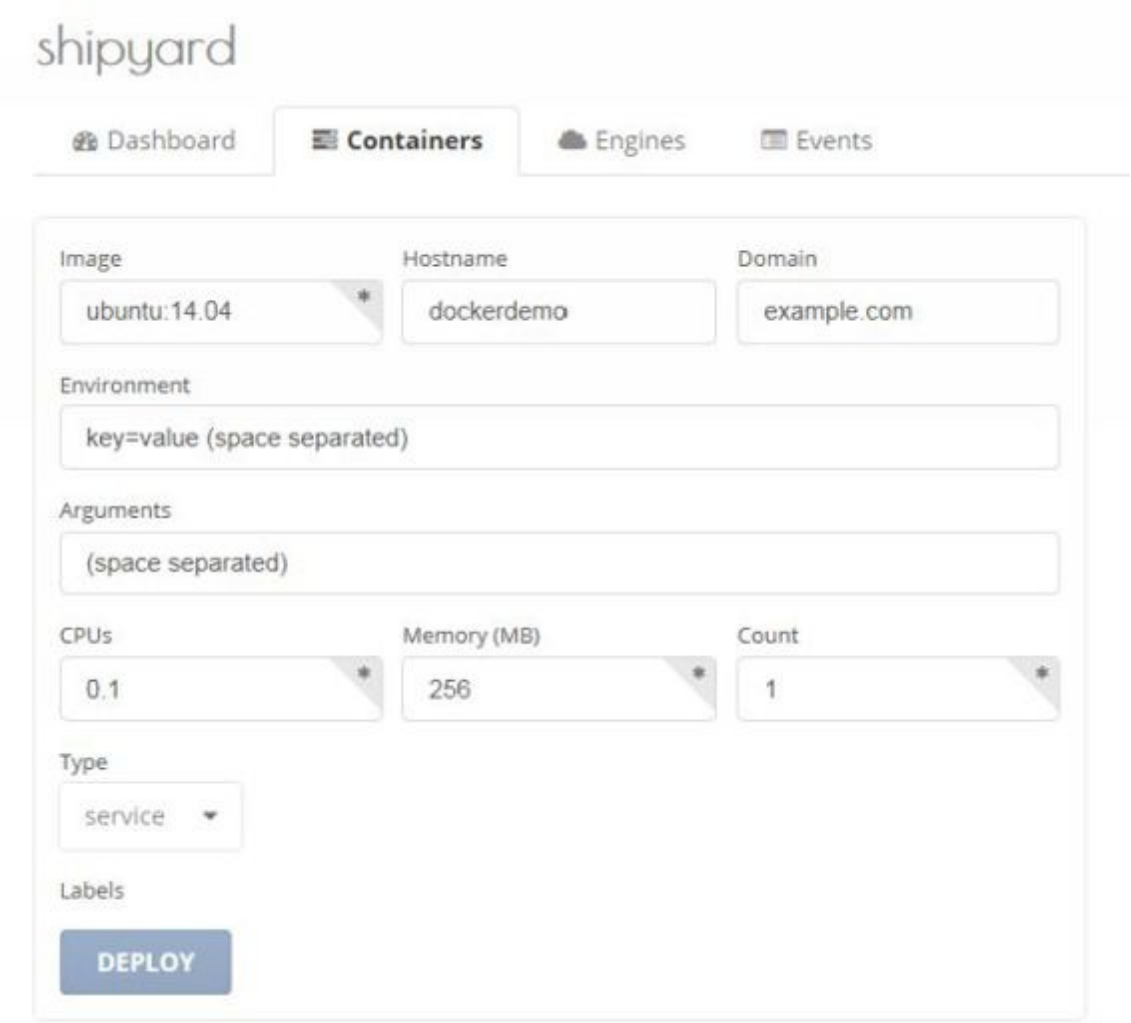
Port determines the ports which have to be exposed on a container.

Pull

This option is used where you have to pull the latest container image from the container hub.

Count

This determines the number of containers to be launched on a deploy process.



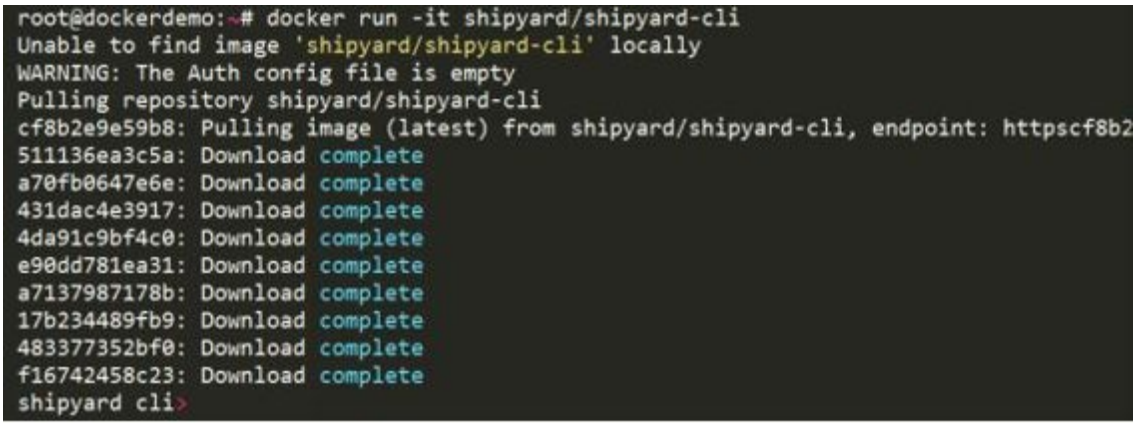
Enter the necessary parameters as shown in the image above and click the deploy option to deploy the container.

Deploying containers with shipyard CLI

Containers can be deployed using the shipyard cli. In order to use cli, you have to launch an instance of shipyard cli container.

1. Launch a shipyard cli instance using the following command.

docker run -it shipyard/shipyard-cli



2. You can view the list of available shipyard command using the following command.

shipyard help

Working with cli

1. Login to shipyard using the following command.

shipyard login

```
shipyard cli> shipyard login
URL: http://54.58.271.26:8080
Username: admin
Password:
shipyard cli>
```

2. View the containers in your Docker host using the following command

shipyard containers

```
shipyard cli> shipyard containers
ID              Image                          Name                  Host
d465329a39ea    shipyard/shipyard-cli:latest   distracted_babbage    dockerdemo
0f43a62e6b94    ubuntu:14.04                   high_ardinghelli      dockerdemo
e3bd4ef5b045    shipyard/shipyard:latest       shipyard              dockerdemo
520b9f501753    shipyard/rethinkdb:latest      shipyard-rethinkdb    dockerdemo
e4ddc3e25a07    shipyard/rethinkdb:latest      shipyard-rethinkdb-data dockerdemo
92d827c50c51    training/webapp:latest         evil_lalande          dockerdemo
be726c758377    centos:centos7                 jolly_euclid          dockerdemo
a109c69e1d88    ubuntu:latest                  hungry_goodall        dockerdemo
0fe8efdbe8df    ubuntu:latest                  mad_newton            dockerdemo
c07b368b9b90    ubuntu:latest                  drunk_fermat          dockerdemo
eb5b48ff5b2a    ubuntu:latest                  ubuntu1               dockerdemo
51839be5dbc7    ubuntu:latest                  romantic_heisenberg   dockerdemo
296a2fad8146    ubuntu:latest                  furious_einstein      dockerdemo
c83cd43eb85f    ubuntu:latest                  trusting_archimedes   dockerdemo
1a2ff1406d35    ubuntu:latest                  boring_kirch          dockerdemo
```

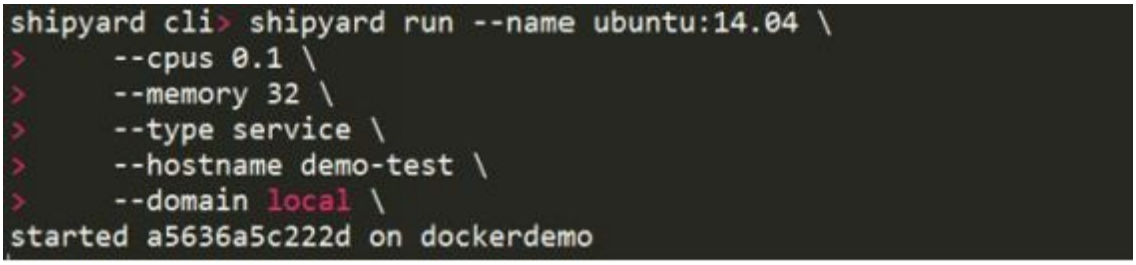
3. You can inspect a container using the following command along with the container id.

shipyard inspect d465329a39ea

```
shipyard cli> shipyard inspect d465329a39ea
{
    "id": "d465329a39ea2eb5beac43ea5c43e1c334823da4bcee954f820dcfac81d70bb1",
    "name": "/distracted_babbage",
    "image": {
        "name": "shipyard/shipyard-cli:latest",
        "environment": {
            "PROG": "shipyard"
        },
        "hostname": "d465329a39ea",
        "restart_policy": {},
        "network_mode": "bridge"
    },
    "engine": {
        "id": "dockerdemo",
        "addr": "unix:///docker.sock",
        "cpus": 1,
        "memory": 400,
        "labels": [
            "engine1"
        ]
    },
    "state": "running"
}
shipyard cli>
```

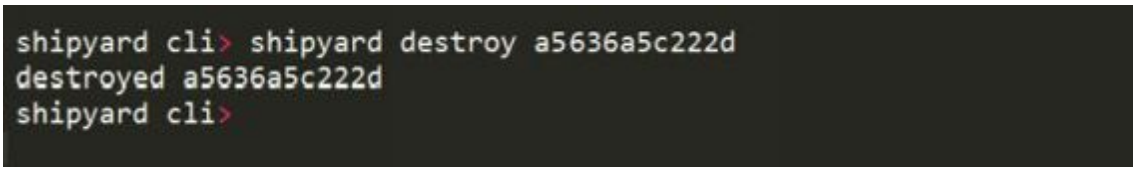
4. Use the following command and parameters to deploy a container.

```
shipyard run --name ubuntu:14.04 \
 --cpus 0.1 \
--memory 32 \
--type service \
--hostname demo-test \
--domain local \
```



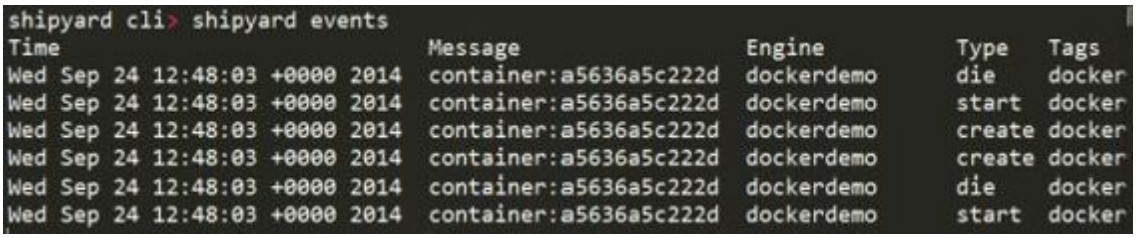
5. To destroy a container, execute the following command with container id.

```
shipyard destroy a5636a5c222d
```



6. You can view shipyard events as you see in the UI using the following command.

```
shipyard events
```



7. To view all the information about shipyard, use the following command.

```
shipyard info
```

Panamax

Panamax is an open source application, created by centurylink labs for deploying complex Docker applications.

Using panamax, you can create templates for your Docker applications and deploy them on the Docker host using an easy to use interface.

Panamax has its own template repository on github and it is integrated with the panamax UI. The UI has rich features for searching the panamax default templates and Docker hub images.

Panamax works with coreOS. You can run panamax in any platform which supports coreOS. Few containers will be created during the initial configuration to set up the UI for searching Docker images and templates from the Docker hub and panama repository.

Installation

Panamax can be installed on a local workstation running virtual box or you can set up a workstation on any cloud service which supports coreOS. In this section we will install and set up panamax on Google compute engine.

Follow the steps given below to install and configure panamax.

1. Create a coreOS VM from compute engine management console or using the gcloud cli.
2. Connect to the server using the gcloud shell or SSH agent such as putty.
3. Download the panamax installation files using curl, create a folder named Panamax inside /var folder and extract the installation files to that folder using the following command.

```
curl -O http://download.panamax.io/installer/panamax-latest.tar.gz && mkdir -p /var/panamax && tar -C /var/panamax -zxvf panamax-latest.tar.gz
```

```
panamax bibin.w # curl -O http://download.panamax.io/installer/panamax-latest.tr.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 14151  100 14151    0     0  11649      0  0:00:01  0:00:01 --:--:-- 19305
./
./Makefile
./configure
./create-docker-mount
./LICENSE
./desktop
./panamax
./.coreosenv
./README.md
./CHANGELOG.md
./ubuntu.sh
./Vagrantfile
./.version
./coreos
./CONTRIBUTING.md
panamax bibin.w #
```

4. CD in to the panamax directory and install panama using the following command.

mkdir -p /var/panama

./coreos install –stable

```
panamax bibin.w # cd /var/panamax
panamax panamax # ./coreos install --stable
Installing Panamax...
Failed to stop update-engine-reboot-manager.service: Unit update-engine-reboot-anage
Created symlink from /etc/systemd/system/update-engine-reboot-manager.service t /dev/
Created symlink from /etc/systemd/system/update-engine.service to /dev/null.
Created symlink from /etc/systemd/system/sockets.target.wants/systemd-journal-gteway

docker pull centurylink/panamax-api:latest
.....................................................................
docker pull centurylink/panamax-ui:latest
.....................
docker pull google/cadvisor:0.4.1
.....................
Created symlink from /etc/systemd/system/multi-user.target.wants/panamax-metrics.ser
Nov 12 07:45:07 panamax.c.cloud-repository.internal docker[1392]: PMX_UI
Nov 12 07:45:07 panamax.c.cloud-repository.internal docker[821]: [22d5b919] -job dele
Panamax install complete
```

5. Once installed, run the following Docker command to check if three containers for panamax have been launched.

docker ps

```
panamax panamax # docker ps
CONTAINER ID        IMAGE                                COMMAND                  CREATED
06cbe24d4dc3        centurylink/panamax-ui:latest        "/bin/sh -c 'bundle      6 minute
fc2f2c29d708        google/cadvisor:0.4.1                "/usr/bin/cadvisor /     6 minute
2d9c148462e7        centurylink/panamax-api:latest       "/bin/sh -c 'bundle      6 minute
panamax panamax #
```

Accessing panamax Web UI

Panamax-ui container runs the application for panamax UI. You can access the panamax web UI on port 3000.

Eg: http://<server ip>:3000

Deploying a sample rails application

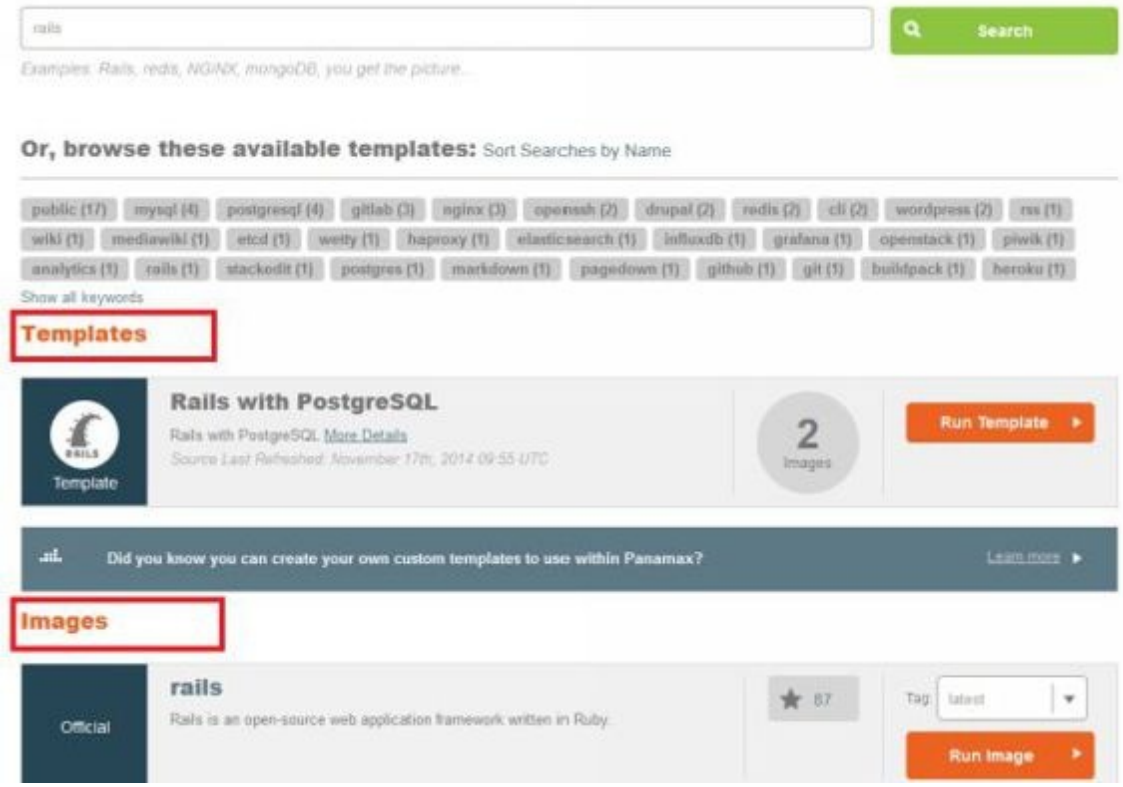
Panamax provides search functionality for searching panamax templates and images from Docker hub.

In this demonstration we will deploy a rails application using the default template with rails and postgres images.

Follow the steps given below to launch a rails application using a panama template.

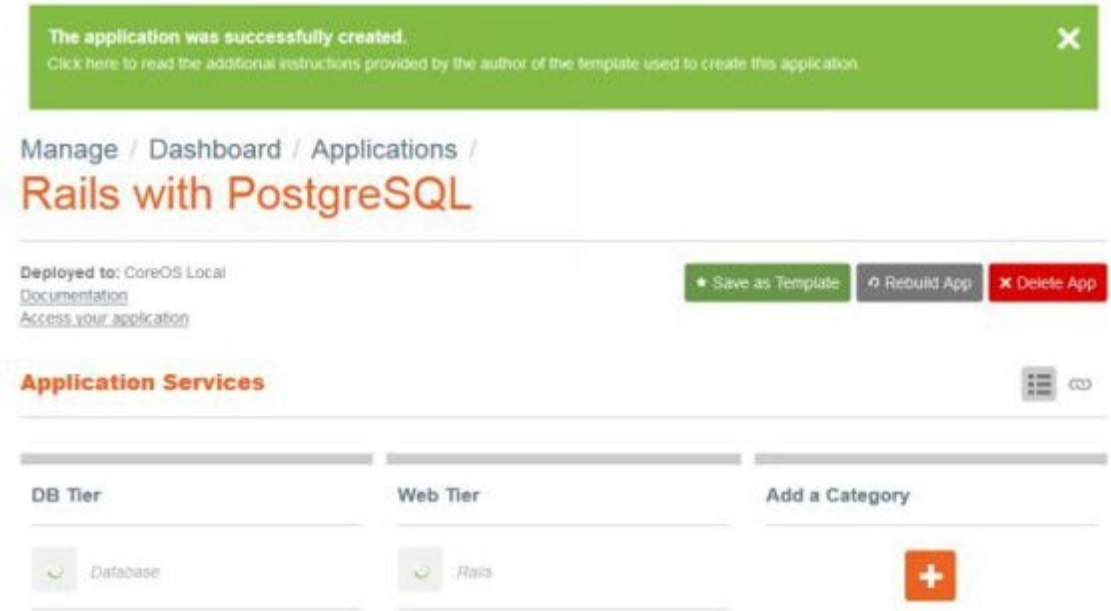
1. Type rails in the panamax search box and hit search button. You will

see a template section and the image section. The template is the default template from panamax repository and images are searched from the official Docker hub.

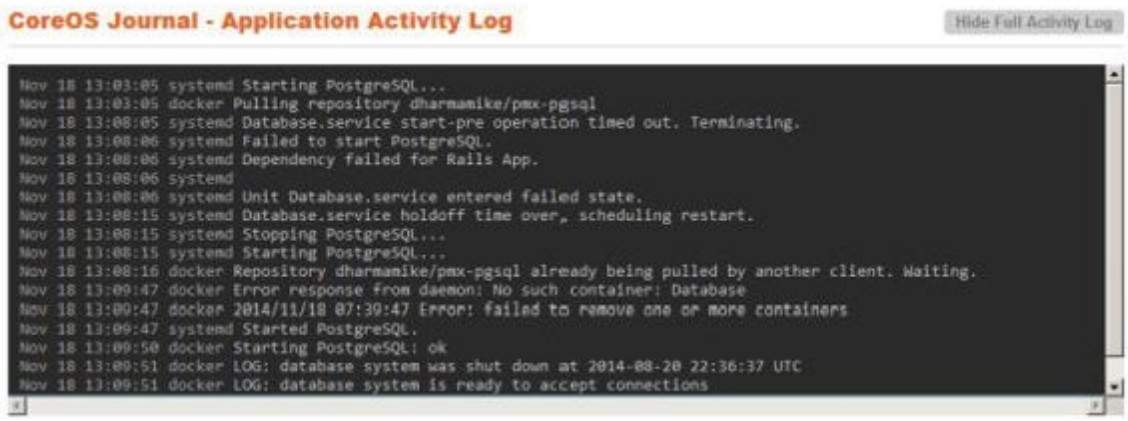


2. Click run template and select the “Run locally” option. This will deploy the rails and postgres container on the local Docker host on which panamax is running. You can also deploy the template to a target Docker host.

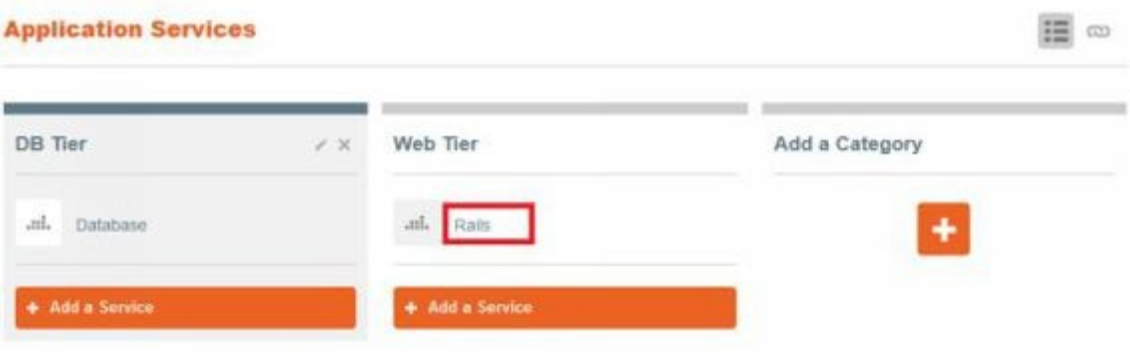
3. Once the application is created, you can see the containers being launched on the host.



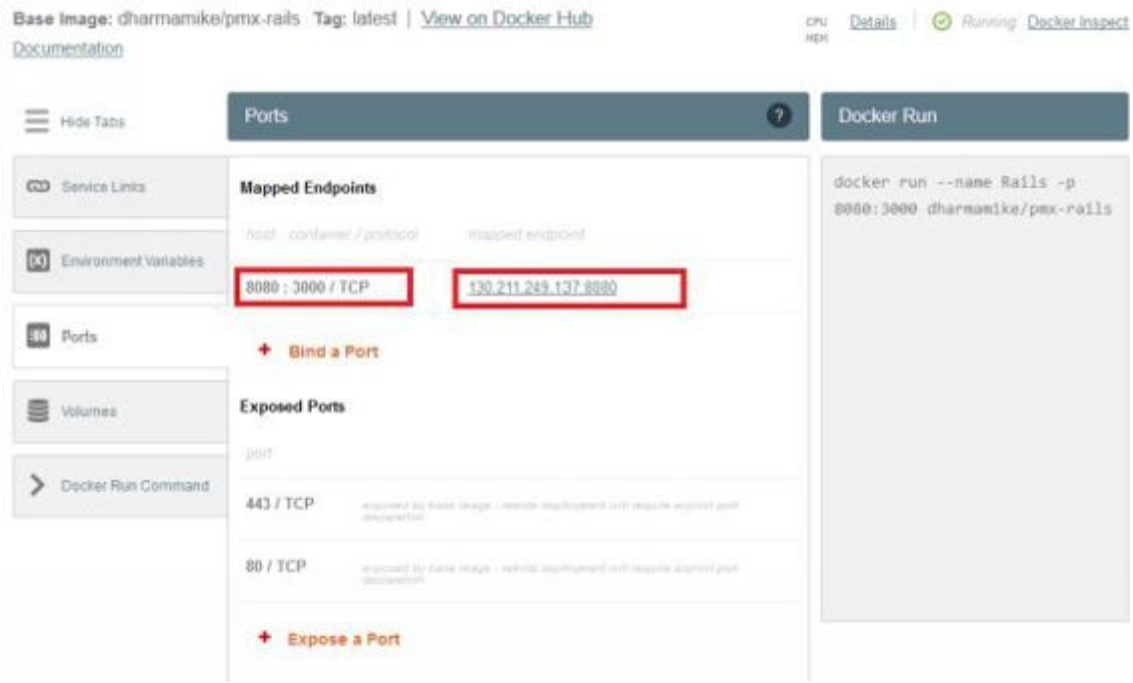
4. You can view the full logs from the “Activity log”.



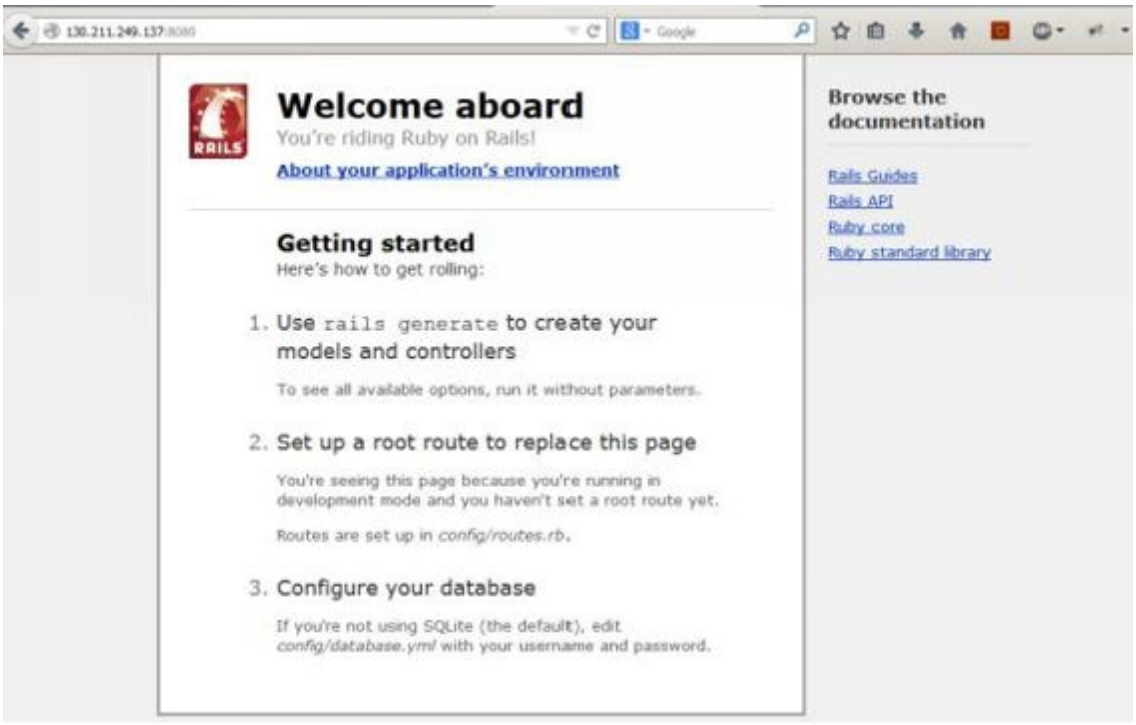
5. To view and edit the container ports and other preferences, click on the specific container. Let's view the ports and links associated with the rails container. Click the rails container from the dashboard.



6. In the dashboard you can view and edit the information for a particular container. Our rails container is linked to postgre SQL container. If you click on the ports tab, you can see on which host post the rails container has been linked. Host port 8080 is mapped on port 3000 of the rails container



7. Now if you access the host IP on port 8080, you can view the sample rails application running on the rails container.



11
Docker Service Discovery and Orchestration

One of the challenges in using Docker containers is the communication between hosts.

One of the solutions for Docker host to host communication is service discovery. In this section we will demonstrate the use of consul service discovery tool with Docker.

Service discovery with consul

Service discovery is a key component in an architecture which is based on micro services. Service discovery is the process of knowing when a process is listening to applications processes running specific TCP or UDP port and connecting to those processes using names.

In modern cloud infrastructure, every application should be designed for failure. So, multiple instances of web servers, databases and application servers will be running and they interact with each other using API's, message queues etc. Any of the services may fail at a given point of time and scale horizontally. When these new instances of services come up, it should be able to advertise itself to the other components in your infrastructure. Here is where consul comes in.

Consul is a service discovery tool for configuring services in an infrastructure. It helps in achieving the two main principles of service oriented architecture such as loose coupling and service discovery. Consul has the following features.

Service discovery

The nodes which are running consul client can advertise its service and the nodes which want to consume a particular service can use a consul client to discover the service.

Health checking

Consul is capable of doing health checks based on various parameters like http status codes, memory utilization etc. The health check information obtained by consul clients can be used for routing traffic to healthy hosts in a cluster.

Key/value store

Consul has its own key/value store. Applications can use its key/value store for operations such as leader election, coordination and flagging. All the operations can be performed using API calls.

Multi Datacentre

You can have consul configured with multiple datacenters. So, if your application spawns to multiple regions, you don’t need to create another layer of abstraction for multi region support.

Consul Architecture

Consult agent runs on every node (client) which provides a service. There is no need for consul agent on nodes which consumes a service. Consult agent will do the health check for the services running in the node and also it does the health check for node itself. Every consul agent will talk to consul servers where all the data about the services are stored and replicated among other servers.

A consul server is elected automatically during the cluster configuration. A cluster can have one to many servers but, more than one with a consul cluster is recommended for deployments.

When a service or a node wants to discover a service from consul cluster, it can query the consul server for information. A request can be made using DNS or http request. You can also query other consul agents for service information. When a request is made to consul agent for service enquiry, the consul agent will forward the request to the consul server.

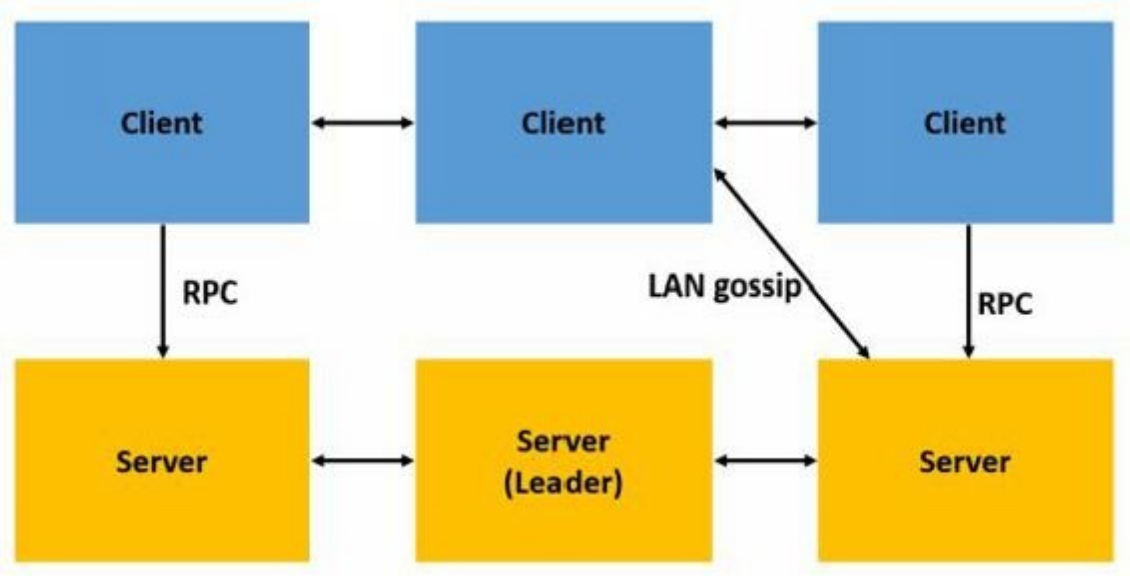


Fig 11-1: consul Architecture

Gossip

Consul is built on top of serf. Serf is a decentralized solution for cluster membership. Serf provides the gossip protocol for operations like membership, failure detection and event broadcast mechanism. Gossip protocol uses UDP for random node to node communication. All the servers in the consul cluster participate using the gossip pool. The gossip pool in a cluster contains all the nodes.

In this section we will do the following,

1. Build a Docker consul image from scratch
2. Set up a three node consul cluster
3. Set up registrator for auto registering and deregistering containers in consul registry
4. Deploy containers on consul cluster.

Building a consul image from scratch

To build a consul image from scratch we need the following:

1. A Docker file with all the image specifications and commands.
2. Config.JSON file for consul agent
3. Launch.sh this file is required for launching the consul agent.

Follow the steps given below create necessary configuration files for building a consul image;

1. Create a folder name consul in your Docker host.
2. CD in to consul folder and create another folder name config
3. CD in to config folder and create a file name config.JSON and copy the following contents on to that file.

```
{
"data_dir": "/data",
"ui_dir": "/ui",
"client_addr": "0.0.0.0",
"ports": {
"dns": 53
},
"recursor": "8.8.8.8"
}
```

4. Inside consul folder create a file name launch.sh and copy the following contents on to the file.

```
#!/bin/bash
set -eo pipefail
echo "Starting consul agent"
consul agent -config-dir=/config "$@"
```

5. Inside consul folder create a file name Dockerfile and copy the following contents on to the file.

```
FROM centos
RUN yum -y update
RUN yum -y install which
RUN yum -y install git
RUN yum -y install unzip
# Add consul binary
ADD
https://dl.bintray.com/mitchellh/consul/0.3.1_linux_amd64.zi
/tmp/consul.zip
RUN cd /bin && unzip /tmp/consul.zip && chmod +x
```

```
/bin/consul && rm /tmp/consul.zip
# Add consul UI
ADD
https://dl.bintray.com/mitchellh/consul/0.3.1_web_ui.zip
/tmp/webui.zip
RUN cd /tmp && unzip /tmp/webui.zip && mv dist /ui &&
rm /tmp/webui.zip
# Add consul config
ADD ./config /config/
# ONBUILD will make sure that any additional service
configuration file is added to Docker conatiner as well.
ONBUILD ADD ./config /config/
# Add startup file
ADD ./launch.sh /bin/launch.sh
RUN chmod +x /bin/launch.sh
# Expose consul ports
EXPOSE 8300 8301 8301/udp 8302 8302/udp 8400 8500
53/udp
#Create a mount point
VOLUME ["/data"]
# Entry point of container
ENTRYPOINT ["/bin/launch.sh"]
```

The above Dockerfile will create an image from centos and configures consul agent using the config and launch file.

6. Now you should have the folder and file structure as shown below.

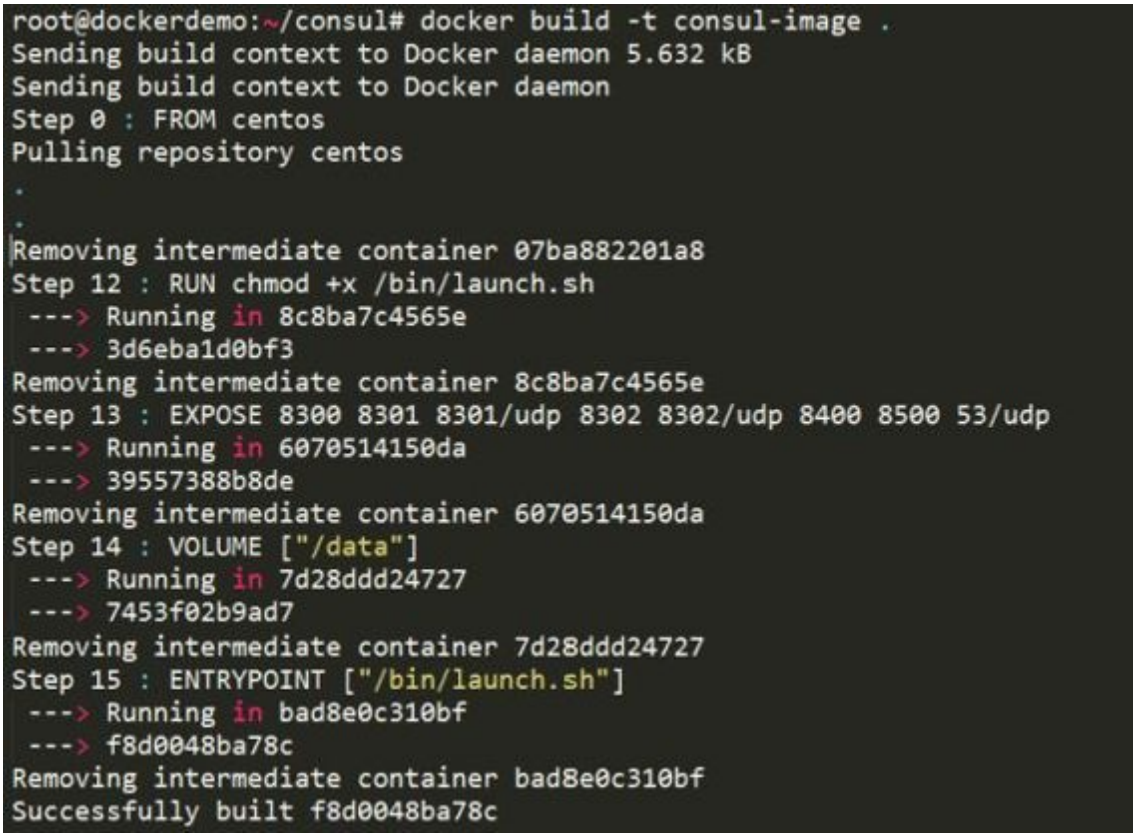
```
consul
  --config
    --config.json
  --Dockerfile
  --Launch.sh
```

Building the consul image

Now we have the Dockerfile and consul configuration files in place. From the consul directory run the following Docker build command to build the consul

image.

```
docker build -t consul-image .
```



Creating a single instance of consul

Once you have successfully built the consul image, run the following command to create a single instance of consul on the Docker host to check if everything is working correctly.

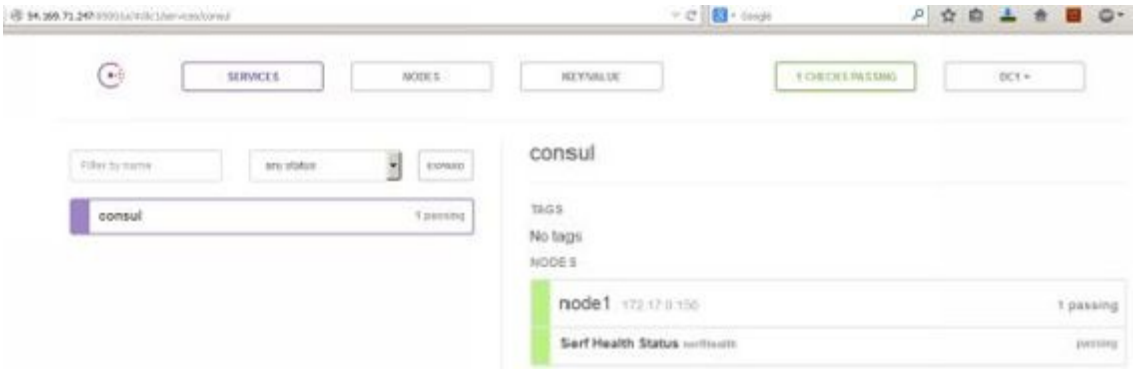
```
docker run -p 8400:8400 -p 8500:8500 -p 8600:53/udp -h node1 consul-image \    -server –bootstrap
```

The above command will use the local consul-image and creates a container with hostname node1. We expose 8400 (RPC), 8500 (HTTP), and 8600 (DNS) to try all the interfaces.

```
root@dockerdemo:~# docker run -p 8400:8400 -p 8500:8500 -p 8600:53/udp -h node1 consu
Starting consul agent
==> WARNING: Bootstrap mode enabled! Do not enable unless necessary
==> WARNING: It is highly recommended to set GOMAXPROCS higher than 1
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Consul agent running!
         Node name: 'node1'
        Datacenter: 'dc1'
            Server: true (bootstrap: true)
       Client Addr: 0.0.0.0 (HTTP: 8500, DNS: 53, RPC: 8400)
      Cluster Addr: 172.17.0.150 (LAN: 8301, WAN: 8302)
    Gossip encrypt: false, RPC-TLS: false, TLS-Incoming: false
    2014/10/07 10:33:29 [INFO] raft: Election won. Tally: 1
    2014/10/07 10:33:29 [INFO] raft: Node at 172.17.0.150:8300 [Leader] entering Lead
    2014/10/07 10:33:29 [INFO] consul: cluster leadership acquired
    2014/10/07 10:33:29 [INFO] consul: New leader elected: node1
    2014/10/07 10:33:29 [INFO] consul: member 'node1' joined, marking health alive
```

Once the container is created, you can access the consul UI from the browser using your server IP followed by port 8500

http://< public or private ip>:8500



Setting up consul cluster

You can set up a consul cluster on a single node and multi node as well. In this section we will look in to single host and multi host consul cluster set up.

Single host consul cluster

In this set up we will launch three consul nodes on the same host for experimentation. We will start our first node with -bootstrap-expect 3, which will wait for the other two nodes to join to form a cluster. We will be using the consul-image created from the Docker file for this single node cluster setup.

Follow the steps given below to create a single node consul cluster.

1. Start the first node with *-bootstrap-expect 3* parameter to wait for

other two nodes to join the cluster. Run the following command to start the first node

docker run -d --name node1 -h node1 consul-image -server -bootstrap-expect 3

```
root@dockerdemo:~# docker run -d --name node1 -h node1 consul-image -server -bootstra
3a19d14ad7a4638e59cf62ce24ffd598adfaac1b11dd8282541774d0b2cb0b56
```

2. We will join the next two nodes to node1 using the hosts internal IP. So, get the host's internal IP in JOIN_IP variable using the following command.

JOIN_IP="$(docker inspect -f '{{ .NetworkSettings.IPAddress }}' node1)"

3. Start the second node with join parameter and JOIN_IP using the following command.

docker run -d --name node2 -h node2 consul-image -server -join $JOIN_IP

```
root@dockerdemo:~# docker run -d --name node2 -h node2 consul-image -server -join $JO
82b338cd30e0edeba1ccad91c3f163817573b954ab79aa95ef67ce16b31768e7
root@dockerdemo:~#
```

4. Start the third node with join parameter and JOIN_IP using the following command

docker run -d --name node3 -h node3 consul-image -server -join $JOIN_IP

```
root@dockerdemo:~# docker run -d --name node3 -h node3 consul-image -server -join \
$JOIN_IP
e07ebac300a1c0e8fcb7d1f105ece7b4869a8e1e6c28f4fe095196d11a0b1b70
root@dockerdemo:~#
```

5. Now if you check the Docker logs for the third container, you will see the message "leader elected". We have a working consul cluster now.

docker logs <container-id>

```
root@dockerdemo:~# docker logs 82b338cd30e0edeba1ccad91c3f163817573b954ab79a
Starting consul agent
==> Starting Consul agent...
==> Starting Consul agent RPC...
==> Joining cluster...
    Join completed. Synced with 1 initial agents
==> Consul agent running!
         Node name: 'node2'
        Datacenter: 'dc1'
            Server: true (bootstrap: false)
       Client Addr: 0.0.0.0 (HTTP: 8500, DNS: 53, RPC: 8400)
    2014/10/07 11:25:33 [INFO] serf: EventMemberJoin: node3 172.17.0.154
    2014/10/07 11:25:33 [INFO] serf: EventMemberJoin: node3.dc1 172.17.0.154
    2014/10/07 11:25:33 [INFO] raft: Node at 172.17.0.154:8300 [Follower] entering Fo
    2014/10/07 11:25:33 [INFO] consul: adding server node3 (Addr: 172.17.0.154:8300)
    2014/10/07 11:25:33 [INFO] consul: adding server node3.dc1 (Addr: 172.17.0.154:83
    2014/10/07 11:25:33 [INFO] agent: (LAN) joining: [172.17.0.152]
    2014/10/07 11:25:33 [INFO] serf: EventMemberJoin: node2 172.17.0.153
    2014/10/07 11:25:33 [INFO] serf: EventMemberJoin: node1 172.17.0.152
    2014/10/07 11:25:33 [INFO] agent: (LAN) joined: 1 Err: <nil>
    2014/10/07 11:25:33 [ERR] agent: failed to sync remote state: No cluster leader
    2014/10/07 11:25:33 [INFO] consul: adding server node2 (Addr: 172.17.0.153:8300)
    2014/10/07 11:25:33 [INFO] consul: adding server node1 (Addr: 172.17.0.152:8300)
    2014/10/07 11:25:34 [INFO] consul: New leader elected: node1
```

Now we have a working three node single host cluster, but we won't be able to access the cluster UI because we did not do any port mapping for the created nodes. We can access consul UI by launching another consul agent node in client mode. It means that, it will not participate in the consensus quorum instead we can use this to access the consul UI.

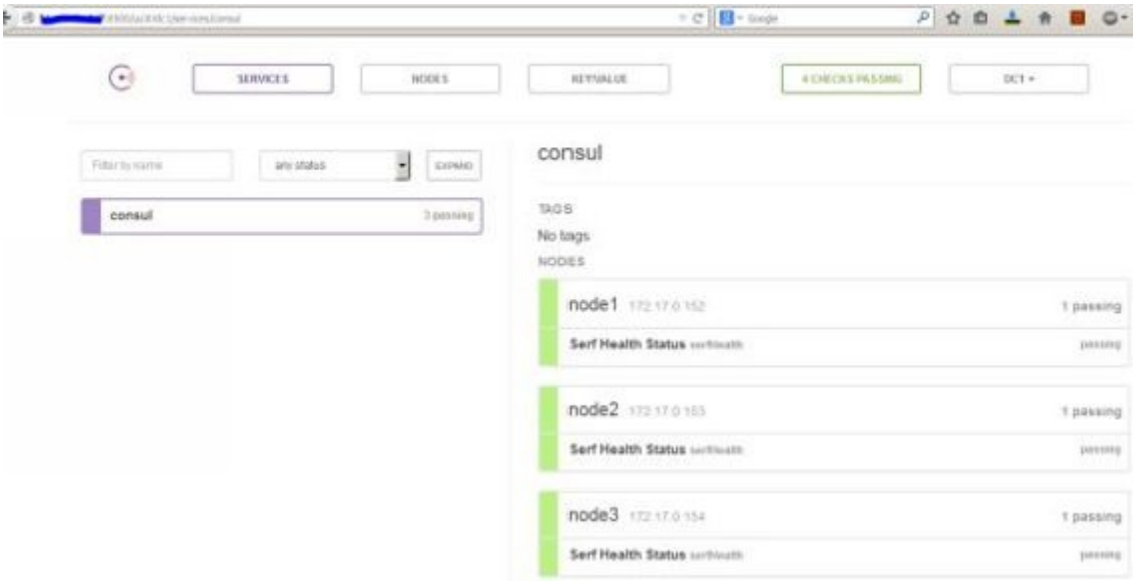
6. So let's create another container without the server parameter to run in client mode for accessing the consul UI. Run the following command to create the fourth consul client node.

docker run -d -p 8400:8400 -p 8500:8500 -p 8600:53/udp -h node4 \ consul-image -join $JOIN_IP

```
root@dockerdemo:~# docker run -d -p 8400:8400 -p 8500:8500 -p 8600:53/udp -h \
node4 consul-image -join $JOIN_IP
d66fe929a83eda6c1d65cd52fc0e4986caa97450e372feeb8a63ae3e33ed4bc5
root@dockerdemo:~#
```

7. We can now access the consul UI using the hosts IP followed by port 8500.

http://hostip:8500



Multi host consul cluster

In this section, we will create a multi host consul cluster with 3 nodes.

Note: In this demo, we will be using three amazon ec2 instances with private networking enabled. All three instances will be able to contact each other using its private IPs.

We have three Docker hosts (node1, Node2 and node3) in which we will install consul agent servers to form a cluster. All the three Docker hosts can communicate with each other using the private IPs.

The Docker image used in this demo is a public image from Docker registry (progrium/consul) which has a preconfigured consul agent. You need to commit and publish the image to your Docker repository if you want to use the image we created in single host set up.

Following are the requirement to launch a multi host cluster.

- Each host should have a private IP and it should be able to communicate to other hosts using its private IP.
- All the necessary ports should be opened on every host.
- An image configured with consul agent.

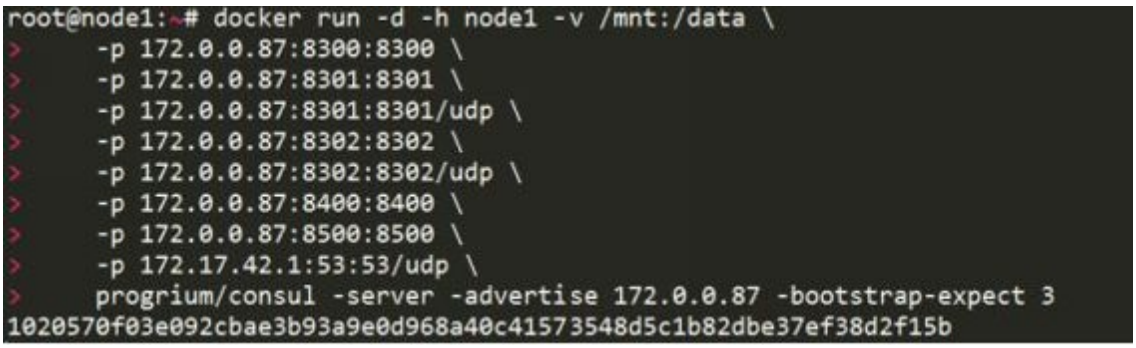
Following are the flags used in commands for cluster setup

- -bootstrap-expect 3 :- This flag will make the host wait until three hosts are connected together to become a cluster. This parameter will be used only on the first node.
- -advertise :- This flag will pass the private IP to consul.
- -join:- This flag will be used by second and third nodes to join the cluster.

Now let's get started with the setup. Follow the steps given below to set up the multi host cluster.

1. On node1 run the following command on node1 (replace 172.0.0.87 with node1's private IP and 172.17.42.1 with Docker bridge's IP)

```
docker run -d -h node1 -v /mnt:/data \
  -p 172.0.0.87:8300:8300 \
  -p 172.0.0.87:8301:8301 \
  -p 172.0.0.87:8301:8301/udp \
  -p 172.0.0.87:8302:8302 \
  -p 172.0.0.87:8302:8302/udp \
  -p 172.0.0.87:8400:8400 \
  -p 172.0.0.87:8500:8500 \
  -p 172.17.42.1:53:53/udp \
  progrium/consul -server -advertise 172.0.0.87 -bootstrap-expect
```



2. On node2 run the command give below with its private IP. The IP mentioned in the –join flag is node1's IP.

```
docker run -d -h node2 -v /mnt:/data  \
  -p 172.0.0.145 :8300:8300 \
  -p 172.0.0.145 :8301:8301 \
  -p 172.0.0.145 :8301:8301/udp \
  -p 172.0.0.145 :8302:8302 \
  -p 172.0.0.145 :8302:8302/udp \
  -p 172.0.0.145 :8400:8400 \
  -p 172.0.0.145 :8500:8500 \
```

*-p 172.17.42.1:53:53/udp *

progrium/consul -server -advertise 172.0.0.145 -join 172.0.0.87

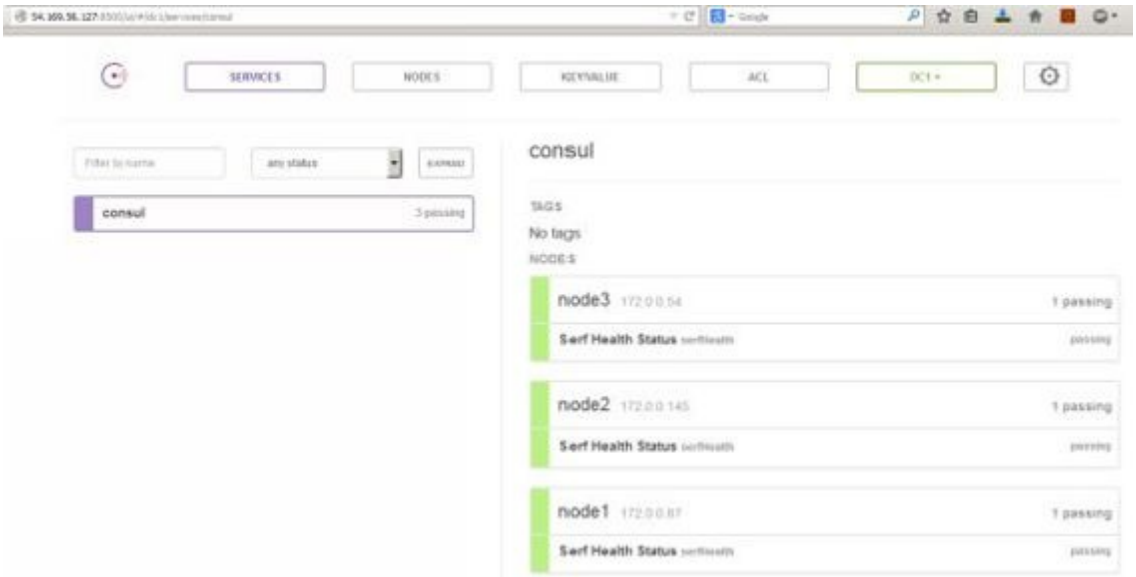
```
root@node2:~# docker run -d -h node2 -v /mnt:/data  \
>      -p 172.0.0.145:8300:8300 \
>      -p 172.0.0.145:8301:8301 \
>      -p 172.0.0.145:8301:8301/udp \
>      -p 172.0.0.145:8302:8302 \
>      -p 172.0.0.145:8302:8302/udp \
>      -p 172.0.0.145:8400:8400 \
>      -p 172.0.0.145:8500:8500 \
>      -p 172.17.42.1:53:53/udp \
>     progrium/consul -server -advertise 172.0.0.145 -join 172.0.0.87
14933fd7741e0c22cd318971628d16393ba2a0d1f57b07e4951b4ae8635bf9c9
```

3. On node3 run the following command with respective private IP addresses.

```
root@node3:~# docker run -d -h node3 -v /mnt:/data  \
>      -p 172.0.0.54:8300:8300 \
>      -p 172.0.0.54:8301:8301 \
>      -p 172.0.0.54:8301:8301/udp \
>      -p 172.0.0.54:8302:8302 \
>      -p 172.0.0.54:8302:8302/udp \
>      -p 172.0.0.54:8400:8400 \
>      -p 172.0.0.54:8500:8500 \
>      -p 172.17.42.1:53:53/udp \
>     progrium/consul -server -advertise 172.0.0.54 -join 172.0.0.87
4f1b610a22b1f955827c99651337dca90b3c4f1d3fb4467d6e32e3627a3d8bfa
```

4. Now you can access the consul UI using any Host IP followed by port 8500

http://hostip:8500

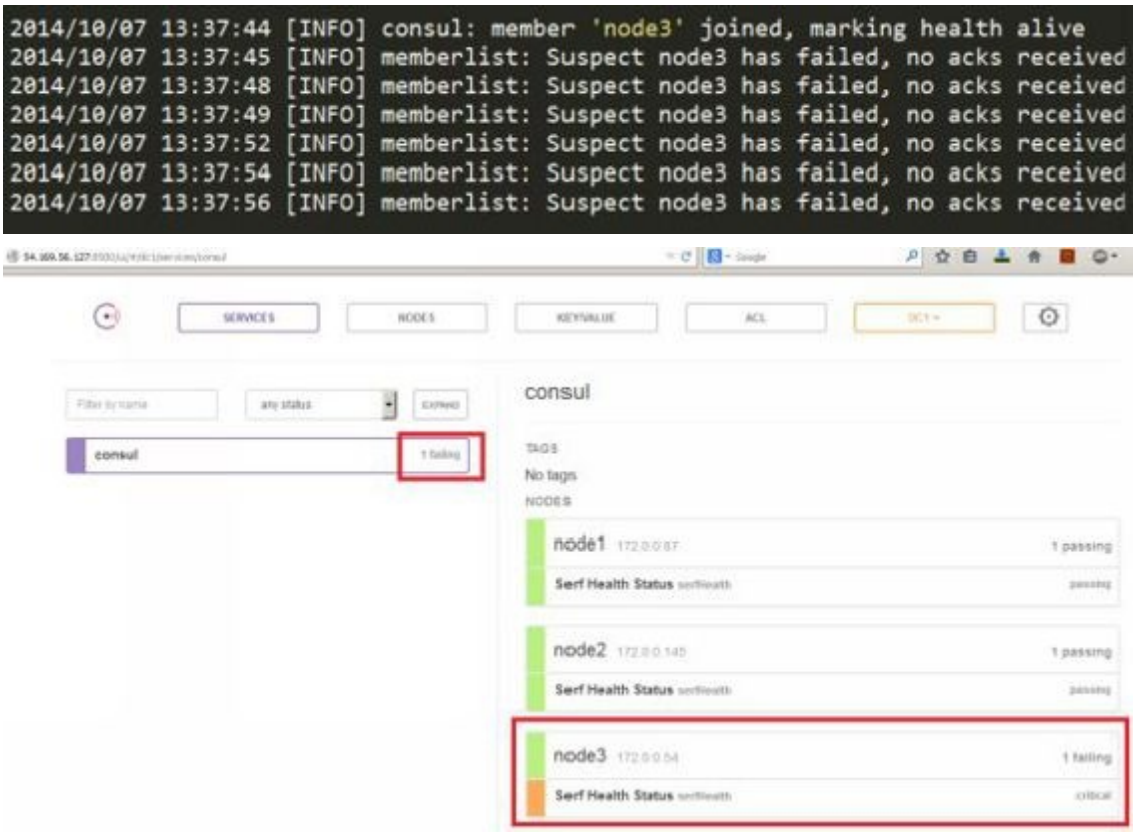


5. You can check the logs of the cluster using the following Docker command.

```
docker logs <container id>
```

You can check the logs by stopping one node or stopping the Docker service. If a service is failing you can view it in the logs and the web UI as shown in the images below.

```
docker logs  1020570f03e0
```



Note: If the leader node fails, another node will elect itself as the leader based on consul's algorithm.

So far we have set up clusters on single and multi-host. Next we will look to auto registration of containers to consul using registrator tool. This tool listens to Docker events on the hosts and registers and deregisters when the containers is launched or terminated.

Registering services using Registrator

If you are using consul, each and every container that is being launched should be registered to the consul registry service. You can run a consul agent on each

container or you can use registrator tool to register container services automatically.

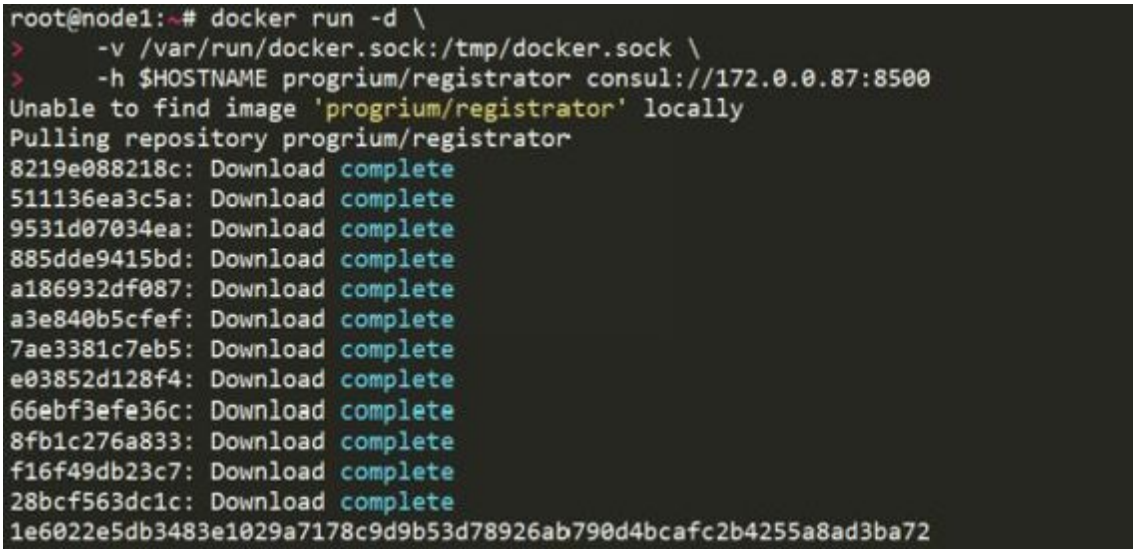
Registrator is a service registry bridge for Docker to automatically register new containers to the consul registry. Registrator makes it really easy by running a registrator container on the Docker host for registering new services without having to run consul agent on each container.

Registrator is modeled in to a container. We just have to run the container on a Docker host in the consul cluster. We can run the registrator container on multi host consul cluster as well. We will use the public registrator Docker image (progrium/registrator) for the set up.

Follow the steps given below to set up registrator on consul cluster.

1. On node1 run the following command:

```
docker run -d \
  -v /var/run/docker.sock:/tmp/docker.sock \
  -h $HOSTNAME progrium/registrator
consul://172.0.0.87:8500
```



Run the registrator container on other two nodes with respective consul IPs.

2. Now you can check the Docker logs using the container id to check if registrator is working as expected.

```
docker logs \
1e6022e5db3483e1029a7178c9d9b53d78926ab790d4bcafc2b4
```

```
root@node1:~# docker logs 1e6022e5db3483e1029a7178c9d9b53d78926ab790d4bcafc2b4255a8ac
2014/10/08 06:11:23 registrator: Using consul registry backend at consul://172.0.0.87
2014/10/08 06:11:23 registrator: ignored: 1e6022e5db34 no published ports
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:53:udp
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:8300
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:8301
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:8301:udp
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:8302
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:8302:udp
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:8400
2014/10/08 06:11:23 registrator: added: 89d8640122c8 node1:loving_archimedes:8500
2014/10/08 06:11:23 registrator: Listening for Docker events...
root@node1:~#
```

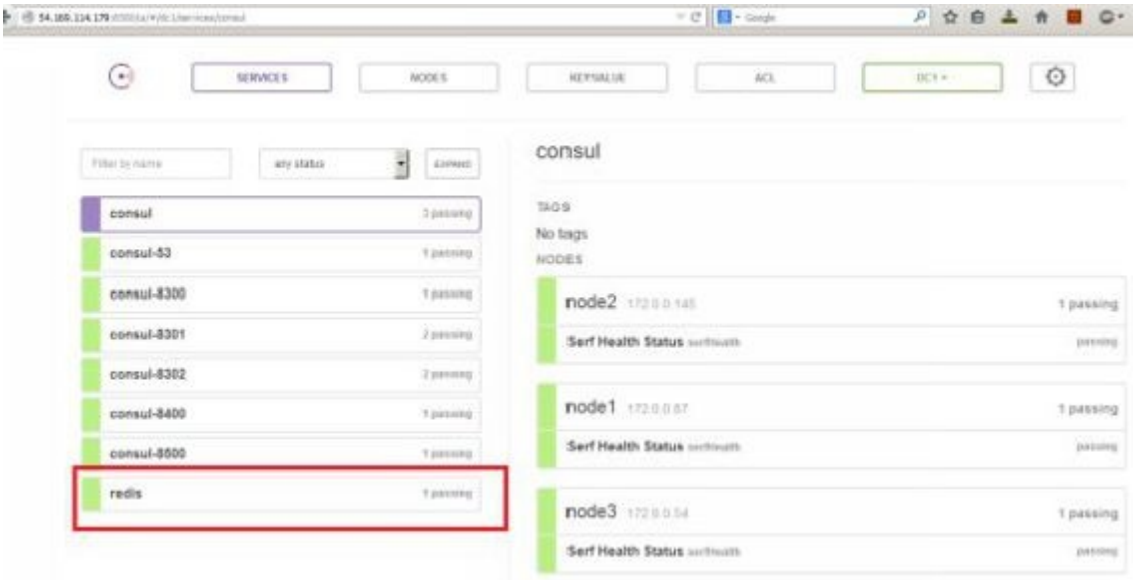
The last line of log says that registrator is listening for Docker events. It means our registrator container is working without any errors. Once it starts running, you can basically start any Docker container and you don't have to do anything extra to register it to the consul registry. Registrator will take care of registration process by listening to the Docker events.

3. Now, lets' run a redis container using the public image (Dockerfile/redis) to check if it is automatically registering to the consul registry. The redis image does not do anything special for service discovery it is just a normal redis image. Run the following command to deploy a redis container.

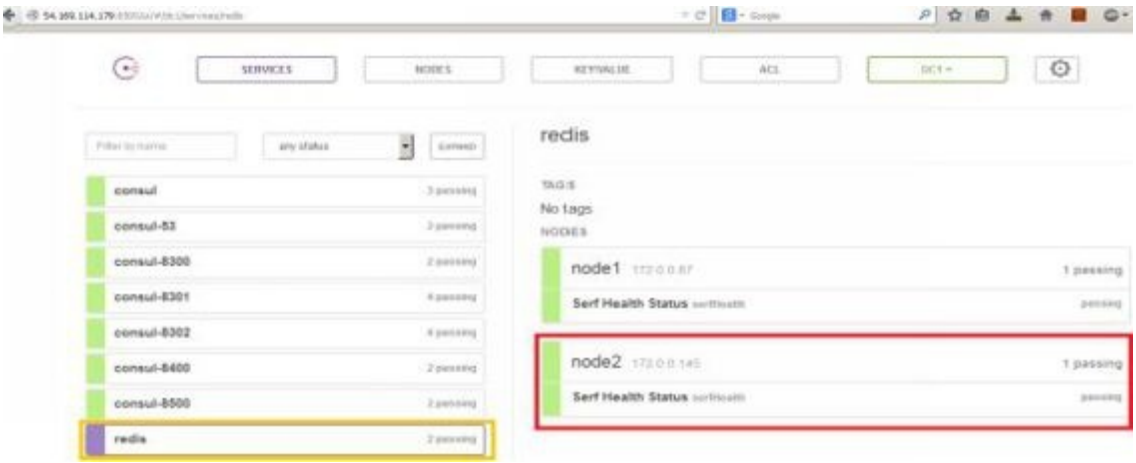
docker run -d -P Dockerfile/redis

```
root@node1:~# docker run -d -P dockerfile/redis
Unable to find image 'dockerfile/redis' locally
Pulling repository dockerfile/redis
fb2ec8afee3e: Download complete
511136ea3c5a: Download complete
b18d0a2076a1: Download complete
4dd7cd44551b: Download complete
28dce010c4e9: Download complete
127c46d9f48e4b938b4c30fadca74fd2405639df61f339e68471c36c30db7084
root@node1:~#
```

4. Now if you check the consul UI, you can see that redis service has been automatically registered to our consul cluster without any extra work.



5. Now run the same redis container on node 2 to check if it is registering with the same service name on the other nodes. Once the redis containers are deployed on node 2 check the consul UI and click redis service. You will see two instances of redis running on two hosts with service name redis as shown in image below.

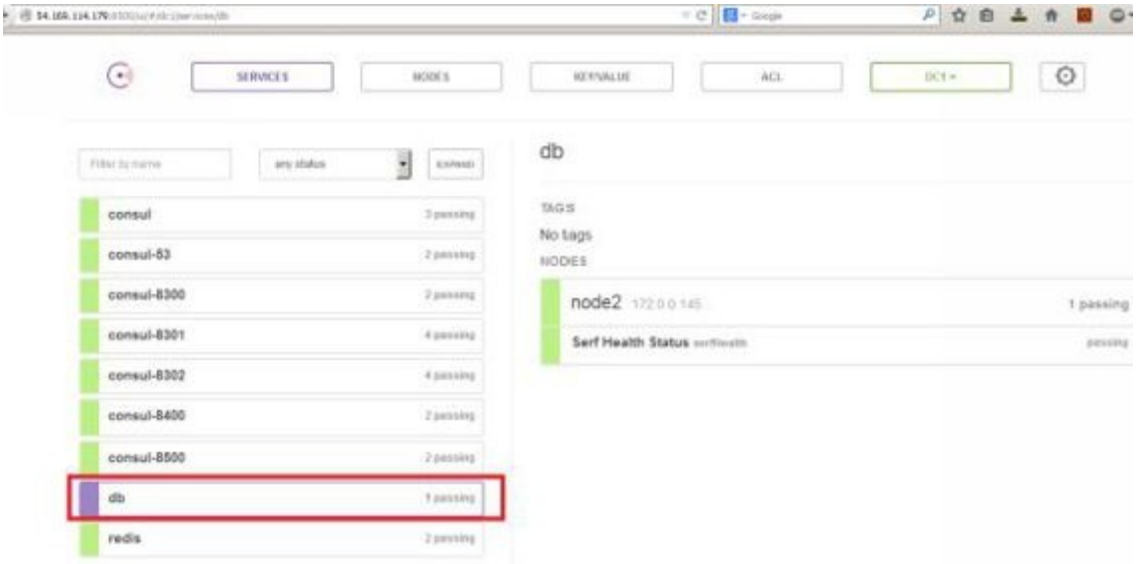


6. By default registrator used the image name as the service name used by the author. You can override this by specifying the service name, service tags as an environment variable while deploying a container. Let's deploy the same redis container with a different service name using the following.

```
docker run -d -P -e "SERVICE_NAME=db" dockerfile/redis
```

```
root@node2:~# docker run -d -P -e "SERVICE_NAME=db" dockerfile/redis
bcab74fa48d328e26c5d6a30dc080942b1b033c7404f4dd0275c8fe7881938aa
root@node2:~#
```

Once the container is launched, you can check the consul UI for the newly registered redis container with service name db.

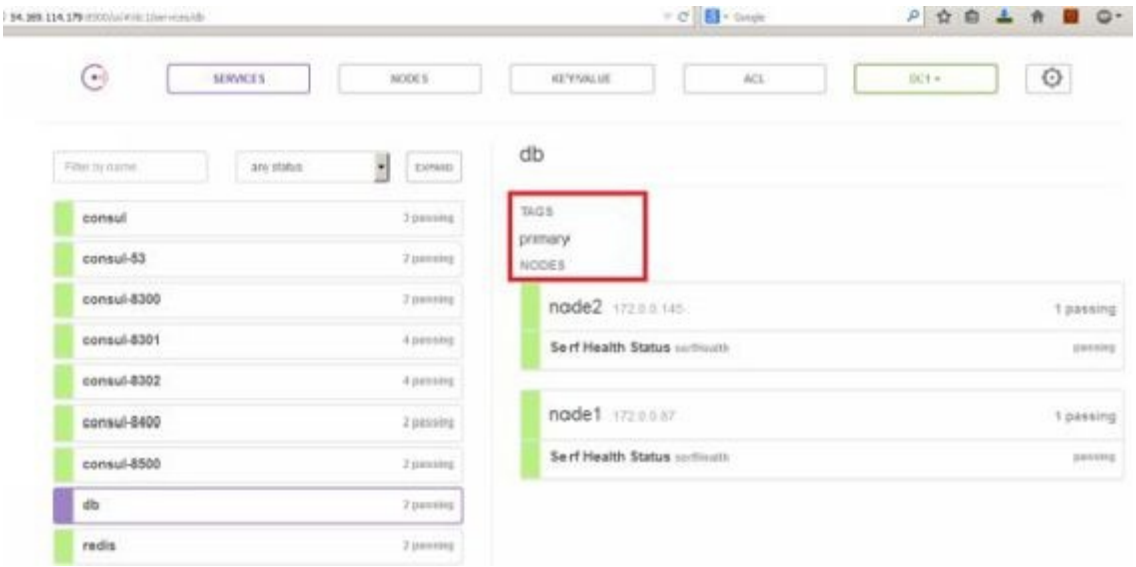


7. Now let’s run db service on node 1 with a service tag primary and service name db

```
docker run -d -P -e "SERVICE_NAME=db" -e "SERVICE_TAGS=primary" \ dockerfile/redis
```

```
root@node1:~# docker run -d -P -e "SERVICE_NAME=db" -e "SERVICE_TAGS=primary" dockerf
b14bbb52ca3d500e522fcc815608e7d74bec2fca79bcfae4892024a4873a6b87
root@node1:~#
```

Now if you look at the db service in consul UI, you will find a primary tag for a redis instance. Tagging can be useful in services like databases using primary and secondary services.



Now we have a cluster with auto registering containers services. Next we will look in to how to discover the registered service to be used by other application containers. Services in consul can be discovered using DNS or HTTP API.

Example http request:

Using http request you can get the service details and use it on your application. Run the following command to get the details of db service we deployed earlier.

curl -s http://54.169.114.179:8500/v1/catalog/service/db

```
root@node1:~# curl -s http://54.169.114.179:8500/v1/catalog/service/db
[
{"Node":"node2",
"Address":"172.0.0.145",
"ServiceID":"node2:sleepy_heisenberg:6379",
"ServiceName":"db",
"ServiceTags":[],
"ServicePort":49155}]
root@node1:~#
```

Example DNS request

Same as http, using consul's DNS service you can get the details of a particular service registered on consul. Run the following command on any node on the cluster to get the service details using DNS

dig @$BRIDGE_IP -t SRV db.service.consul

```
root@node2:~# dig @$BRIDGE_IP -t SRV db.service.consul
; <<>> DiG 9.9.5-3-Ubuntu <<>> @172.17.42.1 -t SRV db.service.consul
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 39439
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; QUESTION SECTION:
;db.service.consul.             IN      SRV
;; ANSWER SECTION:
db.service.consul.      0       IN      SRV     1 1 49155 node2.node.dc1.consul.
;; ADDITIONAL SECTION:
node2.node.dc1.consul.  0       IN      A       172.0.0.145
;; Query time: 1 msec
;; SERVER: 172.17.42.1#53(172.17.42.1)
;; WHEN: Wed Oct 08 12:44:32 UTC 2014
;; MSG SIZE  rcvd: 130
root@node2:~#
```

You can also look in to services by specifying tags. You can also look in to services specifying tags. We created a db service with primary tag earlier in this section.

Run the following command for looking up db service with tag primary.

```
root@node2:~# dig @$BRIDGE_IP -t SRV primary.db.consul.service
; <<>> DiG 9.9.5-3-Ubuntu <<>> @172.17.42.1 -t SRV primary.db.consul.service
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 63497
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;primary.db.consul.service.     IN      SRV
;; AUTHORITY SECTION:
.       1799    IN      SOA     a.root-servers.net. nstld.verisign-grs.com. 2014100800 180
;; Query time: 119 msec
;; SERVER: 172.17.42.1#53(172.17.42.1)
;; WHEN: Wed Oct 08 13:05:32 UTC 2014
;; MSG SIZE  rcvd: 129
root@node2:~#
```

Docker cluster management using Mesos

Apache Mesos is an open source centralized fault-tolerant cluster manager. It’s designed for distributed computing environments to provide resource isolation and management across a cluster of slave nodes. It schedules CPU and memory resources across the cluster in much the same way the Linux Kernel schedules local resources. Following are the features offered by Mesos.

1. It can scale to more than 10000 nodes
2. Leverages Linux containers for resource isolation.
3. Schedules CPU and memory efficiently.
4. Provides a highly available master architecture using Apache Zookeeper.
5. Provides a web interface for monitoring the cluster state.

Key differences between Mesos and Virtualization:

- Virtualization splits a single physical resource into multiple virtual resources
- Mesos joins multiple physical resources into a single virtual resource

It schedules CPU and memory resources across the cluster in much the same way the Linux Kernel schedules local resources. Let’s have a look at Mesos components and its relevant terms.

A Mesos cluster is made up of four major components:

1. ZooKeeper
2. Mesos masters
3. Mesos slaves
4. Frameworks

ZooKeeper

Apache ZooKeeper is a centralized configuration manager, used by distributed applications such as Mesos to coordinate activity across a cluster. Mesos uses ZooKeeper to elect a leading master and for slaves to join the cluster.

Mesos master

A Mesos master is a Mesos instance in control of the cluster. A cluster will typically have multiple Mesos masters to provide fault-tolerance, with one instance being elected as the leading master. The master manages the slave daemons

Mesos slave

A Mesos slave is a Mesos instance which offers resources to the cluster. They are the ‘worker’ instances – tasks are allocated to the slaves by the Mesos master.

Frameworks

On its own, Mesos only provides the basic “kernel” layer of your cluster. It lets other applications request resources in the cluster to perform tasks, but does nothing itself.

Frameworks bridge the gap between the Mesos layer and your applications. They are higher level abstractions which simplify the process of launching tasks on the cluster.

Chronos

Chronos is a cron-like fault-tolerant scheduler for a Mesos cluster. You can use it to schedule jobs, receive failure and completion notifications, and trigger other dependent jobs.

Marathon

Marathon is the equivalent of the Linux upstart or init daemons, designed for long-running applications. You can use it to start, stop and scale applications across the cluster.

Others

There are a few other frameworks,

1. Aurora – service scheduler
2. Hadoop – data processing
3. Jenkins – Jenkins slave manager
4. Spark – data processing
5. Torque – resource manager

You can also write your own framework, using Java, Python or C++.

Mesos Architecture

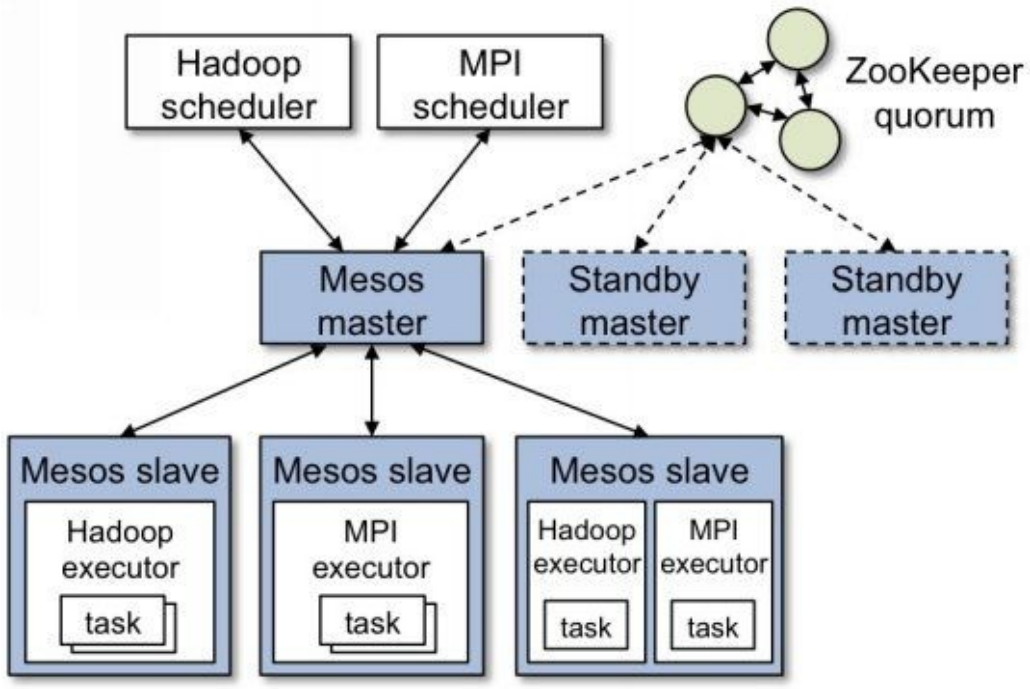


Fig 11-2 : Mesos architecture

The above figure shows the main components of Mesos. Mesos consists of a master daemon that manages slave daemons running on each cluster node, and mesos applications (also called frameworks) that run tasks on these slaves.

The master enables fine-grained sharing of resources (CPU, RAM) across applications by making them resource offers. Each resource offer contains a list of resources from a single slave. The master decides how many resources to offer to each framework according to a given organizational policy, such as fair sharing, or strict priority. To support a diverse set of policies, the master employs a modular architecture that makes it easy to add new allocation modules via a plugin mechanism.

A framework running on top of Mesos consists of two components: a scheduler that registers with the master to be offered resources, and an executor process that is launched on slave nodes to run the frameworks. While the master determines how many resources are offered to each framework, the frameworks' schedulers select which of the offered resources to use. When a framework accepts offered resources, it passes to Mesos a description of the tasks it wants to run on them. In turn, Mesos launches the tasks on the corresponding slaves.

High availability

High availability for a Mesos cluster is achieved by Apache zookeeper. The masters are replicated by zookeeper to form a quorum. Cluster leader is selected by zookeeper and it helps in detecting the leader for other cluster components like slaves and frameworks.

For a high availability Mesos cluster architecture, at least three master nodes should be configured to maintain the quorum even if one master node fails. For a resilient production setup, at least five master nodes should be configured by maintaining the quorum with two offline masters.

Mesosphere

Mesosphere is a software solution which works on top of Apache Mesos. Using mesosphere you can use all the capabilities of Apache Mesos with additional components to manage an infrastructure. For scaling applications, you can use frameworks like marathon and chromos with mesosphere by eliminating a lot of challenges associate with application scaling. Following are the main features provided by mesosphere

1. Application scheduling
2. Application scaling
3. Fault-tolerance
4. Self-healing
5. Service discovery

Till now we have discussed the basics of Apache Mesos.

In the next section we will learn the following.

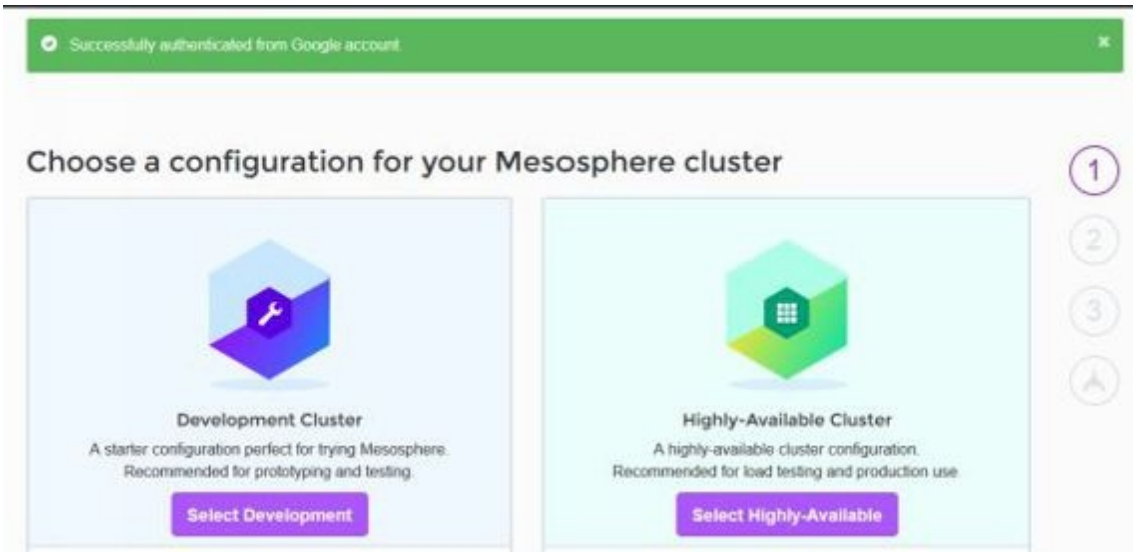
1. Mesos cluster setup on google compute engine using mesosphere
2. Deploying Docker containers on to the cluster using marathon framework.
3. Scaling up and scaling down Docker container on the cluster.

Cluster setup using mesosphere

We will launch our Mesos cluster on google compute engine. Follow the steps given below to setup the Mesos cluster.

1. Got to https://google.mesosphere.io/

2. Click the get started option. It will ask you to authenticate to your google compute account. Once authenticated, click start development button.

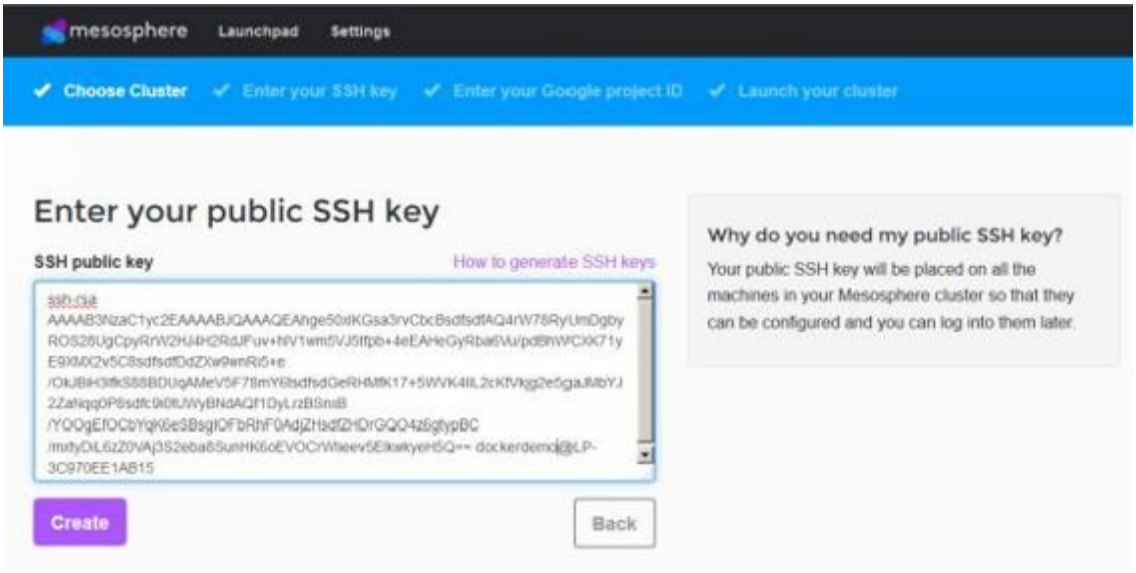


3. You will be asked to enter the ssh public key. If you have one, you can continue with the next step, if not , create a ssh public key using the following command,

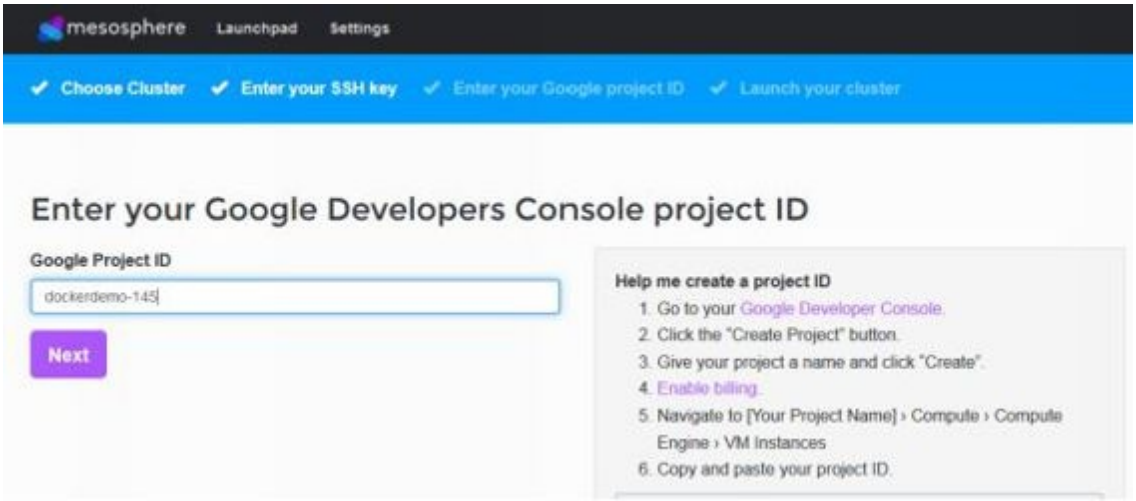
```
ssh-keygen
```

```
dockerdemo@LP-3C970EE1AB15 ~
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/dockerdemo/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/dockerdemo/.ssh/id_rsa.
Your public key has been saved in /c/Users/dockerdemo/.ssh/id_rsa.pub.
The key fingerprint is:
b4:a6:59:72:22:21:28:3c:9e:de:7e:c8:1d:1c:e7:a9 dockerdemo@LP-3C970EE1AB15
dockerdemo@LP-3C970EE1AB15 ~
```

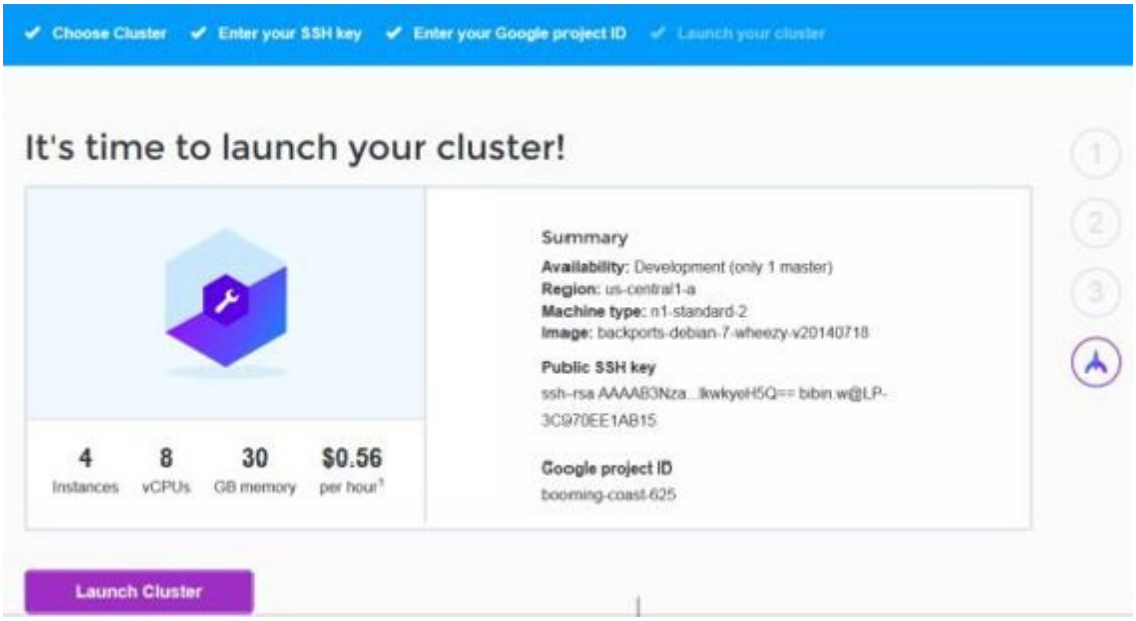
4. Copy the contents of id_rsa.pub file and paste in the ssh-public key text box.



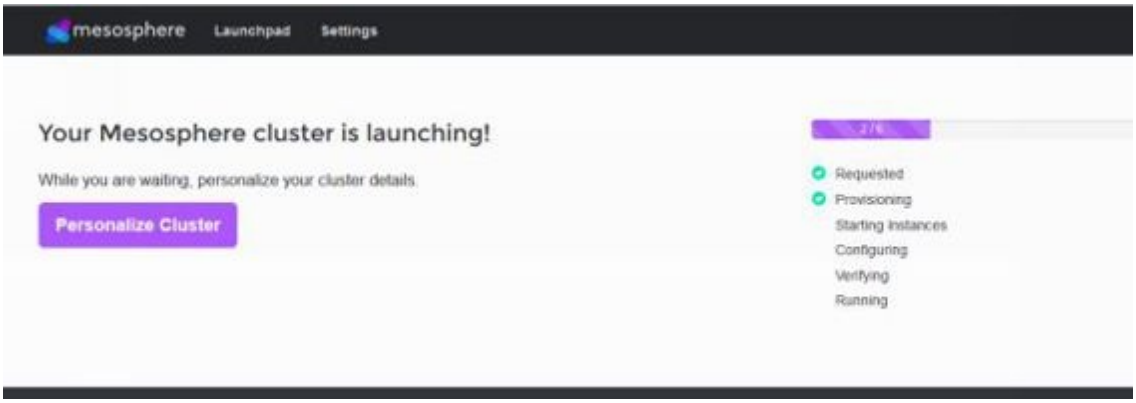
5. Click the create option and enter your compute engine project id in the next page and click next. You can follow the instruction in the page to get your compute engine project id.



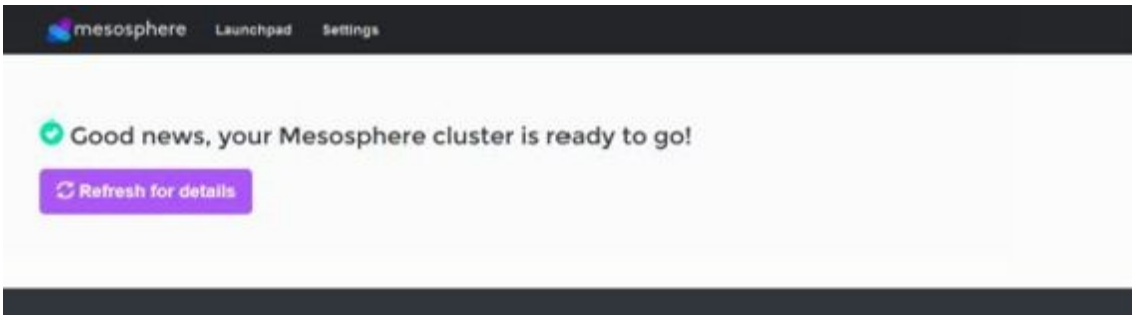
6. Now you will see all the instance specifications and cost for your mesos development cluster. Your cluster will have 4 instances, 8vCPU's and 30 GB memory .Click "launch cluster" to launch your mesos cluster.



7. Now you will see the status of you launching cluster.

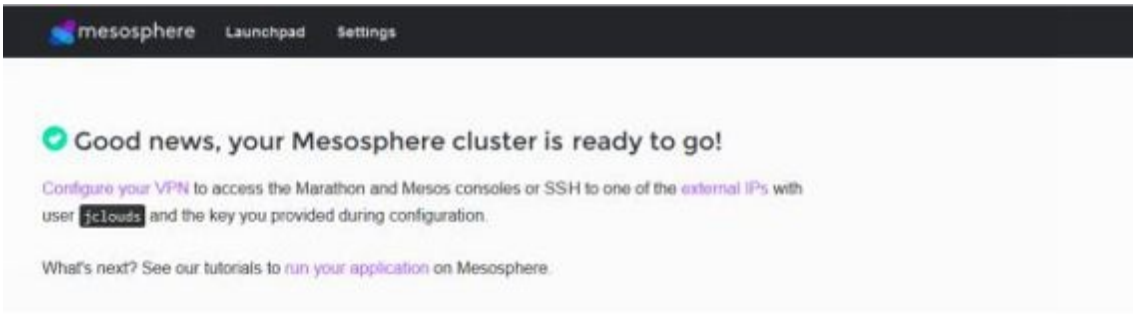


8. Once your cluster is ready, you will get a message as shown in the image below.



9. Click refresh details option to see the instructions to connect to mesos console.

You need to configure OpenVPN to access mesos and marathon consoles.



Setting up OpenVPN

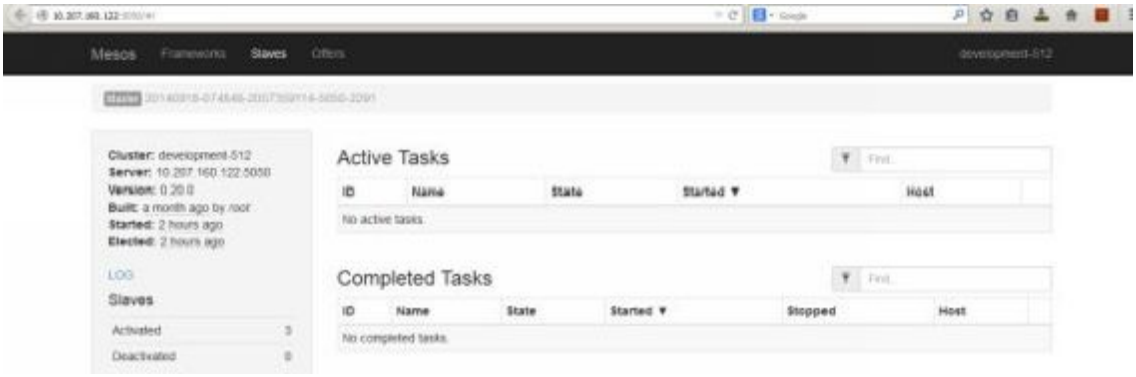
1. If you scroll down the mesosphere console, you will see instructions to download OpenVPN client and the openVPN config file generated my mesosphere to access marathon and mesos consoles.

2. Once openVPN is installed on your system, right click the config file and click

"*start openVPN on this config file*". It will connect your machine to Google compute using the VPN gateway created by mesosphere.

3. Once connected, click marathon button to view marathons console.

4. You can see view mesos console by clicking the mesos button.



5. Also, if you go to your compute engine dashboard, under VM instances option, you can view all four launched instances for your mesos cluster.

Deploying Docker containers using marathon

Note: Docker has to be installed on all the mesos slaves. Compute engine instance deployed with mesosphere comes bundled with Docker.

Docker containers can be deployed to mesos cluster using marathon REST API's. Follow the steps given below to deploy container on the mesos cluster.

1. Create a JSON file named Docker.JSON and save the file with following contents.

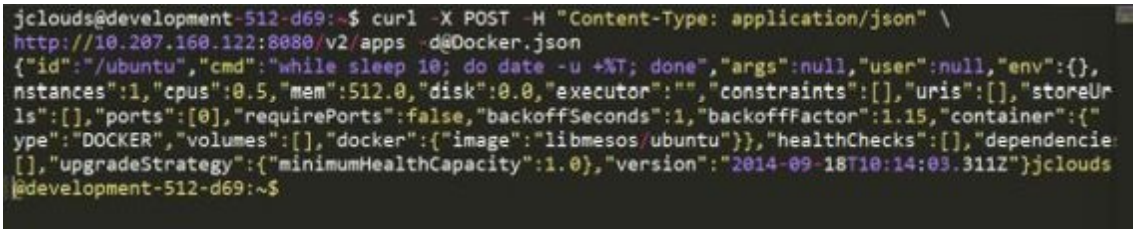
```
{
 "container": {
  "type": "DOCKER",
```

```
  "Docker": {
    "image": "training/postgres"
  }
 },
 "id": "database",
 "instances": "1",
 "cpus": "0.5",
 "mem": "512",
 "uris": [],
 "cmd": "while sleep 10; do date -u +%T; done"
}
```

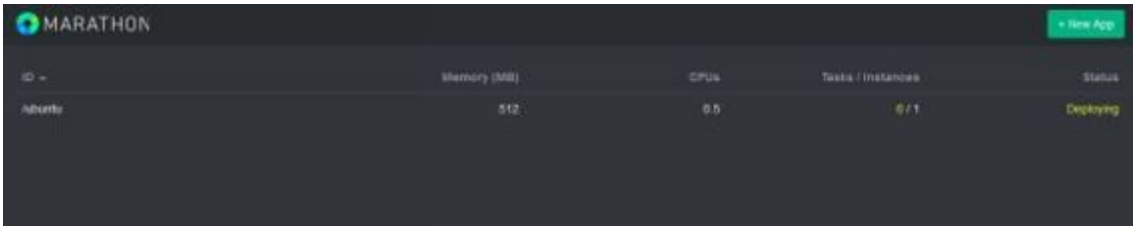
2. Now, you need to post a task to marathon using the JSON file, which can be done using curl. If curl is not installed on your system, install curl and run the following command.

```
curl -X POST -H "Content-Type: application/JSON" \
http://<master>:8080/v2/apps -d@Docker.JSON
```

In the above command, replace master with your mesos master IP which is running marathon.



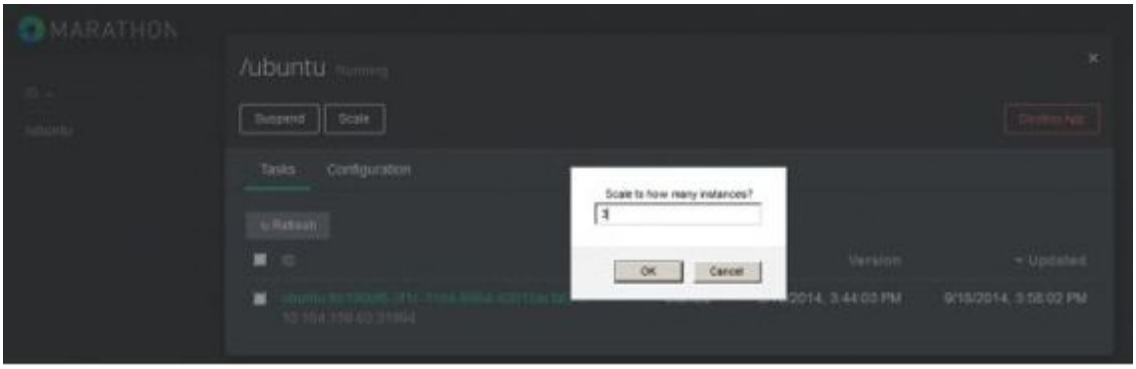
After successful execution of the above command, you can see the deploying app in marathon console.



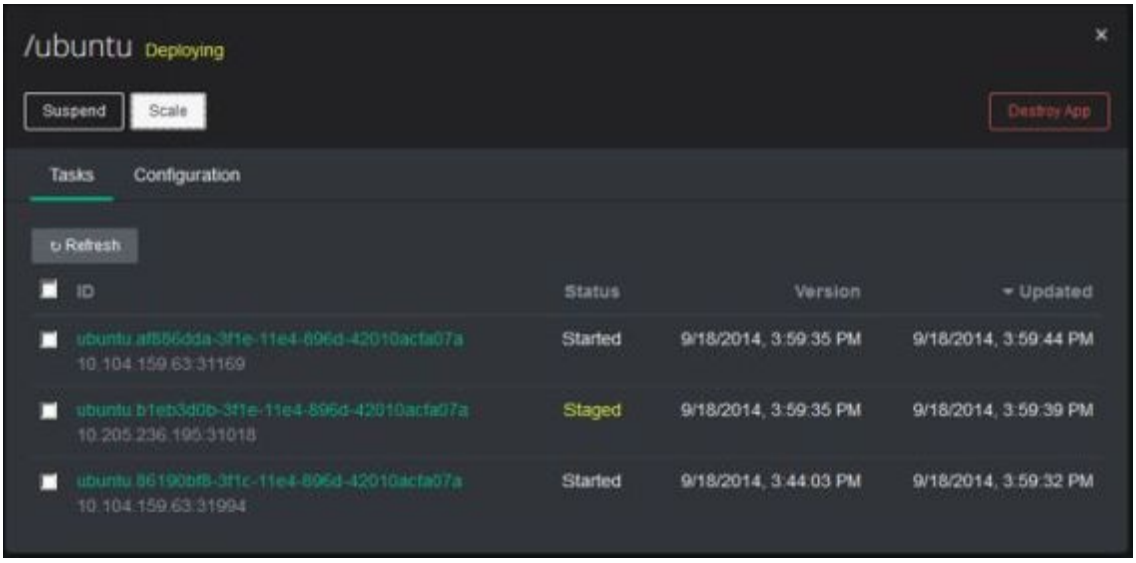
Scaling up Docker containers:

Docker instances can be scaled up very easily using marathon. Click the deployed application and select the scale option. Give it a number, e.g.: 3 and

click ok.

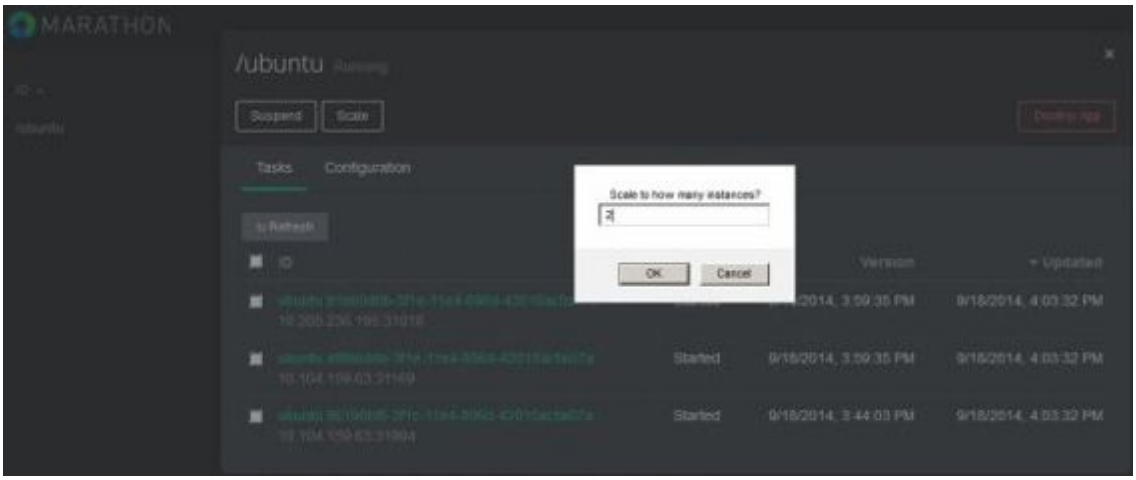


Marathon will scale the Docker containers to 3.

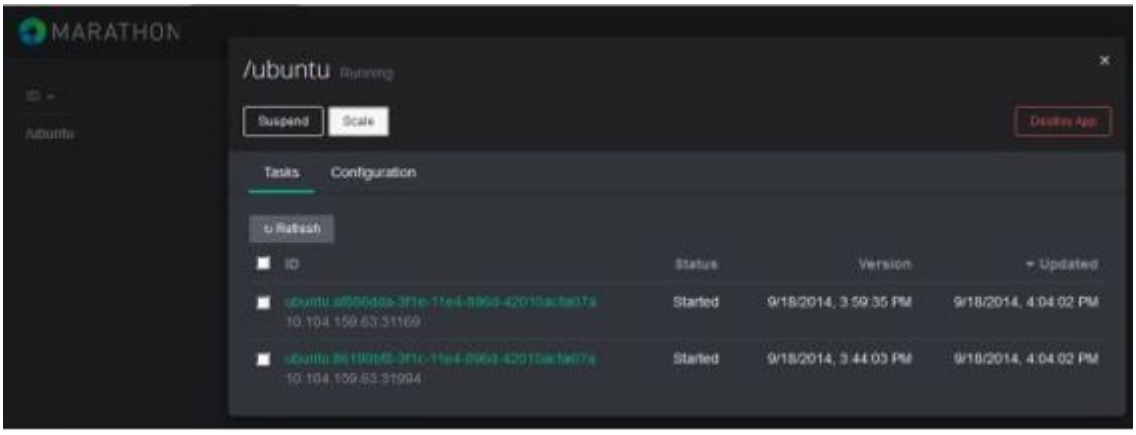


Scaling down Docker containers

Docker containers can be scaled down in the same way we scaled up the containers. Click the scale option and give a number of containers you want to scale down.

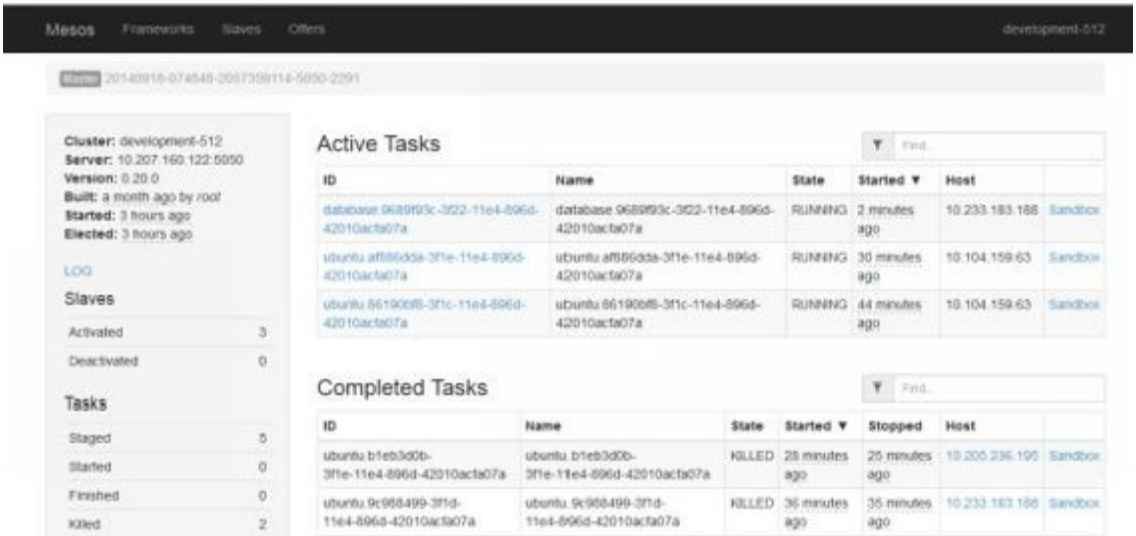


After successful execution, you can see that the containers have been scaled down to 2.



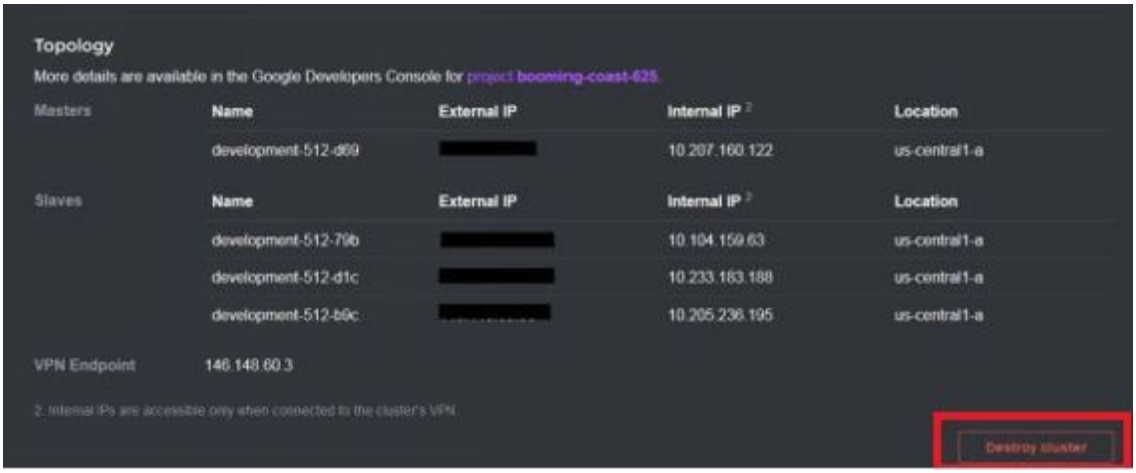
You can use the mesos console to view container deployment status.

Also you can see in which host a particular image has been deployed.



Tearing down the cluster

Once you are done working with the development cluster, you can tear down the cluster from mesosphere console. Just click the “destroy cluster option” in your project window.



Once you click ok, your cluster will start shutting down.

Docker cluster management using Kubernetes

Kubernetes is a cluster management tool developed by Google for managing containerized applications. You can make a bunch of nodes appear as a one big computer and deploy container applications to your public cloud and private cloud. It abstracts away the discrete nodes and optimizes compute resources.

Kubernetes uses a declarative approach to get the desired state for the applications mentioned by the user. When an application is deployed on a kubernetes cluster, the kubernetes master node decides in which underlying host the application has to be deployed. Kubernetes scheduler does the job of application deployment. Moreover, kubernetes self-healing, auto restarting, replication and rescheduling mechanisms make it more robust and suitable for container based applications.

Kubernetes components

Kubernetes consists of two main components, a master server and a minion server. Let’s have a look at the two components and services associated with

each component.

Master server

Master server is the controlling unit of kubernetes. It acts as the main management component for users to deploy applications on the cluster. Master server comprises of various components for operations like scheduling, communication etc.

Following are the services associated with kubernetes master server.

1. etcd
2. API server
3. Controller manager
4. Scheduler

Etcd

Etcd is a project developed by CoreOS team. It is a distributed key/value store that can be made available on multiple nodes in the cluster. Etcd is used by kubernetes to store the configuration data that can be used by the nodes in the cluster. All the master states are stored in an etcd instance residing on the kubernetes master. It stores all the configuration data. The watch functionality of etcd notifies components for all the changes in the cluster. Etcd can be queried using HTTP API's for retrieving values for a node.

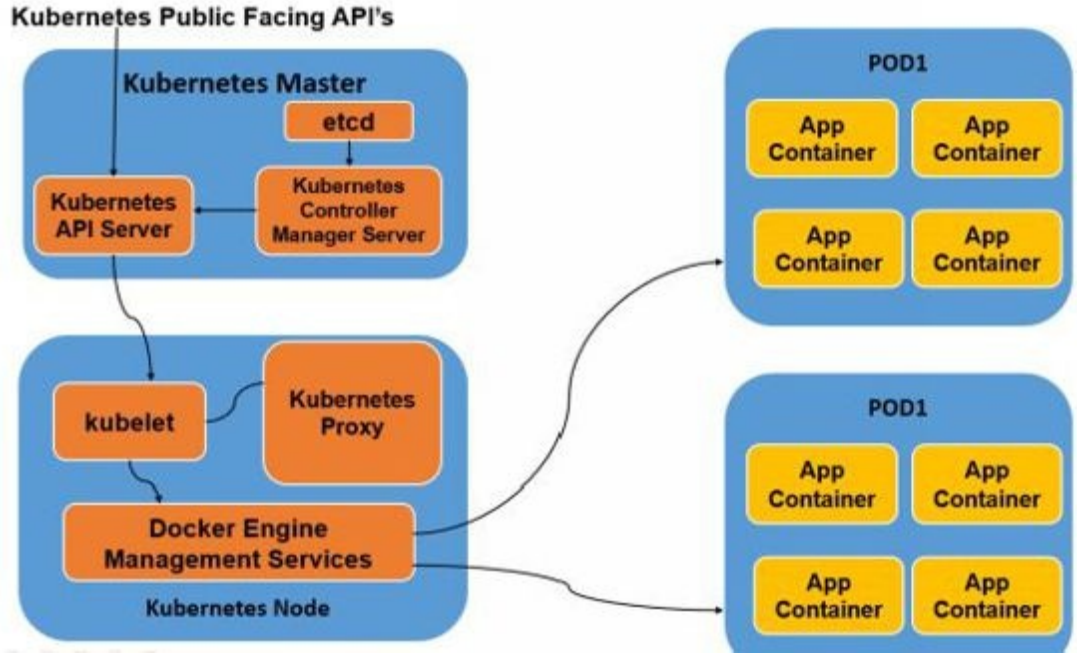


Fig 11-3 : Mesos architecture

API server

API server is an important service run buy the master server. It acts as the central hub for the user to interact with the master server. All the communication to the API server is carried out through Restful API's so that, you can integrate other tools and libraries to talk to the kubernetes cluster.

A lightweight client tool called kubecfg comes bundled with the server tools and you can use kubecfg from a remote machine to talk to the master server.

Controller manager

The controller manager is the acting component for container replication. When a user submits a request for replication, the details of the operation are written to etcd. The controller manager always watches etcd for configuration changes. When it sees a replication request in etcd, it starts replicating the containers as per specifications mentioned in the request.

The request can be made for scaling up and scaling down the containers. During replication, if the specified container number is less than the running containers, then kubernetes will destroy the excess running containers to meet the condition specified by the replication manager.

Scheduler

Scheduler is responsible for assigning the workloads to specific hosts in the cluster. It reads the operating requirements for a container and analyzes the cluster for placing the container on to an acceptable host. The scheduler keeps track on the cluster resources, it knows the resources available in a specific node in the cluster and keeps track of the resources used by individual containers.

Minion server

The worker nodes in the kubernetes cluster are called minions. Each minion server should have few services running for networking, communication with the master and for deploying the workloads assigned to it. Let's have a look at each service associated with the minion server.

Docker

Each minion server should run an instance of Docker daemon in it. Docker daemon will be configured with a dedicated subnet on the host.

Kubelet service

Minions connect to kubernetes master server using kubelet. It is responsible for relaying messages to and from the kubernetes master server and it interacts with etcd to store and retrieve configurations. Kublet communicates with the maser server to get the required commands and deployment tasks.

All deployment tasks will be received by minions in the form of manifests. The manifest will contain the rules and desired stated for a container deployment. Once the manifest is received, kubelet will maintain the state of container as specified in the manifest file.

Kubernetes proxy

All the services running in a host should be available for services running in other hosts. In order to deal with the subnets and communication across the hosts, kubernetes runs a proxy server on all the minions. The main responsibility of the proxy server is to isolate the networking environment and make the containers accessible to other services. The proxy server directs the traffic to the respective container in the same host or a different host in the cluster.

Work Units

There are various types of work units associated with container deployment on kubernetes cluster. Let’s have a look at each type of work units.

Pods

A pod is a group of related containers which are deployed on the same host. One or more containers in a pod are treated as a single application. For example, a fleet of web server can be grouped in to a pod. Pods share the same environments and treated as a unit. Applications grouped in to pods can share volumes, IP space and can be scaled as a single unit.

Services

Services offer discover-ability to applications running in the kubernetes cluster.

Service is more of a named load balancer and acts as an interface to a group of containers. You can create service unit which is aware of all the backed services. It acts as a single access point for applications. For example, all the web server containers can access the application containers using the single access point. By this mechanism, you can scale up and down the application containers and the access point remains the same.

Replication controllers

All the pods which have to be horizontally scaled are defined by the replication controller. Pods are defined in a template. This template has all the definition for the replication process. Let's say, a pod has a replication entry for four containers and it is deployed on a host. If one container fails out of four, the replication controller will create another container to meet the specification. If the failed container comes up again, the replication controller will kill the newly created container.

Labels

Labels are the identification factor for the pods. Labels are basically tags for pods and it is stored as a key value in etcd. Label selectors are used for services (named load balancers) and replications. To find a group of backend servers, you can use the pods label.

Till now we have learnt about the concepts involved in kubernetes.

In the next section we will learn how to launch a kubernetes cluster.

Installation

In this section we will learn how to launch a kubernetes cluster on google compute engine. You need to have a google compute engine account to try on the steps given below.

Configuring workstation

You can configure the workstation in your laptop .Workstation should have the following

1. Configured google cloud sdk to launch instances.
2. Access to all compute engine resource API's
3. Go > 1.2 installed

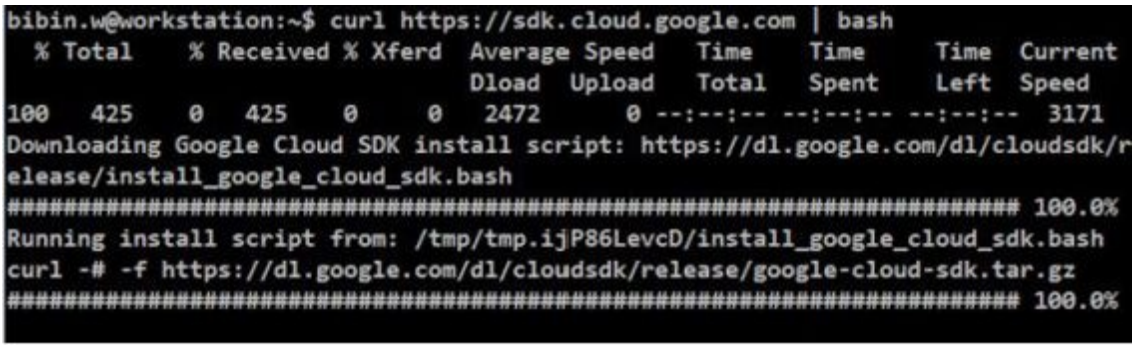
4. Kubernetes launch script.

Configuring google cloud sdk on workstation

Connect the instance using an SSH client like putty using the key generated during the first instance launch and follow the steps given below.

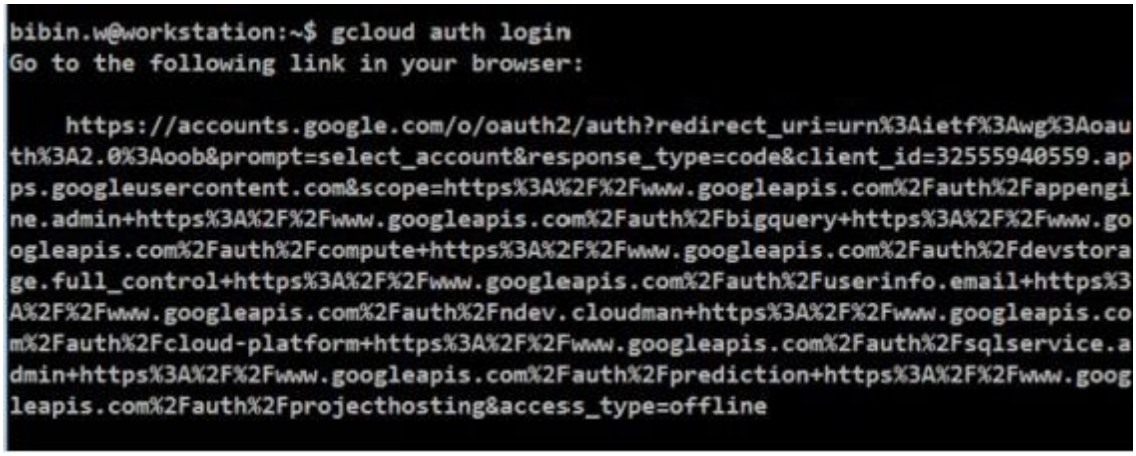
1. Install Google Cloud SDK on the workstation instance using the following command.

curl https://sdk.cloud.google.com | bash



2. Authenticate to google cloud services using the following command.

gcloud auth login



3. Check the SDK authentication by list the instances using cli.

gcutil listinstances

```
bibin.w@workstation:~$ gcutil listinstances
+-------------+--------------+---------+---------------+-----------------+
| name        | zone         | status  | network-ip    | external-ip     |
+-------------+--------------+---------+---------------+-----------------+
| workstation | asia-east1-a | RUNNING | 10.240.40.222 | 107.167.186.138 |
+-------------+--------------+---------+---------------+-----------------+
```

4. Install latest go using gvm using the following commands. You need go version 1.3 to work with kubernetes cli.

```
sudo apt-get install curl git mercurial make binutils bison gcc build-essential
bash < <(curl -s -S -L https://raw.githubusercontent.com/moovweb/gvm/master/bins installer)
gvm install go1.3
```

```
bibin.w@workstation:~$ gvm install go1.3
Downloading Go source...
Installing go1.3...
 * Compiling...
bibin.w@workstation:~$ gvm use go1.3 [--default]
Now using version go1.3
bibin.w@workstation:~$ go version
go version go1.3 linux/amd64
```

*Note: before launching the kubernetes cluster, launch an instance in google compute from your workstation or the web interface and generate a private key using "**gcutil ssh <servername>**" command. Because the launch script needs the ssh key in your workstation to create instances in compute engine.*

Launching kubernetes cluster from the workstation

Kunbernetes cluster can be launched using the launch script file from the kubernetes source. The default script launches a kubernetes master and four minions of small instances using the private key present inside the workstation. These configurations can be changed in the launch configuration file.

Follow the steps given below to launch the kubernetes cluster.

1. Clone the kubernetes source files from github using the following url

```
git clone https://github.com/GoogleCloudPlatform/kubernetes.git
```

```
bibin.w@workstation:~/.ssh$ cd ~
<clone https://github.com/GoogleCloudPlatform/kubernetes.git
Cloning into 'kubernetes'...
remote: Reusing existing pack: 7031, done.
remote: Counting objects: 79, done.
remote: Compressing objects: 100% (75/75), done.
remote: Total 7110 (delta 36), reused 5 (delta 0)
Receiving objects: 100% (7110/7110), 6.40 MiB | 965 KiB/s, done.
Resolving deltas: 100% (4025/4025), done.
```

2. The configurations for cluster instances are present inside /kubernetes/cluster/config-default.sh file.

```
- Copyright 2014 Google Inc. All rights reserved.
- Licensed under the Apache License, Version 2.0 (the "License"); -
you may not use this file except in compliance with the License. -
You may obtain a copy of the License at
http://www.apache.org/licenses/LICENSE-2.0
- Unless required by applicable law or agreed to in writing,
software - distributed under the License is distributed on an "AS
IS" BASIS, - WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either
express or implied. - See the License for the specific language
governing permissions and - limitations under the License.
- TODO(jbeda): Provide a way to override project
FONE=us-centrall-b
MASTER_SIZE=g1-small
MINION_SIZE=f1-micro
NUM_MINIONS=4 - gcloud/gcutil will expand this to the latest
supported image.
IMAGE=backports-debian-7-wheezy
```

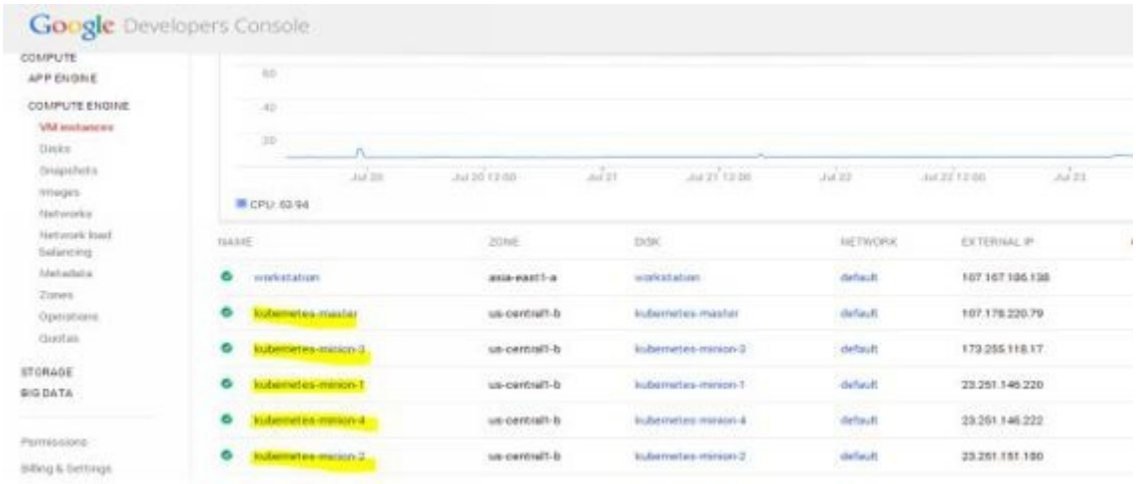
3. Execute the following command to launch the kubernetes cluster.

hack/dev-build-and-up.sh

The above command will launch a kubernetes cluster with one master and four minions. dev-build-and-up.sh script configures the kubernetes cluster with Docker and kubernetes agents.

```
bibin.w@workstation:~/kubernetes$ hack/dev-build-and-up.sh
Building release tree ~/kubernetes ~kubernetes
~/kubernetes
Packaging release
Building launch script
Uploading to Google Storage
Release pushed (devel/bibin.w/r20|40723-093855).
+.........| status |  insert-time | operation-type |
+ .....................................................+.............+.............
.........+.............
| operation-|406|08397462-4fed9|f8be|f0-aaab40a|-36544|eb | DONE | 20|4-
07-23T02:39:57.923-07:00 | insert
.....................................................................................
.................
 Kubernetes cluster created. Warning: Permanently added '23.25|.|46.222' (
 kubelet is running.
 Kubernetes cluster is running.
 Access the master at: https://admin:Wvm7eKlIALfjqXC 107.178.220.76
 Security note: The server above uses a self signed certificate. This is
 subject to "Man in the middle" type attacks.
 bibin.w@workstation:~/kubernetes$
```

In the compute engine console you can see the launched instances.



Deploying a multi-tier application on kubernetes cluster with Docker

In this example, a multi-tier guestbook application (frontend, redis slave and master) will be deployed using preconfigured Docker containers.

The pod description files (JSON files) for deploying this application are present inside kubernetes source file under /kubernetes/examples/guestbook directory.

Note: all the commands executed in this example are executed in the kubernetes root folder.

Before deploying the pods, you have to set up the go workspace by executing the

following command.

hack/build-go.sh

```
bibin.w@workstation:~/kubernetes$
bibin.w@workstation:~/kubernetes$ hack/build-go.sh
+++ Building proxy
+++ Building integration
+++ Building apiserver
+++ Building controller-manager
+++ Building kubelet
+++ Building kubecfg
```

Pod configuration file

The pod configuration file can be formatted as a Json template. The configuration file supports the following fields.

```
{
 "id": string,
 "kind": "Pod",
 "apiVersion": "v1beta1",
 "desiredState": {
  "manifest": {
   manifest object
  }
 },
 "labels": { string: string }}
```

Where,

Id: indicates the name of the pod

Kind: It is always Pod.

apiVersion: At the time of writing it is v1beta1.

desiredState: It is an object with a child manifest object.

Manifest: manifest contains the fields mentioned in the following table.

Field Name	Type	Description
Version	string	The version of the manifest. Must be v1beta1 .
containers[]	List	The list of containers to launch.

containers[].name	string	User defined name for the container
containers[].image	string	The image to run the container
containers[].command[]	list	Command to run when a container is launched
containers[].volumeMounts[]	List	Data volumes that has be exposed
containers[].ports[]	List	List of container ports that has to be exposed
containers[].env[]	List	Sets the environment variables for the container
containers[].env[].name	string	Name of environment variable
containers[].env[].value	string	Value for the environment variable
containers[].ports[].hostPort	Int	Host to container mapping port number

Getting started

Kubernetes scheduler will decide in which host the pod has to be deployed.

Follow the steps given below to deploy containers on the kubernetes cluster.

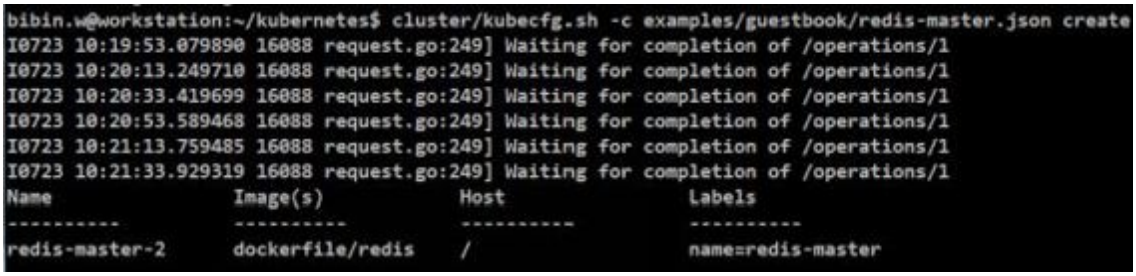
1. The following JSON file will create a redis master pod on the kubernetes cluster. The attributes used in the JSON file are self-explanatory. It uses Dockerfile/redis Docker preconfigure image from Docker public repository to deploy the redis master on kubernetes cluster.

```
{
 "id": "redis-master-2",
 "desiredState": {
  "manifest": {
   "version": "v1beta1",
   "id": "redis-master-2",
   "containers": [{
    "name": "master",
```

```
      "image": "Dockerfile/redis",
      "ports": [{
        "containerPort": 6379,
        "hostPort": 6379
      }]
    }]
  }
 },
 "labels": {
  "name": "redis-master"
 }
}
```

2. Execute the following command to deploy the master pod in cluster.

```
cluster/kubecfg.sh -c examples/guestbook/redis-master.JSON create pods
```



3. You can list the pods and see in which host the pod has been deployed by running the following command using kubecfg CLI.

```
cluster/kubecfg.sh list /pods
```

As you can see, the master pod has been deployed in the minion-1.

4. Once the master pod is up, you have to create a service (named loadbalancer) for master, so that the slave nodes will route the traffic to the master.

The service description for the master looks like the following JSON file:

```
{
 "id": "redismaster",
 "port": 10000,
 "selector": {
  "name": "redis-master"
 }
}
```

5. Execute the following command to create a service for the master.

```
cluster/kubecfg.sh -c examples/guestbook/redis-master-service.JSON create services
```

```
bibin.w@workstation:-./kubernetes$ cluster/kubecfg.sh -c examples/guestbook/
redis-master-service.json create services
10723 10:39:34.990114 16262 request.go:249] Waiting for completion of /
operations/2
Name            Labels              Selector                  Port
--------        ---------           -------                 --------
redismaster                        name=redis-master          10000
```

6. Create 2 redis slaves using the following pod description.

```
{
  "id": "redisSlaveController",
  "desiredState": {
   "replicas": 2,
   "replicaSelector": {"name": "redis-slave"},
   "podTemplate": {
    "desiredState": {
     "manifest": {
      "version": "v1beta1",
      "id": "redisSlaveController",
      "containers": [{
       "image": "brendanburns/redis-slave",
       "ports": [{"containerPort": 6379, "hostPort": 6380}]
      }]
     }
    },
    "labels": {"name": "redis-slave"}
   }},
  "labels": {"name": "redis-slave"} }
```

7. Execute the following command to deploy 2 slave pods on the cluster.

```
cluster/kubecfg.sh -c examples/guestbook/redis-slave-controller.JSON create replicationControllers
```

The above command used brendanburns/redis-slave Docker image to deploy the slave pods.

```
bibin.w@workstation:~/kubernetes$ cluster/kubecfg.sh -c examples/guestbook/
redis-slave-controller.json create replicationControllers
10723 10:46:58.589757 16326 request.go:249] Waiting for completion of /
operations/3
10723 10:47:18.755887 16326 request.go:249] Waiting for completion of /
operations/3
10723 10:47:38.921947 16326 request.go:249] Waiting for completion of /
operations/3
Name                Image(s)                Selector              Replicas
redisSlaveController brendanburns/redis-slave name=redisslave       2
```

8. The redis slaves need a service (named load balancer) so that it can talk to frontend pods. This can be created using the following service description.

```
{
 "id": "redisslave",
 "port": 10001,
 "labels": {
  "name": "redis-slave"
 },
 "selector": {
  "name": "redis-slave"
 }
}
```

9. Execute the following command to create the slave service.

```
cluster/kubecfg.sh -c examples/guestbook/redis-slave-
service.JSON create services
```

```
bibin.w@workstation:~/kubernetes$ cluster/kubecfg.sh -c examples/guestbook/
redis-slave-service.json create services
10723 10:53:13.413178 16379 request.go:249] Waiting for completion of /
operations/6
Name           Labels          Selector           Port
-------        -------         --------           ------
redisslave    name=redisslave  name=redisslave    10001
```

10. Frontend redis description file creates 3 frontend replicas. The pod description for redis frontend looks like the following.

```
{
 "id": "frontendController",
 "desiredState": {
  "replicas": 3,
  "replicaSelector": {"name": "frontend"},
  "podTemplate": {
   "desiredState": {
    "manifest": {
     "version": "v1beta1",
     "id": "frontendController",
     "containers": [{
      "image": "brendanburns/php-redis",
      "ports": [{"containerPort": 80, "hostPort": 8000}]
     }]
    }
   },
   "labels": {"name": "frontend"}
  }},
 "labels": {"name": "frontend"}
}
```

11. Frontend pod uses brendanburns/php-redis Docker image to deploy redis frontend replicas. Execute the following command to deploy the frontend pods using the frontend pod description file.

```
cluster/kubecfg.sh -c examples/guestbook/frontend-controller.JSON create replicationControllers
```

```
bibin.w@workstation:-./kubernetes$ cluster/kubecfg.sh -c examples/guestbook/
frontend-controller.json create replicationControllers
10723 10:57:14.558738 16429 request.go:249] Waiting for completion of /
operations/7
10723 10:57:34.728778 16429 request.go:249] Waiting for completion of /
operations/7
10723 10:57:54.898487 16429 request.go:249] Waiting for completion of /
operations/7
10723 10:58:15.068429 16429 request.go:249] Waiting for completion of /
operations/7
10723 10:58:35.238101 16429 request.go:249] Waiting for completion of /
operations/7
10723 10:58:55.407921 16429 request.go:249] Waiting for completion of /
operations/7
10723 10:59:15.577644 16429 request.go:249] Waiting for completion of /
operations/7
Name                Image(s)              Selector      Replicas
---------           --------            ----------      --------
frontendController brendanburns/php-redis name=frontend   3
```

12. If you list the pods in kubernetes cluster you can see in which minion the frontend pods have been deployed

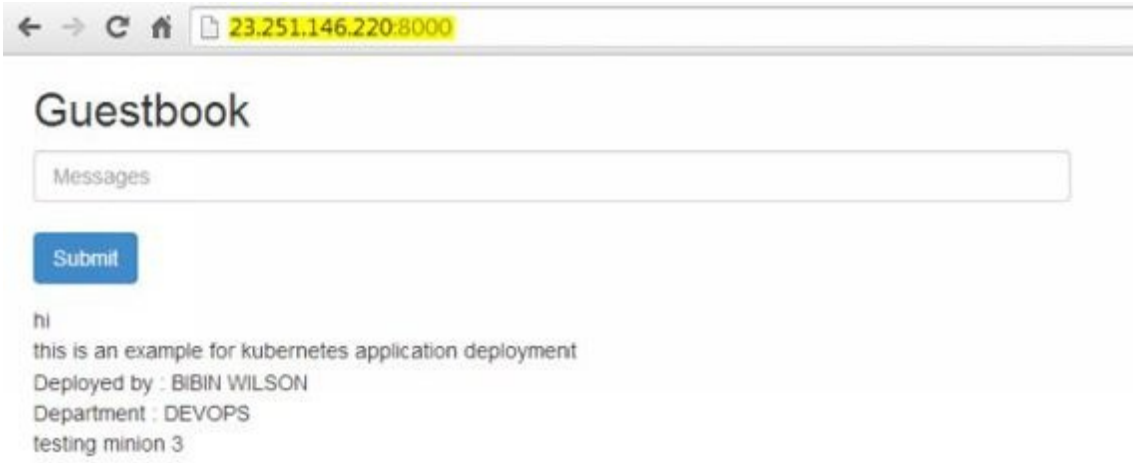
cluster/kubecfg.sh list /pods

```
bibin.w@workstation:~/kubernetes$ cluster/kubecfg.sh list /pods
Name                Image(s)               Host
--------            ------------           ------------
redis-master-2      dockerfile/redis            kubernetes-minion-1.c.booming-
coast-625.internal/
e77a43a1-1256-11e4-b7e2-42010af007f2 brendanburns/redis-slave   kubernetes-
minion-1.c.booming-coast-625.internal/
roller=redisSlaveController
94a9b121-1258-11e4-b7e2-42010af007f2 brendanburns/php-redis kubernetes-minion-1.
c.booming-coast-625.internal/
ller=frontendController abde9ff5-1256-11e4-b7e2-42010af007f2 brendanburns/redis-
slave kubernetes-minion-3.c.booming-coast-625.internal/
roller=redisSlaveController 64f9d6b5-1258-11e4-b7e2-42010af007f2 brendanburns/
php-redis kubernetes-minion-3.c.booming-coast-625.internal/
ller=frontendController 1d721ba5-1258-11e4-b7e2-42010af007f2 brendanburns/php-
redis kubernetes-minion-4.c.booming-coast-625.internal/
ller=frontendController
```

The above image shows that front end pods have been deployed in minions 1, 3 and 4.

You can view the deployed guestbook application in the browser by grabbing any one of the public IP's of the minion in which the frontend has been deployed. You will be able to access the frontend from all the three minions in which the front end has been deployed.

The following image shows the final deployed redis three tiered application.



Deleting the cluster

By deleting the cluster, you will delete all the computing engine configurations associated with the cluster. All the networks, instances, and route will be deleted from the project.

The cluster can be brought down using the following command.

cluster/kube-down.sh

```
bibin.w@workstation:~/kubernetes$ cluster/kube-down.sh
Project: booming-coast-625 (autodetected from gcloud config)
Bringing down cluster
INFO: Enabling auto-delete on kubernetes-master (kubernetes-master).
INFO: Waiting for delete of firewall kubernetes-master-https. Sleeping for 3s
+-------------------------------------------------------+--------+-------------------------------+----
------------+
| name                                                  | status | insert-time                   |
operation-type |
+-------------------------------------------------------+--------+-------------------------------+----
------------+
| operation-1406122829466-4fedc7bc2c990-b49284d2-e92e3cf2 | DONE   | 2014-07-23T06:40:30.299-07:00 |
delete         |
+-------------------------------------------------------+--------+-------------------------------+----
------------+
INFO: Waiting for delete of instance kubernetes-master. Sleeping for 3s.
stty: standard input: Inappropriate ioctl for device
+-------------------------------------------------------+--------+-------------------------------+----
------------+
| name                                                  | status | insert-time                   |
operation-type |
+-------------------------------------------------------+--------+-------------------------------+----
------------+
| operation-1406122829841-4fedc7bc88269-f677d4ef-09140e06 | DONE   | 2014-07-23T06:40:30.668-07:00 |
delete         |
+-------------------------------------------------------+--------+-------------------------------+----
```

Docker orchestration using CoreOS

and fleet

CoreOS website defines coreOS as “Linux for Massive Server Deployments. CoreOS enables warehouse-scale computing on top of a minimal, modern operating system”.

CoreOS is a Linux distribution based system designed to run services as containers and aimed at running high availability clusters. The overall design of coreOS aims at clustering and containerization.

CoreOS has the following features

1. CoreOS has an update system like in chromeOS which downloads the latest patches and updates automatically and configures itself every time you reboot your machine. So there is never a point of time where your system is in an unstable state.
2. It has a distributed key value store etcd which helps in coordinating a group of servers to share configurations and service discovery.
3. Isolates services using Docker containers. It does not have a package manager instead you have to use Docker containers for running your applications.
4. Fleet cluster management tool to manage the cluster and services
5. Easy cluster configuration using the cloud-config file given by user , which coreOS reads during boot

CoreOS Architecture

In this section we will learn about the components in coreOS architecture.

Etcd

Etcd is a distributed configuration store like consul and zookeeper. Etcd runs on all the hosts in the cluster. When a cluster is launched, one of the etcd instances becomes the master and shares the logs with other hosts and if the master goes down, another working etcd instance will become the master. All the applications running in coreOS cluster can write to etcd. This enables applications to discover services it needs in a distributed environment. The information about the services is distributed globally by etcd, so applications can connect to etcd interface and query for the service it needs. Etcd is also used by

cluster management tool like kubernetes for service discovery and cluster coordination.

Systemd

Systemd is the system management daemon. It is not just used for managing services but it can be used to run scheduled jobs. Following are the features of system

1. It boots up the system really fast
2. Its logging system called journal had good features like JSON export and indexing.

Fleet

Fleet works on top of systemd and it schedules jobs for the cluster. Let’s say you want to run three instances of web containers on the cluster, fleet will launch those instances in the cluster without much configuration.

You can say fleet as a cluster management tool for CoreOS. You can also define conditions like no host should have the same instance of web containers and instead it should be distributed to different host machines.

Another important feature of fleet is that when a machine running a service fails, fleet will automatically reschedule the service to another machine in the cluster.

Units

A unit is a systemd file and referenced unit files. Once these unit files are pushed to the cluster, it will be immutable. For any modifications in the unit file that has been pushed can only be made by deleting and resubmitting the unit file to the cluster.

Fleetd

A fleetd daemon will run on every fleet cluster. Every daemon has an engine and agent role associated with it. Engine is responsible for scheduling units in the cluster. Least-loaded scheduling algorithm is used by the engine to decide on which host the unit has to be deployed. Agent is responsible for executing the units in the host. It reports the state of the unit to etcd.

Unit files

Before launching container in our cluster, we should know about unit files. Unit file are the basic unit of fleet. It is used to describe the service and commands to manage the service. A typical unit file is shown below.

```
[Unit]
Description=Hello World
After=Docker.service
Requires=Docker.service
[Service]
EnvironmentFile=/etc/environment
ExecStartPre=/usr/bin/etcdctl set /test/%m
${COREOS_PUBLIC_IPV4}
ExecStart=/usr/bin/Docker run --name test --rm busybox
/bin/sh -c "while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/etcdctl rm /test/%m
ExecStop=/usr/bin/Docker kill test
```

Let’s breakdown the unit file and see what each section really mean.

Unit

```
[Unit]
Description=Hello World
After=Docker.service
Requires=Docker.service
```

1. The unit header holds the common information about the unit and its dependencies.
2. Description can be any user defined description
3. “After=Docker.service” Conveys systemd to begin the unit after Docker.service
4. “Requires=Docker.service” convers system that Docker.service is required for normal operation.

Service

```
[Service]
EnvironmentFile=/etc/environment
ExecStartPre=/usr/bin/etcdctl set /test/%m
${COREOS_PUBLIC_IPV4}
ExecStart=/usr/bin/Docker run --name test --rm busybox
/bin/sh -c "while true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/etcdctl rm /test/%m
ExecStop=/usr/bin/Docker kill test
```

The service header is associated with start and stop commands. The parameters used in service header are explained below.

“EnvironmentFile=/etc/environment” – used for exposing environment variables for unit file.

“ExecStartPre” – this runs before the service for creating a key in etcd.

“ExecStart” –this starts the real service. In our case we will start a busybox container running echo in infinite loop.

“ExecStop” – this stops the action mention in it.

Till now we have learned about the features and components of coreOS.

In next section, you will learn how to deploy Docker containers on a coreOS cluster.

Launching a coreOS cluster

In this demo we will be using Google compute engine to launch our CoreOS cluster. Following are the requirements for this set up.

1. Google compute engine account
2. Google cloud SDK configured and authenticated to compute engine account on your local workstation.

Note: in this demo, a windows workstation is used. Follow the steps given below to launch a three node coreOS cluster.

1. In order to start fleet and etcd services on startup, we will use a yaml config file called cloud-config.yaml. In this file we will mention a discovery token for machines to find each other in the cluster using

etcd. You can create your own discovery token by visiting the following link.

https://discovery.etcd.io/new

2. Open gcloud SDK directory and create a cloud-config.yaml file and copy the following contents on to the file.

Note: create a new token form the link given on step 1 and replace that token with the token mentioned in the following snippet.

```
#cloud-config
coreos:
 etcd:
   discovery: https://discovery.etcd.io/<token>
  # multi-region and multi-cloud deployments need to use $public_ipv4
  addr: $private_ipv4:4001
  peer-addr: $private_ipv4:7001
 units:
  - name: etcd.service
    command: start
  - name: fleet.service
    command: start
```

3. Open gcloud SDK shell and run the gcloud CLI command given below to launch the cluster with nodes core1, core2 and core3. Make sure that the cloud-config file is present in the same directory as you are running the command.

Note: n1-standard-1 instance type is being used in the following snippet. You can modify it to a small or micro instance type based on your requirement.

```
gcloud compute instances create core1 core2 core3 --image \
https://www.googleapis.com/compute/v1/projects/coreos-cloud/global/images/coreos-stable-410-2-0-v20141002 \
--zone us-central1-a --machine-type n1-standard-1 \
--metadata-from-file user-data=cloud-config.yaml
```

```
C:\Program Files\Google\Cloud SDK>gcloud compute instances create core1 core2 core3 -
ompute/v1/projects/coreos-cloud/global/images/coreos-stable-410-2-0-v20141002 --zone
ndard-1 --metadata-from-file user-data=cloud-config.yaml
Created [https://www.googleapis.com/compute/v1/projects/cloud-repository/zones/us-cen
Created [https://www.googleapis.com/compute/v1/projects/cloud-repository/zones/us-cen
Created [https://www.googleapis.com/compute/v1/projects/cloud-repository/zones/us-cen
NAME  ZONE          MACHINE_TYPE  INTERNAL_IP    EXTERNAL_IP    STATUS
core1 us-central1-a n1-standard-1 10.240.24.9    [redacted]2    RUNNING
core2 us-central1-a n1-standard-1 10.240.9.120   [redacted]7    RUNNING
core3 us-central1-a n1-standard-1 10.240.56.247  [redacted]     RUNNING
C:\Program Files\Google\Cloud SDK>
```

On successful execution of above commands, you will have a working three node cluster with fleet and etcd configured.

4. We can control the cluster with fleet using fleetctl command. SSH in to any one host in our cluster, let's say core1 and execute the following command to list the available fleetctl commands.

fleetctl

```
bibin.w@core1 ~ $ fleetctl
NAME:
        fleetctl - fleetctl is a command-line interface to fleet, the cluster-wide Co
USAGE:
        fleetctl [global options] <command> [command options] [arguments...]
VERSION:
        0.6.2
COMMANDS:
        cat             Output the contents of a submitted unit
        destroy         Destroy one or more units in the cluster
        help            Show a list of commands or help for one command
        journal         Print the journal of a unit in the cluster to stdout
        list-machines   Enumerate the current hosts in the cluster
        list-unit-files List the units that exist in the cluster.
        list-units      List the current state of units in the cluster
        load            Schedule one or more units in the cluster, first submitting t
        ssh             Open interactive shell on a machine in the cluster
        start           Instruct systemd to start one or more units in the cluster, f
        status          Output the status of one or more units in the cluster
        stop            Instruct systemd to stop one or more units in the cluster.
        submit          Upload one or more units to the cluster without starting them
        unload          Unschedule one or more units in the cluster.
        verify          DEPRECATED - Verify unit file signatures using local SSH iden
        version         Print the version and exit
```

5. To make sure everything worked as expected run the following fleet command on core1 to list the servers in the cluster.

fleetctl list-machines

```
bibin.w@core1 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
0aa2a6f1...     10.240.9.120    -
77ee9831...     10.240.24.9     -
d8898912...     10.240.56.247   -
bibin.w@core1 ~ $
```

Launching containers on cluster using fleet

In this section we will look in to how to launch a container on the cluster using fleet unit file. Fleet decides on which host the container should be deployed. Follow the steps below to launch a container in our cluster using a unit file.

1. Create a unit file named hello-world.service on core1 and copy the snippet we have under unit files section.
2. Once created submit the unit file to fleet using fleetctl to schedule it on the cluster using the following command.

Note: make sure you run the fleetctl command from where you have the service file or you should give the full path of the file if you are running the command from some other location.

fleetctl submit hello-world.service

```
bibin.w@core1 ~/demo $ ls
hello-world.service
bibin.w@core1 ~/demo $ fleetctl submit hello-world.service
bibin.w@core1 ~/demo $
```

Now we have a unit file submitted to fleet for scheduling it to some host in the cluster.

Run the following fleetctl command to start the service.

fleetctl start hello-world.service

```
bibin.w@core1 ~/demo $ fleetctl start hello-world.service
Job hello-world.service launched on 0aa2a6f1.../10.240.9.120
bibin.w@core1 ~/demo $
```

We have now successfully started a service on the cluster. You can list the running services using the following fleetctl command. It shows all the information about the service and in which host it has been launched

fleetctl list-units

```
bibin.w@core3 ~ $ fleetctl list-units
UNIT                DSTATE          TMACHINE                    STATE   MACHINE                     ACTIVE
hello-world.service launched        0aa2a6f1.../10.240.9.120    launched0aa2a6f1.../10.240.9.120    active
bibin.w@core3 ~ $
```

Our hello-world service echo’s out hello world in an infinite loop. To check the

output of the hello world service, run the following fleetctl command on core1.

```
fleetctl journal hello-world.service
```

Scaling fleet units

You can scale your fleet units for high availability service. To do that, create multiple unit files of the same service with X-fleet header. In x-fleet header we will define the relationships between the units.

Let's say we want three Nginx services running in different hosts. For this we will mention a parameter X-Conflicts=nginx*.service in the X-Fleet header, which will make sure that three instances never run on the same host in the cluster.

Let's look at a demo for running nginx containers in high availability mode.

Nginx unit file

We will be running nginx server using Dockerfile/nginx public image from the unit file. In this unit file we will add X-fleet header which was not there in the hello-world service we deployed earlier. X-Fleet ensures that all nginx services will be distributed across the cluster.

Follow the steps given below to launch a highly available Nginx service.

1. Create three unit files named nginx.1.service, nginx.2.service and nginx.3.service and copy the contents of the following snippet on to all three unit files.

```
[Unit]
Description=Hello World
After=Docker.service
Requires=Docker.service
[Service]
EnvironmentFile=/etc/environment
ExecStartPre=/usr/bin/etcdctl set /test/%m
${COREOS_PUBLIC_IPV4}
ExecStart=/usr/bin/Docker run -P --name nginx --rm
Dockerfile/nginx
ExecStop=/usr/bin/etcdctl rm /test/%m
ExecStop=/usr/bin/Docker kill test
```

[X-Fleet]

X-Conflicts=nginx.service*

2. We should have three unit files as shown below

```
core1 demo # ls
nginx.1.service  nginx.2.service  nginx.3.service
core1 demo #
```

3. In order to start all the three services will use wildcard to start service with unit name starting with name nginx. Run the following command to start all the three services.

*fleetctl start nginx**

```
bibin.w@core1 ~/demo $ fleetctl start nginx*
Job nginx.1.service launched on 77ee9831.../10.240.24.9
Job nginx.2.service launched on 0aa2a6f1.../10.240.9.120
Job nginx.3.service launched on d8898912.../10.240.56.247
bibin.w@core1 ~/demo $
```

4. You can view the logs of a service using fleetctl journal command as shown below.

fleetctl journal nginx.1.service

```
core1 demo # fleetctl journal nginx.1.service
-- Logs begin at Thu 2014-10-09 08:43:12 UTC, end at Fri 2014-10-10 11:31:58 UTC. --
Oct 10 08:12:23 core1.c.cloud-repository.internal systemd[1]: Started Hello World.
Oct 10 08:56:06 core1.c.cloud-repository.internal systemd[1]: Stopping Hello World...
Oct 10 08:56:06 core1.c.cloud-repository.internal docker[1287]: Error response from c
Oct 10 08:56:06 core1.c.cloud-repository.internal docker[1287]: 2014/10/10 08:56:06 E
Oct 10 08:56:06 core1.c.cloud-repository.internal systemd[1]: nginx.1.service: contro
Oct 10 08:56:07 core1.c.cloud-repository.internal systemd[1]: Stopped Hello World.
Oct 10 08:56:07 core1.c.cloud-repository.internal systemd[1]: Unit nginx.1.service er
Oct 10 08:56:33 core1.c.cloud-repository.internal systemd[1]: Starting Hello World...
Oct 10 08:56:33 core1.c.cloud-repository.internal etcdctl[1337]: 146.148.45.212
Oct 10 08:56:33 core1.c.cloud-repository.internal systemd[1]: Started Hello World.
```

5. To destroy all the nginx services use the destroy command as shown below.

*fleetctl destroy nginx**

```
core1 demo #~ fleetctl destroy nginx*
Destroyed nginx.1.service
Destroyed nginx.2.service
Destroyed nginx.3.service
core1 demo #
```

Networking, security and API’s

In this chapter, we will learn about Docker advanced networking, security and API’s.

Docker networking

When you create a container, Docker will create virtual interface called Docker0. Docker will look for an IP address from the pool which has not been assigned to any other containers and assigns it to Docker0. The CIDR block assigned by Docker for containers is 172.17.43.1/16.

Docker0 interface:

Docker0 is considered as a virtual Ethernet bridge which is capable of sending and receiving packets to any network interface attached to it. This is how a Docker container is able to interact with the host machine and other containers.

When a container is created, a pair of “peers” interfaces are created by Docker and it is like two sides of a pipe. If you send a packet at one side, it will reach the other side of the interface. So when the “peer” interfaces are created, one will act as the eth0 of the container and other will be exposed to the namespace having a unique name that starts with veth. This veth interface connects to Docker0 Bridge, thus forming a virtual subnet for containers to talk to each other.

Networking options:

There are many options available for configuring networking in Docker. Most of the commands will work only when the Docker server starts and will not work when Docker service is in running state.

Following are the options used to modify the networking settings in Docker.

1. -b BRIDGE: This option is used for specifying the Docker bridge explicitly.
2. --bip=CIDR: This option changes the default CIDR assigned to Docker
3. --fixed-cidr=CIDR: used for restricting IP addresses from Docker0 subnet.

4. -H SOCKET: using this option you can specify from which channel the Docker daemon should receive commands. For example, a host IP address or through tcp socket.
5. --ip=IP_ADDRESS: Used for setting the Docker bind address.

DNS configuration

There are four options to modify the DNS configuration of a container.

1. -h HOSTNAME: This option is used for setting the hostname. The hostname will be written to /etc/hostname file of the container. It is not possible to view the hostname outside the container.
2. --link=CONTAINER_NAME:ALIAS: This option allows us to create another name for the container which can be used for linking. This name will point to the IP address of the container.
3. --dns=IP_ADDRESS : This option will create a server entry inside /etc.resolv.conf file.
4. --dns-search=DOMAIN : This option is used for setting the DNS search domain

Container communication with wider world

There is one factor which determines whether the container should talk to the outside world. It is the ip_forward parameter. This parameter should be set to 1 for forwarding packets between containers. By default, Docker sets the --ip-forward parameter to true and Docker will set ip_forward parameter to 1.

Execute the following command on the Docker host to check the value of the ip_forward parameter.

cat /proc/sys/net/ipv4/ip_forward

```
root@node2:~# cat /proc/sys/net/ipv4/ip_forward
1
root@node2:~#
```

From the output you can see that the ip_forward value is set to 1.

Communication between containers

All the containers are attached to Docker0 bridge by default and this allows all the containers to send and receive packets among them.

This is achieved by Docker by adding a ACCEPT policy to iptables FORWARD

chain. If you set the value -- *icc=false* when the Docker daemon starts, Docker will add DROP policy to the FORWARD chain.

If you set --*icc=false* Docker containers won't be able to talk to each other. You can overcome this issue by using --*link=contianer-name:alias* for linking containers together.

Execute the following command in your Docker host to check the iptables FORWARD policy.

```
sudo iptables -L –n
```

```
root@node2:~# sudo iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination

Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
ACCEPT     tcp  --  0.0.0.0/0            172.17.0.3           tcp dpt:80
ACCEPT     tcp  --  0.0.0.0/0            172.17.0.190         tcp dpt:80
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0            ctstate RELATED,ESTABLISHED
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0
ACCEPT     all  --  0.0.0.0/0            0.0.0.0/0

Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
root@node2:~#
```

The output above shows the default ACCEPT policy.

Building your own bridge

By default, Docker uses Docker0 bridge. You can use a custom bridge for Docker using

-b BRIDGE parameter.

Let's see how to assign a custom bridge to Docker.

If your Docker host is in running mode, execute the following commands to stop the Docker server and to bring down the Docker0 bridge.

Note: if bridge utils is not installed on your Docker host, install it using the *"sudo apt apt-get install bridge-utils"* command before executing the following commands.

```
sudo service docker.io stop
sudo ip link set dev docker0 down
sudo brctl delbr docker0
```

```
root@node2:~# sudo service docker.io stop
docker.io stop/waiting
root@node2:~# sudo ip link set dev docker0 down
root@node2:~# sudo brctl delbr docker0
root@node2:~#
```

Execute the following commands for creating a new bridge and CIDR block associations.

sudo brctl addbr bridge0

sudo ip addr add 192.168.5.1/24 dev bridge0

sudo ip link set dev bridge0 up

```
root@node2:~# sudo brctl addbr bridge0
root@node2:~# sudo ip addr add 192.168.5.1/24 dev bridge0
root@node2:~# sudo ip link set dev bridge0 up
root@node2:~#
```

Execute the following command to check if our new bridge is configured and running.

ip addr show bridge0

```
root@node2:~# ip addr show bridge0
490: bridge0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UNKNOWN group
default
    link/ether 6e:21:f4:3c:50:b3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global bridge0
       valid_lft forever preferred_lft forever
    inet6 fe80::6c21:f4ff:fe3c:50b3/64 scope link
       valid_lft forever preferred_lft forever
root@node2:~#
```

From the output, you can see that our new bridge is configured and running.

Now let's add the new bridge configuration to Docker defaults using the following command and start the Docker service.

echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker

sudo service docker.io start

```
root@node2:~# echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
root@node2:~# sudo service docker.io start
docker.io start/running, process 2102
root@node2:~#
```

Docker security

In this section we will discuss the major areas of security that Docker focuses on and why they are important. Since Docker uses Linux Containers, we will discuss security in the context of linux containers also.

In previous chapters of this book, we learned that a Docker run command is executed to launch and start a container. However, here's what really happens:

1. A Docker run command gets initiated.
2. Docker uses lxc-start to execute the run command.
3. Lxc-start creates a set of namespaces and control groups for the container.

Let's recap what namespace means. Namespace is the first level of isolation whereas no two containers can see or modify the processes running inside. Each container is assigned a separate network stack, and, hence, one container does not get access to the sockets of another container.

To allow IP traffic between the containers, you must specify public IP ports for the container.

Control Groups, the key component, has the following functionalities:

1. Is responsible for resource accounting and limiting.
2. Provides metrics pertaining to the CPU, memory, I/O and network.
3. Tries to avoid certain DoS attacks.
4. Enables features for multi-tenant platforms.

Docker Daemon's Attack Surface

Docker daemon runs with root privileges, which implies there are some aspects that need extra care. Some of the points for security are listed here:

1. Control of Docker daemon should only be given to authorized users as Docker allows directory sharing with a guest container without limiting access rights.
2. The REST API endpoint now supports UNIX sockets, thereby preventing cross-site-scripting attacks.
3. REST API can be exposed over HTTP using appropriate trusted networks and VPNs.
4. Run Docker exclusively on a server (when done), isolating all other services.
5. Processes, when run as non-privileged users in the containers, maintain a good level of security.

6. Apparmor, SELinux, GRSEC solutions can be used for an extra layer of security.
7. There’s a capability to inherit security features from other containerization systems.

An important aspect in Docker to be considered is the fact that everything is not a name spaced in Linux. If you want to reach the kernel of the host running a VM, you have to break through the VM, then Hypervisor and then kernel. But in containers you are directly talking to the kernel.

Containers as a regular service

Containers should be treated just like running regular services. If you run an apache web server in your system, you will be following some security practices to securely run the service. If you are running the same service in a container, you need to follow the same security measure to secure your application. It is not secure just because of the fact that it is running inside a container. While running applications in containers consider the following things:

1. Drop privileges as soon as possible.
2. All the services should run as a non-root user whenever possible.
3. Treat root inside a container same as the root running outside the container.
4. Do not run random containers in your system from the public registry. It might break your system. Always used official and trusted images. It is recommended to start building your own images for container deployments.

What makes Docker secure

Following are the features which make Docker secure:

1. **Read only mount points:** files such as /sys, / proc/sys, proc/irq etc. are mounted in containers with read only mode
2. **CAPABILITIES:** certain Linux kernel CAPABILITIES in containers are removed to make sure it does not modify anything in the system kernel. For example, CAP_NET_ADMIN CAPABILITY is remove make sure that no modifications to the network setting or IPtables can be done from inside a container.

Security issues of Docker

All the Docker containers are launched with root privileges, and allow the containers to share the filesystem with the host. Following are the things to be considered to make Docker more secure.

1. Mapping root user of the container to a non-root user of the host, to mitigate the issue of container to host privileges.
2. Allowing Docker daemon to run without root privileges.

The recent improvements in Linux namespaces allows Linux containers to be launched without root privileges, but it has not been implemented yet in Docker (as of writing of this book)

Docker Remote API: (version v1.15)

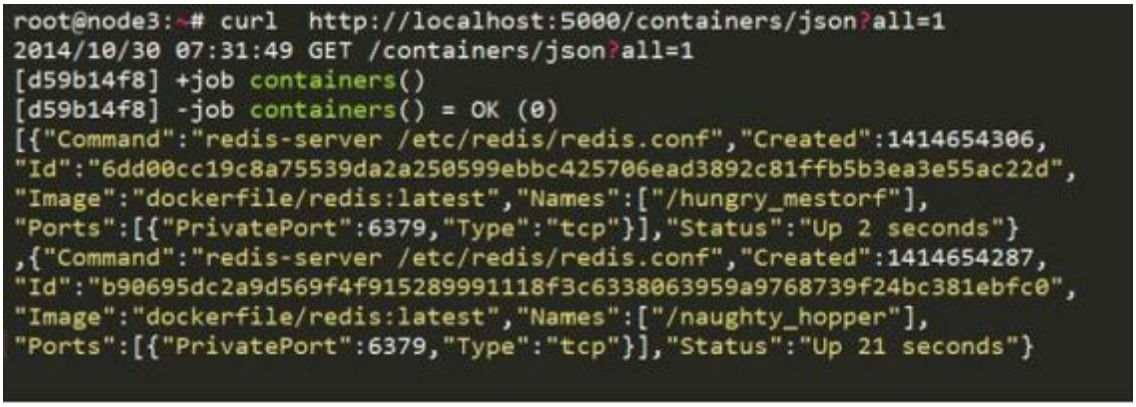
Docker remote API replaces the remote command line interface (rcli). To demonstrate how API works, we will use curl utility to work with GET and POST methods. Let's look at each API operations.

Listing containers

REST syntax: GET /containers/JSON

To list all the containers in a Docker host, run the following command.

curl http://localhost:5000/containers/JSON?all=1



Creating containers

To create a container, run the following command.

*curl -X POST -H "Content-Type: application/JSON" -d *

'{"Hostname":"","Domainname": "","User":"","Memory":0,
"MemorySwap":0,"CpuShares": 512,"Cpuset":
"0,1","AttachStdin":false,
"AttachStdout":true,"AttachStderr":true,"PortSpecs":null,"T
"OpenStdin":false,"StdinOnce":false,"Env":null,"Cmd":
["date"],
"Image":"Dockerfile/redis","Volumes":{"/tmp":
{}},"WorkingDir":"",
"NetworkDisabled": false,"ExposedPorts":{"22/tcp":
*{}},"RestartPolicy": *
{ "Name": "always" }}'
http://localhost:5000/containers/create

```
root@node3:~# curl -X POST -H "Content-Type: application/json" -d '{"Hostname":"",
"Domainname": "","User":"","Memory":0,"MemorySwap":0,"CpuShares": 512,"Cpuset":
"0,1","AttachStdin":false,"AttachStdout":true,"AttachStderr":true,"PortSpecs":null,
Tty":false,"OpenStdin":false,"StdinOnce":false,"Env":null,"Cmd":["date"],"Image":
"dockerfile/redis","Volumes":{"/tmp": {}},"WorkingDir":"","NetworkDisabled": false,
"ExposedPorts":{"22/tcp": {}},"RestartPolicy": { "Name": "always" }}'
http://localhost:5000/containers/create
2014/10/30 09:08:21 POST /containers/create
[d6693bc5] +job create()
[d6693bc5] -job create() = OK (0)
{"Id":"d643811d3707c01d26c381ddd641e93b1f5221c85568ebb3fec2b0a18d116249",
"Warnings":null}
root@node3:~#
```

Inspecting a container

Syntax: GET /containers/<container-id>/JSON

You can inspect a container using the following API request.

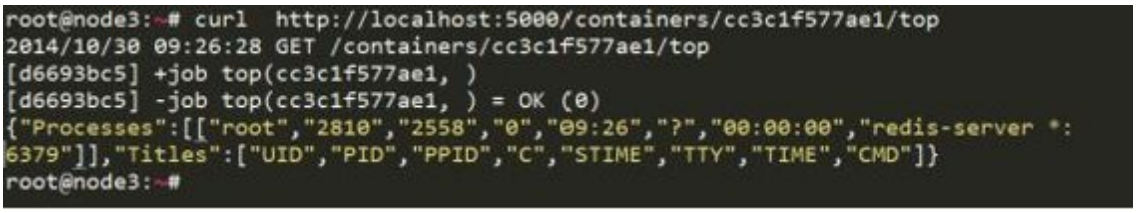
curl http://localhost:5000/containers/d643811d3707/JSON

Listing container processes

Syntax: GET /containers/<container-id>/top

You can list the processes running inside a container using the following API request.

curl *http://localhost:5000/containers/cc3c1f577ae1/top*



Getting container logs

Syntax: GET /containers/<container-id>/logs

Curl *http://localhost:5000/containers/cc3c1f577ae1/logs?stderr=1&stdout=1&timestamps=1&follow=1*

```
root@node3:~# curl  http://localhost:5000/containers/cc3c1f577ae1/logs?stderr=1&stdou
[2] 2864
[3] 2865
[4] 2866
root@node3:~# 2014/10/30 09:30:50 GET /containers/cc3c1f577ae1/logs?stderr=1
[d6693bc5] +job container_inspect(cc3c1f577ae1)
[d6693bc5] -job container_inspect(cc3c1f577ae1) = OK (0)
[d6693bc5] +job logs(cc3c1f577ae1)
[d6693bc5] -job logs(cc3c1f577ae1) = OK (0)
[2]   Done                    curl http://localhost:5000/containers/cc3c1f577ae1/logs
[3]-  Done                    stdout=1
[4]+  Done                    timestamps=1
root@node3:~#
```

Exporting a container

Syntax: POST /containers/<container-id>/export

You can export the contents of a container using the following API request.

curl -o rediscontainer-export.tar.gz
http://localhost:5000/containers/cache/export

```
root@node3:~# curl -o reddiscontainer-export.tar.gz http://localhost:5000/containers/
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     02014/10
[d6693bc5] +job export(cache)
100  390M    0  390M    0     0  30.2M      0 --:--:--  0:00:12 --:--:-- 21.5M[d6693b
100  401M    0  401M    0     0  30.3M      0 --:--:--  0:00:13 --:--:-- 16.6M
root@node3:~# ls
reddiscontainer-export.tar.gz
root@node3:~#
```

Starting a container

Syntax: POST /containers/<container-id>/start

You can start a container using the following API request.

curl -v -X POST
http://localhost:5000/containers/cache/start

```
root@node3:~# curl -v -X POST  http://localhost:5000/containers/cache/start
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
>2014/10/30 09:41:34 POST /containers/cache/start
 POST /containers/cache/start HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
[error] common.go:45 Error parsing media type:  error: mime: no media type
[d6693bc5] +job start(cache)
[d6693bc5] +job allocate_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3
[d6693bc5] -job allocate_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3
[d6693bc5] -job start(cache) = OK (0)
< HTTP/1.1 204 No Content
< Date: Thu, 30 Oct 2014 09:41:34 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
root@node3:~#
```

Stopping a container

Syntax: POST /containers/<container-id>/stop

You can stop a container using the following API request.

curl -v -X POST
http://localhost:5000/containers/cache/stop

```
root@node3:~# curl -v -X POST  http://localhost:5000/containers/cache/stop
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
2014/10/30 09:44:01 POST /containers/cache/stop
[d6693bc5] +job stop(cache)
2014/10/30 09:44:01 Container cc3c1f577ae1489272e3e70505750e27cae2932c474e3a83320638
[d6693bc5] +job release_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3a8332
[d6693bc5] -job release_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3a8332
> POST /containers/cache/stop HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
[d6693bc5] -job stop(cache) = OK (0)
< HTTP/1.1 204 No Content
< Date: Thu, 30 Oct 2014 09:44:01 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
root@node3:~#
```

Restarting a container

Syntax: POST /containers/<container-id>/restart

You can restart a container using the following API request.

curl -v -X POST
http://localhost:5000/containers/cache/restart

```
root@node3:~# curl -v -X POST  http://localhost:5000/containers/cache/restart
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
>2014/10/30 09:46:14 POST /containers/cache/restart
 POST /containers/cache/restart HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
[d6693bc5] +job restart(cache)
[d6693bc5] +job allocate_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3a833
[d6693bc5] -job allocate_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3a833
[d6693bc5] -job restart(cache) = OK (0)
< HTTP/1.1 204 No Content
< Date: Thu, 30 Oct 2014 09:46:14 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
root@node3:~#
```

Killing a container

Syntax: POST /containers/<container-id>/kill

You can kill a container using the following API request.

curl -v -X POST http://localhost:5000/containers/cache/kill

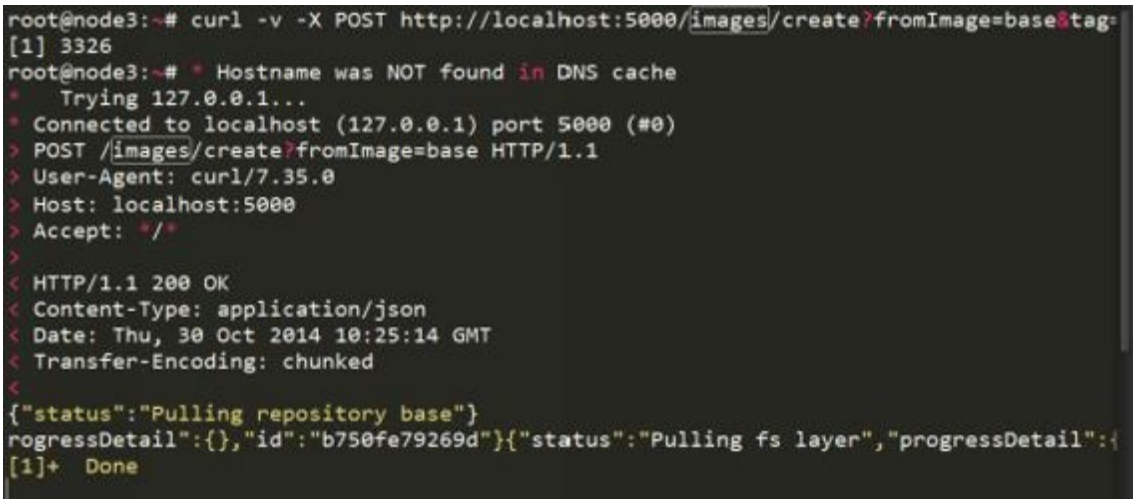
```
root@node3:~# curl -v -X POST  http://localhost:5000/containers/cache/kill
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 5000 (#0)
>2014/10/30 09:48:41 POST /containers/cache/kill
 POST /containers/cache/kill HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:5000
> Accept: */*
>
[d6693bc5] +job kill(cache)
[d6693bc5] +job release_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3a8332
[d6693bc5] -job release_interface(cc3c1f577ae1489272e3e70505750e27cae2932c474e3a8332
[d6693bc5] -job kill(cache) = OK (0)
< HTTP/1.1 204 No Content
< Date: Thu, 30 Oct 2014 09:48:42 GMT
< Content-Length: 0
< Content-Type: text/plain; charset=utf-8
<
* Connection #0 to host localhost left intact
root@node3:~# docker
```

Creating an Image:

Syntax : POST /images/create

You can create an image using the following API request.

curl -v -X POST http://localhost:5000/images/create?fromImage=base&tag=latest



Inspecting an image

Syntax: GET /images/<image-name>/JSON

You can inspect an image using the following API request.

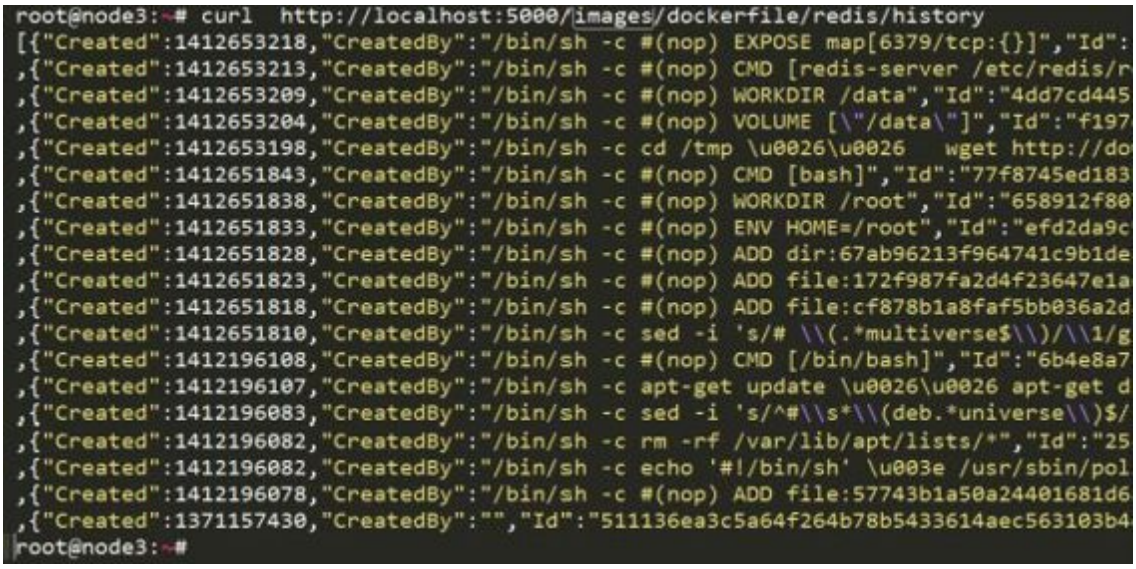
http://localhost:5000/images/Dockerfile/redis/JSON

Getting the history of an Image

Syntax: GET /images/<image-name>/history

You can get the history of an image using the following API request.

curl *http://localhost:5000/images/Dockerfile/redis/history*



Listing all images

Syntax: GET /images/<image-name>/history

You can list all the images in your Docker host using the following API request.

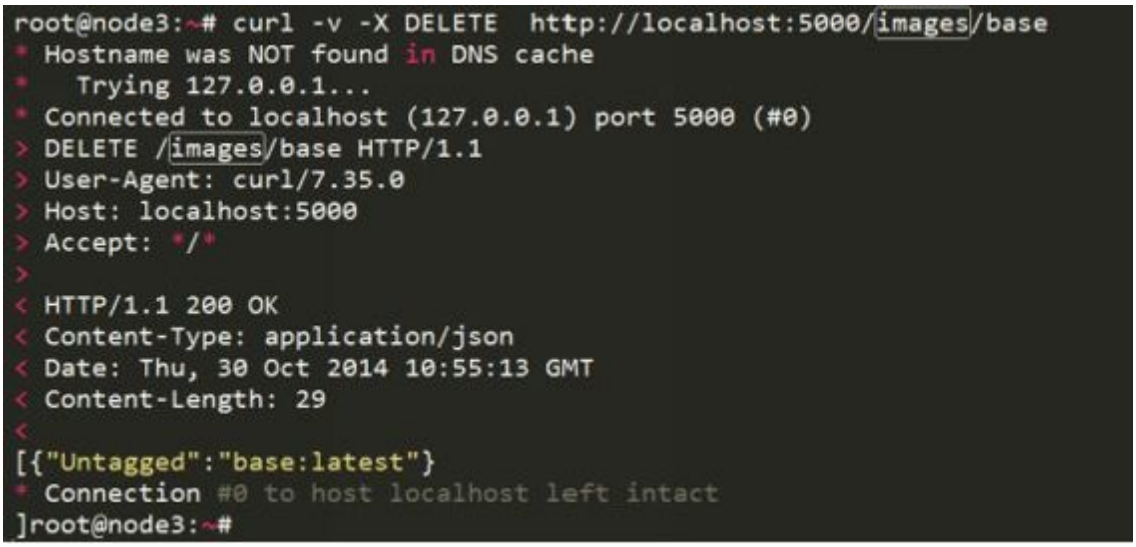
curl *http://localhost:5000/images/JSON*

Deleting an image

Syntax: DELETE /images/<image-name>

You can delete an image using the following API request.

curl -v -X DELETE http://localhost:5000/images/base

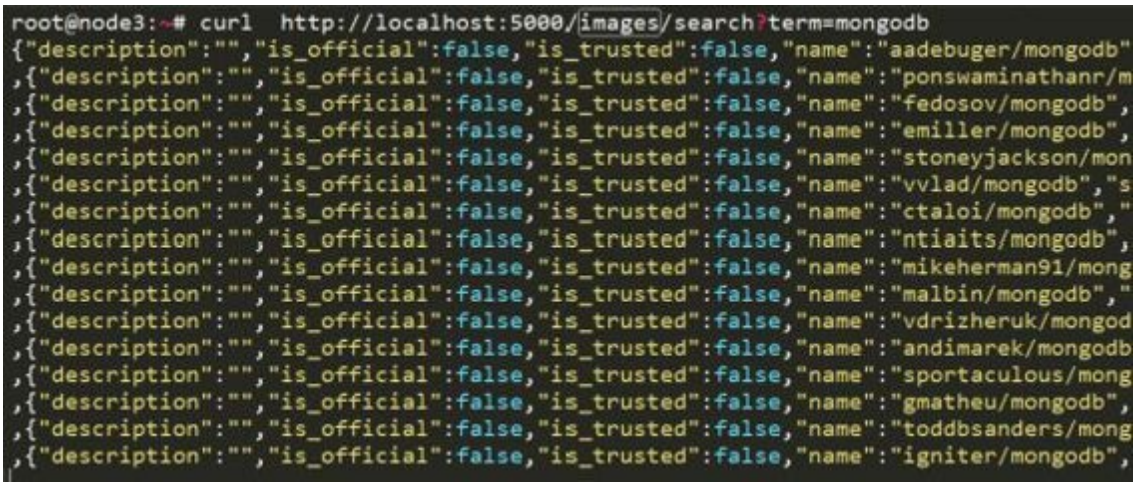


Searching an Image

Syntax : GET /images/search

You can search an image using the following API request.

curl http://localhost:5000/images/search?term=mongodb



12
Cloud container services

In this chapter we will look into the following container service based on cloud,

1. Google container engine
2. Amazon container service (ECS)

Google container engine

Google has been operating on containers for years. The container engine project was inspired by Google's experience in operating container based applications in distributed environments. It is in alpha stage at the time of writing of this book and it is not recommended for production use yet.

At the backend container engine uses kubernetes cluster for scheduling and managing containers. We have learnt to work with kubernetes in the previous chapter. Container engine is a wrapper on top of kubernetes cluster by which you can create kubernetes cluster from a web interface and google cloud CLI's.

In this section we will learn how to create a cluster and deploy a sample wordpress application on to a container engine.

Note: to work with container engine, you should have a compute engine account and workstation authenticated. If you do not have a workstation configured, you can refer the kubernetes section in the previous chapters.

Creating a cluster

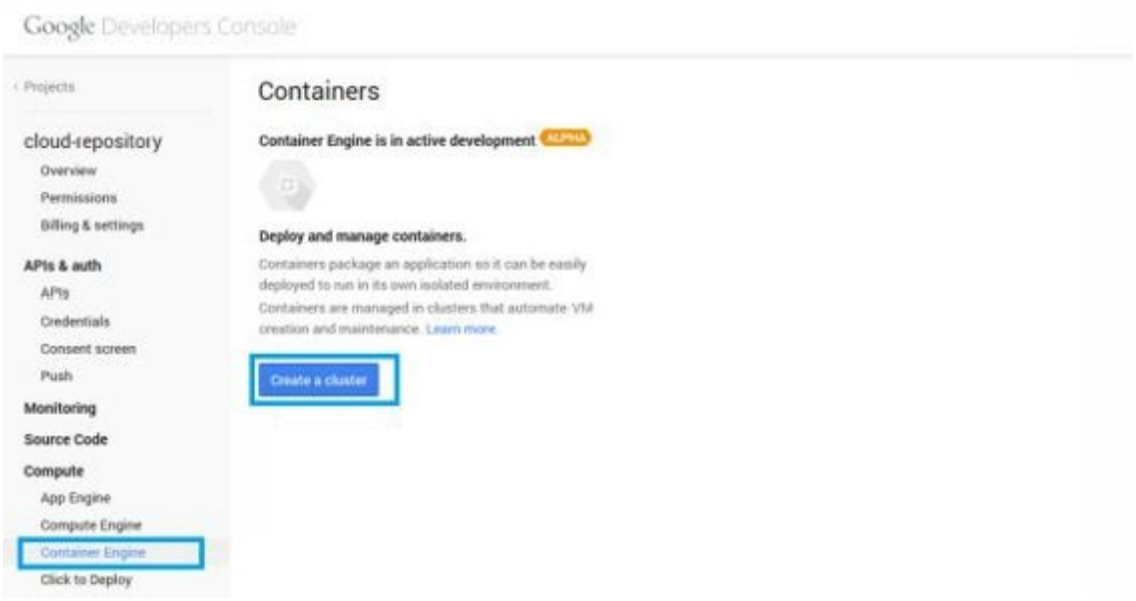
In this section we will learn how to create a container engine cluster from the web interface and using gcloud cli.

Using web interface

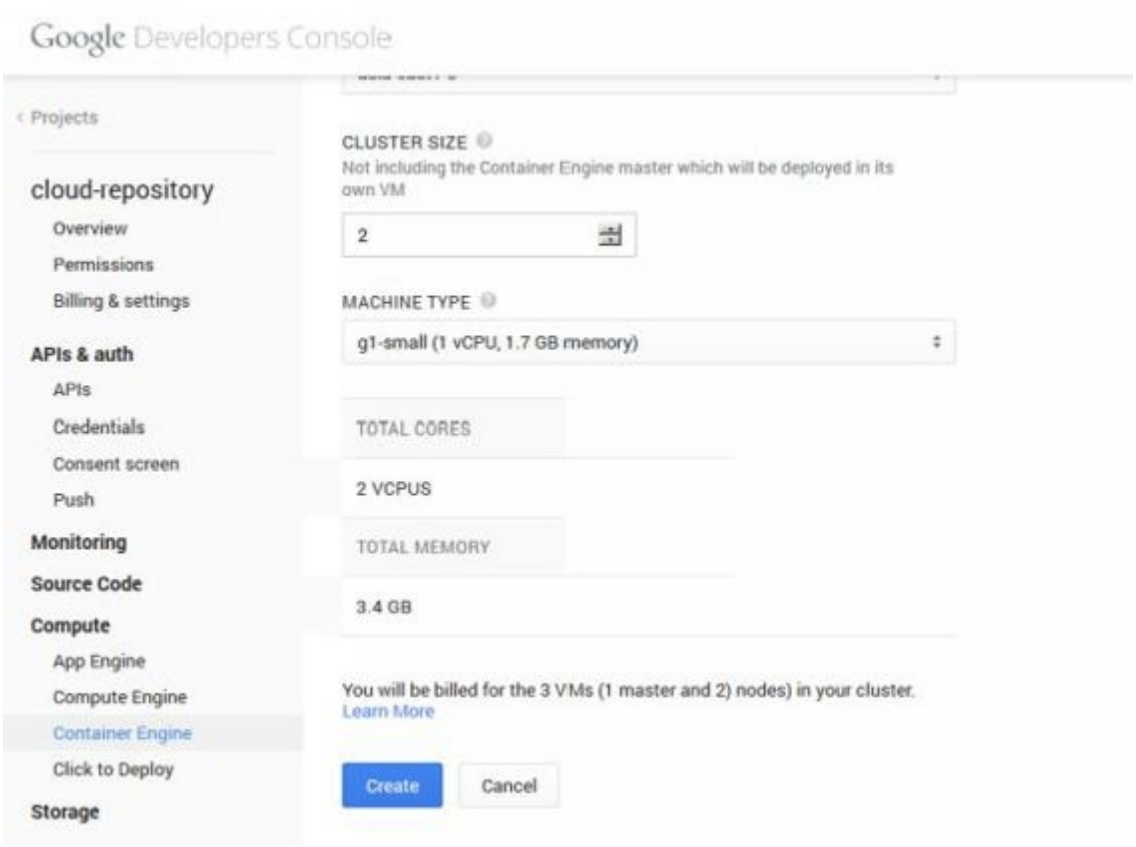
Creating a cluster from web interface is very easy and can be done with few clicks. Follow the steps given below to set up a cluster using the web interface.

1. Login to compute engine and under compute tab you can find the

“container engine” option. Click that and you will see a “create cluster” option. Click on “create cluster” option.



2. In the next window, enter the necessary credentials for cluster configuration. For example, cluster size and machine type. Click the create option once you have entered all the necessary information.



3. A cluster will be created with the given credentials and it takes few minutes for the cluster to set up.

Using gcloud cli

Another way for creating a container engine cluster is through gcloud cli. Follow the steps given below to create a cluster from command line.

1. When you install google cloud sdk, the preview components will not be included in that. So you need to update preview components using

the following command.

sudo gclolud components update preview

```
bibin.w@workstation:~$ sudo gcloud components update preview

The following components will be installed:
    ---------------------------------------------------------------------------
    | Developer Preview gcloud Commands                      | 2014.11.06 | < 1 MB |
    | Native extensions for preview commands (Linux, x86_64) |        4.1 | 2.4 MB |
    ---------------------------------------------------------------------------
Do you want to continue (Y/n)?  y
Creating update staging area...
Installing: Developer Preview gcloud Commands ... Done
Installing: Native extensions for preview commands (Linux, x86_64) ... Done
Creating backup and activating new installation...
Done!
bibin.w@workstation:~$
```

2. We already have a cluster created from the web interface. Let's try getting the information about that cluster from the cli using the following command.

Note: replace "Docker" with your created cluster name and "asia-east1-a" with the zone name where you created the cluster. This information can be obtained from the web interface.

sudo gclolud preview container clusters describe docker --zone asia-east1-a

```
bibin.w@workstation:~$ gcloud preview container clusters describe docker --zone asia-east1-a
clusterApiVersion: 0.4.2
containerIpv4Cidr: 10.187.0.0/16
creationTimestamp: '2014-11-17T06:17:32+00:00'
description: docker cluster
endpoint: 104.155.206.250
masterAuth:
  password: hZa8QID0PIjuceYr
  user: admin
name: docker
nodeConfig:
  machineType: g1-small
  sourceImage: https://www.googleapis.com/compute/v1/projects/debian-cloud/global/images/bac
nodeRoutingPrefixSize: 24
numNodes: 1
servicesIpv4Cidr: 10.0.0.0/16
status: running
zone: asia-east1-a
bibin.w@workstation:~$
```

3. Set two environment variables $CLUSTER_NAME and $ZONE for creating a cluster from cli using the following command.

*gcloud preview container clusters create $CLUSTER_NAME --num-nodes 1 *

--machine-type g1-small --zone $ZONE

```
bibin.w@workstation:~$ gcloud preview container clusters create $CLUSTER_NAME --  \
   num-nodes 1 --machine-type g1-small --zone $ZONE
Waiting for cluster creation...done.
bibin.w@workstation:
```

Install kubecfg client

The workstation need to have kubecgf client installed on it to deploy kubernetes pods on the cluster.

Download the kubecfg linux client using the following command.

wget http://storage.googleapis.com/k8s/linux/kubecfg

```
root@workstation:~# wget http://storage.googleapis.com/k8s/linux/kubecfg
--2014-11-17 08:57:58--  http://storage.googleapis.com/k8s/linux/kubecfg
Resolving storage.googleapis.com (storage.googleapis.com)... 74.125.31.132, 2404:6800:4008:c
Connecting to storage.googleapis.com (storage.googleapis.com)|74.125.31.132|:80... connected
HTTP request sent, awaiting response... 200 OK
Length: 9428664 (9.0M) [binary/octet-stream]
Saving to: `kubecfg'

100%[======================================>] 9,428,664   19.4M/s   in 0.5s

2014-11-17 08:57:59 (19.4 MB/s) - `kubecfg' saved [9428664/9428664]

root@workstation:
```

Change the permissions of the kubecfg folder and move it to /usr/local/bin folder using the following commands.

chmod -x kubecfg

mv kubecfg /usr/local/bin

```
root@workstation:~# chmod +x kubecfg
root@workstation:~# mv kubecfg /usr/local/bin/
root@workstation:~#
```

Now we have every configuration set to deploy containers from the cli. Execute the following gcloud command to deploy a wordpress container from tutum/wordpress image.

*gcloud preview container pods --cluster-name $CLUSTER_NAME create wordpress *

--image=tutum/wordpress --port=80 --zone $ZONE

```
root@workstation:~# gcloud preview container pods --cluster-name $CLUSTER_NAME create wordp
ID                  Image(s)            Host                Labels              Status
----------          ----------          ----------          ----------          ----------
wordpress           tutum/wordpress     <unassigned>        name=wordpress      Waiting
```

To get the information about the container we just created, execute the following gcloud describe command.

sudo gclolud preview container pods --cluster-name \ $CLUSTER_NAME describe wordpress

```
root@workstation:~# gcloud preview container pods --cluster-name $CLUSTER_NAME describe wor
ID                  Image(s)            Host
----------          ----------          ----------
wordpress           tutum/wordpress     k8s-docker-node-1.c.cloud-repository.internal
/104.155.207.187    name=wordpress      Running
root@workstation:~#
```

From the output, you can view the container IP, status and other information. The status shows it is running.

Let's try accessing the wordpress application from the browser.

Note: Open port 80 in the network settings of the instance from the cluster to view the application in the browser.

Now we have a running WordPress application on container engine.

Amazon container service (ECS)

Amazon web services has a container service called ec2 container service. At the time of writing this book, it is in preview mode. In this section we will discuss the ec2 container service and its features.

EC2 container service is a highly scalable container management service. ECS supports Docker and it can manage containers to any scale. This service makes it easy for deploying containers in a cluster to run distributed applications. ECS has the following features.

1. Manages your containers (metadata, instance type etc.,)
2. Schedules containers on to your cluster.

3. Scaling containers from few to hundreds
4. High performance, the cluster runs inside VPC.
5. Manages cluster state. Using simple API calls , you can start and terminate containers
6. You can get the state of the container by querying the cluster from a centralized service.
7. Along with containers, you can leverage the other AWS features such as security groups, ECS volumes, policies etc. for enhanced security.
8. You can distribute the containers in a cluster among availability zones
9. Eliminates the need for third party container cluster management tools.
10. ECS is a free service you will have to pay only for the backend ec2 resources you use.

Docker compatibility

Docker platform is supported by ec2 container service. You can run and manage your Docker container in ECS cluster. All the ec2 machines in the ECS cluster come bundled with Docker daemon. So there is no additional need to configure the server to setup the Docker daemon. You can seamlessly deploy containers from the development environment to the ECS cluster.

Managed Clusters

A challenging part in Docker container deployments is the cluster management and monitoring. ECS make your life so easy by handling all the complex cluster configurations. You can focus on container deployments and its tasks, leaving all the complex cluster configurations to ECS.

Programmatic Control

You can integrate ECS with any application using its rich API features. Cluster management and container operations can be managed programmatically using ECS API.

Scheduler

ECS has an efficient scheduler which schedules containers on to the cluster. The

scheduler decides in which host the container should be deployed. Another interesting ECS feature is that you can develop your own scheduler or you can use some other third party scheduler.

Docker Repository

You can use Docker public registry, your own private registry and third party registries for image management and container deployments. The registry you want to use with ECS should be specified in the task file used for container deployments.

Now let's look at the core components of ECS.

Ec2 container service has four main components,

1. Tasks
2. Containers
3. Clusters
4. Container instances

Tasks

Task is a declarative JSON template for scheduling the containers. A task is a grouping of related containers. A task could range from one to many containers with links, shared volumes etc. An example task definition is given below.

```
{
"family" : "Docker-website",
"version" : "1.0"
"contianers" : [
<container definitions>
]}
```

Container definition

A container definition has the following

1. Container name
2. Images
3. Runtime attributes (ports , env variables ,etc)

An example container definition is given below

```
{
“name” : “dbserver” ,
  “image” : “Ubuntu:latest”,
  “portmappings” : [ { “contianerport” “ 3306 , “hostport”
: 3308
}
```

Clusters

Clusters provide a pool of resources for your tasks. It groups all the container instances.

Container instance

A container instance is an EC2 instance on which the ECS agent is installed or an AMI with ECS agent installed. Each container instance will register itself to the cluster during launch.