

THE DOCKER ECHO SYTEM

By: Walid Ashraf

Table of Contents

Introduction	0
Introduction to docker	1
What are Containers	1.1
History Of Containers	1.1.1
Containers in Reallife	1.1.2
Containers implementations	1.1.3
What is docker	1.2
Docker Slogans	1.2.1
Docker Vocab	1.2.2
Docker is not	1.2.3
Techniques Behind Docker	1.3
Control Groups	1.3.1
Kernel NameSpaces	1.3.2
Union File System	1.3.3
Capabilities	1.3.4
Copy On Write	1.3.5
Docker vs Virtual Machines	1.4
Docker Architecture	1.5
Docker Client	1.5.1
Docker Engine	1.5.2
Docker HUB	1.5.3
Docker Interfaces	1.6
The EcoSystem	1.7
Docker First Steps	2
Installation	2.1
Install On CentOS	2.1.1
Installation From source	2.1.2
Installation On Ubuntu	2.1.3
Docker Engine Control	2.2
Running Docker Container	2.3

Run Command	2.3.1
Example Running Containers	2.3.2
Container Control Commands	2.4
Docker Create	2.4.1
Docker Start, Stop and Restart	2.4.2
Docker Execute	2.4.3
Docker Copy	2.4.4
Docker Attach	2.4.5
Docker Inspect	2.4.6
Docker delete	2.4.7
Docker Generic Commands	2.5
Docker Login	2.5.1
Docker Info	2.5.2
Docker Diff	2.5.3
Docker PS	2.5.4
Tips and Tricks	2.6
Images and docker HUB	3
Docker Images and Containers	3.1
Docker Images Commands	3.2
Building Docker Images	3.3
Docker Commit	3.3.1
Docker Build	3.3.2
Docker File Instructions	3.4
Using Docker HUB	3.5
Push Image to Docker HUB	3.5.1
Docker in Depth	4
Docker Links	4.1
Docker Networking	4.2
Port Mapping	4.2.1
Advanced Network Configuration	4.2.2
Docker Weave	4.2.3
Data Volumes	4.3
Resource Control	4.4
Memory Control	4.4.1

CPU Control	4.4.2
IO Control	4.4.3
Control Capabilities	4.4.4
Accounting for usage	4.4.5
Docker Behind the Scenes	4.5
Process Name Spaces	4.5.1
File System	4.5.2
Docker Deamon Configuration	4.6
Docker Security	4.7
Docker API	4.8
Docker Demos	5
Node JS	5.1
Wordpress and mysql	5.2
RubyonRails and mysql	5.3
Dot Net	5.4

Introduction to the book

What this book is?

The Book is an initiative to create the ultimate docker book for all the docker echo systems and usages e.g. (Micro Service, PaaS, etc.)

The book is open source technical handbook for all the docker tools and applications. Being an open source book that allows all the docker enthusiastic developers, system administrators and business users to view, review and modify its contents.

In terms of tools the book will discuss the integration with orchestration and clustering tools (e.g., Swarm, Kubernetes, Mesos).

Also, the book discusses the integration with automation tools like puppet, chef and JUJU.

The Book is a journey from zero or very basic understanding of docker to a fully mature understanding of the containers and the integration with various tools.

About the authors

Feel free to edit and review the book and Feel free to add your self to the book authors as long as you continuously contribute to this body of knowledge.

Introduction to docker

In this chapter we are going to introduce docker concepts, tools, ecosystem and what it's used for.

We start by defining what are containers and their history and stating their applications in real life and what are the famous implementations of containers.

The next sections contains a quick view of the docker vocab,architecture, techniques behind it and a comparison between container and Virtual Machines.

The chapter ends with the interface to the docker engine and an introduction about the echo system.

What are Containers ?

Containers is a light weight virtualization which is also known as Operating system level virtualization. As a definition its a middle-ware and echo system that allows sharing the Operating system between multiple instances that are called containers.

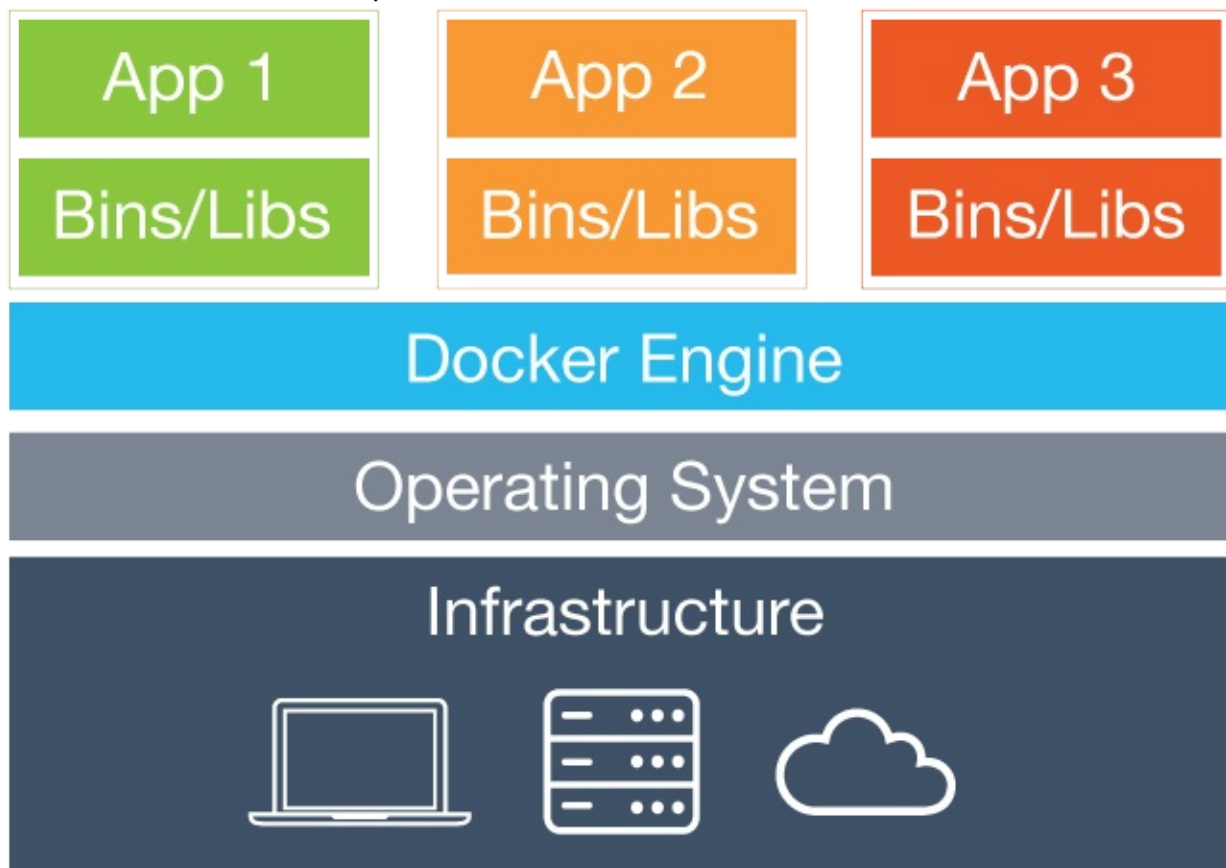
Each container is separated from other containers and could have a different Linux flavor than other container and the base operating system.

Although docker is an implementation of the idea but it have been greatly adapted through all the community the open source and the great cloud providers e.g. Amazon, Microsoft and Google.

Moreover, at DockerCon 2014, Google's Eric Brewer announced that Google would be supporting Docker as its primary internal container format.

Also, Microsoft announced that windows server 2016 will support docker containers in the docker standard image format.

There are various contributions to the container idea by the open source community and even the multinational companies.



History Of Containers

One of most basic laws of science is the Law of the Conservation of Energy “Energy cannot be created or destroyed; it can only be changed from one form to another”.

Also, Darwin states that “It is not the strongest of the species that survives, nor the most intelligent that survives. It is the one that is most adaptable to change”.

From those two golden rules we can figure three things about any technology, tool or concept:

1. Technologies evolve the techniques for software development in the 90's are far different from the ones used now. Tools that doesn't adapt to new techniques, requirements will soon be replace by another by a more change adaptable ones even if it's very smart, powerful or cheap.
2. Ideas doesn't come out from the vacume most concepts are incrementally created.
3. A technology like containers or a tool like docker didn't invent the wheel; as, Controlling users, resource and tasks is the ultimate role for any Operating system.

The following shows the historical time-line of the changes that lead to what we have now.

- **1979 :**

During development of Unix Version 7, A chroot on Unix operating systems is an operation that changes the apparent root directory for the current running process and its children. For us now its very standard for a root user to have its own root directories as well as each user.

- **2000 :**

FreeBSD 4.0 was released with a new command, called jail, which was designed to allow shared-environment hosting providers to easily and securely create a separation between their processes and those of their individual customers. Also, allowed isolated resource sharing between processes. FreeBSD jail expanded chroot's capabilities, but restricted everything a process could do with the underlying system and processes in other jails. And It had problems with shared devices and resources.

- **2004 :**

Sun released an early build of Solaris 10, which included Solaris Containers, and later evolved into Solaris Zones. This was the first major commercial implementation of container technology and is still used today to support many commercial container implementations.

- **2005 :** Open VZ set a complete vision but it was hyper visor like implementation with no speed improvements it was based on chroot and rlimit but it had a complete visions. The main problem with openVZ with that they changed the linux kernel it self which by time drifted more and more from the standard releases.

- **2006 :**

Engineers at Google (primarily Paul Menage and Rohit Seth) started the work on this feature under the name "process containers". That was later named to control groups. And it's now a part from the Linux kernel 2.6.24 release January 2008. This feature is the most important linux feature that motivated the existance of containers and its explained in depth a little further in the road.

- **2007 :**

HP released Secure Resource Partitions for HP-UX, later renamed to HP-UX Containers;

- **2008 :** LXC (Linux Containers) as the first Cgroups based frame work to support containers. LXC provided a middle ware component that allowed the idea of containers to exist.

- **2013 :**

Docker was realeased and doin't expect to explan docker in a couple of words but docker is a framework to create, monitor and maintain containers that was built on LXC but shifted recently to lib container.

- **2013 :**

Imctfy (Let Me contain that for you) was release by Google to compete with Docker. [The Project Git Repository](#)

- **2014 :**

Google Announces that it has used containers as a concept since 2006 and released its open source project [kubernetes](#) that allows container clustering.

- **2015 :**

[OPEN CONTAINER INITIATIVE](#) annoced that the Docker lib containers is standardized as the base for a Linux kernel feature.

Containers in Reallife

The containers are a concept that applied to many of the products we use daily. For example, Android uses containers to separate Applications (Dalvik VM). Also Google announced in Google I/O 2014, that use containers to host user Apps and all their cloud services and they announced that they create 2 billion containers/week. Also containers as an ecosystem have raised in the latest few years with some of the applications that includes:

Highly performant, hyper scale deployments of hosts

Containers are known for both their high density/host and rapid start time which are a core feature to hyper scale deployment.

Micro-Services architecture

Micro-services is a concept that adds to the SOA Architecture in a way to divide a service into Sub services to increase isolation and allow better control to the deployment ways of the services.

Building a multi-user Platform-as-a-Service (PAAS) infrastructure

Sharing an OS layer is one of the best ways to implement a shared platform

Sharing Environments between Developers

Containers could encapsulate both an infrastructure and a code which allows synchronous development tools and stack.

Testing Environments

Bootstrapping a container with a test environment is a matter of seconds which eases the process of testing new tools and code on the required environment.

Research Validation

Shipping a research with a container containing all the needed code and tools to prove a research is an ultimate utility for research verification.

Containers implementations

Although this document is focused on **docker** we must clarify that there are different containers implementations.

LXC (Linux Containers)

This is the father of all kinds of containers and it represents an operating-system-level virtualization environment for running multiple isolated Linux systems (containers) on a single Linux machine. You can get more information [Click Here](#)

OpenVZ

This is an OS-level virtualization technology based on the Linux kernel and the operating system. OpenVZ allows a physical server to run multiple isolated operating system instances, called containers, virtual private servers (VPSs), or virtual environments (VEs).

Lmctfy

This was by Google to compete with docker but it was not a success.

The FreeBSD jail

This is a mechanism that implements an OS-level virtualization, which lets the administrators partition a FreeBSD-based computer system into several independent mini-systems called jails. The AIX Workload partitions (WPARs) These are the software implementations of the OS-level virtualization technology, which provide application environment isolation and resource control.

Solaris Containers (including Solaris Zones)

This is an implementation of the OS-level virtualization technology for the x86 and SPARC systems. A Solaris Container is a combination of the system resource controls and boundary separation provided by zones. Zones act as completely isolated virtual servers within a single operating system instance.

What is docker

Although Docker has a huge echo system around its ideas. It container management tool that provides virtualization, management and orchestration. But there are five differences that makes docker a game changer:

- Docker is a company that was named **dotcloud** that used to provide Cloud services and shifted its name and interest to container [Company Site](#)
- Docker adds a powerful stack of ideas that completed the vision of containers.
- Docker is an opensource project [Docker Repository](#)
- Docker is supported by almot all the big players in the cloud computing market e.g. (Amazon, Azure, RackSpace , etc.)
- Docker is an award winning tool.

For More Information Click Visit [Under Standing Docker](#) Page

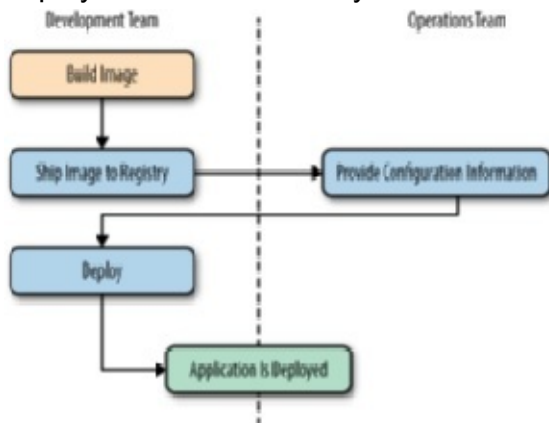
Docker Slogans

Every company aside having a mission and a vision statement might have some slogans that are required to deliver a message about their product. Docker has two main slogans:

1. Build, Ship and Deploy

Docker allows a software to be packaged in a loosely coupled way that enables a software component(s) to be easily deployed without the need for the configuration and installation hustle.

The workflow of the any application ends usually with deployment and here the famous Quote “it worked on my device or it passed all of tests” appears. An image is a deployment component that has all the needed code, libraries, frameworks and filesystems. Abstracting the solution into an image packed with all of its dependences allows the removal of a deployment hell caused by environment conflicts and changes.



2. Batteries included but replaceable

Docker insures a concept well established in the world of Object Oriented Development which allows a component to be replaced with another one by just implementing the main interface used by the top layer.

Docker Vocab

Docker Image and Container

A Docker image is template or a class that has all the necessary file to create a container. The difference between an image and a container is like the difference between the class and an object. A class is a manuscript of what should be there without any physical existence and this is just what an image is an image contains what should be there without any actual access to any resources (e.g. CPU time, memory and storage). On the other hand a container is the object it has a physical existence on the host with an IP, exposed ports and real Processes.

At the end of a day docker is just a virtualization and process isolation solution. So if you made the image host all application, the domain and data access layers you're just back at square one. Docker implements and provides tools for orchestration that will make your life easier so please an image should be used as single component in your system.

Docker Engine

The docker command run in daemon mode. This turns a Linux system into a Docker server that can have containers deployed, launched, and torn down via a remote client.

Docker Client

The docker command used to control most of the Docker workflow and talk to remote Docker servers.

Docker Hub (Registry)

The Storage component of docker that allows you to save, use and search for images.

Docker is not

- **Docker is not an Automation or configuration management tool (Puppet, Chef, and JUJU)**
- **Docker is not a Hardware Virtualization Solution (VMware, KVM, and XEN)**
- **Docker is not a Cloud Platform (Open Stack, and Cloud Stack)**
- **Docker is not a Deployment Framework (Capistrano, Fabric, etc.)**
- **Docker is not a Workload Management Tool (Mesos, Fleet, etc.)**
- **Docker is not Development Environment (Vagrant, etc.)**

Techniques Behind Docker

The Techniques behind docker includes:

1. Control Groups
2. Kernel Name Spaces
3. Union File system
4. Capabilities
5. Copy on Write

Control Groups

It's a method to put processes into groups by allowing the **Following**:

- Resource limitation: groups can be set to not exceed a configured memory limit, which also includes the file system cache.
- Prioritization: some groups may get a larger share of CPU utilization or disk I/O throughput.
- Accounting: measures how much resources certain systems use, which may be used, for example, for billing purposes.
- Control: freezing the groups of processes, their checkpointing and restarting

Each cgroup is represented by a directory in the cgroup file system containing the following files describing that cgroup:

- Tasks: list of tasks (by PID) attached to that cgroup.
- Cgroup.procs: list of thread group IDs in the cgroup.
- Notify_on_release flag: run the release agent on exit?
- Release_agent: the path to use for release notifications (this file exists in the top cgroup only).

Kernel Namespaces

A kernel namespace wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

Changes to the global resource are visible to other processes that are members of the namespace, but are invisible to other processes.

There are 6 Name Spaces:

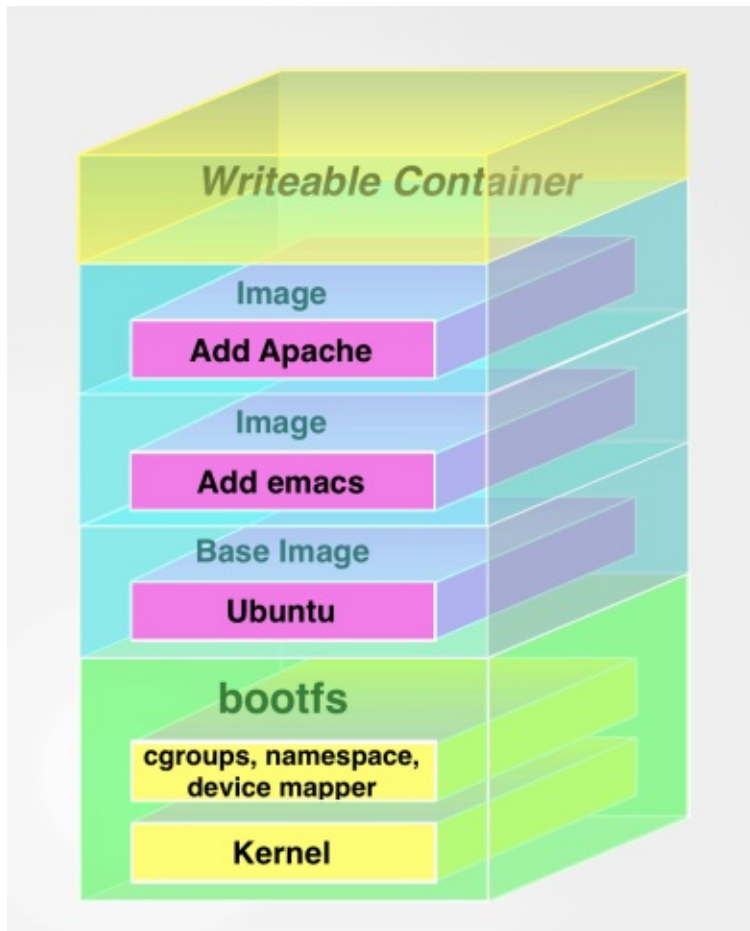
- **PID** namespace provides isolation for the allocation of process identifiers (PIDs), lists of processes and their details. While the new namespace is isolated from other siblings, processes in its "parent" namespace still see all processes in child namespaces—albeit with different PID numbers.
- **Network** namespace isolates the network interface controllers (physical or virtual), iptables firewall rules, routing tables etc. Network namespaces can be connected with each other using the "veth" virtual Ethernet device.
- **"UTS"** namespace allows changing the hostname.
- **Mount** namespace allows creating a different file system layout, or making certain mount points read-only.
- **IPC** namespace isolates the System V inter-process communication between namespaces.
- **User** namespace isolates the user IDs between namespaces.

Namespaces are created with the "unshare" command or syscall, or as new flags in a "clone" syscall.

For More about name spaces [click here](#)

Union File System

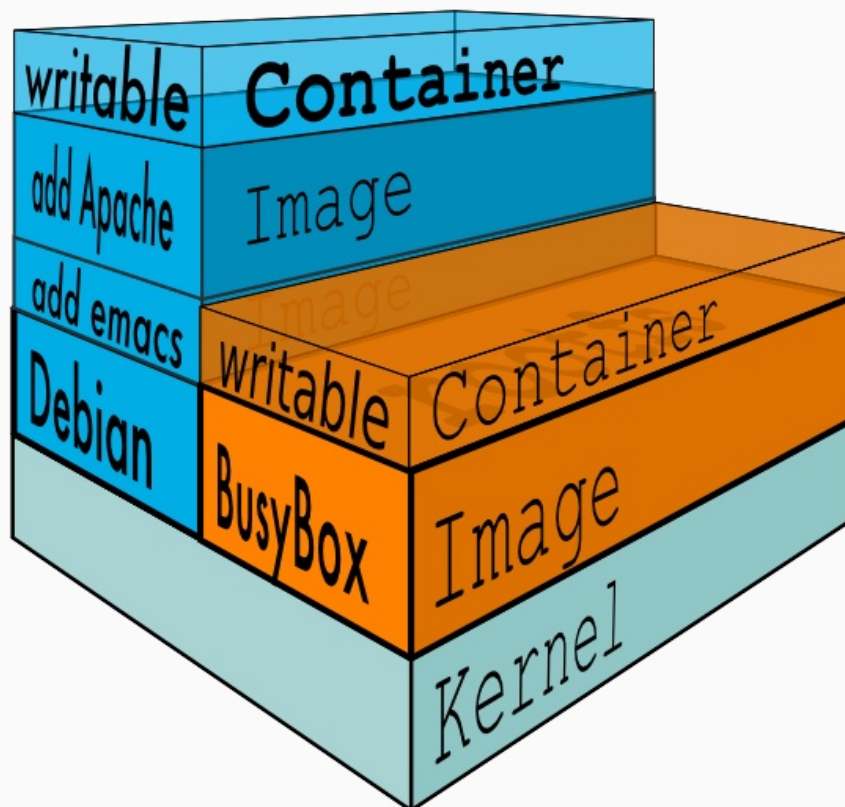
Union file system represents file system by grouping directories and files in branches. A Docker image is made up of filesystems layered over each other (i.e. Branches that is stacked on top of each other) and grouped together. At the base is a boot filesystem, bootfs, which resembles the typical Linux/Unix boot filesystem. A Docker user will probably never interact with the boot filesystem.



Indeed, when a container has booted, it is moved into memory, and the boot filesystem is unmounted to free up the RAM used by the initrd disk image. So far this looks pretty much like a typical Linux virtualization stack. Indeed, Docker next layers a root filesystem, rootfs, on top of the boot filesystem. This rootfs can be one or more operating systems (e.g., a Debian or Ubuntu filesystem).

Docker calls each of these filesystems images. Images can be layered on top of one another. The image below is called the parent image and you can traverse each layer until you reach the bottom of the image stack where the final image is called the base image. Finally, when a container is launched from an image, Docker mounts a read-write filesystem on top of any layers below. This is where whatever processes we want our Docker container to run will execute. When Docker first starts a container, the initial read-write layer is empty.

As changes occur, they are applied to this layer; for example, if you want to change a file, then that file will be copied from the read-only layer below into the read-write layer. The read-only version of the file will still exist but is now hidden underneath the copy.



Capabilities

Process Capabilities

Each Linux process has four sets of bitmaps. By Default each bitmap is 32 bit for 32 different capability.

- **Effective (E):**

The effective capability set indicates what capabilities are effective The Process can use now

- **Permitted (P):**

The process can have capabilities set in the permitted set that are not in the effective set. This indicates that the process has temporarily disabled this capability. A process is allowed to set a bit in its effective set only if it is available in the permitted set. The distinction between effective and permitted makes it possible for a process to disable, enable and drop privileges

- **Inheritable (I):**

Indicates what capabilities of the current process should be inherited by the program executed by the current process

- **Bset:**

Determine forbidden capabilities

File Capabilities

Allows set, the forced set, and the effective set

Copy On Write

The Middleware is responsible for creating and managing the container. When a container is started it appears to have its own Linux files system that you're using. The fact is you don't; at the start of the container it links to files in the base kernel that all containers shared, and the way it handle the changes is via the copy on write model, where each change to the file system is copied to an in memory version of the base file with the changes.

This results two main effects:

- The container will result a small footprint.
- changes in memory are faster than writing to storage.

This works in conjunction with UNION File system. Where, each image layer is read-only; this image never changes. When a container is created, Docker builds from the stack of images and then adds the read-write layer on top. That layer, combined with the knowledge of the image layers below it and some configuration data, form the container.

Docker vs Virtual Machines

	Container	Hyper visor
Redundancy	Only Application Code and Data Edited Files	Computation and Memory redundancy
Flexibility	Less Flexible, Only Supported in Linux, Cannot host Windows OS	More Flexible where Linux OS can host windows and vice verse
Security	Less Secure, DOS attacks can affect others Containers Container with root privileges could affect other Containers	Full Isolation, Hypervisor is responsible of limiting used resource by VM Cross Guest Access is prohibited
Creating Time	Almost instantaneous	Even Preexisting VMs need OS Load Time
Performance	Almost Native	Less than native due to middle ware
Consolidation	Limited by actual Application Usage	Limited By OS reserved
Memory	Allocation Memory	Allocated Files Disk Allocated Files

Docker Architecture

Docker Architecture depends on three main components:

- Docker Client
- Docker Host
- Docker HUB

Docker Client

The responsive part that takes with the engine

Docker Engine

The Docker engine is the actual middle-ware that runs, monitors, orchestrate the containers

Docker HUB

Inside Docker Hub (or on a Docker registry you run yourself), images are stored in repositories. You can think of an image repository as being much like a git repository.

It contains images, layers, and metadata about those images. Each repository can contain multiple images (e.g., the Ubuntu repository contains images for Ubuntu 12.04, 12.10, 13.04, 13.10, and 14.04). Each of these images is identified by tags.

For More Information visit the [hub site](#)

Docker Interfaces

Command Line Interface

The most common tools for a docker GUI Interface

Shipyard

Shipyard gives you the ability to manage Docker resources, including containers, images, hosts, and more from a single management interface. For more information click [here](#)

DockerUI

DockerUI is a web interface that allows you to interact with the Docker Remote API. It's written in JavaScript using the AngularJS framework. For more information click [here](#)

maDocker

A Web UI written in NodeJS and Backbone (in early stages of development). For more information click [here](#)

Docker ToolBox

tool [site](#)

The EcoSystem

The docker ecosystem is rich and allows multiple types of orchestration and clustering tools.

The tools that are built for the docker ecosystem and included in the book are :

1. Kubernetes
2. Mesos Marathon
3. CoreOS
4. Flocker

Also the docker integrate with tools like:

1. Puppet
2. Chef
3. Ansible

Docker First Steps

This chapter demonstrates the installation of docker on various environments and bases.

Environments include:

1. CentOS
2. Ubuntu

Also this chapter contains all basic commands and tools to support docker.

Installation Prerequisites

Docker must be installed on 64 bit OS with kernel more than 3.10. The kernel must support an appropriate storage driver.

For example:

- Device Mapper
- AUFS
- vfs
- btrfs

The default storage driver is usually Device Mapper for centos and redhat and AUFS for ubuntu.

Install On CentOS

The used Version in the installation is CentOS 7 Core 64 Bit, this should be applicable to both native and virtualized environment. For More information visit [docker docs for CentOS](#)

Installation with Script

Install latest updates to help you get the latest version of Docker

```
$ sudo yum update
```

Install Docker via Script

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

Run the Docker Engine

```
$ sudo service docker start
```

Test Docker Version

```
$ docker version
```

Incase of error use this command:

```
yum install docker-selinux
```

And To start docker

```
sudo systemctl start docker
```

Then Test Docker Version

Installation from Source Code Repository (Without the Script)

Install latest updates to help you get the latest version of Docker

```
$ sudo yum update
```

Add the Repo to your package manager

```
$ cat >/etc/yum.repos.d/docker.repo <<-EOF [dockerrepo] name=Docker Repository
baseurl=https://yum.dockerproject.org/repo/main/centos/7 enabled=1 gpgcheck=1
gpgkey=https://yum.dockerproject.org/gpg EOF
```

Install docker Package

```
$ sudo yum install docker-engine
```

Run the docker engine

```
$ sudo service docker start
```

Incase of error use this command:

```
yum install docker-selinux
```

And To start docker

```
sudo systemctl start docker
```

Test Docker Version

```
$ docker version
```

Installation From source


<http://was.id.ly/writing/docker-build-from-source>

<https://docs.docker.com/v1.5/contributing/devenvironment/>

Installation On Ubuntu

1 - Add the new gpg key.

```
$ sudo apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-keys 58118E89F
```



2 - add the repo location

```
$ sudo vi /etc/apt/sources.list.d/docker.list
```

3 - based on you version add `deb https://apt.dockerproject.org/repo ubuntu-trusty main`

4 - Update Package manager `$ apt-get update`

5 - install docker

```
sudo apt-get install docker-engine
```

6 - To test the installation

```
sudo service docker start
```

```
sudo docker version
```

Docker Engine Control

Start Docker engine

Stop

Restart

Configure

Chkconfing

After installing the docker you can make sure the service starts with the OS

```
$ sudo chkconfig docker on
```

Running Docker Container

Running containers is the core functionality of docker because it allows the start of containers and acquisition of the resources and controlling it.

For more about the run behavior of docker visit [the official page](#)

Run Command

Creating containers is the sole purpose of all these infrastructure.

Using this command will initiate the following work-flow:

1. It will check for the container locally
2. Fetch the container from the Docker Hub Repository
3. Run the container

Usage

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]
```

Although the Image Builder has a default value for each of the following one can override them.

[Options]

The run options control the image's runtime behavior in a container. These settings affect:

- detached or foreground running
- container identification and name
- network settings and host name
- runtime constraints on CPU and memory
- privileges and LXC configuration

IMAGE[:TAG|@DIGEST]

Using an image based on a specific version based on tag or custom made identifier (Digest)

[COMMAND]

Specifies commands to run at the start of the container

[ARG...]

Other arguments which include adding users and setting environment variable and mounting devices.

For more information visit the [command line page](#)

Example Running Containers

Hello World container

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

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
535020c3e8ad: Pull complete
af340544ed62: Pull complete
Digest: sha256:a68868bfe696c00866942e8f5ca39e3e31b79c1e50feaae4ce5e28df2f051d5c
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
 3. The Docker daemon created a new container from that image which runs the executable t
 4. The Docker daemon streamed that output to the Docker client, which sent it to your te
Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com
```

Ubuntu Container

Run an Ubuntu container in interactive tty mode. `$ sudo docker run -i -t ubuntu /bin/bash`

For more examples and ideas, visit the [user guide](#)

Container Control Commands

The docker have some commands that allows the some control over a container.

1. Create
2. Start
3. Stop
4. Restart
5. Execute
6. Copy
7. Attache
8. Inspect
9. Delete

Docker Create

The docker create command creates container that allows setting configuration you can set in run command the only difference that the created container is never started and in order to start a container run the start command.

For more the docker create check out the [official documentation](#).

Usage

```
$ sudo docker create [OPTIONS] IMAGE [COMMAND] [ARG...]
```

Example

```
$ docker create -t -i fedora bash
```

```
6d8af538ec541dd581ebc2a24153a28329acb5268abe5ef868c1f1a261221752
```

```
$ docker start -a -i 6d8af538ec5
```

```
bash-4.2#
```

Docker Start, Stop and Restart

These three commands are used to change the state of a container from running to stopped or restart it.

Stop: When a user issues this command, the Docker engine sends SIGTERM (-15) to the main process, which is running inside the container.

The **SIGTERM** signal requests the process to terminate itself gracefully. Most of the processes would handle this signal and facilitate a graceful exit. However, if this process fails to do so, then the Docker engine will wait for a grace period. Even after the grace period, if the process has not been terminated, then the Docker engine will forcefully terminate the process. The forceful termination is achieved by sending **SIGKILL** (-9).

The **SIGKILL** signal cannot be caught or ignored, and so it will result in an abrupt termination of the process without a proper clean-up.

Usage

```
$ docker stop | start | restart [OPTIONS] CONTAINER [CONTAINER...]
```

[Options]

-t, --time=10 Seconds to wait for stop before killing it

[Container]

Control the container either by Name or ID

Example

```
$ sudo docker start Your_Ubuntu_Container
```

Docker Execute

Run a command in a running container

Usage:

```
docker exec [OPTIONS] CONTAINER COMMAND [ARG...]
```

-d, --detach=false	Detached mode: run command in the background
-i, --interactive=false	Keep STDIN open even if not attached
-t, --tty=false	Allocate a pseudo-TTY
-u, --user=	Username or UID (format: <name uid>[:<group gid>])

Example:

```
$ docker exec -d ubuntu_bash touch /tmp/execWorks
```

This will create a new file /tmp/execWorks inside the running container ubuntu_bash, in the background.

```
$ docker exec -it ubuntu_bash bash
```

This will create a new Bash session in the container ubuntu_bash.

Docker Copy

Copy data [From|To] container, It copies data from and to running containers.

For More visit the [Copy documentation](#)

Usage

```
$ docker cp [OPTIONS] CONTAINER:PATH LOCALPATH| -
```

```
$ docker cp [OPTIONS] LOCALPATH| - CONTAINER:PATH
```

Example

Copy Data from Container to host

```
$ sudo docker cp testcopy:/root/file.txt .
```

Copy Data from host to container

```
$ sudo docker cp host.txt testcopy:/root/host.txt
```

Docker Attach

This commands brings a container to the foreground. The container must be running to be attached.

Usage

```
$ docker attach [OPTIONS] CONTAINER
```

Example

```
$ sudo docker attach Your_Ubuntu_Container
```

For more Information visit the [commandline page](#)

Docker Inspect

This command is used to check for the current configuration of a container from network to drivers and name. It prints the information in the form of JSON File

Usage

```
$ docker inspect [OPTIONS] CONTAINER|IMAGE [CONTAINER|IMAGE...]
```

Example

```
$ sudo docker inspect Your_Ubuntu_Container
```


Docker delete

This command is used to delete any container either by name or ID. Put Make sure that the container is stopped before you remove it.

Usage

```
$ docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

Example

```
$ sudo docker rm Your_Ubuntu_Container
```

Docker Generic Commands

The Docker Engine contains many useful containers

Docker Login

Register or log in to a docker registry.

Usage

```
docker login [OPTIONS] [SERVER]
```

[options]

-e, --email="" Email

-p, --password="" Password

-u, --username="" Username

[server]

Docker registry server, if no server is specified ("<https://index.docker.io/v1/>) is the default.

Example

```
$ sudo docker login
```

```
Username: XXXXX
```

```
Password:
```

```
Email: XXX@XXX.com
```

Warning: login credentials saved in /root/.docker/config.json

Login Succeeded

Also check the [logout command](#)

Docker Info

This command returns a list of any containers, any images (the building blocks Docker uses to build containers), the execution and storage drivers Docker is using, and its basic configuration.

Usage:

```
$ Docker info
```

Example

```
$ sudo docker info
```

```
Containers: 0
Images: 0
Storage Driver: devicemapper
Pool Name: docker-253:0-3022913-pool
Pool Blocksize: 65.54 kB
Backing Filesystem: xfs
Data file: /dev/loop0
Metadata file: /dev/loop1
Data Space Used: 1.821 GB
Data Space Total: 107.4 GB
Data Space Available: 12.85 GB
Metadata Space Used: 1.479 MB
Metadata Space Total: 2.147 GB
Metadata Space Available: 2.146 GB
Udev Sync Supported: true
Deferred Removal Enabled: false
Data loop file: /var/lib/docker/devicemapper/devicemapper/data
Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
Library Version: 1.02.93-RHEL7 (2015-01-28)
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.10.0-229.el7.x86_64
Operating System: CentOS Linux 7 (Core)
CPUs: 4
Total Memory: 1.948 GiB
Name: localhost.localdomain
ID: Q3D4:B0RV:ZTAW:QZ2P:I7DI:4SI3:62AT:5UUA:73NN:CVFC:FDKT:PJR4
```

Docker Diff

List the changed files and directories in a **container's filesystem** from the base image.

Usage

```
$ sudo docker diff container
```

Example

Assume There are 3 events that are listed in the diff:

- A - Add
- D - Delete
- C - Change

```
$ docker diff determined_bose
```

- C /root
- A /root/xyz
- A /root/.bash_history
- A /root/abc
- A /root/lmn

For more information visit the [official diff page](#)

Docker PS

Shows the current running containers on the system

Usage

```
$ docker ps [OPTIONS]
```

[OPTIONS]

-a all containers

-filter [] filtering container based on any parameter

Example

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
2e4bea848301	hello-world	"/hello"	About a minute ago	Exited (



For more information visit the [official docker documentation](#)

A very good discussion on the PS Command on [stackoverflow](#)

Tips and Tricks

Remove all containers

```
docker rm $(docker ps -aq)
```

Images and docker HUB

Docker Images and Containers

Docker images

A Docker image is a read-only template. For example, an image could contain an Ubuntu operating system with Apache and your web application installed. Images are used to create Docker containers. Docker provides a simple way to build new images or update existing images, or you can download Docker images that other people have already created. Docker images are the build component of Docker.

Docker containers

A container is the physically existing collection of processes that uses the OS resources to implement a functionality, a container is always based on an image and to turn an image into a container, the Docker engine takes the image, adds a read-write filesystem on top and initializes various settings including network ports, container name, ID and resource limits. A running container has a currently executing process, but a container can also be stopped (or exited to use Docker's terminology). An exited container is not the same as an image, as it can be restarted and will retain its settings and any filesystem changes.

Docker Images Commands

Docker Save and Load Images

Docker has two commands the save and load commands that allows you to have a portable version of a docker image.

For more check the save and load pages in the documentation.

Usage

```
$ docker save [OPTIONS] IMAGE [IMAGE...]
[options]
-o, --output=""      Write to a file, instead of STDOUT
$ docker load [OPTIONS]
[options]
-i, --input=""       Read from a tar archive file, instead of STDIN. The tarball may be com
```



Example

```
$ docker save -o ubuntu.tar ubuntu:luuid ubuntu:saucy
$ docker load --input fedora.tar
```

Docker Images

Shows the currently available images. Images are saved in /var/lib/docker/containers directory.

For More Information visit the [Images command page](#).

Usage

```
$ Docker images [OPTIONS] [REPOSITORY]
```

Example

```
$ sudo docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL S
hello-world	latest	af340544ed62	8 weeks ago	960 B

Docker Pull

Docker pull commands gets an images from the Docker Registry either local or the global one.

This command is called if the run commands can't find the stated image.

For More information visit the [pull command page](#)

usage

```
$docker pull [OPTIONS] NAME[:TAG] | [REGISTRY_HOST[:REGISTRY_PORT]/]NAME[:TAG]
```

[options]

```
-a, --all-tags=false          Download all tagged images in the repository
--disable-content-trust=true Skip image verification
```

Example

```
$ docker pull debian
```

will pull the debian:latest image and its intermediate layers

Docker Search

Docker search commands search on the docker images repositories to find images For More information visit the [search command page](#)

Usage

```
$docker search [OPTIONS] TERM
```

[options]

```
--automated=false    Only show automated builds  
--no-trunc=false     Don't truncate output  
-s, --stars=0        Only displays with at least x stars
```

Building Docker Images

Creating a custom docker image is not an easy task. There is two ways to create such image.

1. Docker Commit

This the old way of creating an image.

2. Docker File

The new and recommend way

Docker Commit

This command will enable you to update the image you're using by installing the required tools and packages and save it to be used by other users. The commit command enable you to save a container to an image to be reusable.

Usage:

```
docker commit [OPTIONS] CONTAINER [REPOSITORY[:TAG]]
[options]
-a, --author=""      Author (e.g., "John Hannibal Smith <hannibal@a-team.com>")
-c, --change=[]      Apply specified Dockerfile instructions while committing the image
-m, --message=""     Commit message
-p, --pause=true     Pause container during commit
```

Demo:

The Demo will start by using an Ubuntu image and install Nginx webserver on it and save it locally and on my local docker hub registry.

Step 1: Run the container and open it from base image

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

Step 2: Install required packages and tools

```
/# apt-get install nginx
```

Step 3: Exit the container

```
/# exit
```

Step 4: Find the edited container ID

```
$ sudo docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
238c9baddb08	Ubuntu	"/bin/bash"	3 days ago	Up 59 min

Step 5: Commit the image

```
$ sudo docker commit -m="A new nginx image" --author="washraf" 238c9baddb08 washraf/nginx
```

```
91064f90fa2c120fb7b10ec7354ab0ed5cdb36b942b2b6df65975f5cdc2403cc
```

Step 6: Check the image `$sudo docker images`

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL S
walid/nginx	latest	91064f90fa2c	32 seconds ago	206.5 MB
ubuntu	latest	91e54dfb1179	6 weeks ago	188.3 MB
hello-world	latest	af340544ed62	8 weeks ago	960 B

Step 7: inspect the image

```
$sudo docker inspect walid/nginx
```

... The data will appear as configured

Step 8: Run the container from the new Image

```
$ sudo docker run walid/nginx $ sudo docker run -i -t walid/nginx /bin/bash
```

Docker Build

This is the recommended way to create a docker image.

Demo

Step 1: Create directory

```
$ mkdir NginxDir
```

Step 2: Create file

```
$ vi Dockerfile
```

Step 3: Write this commands in the file

```
# Version: 0.0.1
FROM ubuntu
MAINTAINER Walid Ashraf "w@w.com"
RUN apt-get install -y nginx
RUN echo 'Hi, I am in your container' \
>/usr/share/nginx/html/index.html
EXPOSE 80
```

Step4: Build Image

```
$ sudo docker build -t="walid/ngtest" .
```

...Successfully built d69edbcc0346

Step 5: Check the Image

```
$ sudo docker images "walid/ngtest" .
```

REPOSITORY	TAG	IMAGE ID	CREATED	VIRTUAL S
washraf/ngtest	latest	d69edbcc0346	2 minutes ago	206.5 MB



Step 6: Run the container will the webserver

```
sudo docker run -d -p 80 washraf/ngtest nginx -g "daemon off;"
```

Step 7: check the assigned port from the docker daemon

```
$ sudo docker ps
```


CONTAINER ID	IMAGE	COMMAND	CREATED	STAT
6d575aeb39f1	washraf/ngtest	"nginx -g 'daemon off'"	2 minutes ago	Up 2



Step 8: visit the site



For more about building docker images check the See Docker File part and the [build command page](#)

Docker File Instructions

Creating a docker file is the most used way to create a docker image. In this section we are going to explore the ways to create a docker image file.

For More information visit [the docker builder](#).

From:

This instructions sets the base image for the new Image. This is considered the most important instruction in the Dockerfile. This instruction must be the first one to be written and it treats the image as the docker run do. If the images are found locally it's used and if not it will be pulled from the assigned registry and finally if the image is not found it will raise a build error.

Usage

```
FROM <image>
```

```
FROM <image>:<tag>
```

```
FROM <image>@<digest>
```

Maintainer:

Who is the Image creator

Usage

```
MAINTAINER <Name>
```

WORKDIR:

Sets the start directory for the container where the run and the CMD instructions are executed.

Usage

```
WORKDIR /path/to/workdir
```

RUN:

The RUN instruction will execute any commands in a new layer on top of the current image and commit the results. The resulting committed image will be used for the next step in the Dockerfile.

Usage

```
RUN <command> (the command is run in a shell - /bin/sh -c - shell form)
```

```
RUN ["executable", "param1", "param2"] (exec form)
```

CMD:

The CMD Command is used as a default entry point to container. And if more than one is added the last one only will take effect.

Usage

```
CMD ["executable","param1","param2"] (exec form, this is the preferred form)
```

```
CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
```

```
CMD command param1 param2 (shell form)
```

COPY:

Copies files from the host file system to the image file system.

Usage

```
COPY <src>... <dest>
```

```
COPY ["<src>","... "<dest>"] (this form is required for paths containing whitespace)
```

ADD:

The add file is similar to the copy but can handle both tar files and remote URLs

Usage

```
ADD <src>... <dest>
```

```
ADD ["<src>","... "<dest>"] (this form is required for paths containing whitespace)
```

ENV:

Sets some environment variables for the container

Usage

```
ENV <key> <value> only one perline
```

```
ENV <key>=<value> ... allows multiple per line
```

USER:

Sets the user that will be running in the container and the default is the root.

Usage

```
USER <UID>|<UName>
```

EXPOSE:

This allows the container ports to be accessed from the global ports. Note that Port mapping has to be specified directly at creation time using the `-p` or `-P` commands

Usage

```
EXPOSE <port> [<port>...]
```

ENTRYPOINT:

The ENTRYPOINT instruction will help in crafting an image for running an application (entry point) during the complete life cycle of the container, which would have been spun out of the image. When the entry point application is terminated, the container would also be terminated along with the application and vice versa. The ENTRYPOINT instruction cannot be overridden in the run command.

Usage

```
ENTRYPOINT ["executable", "param1", "param2"] (the preferred exec form)
```

```
ENTRYPOINT command param1 param2 (shell form)
```

Vlolume:

The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

Usage

`VOLUME ["/data"]`

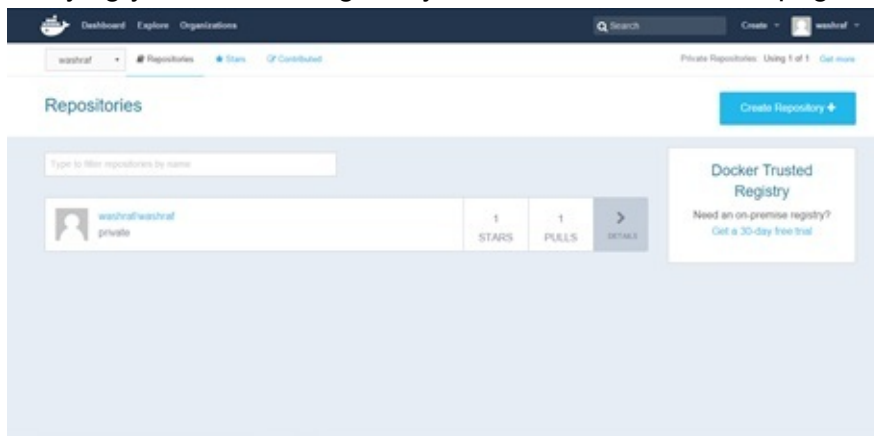
It is generally considered a bad idea to run commands like `apt-get -y update` or `yum -y update` in your application Dockerfiles because it can significantly increase the time it takes for all of your builds to finish. Instead, consider basing your application image on another image that already has these updates applied to it.

Using Docker HUB

Docker hub is the host of the version control and host for all the public and private owned images for both personal and enterprise usage.

Create docker HUB Account

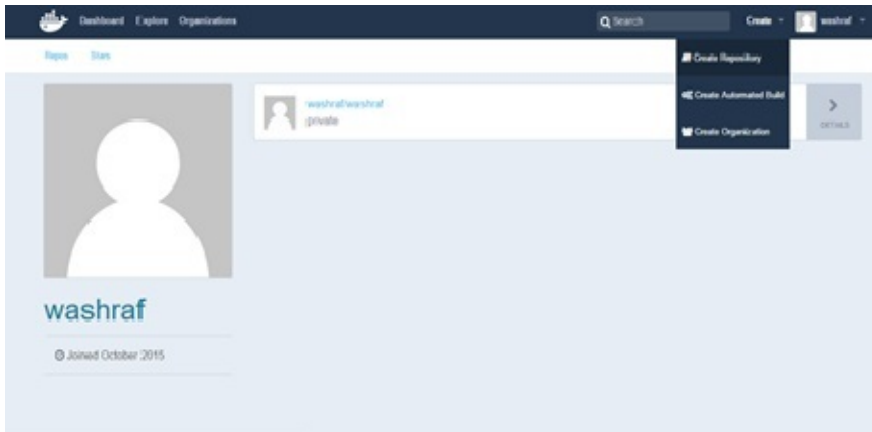
Visit the [signup page](#) and create an account using you username and password. After verifying your account login to your account and visit the page.



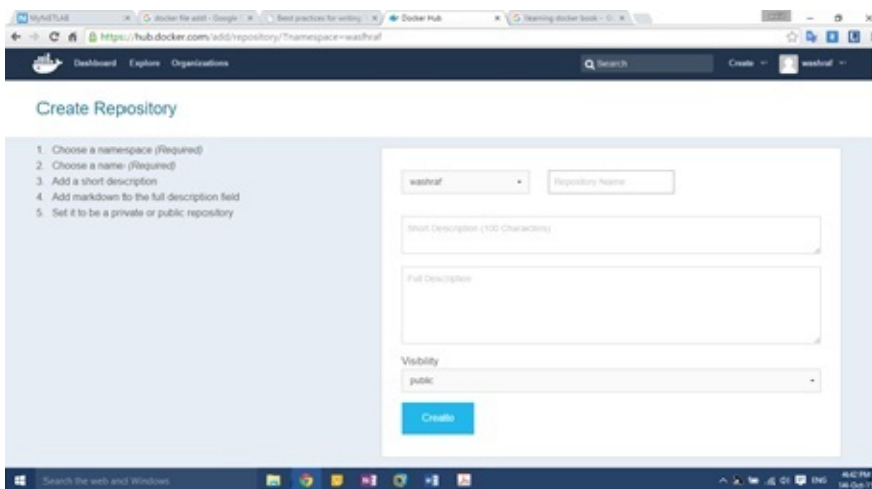
Push Image to Docker HUB

Create a Repository

Login to your account and click the create button and choose to create Repository.



Fill the following form with the repository name, description and visibility.



Note that the name of the repo is always addressed with the name space e.g. washraf/repo this is used to differentiate the original images created by the docker team from the images created by normal users.

Login to you account on engine

Using the login command you must supply your username, password and mail to configure your docker daemon with your account.

```
$sudo docker login
Username: blabla
Password:
Email blabla@mail.com
```

Push an Image

```
$sudo docker push walid/test
```

Make sure that you have a repo by this name or do this command:

```
sudo docker tag walid/nginx username/reponame
```


Docker in Depth

In this chapter we explore some advanced techniques in the docker eco system that includes:

- 1. docker communication**
- 2. Management Data Volumes**
- 3. Resource Control**
- 4. Docker Behind the Scenes**
- 5. Configure Docker Deamon**
- 6. Docker Security**
- 7. Docker API**

Docker Links

Docker has a **linking** system that allows you to link multiple containers together and send connection information from one to another. When containers are linked, information about a source container can be sent to a recipient container. This allows the recipient to see selected data describing aspects of the source container. To establish links, Docker relies on the names of your containers. Links allow containers to discover each other and securely transfer information about one container to another container. When you set up a link, you create a conduit between a source container and a recipient container. The Source can access the linked container in a secured setup, is that the linked containers can communicate using secured tunnels without exposing the ports used for the setup, to the external world.

For more Information visit the [docker links guide](#)

Usage

```
--link <name or id>: alias
```

If the alias is omitted the name will be set to be the same name as the container name but note that both the name and the alias should be unique to the host.

How Linking Works

When two containers are linked together, the Docker engine automatically exports a few environment variables to the recipient container. These environment variables have a well-defined naming convention, where the variables are always prefixed with capitalized form of the alias name.

For example if a container is linked to a MySQL image with alias SQLDB there are three rules to naming environment variables in the target container:

- If the source container has an environment variable called Name an environment variable name SQLDB_NAME will be created
- Environment variables that are set using the `-e` option in the run command or the ENV command in the docker file are named `_ENV_EVNAME` e.g.
SQLDB_ENV_PASSWORD=123456
- Port values that are set using the `-p` option in the run command or the EXPOSE command the docker file are named `_PORT_PROTOCOL_PORTNO_ADDR` e.g.
SQLDB_PORT_80_TCP_PORT=336 or SQLDB_PORT_80_TCP=tcp://172.17.0.4:80

DEMO

Create two container and link them together and check for the /etc/hosts file and all the environment variables.

```
$sudo docker run --name source -it -e w="Walid" -p 80:80 ubuntu /bin/bash
```

This will open an ubuntu container in interactive tty mode and set an enviroment variable called W with Walid as a value also expose the port 80 and map it to the host's port 80.

And in another console:

```
$sudo docker run --name target --link source:src -it ubuntu /bin/bash
```

```
root@ 765ad6da2b4e:/ cat etc/hosts
...
172.17.0.4      src 87de493a5d62 source
172.17.0.4      source
172.17.0.4      source.bridge
172.17.0.5      target
172.17.0.5      target.bridge
...
root@ 765ad6da2b4e:/ env
...
SRC_NAME=/target/src
SRC_ENV_w=Walid
SRC_PORT_80_TCP_PORT=80
SRC_PORT_80_TCP_ADDR=172.17.0.4
SRC_PORT_80_TCP=tcp://172.17.0.4:80
SRC_PORT=tcp://172.17.0.4:80
...
```

Docker Networking

The Docker engine seamlessly selects and assigns an IP address to a container with no intervention from the user, when it gets launched. Well, you might be puzzled on how Docker selects an IP address for a container, and this puzzle is answered in two parts, which are as follows:

1. During the installation, Docker creates a virtual interface with the name **docker0** on the Docker host. It also selects a private IP address range, and assigns an address from the selected range to the **docker0** virtual interface. This selected IP address is always outside the range of the Docker host IP address in order to avoid an IP address conflict.
2. Later, when we spin up a container, the Docker engine selects an unused IP address from the IP address range selected for the **docker0** virtual interface. Then, the engine assigns this IP address to the freshly spun container.

Port Mapping

Port mapping allows a mapping between the host ports with the containers inner port. The port mapping is configured either by `-p` or `-P` options in the run command.

-p example

These run options allows the user to map a port container port to a host port

```
$ sudo docker run -p 1234:80 imagename
```

-P example

These run options allows the user to map all ports exposed by container to ports in the host

```
$ sudo docker run -P imagename
```

Finding mapped port

Finding the mapped port could be found by the command **ps** but as a better way is the port command for more information visit [the port guide](#)

Usage

```
docker port CONTAINER_Name [PRIVATE_PORT[/PROTO]]
```

Example

```
$sudo docker port test_server
```

```
8080/tcp -> 0.0.0.0:1234
```

```
$sudo docker port test_server 1234
```

```
0.0.0.0:1234
```

```
$sudo docker port test_server 1234/tcp
```

```
0.0.0.0:1234
```

Advanced Network Configuration

Docker creates a virtual interface and switch named `docker0` to connect different container and enable each container to have its own network configuration.

Docker configures this switch to private network ip **172.17.42.1/16** allowing **65536** ip and The **MAC address** is generated using the IP address allocated to the container to avoid ARP collisions, using a range from **02:42:ac:11:00:00** to **02:42:ac:11:ff:ff**.

For more check this [article about docker networking](#)

Docker0

Docker0 is the virtual interface created by the docker daemon in the installation time.

```
$ ifconfig docker0
```

The **second** line of the output:

```
inet 172.17.42.1 netmask 255.255.0.0 broadcast 0.0.0.0
```

How container networking work

Containers are bound to a host connection unless two thing are configured to allow such configuration

IP_Forward

Is the host machine willing to forward IP packets? This is governed by the `ip_forward` system parameter for more check out this article about [IP forwarding](#)

IPTables

Iptables is a command-line firewall utility that uses policy chains to allow or block traffic. When a connection tries to establish itself on your system, iptables looks for a rule in its list to match it to. If it doesn't find one, it resorts to the default action.

Types of Chains :

iptables uses three different chains:

- **Input:**

This chain is used to control the behavior for incoming connections. For example, if a user attempts to SSH into your PC/server, iptables will attempt to match the IP address and port to a rule in the input chain.

- **Forward:**

This chain is used for incoming connections that aren't actually being delivered locally. Think of a router – data is always being sent to it but rarely actually destined for the router itself; the data is just forwarded to its target. Unless you're doing some kind of routing, NATing, or something else on your system that requires forwarding, you won't even use this chain.

- **Output:**

This chain is used for outgoing connections. For example, if you try to ping howtogeek.com, iptables will check its output chain to see what the rules are regarding ping and howtogeek.com before making a decision to allow or deny the connection attempt.

Assigned IP:

There are two ways to find the assigned IP Address for a container

Ifconfig:

you can find the ip address from the container it self

```
root@153151351# ifconfig
```

inspect

the inspect command shows every configuration about the container

```
$ sudo docker inspect --format='{.NetworkSettings.IPAddress}' container_name
```

```
172.17.0.69
```

This is the assigned IP address to the container and you can even ping it.

Assign an IP to a container

Assign a DNS to a container

Mac Address

Docker Weave

I have not check this tool out but these are very useful **links**:

1. <http://weave.works/guides/weave-docker-ubuntu-simple.html>
2. <https://github.com/weaveworks/weave>
3. <http://blog.weave.works/2015/11/03/docker-networking-1-9-weave-plugin/>

Data Volumes

Docker file system is in memory files systems (i.e. files are deleted with the container itself) to mitigate such problems docker has two methods for managing a data in containers Data volumes and Data Volume Containers.

Another important thing Data volumes that they allow you to specify a file outside the UNION file system which is faster.

Usecases for Data Volumes :

1. The need for faster R/W than Union file system.
2. Sharing Data between Container and Host.
3. Data Persistence.

For More Information check the datavolumes [user guide](#)

Mount a host directory as a data volume

This could be achieved by the `-v` option in the run command. This allows you both to create a directory or to map an already existing one to a container.

Example

```
root@localhost# docker run --name Name -v /Shared:/SharedData -i -t ubuntu /bin/bash
```

This command will create a container with a shared folder with the host **/ValidShared** Folder and will be mounted as **/SharedData** in the container and if the folder didn't exist in the host it will be automatically created.

The other options is giving a name to the container, opening its command shell with bash shell.

```
root@localhost# docker inspect Name
```

```
...
"Mounts": [
  {
    "Source": "Shared",
    "Destination": "/SharedData",
    "Mode": "",
    "RW": true
  }
]
...
```

```
root@d2040412b5bf:/# ls
```

```
...
SharedData
...
```

If any thing is written in the **SharedData** Volume it will be written in the **/WalidShared** volume

Creating a Data Volume

This is done using the VOLUME command in the Docker file. And its saved in this option limits to creating mounted volumes.

Container Data Volumes

This method uses a container as a shared data volume and called data only container.

```
Docker run -v /var_volume1 --name datavolume postgres
```

```
Docker run -volumes-from datavolume --name client 1 postgres
```

```
Docker run -volumes-from datavolume --name client 2 postgres
```

this will create two postgres containers that shares the same data volume container

Remember to use the same base image

Resource Control

Control Memory, CPU, IO Resources and also control the process capabilities and how to account for resource usage.

Memory Control

Controlling memory is core management utility of containers.

Options to control the memory of a container :

-m, --memory=""

Memory limit (format: [], where unit = b, k, m or g)

--memory-swap=""

Total memory limit (memory + swap, format: [], where unit = b, k, m or g)

--oom-kill-disable

By default kernel destroys a process which tries to pass its limit or have no more available memory this option allows us to disable this for the kernel.

--memory-swappiness=0

By default, a container's kernel can swap out a percentage of anonymous pages. To set this percentage for a container, specify a --memory-swappiness value between 0 and 100. A value of 0 turns off anonymous page swapping. A value of 100 sets all anonymous pages as swappable.

--cpuset-mems=""

Memory nodes (MEMs) in which to allow execution (0-3, 0,1). Only effective on NUMA systems. Controlling memory resources is done via the combinations of two options (Memory and Memory Swap).

We have four main ways to set memory usage:

- **memory=inf, memory-swap=inf (default)**

There is no memory limit for the container. The container can use as much memory as needed.

- **memory=L<inf, memory-swap=inf**

(specify memory and set memory-swap as -1) The container is not allowed to use more than L bytes of memory, but can use as much swap as is needed (if the host supports swap memory).

- **memory=L<inf, memory-swap=2*L**

(specify memory without memory-swap) The container is not allowed to use more than L bytes of memory, swap *plus* memory usage is double of that.

- **memory=L<inf, memory-swap=S<inf, L<=S**

(specify both memory and memory-swap) The container is not allowed to use more than L bytes of memory, swap *plus* memory usage is limited by S.

Examples

1- Run an Ubuntu container in interactive tty mode with a 300Mega memory and no limit on memory swap (-1)

```
$ sudo docker run -it -m 300M --memory-swap -1 ubuntu:14.04 /bin/bash
```

2- Run an Ubuntu container in interactive tty mode with a 300Mega memory and max memory of 1G.

```
$ docker run -ti -m 300M --memory-swap 1G ubuntu:14.04 /bin/bash
```

3- Run an Ubuntu container in interactive tty mode which restricts the processes in the container to only use memory from memory nodes 1 and 3.

```
$ docker run -ti --cpuset-mems="1,3" ubuntu:14.04 /bin/bash
```

4- Run an Ubuntu container in interactive tty mode restricting the processes in the container to only use memory from memory nodes 0, 1 and 2.

```
$ docker run -ti --cpuset-mems="0-2" ubuntu:14.04 /bin/bash
```

CPU Control

Controlling CPU is core management utility of containers.

Options to control the CPU of a container:

-c, --cpu-shares=0

CPU shares (relative weight)

--cpu-period=0

Limit the CPU CFS (Completely Fair Scheduler) period

--cpu-quota=0

Limit the CPU CFS (Completely Fair Scheduler) quota

--cpuset-cpus=""

CPUs in which to allow execution (0-3, 0,1)

CPU share constraint:

By default, all containers get the same proportion of CPU cycles. This proportion can be modified by changing the container's CPU share weighting relative to the weighting of all other running containers. To modify the proportion from the default of **1024**, use the **-c** or **--cpu-shares** flag to set the weighting to 2 or higher.

The proportion will only apply when CPU-intensive processes are running. When tasks in one container are idle, other containers can use the left-over CPU time. The actual amount of CPU time will vary depending on the number of containers running on the system.

CPU Period Constraint and Quota

The default CPU CFS (Completely Fair Scheduler) period is 100ms. We can use **--cpu-period** to set the period of CPUs to limit the container's CPU usage. And usually **--cpu-period** should work with **--cpu-quota**.

The **--cpu-quota** flag limits the container's CPU usage. The default 0 value allows the container to take 100% of a CPU resource (1 CPU). The CFS (Completely Fair Scheduler) handles resource allocation for executing processes and is default Linux Scheduler used by

the kernel. Set this value to 50000 to limit the container to 50% of a CPU resource. For multiple CPUs, adjust the **--cpu-quota** as necessary For more about scheduling check the [Linux Scheduling documentation](#)

CPU Set (CPU pinning)

It is also possible to pin a container to one or more CPU cores. This means that work for this container will only be scheduled on the cores that have been assigned to this container.

Examples:

1- CPU Period Control: this means the container can get 50% CPU worth of run-time every 50ms.

```
$ docker run -ti --cpu-period=50000 --cpu-quota=25000 ubuntu:14.04 /bin/bash
```

2- Run and Ubuntu container in interactive tty mode where it can be executed on cpu1 and cpu3.

```
$ docker run -ti --cpuset-cpus="1,3" ubuntu:14.04 /bin/bash
```

3 - Run and Ubuntu container in interactive tty mode where it can be executed on cpu0, cpu1 and cpu2.

```
$ docker run -ti --cpuset-cpus="0-2" ubuntu:14.04 /bin/bash
```


IO Control

Block IO bandwidth (Blkio) constraint

By default, all containers get the same proportion of block IO bandwidth (blkio). This proportion is 500. To modify this proportion, change the container's blkio weight relative to the weighting of all other running containers using the **--blkio-weight** flag. The **--blkio-weight** flag can set the weighting to a value between 10 to 1000.

Examples

```
$ docker run -ti --name c1 --blkio-weight 300 ubuntu:14.04 /bin/bash
```

```
$ docker run -ti --name c2 --blkio-weight 600 ubuntu:14.04 /bin/bash
```

The blkio weight setting is only available for direct IO. Buffered IO is not currently supported.

Control Capabilities

Accounting for usage

1. <https://docs.docker.com/engine/articles/runmetrics/>
2. <https://www.datadoghq.com/blog/monitor-docker-datadog/>
3. <https://goldmann.pl/blog/2014/09/11/resource-management-in-docker/>
4. <http://serverfault.com/questions/615854/counting-bandwidth-from-a-docker-container>

Docker Behind the Scenes

Docker is a process isolation tool which allows the existence of two processes without affecting each other. In this section, we will discuss how the Docker engine provides process isolation by leveraging the Linux namespaces.

Process NameSpaces

The process namespace allows a translation between a two versions of a process id which is ideal for containers allowing a process two have an id that is used inside the container and another one outside it.

Validating Name Spaces translation

In the following set of experiments we are going to show how it's done.

Step 1: Create A normal contianer Create an Ubuntu container in an interactive mode

```
# docker run -it ubuntu /bin/bash
```

```
root@1b51c60ecc24:/#
```

Step 2: Find about the current containers

```
# docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS
1b51c60ecc24	ubuntu	"/bin/bash"	About a minute ago	Up About

Step 3: Find process information about the created container

```
# docker inspect --format "{{.State.Pid }}" 1b51c60ecc24
```

```
$$40085$$
```

Step 4: Test this process on the host itself

```
# ps -fp 40085
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	40085	18860	0	14:06	pts/3	00:00:00	/bin/bash

```
# cat /proc/40085/environ
```

```
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin^@HOSTNAME=1b51c60ecc24^
```

Step 5: A view inside the container itself `root@1b51c60ecc24:/# ps -ef`

UID	PID	PPID	C	STIME	TTY	TIME	CMD
root	1	0	0	12:06	?	00:00:00	/bin/bash
root	15	1	0	12:19	?	00:00:00	ps -e

How this works

In the Linux world, every system has just one root process with the **PID 1** and **PPID 0**, which is the root of the complete process tree of that system. The Docker framework cleverly leverages the Linux PID namespace to spin a completely new process tree; thus, the processes running inside a container have no access to the parent process of the Docker host. However, the Docker host has a complete view of the child PID namespace spun by the Docker engine.

The **PID** namespace provides consistent, virtual resource names in place of host-dependent resource names. Such PIDs within a container are trivially assigned in a unique manner in the same way that traditional operating systems assign names, but such names are localized to the container. Since the namespace is private to a given container, there are no resource naming conflicts for processes in different containers.

As a result, processes are created inside of a container and spend their entire lifetimes in the context of that container; they are not allowed to leave one container and join another.

Docker uses some sort of **translation hash table** between the process ids in the container and how its viewed by the host.

File System

The container root file system could be accessed using the process id we learned from the previous example.

Step 1 : Using the last process id 40085

```
# cd /proc/40085/root; ls
```

```
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys
```

A terminal window with a light gray background. It shows the command '# cd /proc/40085/root; ls' and the output 'bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys'. The window has a scrollbar on the right side.

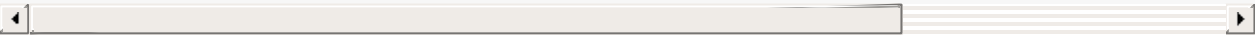
Step 2 : Adding a file from host

```
# touch testfromroot
```

Step 3 : Go to container

```
root@1b51c60ecc24:/# ls
```

```
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys
```

A terminal window with a light gray background. It shows the command 'root@1b51c60ecc24:/# ls' and the output 'bin boot dev etc home lib lib64 media mnt opt proc root run sbin srv sys'. The window has a scrollbar on the right side.

Step 4 : Stop the container and check the file system

```
# cd /proc/40085
```

```
-bash: cd: /proc/40085: No such file or directory
```

Docker Daemon Configuration

Docker Daemon [reference page](#)

Storage Drivers

Storage drivers for Daemon

<https://zwischenzugs.wordpress.com/2015/04/20/storage-drivers-and-docker/>

Storage drivers for Registry

<https://docs.docker.com/registry/storagedrivers/>

There are three options:

1. Local File System
2. Amazon S3
3. Azure Blobs

Difference between Storage drivers

AUFS:

Device Mapper

<http://developerblog.redhat.com/2014/09/30/overview-storage-scalability-docker/>

Changing the storage drivers

<http://muehe.org/posts/switching-docker-from-aufs-to-devicemapper/>

Docker Security

Docker Sockets and Network Ports

Docker registered its own TCP port with IANA and it's now generally configured to use TCP port 2375 when running un-encrypted, or 2376 when handling encrypted traffic. In Docker 1.3 and later, the default is to use the encrypted port on 2376, but this is easily configurable. The UNIX socket is located in different paths on different operating systems, so you should check where yours is located. If you have strong preferences, you can usually specify this at install time. If you don't, then the defaults will probably work for you.

Configuring sockets and security

Docker non root main user

Docker API

https://docs.docker.com/reference/api/docker_remote_api_v1.20/

Docker Demos

This chapter introduces some of the use cases addressed by the docker tools.

The demos includes:

1. Node JS application
2. Wordpress site
3. Ruby on Rails application
4. BUILDING PUPPET OR CHEF OR JUJU OR ANSIBLE
5. USING JENKINS FOR CONTINOUS DELIVERY
6. BUILDING MPI CLUSTER ON DOCKER
7. BUILDING HADOOP CLUSTER ON DOCKER

Node JS

In this demo we are going to create a container that hosts a NodeJS application. This demo is based on [docker example](#)

Create NodeJS Application

Step 1: Create Package.Json file

```
{
  "name": "docker-ubuntu-hello",
  "private": true,
  "version": "0.0.1",
  "description": "Node.js Hello world app on ubuntu using docker",
  "author": "Yourname<Mail@Mail.com>",
  "dependencies": {
    "express": "3.2.4"
  }
}
```

Step 2: Create Index.JS File

```
var express = require('express');

// Constants
var PORT = 8080;

// App
var app = express();
app.get('/', function (req, res) {
  res.send('Hello world\n');
});

app.listen(PORT);
console.log('Running on http://localhost:' + PORT);
```

Step 3: Create A Docker File

```
FROM      ubuntu
#make ure apt is up to date
RUN apt-get update
# install nodejs and npm
RUN apt-get install -y nodejs npm
# Get The Code
COPY . /src
# Install app dependencies
RUN cd /src; npm install
EXPOSE    8080
CMD ["nodejs", "/src/index.js"]
```

Step 4: Build The Image from Docker File

```
$ sudo docker build -t washraf/washraf .
```

Step 5: Run the Docker Image

```
$ sudo docker run -p 1234:8080 -d washraf/washraf
```

Step 6: test the application

```
$curl -i localhost:1234
```

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 12
Date: Tue, 06 Oct 2015 23:57:48 GMT
Connection: keep-alive

Hello world
```

Wordpress and mysql

This demo will allow you to create a two tier application using ruby on rails and mysql.

Create MySQL server

Mysql is by far the most used database in the open source community so a docker image have been created with mysql installed on it.

Step 1: Create Docker File

Create a folder name mysqlImage and create Dockerfile that contains the instructions needed for the image.

```
#mkdir mysqlImage
```

```
#cd mysqlImage
```

```
#vi Dockerfile
```

In the docker file write the needed configurations:

```
# Version: 0.0.1
FROM mysql
#use mysql image to be our base image
MAINTAINER Name name@name.com
VOLUME /SQLDBFILES:/var/lib/mysql
#set the Database super user password
ENV MYSQL_ROOT_PASSWORD w@lid
#make sure that mysql Server is running
RUN service mysql start
# The Image default expose
# EXPOSE 3306
```

Step 2: Build docker Image

```
# docker build -t washraf/mysql .
```

Step 3: run the container

```
# docker run --name wpsql -d washraf/mysql
```

Create My WordPress container

Step 1: run the wordpress image

```
# docker run -p 80:80 --link wpsql:mysql -d wordpress
```

Step2: open the browser

RubyonRails and mysql

Create MySQL server

Mysql is by far the most used database in the open source community so a docker image have been created with mysql installed on it.

Step 1: Create Docker File

Create a folder name mysqlImage and create Dockerfile that contains the instructions needed for the image.

```
#mkdir mysqlImage
```

```
#cd mysqlImage
```

```
#vi Dockerfile
```

In the docker file write the needed configurations:

```
# Version: 0.0.1
FROM mysql
#use mysql image to be our base image
MAINTAINER Name name@name.com
VOLUME /SQLDBFILES:/var/lib/mysql
#set the Database super user password
ENV MYSQL_ROOT_PASSWORD w@lid
#make sure that mysql Server is running
RUN service mysql start
# The Image default expose
# EXPOSE 3306
```

Step 2: Build docker Image

```
# docker build -t washraf/mysql .
```

Step 3: run the container

```
# docker run --name wsql -d washraf/mysql
```


Step 4: find the assigned IP

```
# docker inspect wsq1
```

Ruby on Rails

Step 1: create the project and configure it to use MySQL

Step 2: configure the database source in the code

```
username: root #the user you uses  
password: w@lid #the password you save  
host: 172.17.0.61 #the assigned ip
```

Step 3: Create a deployment and run script SCRIPT.sh

```
/bin/bash  
# My first script  
cd /root/trial  
bundle install  
rake db:create  
rake db:migrate  
rails server -b 0.0.0.0
```

Step 4: Create Dockerfile

```
From rails  
COPY /trial/ /root/trial/  
COPY /SCRIPT.sh /root/  
#ENTRYPOINT /root/SCIRPT.sh  
EXPOSE 3000
```

Step 5: Build DockerFile

```
docker build --no-cache -t washraf/rails .
```

Step 6: start the container

```
docker run -d -p 3000:3000 washraf/rails /root/SCRIPT.sh
```

Dot Net

https://blogs.msdn.microsoft.com/mvpawardprogram/2015/12/15/getting-started-with-net-and-docker/?hash=c3229b6d-74c6-42bc-84aa-0aa8f6757f95&utm_medium=social&utm_source=facebookpage&utm_campaign=docker-2681022