CLIENT SERVER COMMUNICATION USING DOCKER

Introduction to Docker:

Docker is a platform for Developers and System admins to develop, ship, and run applications. Docker lets you quickly assemble applications from components and eliminates the friction that you can come when shipping the code. Docker lets your code tested and deployed into production as fast as possible. Docker provides an additional layer of abstraction and automation of operating-system-level virtualization on Windows and Linux. Docker uses resource isolation features of Linux Kernel such as cgroups and kernel namespaces, and a union-capable file system such as OverlayFS and others to allow independent “containers” to run within a single Linux instance, avoiding the overhead of starting and maintaining the virtual machines.

Docker Containers:

Using Docker Containers, everything required to make a piece of software run is packaged into isolated containers. Unlike VMs, containers don’t bundle a full operating system-only libraries and settings required to make the software work are needed. This makes for efficient, lightweight, self-contained systems and guarantees that software will always run the same, regardless of where it is deployed.

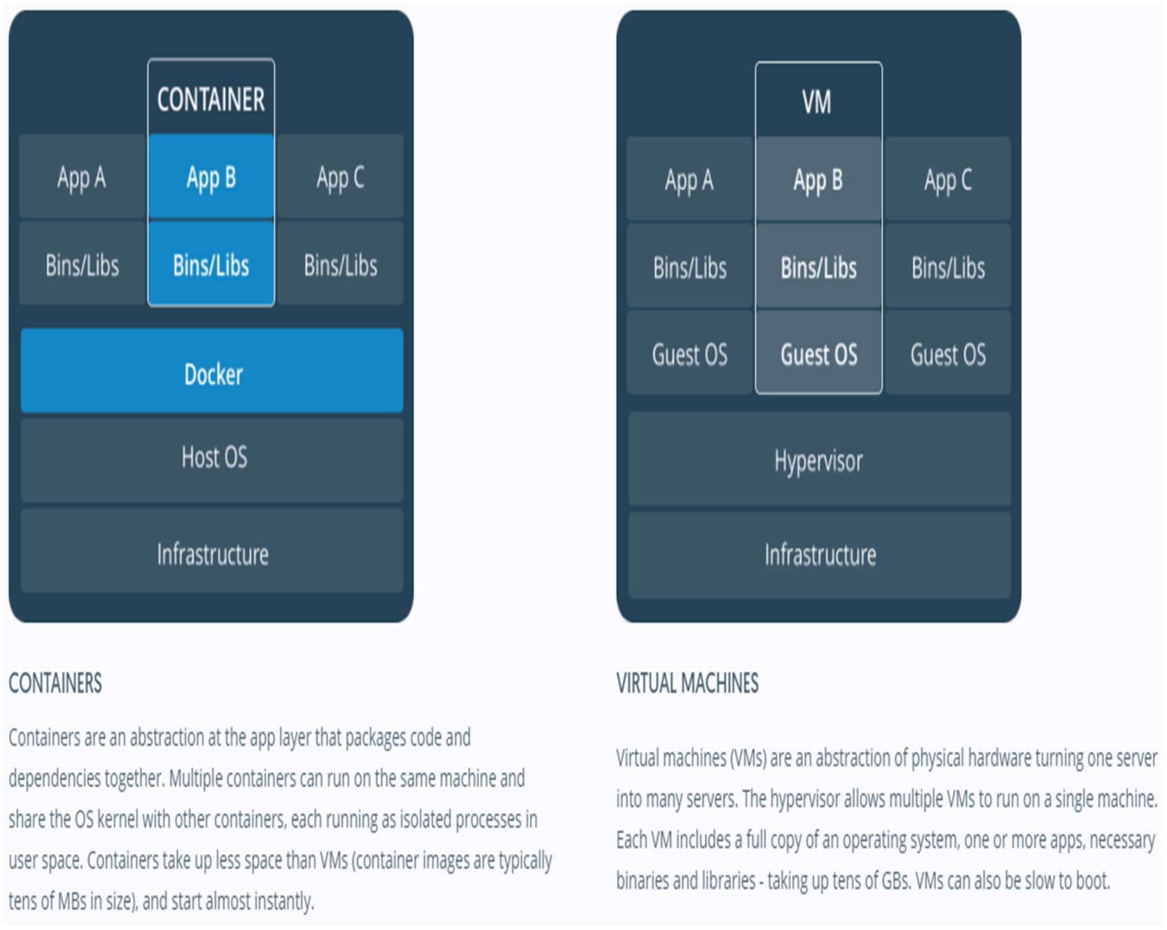
Docker Container versus Virtual Machines:

When it comes to comparing the two, it could be said that Docker Containers have much more potential than Virtual Machines. It’s evident as Docker Containers are able to share a single kernel and share application libraries. Containers present a lower system overhead than Virtual Machines and performance of the application inside a container is generally same or better as compared to the same application running within a Virtual Machine.

There is one key metric where Docker Containers are weaker than Virtual Machines, and that’s “Isolation”. Intel’s VT-d and VT- x technologies have provided Virtual Machines with ring-1 hardware isolation of which, it takes full advantage. It helps Virtual Machines from breaking down and interfering with each other. Docker Containers yet don't have any hardware isolation, thus making them receptive to exploits.

As compared to virtual machines, containers can be faster and less resource heavy as long as the user is willing to stick to a single platform to provide the shared OS. A virtual machine could take up several minutes to create and launch whereas a container can be created and launched just in a few seconds. Applications contained in containers offer superior performance, compared to running the application within a virtual machine.

There is an estimation being done by Docker that application running in a container can go twice as fast as one in a virtual machine. Also, a single server can pack more than one containers as OS is not duplicated for each application.



CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), and start almost instantly.

VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Fig: Docker Container versus Virtual Machines

A Container is a runtime instance of a Docker Image — what the image becomes in memory when actually executed. It runs completely isolated from the host environment by default, only accessing host files and ports if configured to do so.

Containers run apps natively on the host machine's kernel. They have better performance characteristics than virtual machines that only get virtual access to host resources through a hypervisor. Containers can get native access, each one running in a discrete process, taking no more memory than any other executable.

Containers can share a single kernel, and the only information that needs to be in a container image is the executable and its package dependencies, which never need to be installed on the host system. These processes run like native processes, and you can manage them individually by running commands like docker ps—just like you would run ps on Linux to see active processes. Finally, because they contain all their dependencies, there is no configuration entanglement; a containerized app "runs anywhere."

The most important concept between the two is Security Isolation can be achieved by both the Docker Containers and Virtual Machines equally.

Docker Image:

A Docker Image is a lightweight, stand-alone, executable package that includes everything needed to run a piece of software, including the code, a runtime, libraries, environment variables, and configure files.

Use Cases of Docker:

- Simplifying Configuration
- Code Pipeline Management
- Developer Productivity
- App Isolation
- Server Consolidation
- Debugging Capabilities
- Multi-tenancy
- Continuous Integration and Deployment

What can Docker do?

- Development team on same page and environment
- Development, QA, and Production sync up
- Get Higher density and run more workloads
- Desktop to cloud easy deployment and scaling
- Build and deploy fast and better

Who is using Docker and Why?

- GILT for Distributed applications to scale
- Yelp for Continuous Integration
- Bleacher Report for Application Recycle
- Spotify for Continuous Delivery
- Baidu, Inc. for Platform-as-a-Service(PaaS)
- New Relic for Distributed Application Composition
- Rackspace for Continuous Integration
- Yandex for Platform-as-a-Service(PaaS)
- Microsoft Azure for Continuous Integration
- Cambridge Health Care for Continuous Delivery
- eBay for Easy Application Development
- Amazon Web Services for Continuous Integration
- Google Cloud Platform for Continuous Integration
- Apache Cassandra for Platform-as-a-Service(PaaS)
- MongoDB for Platform-as-a-Service(PaaS)

Installation of Docker:

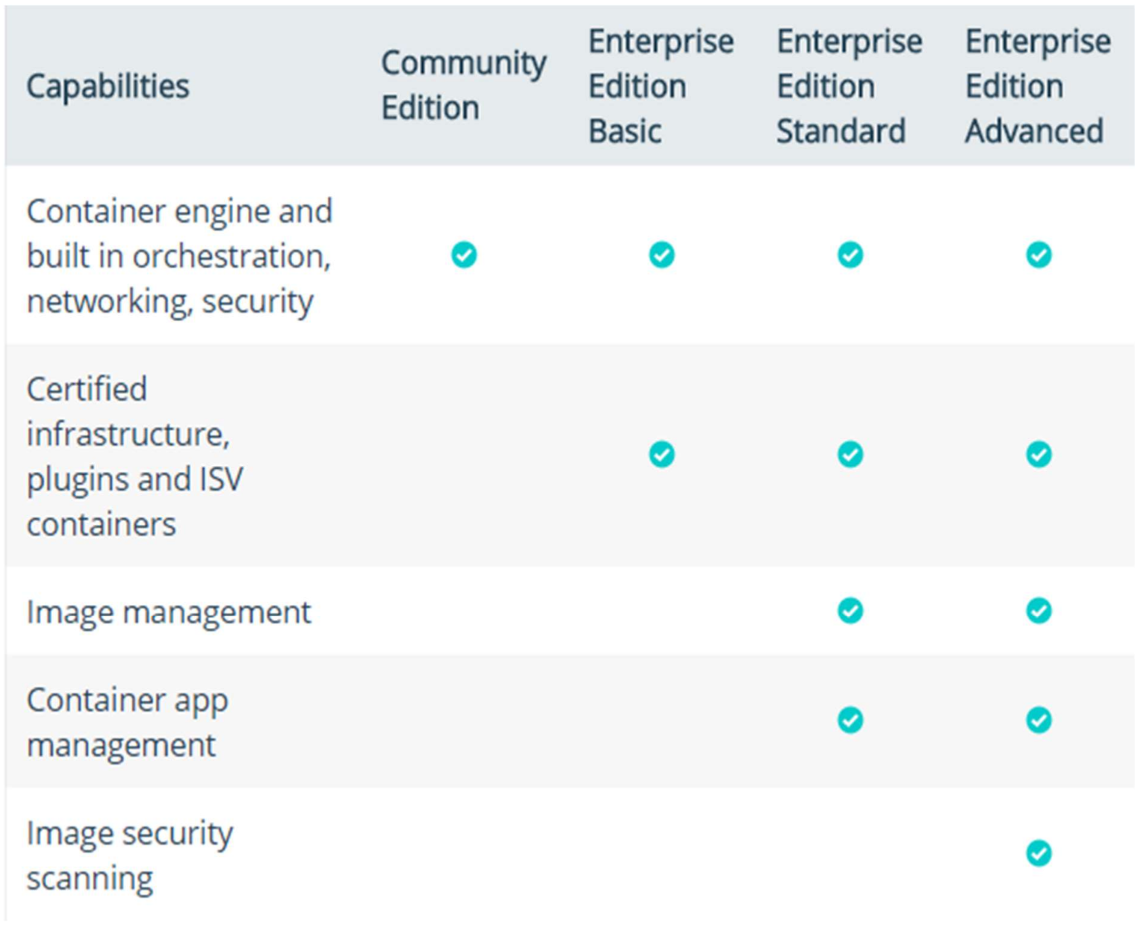
Docker is available in two editions: **Community Edition (CE)** and **Enterprise Edition (EE)**.

Docker Community Edition (CE) is ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. Docker CE has two update channels, **stable** and **edge**:

- **Stable** gives you reliable updates every quarter
- **Edge** gives you new features every month

Docker Enterprise Edition (EE) is designed for enterprise development and IT teams who build, ship, and run business critical applications in production at scale.

The following figure illustrates Docker Editions and their Capabilities:

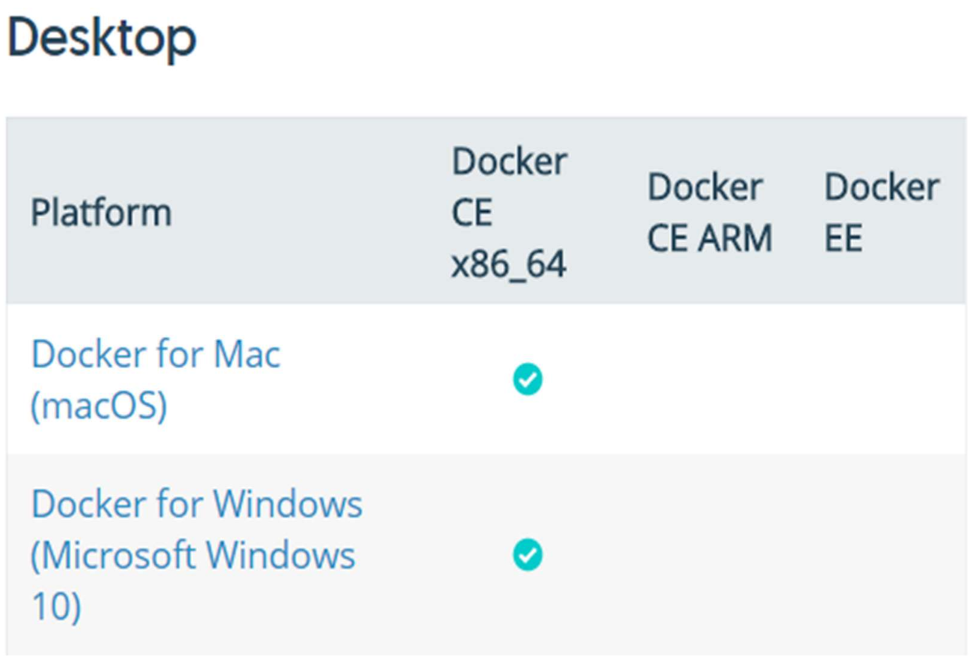


Capabilities	Community Edition	Enterprise Edition Basic	Enterprise Edition Standard	Enterprise Edition Advanced
Container engine and built in orchestration, networking, security	✓	✓	✓	✓
Certified infrastructure, plugins and ISV containers		✓	✓	✓
Image management			✓	✓
Container app management			✓	✓
Image security scanning				✓

Fig: Docker Editions and their Capabilities

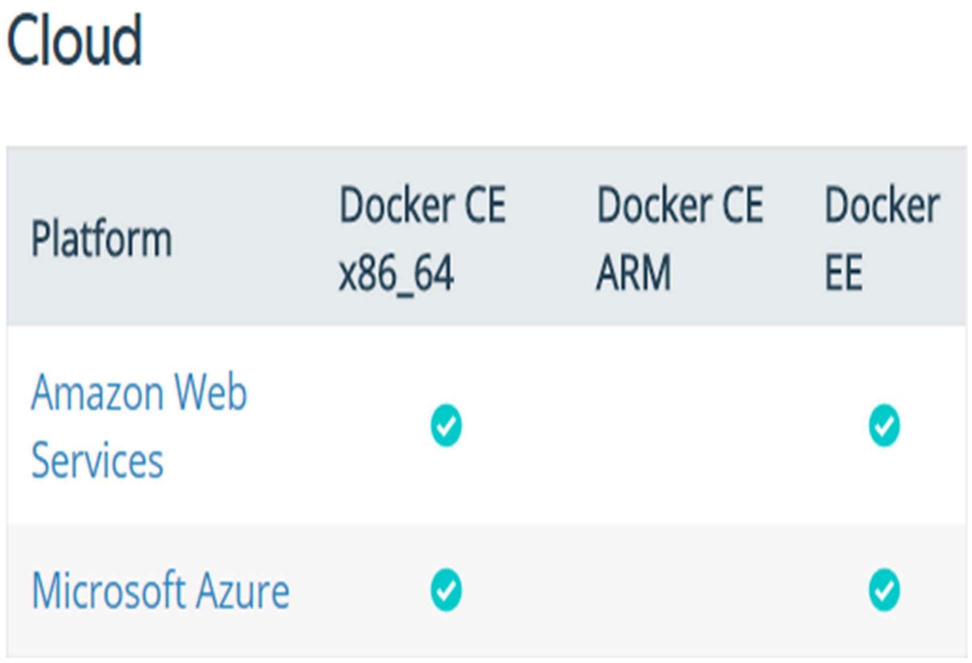
Supported Platforms:

Docker CE and EE are available on multiple platforms, on cloud and on-premises. Use the following tables to choose the best installation path for you.



Desktop

Platform	Docker CE x86_64	Docker CE ARM	Docker EE
Docker for Mac (macOS)	✓		
Docker for Windows (Microsoft Windows 10)	✓		



Cloud

Platform	Docker CE x86_64	Docker CE ARM	Docker EE
Amazon Web Services	✓		✓
Microsoft Azure	✓		✓

Server

Platform	Docker CE x86_64	Docker CE ARM	Docker EE
CentOS	✓		✓
Debian	✓	✓	
Fedora	✓		
Microsoft Windows Server 2016			✓
Oracle Linux			✓
Red Hat Enterprise Linux			✓
SUSE Linux Enterprise Server			✓
Ubuntu	✓	✓	✓

Time-based release schedule

Starting with Docker 17.03, Docker uses a time-based release schedule, outlined below.

Month	Docker CE Edge	Docker CE Stable	Docker EE
January	✓		
February	✓		
March	✓[1]	✓	✓
April	✓		
May	✓		
June	✓[1]	✓	✓
July	✓		
August	✓		
September	✓[1]	✓	✓
October	✓		
November	✓		
December	✓[1]	✓	✓

`1` : On Linux distributions, these releases will only appear in the `stable` channels, not the `edge` channels. For that reason, on Linux distributions, you need to enable both channels.

OS Requirements for Docker Installation:

To install Docker, you need the 64-bit version of one of these Ubuntu versions:

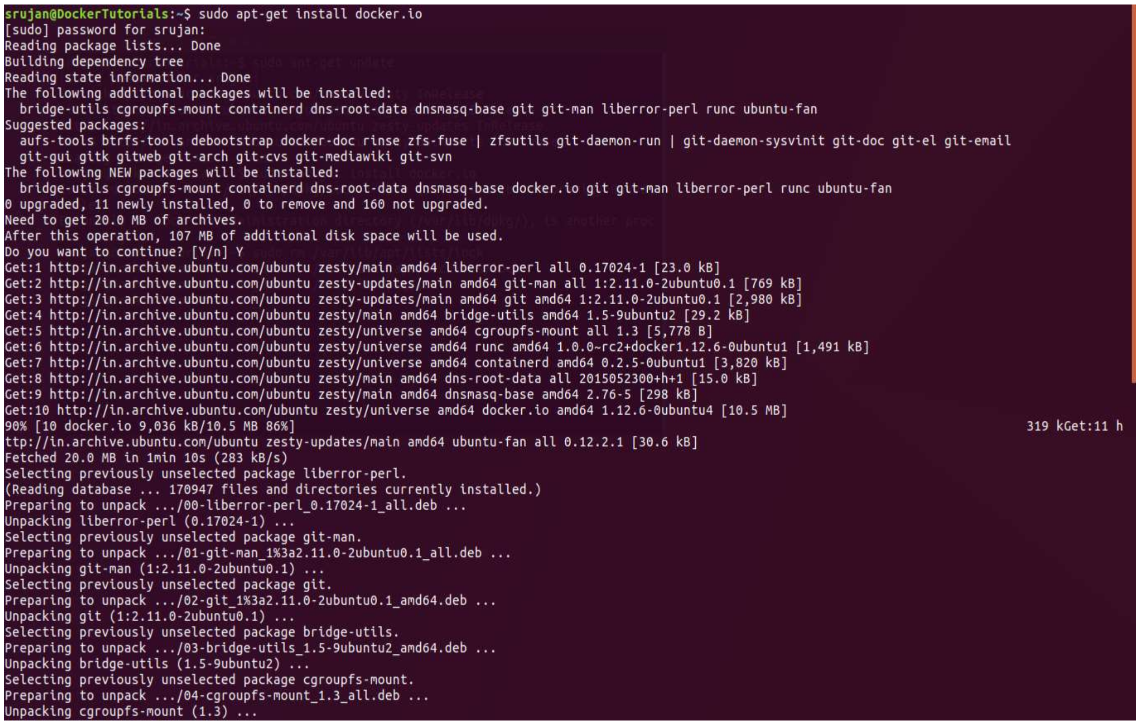
- Zesty 17.04
- Yakkety 16.10
- Xenial 16.04(LTS)
- Trusty 14.04(LTS)

Docker CE is supported on Ubuntu on x86_64, armhf, and s390x (IBM z Systems) architectures.

- **s390x limitations**: System Z is only supported on Ubuntu Xenial, Yakkety, and Zesty.

To install Docker, open the terminal and type the following command:

- $ **sudo apt-get install docker.io**



```
srujan@DockerTutorials:~$ sudo apt-get install docker.io
[sudo] password for srujan:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  bridge-utils cgroupfs-mount containerd dns-root-data dnsmasq-base git git-man liberror-perl runc ubuntu-fan
Suggested packages:
  aufs-tools btrfs-tools debootstrap docker-doc rinse zfs-fuse | zfsutils git-daemon-run | git-daemon-sysvinit git-doc git-el git-email
  git-gui gitk gitweb git-arch git-cvs git-mediawiki git-svn
The following NEW packages will be installed:
  bridge-utils cgroupfs-mount containerd dns-root-data dnsmasq-base docker.io git git-man liberror-perl runc ubuntu-fan
0 upgraded, 11 newly installed, 0 to remove and 160 not upgraded.
Need to get 20.0 MB of archives.
After this operation, 107 MB of additional disk space will be used.
Do you want to continue? [Y/n] Y
Get:1 http://in.archive.ubuntu.com/ubuntu zesty/main amd64 liberror-perl all 0.17024-1 [23.0 kB]
Get:2 http://in.archive.ubuntu.com/ubuntu zesty-updates/main amd64 git-man all 1:2.11.0-2ubuntu0.1 [769 kB]
Get:3 http://in.archive.ubuntu.com/ubuntu zesty-updates/main amd64 git amd64 1:2.11.0-2ubuntu0.1 [2,980 kB]
Get:4 http://in.archive.ubuntu.com/ubuntu zesty/main amd64 bridge-utils amd64 1.5-9ubuntu2 [29.2 kB]
Get:5 http://in.archive.ubuntu.com/ubuntu zesty/universe amd64 cgroupfs-mount all 1.3 [5,778 B]
Get:6 http://in.archive.ubuntu.com/ubuntu zesty/universe amd64 runc amd64 1.0.0~rc2+docker1.12.6-0ubuntu1 [1,491 kB]
Get:7 http://in.archive.ubuntu.com/ubuntu zesty/universe amd64 containerd amd64 0.2.5-0ubuntu1 [3,820 kB]
Get:8 http://in.archive.ubuntu.com/ubuntu zesty/main amd64 dns-root-data all 2015052300+h+1 [15.0 kB]
Get:9 http://in.archive.ubuntu.com/ubuntu zesty/main amd64 dnsmasq-base amd64 2.76-5 [298 kB]
Get:10 http://in.archive.ubuntu.com/ubuntu zesty/universe amd64 docker.io amd64 1.12.6-0ubuntu4 [10.5 MB]
90% [10 docker.io 9,036 kB/10.5 MB 86%]                                                               319 kGet:11 h
ttp://in.archive.ubuntu.com/ubuntu zesty-updates/main amd64 ubuntu-fan all 0.12.2.1 [30.6 kB]
Fetched 20.0 MB in 1min 10s (283 kB/s)
Selecting previously unselected package liberror-perl.
(Reading database ... 170947 files and directories currently installed.)
Preparing to unpack .../00-liberror-perl_0.17024-1_all.deb ...
Unpacking liberror-perl (0.17024-1) ...
Selecting previously unselected package git-man.
Preparing to unpack .../01-git-man_1%3a2.11.0-2ubuntu0.1_all.deb ...
Unpacking git-man (1:2.11.0-2ubuntu0.1) ...
Selecting previously unselected package git.
Preparing to unpack .../02-git_1%3a2.11.0-2ubuntu0.1_amd64.deb ...
Unpacking git (1:2.11.0-2ubuntu0.1) ...
Selecting previously unselected package bridge-utils.
Preparing to unpack .../03-bridge-utils_1.5-9ubuntu2_amd64.deb ...
Unpacking bridge-utils (1.5-9ubuntu2) ...
Selecting previously unselected package cgroupfs-mount.
Preparing to unpack .../04-cgroupfs-mount_1.3_all.deb ...
Unpacking cgroupfs-mount (1.3) ...
```

Fig: Docker Installation on Ubuntu

To verify the installation, type the following command in the terminal:

- **$ sudo docker**

To check the installed version of Docker, type the following command in the terminal:

- **$ sudo docker version**

Container with Dockerfile:

Dockerfile will define what goes on in the environment inside your container. Access to resources like networking interfaces and disk drives is virtualized inside this environment, which is isolated from the rest of your system, so you have to map ports to the outside world, and be specific about what files you want to "copy in" to that environment. However, after doing that, you can expect that the build of your app defined in this **Dockerfile** will behave exactly the same wherever it runs.

Creating and Deleting a Docker Container:

To create a Docker Container, type the following command in the terminal:

- **$ sudo docker run –t -i –name containername ubuntu:17.04 /bin/bash**

To delete a container, type the following command in the terminal:

- **$ sudo docker rm containername**

Socket:

A **socket** is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can now communicate by writing to or reading from their sockets.

The **java.net** package in the Java platform provides a class, Socket, that implements one side of a two-way connection between your Java program and another program on the network. The Socket class sits on top of a platform-dependent implementation, hiding the details of any particular system from your Java program. By using the **java.net.Socket** class instead of relying on native code, your Java programs can communicate over the network in a platform-independent fashion.

Additionally, **java.net** includes the **ServerSocket** class, which implements a socket that servers can use to listen for and accept connections to clients.

InetAddress:

InetAddress is a class encapsulating both the IP address and domain name. The **java.net.InetAddress** class provides methods to get the IP of any host name for example www.google.com, www.facebook.com etc. This class can be used for getting the IP address of systems by providing the name and vice versa. This is done by some functions in the class such as getaddress ( ), getbyname ( ). Instances of this class are used together with **UDP Datagram Sockets**, normal Socket's and **Server Sockets**.

InetAddress has no public constructor, so you must obtain instances via a set of static class methods.

Port:

On computer and telecommunication devices, a **port** is generally a specific place for being physically connected to some other device, usually with a socket and plug of some kind. Typically, a personal computer is provided with one or more **serial** ports and usually one **parallel** port. The serial port supports sequential, one bit-at-a-time transmission to peripheral devices such as scanners and the parallel port supports multiple-bit-at-a-time transmission to devices such as printers.

In programming, a **port** is a "logical connection place" and specifically, using the Internet's protocol, **TCP/IP**, the way a client program specifies a particular server program on a computer in a network. Higher-level applications that use TCP/IP such as the Web protocol, Hypertext Transfer Protocol, have ports with preassigned numbers. These are known as "well-known ports" that have been assigned by the Internet Assigned Numbers Authority (**IANA**). Other application processes are given port numbers dynamically for each connection. When a service (**server** program) initially is started, it is said to **bind** to its designated port number. As any **client** program wants to use that server, it also must request to bind to the designated port number.

Port numbers are from 0 to 65535. Ports 0 to 1024 are reserved for use by certain privileged services. For the HTTP service, port 80 is defined as a default and it does not have to be specified in the Uniform Resource Locator (**URL**).

OutputStream:

Java application uses an output stream to write data to a destination, it may be a file, an array, peripheral device or socket.

The OutputStream class is an abstract superclass that provides a minimal programming interface and a partial implementation of output streams. OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when you create it. You can explicitly close an output stream with the close method, or let it be closed implicitly when the OutputStream is garbage collected, which occurs when the object is no longer referenced.

The **java.io** package contains several subclasses of OutputStream that implement specific output functions. For example, FileOutputStream is the output stream that operate on files on the native file system.

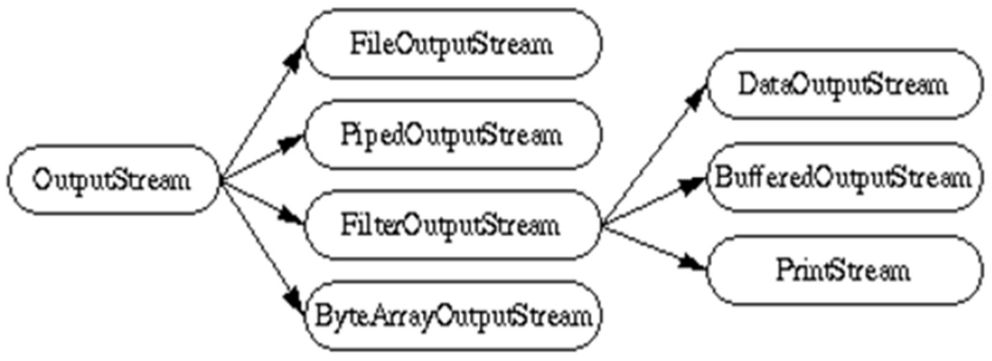


Fig: OutputStream Hierarchy

ServerSocket:

The **java.net.ServerSocket** class represents a server socket. It is constructed on a particular port.

A special type of socket, the server socket, is used to provide TCP services. Client sockets bind to any free port on the local machine, and connect to a specific server port and host. The difference with server sockets is that they bind to a specific port on the local machine, so that remote clients may locate a service. Client socket connections will connect to only one machine, whereas server sockets are capable of fulfilling the requests of multiple clients.

Once a server socket is created, it will be bound to a local port and ready to accept incoming connections. When clients attempt to connect, they are placed into a queue. Once all free space in the queue is exhausted, further clients will be refused.

InputStream:

Java application uses an input stream to read data from a source, it may be a file, an array, peripheral device or socket.

The InputStream class is an abstract superclass that provides a minimal programming interface and a partial implementation of input streams. The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, and finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when you create it. You can explicitly close a stream with the close method, or let it be closed implicitly when the InputStream is garbage collected. Remember that garbage collection occurs when the object is no longer referenced.

The **java.io** package contains several subclasses of InputStream that implement specific input functions. For example, FileInputStream is the input stream that operate on files on the native file system.

InputStream is subclass of Reader class. It converts bytes to characters. InputStream class is an abstract class. It is the super class of all classes representing an input stream of bytes.

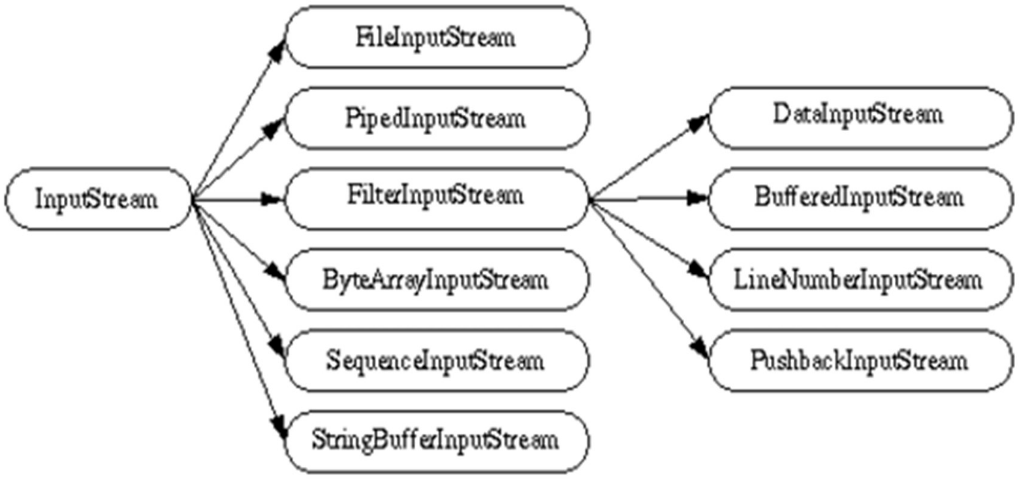


Fig: InputStream Hierarchy

Dockerfile to Docker Image to Docker Container:

A Docker **image** is a read-only template used to create and launch a Docker **container.**

A Docker image can be plain and simple Linux distro (known as **base image**) or fully-configured enterprise software install, ready to run. It all depends on the Dockerfile.

A **Dockerfile** is a text file with instructions written in a format understood by the Docker **daemon** (Docker daemon is one of the Docker components, that does the heavy-duty tasks of building, running, and distributing the Docker images, after getting commands from the Docker **client**). To create an image, you need to write your own Dockerfile. But don't worry, there are only a handful of Dockerfile configuration options (called **instructions**), and they are very easy to understand.

After you have your Dockerfile ready, you can execute **docker build** from command prompt to create the corresponding docker image. Then run **docker run <image-name>** to create a container out of image and execute it.

Commands used in Docker:

The following are the different commands which are used in Docker

i. docker attach: Attach local standard input, output, and error streams to a running container

ii. docker build: Build an image from Dockerfile

iii. docker checkpoint: Manage checkpoints

iv. docker commit: create a new image from a container's change

v. docker config: Manage Docker configs

vi. docker container: Manage Containers

vii. docker cp: Copy files/folders between a container and local file system

viii. docker create: Create a new container

ix. docker deploy: Deploy a new stack or update an existing stack

x. docker diff: Inspect changes to files or directories on a container file system

xi. docker events: Get real time events from server

xii. docker exec: Run a command in a running container

xiii. docker export: Export a container's file system as a tar or archive

xiv. docker history: Shows the history of an image

xv. docker image: Manage images

xvi. docker images: List images

xvii. docker import: Import the contents from a tarball to create a filesystem image

xviii. docker info: Display system-wide information

xix. docker inspect: Return low-level information on docker objects

xx. docker kill: Kill one or more running containers

xxi. docker load: Load an image for a tar archive or STDIN

xxii. docker login: Log in to a Docker Registry

xxiii. docker logout: Log out from a Docker Registry

xxiv. docker logs: fetch the logs of a container

xxv. docker network: Manage networks

xxvi. docker node: Manage Swarm nodes

xxvii. docker pause: Pause all processes within one or more containers

xxviii. docker plugin: Manage plugins

xxix. docker port: list port mapping or a specific mapping for the container

xxx. docker ps: List containers

xxxi. docker pull: Pull an image or a repository from a registry

xxxii. docker push: Push an image or a repository to a registry

xxxiii. docker rename: Rename a container

xxxiv. docker restart: Restart one or more containers

xxxv. docker rm: Remove one or more containers

xxxvi. docker rmi: Remove one or more images

xxxvii. docker run: Run a command in a new container

xxxviii. docker save: Save one or more images to a tar archive (streamed to STDOUT by default)

xxxix. docker search: Search the Docker Hub for images

xl. docker secret: Manage Docker secrets

xli. docker service: Manage Services

xlii. docker stack: Manage Docker stacks

xliii. docker start: Start one more stopped Containers

xliv. docker stats: Display a live stream of container(s) resource usage statistics xlv. docker stop: Stop one or more running containers

xlvi. docker swarm: Manage Swarm

xlvii. docker system: Manage Docker

xlviii. docker tag: Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE

xlix. docker top: Display the running processes of the container

l. docker unpause: Unpause all processes within one or more containers

li. docker update: Update configuration of one or more containers

lii. docker version: Show the Docker Version information

liii. docker volume: Manage volumes

liv. docker wait: Block until one or more containers stop, then print their exit codes

Dockerfile Instructions:

1) **FROM:** This instruction is used to set the base image for subsequent instructions. It is mandatory to set this in the first line of a Dockerfile. You can use it any number of times though.

   a. **Example:** FROM ubuntu

2) **MAINTAINER:** This is a non-executable instruction used to indicate the author of the Dockerfile.

   a. **Example:** MAINTAINER <name>

3) **RUN:** This instruction lets you execute a command on top of an existing layer and create a new layer with the results of command execution. For example, if there is a pre-condition to install PHP before running an application, you can run appropriate commands to install PHP on top of base image (say Ubuntu) like this:

   a. **Example:** RUN apt-get update install php5

4) **<u>CMD:</u>** The major difference between CMD and RUN is that CMD doesn't execute anything during the build time. It just specifies the intended command for the image. Whereas RUN actually executes the command during build time.

    a. **<u>NOTE:</u>** there can be only one **CMD** instruction in a Dockerfile, if you add more, only the last one takes effect.

    b. **<u>Example:</u>** CMD "echo" "Hello World!"

5) **<u>LABEL:</u>** You can assign metadata in the form of key-value pairs to the image using this instruction. It is important to notice that each LABEL instruction creates a new layer in the image, so it is best to use as few LABEL instructions as possible.

    a. **<u>Example:</u>** LABEL version = "1.0" description = "This is a sample desc"

6) **<u>EXPOSE:</u>** While running your service in the container you may want your container to listen on specified ports. The EXPOSE instruction helps you do this.

    a. **<u>Example:</u>** EXPOSE 8080

7) **<u>ENV:</u>** This instruction can be used to set the environment variables in the container.

    a. **<u>Example:</u>** ENV var_home = " /var/etc"

8) **<u>COPY:</u>** This instruction is used to copy files and directories from a specified source to a destination (in the file system of the container).

    a. **<u>Example:</u>** COPY preconditions.txt /usr/temp

9) **<u>ADD:</u>** This instruction is similar to the **COPY** instruction with few added features like remote URL support in the source field and local-only tar extraction. But if you don't need an extra features, it is suggested to use **COPY** as it is more readable.

    a. **<u>Example:</u>** ADD http: //www.site.com/downloads/sample.tar.xz /usr/src

10) **<u>ENTRYPOINT:</u>** You can use this instruction to set the primary command for the image. For example, if you have installed only one application in your image and want it to run whenever the image is executed, **ENTRYPOINT** is the instruction for you.

a. **Note:** Arguments are optional, and you can pass them during the runtime with something like **docker run <image-name>**

b. Also, all the elements specified using **CMD**, will be overridden, except the arguments. They will be passed to the command specified in **ENTRYPOINT**

c. **Example:**

```
CMD "Hello World!"

ENTRYPOINT echo
```

11) **VOLUME:** You can use the VOLUME instruction to enable access to a location on the host system from a container. Just pass the path of the location to be accessed.

    a. **Example:** VOLUME /data

12) **USER:** This is used to set the UID (or username) to use when running the image.

    a. **Example:** USER daemon

13) **ONBUILD:** This instruction adds a trigger instruction to be executed when the image is used as the base for some other image. It behaves as if a RUN instruction is inserted immediately after the FROM of the downstream Dockerfile. This is typically helpful in cases where you need a static base image with a dynamic config value that changes whenever a new image has to be built (on top of the base image).

    a. **Example:** ONBUILD RUN rm –rf /usr/temp

14) **WORKDIR:** This is used to set the currently active directory for other instructions such as RUN, CMD, ENTRYPOINT, COPY and ADD

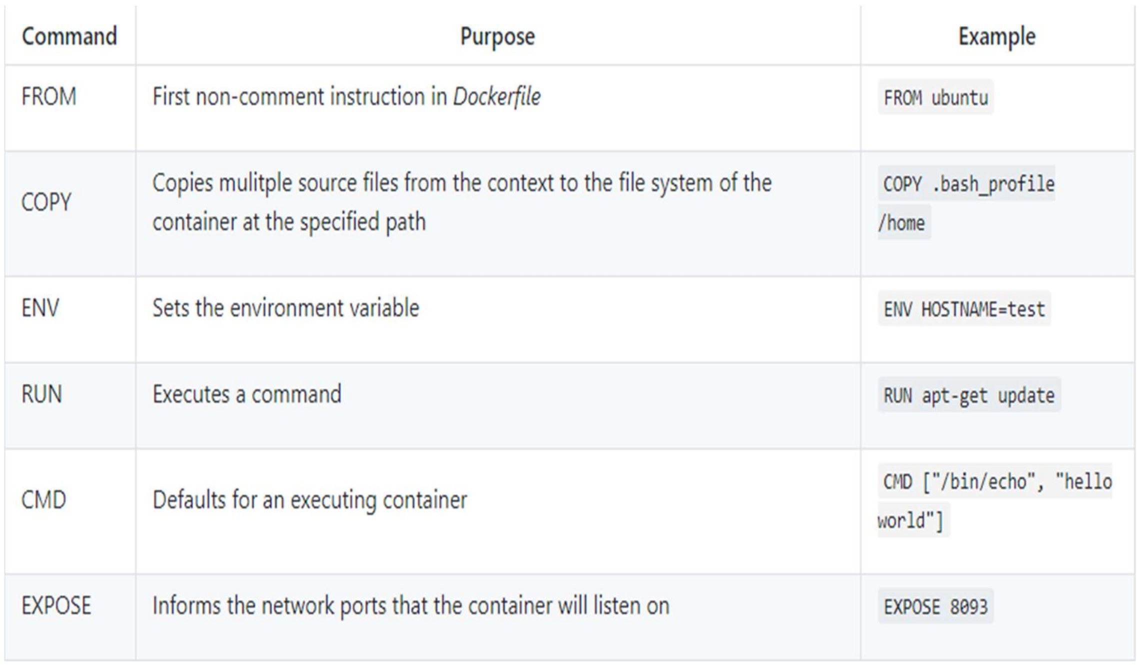
    a. **Note:** If relative path is provided, the next WORKDIR instruction will take it as relative to the path of previous WORKDIR instruction

    b. **Example:**

```
WORKDIR /user

WORKDIR home

RUN pwd
```



Command	Purpose	Example
FROM	First non-comment instruction in *Dockerfile*	`FROM ubuntu`
COPY	Copies mulitple source files from the context to the file system of the container at the specified path	`COPY .bash_profile /home`
ENV	Sets the environment variable	`ENV HOSTNAME=test`
RUN	Executes a command	`RUN apt-get update`
CMD	Defaults for an executing container	`CMD ["/bin/echo", "hello world"]`
EXPOSE	Informs the network ports that the container will listen on	`EXPOSE 8093`

Fig: Common Commands for Dockerfile

Docker Client:

The client communicates with the Docker Host and lets you work with images and containers. Check if your client is working using the following command:

- **$ sudo docker –v**

The complete set of commands can be seen using the following command:

- **$ sudo docker –help**

Inspect:

Low-level information about containers and images can be collected using the Docker inspect command. Information, including the following, can be gathered using the inspect command:

- IP addresses of an instance
- List of port bindings

- Search for specific port mapping
- Collect Configuration Details

The syntax for inspect is as follows:

- **$ sudo docker inspect container/image**

Dockerfile Syntax:

Dockerfile use simple, clean, and clear syntax which makes them strikingly easy to create and use. They are designed to be self-explanatory, especially because they allow commenting just like a good and properly written application source-code.

Dockerfile syntax consists of two kind of main line blocks: comments and commands + arguments.

- **# Line blocks used for commenting**
- **command argument argument ..**

A simple Example for Dockerfile Syntax is as follows:

- **# Print "Hello docker!"**
- **RUN echo "Hello docker!"**

Docker Compose:

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application's services. Then, using a single command, you create and start all the services from your configuration.

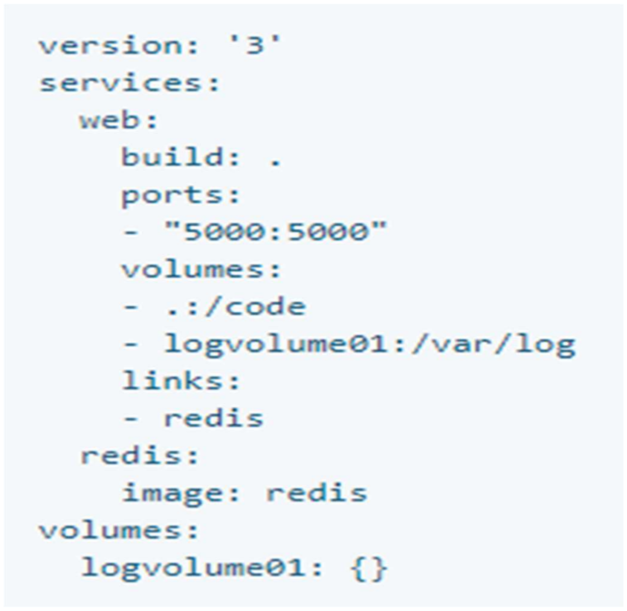
Compose is great for development, testing, and staging environments, as well as CI workflows.

Using Compose is basically a three-step process:

1) Define your app's environment with a Dockerfile so it can be reproduced anywhere.

2) Define the services that make up your app in **docker-compose.yml** so they can be run together in an isolated environment.
3) Lastly, run **docker-compose up** and Compose will start and run your entire app.

A **docker-compose.yml** looks like the following:



```
version: '3'
services:
  web:
    build: .
    ports:
    - "5000:5000"
    volumes:
    - .:/code
    - logvolume01:/var/log
    links:
    - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Compose has commands for managing the whole lifecycle of your application

a) Start, stop and rebuild services
b) View the status of running services
c) Stream the log output of running services
d) Run a one-off command on a service

Environment Variables for Docker:

- **DOCKER_API_VERSION:** The API version to use
- **DOCKER_CONFIG:** The location of your client configuration files.
- **DOCKER_CERT_PATH:** The location of your authentication keys.
- **DOCKER_DRIVER:** The graph driver to use.
- **DOCKER_HOST:** The Daemon socket to connect to.
- **DOCKER_NORWAN_KERNEL_VERSION:** Prevent warnings that your Linux kernel is unsuitable for Docker.

- **DOCKER_RAMDISK:** If set this will disable ‘pivot_root’.
- **DOCKER_TLS_VERIFY:** When set Docker uses TLS and verifies the remote.
- **DOCKER_CONTENT_TRUST:** When set Docker uses notary to sign and verify images. Equates to **–disable-content-trust = false** for build, create, pull, push, run.
- **DOCKER_CONTENT_TRUST_SERVER:** The URL of the Notary server to use. This defaults to the same URL as the registry.
- **DOCKER_TMPDIR:** Location for temporary Docker files.
- **DOCKER_HIDE_LEGACY_COMMANDS:** When set, Docker hides “legacy” top-level commands (such as **docker rm** and **docker pull**) in **docker help** output, and only **Management Commands** per object-type (e.g., **docker container**) are printed. This may become the default in a future release, at which point this environment-variable is removed.

Client Server Communication using Docker:

Here, we need two java files i.e., Server.java and Client.java which are in the same container. After this, we need to create Dockerfile for the server and client for building the containers and the images.

Server.java

```
import java.io.*;

import java.net.*;

public class Server{

        private static Socket socket;

        public static void main(String args[]){

        try{

                int port=25000;

                ServerSocket serverSocket=new ServerSocket(port);

                System.out.println("server started and listening to the port 25000");
        while (true) {
            socket =serverSocket.accept ();
            InputStream is=socket.getInputStream ();
            InputStreamReader isr=new InputStreamReader (is);
            BufferedReader br=new BufferedReader (isr);
            String num=br.readLine ();
            System.out.println ("msg received from the client is:"+ num);
            String retMsg="Hello from server";
            String sendretMsg=retMsg+"\n";
            OutputStream os=socket.getOutputStream ();
            OutputStreamWriter osw=new OutputStreamWriter (os);
            BufferedWriter bw =new BufferedWriter (osw);
            bw.write (sendretMsg);
            bw.flush ();
            System.out.println ("msg sent to the client:"+ retMsg);
            System.out.println ("\n process complete\n");
        }
    }
    catch (Exception e) {
        e.printStackTrace ();
    }
    finally {
        try {
            socket.close ();
```

```
                }
                catch (Exception e)
                {
                        e.printStackTrace ();
                }
        }
        }
}
```

Client.java:

```
import java.net.*;
import java.io.*;
public class Client {
        private static Socket socket;
        public static void main (String args []) {
        try {
                String host="localhost";
                int port=25000;
                //InetAddress address=InetAddress.getByName (host);
                socket = new Socket (address, port);
                OutputStream os=socket.getOutputStream ();
                OutputStreamWriter osw=new OutputStreamWriter (os);
                BufferedWriter bw =new BufferedWriter (osw);
                String msg="HEllo from client";
```

```
                    String sendMsg=msg+"\n";

                    bw.write (sendMsg);

                    bw.flush ();

                    System.out.println ("msg sent to the server:"+sendMsg);

                    InputStream is=socket.getInputStream ();

                    InputStreamReader isr=new InputStreamReader (is);

                    BufferedReader br=new BufferedReader (isr);

                    String mesg=br.readLine ();

                    System.out.println ("msg received from the server:"+mesg);

            }

            catch (Exception exception) {

                    exception.printStackTrace();

            }

            finally{

                    try{

                            socket.close();

                    }

                    catch(Exception e)

                    {

                            e.printStackTrace();

                    }

            }

            }

    }
```

Dockerfile for Server (Server Image):

```
FROM java:8

COPY Server.java /

RUN javac Server.java

EXPOSE 25000

ENTRYPOINT ["java"]

CMD ["Server"]
```

Dockerfile for Client (Client Image):

```
FROM java:8

COPY Client.java /

RUN javac Client.java

EXPOSE 25000

ENTRYPOINT ["java"]

CMD ["Client"]
```

Now, you have to build the Dockerfile of Server and the Client. After this step, you have to run the images using the Docker Containers obtained from the Dockerfile.

Containers are built using the following command:

- **$ sudo docker build –t serverimage .**

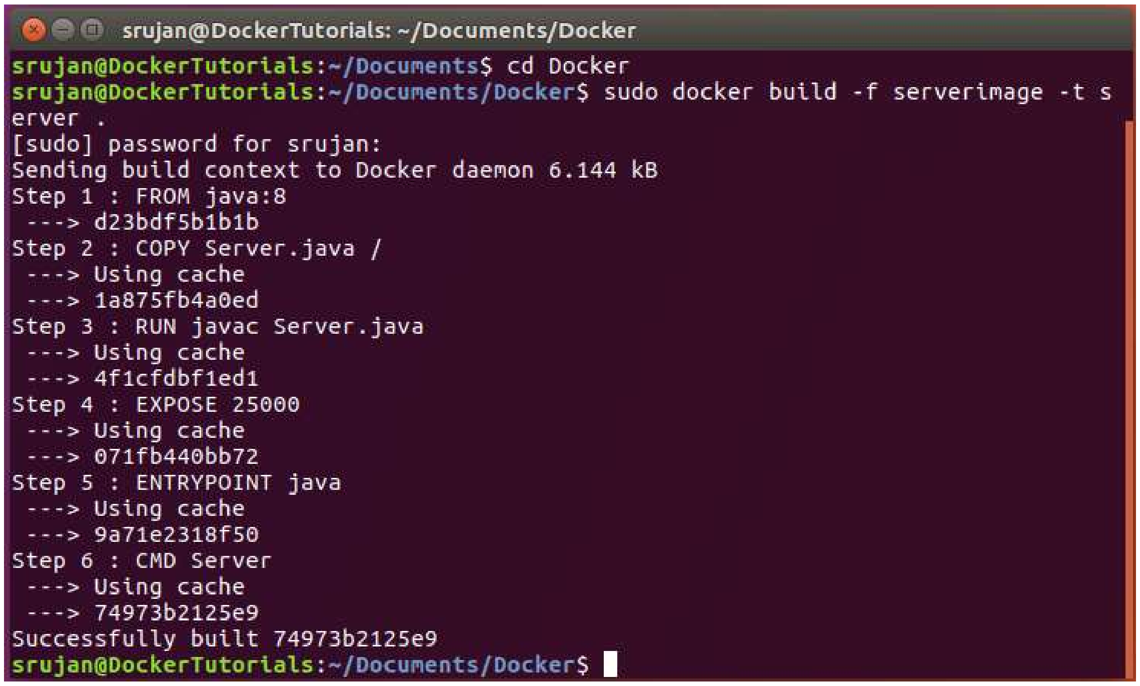
Now, Run the images using the following command:

- **$ sudo docker run <IMAGE_NAME>**

Build Docker Container for Server Image:

Now, build the Dockerfile of Server using the following command:

- **$ sudo docker build –f serverimage –t server .**

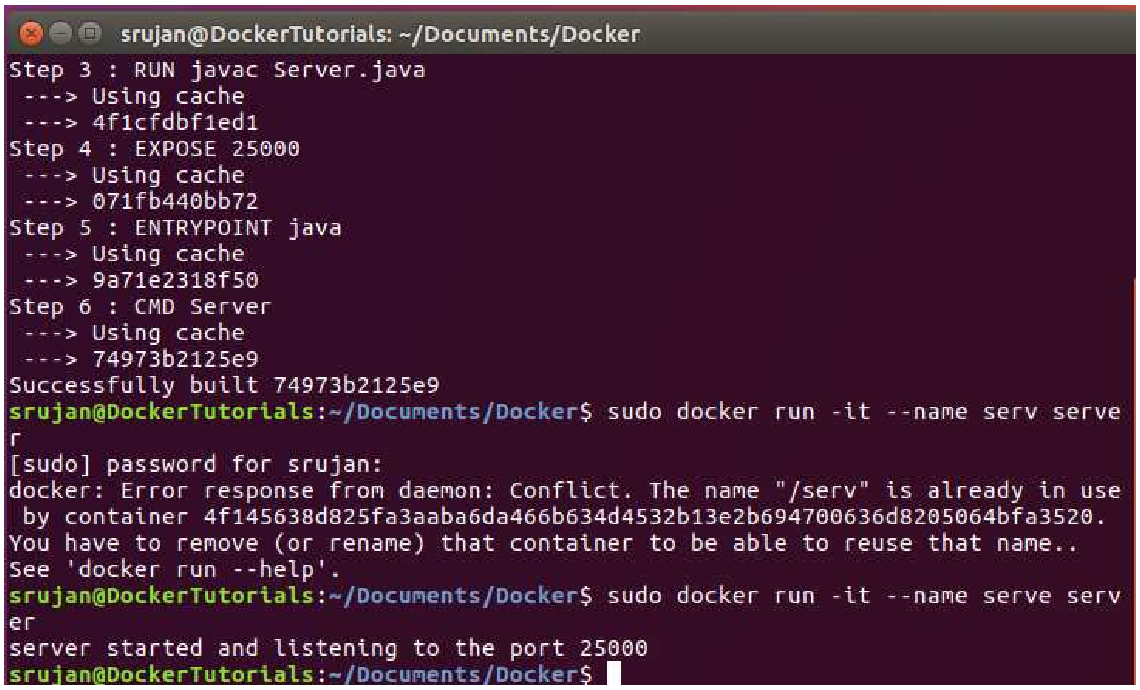


```
srujan@DockerTutorials: ~/Documents/Docker
srujan@DockerTutorials:~/Documents$ cd Docker
srujan@DockerTutorials:~/Documents/Docker$ sudo docker build -f serverimage -t s
erver .
[sudo] password for srujan:
Sending build context to Docker daemon 6.144 kB
Step 1 : FROM java:8
 ---> d23bdf5b1b1b
Step 2 : COPY Server.java /
 ---> Using cache
 ---> 1a875fb4a0ed
Step 3 : RUN javac Server.java
 ---> Using cache
 ---> 4f1cfdbf1ed1
Step 4 : EXPOSE 25000
 ---> Using cache
 ---> 071fb440bb72
Step 5 : ENTRYPOINT java
 ---> Using cache
 ---> 9a71e2318f50
Step 6 : CMD Server
 ---> Using cache
 ---> 74973b2125e9
Successfully built 74973b2125e9
srujan@DockerTutorials:~/Documents/Docker$
```

Running Containers for Server:

Now, run the images for server using the following command:

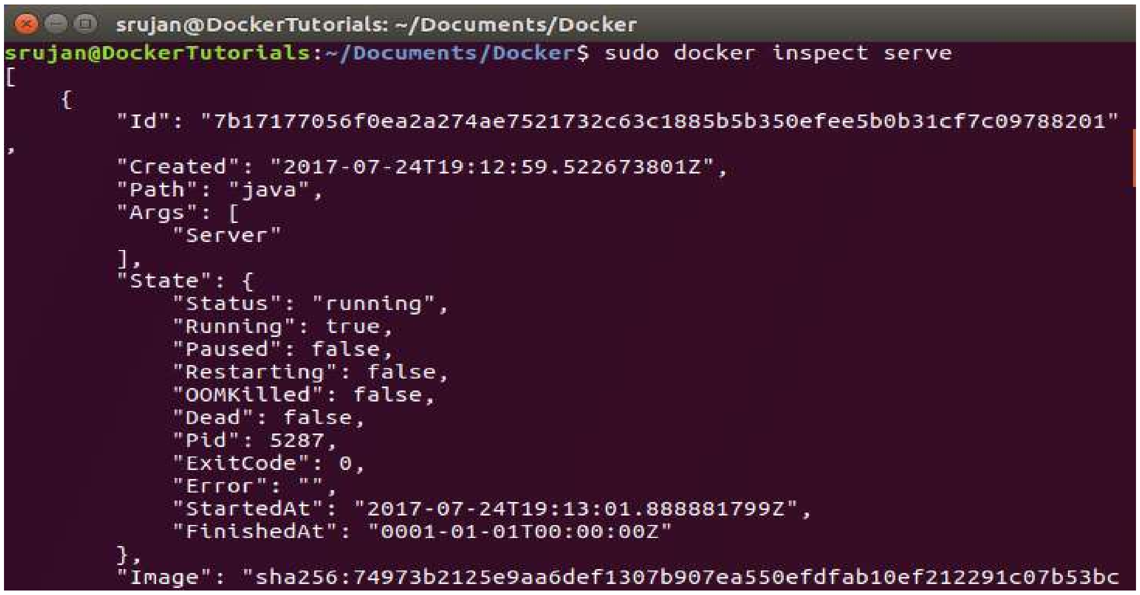
- **$ sudo docker run –it –name serve server**



```
srujan@DockerTutorials: ~/Documents/Docker
Step 3 : RUN javac Server.java
 ---> Using cache
 ---> 4f1cfdbf1ed1
Step 4 : EXPOSE 25000
 ---> Using cache
 ---> 071fb440bb72
Step 5 : ENTRYPOINT java
 ---> Using cache
 ---> 9a71e2318f50
Step 6 : CMD Server
 ---> Using cache
 ---> 74973b2125e9
Successfully built 74973b2125e9
srujan@DockerTutorials:~/Documents/Docker$ sudo docker run -it --name serv serve
r
[sudo] password for srujan:
docker: Error response from daemon: Conflict. The name "/serv" is already in use
 by container 4f145638d825fa3aaba6da466b634d4532b13e2b694700636d8205064bfa3520.
You have to remove (or rename) that container to be able to reuse that name..
See 'docker run --help'.
srujan@DockerTutorials:~/Documents/Docker$ sudo docker run -it --name serve serv
er
server started and listening to the port 25000
srujan@DockerTutorials:~/Documents/Docker$
```

Now, type the following command to get the IP address of server image as follows:

- **$ sudo docker inspect serve**

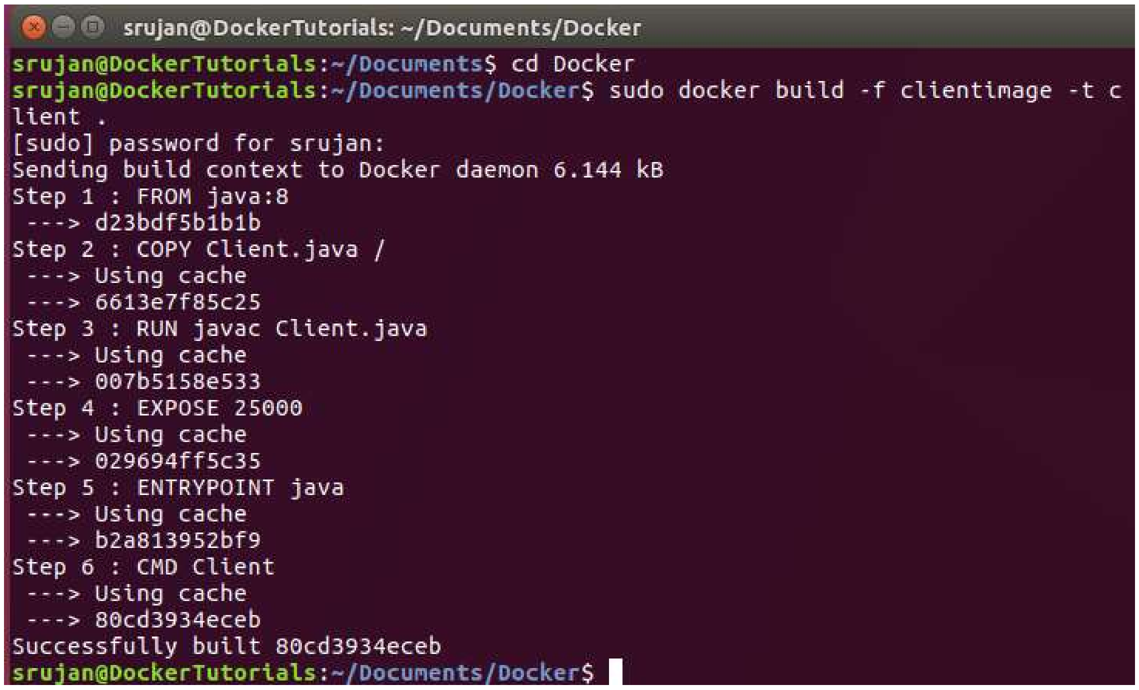


```
srujan@DockerTutorials: ~/Documents/Docker
srujan@DockerTutorials:~/Documents/Docker$ sudo docker inspect serve
[
    {
        "Id": "7b17177056f0ea2a274ae7521732c63c1885b5b350efee5b0b31cf7c09788201"
,
        "Created": "2017-07-24T19:12:59.522673801Z",
        "Path": "java",
        "Args": [
            "Server"
        ],
        "State": {
            "Status": "running",
            "Running": true,
            "Paused": false,
            "Restarting": false,
            "OOMKilled": false,
            "Dead": false,
            "Pid": 5287,
            "ExitCode": 0,
            "Error": "",
            "StartedAt": "2017-07-24T19:13:01.888881799Z",
            "FinishedAt": "0001-01-01T00:00:00Z"
        },
        "Image": "sha256:74973b2125e9aa6def1307b907ea550efdfab10ef212291c07b53bc
```

Build Docker Container for Client Image:

Now, build the Dockerfile of the client using the following command:

- **$ sudo docker build –f clientimage –t client .**

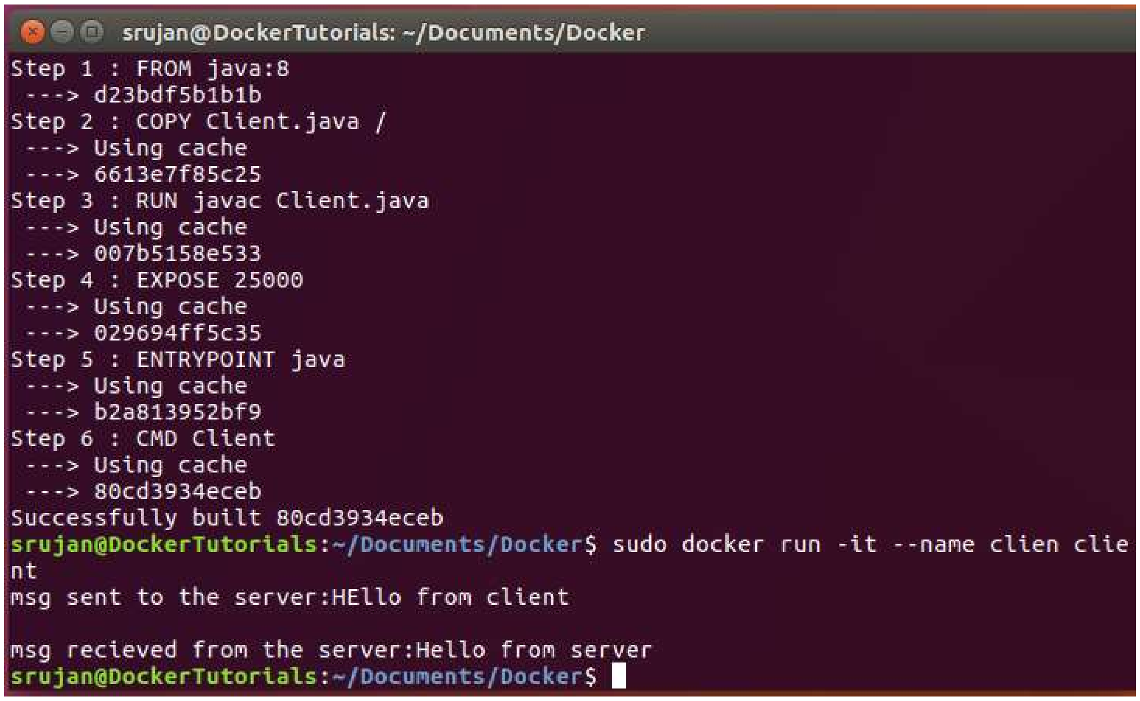


```
srujan@DockerTutorials: ~/Documents/Docker
srujan@DockerTutorials:~/Documents$ cd Docker
srujan@DockerTutorials:~/Documents/Docker$ sudo docker build -f clientimage -t c
lient .
[sudo] password for srujan:
Sending build context to Docker daemon 6.144 kB
Step 1 : FROM java:8
 ---> d23bdf5b1b1b
Step 2 : COPY Client.java /
 ---> Using cache
 ---> 6613e7f85c25
Step 3 : RUN javac Client.java
 ---> Using cache
 ---> 007b5158e533
Step 4 : EXPOSE 25000
 ---> Using cache
 ---> 029694ff5c35
Step 5 : ENTRYPOINT java
 ---> Using cache
 ---> b2a813952bf9
Step 6 : CMD Client
 ---> Using cache
 ---> 80cd3934eceb
Successfully built 80cd3934eceb
srujan@DockerTutorials:~/Documents/Docker$ 
```

Running Container for Client:

Now, run the images for client using the following command:

- **$ sudo docker run –it –name clien client**

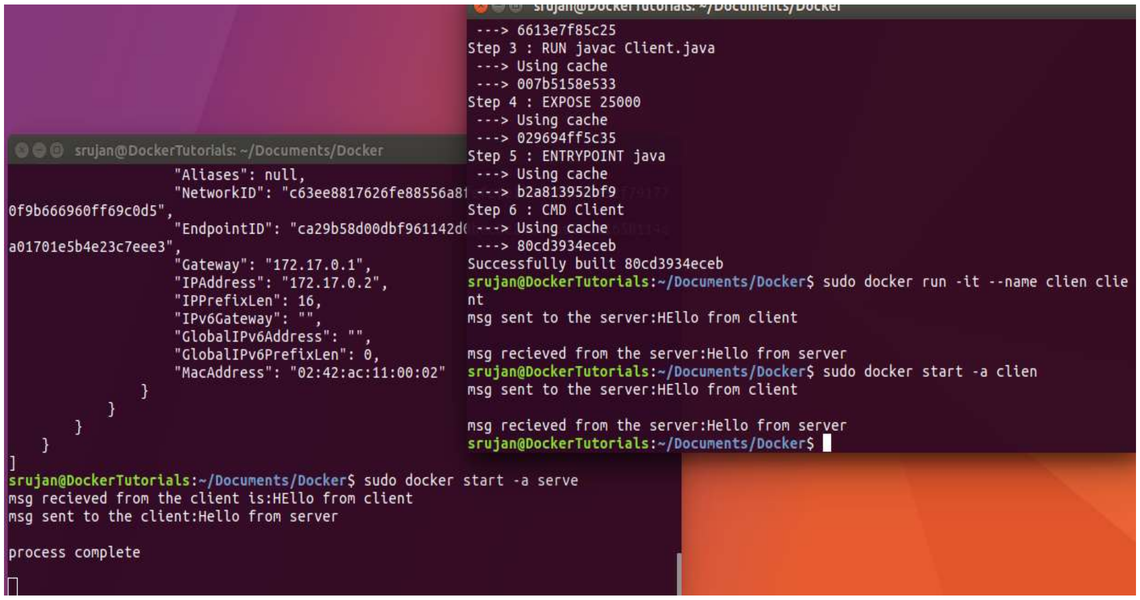


```
srujan@DockerTutorials: ~/Documents/Docker
Step 1 : FROM java:8
 ---> d23bdf5b1b1b
Step 2 : COPY Client.java /
 ---> Using cache
 ---> 6613e7f85c25
Step 3 : RUN javac Client.java
 ---> Using cache
 ---> 007b5158e533
Step 4 : EXPOSE 25000
 ---> Using cache
 ---> 029694ff5c35
Step 5 : ENTRYPOINT java
 ---> Using cache
 ---> b2a813952bf9
Step 6 : CMD Client
 ---> Using cache
 ---> 80cd3934eceb
Successfully built 80cd3934eceb
srujan@DockerTutorials:~/Documents/Docker$ sudo docker run -it --name clien clie
nt
msg sent to the server:HEllo from client

msg recieved from the server:Hello from server
srujan@DockerTutorials:~/Documents/Docker$ 
```

Now, run the following commands for both the Client and Server

- **$ sudo docker start –a serve**
- **$ sudo docker start –a clien**



```
srujan@DockerTutorials: ~/Documents/Docker
                        "Aliases": null,
                        "NetworkID": "c63ee8817626fe88556a81
0f9b666960ff69c0d5",
                        "EndpointID": "ca29b58d00dbf961142d(
a01701e5b4e23c7eee3",
                        "Gateway": "172.17.0.1",
                        "IPAddress": "172.17.0.2",
                        "IPPrefixLen": 16,
                        "IPv6Gateway": "",
                        "GlobalIPv6Address": "",
                        "GlobalIPv6PrefixLen": 0,
                        "MacAddress": "02:42:ac:11:00:02"
                    }
                }
            }
        }
]
srujan@DockerTutorials:~/Documents/Docker$ sudo docker start -a serve
msg recieved from the client is:HEllo from client
msg sent to the client:Hello from server

process complete

```

```
srujan@DockerTutorials: ~/Documents/Docker
 ---> 6613e7f85c25
Step 3 : RUN javac Client.java
 ---> Using cache
 ---> 007b5158e533
Step 4 : EXPOSE 25000
 ---> Using cache
 ---> 029694ff5c35
Step 5 : ENTRYPOINT java
 ---> Using cache
 ---> b2a813952bf9
Step 6 : CMD Client
 ---> Using cache
 ---> 80cd3934eceb
Successfully built 80cd3934eceb
srujan@DockerTutorials:~/Documents/Docker$ sudo docker run -it --name clien clie
nt
msg sent to the server:HEllo from client

msg recieved from the server:Hello from server
srujan@DockerTutorials:~/Documents/Docker$ sudo docker start -a clien
msg sent to the server:HEllo from client

msg recieved from the server:Hello from server
srujan@DockerTutorials:~/Documents/Docker$ 
```