Docker is an open-source platform that allows developers to create, deploy, and run applications in containers. Containers are lightweight, standalone, and executable packages that contain everything needed to run an application, including code, libraries, and system tools.

There are several reasons why Docker is useful for software testing:

1. Consistent Environments: Docker provides a way to create and run applications in a consistent environment, which means that you can test your application in the same environment as the one it will be deployed in.
2. Isolation: Docker containers are isolated from each other and from the host system, which means that you can test your application without worrying about interfering with other applications or the host system.
3. Reproducibility: Docker allows you to create reproducible environments, which means that you can easily recreate a specific environment to reproduce a bug or to test a new feature.
4. Scalability: Docker makes it easy to scale your testing infrastructure up or down, depending on your needs. You can easily spin up new containers to test different parts of your application, or to test your application on different platforms.
5. Cost savings: Docker allows you to test your application on a smaller infrastructure footprint than traditional virtualization, which can save costs on hardware, maintenance, and energy.

Overall, Docker makes it easy to test your application in a controlled, consistent, and reproducible environment, which can help you find and fix bugs faster and ensure that your application works as intended in production.

1. docker --**version** -> To check Docker Version.

2. docker **pull** <Image-Name> -> To download Docker Image from Docker Hub.Pull an image or a repository from a registry

3. docker **ps** -> To check how many Containers are Up and Running.

4. docker **ps –a** -> To check how many Containers are Available, Up and Running.

5. docker **run -it -d** <Image-Name> -> To Create Docker Container from Docker Image.

6. docker **start** <Container-Id> -> To Start Container.

7. docker **restart** <Container-Id> -> To Restart Container.

8. docker **stop** <Container-Id> -> To Stop Container when in Running.

9. docker **rm** <Container-Id> -> To Delete Docker Container.

10. What are the commands for Dockerfile?

**Ans**: Dockerfile is a very important section, check the link [**HERE**](https://draft.blogger.com/blog/post/edit/2695082220643154739/5029479436470806758#) for all commands related to Dockerfile.

10. docker **rmi** <Image-Id> -> To Delete Docker Image.

11. docker **images** -> To check for Available Docker Images in System.

12. docker **exec –it <Container Id> bash** -> To Get into Container and take Control on it.

13. exit -> To Come Out from Container to Docker.

14. docker **kill** <Container Id> -> To Stop Container Forcefully.

15. docker **inspect** <Container Id> -> It will give complete information about Container.

16. docker **image prune –a** -> It will delete images which doesn’t have even a Single Container.

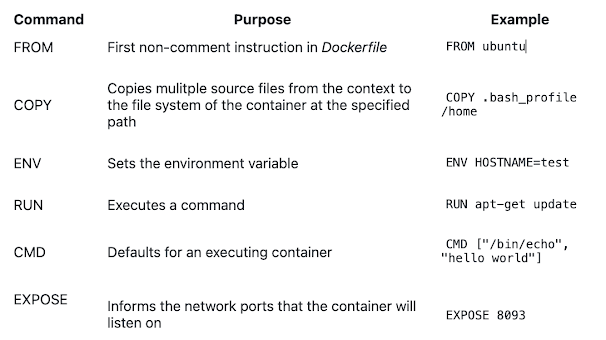
17. docker **run --rm** <Image-Name> -> Create a docker container and auto remove on exit

## **18. How to setup docker with selenium grid for cross browser testing?**

## 

## **Ans**: [**CLICK HERE FOR DETAIL CODE**](https://draft.blogger.com/blog/post/edit/2695082220643154739/5029479436470806758#)

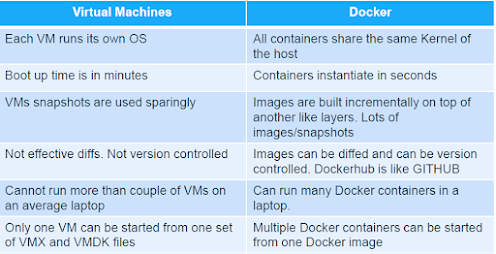
19. docker **system prune -a** : it will delete all the images, containers and networks which are not used to be active anymore.



**What is the difference between VM and Doker?**

Virtual machines have a full OS with its own memory management installed with the associated overhead of virtual device drivers. In a virtual machine, valuable resources are emulated for the guest OS and hypervisor, which makes it possible to run many instances of one or more operating systems in parallel on a single machine (or host). Every guest OS runs as an individual entity from the host system. Hence, we can look at it an independent full-fledge house where we don't share any resources

In the other hand, Docker containers are executed with the Docker engine rather than the hypervisor. Containers are therefore smaller than Virtual Machines and enable faster start up with better performance, less isolation and greater compatibility possible due to sharing of the host’s kernel. Hence, it looks very similar to residental flats system where we share resources of the building.



# **How is Dockerfile different from Docker Compose?**

A Dockerfile is a simple text file that contains the commands a user could call to assemble an image whereas Docker Compose is a tool for defining and running multi-container Docker applications.

Docker Compose define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment. It get an app running in one command by just running docker-compose up.

Docker compose uses the Dockerfile if one add the build command to your project's docker-compose.yml. Your Docker workflow should be to build a suitable Dockerfile for each image you wish to create, then use compose to assemble the images using the build command.

# **What is the maximum number of containers you can run per host?**

This really depends on your environment. The size of your applications as well as the amount of available resources (i.e like CPU) will all affect the number of containers that can be run in your environment. Containers unfortunately are not magical.

They can’t create new CPU from scratch. They do, however, provide a more efficient way of utilizing your resources. The containers themselves are super lightweight (remember, shared OS vs individual OS per container) and only last as long as the process they are running.

# **What is Docker Swarm?**

Docker Swarm is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual Docker host.

Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.

### **Does Docker Swarm do load balancing?**

Yes, Docker Swarm does load balancing. Docker Swarm's load balancer runs on every node and is capable of balancing load requests across multiple containers and hosts.

1. Difference between CMD and ENTRYPOINT

TL;DR CMD will work for most of the cases.

Default entry point for a container is /bin/sh, the default shell.

Running a container as docker container run -it ubuntu uses that command and starts the default shell. The output is shown as:

> docker container run -it ubuntu

root@88976ddee107:/#

ENTRYPOINT allows to override the entry point to some other command, and even customize it. For example, a container can be started as:

> docker container run -it --entrypoint=/bin/cat ubuntu /etc/passwd

root:x:0:0:root:/root:/bin/bash

daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin

bin:x:2:2:bin:/bin:/usr/sbin/nologin

sys:x:3:3:sys:/dev:/usr/sbin/nologin

. . .

This command overrides the entry point to the container to /bin/cat. The argument(s) passed to the CLI are used by the entry point.

### **3. Difference between ADD and COPY**

TL;DR COPY will work for most of the cases.

ADD has all capabilities of COPY and has the following additional features:

1. Allows tar file auto-extraction in the image, for example, ADD app.tar.gz /opt/var/myapp.
2. Allows files to be downloaded from a remote URL. However, the downloaded files will become part of the image. This causes the image size to bloat. So its recommended to use curl or wget to download the archive explicitly, extract, and remove the archive.

**4. Import and export images**

Docker images can be saved using image save command to a .tar file:

docker image save helloworld > helloworld.tar

These tar files can then be imported using load command:

docker image load -i helloworld.tar

**What is Dockerfile? Dockerfile**

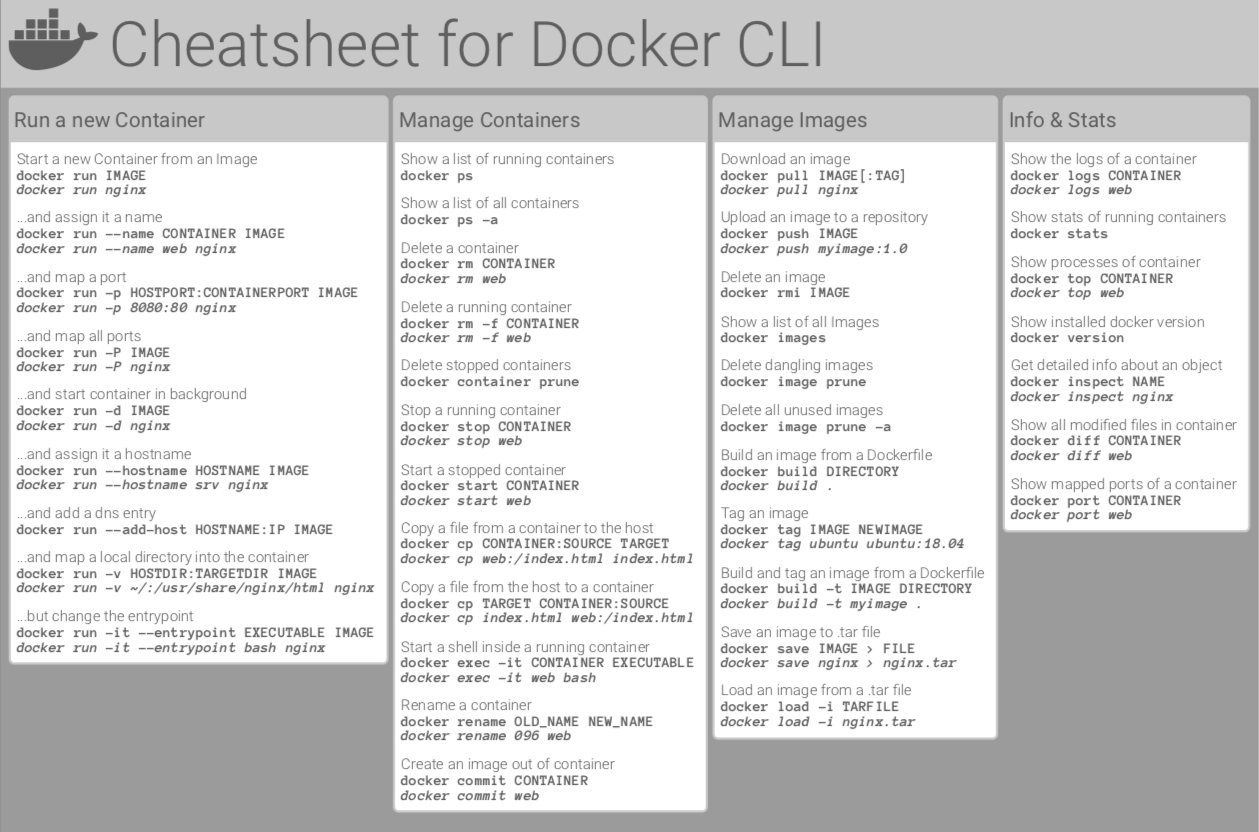
Docker builds images by reading instructions from a *Dockerfile*. A *Dockerfile* is a text document that contains all the commands a user could call on the command line to assemble an image. docker image build command uses this file and executes all the commands in succession to create an image.

build command is also passed a context that is used during image creation. This context can be a path on your local filesystem or a URL to a Git repository.

# NEWLY ADDED

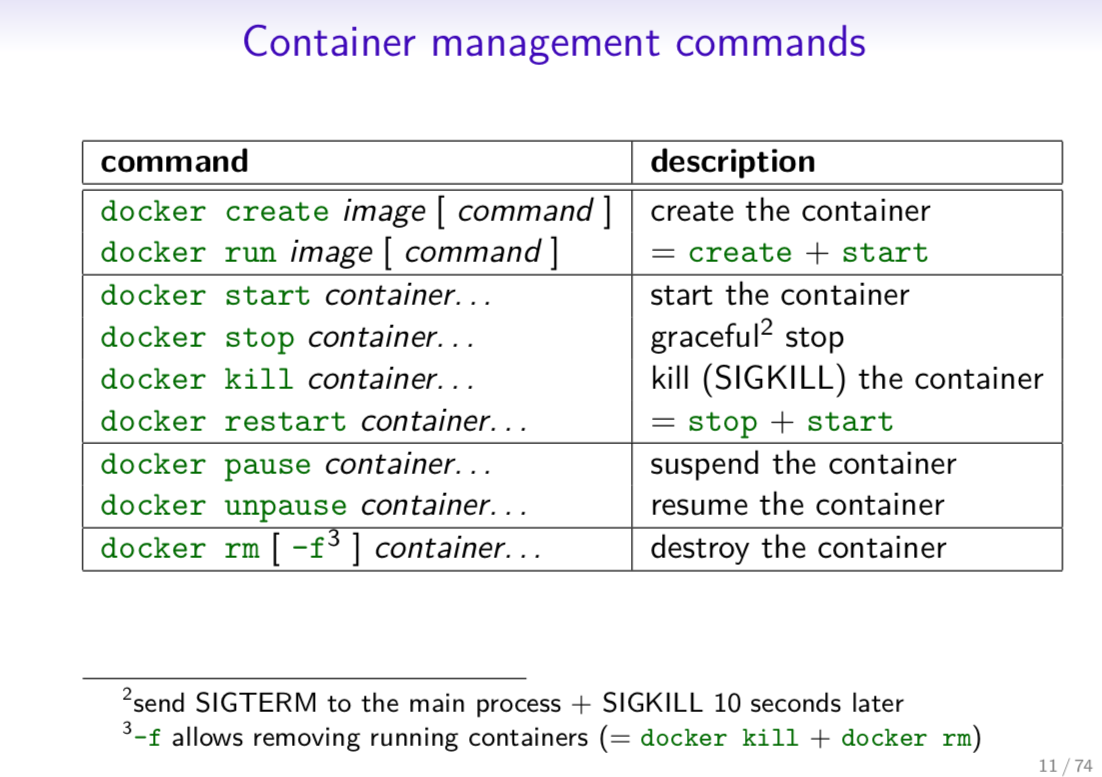
## Basic Docker CLIs

Here’s the list of the basic Docker commands that works on both Docker Desktop as well as Docker Engine:



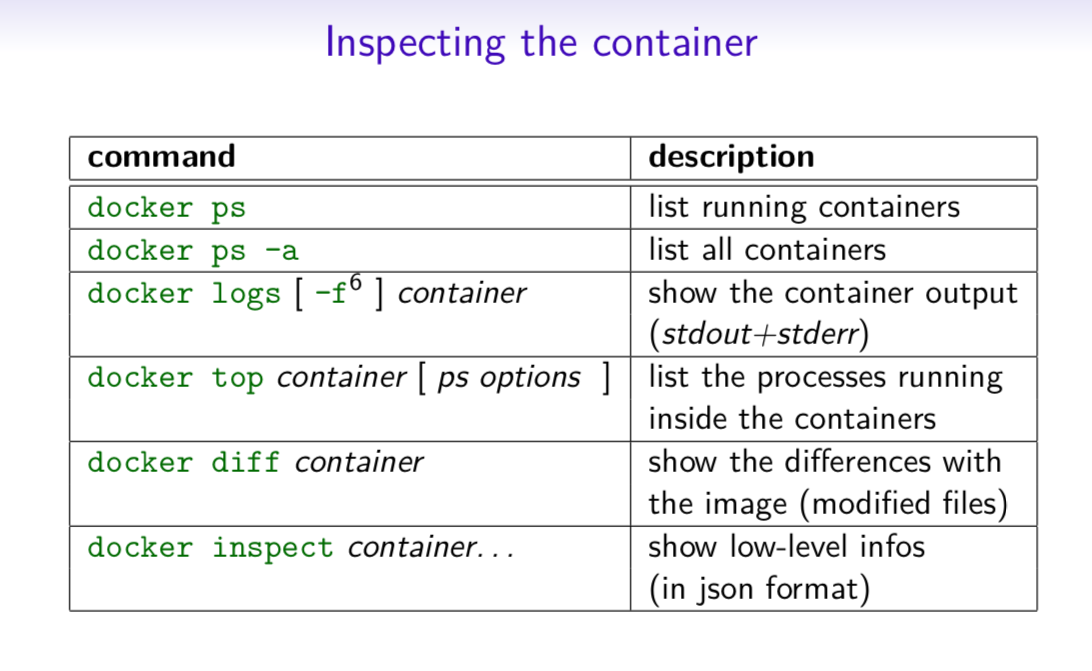
## Container Management CLIs

Here’s the list of the Docker commands that manages Docker images and containers flawlessly:



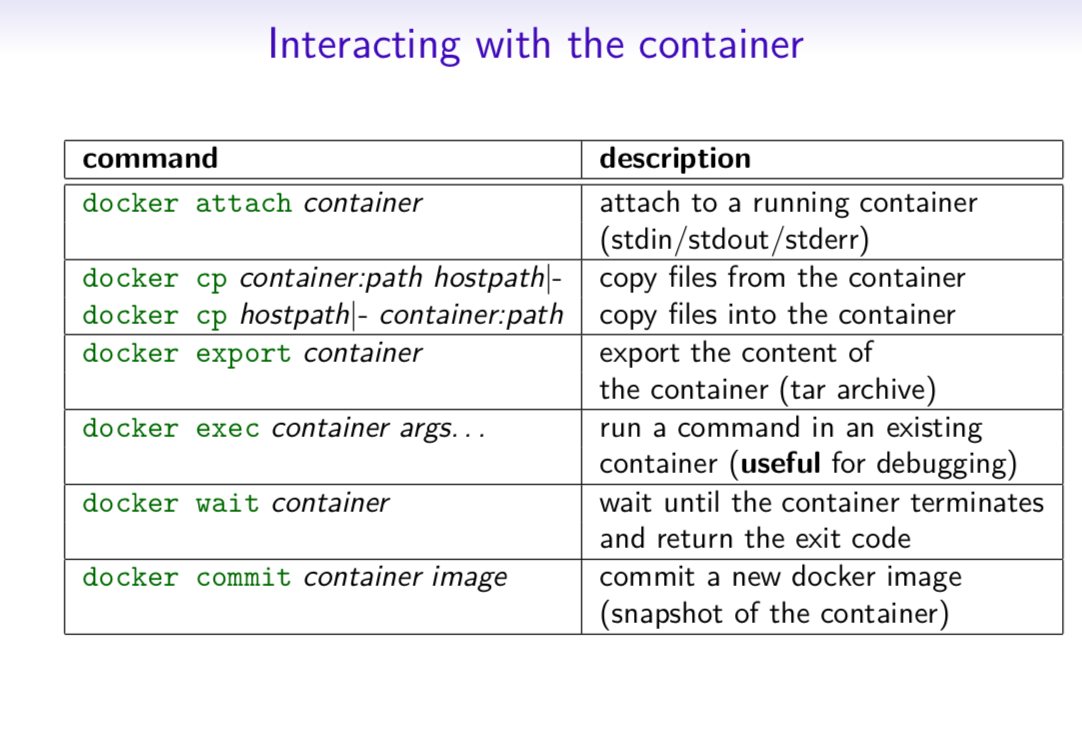
## Inspecting The Container

Here’s the list of the basic Docker commands that helps you inspect the containers seamlessly:



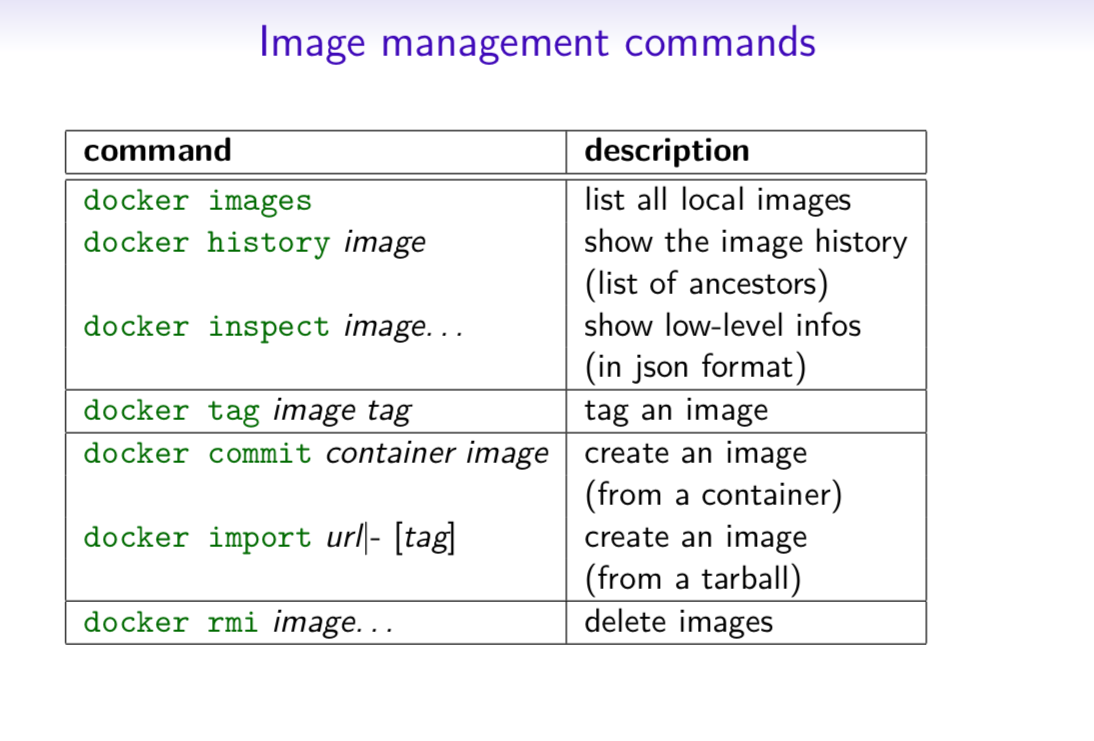
## Interacting with Container

Do you want to know how to access the containers? Check out these fundamental commands:



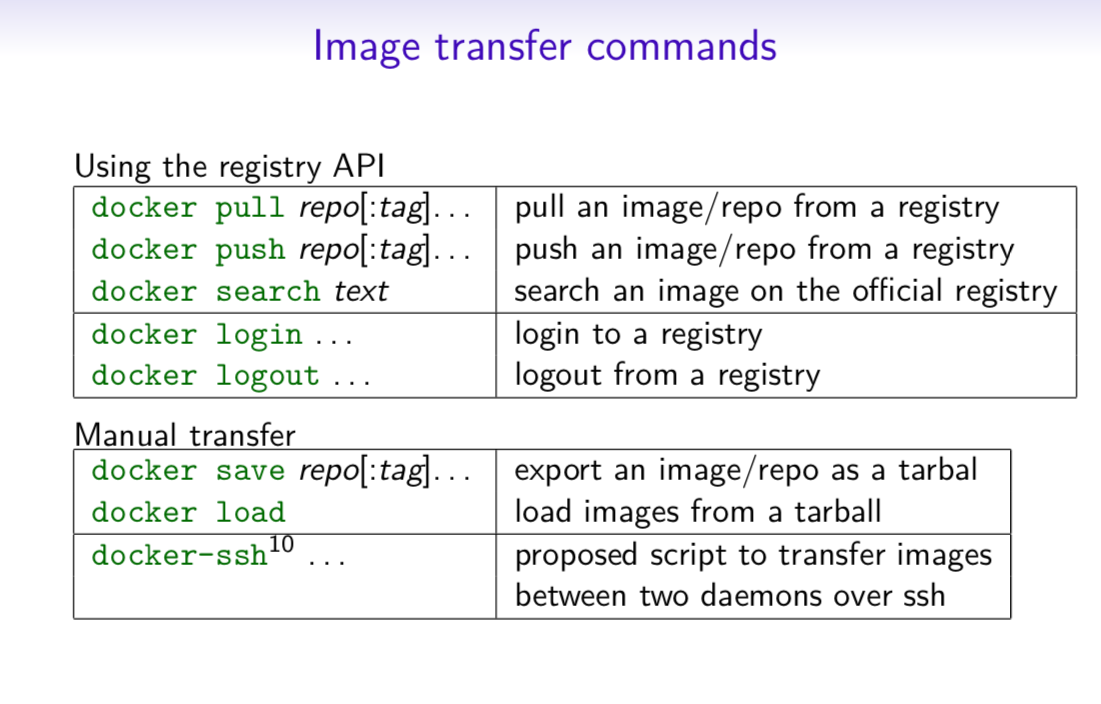
## Image Management Commands

Here’s the list of Docker commands that helps you manage the Docker Images:



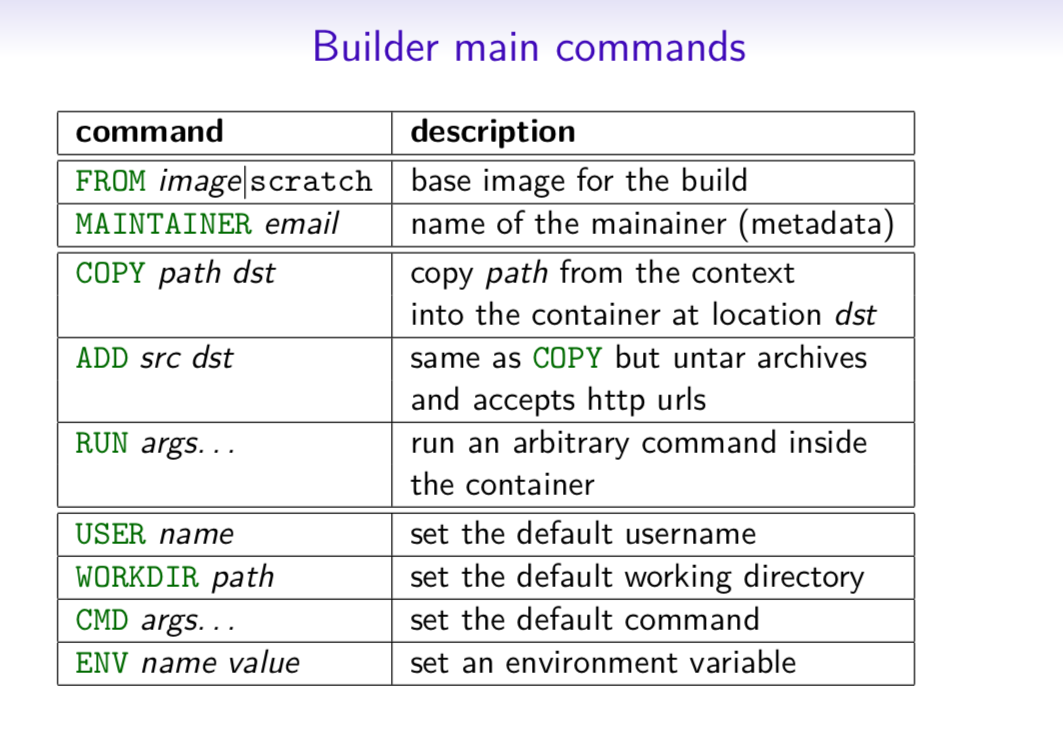
## Image Transfer Commands

Here’s the list of Docker image transfer commands:



## Builder Main Commands

Want to know how to build Docker Image? Do check out the list of Image Build Commands:



## The Docker CLI

## Manage images

### docker build

docker build [options] .

-t "app/container\_name" *# name*

Create an image from a Dockerfile.

### docker run

docker run [options] IMAGE

# see `docker create` for options

Run a command in an image.

## Manage containers

### docker create

docker create [options] IMAGE

-a, --attach *# attach stdout/err*

-i, --interactive *# attach stdin (interactive)*

-t, --tty *# pseudo-tty*

--name NAME *# name your image*

-p, --publish 5000:5000 *# port map*

--expose 5432 *# expose a port to linked containers*

-P, --publish-all *# publish all ports*

--link container:alias *# linking*

-v, --volume `pwd`:/app *# mount (absolute paths needed)*

-e, --env NAME=hello *# env vars*

#### Example

$ docker create --name app\_redis\_1 \

--expose 6379 \

redis:3.0.2

Create a container from an image.

### docker exec

docker exec [options] CONTAINER COMMAND

-d, --detach *# run in background*

-i, --interactive *# stdin*

-t, --tty *# interactive*

#### Example

$ docker exec app\_web\_1 tail logs/development.log

$ docker exec -t -i app\_web\_1 rails c

Run commands in a container.

### docker start

docker start [options] CONTAINER

-a, --attach *# attach stdout/err*

-i, --interactive *# attach stdin*

docker stop [options] CONTAINER

Start/stop a container.

### docker ps

$ docker ps

$ docker ps -a

$ docker kill $ID

Manage containers using ps/kill.

## Images

### docker images

$ docker images

REPOSITORY TAG ID

ubuntu 12.10 b750fe78269d

me/myapp latest 7b2431a8d968

$ docker images -a *# also show intermediate*

Manages images.

### docker rmi

docker rmi b750fe78269d

Deletes images.

## Also see

* [Getting Started](http://www.docker.io/gettingstarted/) *(docker.io)*

# Dockerfile

### Inheritance

**FROM** ruby:2.2.2

### Variables

**ENV** APP\_HOME /myapp

**RUN** mkdir $APP\_HOME

### Initialization

**RUN** bundle install

**WORKDIR** /myapp

**VOLUME** ["/data"]

*# Specification for mount point*

**ADD** file.xyz /file.xyz

**COPY** --chown=user:group host\_file.xyz /path/container\_file.xyz

### Onbuild

**ONBUILD RUN** bundle install

*# when used with another file*

### Commands

**EXPOSE** 5900

**CMD** ["bundle", "exec", "rails", "server"]

### Entrypoint

**ENTRYPOINT** ["executable", "param1", "param2"]

**ENTRYPOINT** command param1 param2

Configures a container that will run as an executable.

**ENTRYPOINT** exec top -b

This will use shell processing to substitute shell variables, and will ignore any CMD or docker run command line arguments.

### Metadata

**LABEL** version="1.0"

**LABEL** "com.example.vendor"="ACME Incorporated"

**LABEL** com.example.label-with-value="foo"

**LABEL** description="This text illustrates \

that label-values can span multiple lines."

## See also

* <https://docs.docker.com/engine/reference/builder/>

# docker-compose

### Basic example

*# docker-compose.yml*

version: '2'

services:

web:

build: .

*# build from Dockerfile*

context: ./Path

dockerfile: Dockerfile

ports:

- "5000:5000"

volumes:

- .:/code

redis:

image: redis

### Commands

docker-compose start

docker-compose stop

docker-compose pause

docker-compose unpause

docker-compose ps

docker-compose up

docker-compose down

## Reference

### Building

web:

*# build from Dockerfile*

build: .

*# build from custom Dockerfile*

build:

context: ./dir

dockerfile: Dockerfile.dev

*# build from image*

image: ubuntu

image: ubuntu:14.04

image: tutum/influxdb

image: example-registry:4000/postgresql

image: a4bc65fd

### Ports

ports:

- "3000"

- "8000:80" *# guest:host*

*# expose ports to linked services (not to host)*

expose: ["3000"]

### Commands

*# command to execute*

command: bundle exec thin -p 3000

command: [bundle, exec, thin, -p, 3000]

*# override the entrypoint*

entrypoint: /app/start.sh

entrypoint: [php, -d, vendor/bin/phpunit]

### Environment variables

*# environment vars*

environment:

RACK\_ENV: development

environment:

- RACK\_ENV=development

*# environment vars from file*

env\_file: .env

env\_file: [.env, .development.env]

### Dependencies

*# makes the `db` service available as the hostname `database`*

*# (implies depends\_on)*

links:

- db:database

- redis

*# make sure `db` is alive before starting*

depends\_on:

- db

### Other options

*# make this service extend another*

extends:

file: common.yml *# optional*

service: webapp

volumes:

- /var/lib/mysql

- ./\_data:/var/lib/mysql

## Advanced features

### Labels

services:

web:

labels:

com.example.description: "Accounting web app"

### DNS servers

services:

web:

dns: 8.8.8.8

dns:

- 8.8.8.8

- 8.8.4.4

### Devices

services:

web:

devices:

- "/dev/ttyUSB0:/dev/ttyUSB0"

### External links

services:

web:

external\_links:

- redis\_1

- project\_db\_1:mysql

### Hosts

services:

web:

extra\_hosts:

- "somehost:192.168.1.100"

### services

To view list of all the services running in swarm

docker service ls

To see all running services

docker stack services stack\_name

to see all services logs

docker service logs stack\_name service\_name

To scale services quickly across qualified node

docker service scale stack\_name\_service\_name=replicas

### clean up

To clean or prune unused (dangling) images

docker image prune

To remove all images which are not in use containers , add - a

docker image prune -a

To prune your entire system

docker system prune

To leave swarm

docker swarm leave

To remove swarm ( deletes all volume data and database info)

docker stack rm stack\_name

To kill all running containers

docker kill $(docker ps -q )

## Docker Security

### Docker Scout

Command line tool for Docker Scout:

docker scout

Analyzes a software artifact for vulnerabilities

docker scout cves [OPTIONS] IMAGE|DIRECTORY|ARCHIVE

Display vulnerabilities from a docker save tarball

docker save redis > redis.tar

Display vulnerabilities from an OCI directory

skopeo copy --override-os linux docker://alpine oci:redis

Export vulnerabilities to a SARIF JSON file

docker scout cves --format sarif --output redis.sarif.json redis

Comparing two images

docker scout compare --to redis:6.0 redis:6-bullseye

Displaying the Quick Overview of an Image

docker scout quickview redis:6.0